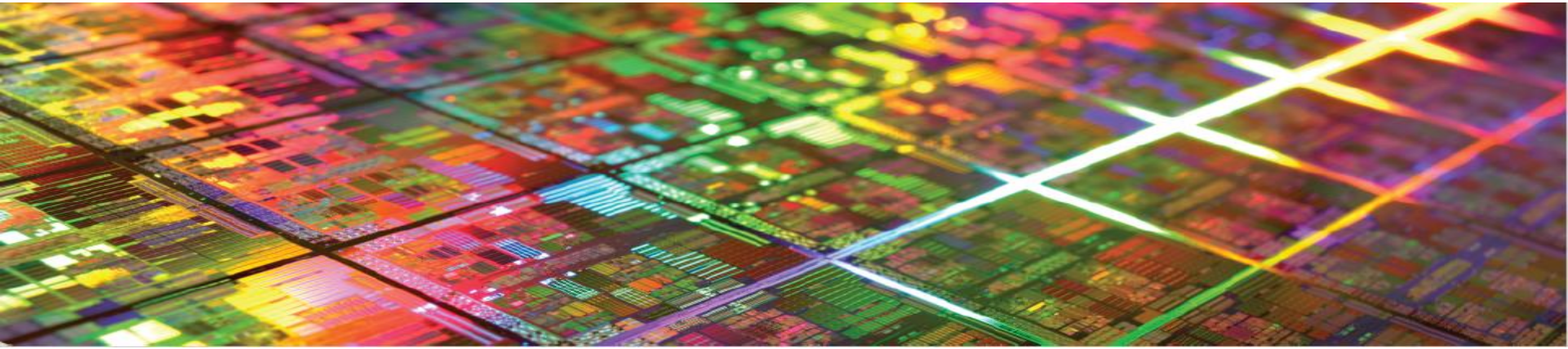
Karlsruher Institut für Technologie

Rechnerorganisation

Prof. Dr. Wolfgang Karl, Roman Lehmann

Vorlesung im Wintersemester 2025/2026 – Foliensatz: RO25-FS05



Kapitel 4
Befehlssatzarchitektur

- Grundlagen
- Ausführungsmodelle
- Datentypen, Datenformate
- Adressierungsarten
- Befehlsformat
- RISC-V Assembler-Programmierung
- Fallstudien (IA-32, RISC-V)
- Diskussion: RISC & CISC

Befehlssatzarchitektur

- **Entwurf des Befehlsformats**
  - Ausführungsmodelle
    - Komponenten, in denen die Operanden und das Ergebnis (Ziel) einer Operation stehen können
      - Hauptspeicher, Register, Akkumulator, Keller (Stack), …
    - Explizite oder implizite Angabe der Adressinformationen im Befehl
    - Überdeckte Adressierung
  - Datentypen und Datenformate
    - Information über die Interpretation der Operanden und des Ergebnisses
  - **Wie werden die Adressen der Speicheroperanden spezifiziert?**
    - ➢ **Adressierungsarten**

Befehlssatzarchitektur

- **Adressierungsarten**
  - **Spezifikation der Adresse eines Operanden**
    - **Adressmodifikationen (Adressrechnung)**
      - Berechnung der effektiven Adresse aus mehreren Teilen, die im Befehlswort, in Registern oder in Speicherzellen stehen
    - **Effektive Adresse (EA)**
      - Tatsächliche Adresse des Orts, in dem der referenzierte Operand steht
      - Wird während der Abarbeitung des Befehls im Prozessor generiert (Adressrechnung zur Laufzeit)

Befehlssatzarchitektur

- **Adressierungsarten**
  - **Dynamische Adressrechnung: Gründe**
    - Die Adresse einer Datenstruktur setzt sich z.B. additiv aus der Anfangsadresse der Datenstruktur und der Distanz innerhalb der Datenstruktur zusammen, wobei die Distanz erst zur Laufzeit bekannt ist.
    - Zerlegen der Adressen in Basisadresse und Distanzen erleichtert die Erzeugung von verschiebbaren Variablenbereichen und verschiebbaren Code.
    - Die Adresse eines Operanden ergibt sich häufig erst zur Laufzeit aus Berechnungen durch das Programm.

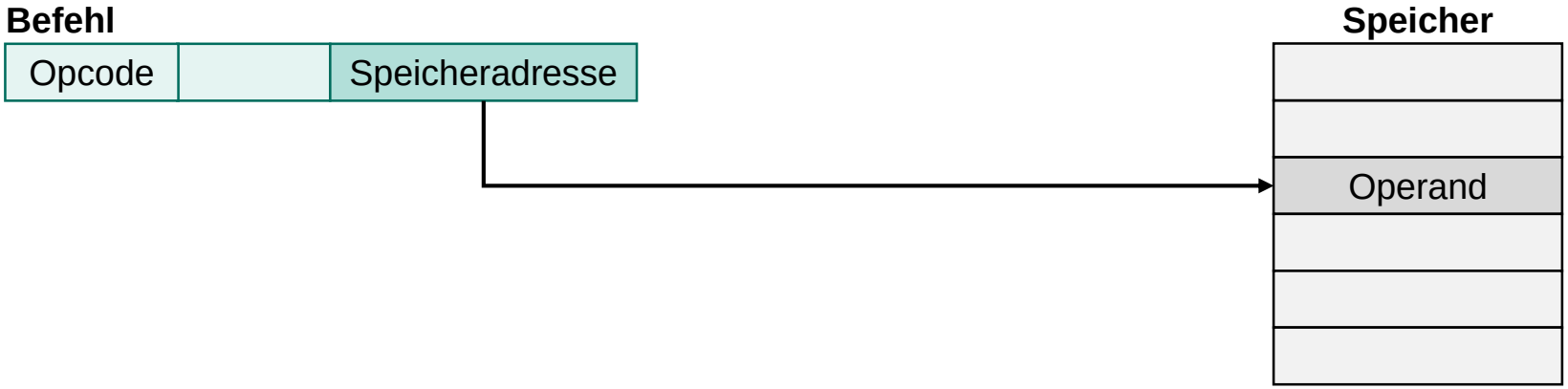
Adressierungsarten

- **Direktoperand (immediate)**
  - Der Operand steht als Konstante im Befehl
  - Die eigentliche Adressierung entfällt

Befehl

Adressierungsarten

- **Direkte Adressierung (direct addressing)**
  - Die Speicheradresse des Operanden steht im Befehl

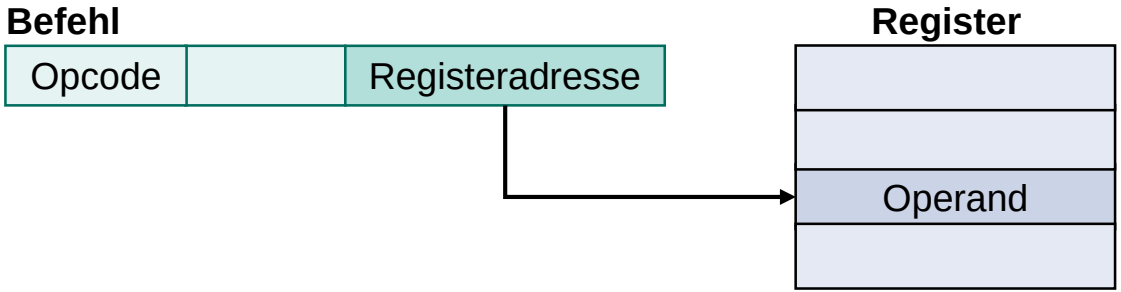


Adressierungsarten

- **Indirekte Adressierung (indirect addressing)**
  - Die Adresse im Befehl zeigt auf ein Wort im Speicher, das die Adresse des Operanden enthält.

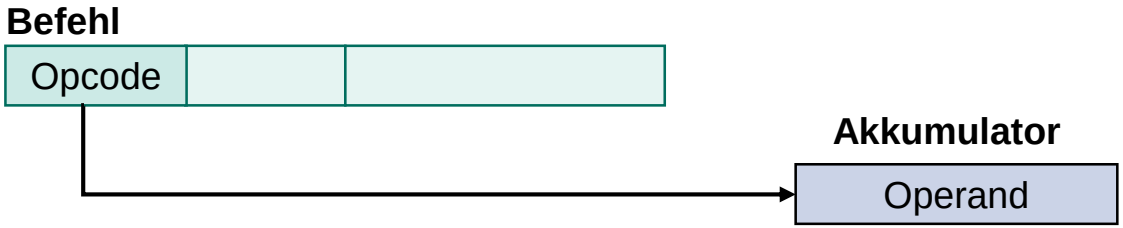
Adressierungsarten

- **Registeradressierung (register addressing)**
  - Die Registeradresse steht im Befehl, der Operand steht im referenzierten Register.



