Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Vorlesung: Informationstechnik (IT)

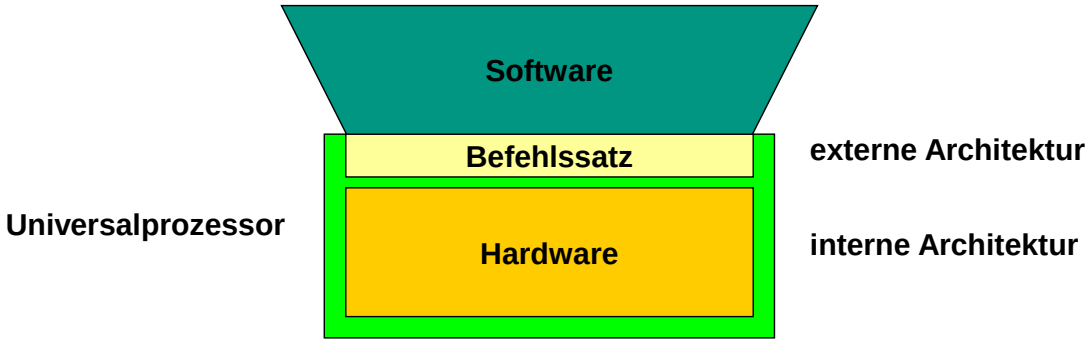
Prof. Dr.-Ing. K.D. Müller-Glaser

Rechnerarchitektur (I)

Wiederholung aus DT: Rechnerarchitektur

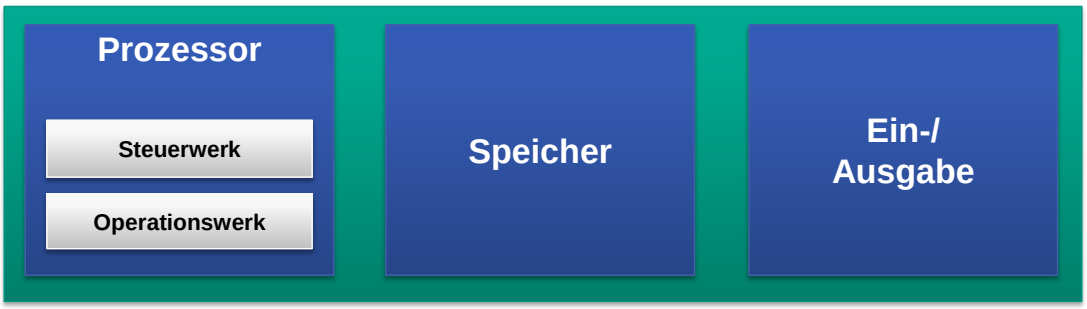
Aufbau aus verschiedenen Komponenten und Organisation der Abläufe

- externe Architektur: Maschinenbefehlssatz
- interne Architektur: der innere Hardwareaufbau eines Rechners
- Universalprozessoren: für alle Anwendungen
  vom einfachen C-Programm bis zum
  Compiler, Betriebssystem, Datenbank, Textverarbeitung



Aufbau eines Rechners

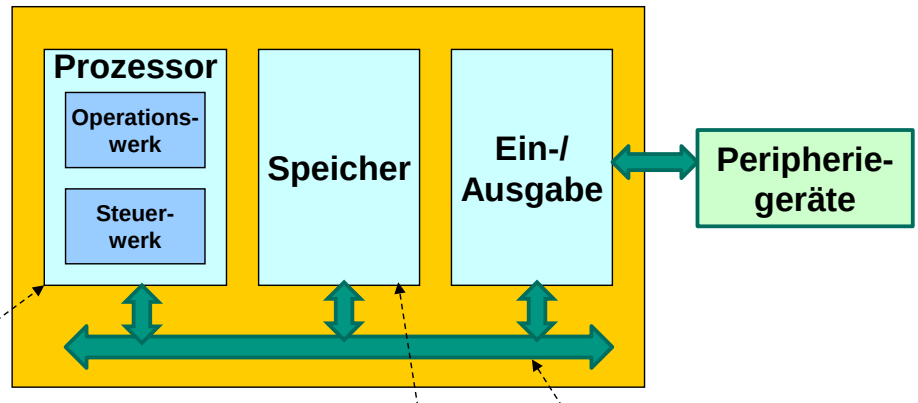
- Bestandteile der internen Architektur:



Central Processing Unit	Festplatte RAM Cache ...	Tastatur, Bildschirm Drucker Maus …

Rechnerarchitekturen

- Interne Architektur (John von Neumann Architektur 1903 – 1957)



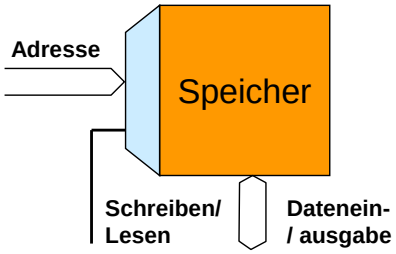
„Rechnerkern“
CPU (central processing unit)
führt Maschinenprogramm aus
Programm ist Folge von Maschinenbefehlen

Speicher
Arbeitsspeicher speichert Maschinenbefehle
und zugehörige Daten

Systembus
(bidirektional)
verbindet vorhandene
Komponenten,
überträgt Daten, Befehle,
Steuersignale,
Busbelegung nach Vorrang
der Geräte

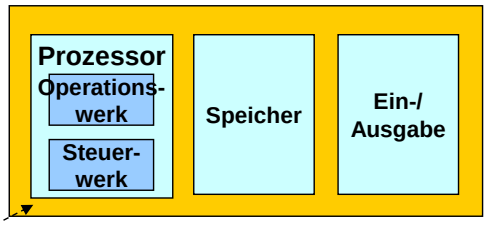
Speicherwerk

- Register im Rechnerkern zur Speicherung von Operanden, Bedingungsvektor und Ergebnis
- wahlfrei adressierbarer Schreib/Lese-Speicher (RAM, RWM), nicht für jede Operation muss Ein-/ Ausgabewerk benutzt werden
- Die Breite des Adressvektors richtet sich nach der Zahl der verfügbaren Speicherworte: $\lceil \log_2 (\text{Anzahl Speicherworte}) \rceil$



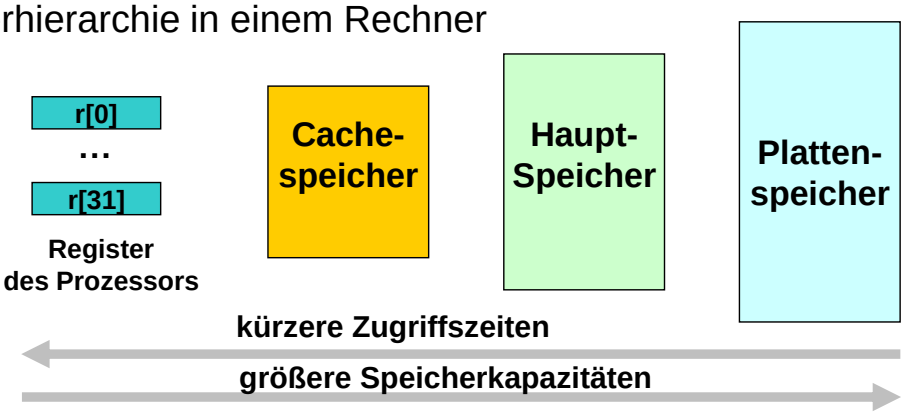
Rechnerarchitekturen

- Interne Architektur



„Rechnerkern“,
CPU (central processing unit)
Operationswerk enthält Registerspeicher

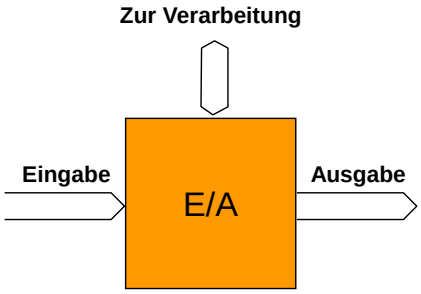
- Speicherhierarchie in einem Rechner



Ein-/Ausgabewerk

- Kommunikation mit dem Benutzer
  Ein-/Ausgabegeräte (Tastatur, Maus, Bildschirm)
  und mit den Peripheriegeräten (z.B. Drucker, Plattenspeicher)

- Eingabe:
  - Übernahme Daten von extern angeschlossenen Geräten
  - Umsetzung in eine geeignete, normierte Darstellung zur weiteren Verarbeitung
- Ausgabe:
  - Datenformatmäßige und elektrische Aufbereitung der Daten
  - Weiterleitung an externe Ausgabegeräte
  - Eventuell Auswahl des Ausgabegerätes

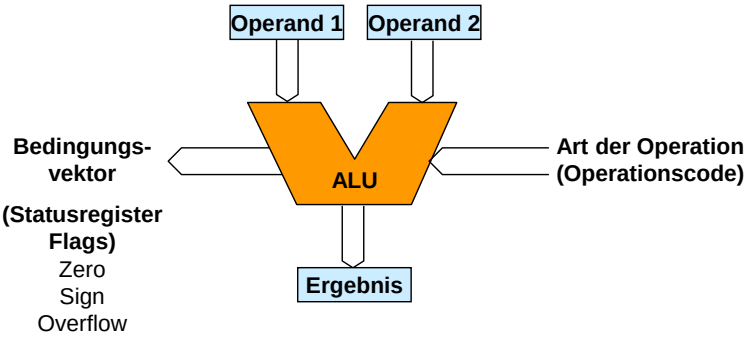


Rechenwerk (Arithmetisch/Logische Einheit ALU)

- Verarbeitung der Daten

- arithmetische und logische Basisfunktionen

- für ein einfaches Rechenwerk:
  - Arithmetische Befehle:
    Addition
    Komplementbildung
  - Logische Befehle:
    Negation
    Konjunktion
    Disjunktion
  - Liefert „Flags“ zurück an Steuerwerk



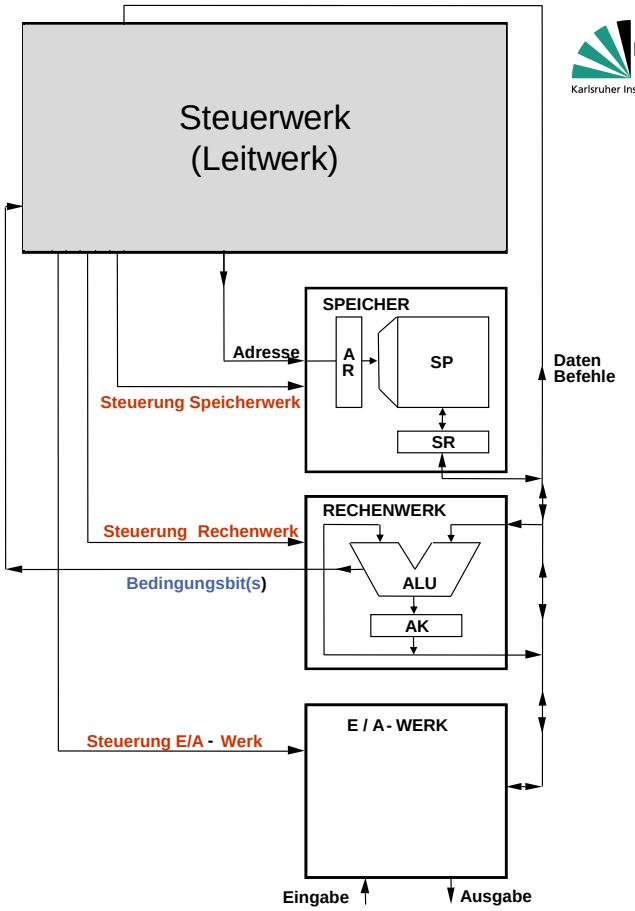
Schema eines einfachen Rechners

Befehle:

1. Transportbefehle
   - Externer Transport (Arbeitsspeicher, E/A-System)
   - Interner Transport (Register-Register)
2. Verarbeitungsbefehle
   - Arithmetische und logische Befehle
   - Schiebebefehle
   - Vergleichsbefehle
3. Sprungbefehle
4. Systembefehle (z.B. Unterbrechungsbearbeitung, Speicherverwaltung)

Abkürzungen:

SP	Speicher
AR	Adressregister
SR	Speicherregister
ALU	Arithmetik / Logik-Einheit
AK	Akkumulator
E/A	Ein-/Ausgabe-Werk



Von-Neumann-Architektur

Arbeitsweise:

1. Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
2. Befehl dekodieren,
   gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
3. Befehl ausführen,
   gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)

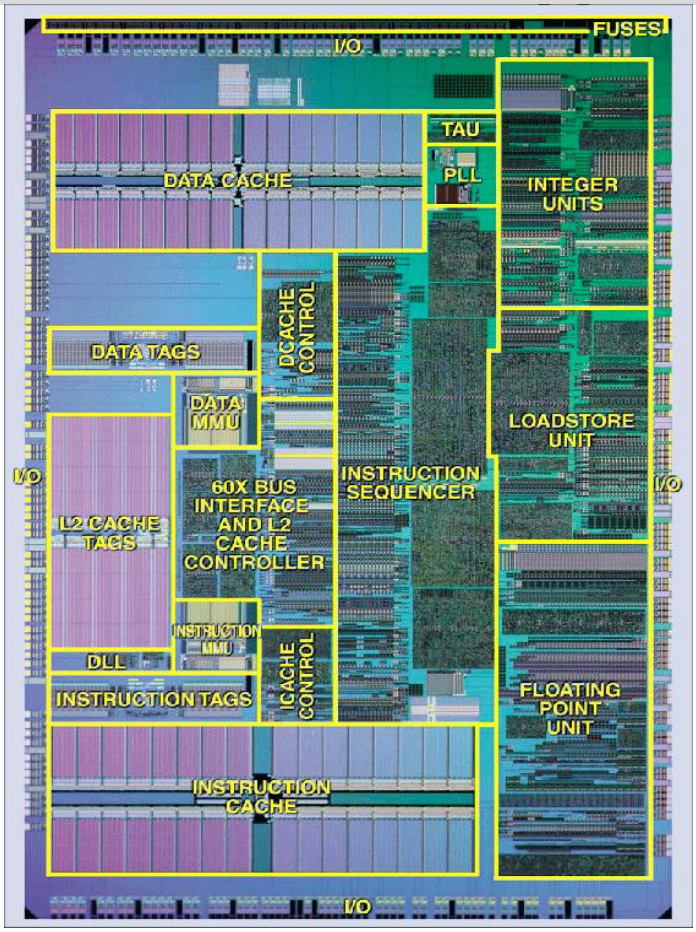
Weiter bei Schritt 1

Rechner führt im Prozessor ständig Befehle aus

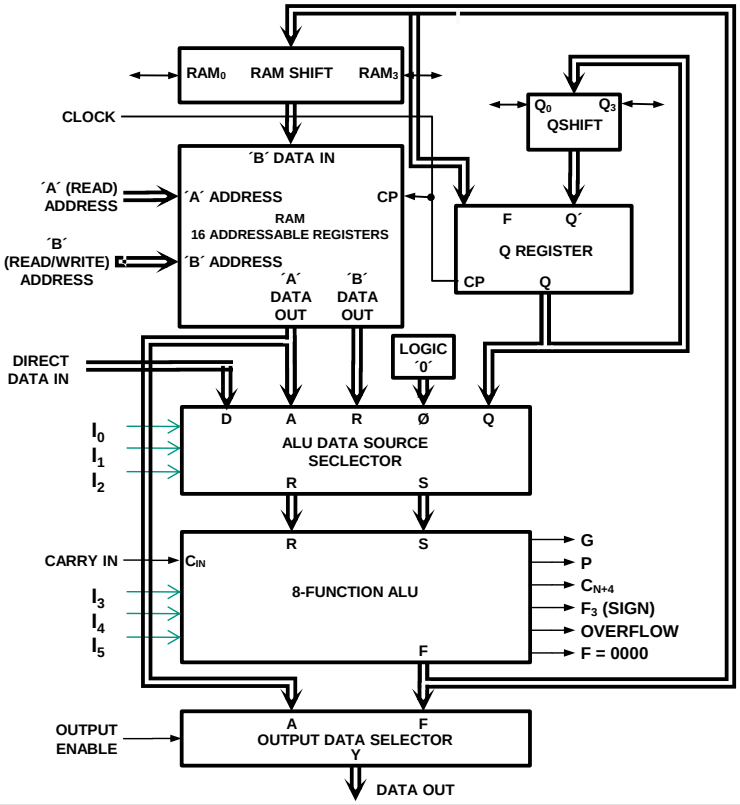
Befehlsausführungszeiten im Prozessor heute um Faktor 10-100 kürzer als Speicherzugriff: „Von-Neumann-Bottleneck“

- ➡ mehr Speicher im Prozessor (Register, Cache-Speicher)
- ➡ separate Speicher und Busse für Daten und Befehle (Harvard-Architektur)

IBM
Power PC 750
Chip-Photo

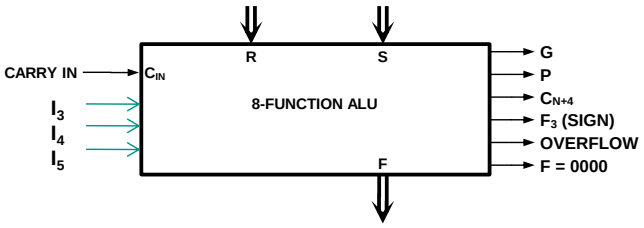


4-Bit Arithmetik-Logik-Einheit
(AMD 2901 Slice)



4-Bit Arithmetik-Logik-Einheit
(AMD 2901 ALU Slice)

Mnemonic	Micro Code I_5	I_4	I_3	Octal Code	ALU Function	Symbol
ADD	L	L	L	0	R Plus S	R + S
SUBR	L	L	H	1	S Minus R	S - R
SUBS	L	H	L	2	R Minus S	R - S
OR	L	H	H	3	R OR S	$R \vee S$
AND	H	L	L	4	R AND S	$R \wedge S$
NOTRS	H	L	H	5	R AND S	$\overline{R} \wedge S$
EXOR	H	H	L	6	R EXOR S	$R \veebar S$
EXNOR	H	H	H	7	R E NOR S	$\overline{R \veebar S}$



Schaltnetze (aus DT)

- **Definition (nach DIN IEC 748 Teil 2):**
  Ein **Schaltnetz** ist eine **Digitalschaltung**, in der es für **jede mögliche Kombination** von **digitalen Signalen** an den **Eingängen** eine – **und nur eine** – **Kombination von digitalen Signalen** an den **Ausgängen** gibt:

$$X \Rightarrow \boxed{F} \Rightarrow Y \qquad Y^t = F(X^t)$$

Dabei ist der hochgestellte **Index t** lediglich ein Hinweis darauf, dass die **Größe Y zeitlich unmittelbar** aus der **Größe X gebildet** wird.

Entwurf von Schaltnetzen (aus DT)

Vorgehensweise:

1. Schritt: Umsetzung der verbalen **Aufgabenstellung** in eine **formale Form**,
z.B. in eine **Funktionstabelle**;
(dazu gehört die Festlegung der Wertezuordnung für die Binärvariablen)

2. Schritt: **Bildung** einer **Normalform**; Bildung einer vollständigen Blocküberdeckung

3. Schritt: **Bildung** einer **Minimalform** (bspw. nach S-Diagramm oder Nelson/Petrick)

4. Schritt: **Umformung** in das gewählte **Basissystem**

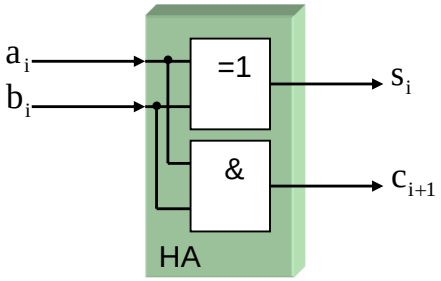
5. Schritt: **Umformen** in den **Strukturausdruck**

6. Schritt: **Umsetzen** in das entsprechende **Schaltnetz**
(unter Umständen sind nicht alle Schritte zum Entwurf notwendig)

Schaltnetzbeispiel: Addiererbaustein (aus DT)

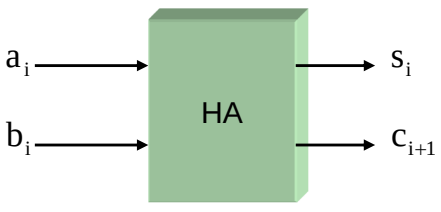
Halbaddierer

- summieren die beiden Eingangsbits a_i und b_i und legen die Summe auf den Ausgang s_i
- zusätzlich wird ein Übertragsbit c_{i+1} erzeugt



$$s_i = a_i \oplus b_i$$
$$s_i = \bar{a}_i \cdot b_i + a_i \cdot \bar{b}_i$$
$$c_{i+1} = a_i \cdot b_i$$

a_i	b_i	s_i	c_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



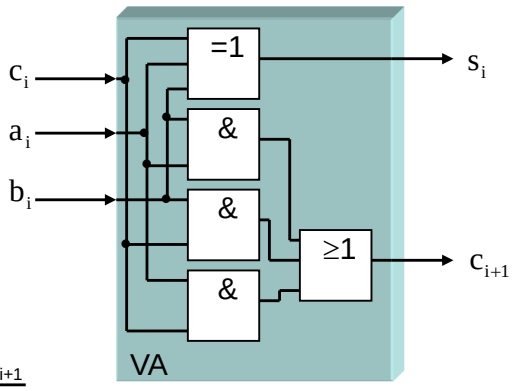
Schaltnetzbeispiel: Addiererbaustein (aus DT)

Volladdierer

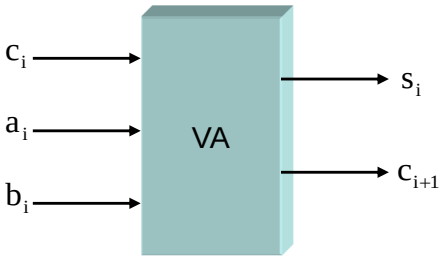
- besitzen zusätzlich einen Übertragseingang und sind somit in der Lage, vorhergehende Stellen in die Berechnung einzubeziehen

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

$$s_i = a_i \oplus b_i \oplus c_i$$



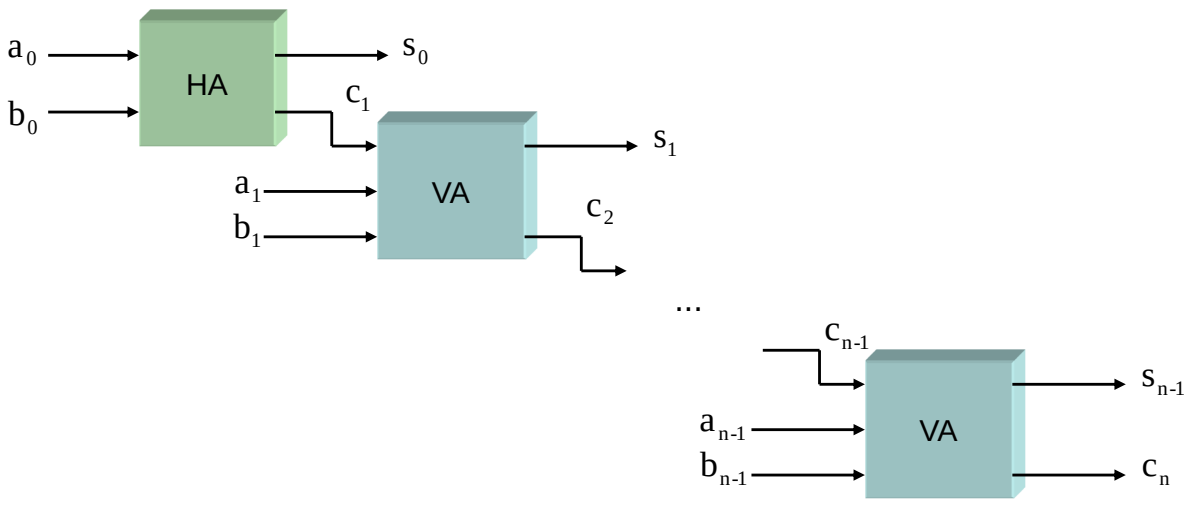
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Schaltnetzbeispiel: Addiererbaustein (aus DT)

n-Bit Ripple-Carry-Addierer

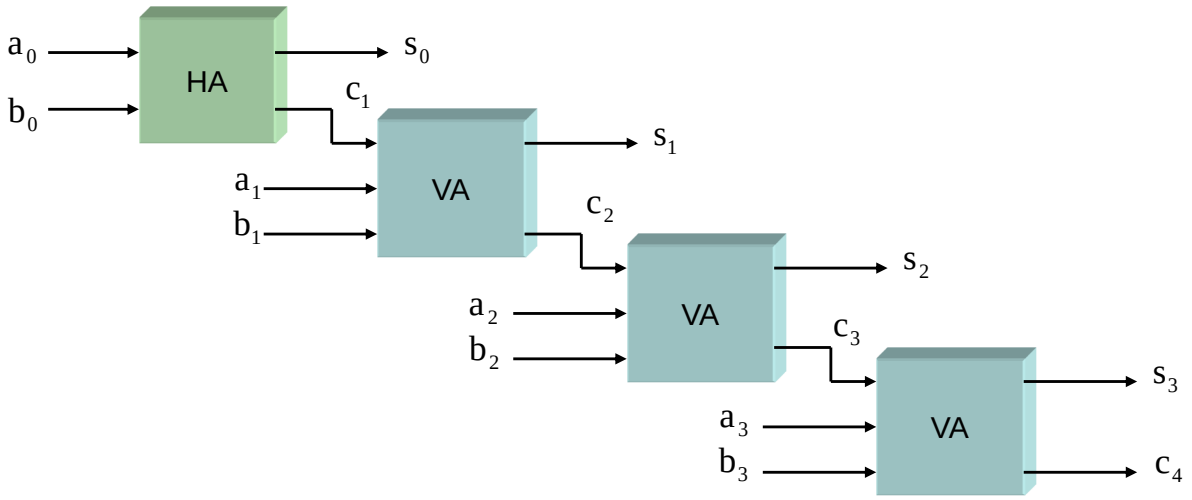
- 1. Baustein ist ein HA, die darauffolgenden sind VA



Schaltnetzbeispiel: Addiererbaustein (aus DT)

Beispiel: **4-Bit Ripple-Carry-Addierer**

- Nachteil: ein sehr langer kritischer Pfad !!!
- die Laufzeit beträgt 2n Gatter für die Berechnung von c_n



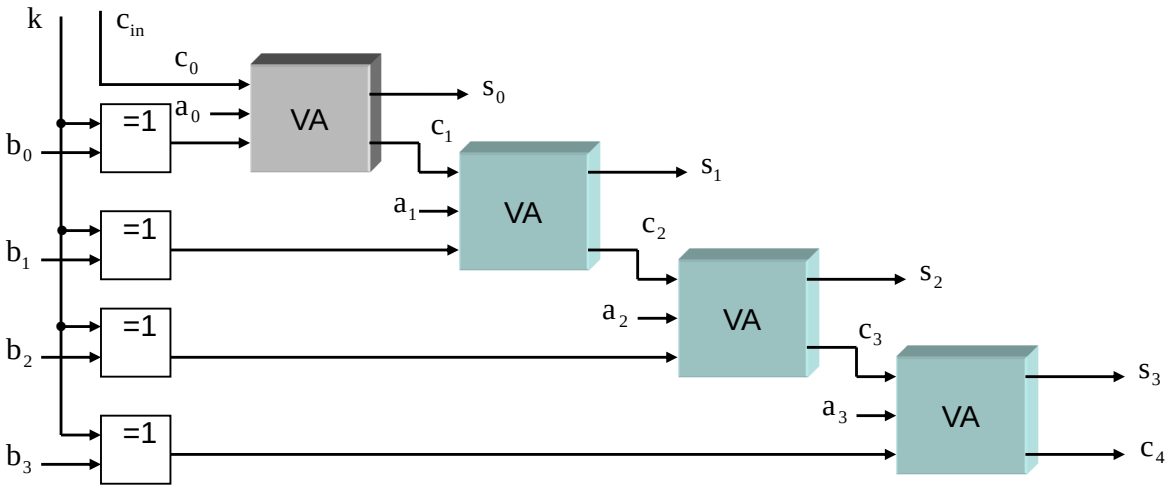
Schaltnetzbeispiel: Subtrahiererbaustein (aus DT)

- 4-Bit Addierer/Subtrahierer

  Addierer mit k=0 und c_0=0

  Subtrahierer mit k=1 und c_0=1 Zweierkomplement

(k: Steuersignal, vom Steuerwerk)



Zwischenübung01 (Teil1): Zweierkomplementdarstellung und Subtraktion

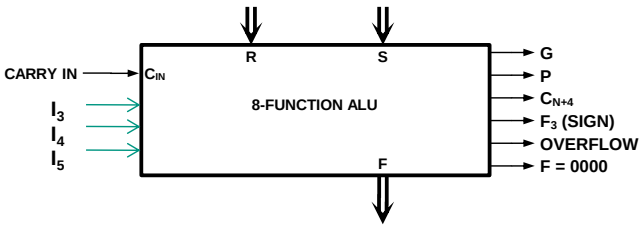
1) Notieren Sie binär alle ganzen Zahlen, die mit einem 4-Bit Slice dargestellt werden können

2) Welcher Bereich für ganze Zahlen ergibt sich für einen n-Bit Slice ohne und mit Vorzeichen?