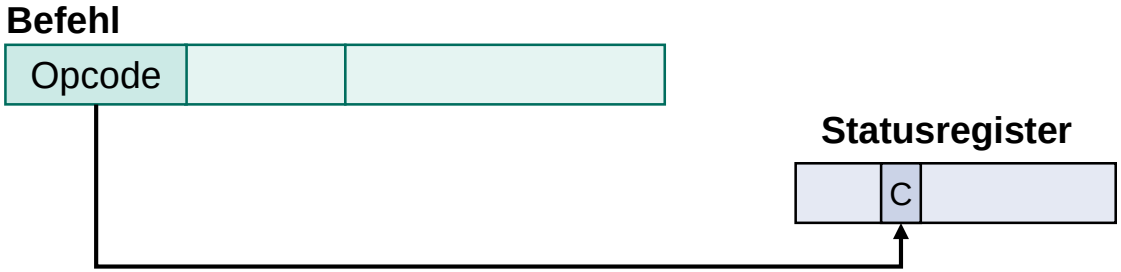
Adressierungsarten

- **Registeradressierung (register addressing)**
  - **Implizite Registeradressierung**
    - Die Nummer des Registers ist im Opcode kodiert.
    - Beispiel: Akkumulator



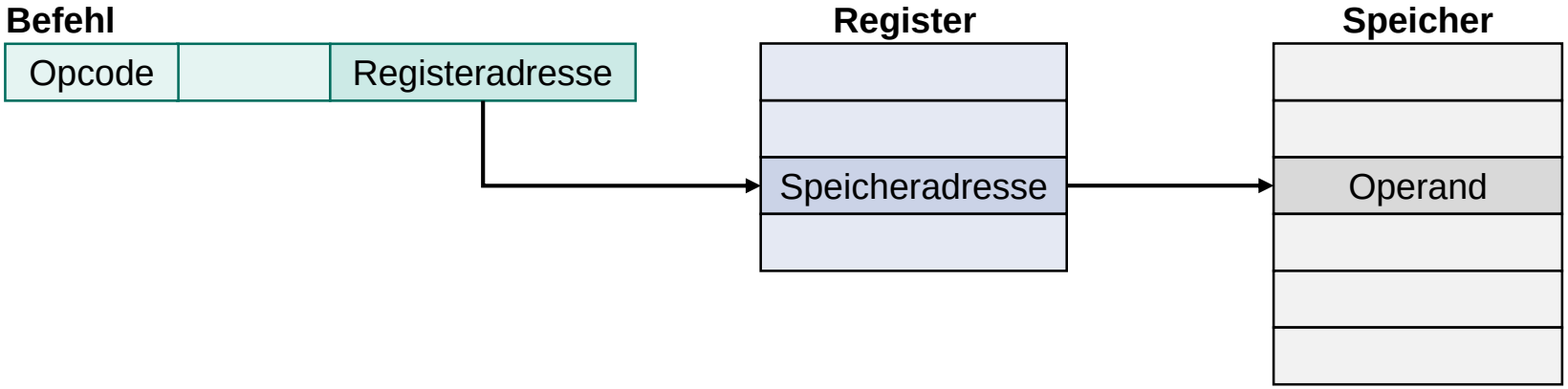
Adressierungsarten

- **Registeradressierung (register addressing)**
  - **Flag-Adressierung**
    - Die Nummer des Registers ist im Opcode kodiert.
    - Es wird ein einzelnes Bit (Flag) in einem Register referenziert.
    - Beispiel: Statusregister



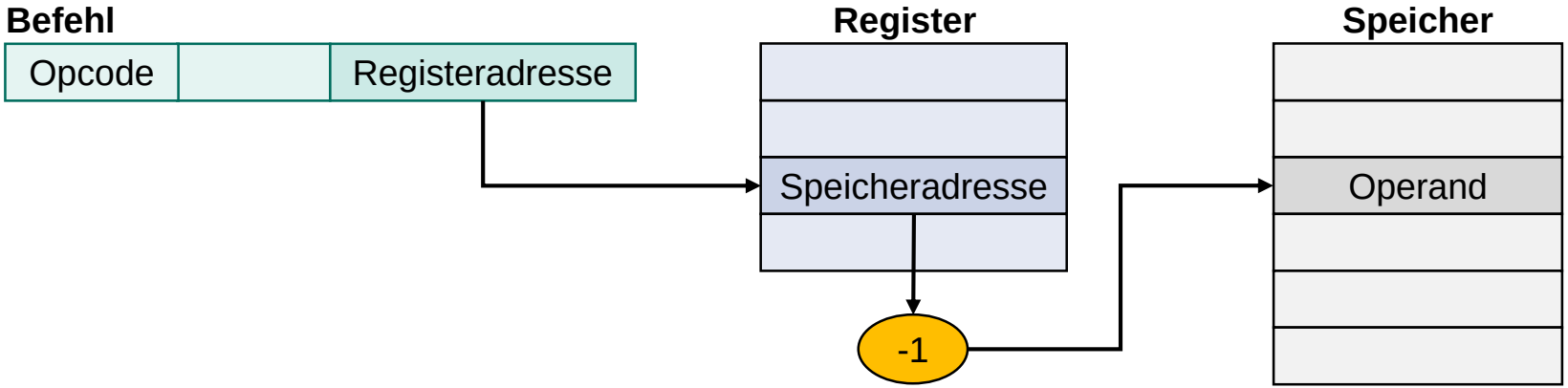
Adressierungsarten

- **Registerindirekte Adressierung (register indirect addressing)**
  - Das im Befehl referenzierte Register enthält die Speicheradresse des Operanden.
  - Adressierung über einen Zeiger;



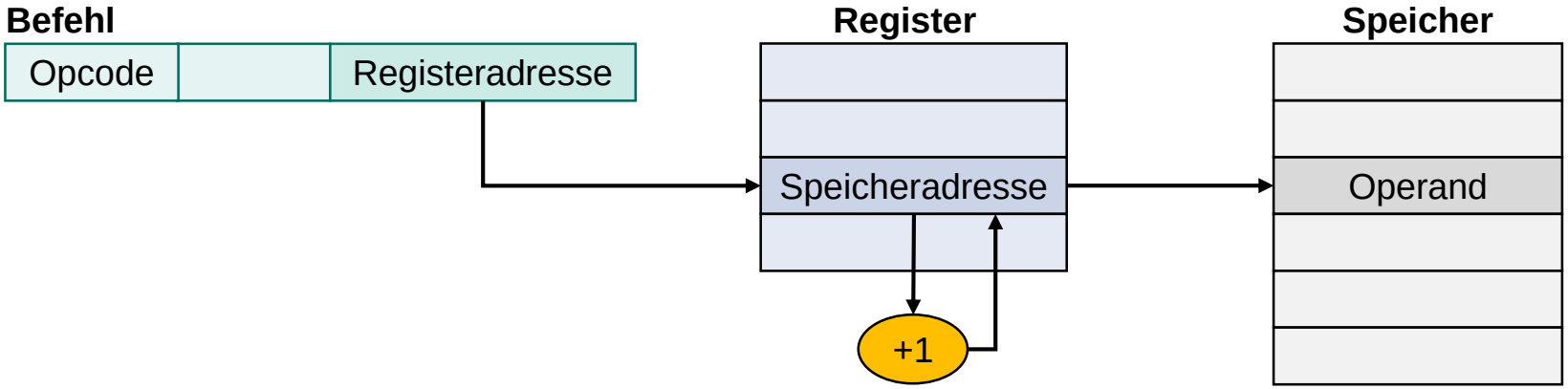
Adressierungsarten

- **Registerindirekte Adressierung (register indirect addressing)**
  - **Mit Prädekrement**
    - Der Inhalt des im Befehl referenzierten Registers wird **vor** dem Speicherzugriff um 1, 2, oder 4 dekrementiert (abhängig vom Datenformat).
    - Beispiel: Modifikation des Kellerzeigers



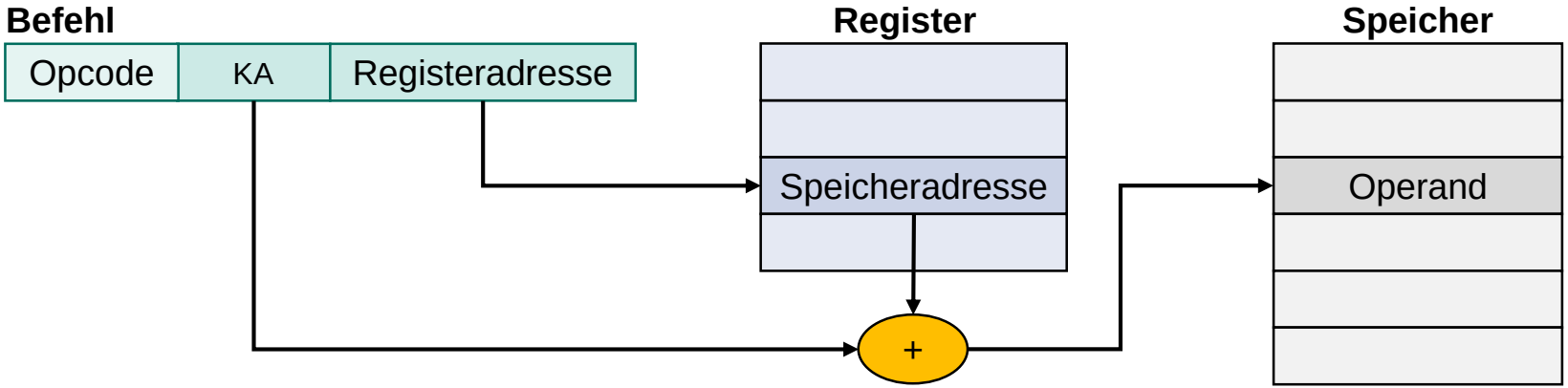
Adressierungsarten

- **Registerindirekte Adressierung (register indirect addressing)**
  - **Mit Postinkrement**
    - Der Inhalt des im Befehl referenzierten Registers wird **nach** dem Speicherzugriff um 1, 2, oder 4 inkrementiert (abhängig vom Datenformat).
    - Beispiel: Modifikation des Kellerzeigers



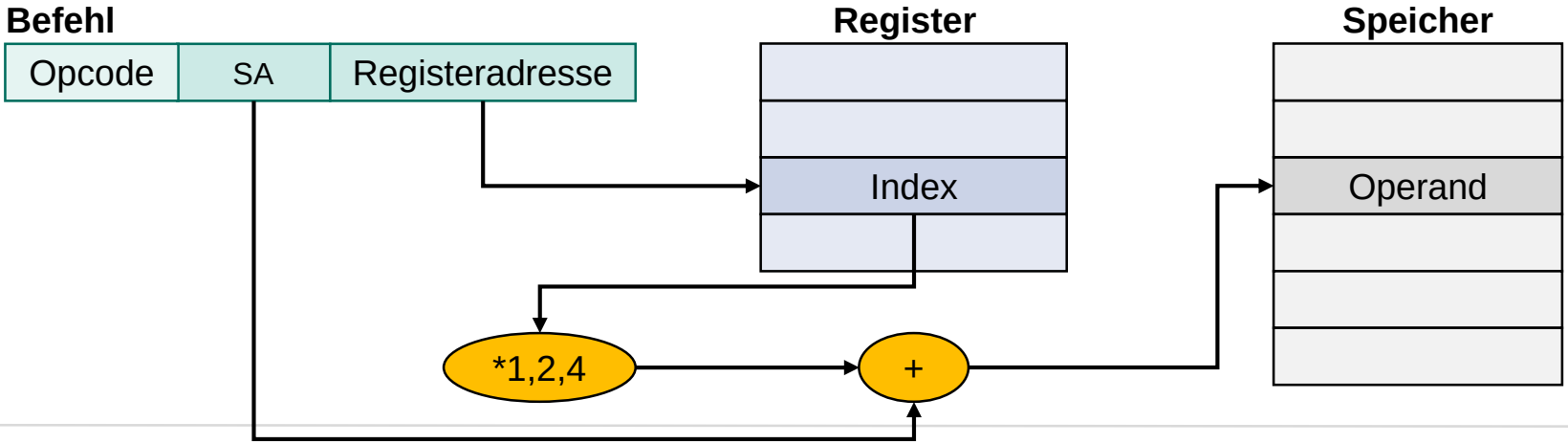
Adressierungsarten

- **Registerindirekte Adressierung mit konstanter Abstandsgröße (register indirect addressing with displacement), basisrelative Adressierung**
  - Auf den Inhalt des im Befehl referenzierten Registers (Basisregister) wird eine konstante Abstandsgröße (KA, Displacement, Offset) als 2‘er-Komplementzahl addiert.



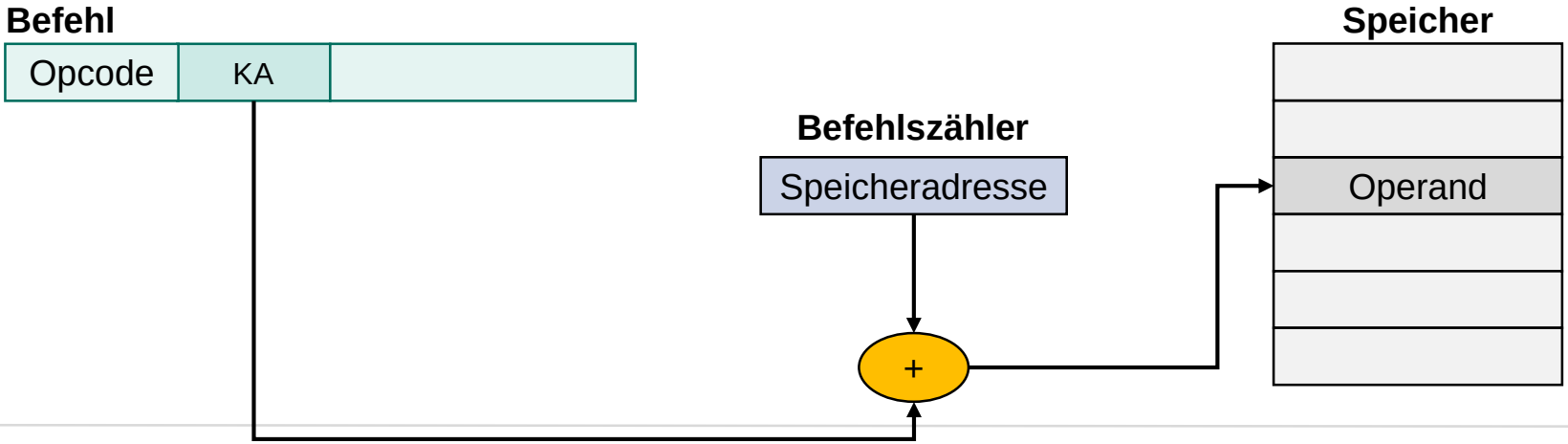
Adressierungsarten

- **Indizierte Adressierung (mit Skalierungsfaktor)**
  - Auf die im Befehl stehende Speicheradresse (SA) wird eine variable Abstandsgröße (Index) als 2‘er-Komplementzahl addiert.
    - Der Index kann mit einem Skalierungsfaktor multipliziert werden.
    - Die Speicheradresse kann auch in einem Basisregister stehen.



Adressierungsarten

- **Befehlszähler-relative Adressierung (program counter relative addressing)**
  - Auf die im Befehlszähler stehende Speicheradresse wird eine konstante Abstandsgröße als 2‘er-Komplementzahl addiert.



Adressierungsarten

- **Fallstudie IA-32**
  - **Registermodell**
    - **Allgemeine Register (General Purpose Registers)**

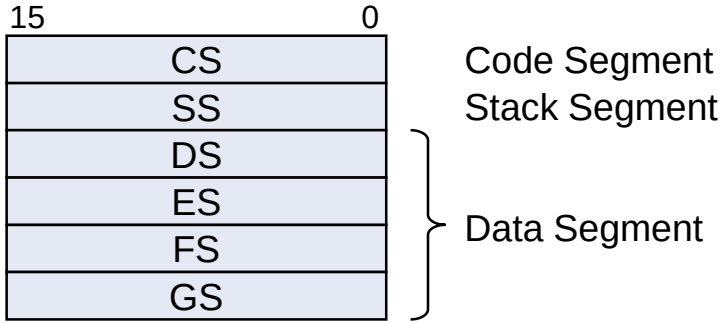
	31	15 0
EAX	Arithm. Ergebnisse	AX
EDX	Ein-/Ausgabe	DX
ECX	Zählerregister	CX
EBX	Basisregister	BX
EBP	Stack-Pointer	BP
ESI	Index-Register	SI
EDI	Index-Register	DX
ESP	Stack-Pointer	SP

    - **Spezialregister**

EIP	Instruction Pointer

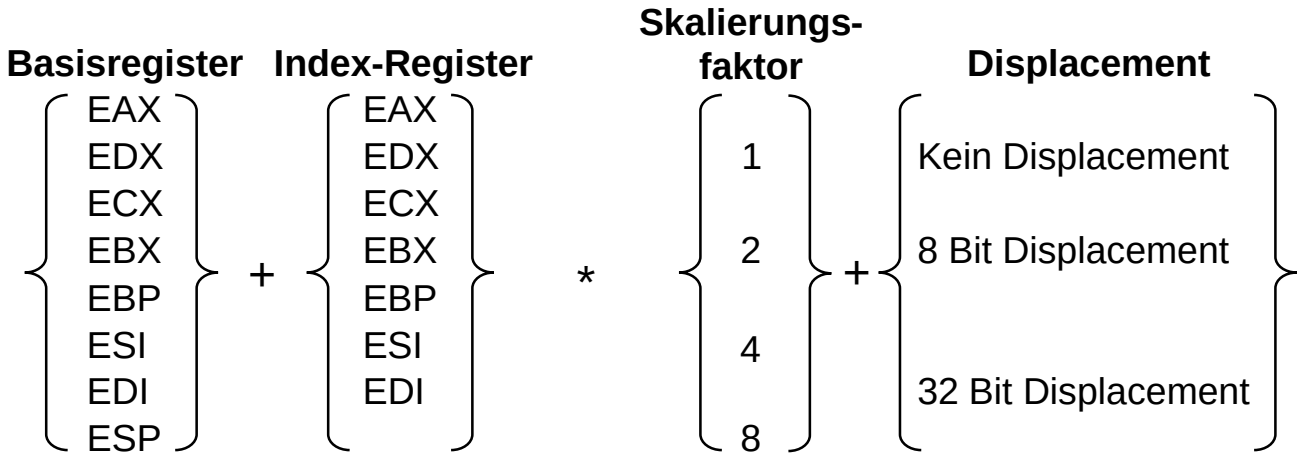
Adressierungsarten

- **Fallstudie IA-32**
  - **Registermodell**
    - **Segmentregister**



Adressierungsarten

- **Fallstudie IA-32**
  - Immediate Operands: Direktoperanden-Adressierung
  - Register Operands: Registeradressierung
  - Memory Operands: Spezifikation der Adresse



Adressierungsarten

- **Fallstudie IA-32 (mit Beispielen in Assembler-Code)**
  - **Immediate Operands**
    - Der Operand steht als Konstante im Befehl
    - `movl $0, %ecx   # Lade die Konstante 0 in das Register ECX`
  - **Register addressing (Registeradressierung)**
    - Der Operand steht in einem Register
    - `movl %edx, %ecx # Kopiere den Wert im Register EDX nach ECX`