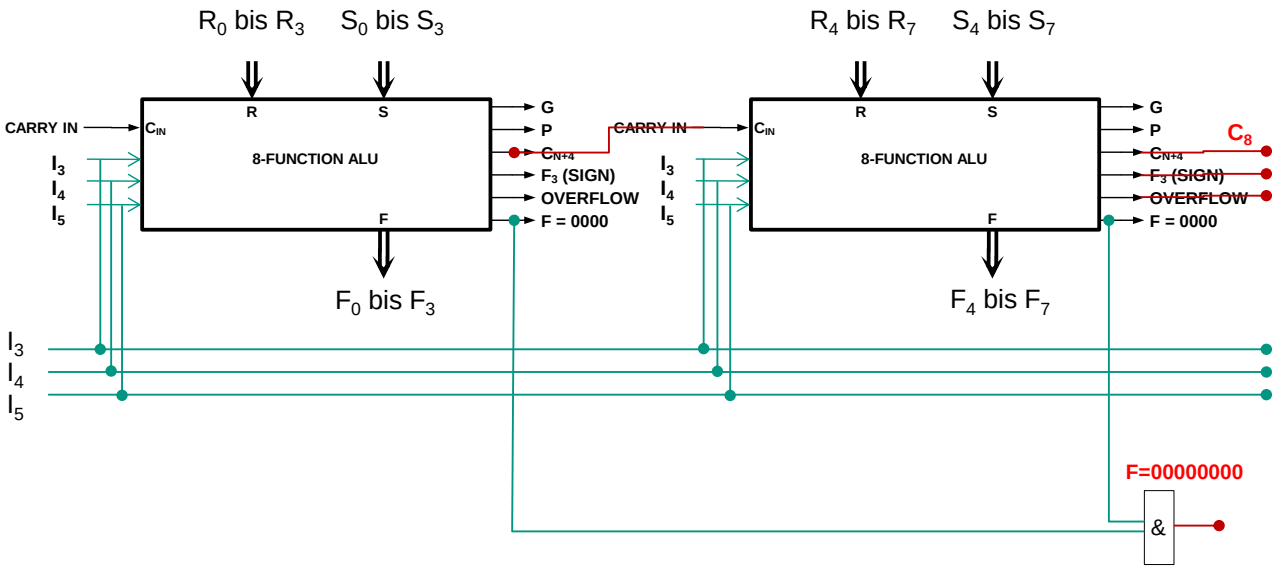
4-Bit Arithmetik-Logik-Einheit
(AMD 2901 ALU Slice)

ALU Function Control

	Micro Code				ALU Function	Symbol
Mnemonic	I_5	I_4	I_3	Octal Code		
ADD	L	L	L	0	R Plus S	R + S
SUBR	L	L	H	1	S Minus R	S - R
SUBS	L	H	L	2	R Minus S	R - S
OR	L	H	H	3	R OR S	$R \vee S$
AND	H	L	L	4	R AND S	$R \wedge S$
NOTRS	H	L	H	5	R AND S	$\overline{R} \wedge S$
EXOR	H	H	L	6	R EXOR S	$R \veebar S$
EXNOR	H	H	H	7	R EXNOR S	$\overline{R \veebar S}$



8-Bit Arithmetik-Logik-Einheit
(2 mal AMD 2901 ALU Slice)



Zwischenübung01 (Teil2): Zweierkomplementdarstellung und Subtraktion

✓ Notieren Sie binär alle ganzen Zahlen, die mit einem 4-Bit Slice dargestellt werden können

✓ Welcher Bereich für ganze Zahlen ergibt sich für einen n-Bit Slice ohne und mit Vorzeichen?

3) Addieren Sie binär mit 4-Bit die Zahlen

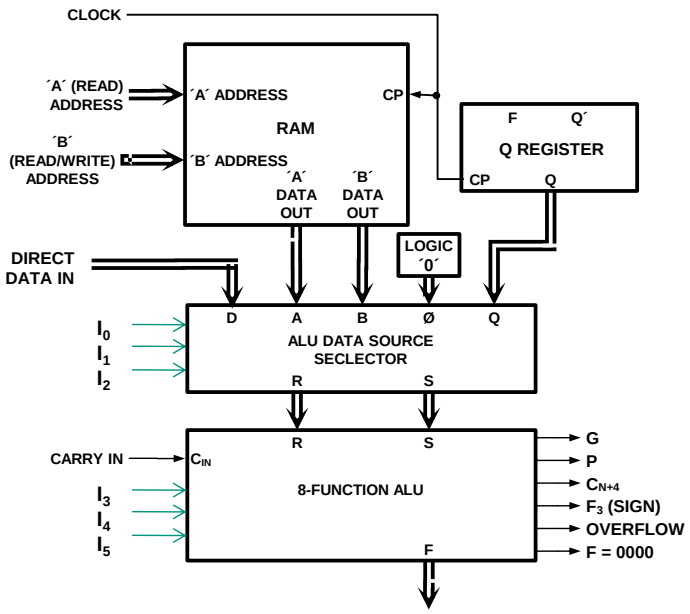
(+3) + (+4)
(+3) + (+6)
(+3) – (+6)
(-3) + (-4)
(-3) + (-6)

4) Wie können Sie erkennen, ob ein Zahlenüberlauf stattfindet?

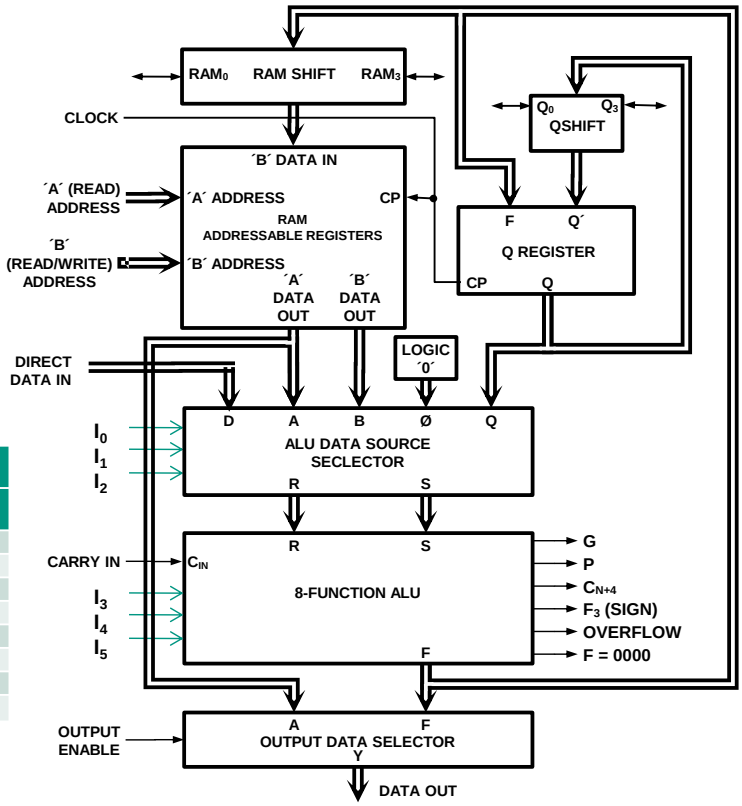
4-Bit Arithmetik-Logik-Einheit
(AMD 2901 ALU Slice)

	Micro Code				ALU Source Operands	
Mnemonic	I_2	I_1	I_0	Octal Code	R	S
AQ	L	L	L	0	A	Q
AB	L	L	H	1	A	B
ZQ	L	H	L	2	0	Q
ZB	L	H	H	3	0	B
ZA	H	L	L	4	0	A
DA	H	L	H	5	D	A
DQ	H	H	L	6	D	Q
DZ	H	H	H	7	D	0

ALU Source Control



4-Bit Arithmetik-Logik-Einheit
(AMD 2901 ALU Slice)



ALU Destination Control

	Micro Code				RAM Function	Q-Reg Function	DATA OUT
Mnemonic	I_2	I_1	I_0	Octal Code	Load	Load	
QREG	L	L	L	0	NONE	F into Q	F
NOP	L	L	H	1	NONE	NONE	F
RAMA	L	H	L	2	F into B	NONE	A
RAMF	L	H	H	3	F into B	NONE	F
RAMQD	H	L	L	4	F/2 into B	Q/2 into Q	F
RAMD	H	L	H	5	F/2 into B	NONE	F
RAMQU	H	H	L	6	2F into B	2Q into Q	F
RAMU	H	H	H	7	2F into B	NONE	F

RAMQD: RAM- und Q-Register werden um 1 Stelle in Richtung LSB (least significant Bit) verschoben, D für downshift
RAMQU: RAM- und Q-Register werden um 1 Stelle in Richtung MSB (most significant Bit) verschoben, U für upshift

Achtung: Logik für Schiebeoperationen (shift logical up, shift logical down, rotate up, rotate down, shift arithmetic up, shift arithmetic down) ist noch hinzu zu fügen:

für logische Schiebeoperationen (logical shift) höchstwertiges Bit (MSB, Vorzeichenstelle) wird mit geschoben, Wahl 0 oder 1 nachschieben, oder rotieren (rotate up) MSB nach LSB oder (rotate down) LSB nach MSB

für arithmetische Schiebeoperationen (arithmetic shift) bleibt MSB erhalten, für upshift 0 nachschieben, für downshift MSB nachschieben
für vorzeichenbehaftete ganze Zahlen arithmetischer Upshift entspricht

ALU Destination Control

	Micro Code				RAM Function		Q-Reg Function		Y Output	RAM - Shifter			Q - Shifter	
Mnemonic	I_8	I_7	I_6	Octal Code	Shift	Load	Shift	Load		RAM_0	RAM_3		Q_0	Q_3
QREG	L	L	L	0	X	NONE	NONE	F to Q	F	X	X		X	X
NOP	L	L	H	1	X	NONE	X	NONE	F	X	X		X	X
RAMA	L	H	L	2	NONE	F into B	X	NONE	A	X	X		X	X
RAMF	L	H	H	3	NONE	F into B	X	NONE	F	X	X		X	X
RAMQD	H	L	L	4	DOWN	F/2 into B	DOWN	Q/2 into Q	F	F_0	IN_3		Q_0	IN_3
RAMD	H	L	H	5	DOWN	F/2 into B	X	NONE	F	F_0	IN_3		Q_0	X
RAMQU	H	H	L	6	UP	2F into B	UP	2Q into Q	F	IN_0	F_3		IN_0	Q_3
RAMU	H	H	H	7	UP	2F into B	X	NONE	F	IN_0	F_3		X	Q_3

X: don't care
B: register addressed by B inputs
UP is towards MSB, DOWN is towards LSB

Zwischenübung02 (Teil 1): Mikroprogrammierung

Wie lautet der Binärcode, um das RAM-Register hexF auf Null zu setzen?

4-Bit Arithmetik-Logik-Einheit

ALU Steuersignale vom Steuerwerk:

8	7	6	5	4	3	2	1	0
DESTINATION CONTROL			ALU FUNCTION			ALU SOURCE		

I_6 I_3 I_0
I_7 I_4 I_1
I_8 I_5 I_2

CARRY IN

DIRECT DATA IN

´A´ (READ) ADDRESS

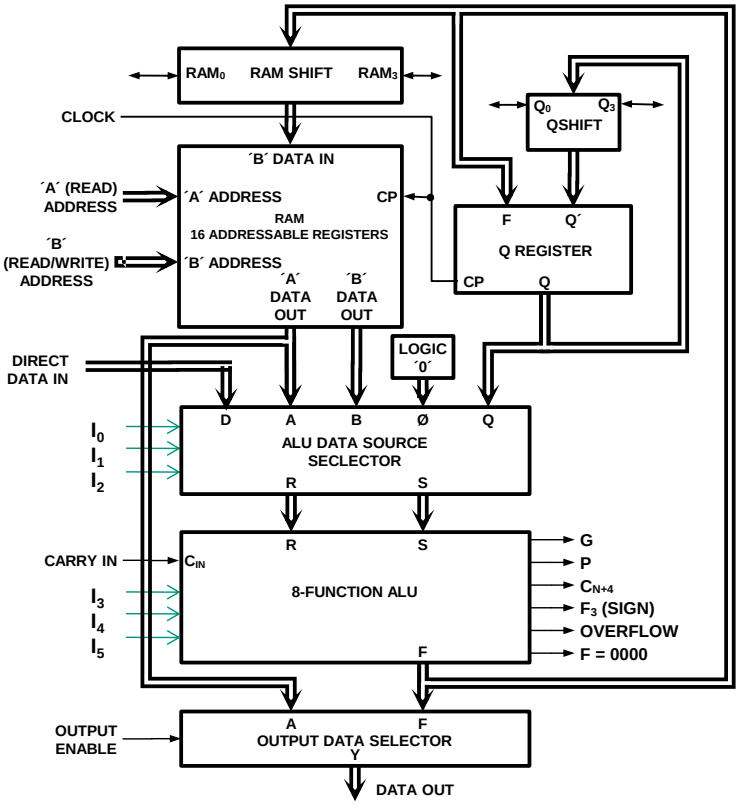
´B´ (READ/WRITE) ADDRESS

OUTPUT ENABLE

SHIFT: RAM0, RAM3, Q0, Q3

ALU Statussignale zum Steuerwerk:

C_{N+4} F_3 (SIGN) OVERFLOW
F = 0000



Zwischenübung02 (Teil 2): Mikroprogrammierung

Schreiben Sie ein Mikroprogramm, das für die 2901 Alu-Slice die Inhalte der RAM Register 3 und 4 addiert und das Ergebnis nach RAM Register 5 speichert.

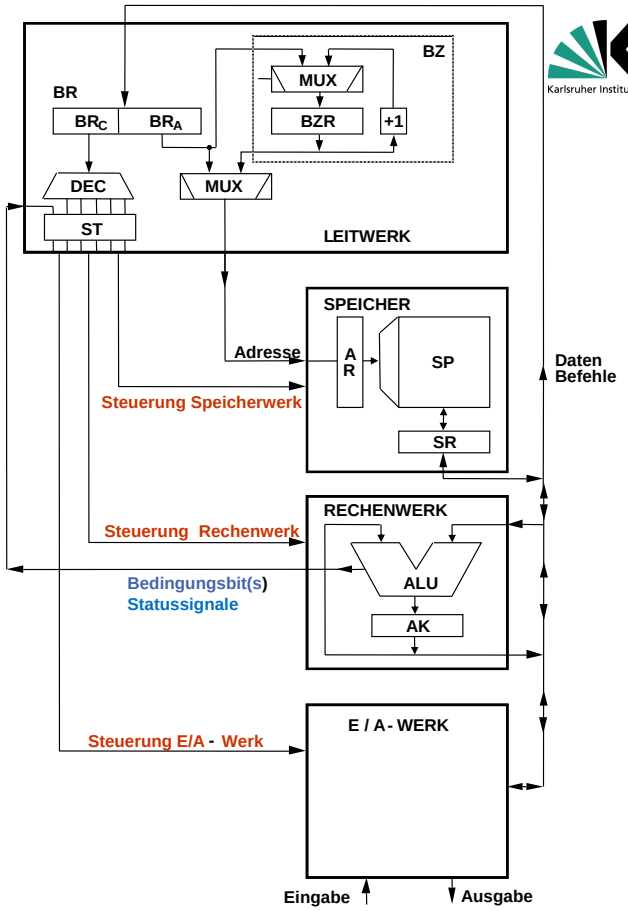
Welche Fehler können auftreten?

Schreiben Sie ein Mikroprogramm, das für die 2901 Alu-Slice die Inhalte der RAM Register 3 und 4 addiert und das korrekte Ergebnis nach Register Q speichert.

Schema eines einfachen Rechners
(von Neumann Architektur)

Abkürzungen:

BR	Befehlsregister
BR_C	Codeteil von BR
BR_A	Adressteil von BR
DEC	Befehlsdecoder
ST	Register für Steuerbits
BZ	Befehlszähler
MUX	Multiplexer
BZR	Befehlszählerregister
AR	Adressregister
SR	Speicherregister
AK	Akkumulator



Steuerwerk (Leitwerk)

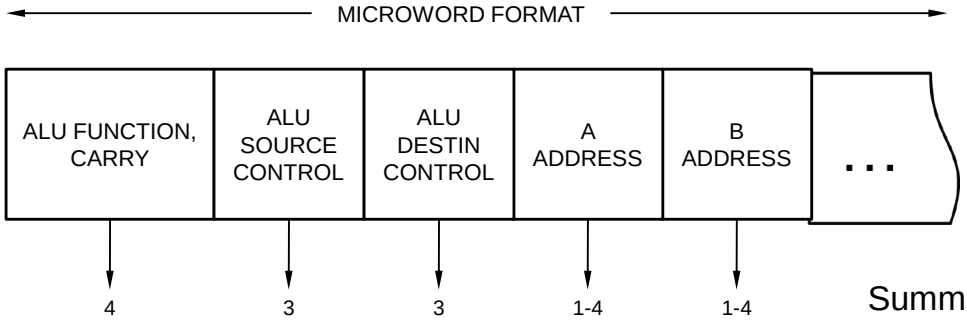
- Steuerung und Koordination des zeitlichen Ablaufs der Komponenten des Prozessors
  - Versorgung mit Daten und Adressen
  - Selektion von Geräten
  - Auswahl von Operanden und Operationen
- Ursprünglich Steuerung durch den Bediener, später Automatisierung durch zeitliche Abfolge von Binärinformationen im Maschinenprogramm (Steuersignale)
- Formalisierung der Anweisung nötig:

  → **Befehl:** (maschinenlesbare) Anweisung, aus der hervorgeht welche Operation mit welchen Operanden durchgeführt werden soll

  Maschinenbefehl startet Mikrobefehlsfolge

  → **Befehlsformat:** Festlegung, wie die für einen Befehl notwendigen Angaben dargestellt werden

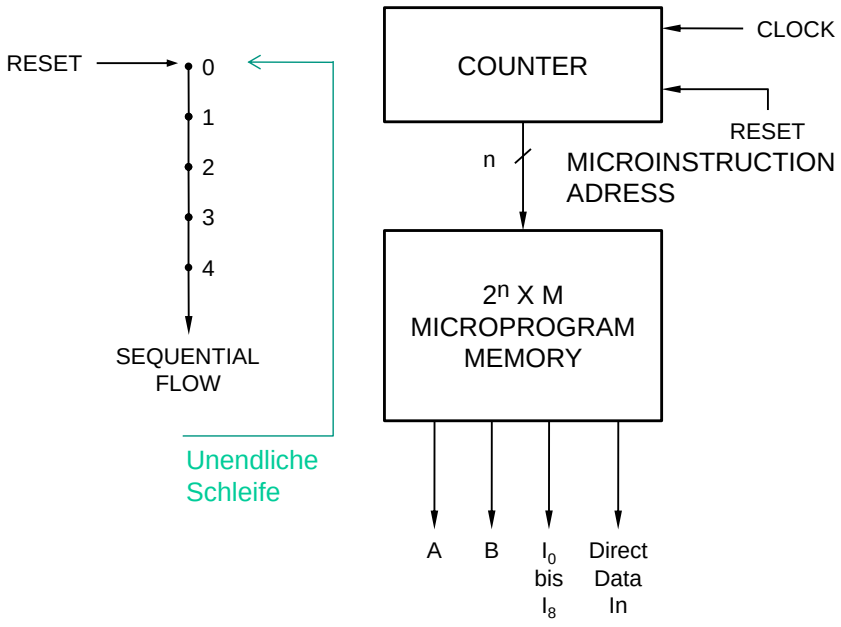
Mikrobefehlsformat



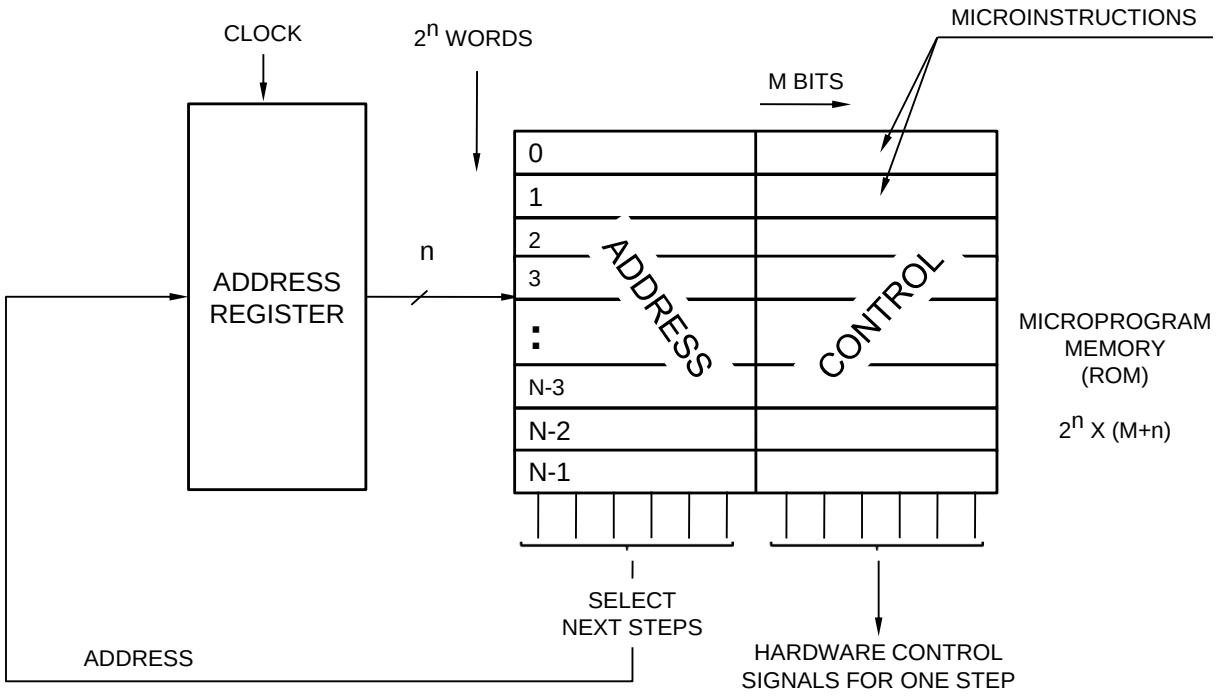
Summe Steuerbits M

CONTROL LINES			ALU FUNCTION	
S_2	S_1	S_0	$C_{IN} = 0$	$C_{IN} = 1$
0	0	0	A + B	A + B + 1
0	0	1	B - A - 1	B - A
0	1	0	A - B - 1	A - B
0	1	1	$A \vee B$	
1	0	0	$A \wedge B$	
1	0	1	$\overline{A} \wedge B$	
1	1	0	$A \veebar B$	
1	1	1	$\overline{A \veebar B}$	

Ablaufsteuerung: Sequentielle Mikrobefehlsfolge



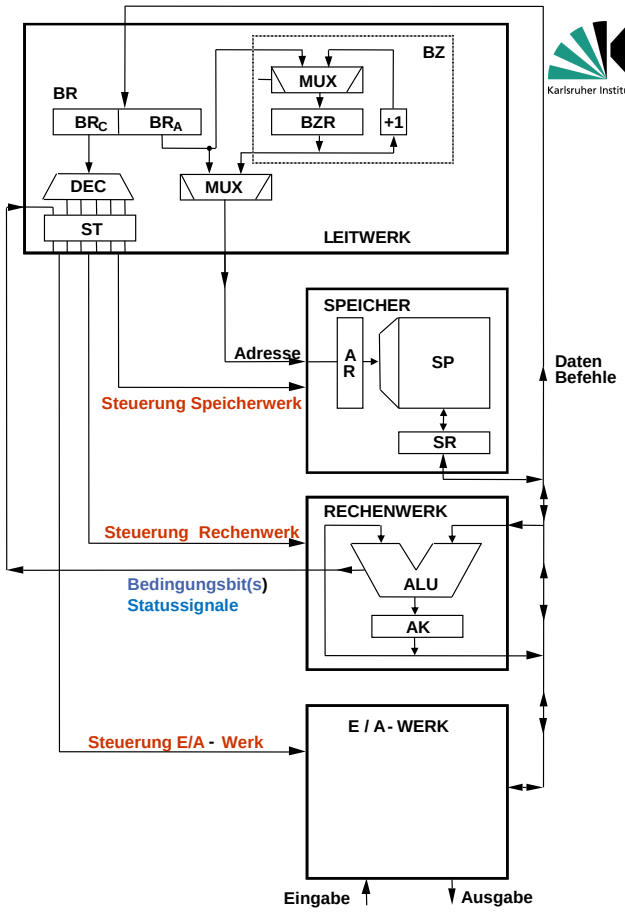
Mikrobefehlsfolge mit wählbarer Adressfolge



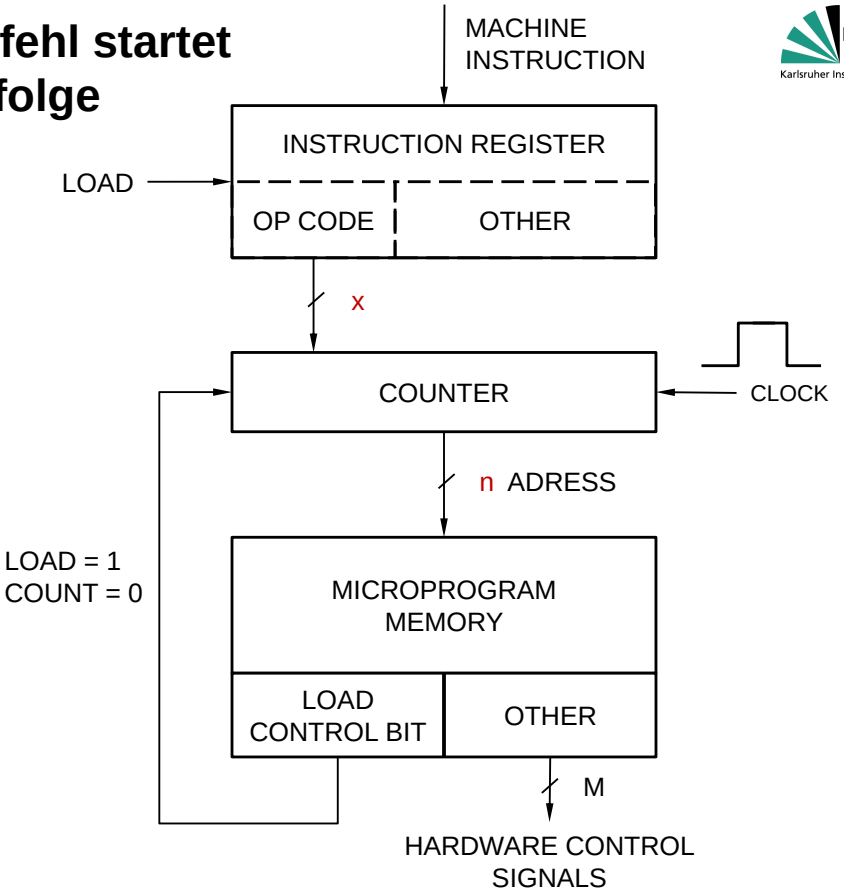
Schema eines einfachen Rechners

Abkürzungen:

BR	Befehlsregister
BR_C	Codeteil von BR
BR_A	Adressteil von BR
DEC	Befehlsdecoder
ST	Register für Steuerbits
BZ	Befehlszähler
MUX	Multiplexer
BZR	Befehlszählerregister
AR	Adressregister
SR	Speicherregister
AK	Akkumulator



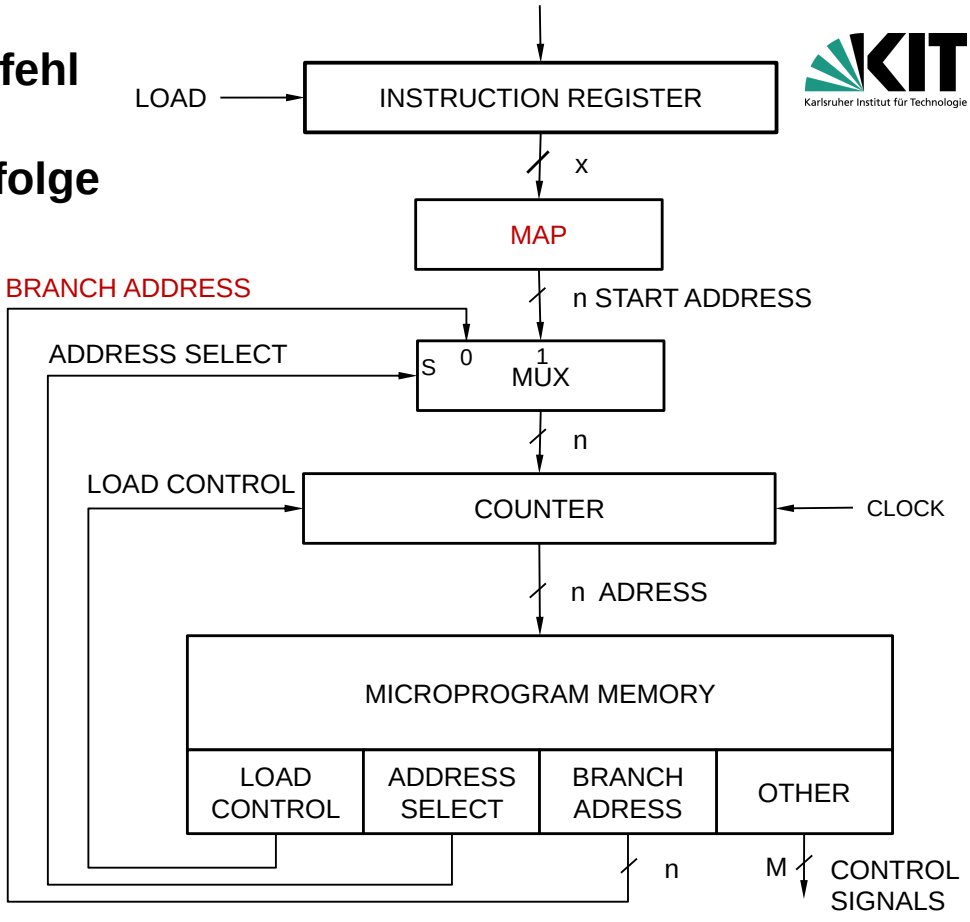
Maschinenbefehl startet Mikrobefehlsfolge



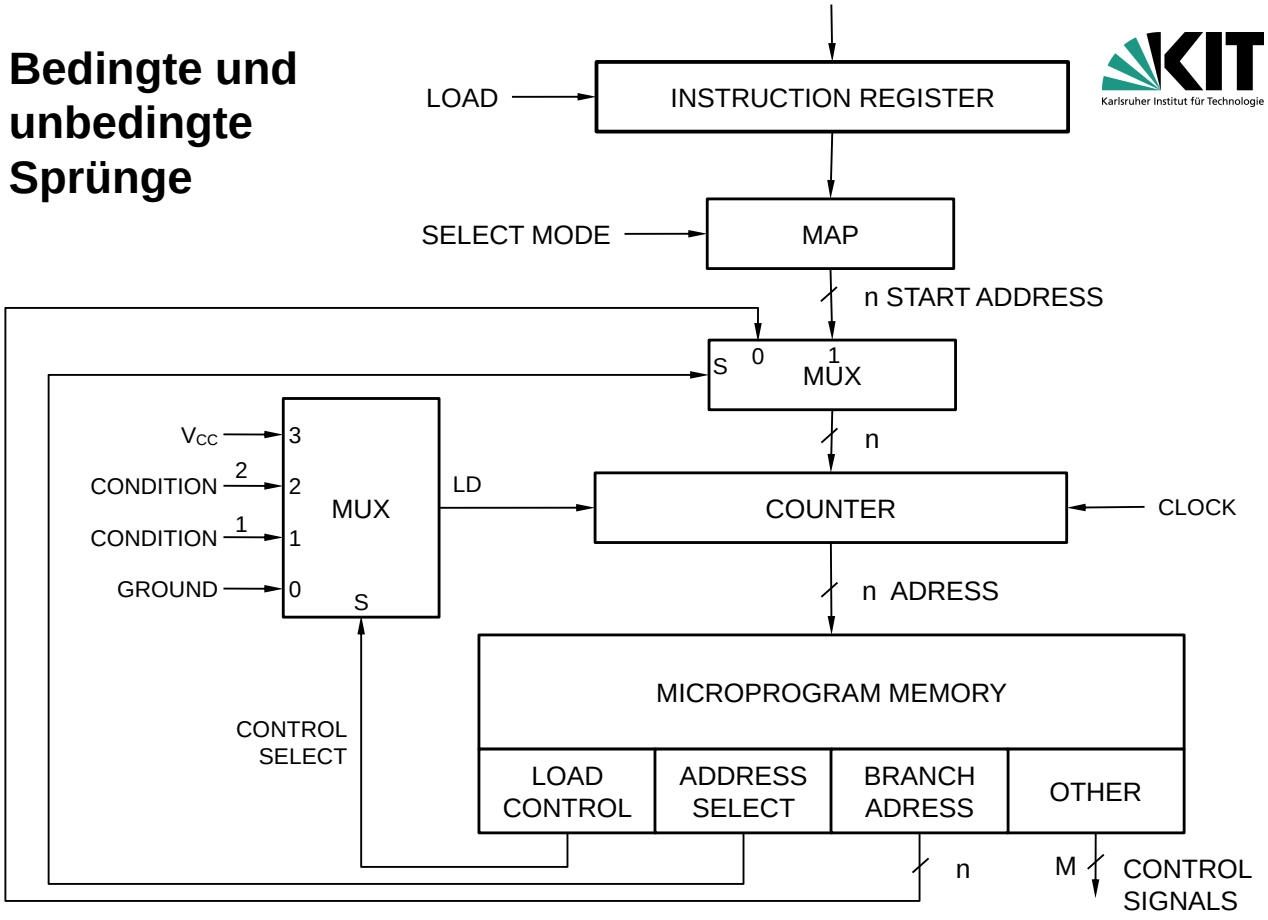
Maschinenbefehl startet Mikrobefehlsfolge

Address-Mapping

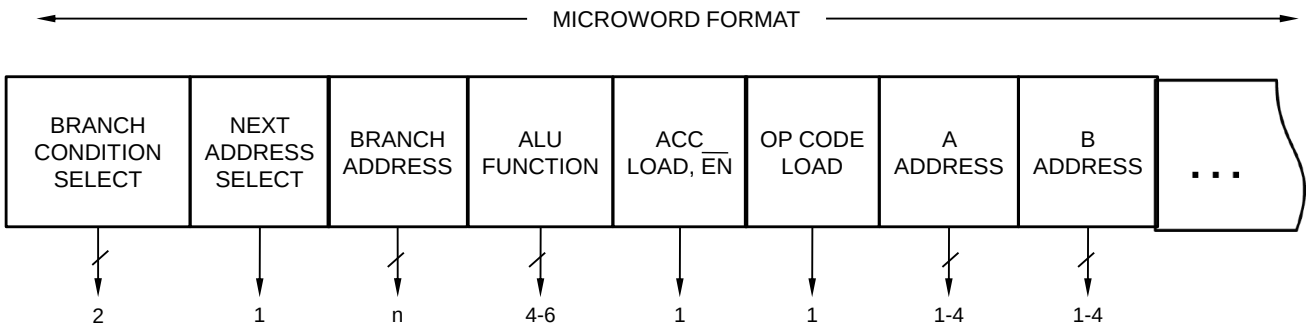
Branch-Address

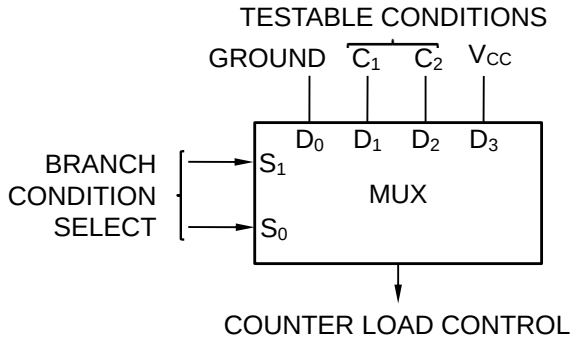


Bedingte und unbedingte Sprünge



Bedingte und unbedingte Sprünge





S_1	S_0	RESULT
0	0	INCREMENT COUNTER
0	1	BRANCH IF C_1 = TRUE
1	0	BRANCH IF C_2 = TRUE
1	1	LOAD COUNTER

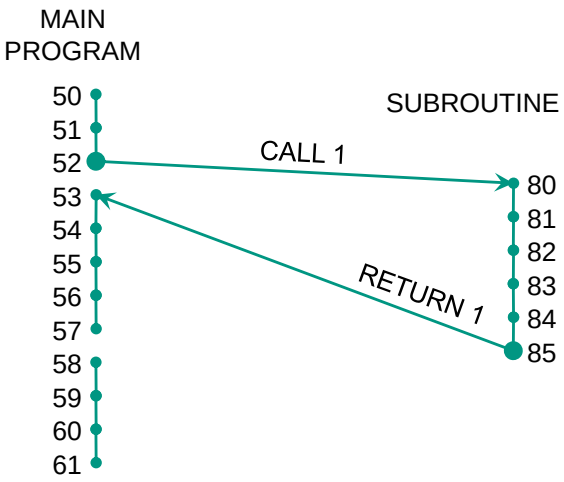
Bedingte und unbedingte Sprünge

BRANCH CONDITION SELECT		BRANCH ADDRESS
1	1	90

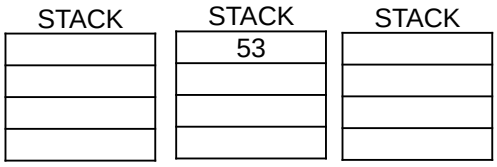
RESULT

LOAD COUNTER 90

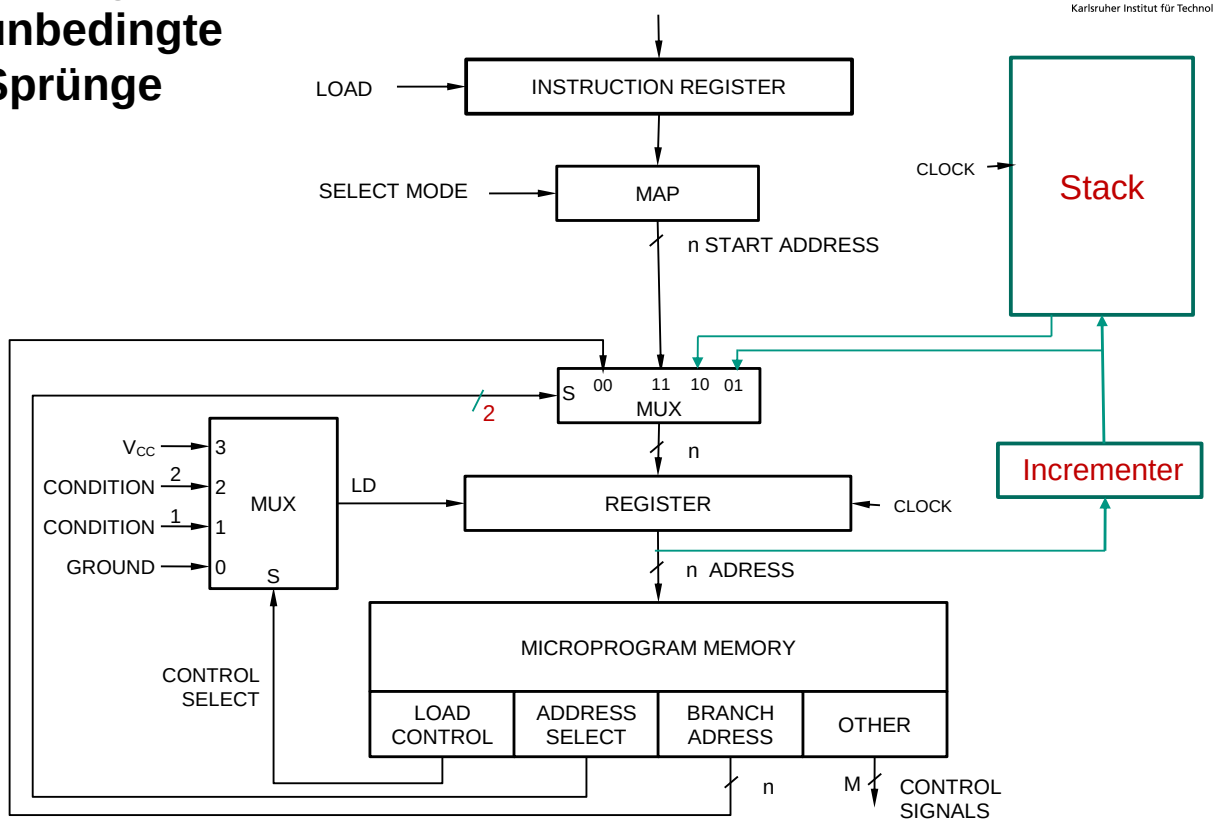
Unterprogrammaufruf mit Stack-Speicher (Last In - First Out Prinzip)



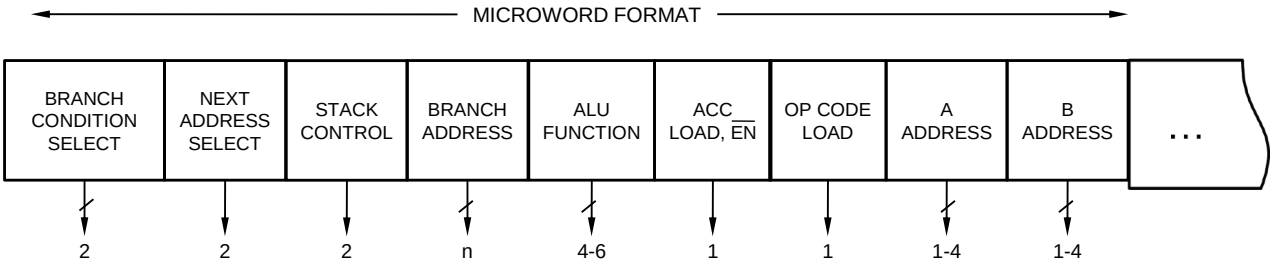
BRANCH CONDITION SELECT		BRANCH ADDRESS	STACK
1	1	80	PUSH
1	1	TOS	POP

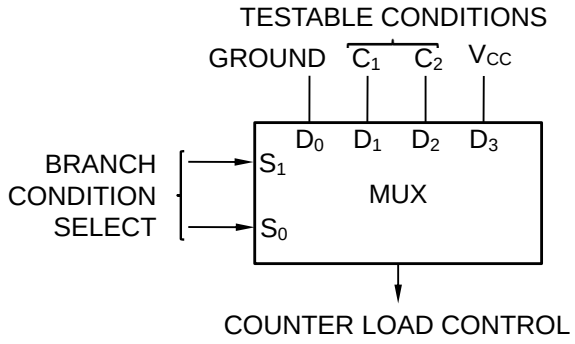


Bedingte und unbedingte Sprünge



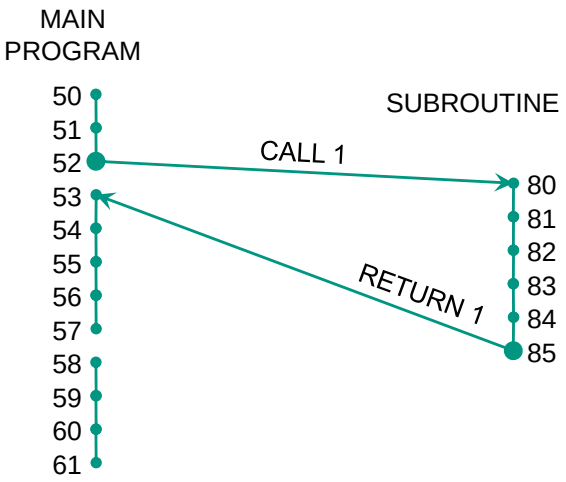
Bedingte und unbedingte Sprünge, Stack



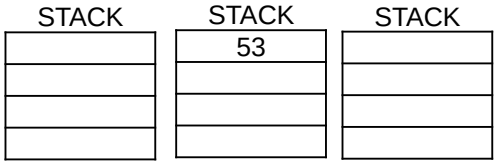


S_1	S_0	RESULT
0	0	INCREMENT COUNTER
0	1	BRANCH IF C_1 = TRUE
1	0	BRANCH IF C_2 = TRUE
1	1	LOAD COUNTER

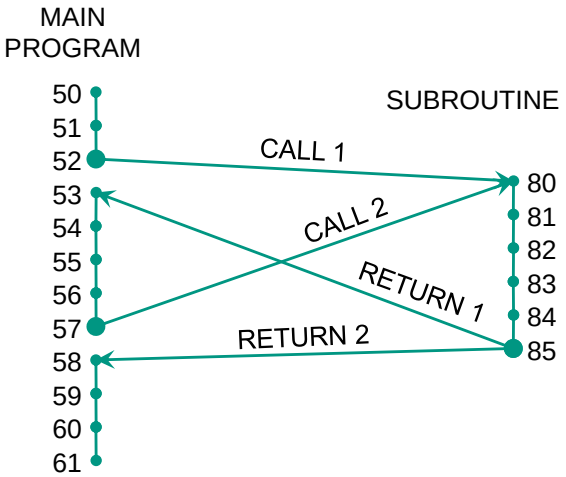
Unterprogrammaufruf mit Stack-Speicher (Last In - First Out Prinzip)



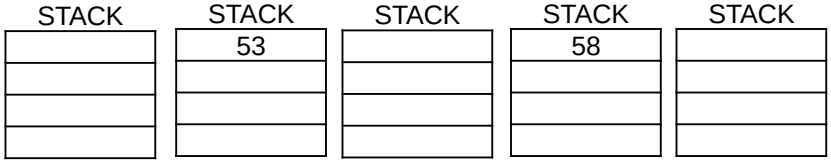
BRANCH CONDITION SELECT		BRANCH ADDRESS	STACK
1	1	80	PUSH
1	1	TOS	POP



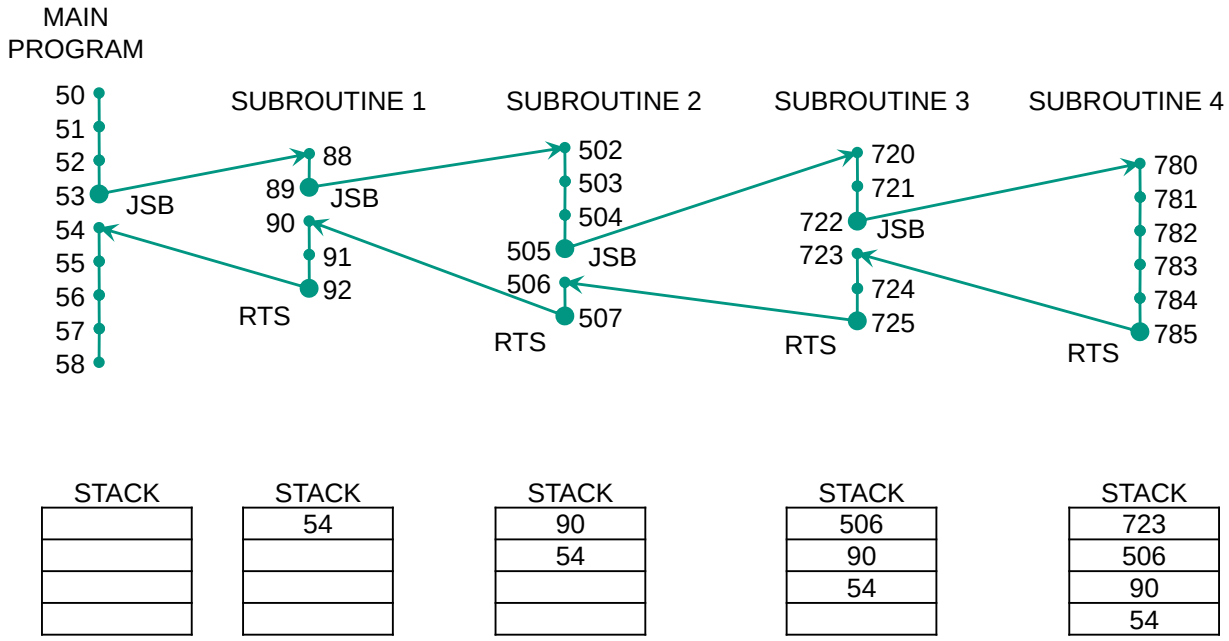
Unterprogrammaufruf mit Stack-Speicher (Last In - First Out Prinzip)



BRANCH CONDITION SELECT		BRANCH ADDRESS	STACK
1	1	80	PUSH
1	1	TOS	POP
1	1	80	PUSH
1	1	TOS	POP



Geschachtelter Unterprogrammaufruf mit Stack-Speicher



Unterbrechungsverarbeitung Interrupt

Feststellung Status von Eingabe/Ausgabe-Geräten oder zeitkritischen Prozessen

Häufig ist sofortige Reaktion auf Eingabe/Ausgabe-Ereignisse erforderlich,
(z. B. von Tastatur, Maus, Festplatte, Netzwerk, Zeitgeber/Timer, Zähler,
Division durch 0, Analog-Digital-Wandler, Watchdog, externe Unterbrechungen, usw.)
auch wenn ein anderer Programmcode (z. B. Anwendungsprogramm) gerade abgearbeitet wird

- Feststellen Status durch „Polling“ oder durch „Interrupts“
- Polling (programmierte zyklische Abfragen)
  einfach, benötigt keine zusätzliche Hardware, belegt aber ständig CPU-Leistung.
  Bei Multitasking-Betriebssystemen ist das Polling als alleinige Methode nicht möglich.
- Interrupt (engl. to interrupt, unterbrechen)
  Laufzeiteffizient, aber zusätzliche Hardware notwendig

Interrupt versus Polling - Analogie

- Standard-Analogie für Interrupts im Alltag:

  Tür mit Klingel:

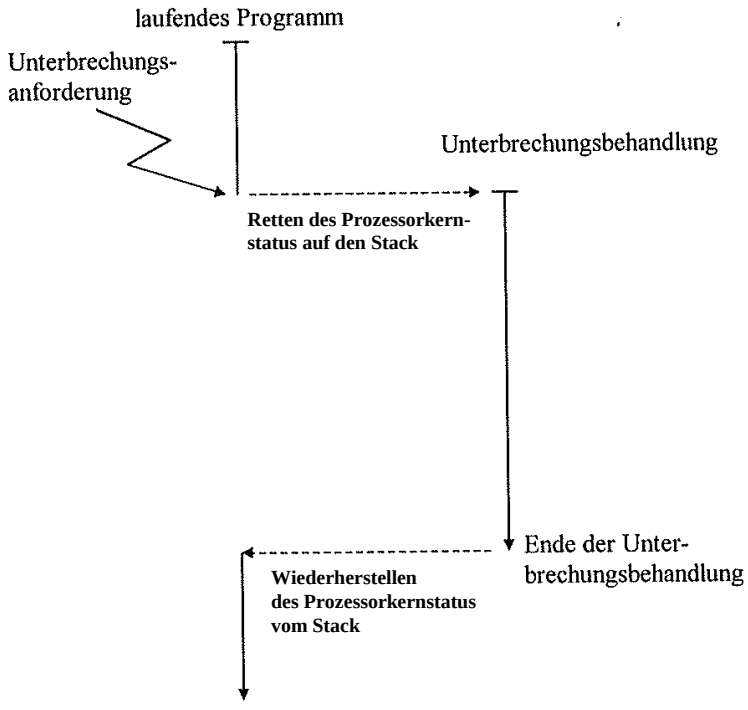
  - Während man seine Aufgaben erledigt, kann man jederzeit durch die Klingel unterbrochen werden, wenn ein Gast eine „Abarbeitung“ wünscht, und sich ihm dann zuwenden.
  - Beim Polling – also ohne Klingel – müsste ständig an die Tür gelaufen werden, um nachzuschauen, ob Besuch da ist oder nicht.