Adressierungsarten

- **Fallstudie IA-32 (mit Beispielen in Assembler-Code)**
  - **Direct Mode (direkte Adressierung)**
    - Die Adresse des Operanden steht im Befehl (8-Bit oder 32-Bit Displacement) und der Operand steht im Speicher;
    - ```
      movl 2000, %ecx # Lade den Wert an der Speicheradresse 2000
                      # in das Register ECX
      ```
    - ```
      movl var, %ecx  # Lade den Wert an der Speicheradresse var
                      # in das Register ECX
      ```
  - **Register Indirect Mode (registerindirekte Adressierung)**
    - Die Adressierung erfolgt indirekt über ein Register, die Adresse des Operanden steht im Basisregister;
    - ```
      movl (%eax), %ecx     # Im Register EAX steht die Speicheradresse
                            # und der Wert wird in das Register ECX
                            # geladen
      ```
      ```

Adressierungsarten

- **Fallstudie IA-32 (mit Beispielen in Assembler-Code)**
  - **Based Mode (Basisrelative Adressierung): registerindirekte Adressierung mit Displacement**
    - Zu der im Basisregister stehenden Adresse wird eine konstante Abstandsgröße addiert;
    - `movl 8(%eax), %ecx`

```
# Auf die Basisadresse im Register EAX wird
# die konstante Abstandsgröße 8 addiert. Der
# Wert an dieser Speicheradresse wird in das
# Register ECX geladen
```

Adressierungsarten

- **Fallstudie IA-32**
  - **Index Mode (indizierte Adressierung)**
    - Zu einer Speicheradresse, die als Displacement im Befehl steht, wird eine variable Abstandsgröße, der Index, addiert, wobei der Index im Indexregister steht;
  - **Scaled Index Mode (indizierte Adressierung mit Skalierungsfaktor)**
    - Der Index im Indexregister wird mit einem Skalierungsfaktor (1,2,4 oder 8) multipliziert, um dann zu einer Speicheradresse, die als Displacement im Befehl steht, addiert zu werden;
  - **Based Index Mode (registerindirekte, indizierte Adressierung)**
    - Der Inhalt des Basisregisters wird zu einem Index im Indexregister addiert;

Adressierungsarten

- **Fallstudie IA-32**
  - **Based Index Mode with Displacement**
    - Der Index des Indexregsisters wird zum Inhalt des Basisregisters und zu einer konstanten Abstandsgröße als Displacement addiert;
  - **Based Scaled Index Mode (registerindirekte, indizierte Adressierung mit Skalierungsfaktor)**
    - Der Index im Indexregister wird multipliziert mit einem Skalierungsfaktor und dann zur Adresse im Basisregister addiert;

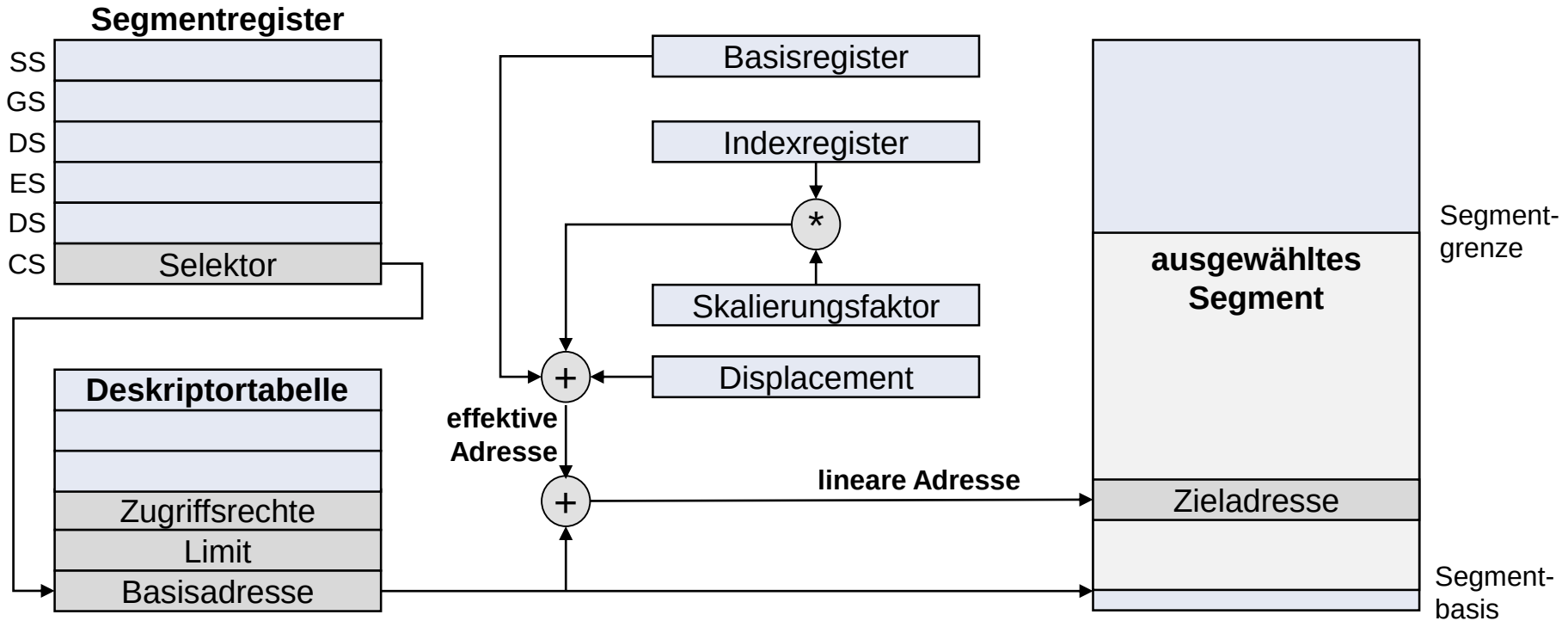
Adressierungsarten

- **Fallstudie IA-32**
  - **Based Scaled Index Mode with Displacement**
    - Der Index im Indexregister wird mit einem Skalierungsfaktor multipliziert und das Ergebnis zum Inhalt des Basisregisters und zu einem Displacement addiert;
    - `movl struct_base(%ebx, %esi, 4), %eax`

```
# Multipliziere den Inhalt des
# Indexregisters ESI mit 4 und
# addiere das Ergebnis auf den
# des Basisregisters EBX. Auf
# das sich daraus ergebende
# Ergebnis wird die konstante
# Abstandsgröße struct_base
# addiert. Der an dieser
# Adresse stehende Wert wird
# in das Register EAX geladen.
```

Adressierungsarten

Fallstudie IA-32



Adressierungsarten

- **Fallstudie RISC-V Befehlssatzarchitektur**
  - **Direktoperanden-Adressierung (immediate addressing)**
    - Der Operand steht als Konstante im Befehl
  - **Registeradressierung (register addressing)**
    - Der Operand steht in einem Register
  - **Register-indirekte Adressierung mit Displacement (base addressing)**
    - Der Operand steht im Speicher
    - Die Adresse ist die Summe des Inhalts des im Befehl referenzierten Registers und einer im Befehl enthaltenen konstanten Abstandsgröße

Adressierungsarten

- **Fallstudie RISV-V Befehlssatzarchitektur**
  - **Befehlszähler-relative Adressierung (PC-relative addressing)**
    - Die Sprungadresse ist die Summe des Inhalts des Befehlszählers und einer im Befehl enthaltenen konstanten Abstandsgröße

Befehlssatzarchitektur

- **Der Befehlssatz (instruction set)**
  - Umfasst die von der Hardware ausführbaren Operationen eines Prozessors;

  - **Befehlsarten (Beispiele):**
    - Transportbefehle
    - Arithmetisch-logische Befehle
    - Schiebe- und Rotationsbefehle
    - Gleitkommabefehle
    - Multimediabefehle, Vektorbefehle
    - Programmsteuerbefehle

    - Systemsteuerbefehle
    - Synchronisationsbefehle

Befehlssatzarchitektur

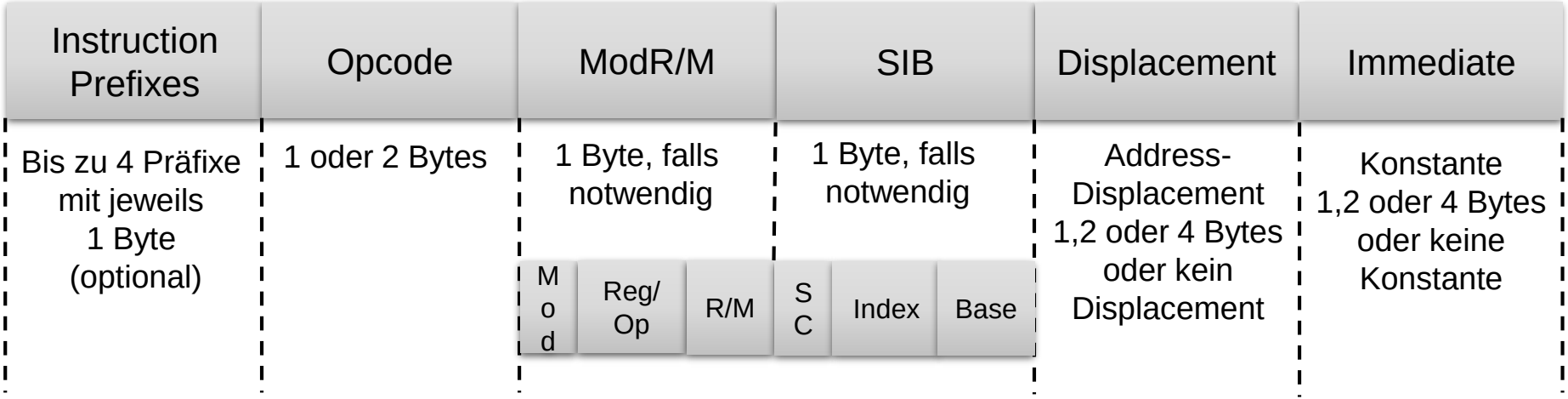
- **Befehlsformat**
  - Legt fest, welche Informationen in einem Maschinenbefehl explizit enthalten sind
  - Legt legt den grundlegenden Aufbau des Befehls fest
  - Wie lange ist ein Befehl (Anzahl der Bits)?
    - Festes oder variabel langes Befehlsformat?
  - Einem Befehlsformat liegt ein Ausführungsmodell zugrunde

Befehlsformat

- **Fallstudie IA-32**
  - Variabel langes Befehlsformat;
  - Byte-Struktur:
    - Die jeweiligen Längen der Befehlsformate sind ein Vielfaches eines Bytes.
    - Die Länge hängt von der Anzahl der Adressen und Adressierungsarten ab.