- Beim Erhitzen von Milch hingegen ist es wohl besser, nicht erst auf den „Interrupt“ des Überkochens zu warten, sondern den Prozess vielmehr regelmäßig zu überwachen.

Unterbrechungsverarbeitung - Interrupt

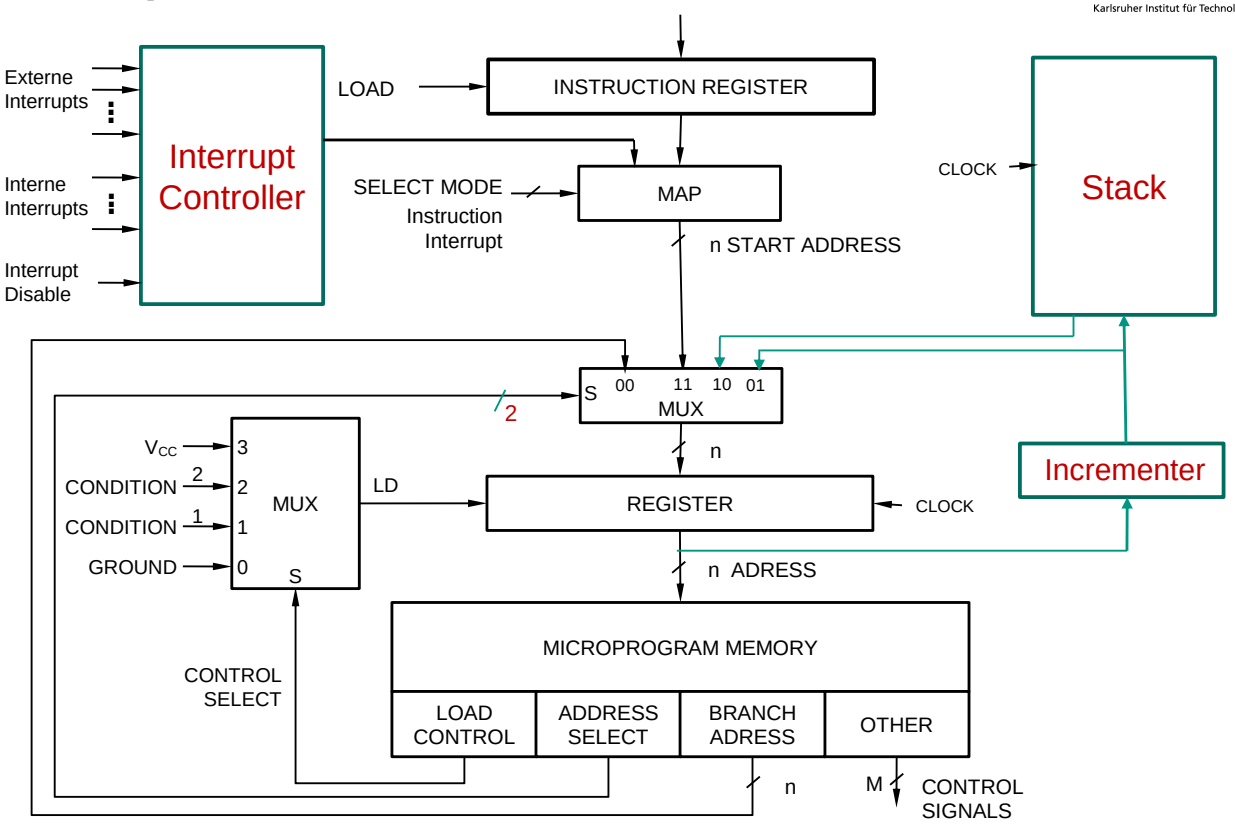
- Interrupt (engl. to interrupt, unterbrechen)
  die kurzfristige Unterbrechung eines Programms, um eine andere, meist kurze, aber zeitkritische Verarbeitung durchzuführen
- auslösendes Ereignis
  Unterbrechungsanforderung (Interrupt Request, IRQ)
- danach Ausführung der Unterbrechungsroutine
  (Interrupt Service Routine, ISR)
- anschließend wird die Ausführung des Programms an der Unterbrechungsstelle fortgesetzt
- Auslösung eines Interrupts nur dann, wenn die nächste Operation auf dem Interface (Hardware) nicht möglich ist
  (Puffer leer (Ausgabe), Puffer voll (Eingabe),
  Fehlermeldungen der Interface-Hardware,
  Ereignis ohne Datentransfer z. B. Timer)

Ablauf einer Unterbrechung

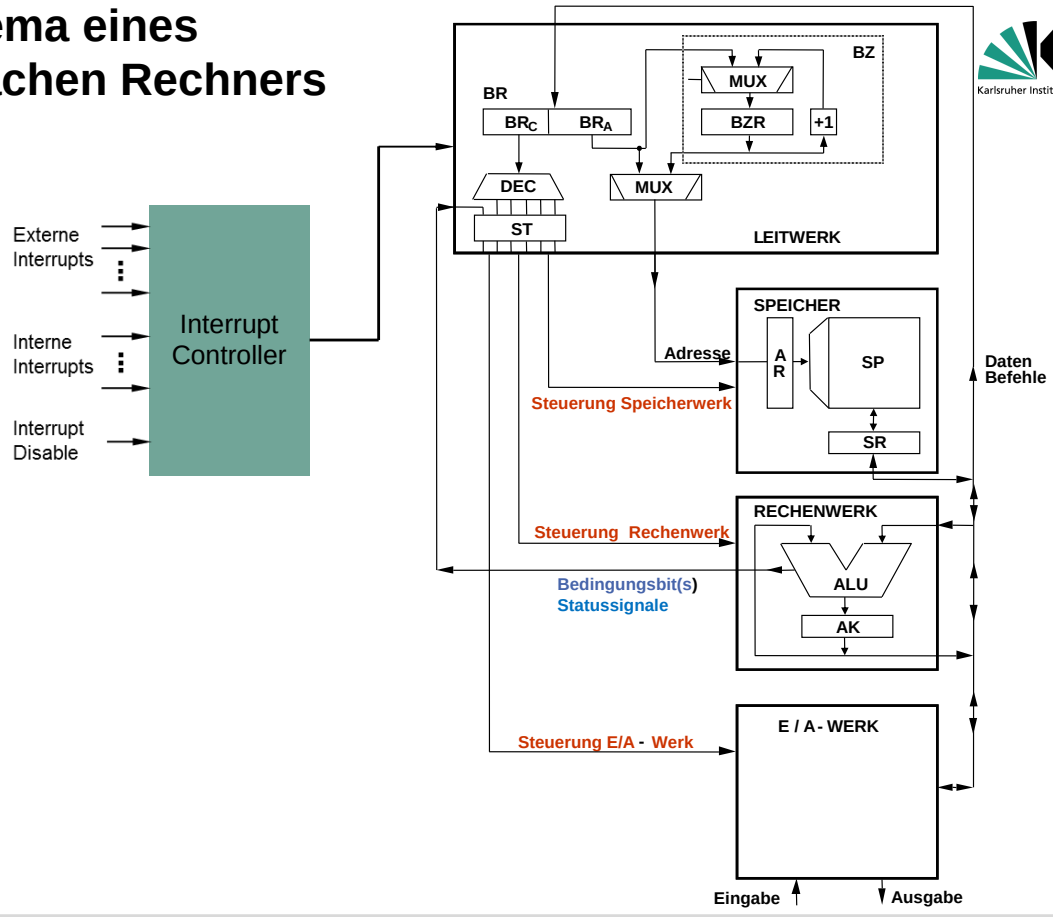


Aus Brinkschulte, Ungerer
„Mikrocontroller, Mikroprozessoren"
Springer-Lehrbuch

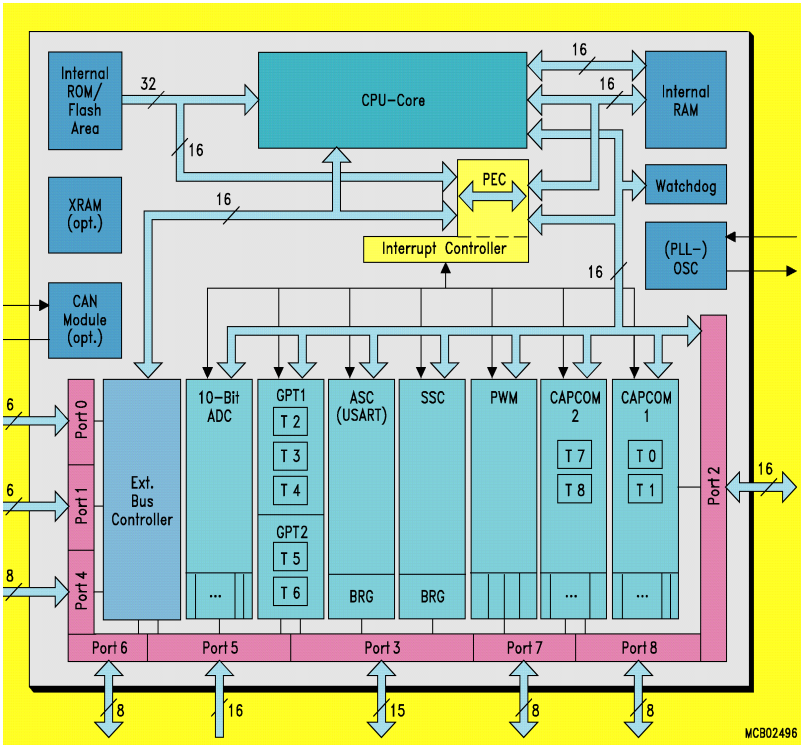
Interrupt Control



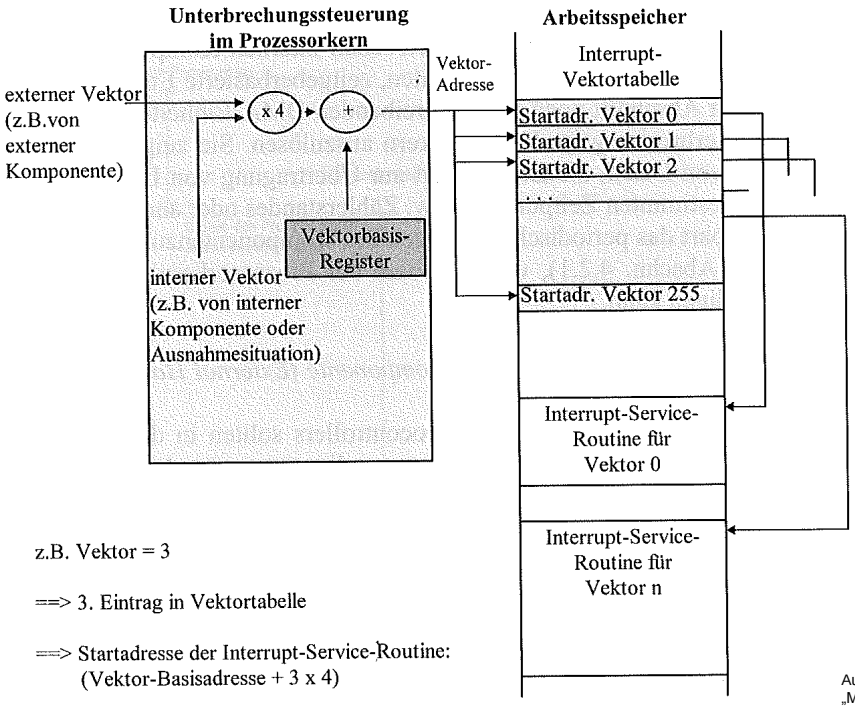
Schema eines einfachen Rechners



Block Diagram Infineon C167 microcontroller



Ermittlung Startadresse ISR durch Interrupt Controller



z.B. Vektor = 3

==> 3. Eintrag in Vektortabelle

==> Startadresse der Interrupt-Service-Routine:
(Vektor-Basisadresse + 3 x 4)

Aus Brinkschulte, Ungerer „Mikrocontroller, Mikroprozessoren“ Springer-Lehrbuch

Unterbrechungsverarbeitung - Interrupt

- Beispiel für Anwendung von Interrupts:

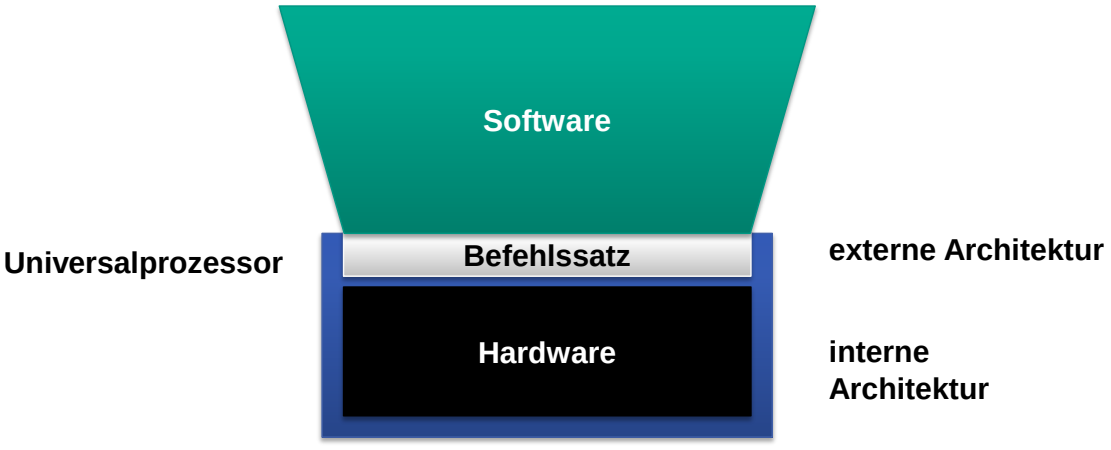
  Prozessor vergibt Auftrag an eine Hardwarekomponente

  - muss nicht aktiv auf die Antwort warten (Polling), sondern kann in der Zwischenzeit andere Aufgaben erledigen
  - wird später mittels Interrupt durch jene Hardwarekomponente wieder zu dieser Ein-/Ausgabeaktion zurückgeholt.

präemptive Multitasking-Betriebssysteme
sind ohne Interrupts nicht möglich, da Programme sonst nicht mehr unterbrochen, vom Betriebssystem umgeschaltet (Timesharing) und Ein-/Ausgabegeräte nicht mehr bedient werden können.

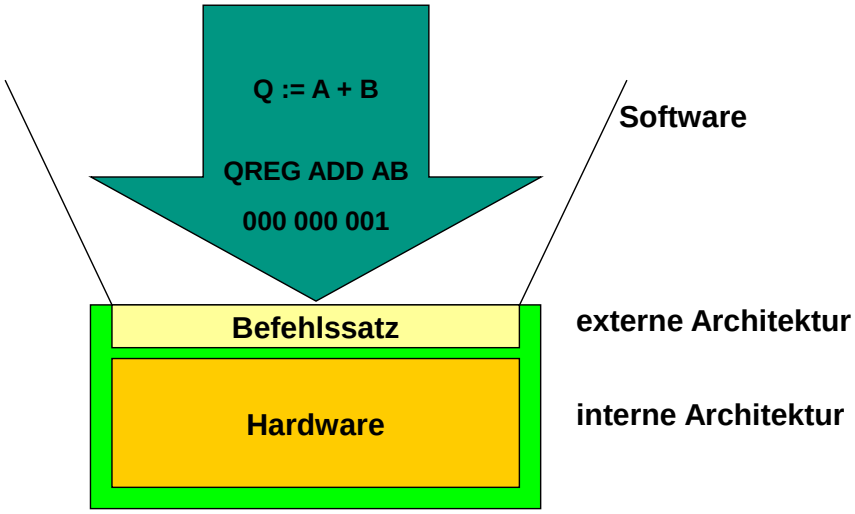
Grundlegende Begriffe

- Externe Architektur: Befehlssatzarchitektur (Instruction Set Architecture ISA)
- Interne Architektur: der innere Hardwareaufbau eines Rechners
- Universalprozessoren: für alle Anwendungen von einfachem C-Programm bis zum Compiler, Betriebssystem, Datenbank...



Rechnerarchitekturen

- Maschinenbefehlssatz definiert die dem Programmierer sichtbaren Befehle
- Häufig wird ISA (Instruction Set Architecture) mit der Prozessor Architektur gleichgesetzt
- Interne Architektur (Prozessor Mikroarchitektur) definiert die interne Struktur des Prozessors
  - Verschiedene interne Architekturen können dieselbe externe Architektur aufweisen
  - Computerfamilie



Instruction Set Architecture ISA

- Die Sicht des Programmierers auf den Computer hängt ab von der Antwort auf fünf Fragen:
  - Wie werden Daten repräsentiert?
  - Wo können Daten gespeichert werden?
  - Wie kann auf Daten zugegriffen werden?
  - Welche Operationen können mit den Daten ausgeführt werden?
  - Wie werden Befehle codiert?
  - Die Antwort auf diese Fragen definiert die externe Architektur oder Befehlssatzarchitektur
    Englisch: Instruction Set Architecture (ISA)

Datenrepräsentation Datenformate

- ISA unterstützt verschiedene Datenformate
  - Integer (z.B. Byte, 16-bit Word, 32-bit Longword, and 64-bit Quadword
  - Es gibt zwei Arten Byte-Adressen in einem Wort zu ordnen:
    - big-endian: most significant byte first und
    - little-endian: least significant byte first
  - Gepackte und ungepackte BCD Zahlen
  - ASCII Character
  - Floating-point data formats (ANSI/IEEE 754-1985):
    standard, basic oder extended, je zwei Längen: single or double
  - Multimedia: 32-, 64- oder 128-bit Worte, 8- or 16-bit pixel Farbtiefe.

Datenspeicherung - Adressraum

- Adressraum Assemblerebene:
  register space, stack space, I/O space, control space
- Außer Register werden alle anderen Adressräume auf einen einzigen zusammenhängenden Speicheradressraum abgebildet
- Eine Reduced Instruction Set (RISC) ISA enthält zusätzlich einen relativ großen Registersatz (general-purpose CPU registers)
- Heutige RISC Prozessoren zusätzlicher Satz (z.B. 32) 64-bit floating-point Register.

Weitere Details ab Folie 12-121

Datenzugriff - Adressierungsmodi (1)

Register	load Reg1,Reg2 Reg1 ← (Reg2)
Immediate	load Reg1,#const Reg1 ← const
Direct	load Reg1,(const) Reg1 ← Mem[const]
Register indirect	load Reg1,(Reg2) Reg1 ← Mem[(Reg2)]

Datenzugriff - Adressierungsmodi (2)

Post-Increment	load Reg1,(Reg2)+ Reg1 ← Mem[(Reg2)], Reg2 ¬ (Reg2) + step
Pre-Decrement	load Reg1,-(Reg2) Reg2 ← (Reg2) - step, Reg1 ¬ Mem[(Reg2)]
Displacement	load Reg1,displ(Reg2) Reg1 ← Mem[displ + (Reg2)]
Indexed and scaled indexed	load Reg1,(Reg2*scale) Reg1 ← Mem[(Reg2)*scale]

Register mode: the operand is stored in one of the registers.
Immediate (or literal) mode: the operand is a part of the instruction.
Direct (or absolute) mode: the address of the operand in memory is stored in the instruction.
Register indirect (or register deferred) mode: the address of the operand in memory is stored in one of the registers.
Autoincrement (or register indirect with postincrement) mode: like the register indirect, except that the content of the register is incremented after the use of the address. This mode offers automatic address increment useful in loops and in accessing byte, halfword, or word arrays of operands.
Autodecrement (register indirect with predecrement) mode: the content of the register is decremented and is then used as a register indirect address. This mode can be used to scan an array in the direction of decreasing indices.

Datenzugriff - Adressierungsmodi (3)

Indirect scaled indexed	load Reg1,(Reg2,Reg3*scale) Reg1 ← Mem[(Reg2) + (Reg3)*scale]
Indirect scaled indexed with displacement	load Reg1,displ(Reg2,Reg3*scale) Reg1 ← Mem[displ + (Reg2) +(Reg3)*scale]
PC-relative	branch displ PC ← PC + step + displ (if branch taken)

Operationen mit Daten

- Datentransfer
- Arithmetisch
- Floating Point
- Logisch
- Bedingte und unbedingte Verzweigung
- Shift, Rotate
- Bit Manipulation
- Multimedia
- (Systemsteuerung, Coprozessorbefehle)

Je nach Befehlsart vom Typ
register-register, memory-register,
register-memory, oder memory-memory.

Befehlsformate

- 3-Adress Befehlsformat: opcode | Dest | Src1 | Scr2
  typische Verwendung für Register-Register Befehl
  (Load/Store Machines)
- 2-Adress Befehlsformat: opcode | Dest/Src1 | Src2
  (Register-Memory Machines)
- 1-addressBefehlsformat: opcode | Src
- (Accumulator Machines)
- 0-address Befehlsformat: nur Befehlscode
  (Stack Machines)

Klassifizierung

- Akkumulator-Maschine
- Stack-Maschine
- Registersatzmaschine
  - CISC-Architektur
  - RISC-Architektur

Akkumulatormaschine

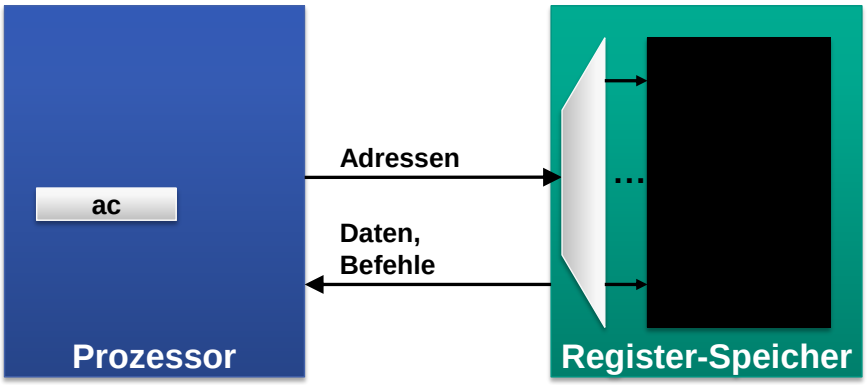
- Organisationsprinzip vieler einfacher Rechner
- Die Rechenergebnisse werden in einem speziellen Register, dem Akkumulator (AC), „akkumuliert“

- Typische Befehle:
  - LAC m: lade Wert unter Adresse m in den AC
  - SAC m: speichere den Inhalt von AC nach Adresse m
  - ADD m: addiere den Inhalt von AC mit dem Wert unter Adresse m und speichere das Resultat nach AC

Beispiel: $c := a+b$

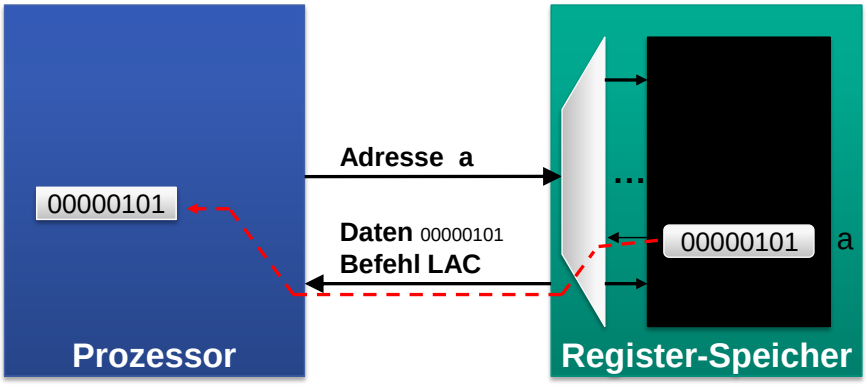
```
LAC a
ADD b
SAC c
```


Struktur der Akkumulatormaschine



- Prozessor beinhaltet den Akkumulator ac
- Speicher beinhaltet Adressmultiplexer und die Register

Befehlsausführung

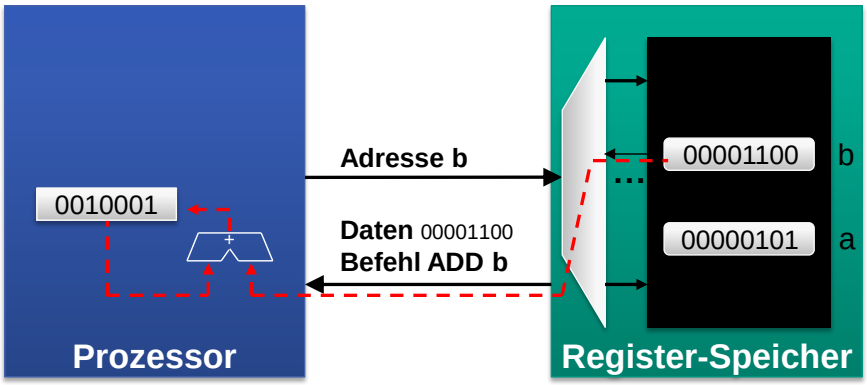


Beispiel: *c := a + b*

LAC a
ADD b
SAC c

- Daten aus Register a in den Akkumulator laden

Befehlsausführung

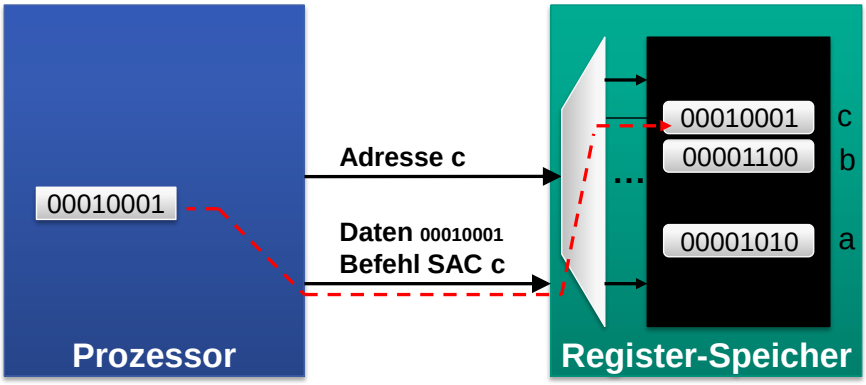


Beispiel: *c := a + b*

LAC a
ADD b
SAC c

- Inhalt des Registers b zum Inhalt des Akkumulators addieren
- Ergebnis in Akkumulator ablegen

Befehlsausführung



Beispiel: *c := a + b*

LAC a
ADD b
SAC c

- Daten aus Akkumulator in das Register c schreiben

Stackmaschine

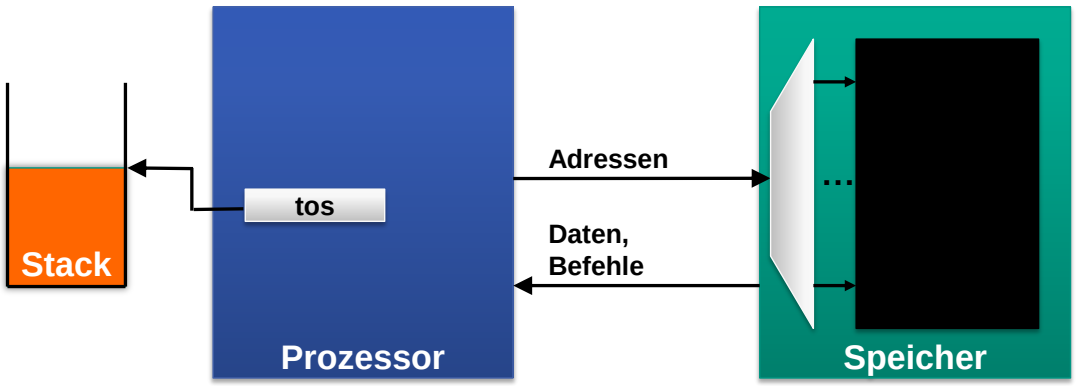
- Befehle beziehen sich auf einen Stack (Stapelspeicher) zur Speicherung u. a. von Zwischenergebnissen

- Typische Befehle:
  - PUSH m: lege den Wert unter der Adresse m auf den Stack
  - ADD: addiere die beiden obersten Werte des Stacks, entferne sie und lege die Summe als oberstes Elemente auf den Stack
  - STORE m: speichere das oberste Element auf dem Stack nach der Adresse m

Beispiel: *c := a+b*

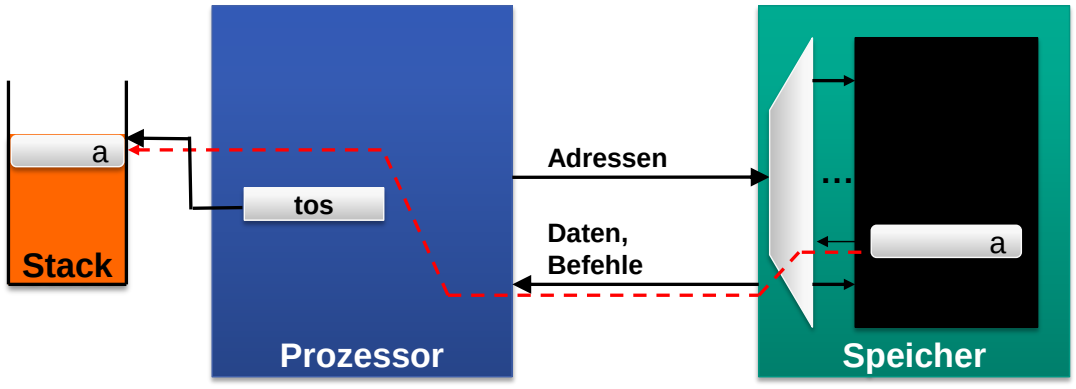
PUSH a
PUSH b
ADD
STORE c

Struktur der Stackmaschine



- Der Prozessor hat Zugriff auf einen Zwischenspeicher mit LIFO-Prinzip, den „Stack“ oder Stapelspeicher
- Der Prozessor verwaltet einen Verweis auf das oberste Stackelement: tos – top of stack

Befehlsausführung

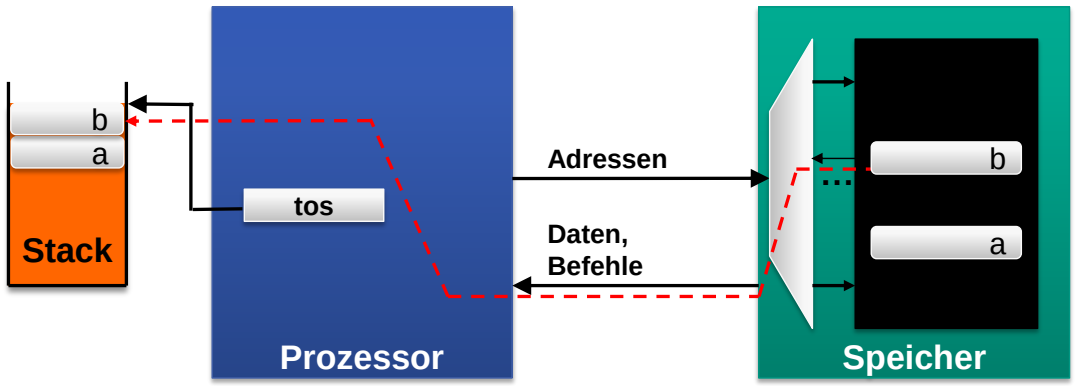


Beispiel: *c := a + b*

PUSH a
PUSH b
ADD
STORE c

- Daten aus Register a auf den Stack legen

Befehlsausführung

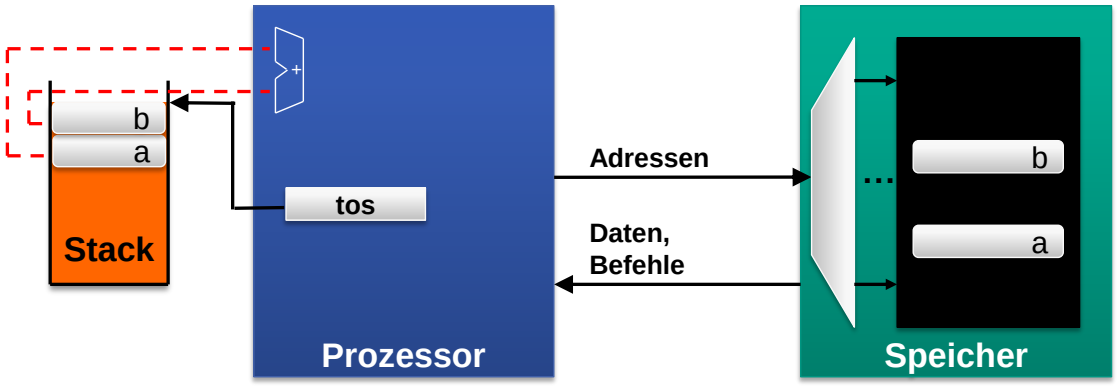


Beispiel: *c := a + b*

PUSH a
PUSH b
ADD
STORE c

- Daten aus Register b auf den Stack legen

Befehlsausführung

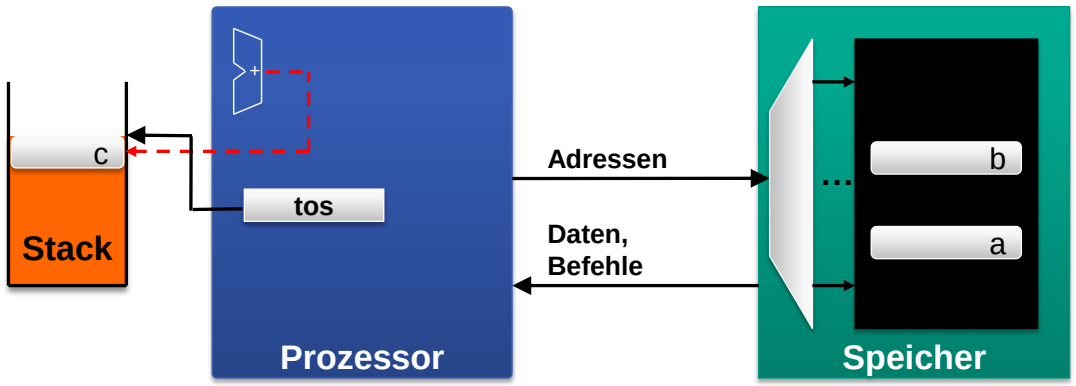


Beispiel: *c := a + b*

PUSH a
PUSH b
ADD
STORE c

- Die ersten beiden Werte auf dem Stack addieren

Befehlsausführung

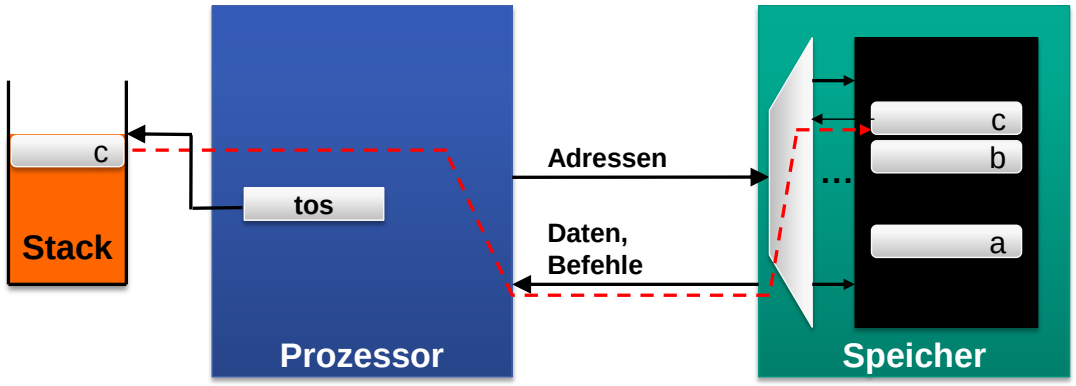


Beispiel: *c := a + b*

PUSH a
PUSH b
ADD
STORE c

- Die Summe wird dann wieder auf dem Stack abgelegt

Befehlsausführung



Beispiel: *c := a + b*

PUSH a
PUSH b
ADD
STORE c

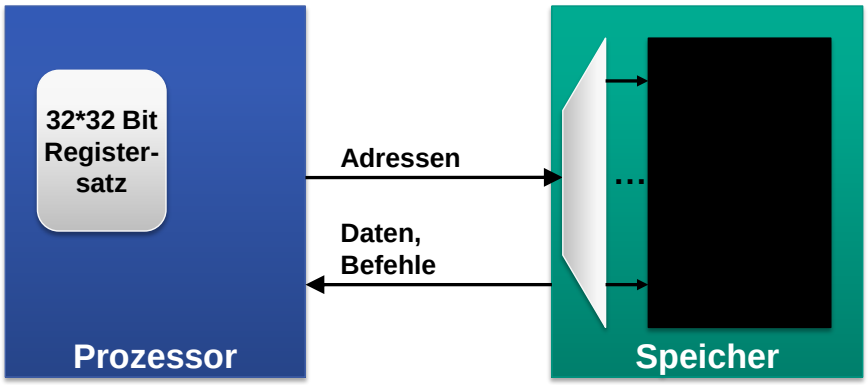
- Das oberste Element des Stacks wird an Adresse c gespeichert

Registersatzmaschine

- Der Prozessor enthält z.B. 16 oder 32 Universalregister
- Üblich sind Dreiadressbefehle: *OP DEST, SRC1, SRC2*
  d.h. *DEST := SRC1 OP SRC2*
- Oder Zweiadressbefehle: *OP DEST, SRC*
  d.h. *DEST := DEST OP SRC*

  z.B.: *ADD R5, R6* d.h. *R5 := R5 + R6*
- Beispiel IBM 360 (1964), IBM 370, DEC VAX 11/780:
  Rechnerfamilie d.h. eine externe Architektur, mehrere interne Architekturen, Resultat: Hardware mit sehr unterschiedlichem Preis/Leistungsverhältnis.

Struktur der Registersatzmaschine



- Im Prozessor befindet sich kein Akkumulator, es existiert auch kein Stack
- Zur Berechnung befüllt der Prozessor seinen internen Registersatz mit Werten aus dem Speicher

Beispiel

C = A + B
D = C – B

codiert in vier Klassen von ISA Befehlsformaten:

Machine			
Register-Register	**Register-Memory**	**Accumulator**	**Stack**
`load Reg1,A`	`load Reg1,A`	`load A`	`push B`
`load Reg2,B`	`add Reg1,B`	`add B`	`push A`
`add Reg3,Reg1,Reg2`	`store C,Reg1`	`store C`	`add`
`store C,Reg3`	`load Reg1,C`	`load C`	`pop C`
`load Reg1,C`	`sub Reg1,B`	`sub B`	`push B`
`load Reg2,B`	`store D,Reg1`	`store D`	`push C`
`sub Reg3,Reg1,Reg2`			`sub`
`store D,Reg3`			`pop D`

Typen von Registersatzmaschinen

CISC - Complex instruction set computer

- Sehr umfangreiche Befehlssätze
- Komplizierte Adressiermodi (z.B: Direktoperand, Registeradressierung…)
- Operanden, Register und Speicheradressen
- Versuch, die Konstrukte höherer Programmiersprachen durch teilweise sehr komplizierte Befehle zu unterstützen
- Beispiele: DEC VAX 11/780 (1979), INTEL x86 (1981)

RISC - Reduced instruction set computer

- Auch: Load/Store-Architektur genannt
- Wenige, i.A. gleich lange, einfache Instruktionen
- Nur Register als Operanden von Verknüpfungen wie z.B: Addition
- i.A. größere Programme als bei CISC
- 2 bis 5 fache Leistung bei gleicher Technologie
- Beispiele: SPARC, Alpha, PowerPC

CISC DEC VAX 11/780

- Complex instruction set
- Microcode
- Consisting of 304 instructions
- 16 addressing modes
- More than 10 different instruction lengths

The ten most frequently used instructions in the SPECint92 for Intel x86

Instruction	Average (% total executions)
load	22
conditional branch	20
compare	16
store	12
add	8
and	6
sub	5
move register-register	4
call	1
return	1
Total	**95**

Rechnerarchitekturen

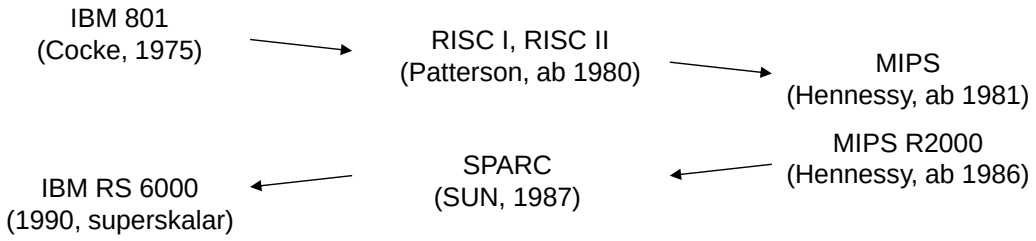
- **RISC (reduced instruction set computer)**, auch Load/Store-Architektur
- Beispiele: SPARC, Alpha, PowerPC
  - Wenige, i.a. gleich-lange, einfache Instruktionen
  - Nur Register als Operanden von Verknüpfungen wie z.B. Addition
  - Beispiel c := a + b

```
LOAD R1, a
LOAD R2, b
ADD R3, R2, R1
STORE c, R3
```

Verknüpfung nur zwischen Registern

Load/Store-Befehle für den Datentransport zwischen CPU und Speicher

  - i.A. größere Programme als bei CISC
  - 2 bis 5 fache Leistung bei gleicher Technologie
  - Historie:



Vergleich: RISC und CISC

	Programmgröße		Transportierte Instruktionen und Daten		CPU Zeit	
	VAX	DEC 3100	VAX	DEC 3100	VAX	DEC 3100
GNU C	410 kB	688 kB	18 MB	21 MB	291 s	90 s
TeX	159 kB	217 kB	67 MB	78 MB	449 s	95 s
Spice	223 kB	372 kB	99 MB	106 MB	352 s	94 s

- RISC Programme (DEC) dagegen sind länger bei gleichzeitig schnellerer Ausführung
- CISC Programme (VAX) sind meist kleiner, benötigen jedoch eine längere Ausführungszeit

Verarbeitungsphasen bei der Befehlsausführung

- genauere Betrachtung der Phasen einer Befehlsausführung
  - Befehl holen
  - Befehl dekodieren
  - Operand holen
  - Operation ausführen
  - Ergebnis abspeichern
  - Programmzähler erhöhen für nächsten Befehl holen

Verarbeitungsphasen bei der Befehlsausführung

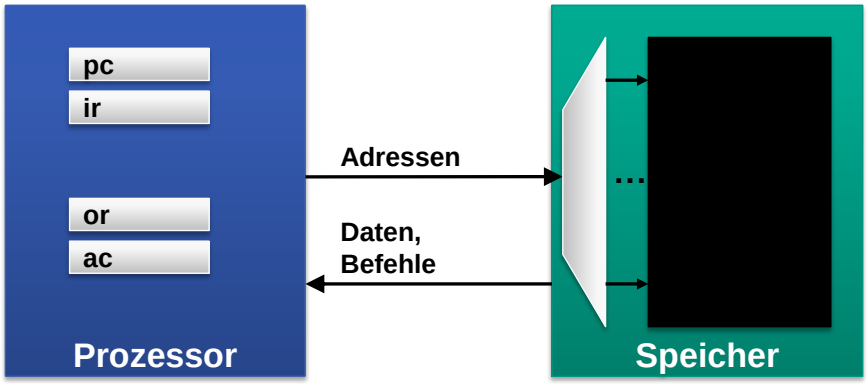
- Verarbeitungsphasen eines Befehls am Beispiel
  - Annahme: einfache Akkumulatormaschine
  - Befehlsformat

Befehlscode
Addiere-Befehl ≅ 7

Adresse 24

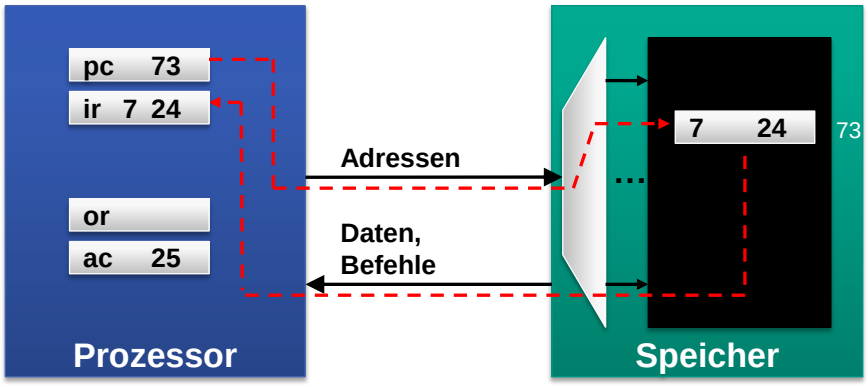
  - Addiere-Befehl holt den Wert unter der im Befehl angegebenen Adresse (im Beispiel 24) und addiert ihn zum Inhalt

Einfache Akkumulatormaschine

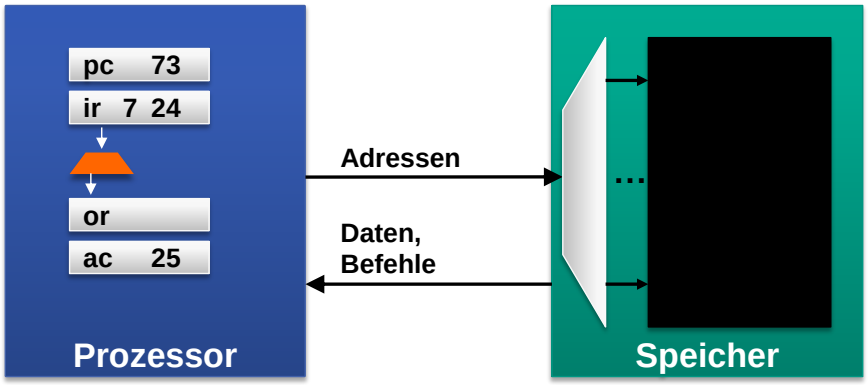


- pc (program counter, auch ip: instruction pointer)
- ir (instruction register)
- or (operand register)
- ac (accumulator)

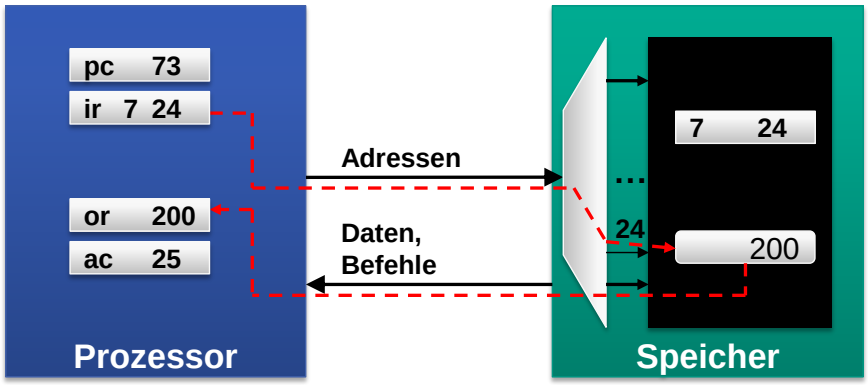
1. Befehl holen (instruction fetch, IF)



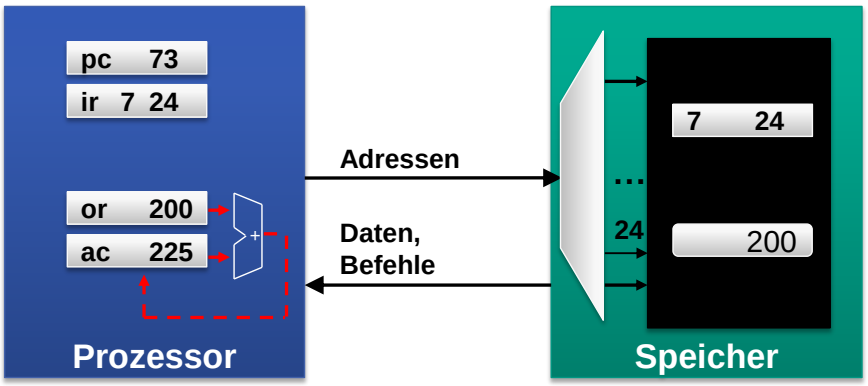
2. Befehl dekodieren (instruction decode, ID)



3. Operand holen (operand fetch, OF)

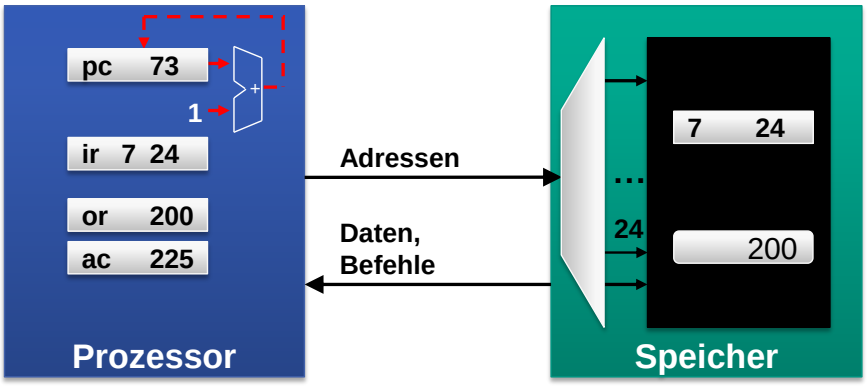


4. Befehl ausführen (instruction execution, EX)

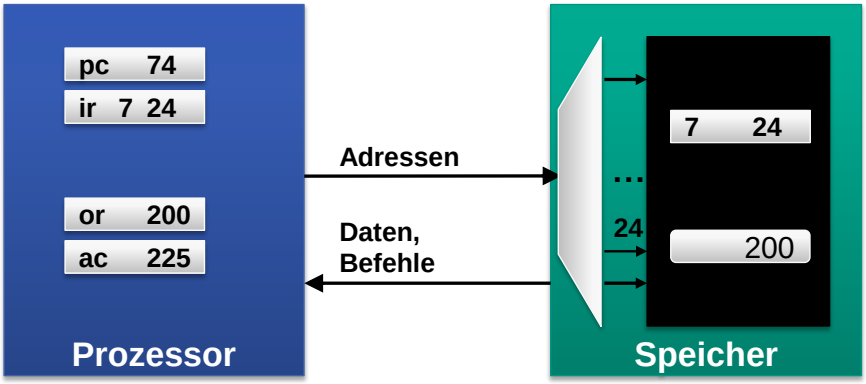


5. Befehlszähler inkrementieren

- Kann als separater Schritt oder parallel erfolgen



Resultat

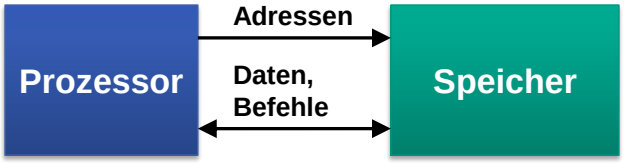


Beschleunigte Befehlsausführung?!