Befehlsformat

- **Fallstudie IA-32**



Befehlsformat

- **Fallstudie IA-32**
  - **Befehls-Präfixe (instruction prefixes):**
  - REP
  - REPNE/REPNZ
  - REPE/REPZ

  } Bewirken die wiederholte Ausführung von Stringbefehlen unter Vorgabe einer Abbruchbedingung

  - LOCK

  } Bewirkt die Nichtunterbrechbarkeit einer Befehlsausführung durch den Busarbiter

  - Address Size Override
  - Operand Size Override

  } Verändert die mit den Befehlsarten vorgegeben Adress- und Datenlängen von 32 Bit bzw. 16 Bit für einzelne Befehle

  - Segment Override

  } Segmentregisterzuordnung

Befehlsformat

- **Fallstudie IA-32**
  - **Opcode**
    - bezeichnet die auszuführende Operation;
    - ist ein oder zwei Bytes lang, wobei ein zusätzliches 3 Bit breites Opcode-Feld im ModR/M Feld codiert sein kann;
    - Im Opcode-Feld können weitere Informationen codiert sein: z. B. Codierung der Transportrichtung, die Länge des Displacements, Bedingungscodes etc.
  - **ModR/M-Feld**
    - spezifiziert die Adressierungsart für einen Speicheroperanden;
  - **SIB**
    - Bestimmte ModR/M Kodierungen benötigen ein zweites Byte zur vollständigen Spezifikation der Adressierungsart;
  - **Displacement (konstante Abstandsgröße)**
  - **Immediate (Angabe einer Konstanten)**

Befehlsformat

- **Fallstudie IA-32**
  - **Kodierung der Adressierungsarten (bei 32 Bit Adresslänge): ModR/M Byte**

r/m	effektive Adresse			Registeroperand		
	mod=00	mod=01	mod=10	mod=11		
000	DS[EAX]	DS[EAX+d8]	DS[EAX+d16]	AL	AX	EAX
001	DS[ECX]	DS[ECX +d8]	DS[ECX +d16]	CL	CX	ECX
010	DS[EDX]	DS[EDX +d8]	DS[EDX +d16]	DL	DX	EDX
011	DS[EBX]	DS[EBX +d8]	DS[EBX +d16]	BL	BX	EBX
100	**sib** folgt			AH	SP	ESP
101	DS:d32	DS[EBP +d8]	DS[EBP +d16]	CH	BP	EBP
110	DS[ESI]	DS[ESI +d8]	DS[ESI +d16]	DH	SI	ESI
111	DS[EDI]	DS[EDI +d8]	DS[EDI +d16]	BH	DI	EDI
				1	2	3

Befehlsformat

- **Fallstudie IA-32**
  - **Kodierung der Adressierungsarten (bei 32 Bit Adresslänge): ModR/M Byte**

Base	effektive Adresse			Index	SC	Faktor
	mod=00	mod=01	mod=10			
000	DS[EAX+(si)]	DS[EAX+(si)+d8]	DS[EAX+(si)+d32]	EAX	00	x1
001	DS[ECX +(si)]	DS[ECX +(si)+d8]	DS[ECX +(si)+d32]	ECX	01	x2
010	DS[EDX +(si)]	DS[EDX +(si)+d8]	DS[EDX +(si)+d32]	EDX	10	x4
011	DS[EBX +(si)]	DS[EBX +(si)+d8]	DS[EBX +(si)+d32]	EBX	11	x8
100	SS[ESP +(si)]	SS[ESP +(si)+d8]	SS[ESP +(si)+d32]	k.I.	00*)	
101	DS:[d32 +(si)]	DS[EBP +(si)+d8]	DS[EBP +(si)+d32]	EBP		
110	DS[ESI +(si)]	DS[ESI +(si)+d8]	DS[ESI +(si)+d32]	ESI		
111	DS[EDI +(si)]	DS[EDI +(si)+d8]	DS[EDI +(si)+d32]	EDI		
Index	mod-Feld im modR/B-Byte					
	Base, Index, SC im SIB kodiert					
	*) wenn im Index-Feld 100 kodiert ist, dann muss im SC=00 sein					

Befehlsformat

- **Fallstudie RISC-V**
  - Offene Befehlssatzarchitektur (ISA)
    - Nicht patentiert;
      - Somit darf jeder einen RISC-V Mikroprozessor entwerfen, herstellen, weiterentwickeln und verkaufen;
      - RISC-V Mikroprozessoren gibt es auch als Open-Source-Hardware;
      - Daher auch sehr weit verbreitet;
  - Festes Befehlsformat
    - Befehle fester Länge (32-Bit);
  - Verschiedene Befehlsgruppen haben unterschiedliche Befehlsformate.

Befehlsformat

- **Fallstudie RISC-V**
  - **R-Format (R-type, arithmetic instruction format)**
    - Arithmetische und logische Befehle mit 2 Quelloperanden und einem Zielregister

    - **Felder**
      - opcode: 7 Bit Operationscode des Befehls
      - rd: 5 Bit Registernummer des Zielregisters
      - funct3: 3 Bit Funktionscode (zusätzlicher OpCode)
      - rs1: 5 Bit Registernummer des ersten Quelloperanden
      - rs2: 5 Bit Registernummer des zweiten Quelloperanden
      - funct7: 7 Bit Funktionscode (zusätzlicher OpCode)

Befehlsformat

- **Fallstudie RISC-V**
  - **R-Format (R-type, arithmetic instruction format)**
    - Arithmetische und logische Befehle mit 2 Quelloperanden und einem Zielregister

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

    - **Beispiel Additionsbefehl:** `add x9,x20,x21`

0	21	20	0	9	51

Dezimale Darstellung

0000000	10101	10100	000	01001	0110011

binäre Darstellung

Befehlsformat

- **Fallstudie RISC-V**
  - **R-Format (R-type, arithmetic instruction format)**

Funct7	Funct3	Opcode	Befehl
0000000	000	0110011	`add`
0100000	000	0110011	`sub`
0000000	001	0110011	`sll`
0000000	100	0110011	`xor`
0000000	101	0110011	`srl`
0000000	101	0110011	`sra`
0000000	110	0110011	`or`
0000000	111	0110011	`and`
0001000	011	0110011	`lr.w`
0001100	011	0110011	`sc.w`

Befehlsformat

- **Fallstudie RISC-V**
  - **I-Format (I-type, loads and immediate arithmetic)**
    - Arithmetische Befehle mit einer Konstanten

    - **Felder :**
      - rs1: Registernummer des Quelloperanden
      - immediate: Konstante wird als 2er-Komplementzahl interpretiert (Bereich -2^{11} bis $2^{11}-1$)

Befehlsformat

- **Fallstudie RISC-V**
  - **I-Format (I-type, loads and immediate arithmetic)**
    - Arithmetische Befehle mit einer Konstanten

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

    - **Beispiel: addi x1,x2,1000**

1000	2	0	1	19	Dezimale Darstellung
001111101000	00010	000	00001	0010011	binäre Darstellung

Befehlsformat

- **Fallstudie RISC-V**
  - **I-Format (I-type, loads and immediate arithmetic)**
    - Lade-Befehle

    - **Felder :**
      - rs1: Registernummer der Basisadresse
      - rd: Registernummer des Registers, in das der Wert geladen wird
      - immediate: Byte-Offset kann ein Wort in einem Bereich von $\pm 2^{11}$ oder $\pm$2048 Bytes ($\pm 2^{9}$ oder $\pm$512 Wörter) bezüglich der Basisadresse referenzieren

Befehlsformat

- **Fallstudie RISC-V**
  - **I-Format (I-type, loads and immediate arithmetic)**
    - Lade-Befehle

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

    - **Beispiel: lw x1, 1000(x2)**

1000	2	2	1	3

Dezimale Darstellung

001111101000	00010	010	00001	0000011

binäre Darstellung

Befehlsformat

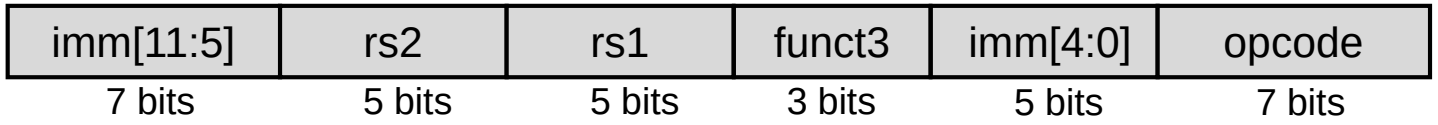
- **Fallstudie RISC-V**
  - **I-Format (I-type, loads and immediate arithmetic)**

Funct6	Funct3	Opcode	Befehl
	000	0000011	`lb`
	001	0000011	`lh`
	010	0000011	`lw`
	100	0000011	`lbu`
	101	0000011	`lhu`
	000	0010011	`addi`
000000	001	0010011	`slli`
	100	0010011	`xori`
000000	101	0100011	`srli`
010000	101	0010011	`srai`

Funct6	Funct3	Opcode	Befehl
	110	0010011	`ori`
	111	0010011	`andi`
	000	1100111	`jalr`

Befehlsformat

- **Fallstudie RISC-V**
  - **S-Format (S-type, stores)**
    - **Speicher-Befehle**

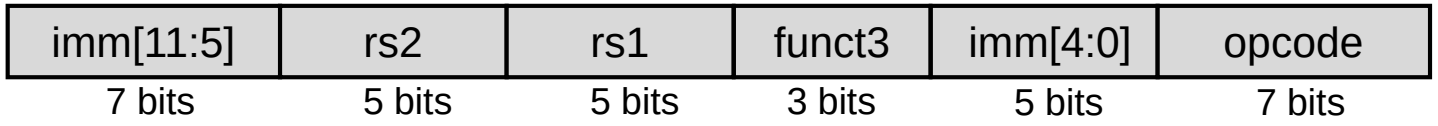


    - **Felder:**
      - rs1: Registernummer der Basisadresse
      - rs2: Registernummer des Quelloperanden
      - imm[11:5]imm[4:0] Byte-Offset kann ein Wort in einem Bereich von ± 2^{11} oder 2048 Bytes (± 2^{8} oder 512 Wörter) bezüglich der Basisadresse referenzieren

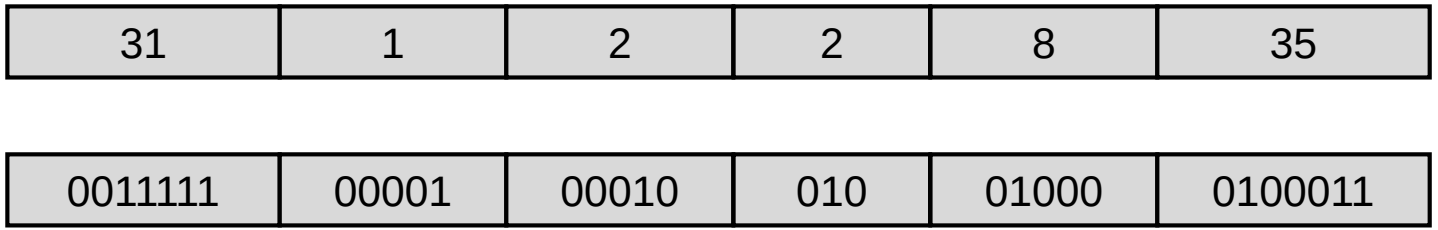
        Aufgeteilt in zwei Felder, damit die rs1- und rs2-Felder in jedem Format immer an denselben Bitstellen sind

Befehlsformat

- **Fallstudie RISC-V**
  - **S-Format (S-type, stores)**
    - **Speicher-Befehle**



    - **Beispiel: sw x1, 1000(x2)**



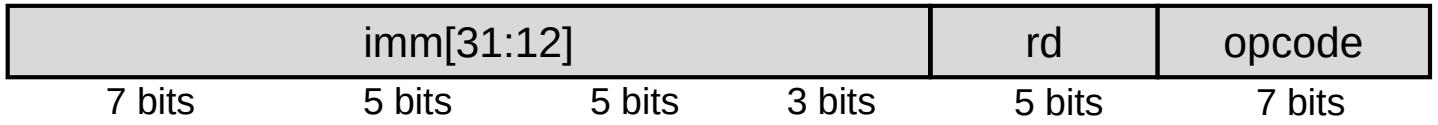
Befehlsformat

- **Fallstudie RISC-V**
  - **S-Format (S-type, stores)**

Funct6	Funct3	Opcode	Befehl
	000	0100011	`sb`
	001	0100011	`sh`
	010	010011	`sw`

Befehlsformat

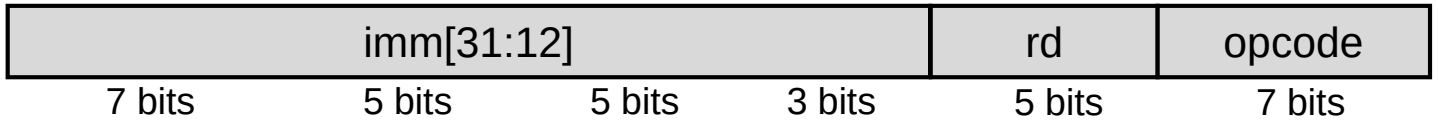
- **Fallstudie RISC-V**
  - **U-Format (U-type, upper immediate format)**
    - Lade-Befehl mit langen Konstanten



    - **Felder:**
      - rd — Registernummer des Ziels, in das die Konstante geladen wird
      - Imm[31:12] — höchstwertige 20 Bits einer Konstanten

Befehlsformat

- **Fallstudie RISC-V**
  - **U-Format (U-type, upper immediate format)**
    - Lade-Befehl mit langen Konstanten

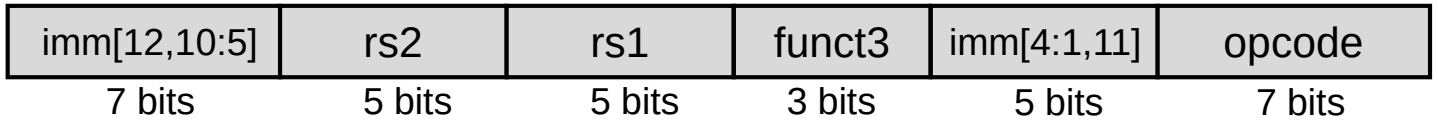


- **Beispiel:**

```
lui x10,0x87654          # x10 = 0x87654000
addi x10,x10,0x321       # x10 = 0x87654321
```

Befehlsformat

- **Fallstudie RISC-V**
  - **SB-Format (SB-type, conditional branch format)**
    - Verzweigungen (bedingte Sprünge)



    - **Felder:**
      - rs1, rs2: Registernummern der Register, deren Inhalte verglichen werden
      - imm[12,10:5]: 13-Bit langes Byte-Offset, das auf den Inhalt des PCs addiert wird
      - imm[4:1,11]: Anzahl der Halbwörter zwischen Verzeigung und Sprungziel
      - Aufgeteilt in zwei Felder, damit die rs1- und rs2-Felder in jedem Format immer an denselben Bitstellen sind

Befehlsformat

- **Fallstudie RISC-V**
  - **SB-Format (SB-type, conditional branch format)**
    - **Verzweigungen (bedingte Sprünge)**

imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- **Beispiel:** `beq x4,x5,End`

```
# Sprung zur Marke „End“
# 12 Bytes entfernt
```

0 0	4	5	0	6 0	99

Dezimale Darstellung

0 000000	00100	00101	0	0110 0	1100011

binäre Darstellung

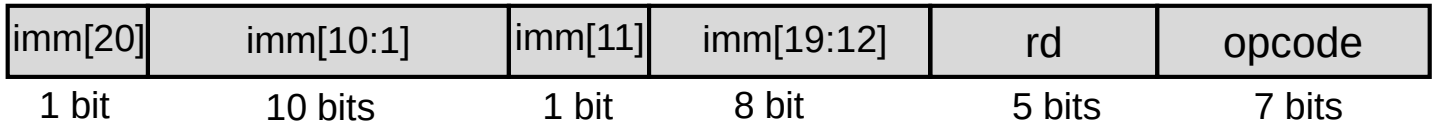
Befehlsformat

- **Fallstudie RISC-V**
  - **SB-Format (SB-type, conditional branch format)**

Funct6	Funct3	Opcode	Befehl
	000	1100111	`beq`
	001	1100111	`bne`
	100	1100111	`blt`
	101	1100111	`bge`
	110	1100111	`bltu`
	111	1100111	`bgeu`

Befehlsformat

- **Fallstudie RISC-V**
  - **UJ-Format (UJ-type, unconditional jump format)**
    - Prozeduraufruf (jump-and-link)

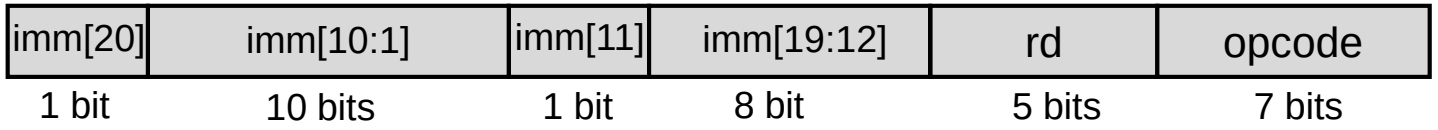


    - **Felder:**
      - rd — Registernummer, in der die Link-Adresse (Adresse, die auf die Adresse des Sprungbefehls folgt) gespeichert wird
      - imm[20]
      - imm[10:1]
      - imm[11]
      - imm[19:12]

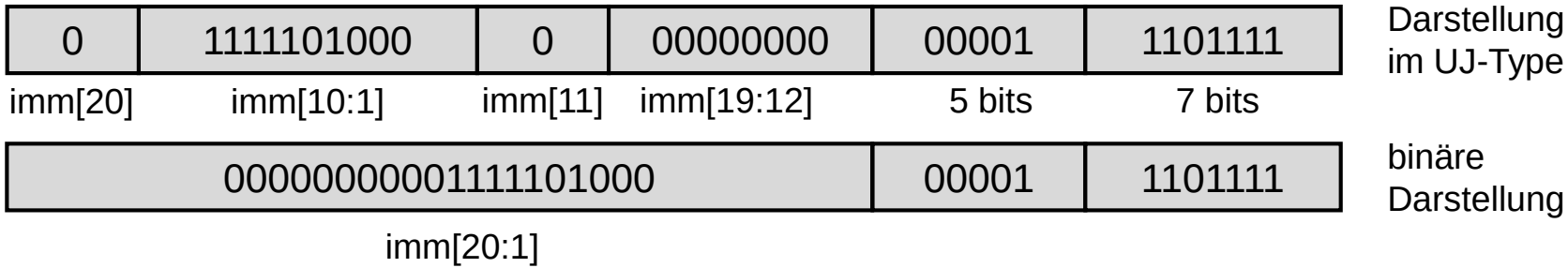
      } Sprungadresse

Befehlsformat

- **Fallstudie RISC-V**
  - **UJ-Format (UJ-type, unconditional jump format)**
    - Prozeduraufruf (jump-and-link)