- Parallellisierung von Befehls- und Datenzugriffen im Hauptspeicher
  - Harvard Architektur
- Überlappende Befehlsausführung
  - Pipelining
- Mehrere Rechenwerke
  - Superskalar
- Beschleunigung Hauptspeicherzugriff
  - Multi Level Caches
- Multiprozessor / Multi Core Systeme

Verschiedene Architekturen

- Von Neumann-Architektur:



- Harvard-Architektur:

- Vorteil Harvard-Architektur: Kein „von Neumann-Flaschenhals“ (Daten und Befehle teilen sich den gleichen Übertragungsweg)

Beschleunigte Befehlsausführung

Parallelisierung von Befehls- und Datenzugriffen im Hauptspeicher

Harvard Architektur

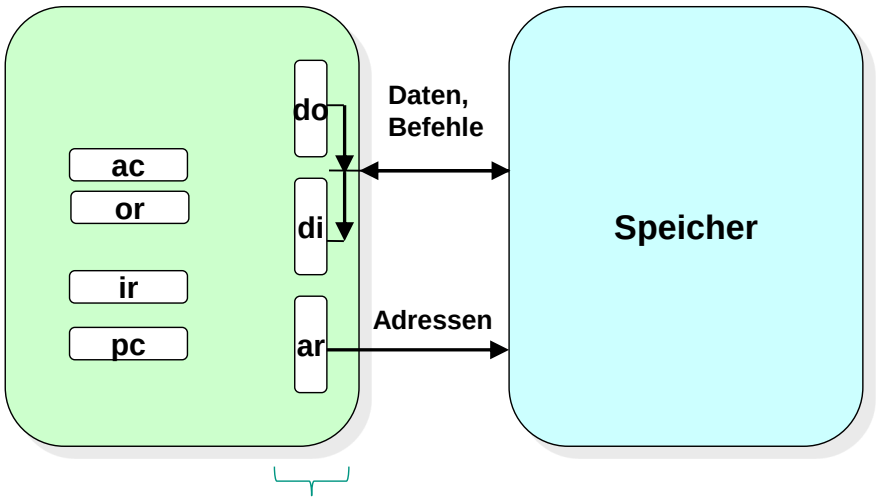
Seriell **Überlappung von Befehlshol- und Ausführungsphase**
Techniken zur Beschleunigung der Instruktionsausführung

- Seriell mit Überlappung Hol-/Ausführungsphase
- Pipelining
- Superpipelining
- Superskalar

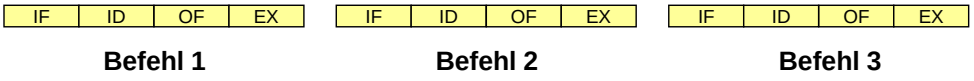
Vervielfältigung von Funktionseinheiten

Befehlsausführung

- Beispiel: einfache Akkumulatormaschine

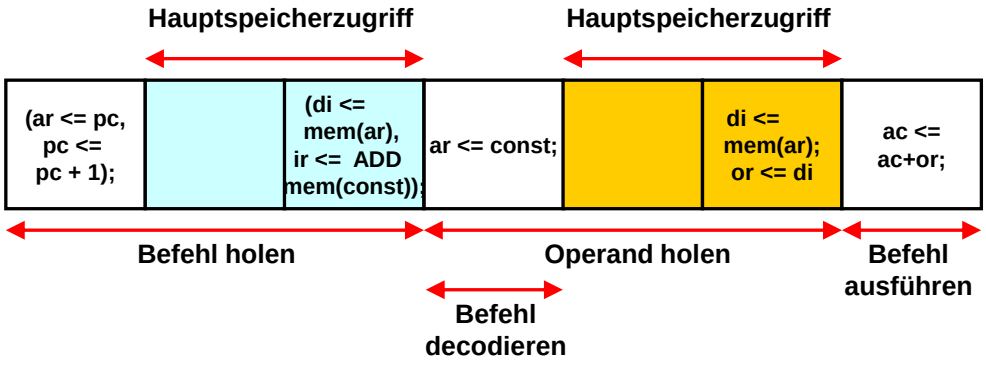


Befehlsausführung seriell



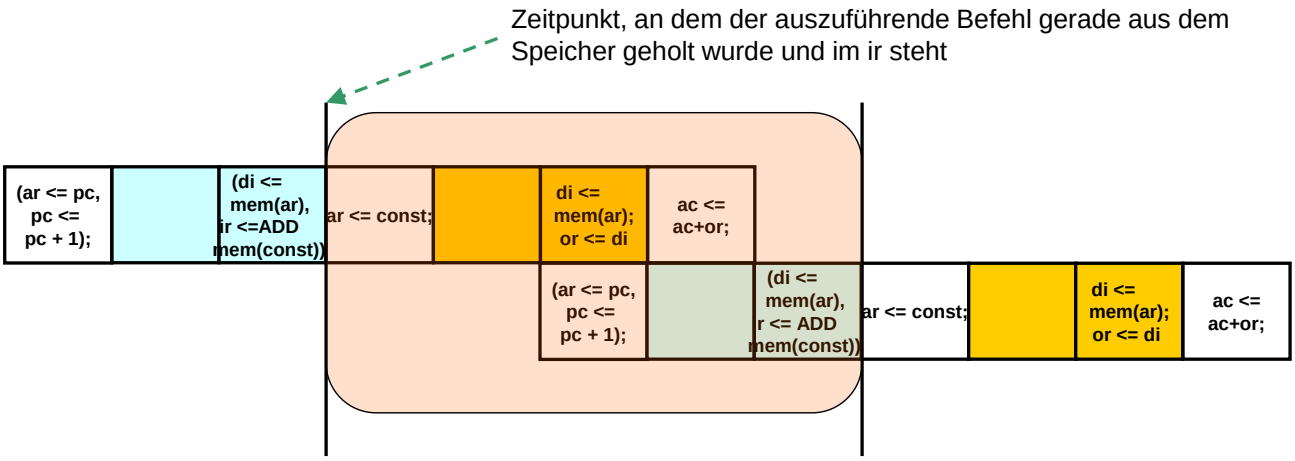
Befehlsausführung - Annahmen

- Hauptspeicherzugriffszeit = 2 * Prozessortaktzeit
- Befehlsauslegung für größtmögliche Geschwindigkeit
- Beispiel: Addiere-zu-Akkumulator-Befehl
- Auslastung CPU: 71%, Speicher 57%



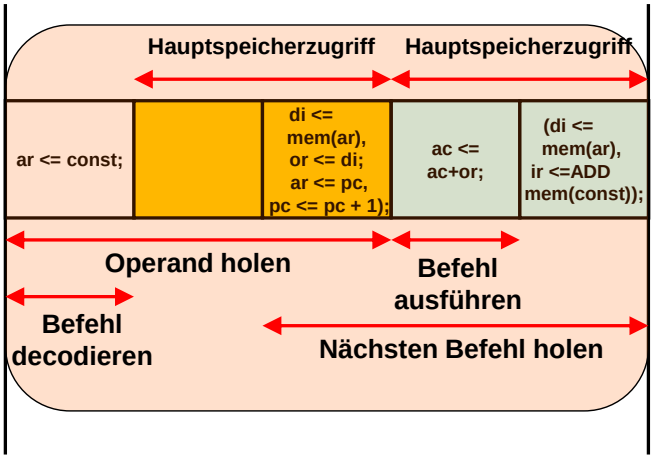
Befehlsausführung überlappend

- Überlappung der Ausführungsphase mit dem Holen des nächsten Befehls:



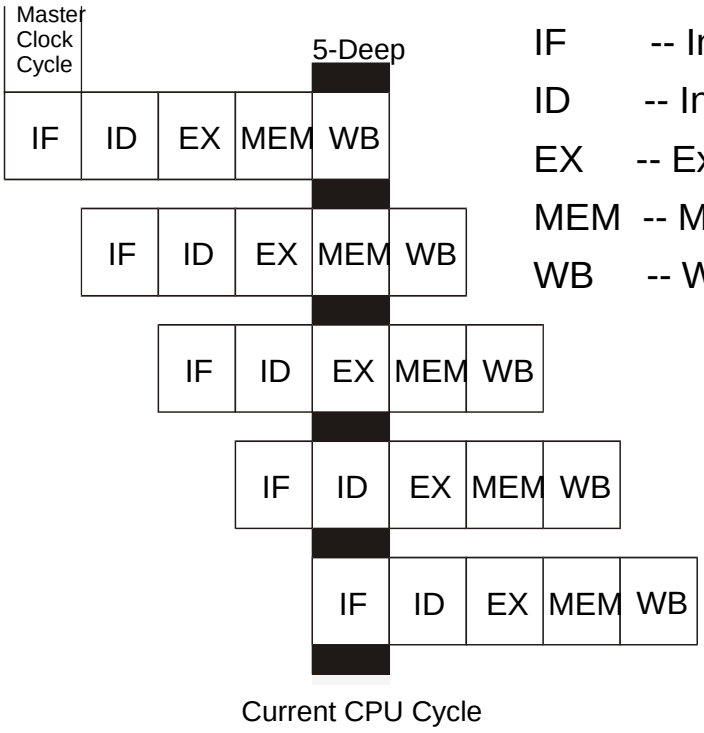
- Es werden keine zusätzlichen Ressourcen benötigt
- Eine zeitliche Einsparung von 2 Takten pro Befehl => 28,6 % Leistungssteigerung

Befehlsausführung überlappend Pipelining



- Während eines Speicherzugriffs wird bereits der nächste Speicherzugriff vorbereitet
- Ebenfalls Parallel zum Speicherzugriff: Kalkulation des Ergebnisses (Befehl ausführen)
- Nur noch 5 Takte notwendig
- Keine zusätzlichen Kosten
- Auslastung: CPU 80%, Speicher 80%

Typische Pipeline (fünf-stufig)



IF -- Instruction Fetch

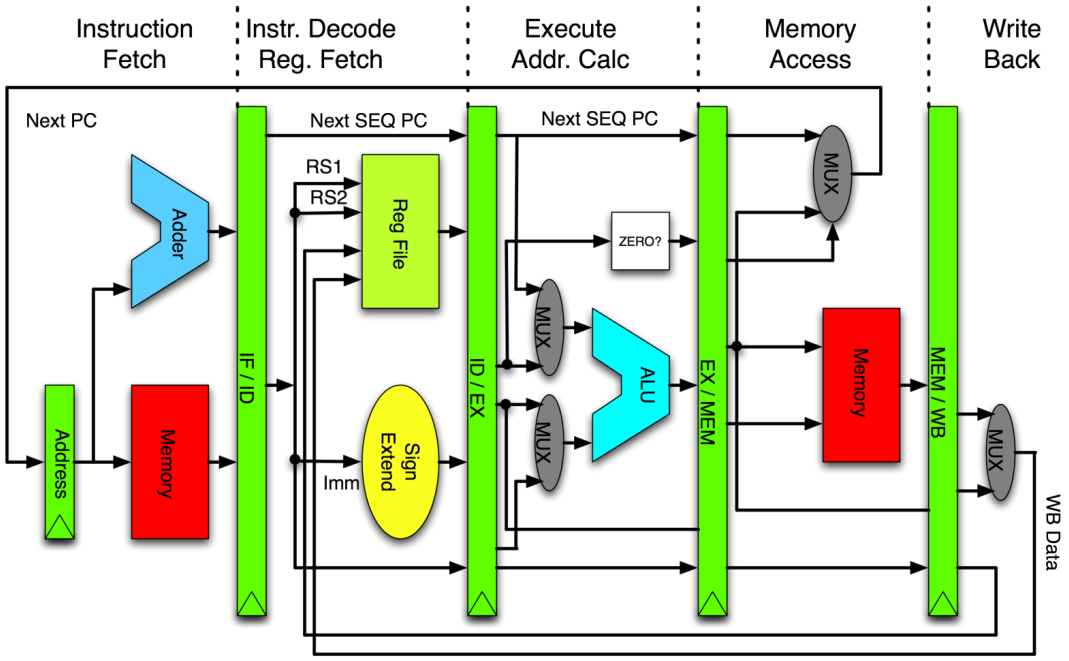
ID -- Instruction Decode/Register Fetch

EX -- Execute/Address Calculation

MEM -- Memory Access

WB -- Write Back

MIPS Technologies, Inc. RISC processor 5 stage pipeline



The stage-by-stage architecture of a MIPS microprocessor with a pipeline. Although the memory is shown twice for clarity of the pipeline, MIPS architectures have only one memory bank (i.e. von Neumann architecture).

Basic Pipeline Steps

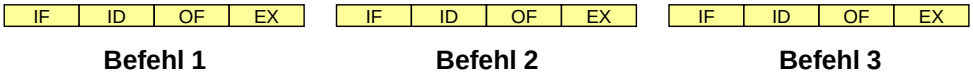
- Instruction fetch (IF): the instruction pointed to by the PC is fetched from memory into the instruction register of the CPU, and the PC is incremented to point to the next instruction in the memory.
- Instruction decode/register fetch (ID): the instruction is decoded, and in the second half of the stage the operands are transferred from the register file into the ALU input registers (here meaning: latches).
- Execution/effective address calculation (EX): the ALU operates on the operands from ALU input registers and eventually puts the result into ALU output register. The contents of this register depend on the type of instruction. If the instruction is:
  - Register-register (e.g. arithmetic/logical): the ALU outputs the result of the operation into the ALU output register;
  - Memory reference (e.g. load/store), the ALU output register contains an effective memory address;
  - Control transfer (e.g. branch on equal), then the ALU produces the jump / branch target address (which is stored in the ALU output register) and, at the same time, the branch direction.

Basic Pipeline Steps (continued)

- Memory access/branch completion (MEM): only for load, store, and branch instructions. If the instruction is:
  - register-register: the content of the ALU output register is transferred to the ALU result register.
  - load: the data is read from memory (as pointed to by the ALU output register) and is placed in the load memory data register;
  - store: the data in the store value register is written into the D-cache (as pointed to by the ALU output register);
  - control transfer: for jump and branch that is taken: the PC is replaced by the ALU output register content; otherwise, the PC remains unchanged (in both cases, the next step WB is skipped);
- Write back (WB): the result of the instruction execution (register-register or load instruction) is stored into the register file in the first half of the phase.
  In particular, the load memory data register or the ALU result register is written into the register file.

Fortgeschrittene Organisationsprinzipien

- Die Verarbeitung mehrerer Befehle nacheinander erfolgt im Prinzip in einer Schleife

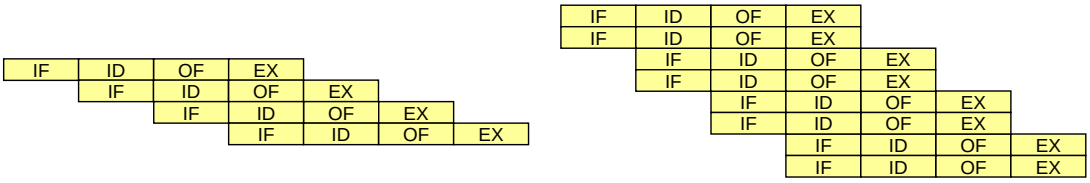


Befehl 1 **Befehl 2** **Befehl 3**

- Die Schleife kann auf verschiedene Arten parallelisiert werden

Pipelining: Überlappung der Ausführungsphase zur Reduktion der Latenzzeiten resultiert in einer höheren Verarbeitungsgeschwindigkeit

Superskalar: zusätzlich zum Pipelining paralleles Ausführung von mehreren Befehlen, im Idealfall Vervielfachung der Leistung

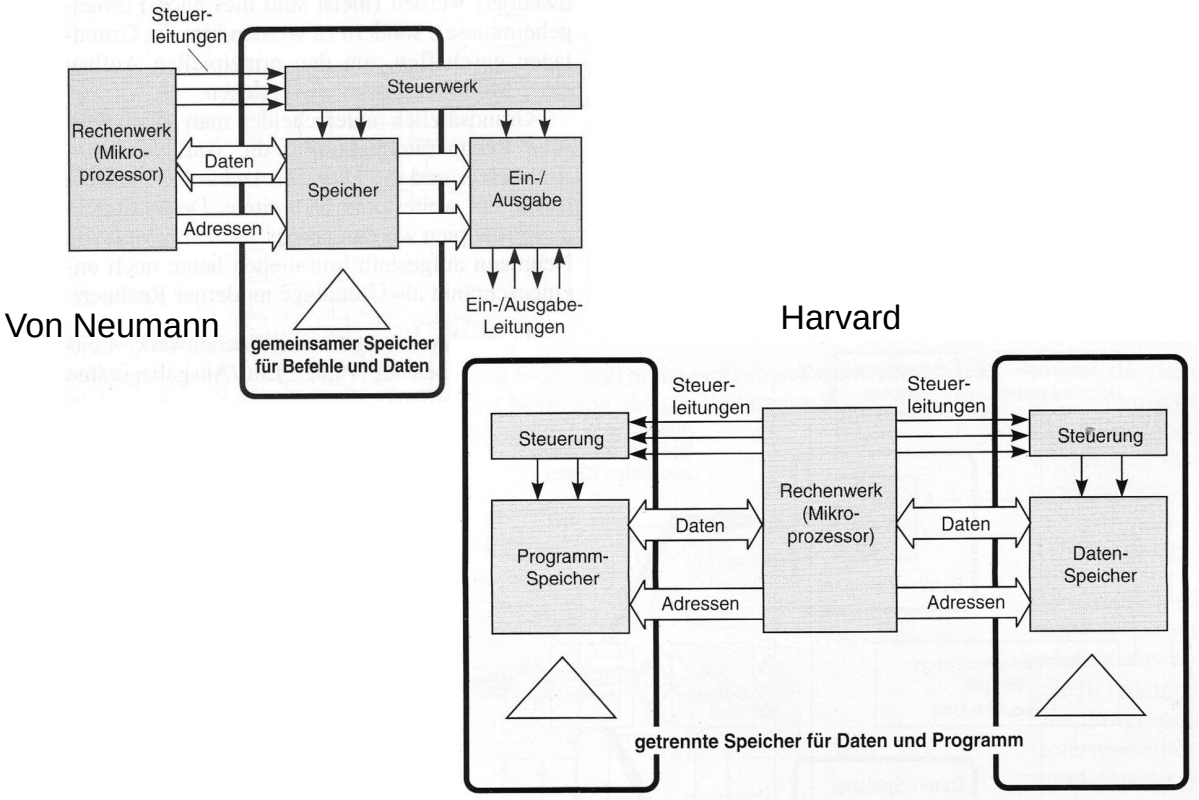


Pipelining **Superskalar**

Fortgeschrittene Organisationsprinzipien

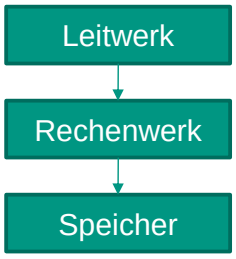
- Superskalar
- VLIW Very long instruction word computer
- EPIC explicitly parallel instruction computing
- Multi-Prozessor / Multi-Core Systeme
  - SISD, SIMD
  - MIMD, MISD

Von Neumann und Harvard Architektur

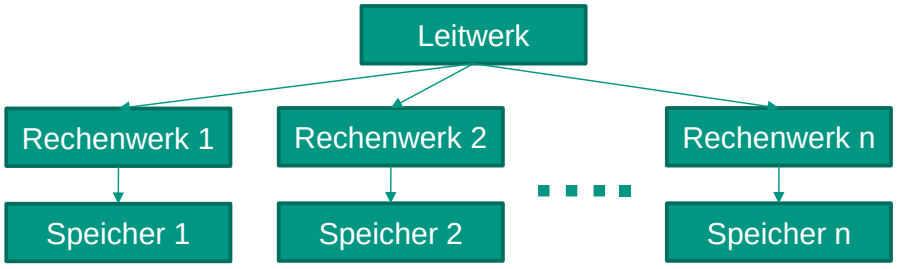


Klassifikation nach Flynn (1966)

- Single Instruction, Single Data (SISD):

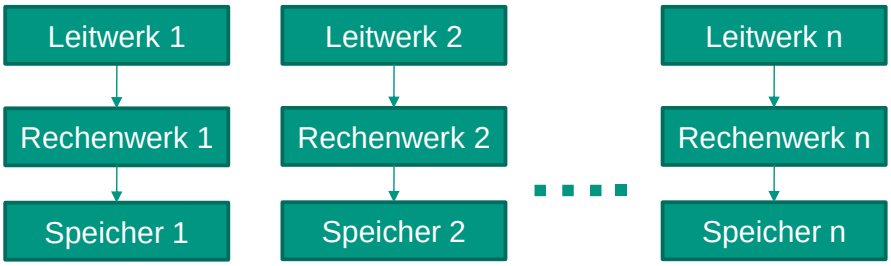


- Single Instruction, Multiple Data (SIMD):

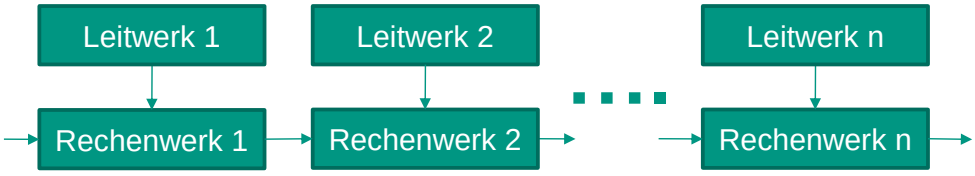


Klassifikation nach Flynn (1966)

- Multiple Instruction, Multiple Data (MIMD):



- Multiple Instruction, Single Data (MISD):



Beschleunigte Befehlsausführung

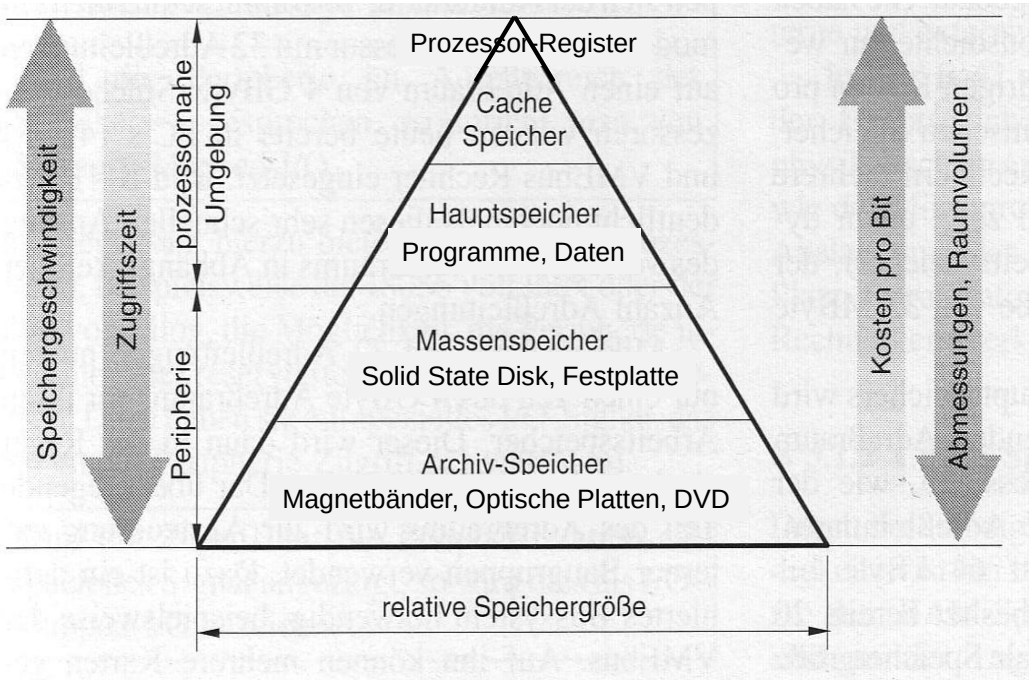
- Parallelisierung von Befehls- und Datenzugriffen im Hauptspeicher
- Harvard Architektur
- Überlappung von Befehlshol- und Ausführungsphase
  - Techniken zur Beschleunigung der Instruktionsausführung
  - Seriell
  - Seriell mit Überlappung Hol-/Ausführungsphase
  - Pipelining
  - Superpipelining
  - **Superskalar**

Vervielfältigung von Funktionseinheiten

- **Beschleunigung im Speicherzugriff durch Cache-Speicher**

Prozessor-Speicher-System

- Hierachische Speicherstruktur in einem Rechnersystem



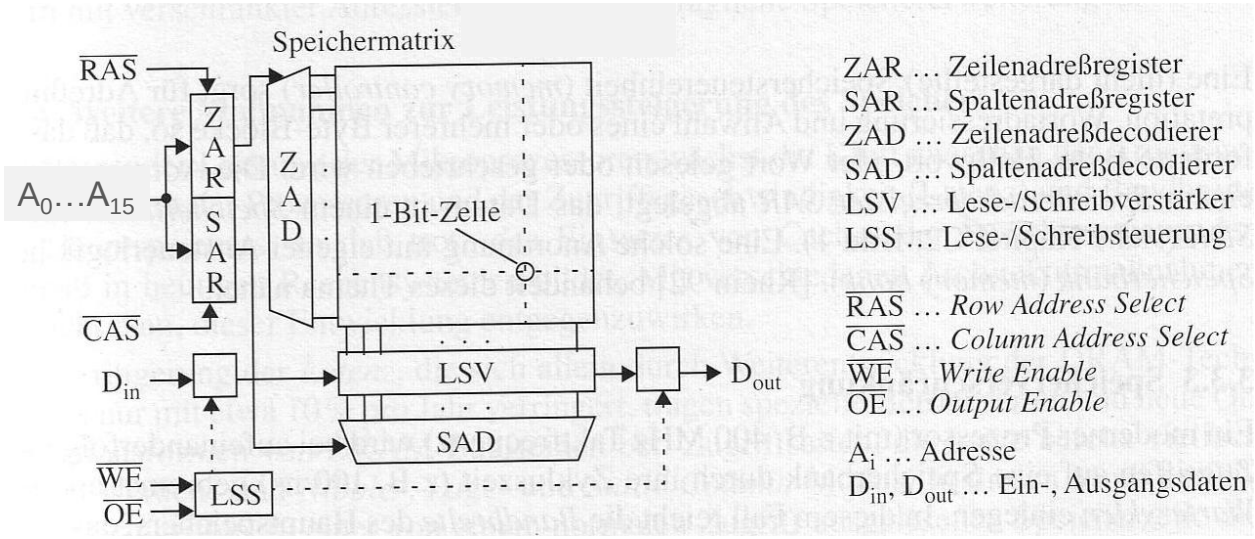
Hierarchie Prozessor-Speicher-System

Zugriffszeiten auf unterschiedliche Speicher

Speicherart	Zugriffszeit	relativ
Prozessor-Register	100 ps bis 1 ns	0,005
Prozessorcache L1	1 ns bis 5 ns	0,05
Prozessorcache L2	2 ns bis 20 ns	0,1
Prozessorcache L3	30 ns bis 50 ns	0,4
Arbeitsspeicher (DRAM, DDR-SDRAM)	50 ns bis 200 ns	1
Massenspeicher (Festplatte, Solid State Disk)	8 ms bis 10 ms (0,5 ms bis 1 ms)	50
Wechseldatenträger (Magnetband, DVD, USB-Stick)	Sekunden bis Minuten	>1000