- **Felder:** `jal x1, 2000 # go to location 2000`$_{10}$ `= 0111 1101 0000`



RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur swap**
  - Tauscht die Inhalte zweier Speicherelemente
  - Pseudo-Hochsprache:

```
swap(int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp:
}
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur swap**
  - Übersetzung in Assemblersprache

1. Register Allokation für **`swap`**
   - Die Variablen **`v`** und **`k`** werden den Registern **`x10`** und **`x11`** zugeordnet
   - Die lokale Variable **`temp`** wird dem Register **`x5`** zugeordnet

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur swap**
  - Übersetzung in Assemblersprache

2. Addressrechnung
   - Berechnung der Adresse von **`v[k]`** durch Multiplikation von **`k`** mit **`4`**
   - Adresse von v in **`x6`**

```
slli    x6, x11, 2      # reg x6 = k * 4
add     x6, x10, x6     # reg x6 = v + (k*4)
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur swap**
  - Übersetzung in Assemblersprache

3. Laden der Elemente von **`v`** in die Register
   - Lade **`v[k]`** mit Hilfe **`x6`** und **`v[k+1]`** mit **`x6`** und dem Offset **`4`**

```
lw      x5, 0(x6)       # reg x5 (temp) = v[k]
lw      x7, 4(x6)       # reg x7 = v[k + 1]
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur swap**
  - Übersetzung in Assemblersprache

4. Speichern der Registerinhalte in die vertauschten Speicheradressen
   - Lade **`v[k]`** mit Hilfe **`x6`** und **`v[k+1]`** mit **`x6`** und dem Offset **`4`**

```
sw      x7, 0(x6)       # v[k] = reg x7
sw      x5, 4(x6)       # v[k+1] = reg x5 (temp)
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur swap**
  - RISC-V Assemblercode für **`swap`**

```
swap:   slli    x6, x11, 2      # reg x6 = k * 4
        add     x6, x10, x6     # reg x6 = v + (k*4)
        lw      x5, 0(x6)       # reg x5 (temp) = v[k]
        lw      x7, 4(x6)       # reg x7 = v[k + 1]
        sw      x7, 0(x6)       # v[k] = reg x7
        sw      x5, 4(x6)       # v[k+1] = reg x5 (temp)
        jal     x0, 0(x1)       # return to calling routine
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: die Prozedur sort**
  - Sortiert die Elemente eines Feldes (Insertion Sort mit paarweisem Austausch)
  - Pseudo-Hochsprache:

```
sort(int v[ ], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i-1; j >= 0 && v[j] > v[j+1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**
  1. Register Allokation für **`sort`**
     - Die Variablen **`v`** und **`n`** werden den Registern **`x10`** und **`x11`** zugeordnet
     - Die lokalen Variablen **`i`** und **`j`** werden den Registern **`x19`** und **`x20`** zugeordnet

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  2. Der Code für den Rumpf der Prozedur **`sort`**
     - Übersetzung der 1. **`for`**-Schleife: **`for (i=0; i<n; i+=1) {`**
     - Initialisiere **`i = 0`**

```
addi    x19, x0, 0 # i = 0
```

     - Inkrementiere **`i`**

```
addi    x19, x19, 1# i += 1
```

     - Bedingung für Schleifenende: **`i ≥ n`**

```
for1tst: bge   x19, x11, exit1 # go to exit1 if x19 ≥ x11 (i ≥ n)
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  2. Der Code für den Rumpf der Prozedur **`sort`**
     - Gerüst für die 1. **`for`**-Schleife: **`for (i=0; i<n; i+=1) {`**

```
        addi    x19, x0, 0 # i = 0
for1tst:
        bge x19, x11, exit1 # go to exit1 if x19 ≥ x11 (i ≥ n)
            …
            (Rumpf der ersten Schleife)
            …
        addi    x19, x19, 1# i += 1
        jal  x0, for1tst
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  2. Der Code für den Rumpf der Prozedur **`sort`**
     - Übersetzung der 2. **`for`**-Schleife: **`for(j= i-1; j>=0 && v[j]>v[j+1]); j-=1) {`**
     - Initialisiere **`j = i - 1`**

```
addi    x20, x19, -1    # j = i - 1
```

     - Dekrementiere **`j`**

```
addi    x20, x20, -1    # j -= 1
```

     - 1. Bedingung für Schleifenende: **`j < 0`**

```
for2tst: blt   x20, x0, exit2 # go to exit2 if x20 < 0 (j < 0)
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

2. Der Code für den Rumpf der Prozedur `sort`
   - Übersetzung der 2. `for`-Schleife: `for(j= i-1; j>=0 && v[j]>v[j+1]); j-=1) {`
   - 2. Bedingung für Schleifenende: `v[j]` ≤ `v[j+1]`
     - Berechnung des Offsets (Multiplikation von `j` mit `4`) und Addition der Basisadresse von `v`

```
slli    x5, x20, 2      # reg x5 = j*4
add     x5, x10, x5     # reg x5 = v + (j*4)
```

     - Lade `v[j]`

```
lw      x6, 0(x5)       # reg x6 = v[j]
```

     - Lade `v[j+1]`

```
lw      x7, 4(x5)       # reg x7 = v[j+1]
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  2. Der Code für den Rumpf der Prozedur **`sort`**
     - Übersetzung der 2. **`for`**-Schleife: **`for(j= i-1; j>=0 && v[j]>v[j+1]); j-=1) {`**
     - 2. Bedingung für Schleifenende: **`v[j]` ≤ `v[j+1]`**
       - Test für Schleifenende

```
ble  x6, x7, exit2 # go to exit2 if x6 ≤ x7
```

       - Am Ende der Schleife wird zurück zum Test der inneren Schleife gesprungen

```
jal  x0,  for2tst  # verzweige zum Test der inneren Schleife
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  2. Der Code für den Rumpf der Prozedur **`sort`**
     - Gerüst für die 2. **`for`**-Schleife: **`for(j= i-1; j>=0 && v[j]>v[j+1]); j-=1) {`**

```
        addi    x20, x19, -1    # j = i - 1
for2tst:
        blt     x20, x0, exit2  # go to exit2 if x20 < 0 ≥ (j < 0)
        slli    x5, x20, 2      # reg x5 = j*4
        add     x5, x10, x5     # reg x5 = v + (j*4)
        lw      x6, 0(x5)       # reg x6 = v[j]
        lw      x7, 4(x5)       # reg x7 = v[j+1]
        ble     x6, x7, exit2   # go to exit2 if x6 ≤ x7
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**
  2. Der Code für den Rumpf der Prozedur **`sort`**
     - Gerüst für die 2. **`for`**-Schleife: **`for(j= i-1; j>=0 && v[j]>v[j+1]); j-=1) {`**

```
        …
        (Rumpf der zweiten Schleife)
        …
addi    x20, x20, 1     # j -= 1
jal     x0, for2tst     # springe zum Test der inneren Schleife
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  3. Parameterübergabe in der Prozedur **`sort`**
     - Die Prozedur **`sort`** benötigt die Werte in den Register **`x10`** und **`x11`**, ebenso benötigt die Prozedur **`swap`**, dass die Parameter in diesen Registern stehen (Konvention)
     - Lösung: Die Parameter für **`sort`** werden an früherer Stelle in andere Register kopiert, so dass **`x10`** und **`x11`** für **`swap`** zur Verfügung stehen.

```
addi    x21, x10, 0# copy parameter x10 nach x21
addi    x22, x11, 0# copy parameter x11 nach x22
```

     - Dann übergeben die Parameter an swap

```
addi    x10, x21, 0# first swap parameter is v
addi    x11, x20, 0# second swap parameter is j
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Schrittweise Entwicklung des Assembler-Codes für die Prozedur `sort`**

  4. Registerinhalte retten

```
sort:   addi    sp, sp, -20 # reserviert Platz für 5 Register
                            # auf dem Stack
        sw      x1, 16(sp)  # Rette Rücksprungadresse auf dem Stack
        sw      x22, 12(sp) # Rette x22 auf dem Stack
        sw      x21, 8(sp)  # Rette x21 auf dem Stack
        sw      x20, 4(sp)  # Rette x20 auf dem Stack
        sw      x19, 0(sp)  # Rette x19 auf dem Stack
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Prozedur `sort`:**
  - RISC-V Assemblercode
  1. Registerinhalte retten

```
sort:   addi    sp, sp, -20 # reserviert Platz für 5 Register
                            # auf dem Stack
        sw      x1, 16(sp)  # Rette Rücksprungadresse auf dem Stack
        sw      x22, 12(sp) # Rette x22 auf dem Stack
        sw      x21, 8(sp)  # Rette x21 auf dem Stack
        sw      x20, 4(sp)  # Rette x20 auf dem Stack
        sw      x19, 0(sp)  # Rette x19 auf dem Stack
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Prozedur `sort`:**
  - RISC-V Assemblercode