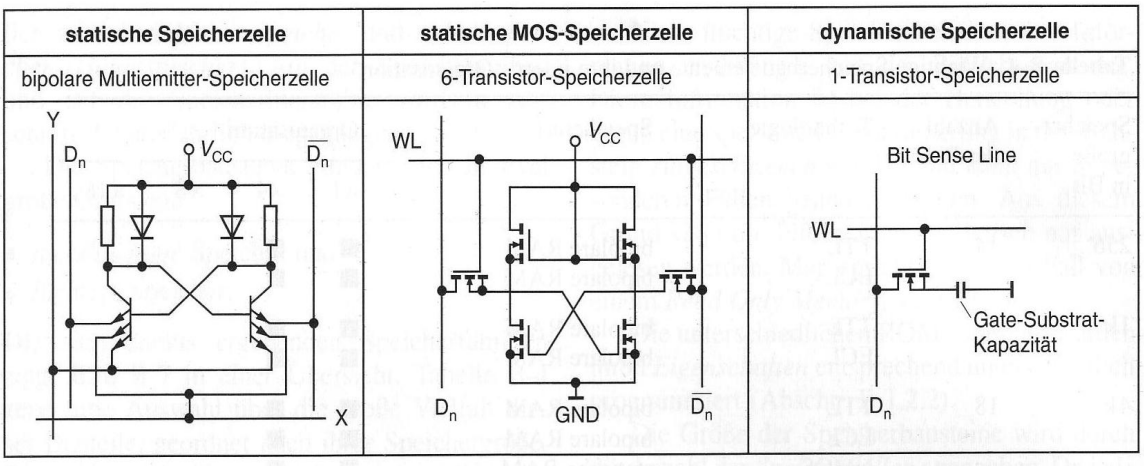
Prozessor-Speicher-System

- Schematische Darstellung eines DRAM-Bausteins (Dynamic RAM ohne Refresh-Logik)



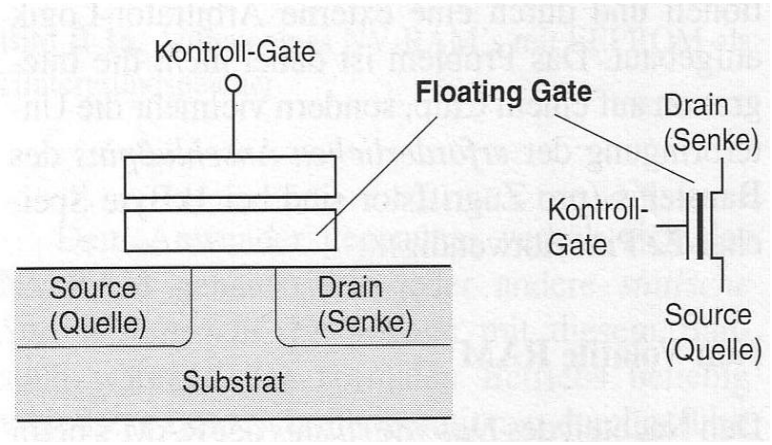
Prozessor-Speicher-System

- Grundelemente der wichtigsten Speicher



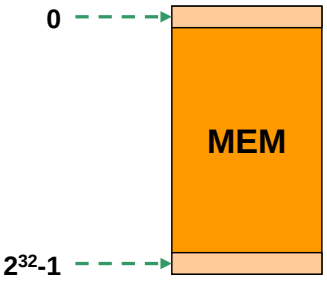
Prozessor-Speicher-System

- Aufbau und Symbol der Flash-EEPROM-Speicherzelle (Electrical Erasable Programmable Read Only Memory)



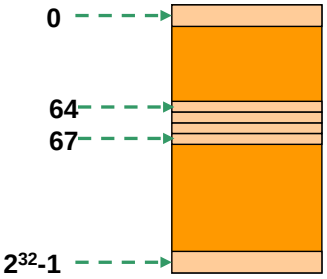
Speicherorganisation

- Üblich ist die Organisation in Bytes:
  z.B. 32bit Adresse $\Rightarrow 2^{32}$ Bytes adressierbar



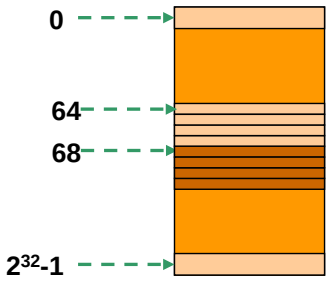
- Organisation z.B. in Bytes, Wörter (2 Bytes), Langwörter (4 Bytes), Quadwörter (8 Bytes),

- Ausrichtung (alignment)
  - Daten mit einer festen Länge können nur beginnend mit einer festen Adresse im Speicher untergebracht werden, z.B. Langwörter nur an Adressen, die ein vielfaches von 4 sind

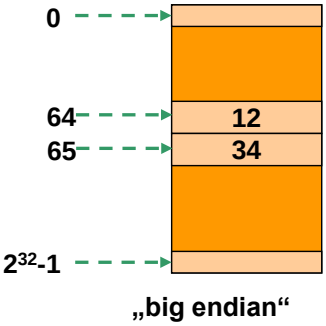


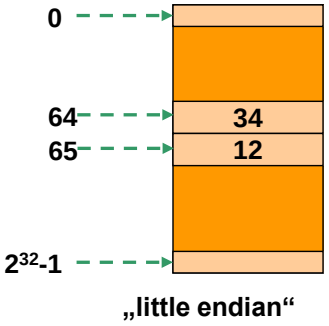
Speicherorganisation

- Besteht ein Befehl aus 4 Bytes, so muss die Byte-Adresse um 4 erhöht werden, um den nächsten Befehl zu holen



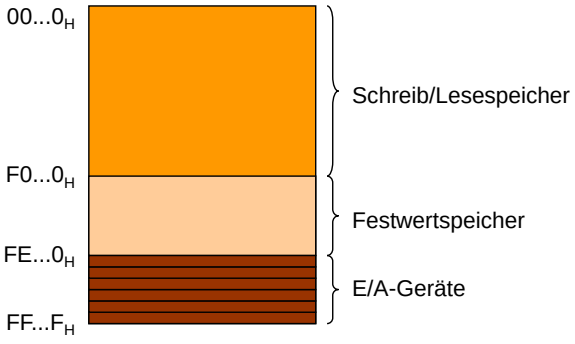
- Dabei ist die Interpretation z.B. als ganze Zahlen zu unterscheiden, ob das niedrigstwerte Byte an der höchsten Adresse („big endian“, z.B. Apple), oder niedrigsten Adresse („little endian“, z.B. Intel x86) gespeichert wird
- Beispiel Speicherung von 1234h in einem Wort:





Speicherorganisation

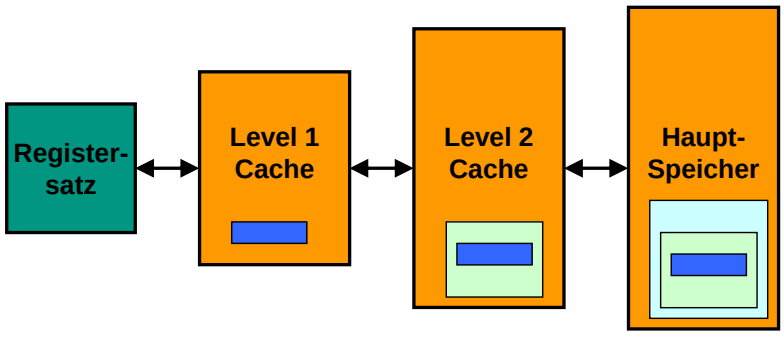
- Neben den Schreib/Lesespeichern existieren weitere Speichertypen
  - Festwertspeicher (ROM), behalten den Inhalt auch nach Abschalten der Spannungsversorgung
  - Speicherung von Konstanten oder grundsätzlichen Befehlsabläufen (z.B. Prozessorstart)
- Aufteilung des Speichers in mehrere Teilspeicher
  - Auswahl der Bereiches über einen Teil des Adressvektors
  - Der Rest des Adressvektors adressiert die Speicherzelle innerhalb des Bereiches
- Auch E/A Geräte können wie ein Speicher behandelt werden („memory mapped I/O“)
  - Auswahl der Geräte über die Speicheradresse
  - Festlegung der Richtung über die Schreib/Lese-Steuerung des Speichers



Memory Map

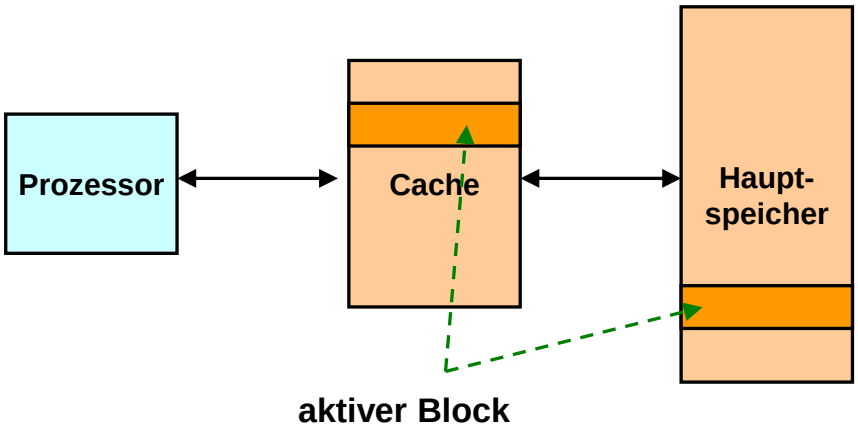
Speicherorganisation, Cache (Puffer) Speicher

- Speicherhierarchie
  - Prinzip der Lokalität:
    - Zeitlich: benutzte Daten/Befehle werden bald wieder benutzt
    - Räumlich: auf Daten/Befehle, die im Adressraum nahe zu gerade benutzten liegen, wird wahrscheinlicher verwiesen als auf weiter weg liegende
  - Räumlich nahe beieinander liegende und oft referenzierende Teile (Blöcke), werden möglichst nahe am Prozessor gehalten



Cacheorganisation

- Cacheorganisation
  - Prinzip: nur die tatsächlich gebrauchten Blöcke werden im Cache gehalten
  - Einfaches Beispiel: zweistufige Hierarchie Cache-Hauptspeicher

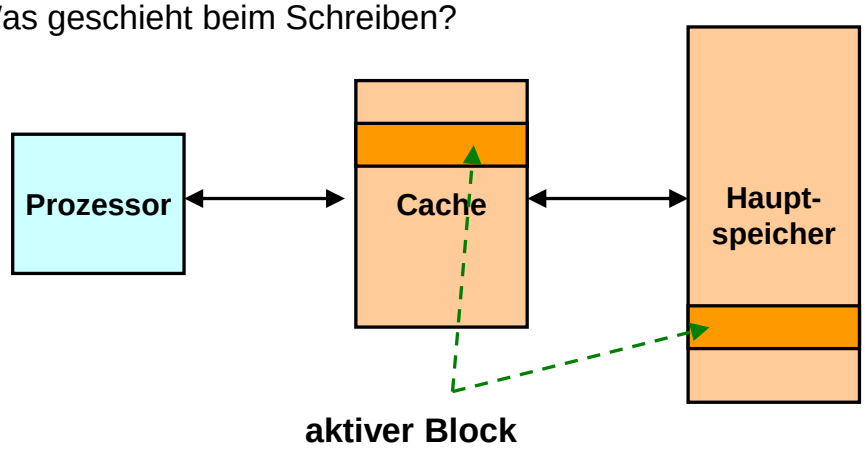


aktiver Block

Blockgröße z.B. 32 Bytes

Cacheorganisation

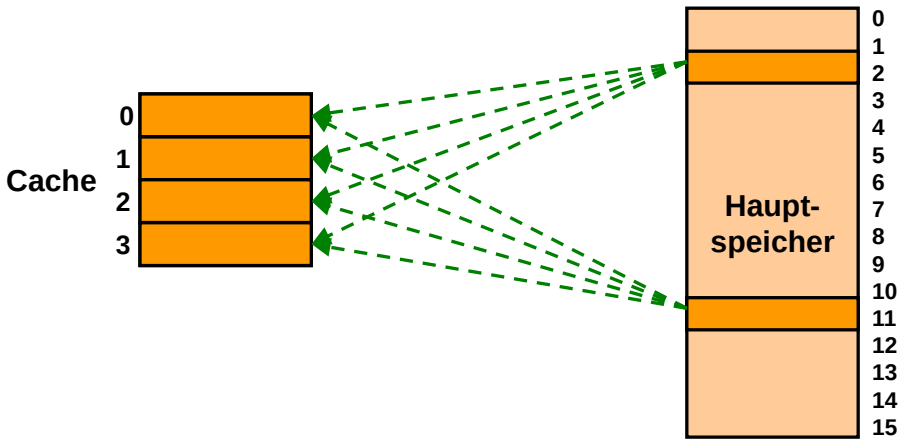
- Antworten auf vier Fragen entscheidend:
  - A: Wo kann ein Block im Cache platziert werden?
  - B: Wie kann festgestellt werden, ob ein Block im Cache ist?
  - C: Welcher Block soll ersetzt werden, falls ein Block gebraucht wird?
  - D: Was geschieht beim Schreiben?



aktiver Block

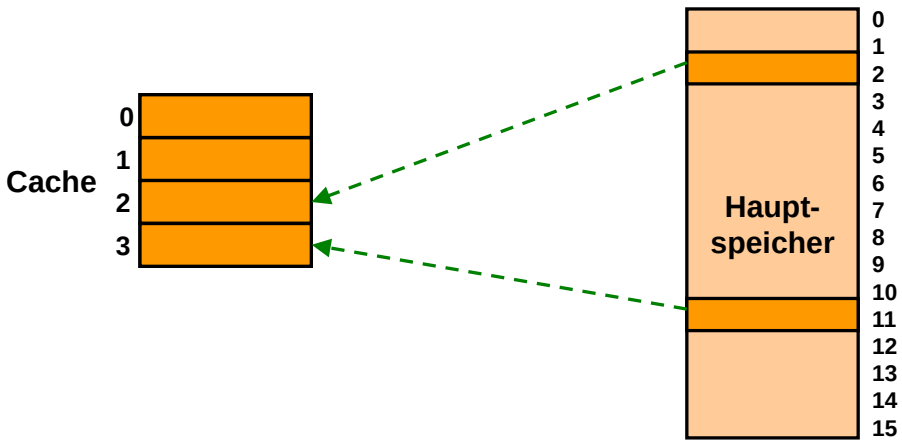
Cacheorganisation

- A: Wo kann ein Block im Cache platziert werden?
  - 1. Verfahren: voll-assoziativ
    - Jeder Block kann an allen Adressen gespeichert werden



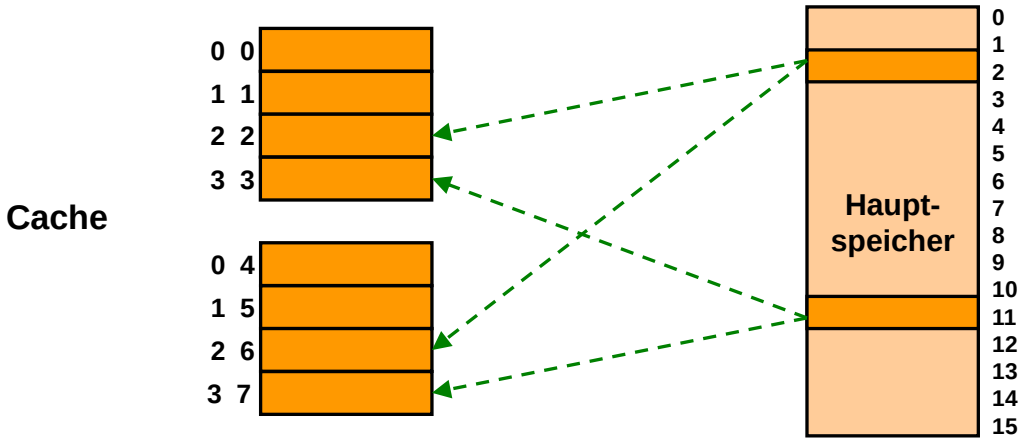
Cacheorganisation

- 2. Verfahren: direkt-abbildend
  - Ein Block kann nur an einer Adresse mod n (n Cache-Größe, hier: 4) gespeichert werden



Cacheorganisation

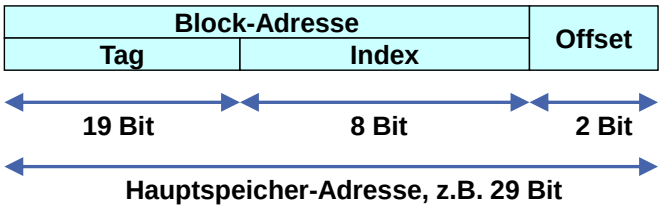
- 3. Verfahren: n-Wege assoziativ
  - Ein Block kann an jeder der n Cache-Adressen gespeichert werden, bei denen die Hauptspeicheradresse mod n gleich der Cache-Adresse mod n ist
  - Physikalischer Aufbau von m-unabhängigen Caches
  - Beispiel: 2-Wege assoziativ



Cacheorganisation

- B: Wie kann festgestellt werden, ob ein Block im Cache ist?

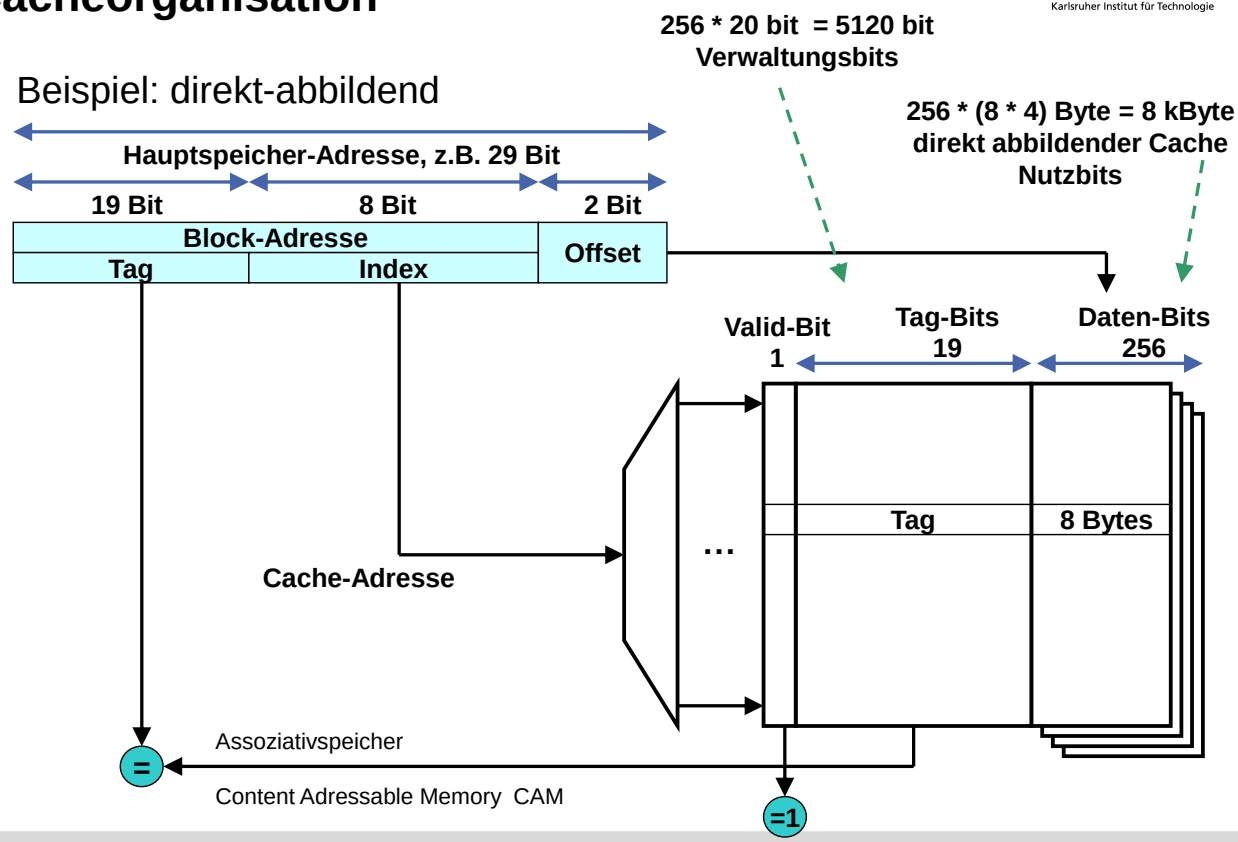
  Verfahren bei direkt abbildendem Cache
  - Annahme: pro Block im Cache werden k Worte gespeichert, z.B. 4 Worte zu 8 Byte bei 64Bit-Datenbusbreite vom Prozessor zum Cache und Hauptspeicher und schnelles Auslesen aus dem DRAM im Burstmode
  - Offset ist die Adresse innerhalb eines Blockes, z.B. 2 Bits bei 4 Worten pro Block
  - Index gibt die Adresse im Cache an (im Beispiel 256 Worte im Cache)



29 Bit Adresse für 64-Bit Worte
32 Bit Adresse auf Byteadressierungsebene
(Byte, 16-Bit-Word, 32-Bit Longword, 64-Bit Quadword)

Cacheorganisation

- Beispiel: direkt-abbildend



Cacheorganisation

- Beispiel: direkt-abbildend
  - Cache ist in Form einer Tabelle mit mehreren Datenworten pro Zeile (Block) organisiert
  - Hauptspeicheradresse von insgesamt 29 Bit Breite setzt sich zusammen aus:

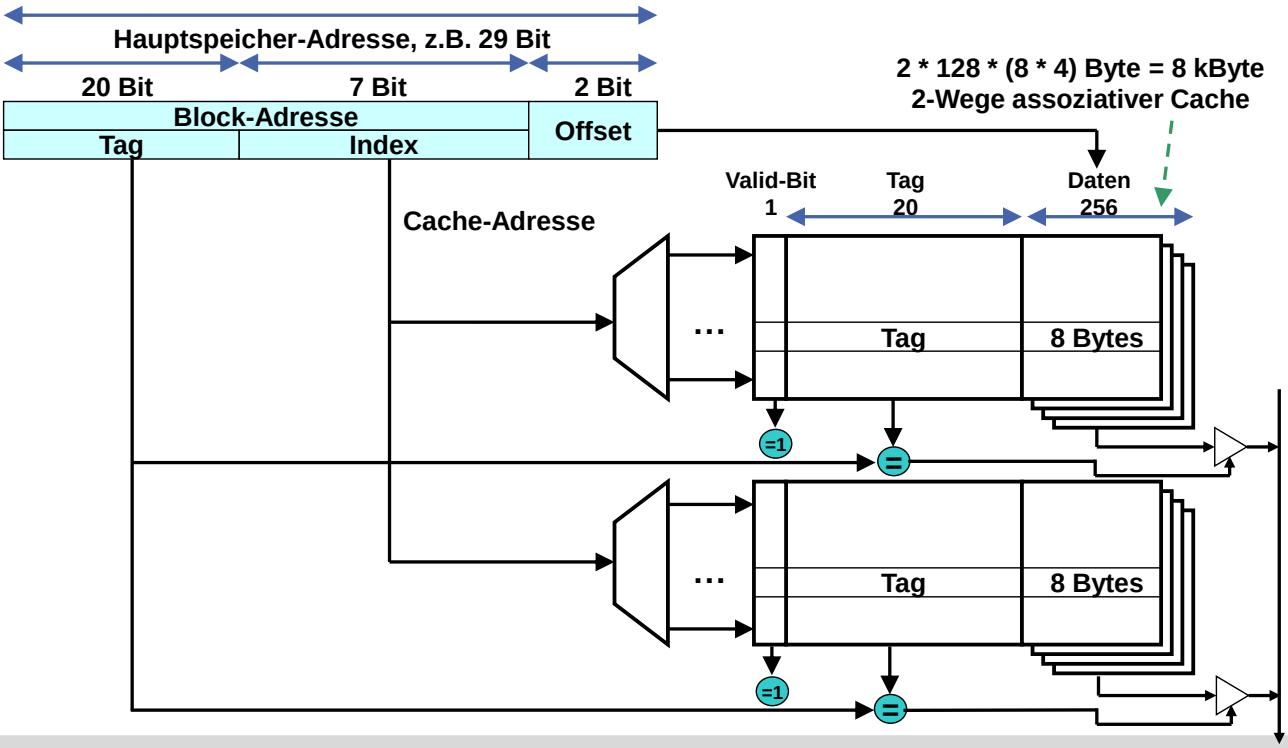
    Offset (2 Bits): adressiert ein 64-Bit Datenwort innerhalb eines Blockes, wobei jeder Block z.B. $2^2 = 4$ Datenworte enthalten kann.

    Index(8 Bit): adressiert einen Block innerhalb des Caches, die Gesamtzahl der Blöcke beträgt $2^8 = 256$.

    Tag (19 Bit): dient als Referenz im Cache, um die Gültigkeit des aktuellen Eintrages festzustellen, ist dies nicht der Fall, werden die alten Werte im Cache bei Bedarf in den Hauptspeicher übertragen und die aktuell referenzierten Daten aus dem Hauptspeicher in den Cache geschrieben.
  - Das Valid-Bit gibt an, ob in dem betreffenden Block die Daten gültig sind, d.h. ob der entsprechende Block in Benutzung ist, oder ob der Block frei ist und einfach überschrieben werden kann
  - Die gesamte Größe des Caches errechnet sich aus der Anzahl der Zeilen, Anzahl der Datenworte pro Zeile und der Größe der Datenworte: 256 * (4 * 8) = 8 Kbytes, hinzu kommen die Verwaltungsbits

Cacheorganisation

- Beispiel: 2-Wege assoziativ

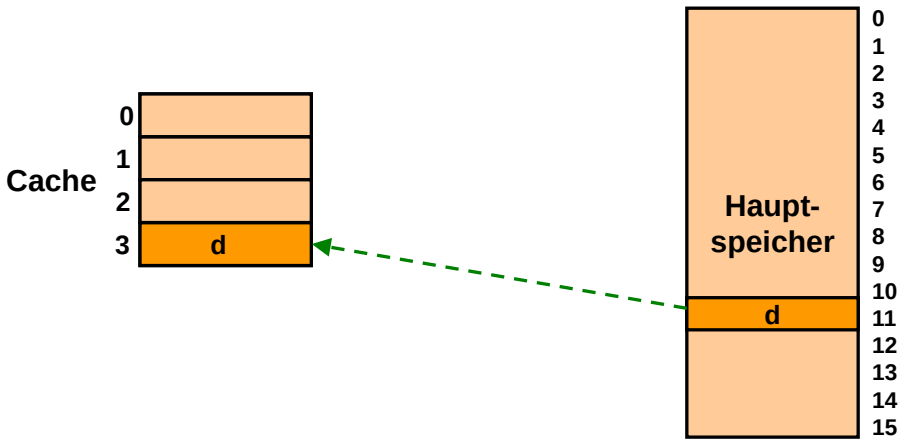


Cacheorganisation

- Beispiel: 2-Wege assoziativ
  - Das Prinzip ähnelt dem direkt-abbildendem Cache, jedoch enthält diese Organisationsform die zweifache Implementierung
  - Über einen Komparator des Tags wird entschieden, welches Datenwort auf den Datenbus des Prozessors gelegt wird
  - Vorteile:
    - Zwei Datenworte, die den selben Index jedoch unterschiedliche Tags in der Adresse enthalten, können gleichzeitig im Cache gepuffert werden
    - Führt je nach Programmbeschaffenheit zu einer Steigerung der Performanz
  - Nachteil:
    - Größerer Aufwand beim Aufbau des Caches / höhere Kosten
  - In modernen Prozessoren werden bis zu 8- bzw. 16-fach assoziative Caches eingesetzt

Cacheorganisation - Verdrängungsstrategie

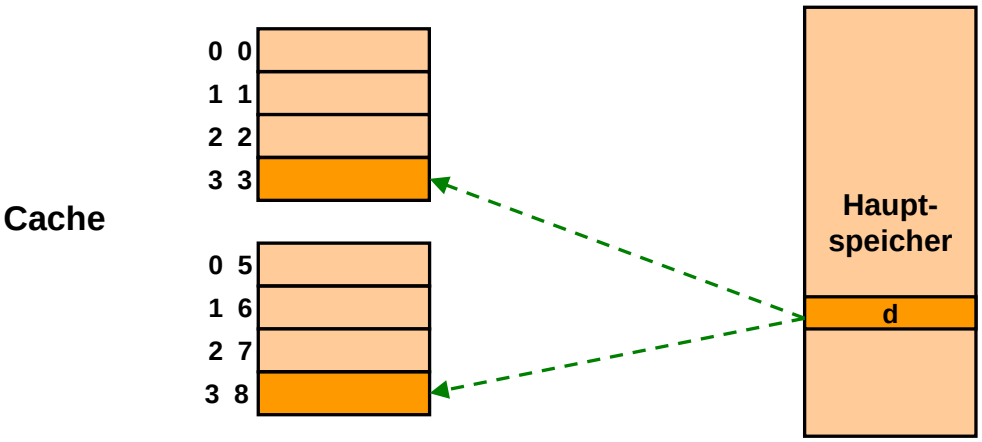
- C: Welcher Block soll ersetzt werden, falls ein Block gebraucht wird?
  - direkt-abbildend: klar



Cacheorganisation

- n-Wege assoziativ:
  - LRU (least recently used): Versuch die zuletzt benutzten Daten zu erhalten, z.B. PowerPC 603 LRU 4-Wege teilassoziativer Cache
  - Zufallsgesteuert

    Experimentell: LRU nur bei kleinen Caches (16kB) geringfügig (<10%) besser



Verdrängungsstrategien

- LRU-Strategie (Least Recently Used)
  Der Eintrag, auf den am längsten nicht zugegriffen wurde, wird verdrängt
- LFR-Strategie (Least Frequently Used)
  Der am seltensten gelesene Eintrag wird verdrängt
- FIFO-Strategie (First In First Out)
  Der jeweils älteste Eintrag wird verdrängt
- Zufallsstrategie
  geringster Hardwareaufwand

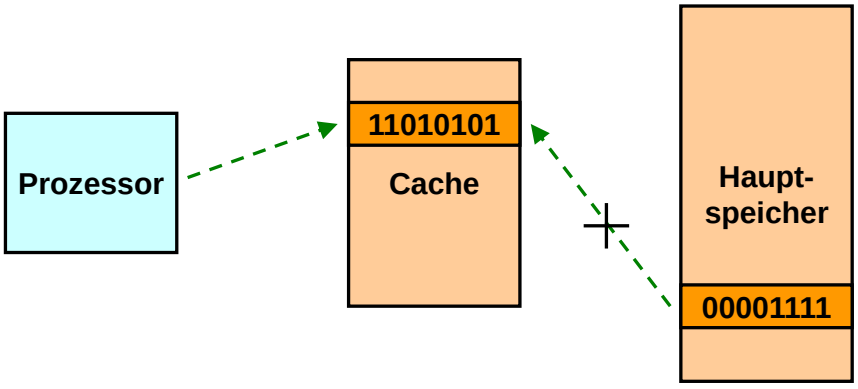
Prozentzahlen Fehlzugriffe (Hennessy, Patterson 1996)

Assoziativität	2-way		4-way		8-way	
Größe	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

$$t_{average-access} = hit - rate * t_{hit} + (1 - hit\text{-}rate) * t_{miss}$$

Cacheorganisation

- D: Was geschieht beim Schreiben?
  - Daten in Cache und Hauptspeicher korrespondieren einander nur solange der Prozessor nicht die Daten im Cache überschreibt
  - 2 Strategien:
    - Schreiben in Cache und Speicher (write through)
    - Schreiben nur in Cache (setzen des dirty bits als weiteres Statusbit), zurück schreiben in Speicher nur bei Austausch des Blocks (write-back)



Aufgabe 8.03: Cacheorganisation

Angenommen, dass ein Prozessor einen Hauptspeicher der Größe **1 GByte** hat, wo jedem einzelnen Byte im Speicher eine **30 Bit** lange Adresse zugeordnet wird. Zusätzlich verfügt der Prozessor über einen Cache Baustein der Technologie SRAM, der **512 KBytes** (nur Daten) vom Hauptspeicher aufnehmen kann (unabhängig vom notwendigen Speicher zur Speicherung der weiteren Bits, wie z.B. Tag). Der Cache hat eine Blockgröße von **8 Bytes**. Zusätzlich wird auf dem Cache für jeden Block **ein Valid-Bit** vorgesehen.

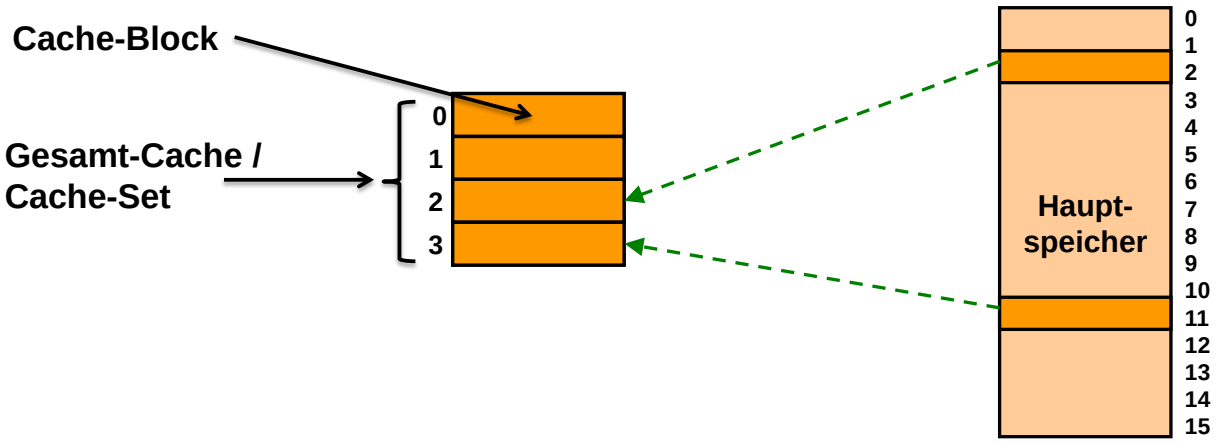
a) Nennen Sie die drei Möglichkeiten der Cacheorganisation.
b) Berechnen Sie für jeden Typ die Anzahl der Cache-Blöcke und der Cache-Sets. Hinweis: Bei dem einen Typ, wo eine Variable n festgelegt werden soll, nehmen Sie den Wert 4.
c) Erläutern Sie zu jeden Typ die Speicherorganisation des Caches bezüglich im Cache abzulegenden Daten und Adressierung.
d) Berechnen Sie für jeden Typ die in Wirklichkeit notwendige Speicherkapazität für den Cache Baustein.

Aufgabe 8.03: Cacheorganisation

- Die drei Möglichkeiten der Cacheorganisation:
  - Direkt-Abbildend (direct mapped: DM)
  - n- Weg assoziativ (set associative: SA)
  - Voll-assoziativ (fully associative: FA)

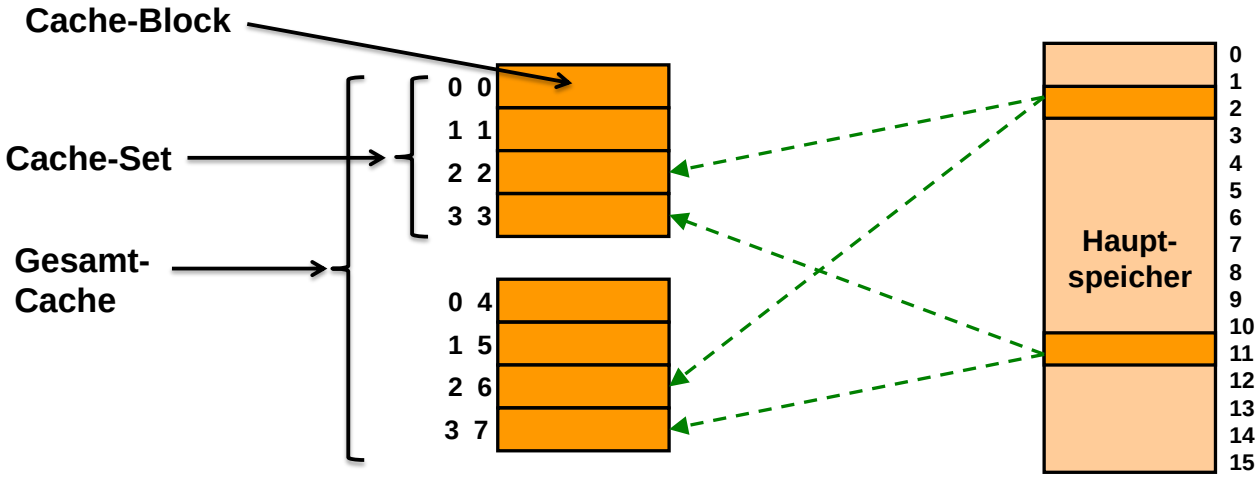
Aufgabe 8.03: direkt-abbildend

- Ein Block kann nur an einer Adresse mod m (m Cache-Größe, hier: 4) gespeichert werden



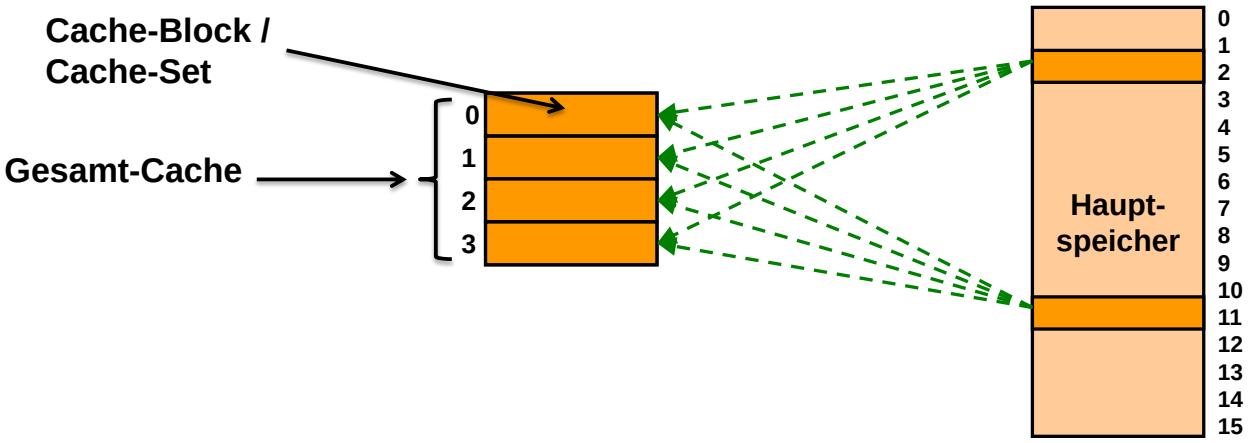
Aufgabe 8.03: n-Weg assoziativ

- Ein Block kann an jeder der m Cache-Adressen gespeichert werden, bei denen die Hauptspeicheradresse mod m gleich der Cache-Adresse mod m ist (m Cache-Set-Größe, hier: 4)
- Physikalischer Aufbau von n-unabhängigen Caches
- Direkt-Abbildend ist n- Weg assoziativ mit n = 1
- Beispiel: 2-Weg assoziativ



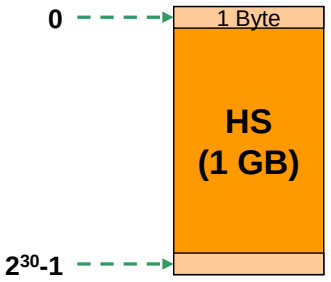
Aufgabe 8.03: voll-assoziativ

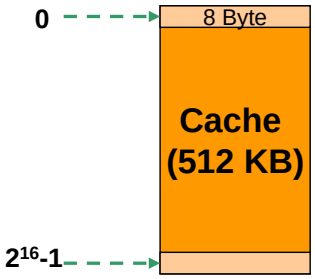
- Jeder Block kann an allen Adressen gespeichert werden
- Voll-assoziativ ist n- Weg assoziativ mit n = m = Anzahl der Cache-Blöcke



Aufgabe 8.03: Cache Dimensionierung

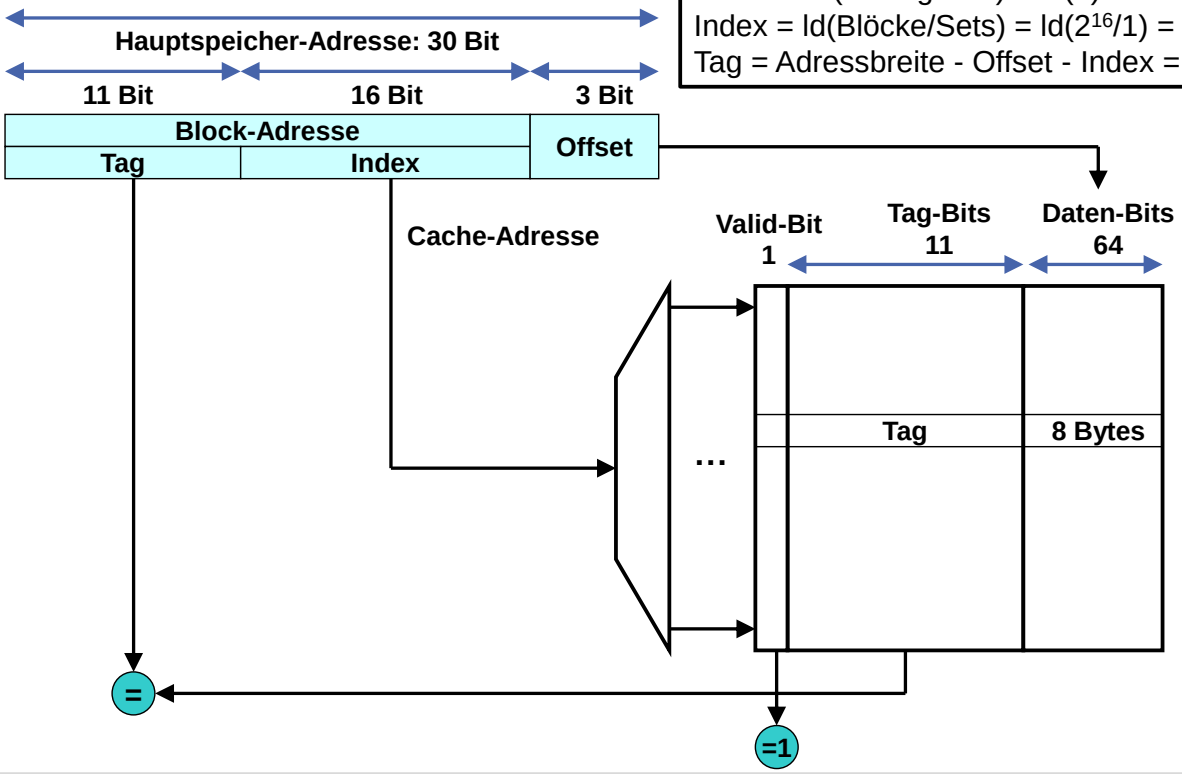
- Direkt-Abbildend (direct mapped: DM)
  - Anzahl Cache-Blöcke = 512 KB / 8 B = 65536 = 2^{16}
  - Anzahl Cache-Sets = n = 1
- 4- Weg assoziativ (set associative: SA)
  - Anzahl Cache-Blöcke = 512 KB / 8 B = 65536 = 2^{16}
  - Anzahl Cache-Sets = n = 4
- Voll-assoziativ (fully associative: FA)
  - Anzahl Cache-Blöcke = 512 KB / 8 B = 65536 = 2^{16}
  - Anzahl Cache-Sets = n = 65536 = 2^{16}





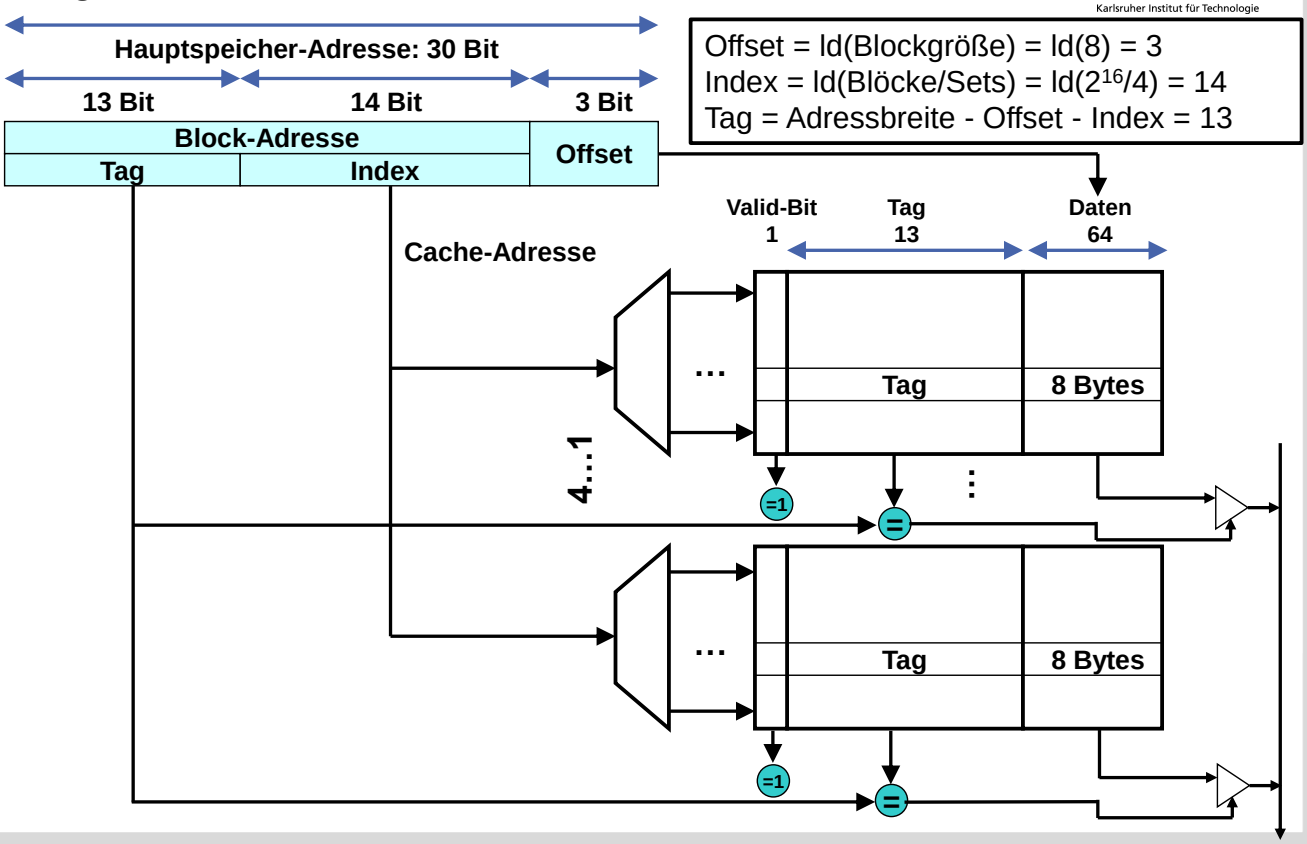
Aufgabe 8.03: Speicherorganisation für den Cachetyp Direkt-Abbildend

Offset = ld(Blockgröße) = ld(8) = 3
Index = ld(Blöcke/Sets) = ld(2^{16}/1) = 16
Tag = Adressbreite - Offset - Index = 11

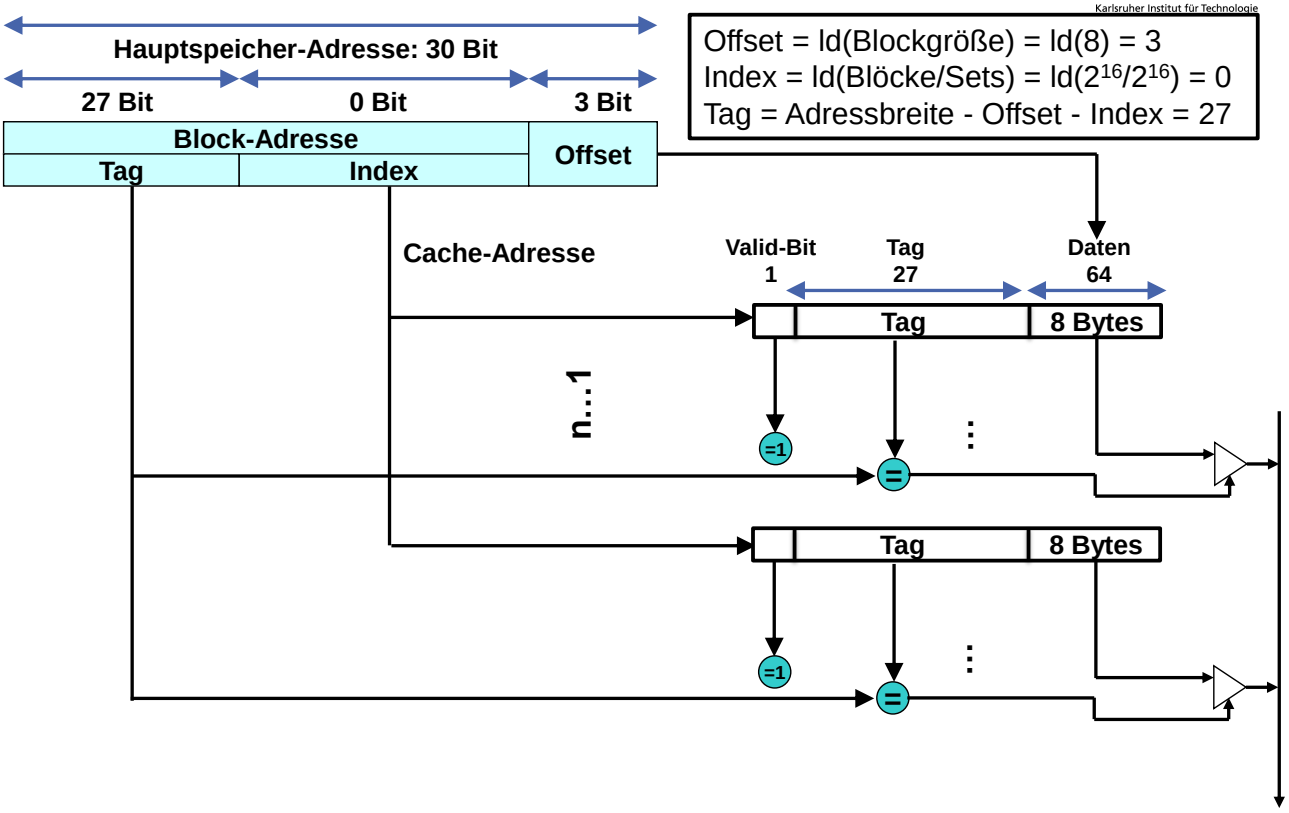


Aufgabe 8.03: Speicherorganisation für den Cachetyp 4-Weg assoziativ

Offset = ld(Blockgröße) = ld(8) = 3
Index = ld(Blöcke/Sets) = ld(2^{16}/4) = 14
Tag = Adressbreite - Offset - Index = 13



Aufgabe 8.03: Speicherorganisation für den Cachetyp voll-assoziativ



Aufgabe 8.03: notwendige Speicherkapazität für den Cache

- Speichergröße = Datenspeicher + Speicher für Tags + Speicher für Valid-Bits
- Direkt-Abbildend (direct mapped: DM)
  - Speichergröße = 512 KB + (12 Bit * 2^{16}) / (8 Bit/Byte) = 512 KB + 96 KB = 608 KB
- 4- Weg assoziativ (set associative: SA)
  - Speichergröße = 512 KB + (14 Bit * 2^{16}) / (8 Bit/Byte) = 512 KB + 112 KB = 624 KB
- Voll-assoziativ (fully associative: FA)
  - Speichergröße = 512 KB + (28 Bit * 2^{16}) / (8 Bit/Byte) = 512 KB + 224 KB = 736 KB
- Der benötigte Speicher ist offensichtlich nicht der ausschlaggebende Faktor bei der Auswahl! Was sonst?
- Trade-off: Anzahl der benötigten Komparatoren (Logik-Platz)

  vs.

  Höhere Flexibilität beim Mapping (== höhere Cache-Hit-Rate)