2. Prozedurrumpf: kopiere Parameter

```
        addi    x21, x10, 0# kopiert Parameter x10 nach x21
        addi    x22, x11, 0# kopiert Parameter x11 nach x22
```

3. Prozedurrumpf: äußere Schleife

```
        addi    x19, x0, 0  # i = 0
for1tst:
        bge x19, x22, exit1# go to exit1 if 1 >= n
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Prozedur `sort`:**
  - RISC-V Assemblercode

4. Prozedurrumpf: innere Schleife

```
        addi    x20, x19,-1# j = i-1
for2tst:
        blt x20, x0, exit2 # go to exit2 if j < 0
        slli    x5, x20, 2 # x5 = j*4
        add     x5, x21, x5# x5 = v + (j*4)
        lw      x6, 0(x5)  # x6 = v[j]
        lw      x7, 4(x5)  # x7 = v[j+1]
        ble x6, x7, exit2  # go to exit 2 if x6 < x7
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Prozedur `sort`:**
  - RISC-V Assemblercode

5. Prozedurrumpf: Parameterübergabe und Prozesuraufruf

```
addi    x10, x21, 0# 1. swap Parameter ist v
addi    x11, x20, 0# 2. swap Parameter ist j
jal x1, swap         # Prozeduraufruf swap
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Prozedur `sort`:**
  - RISC-V Assemblercode

6. Prozedurrumpf: innere Schleife

```
        addi    x20, x20, -1# j -=1
        jal x0, for2tst      # go to for2tst
```

7. Prozedurrumpf: äußere Schleife

```
exit2:  addi    x19, x19, 1# i +=1
        jal x0, for1tst      # go to for1tst
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

RISC-V Assembler-Programmierung

- **Fallstudie: Prozedur `sort`:**
  - RISC-V Assemblercode

6. Wiederherstellen der Register

```
exit1:  lw      x19, 0(sp)  # Lade x19 vom Stack
        lw      x20, 4(sp)  # Lade x20 vom Stack
        lw      x21, 8(sp)  # Lade x21 vom Stack
        lw      x22, 12(sp) # Lade x22 vom Stack
        lw      x1, 16(sp)  # Lade Rücksprungadresse vom Stack
        addi    sp, sp, 20  #   Lade Stackpointer
```

7. Prozedurrücksprung

```
        jalr    x0, 0(x1)   # Rücksprung zur aufrufenden Prozeur
```

Quelle: Patterson, D.; Hennessy, J.: Computer Organization and Design RISC-V Edition. Morgan Kaufman Publishers, 2021, 2. ed.

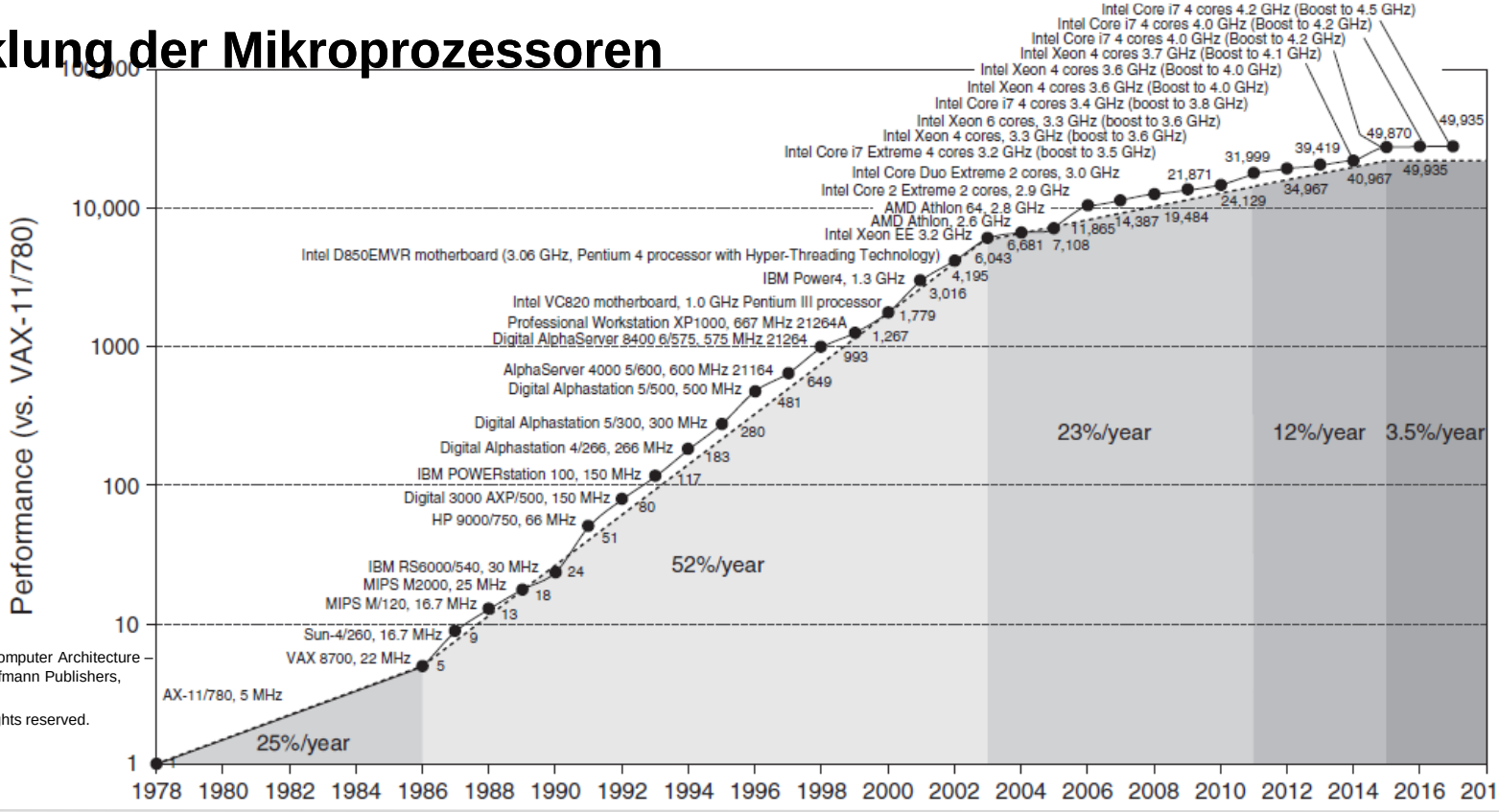
Befehlssatzarchitektur

- **Diskussion CISC / RISC**
  - CISC-Befehlssatz-Architektur (ISA)
  - Complex Instruction Set Computer

  - RISC-Befehlssatz-Architektur (ISA)
  - Reduced Instruction Set Computer

Diskussion CISC / RISC

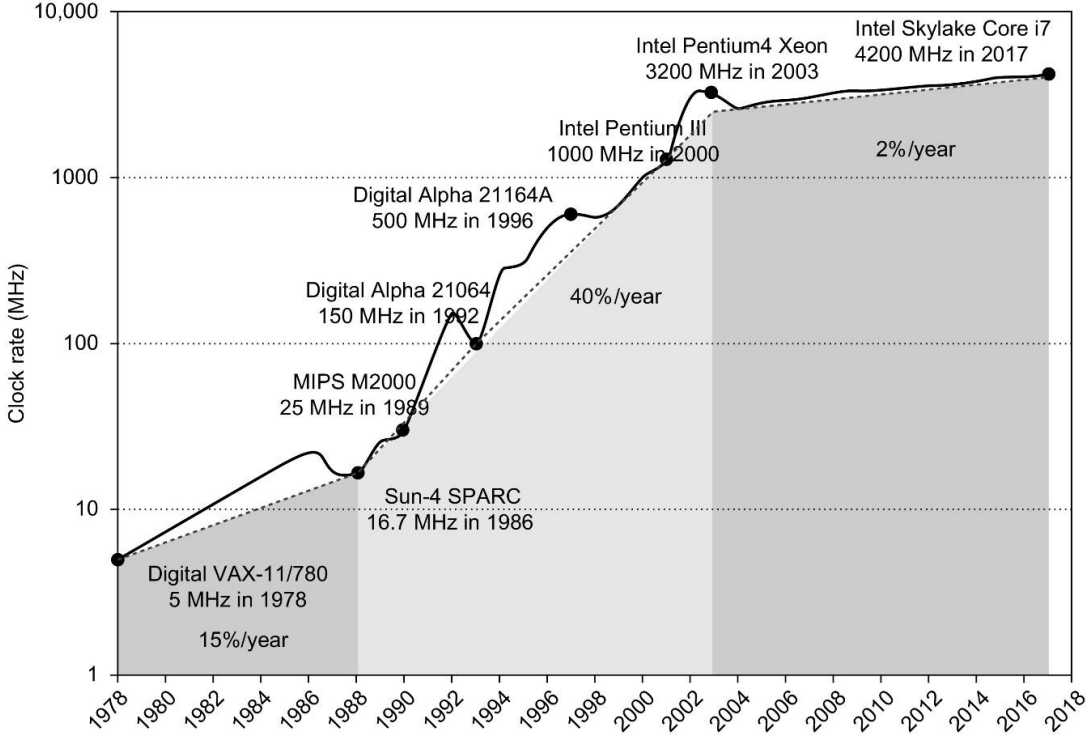
- **Entwicklung der Mikroprozessoren**



Quelle: J. Hennessy, D. Patterson: Computer Architecture – A Quantative Approach, Morgan Kaufmann Publishers, 6th Ed., 2019

Diskussion CISC / RISC

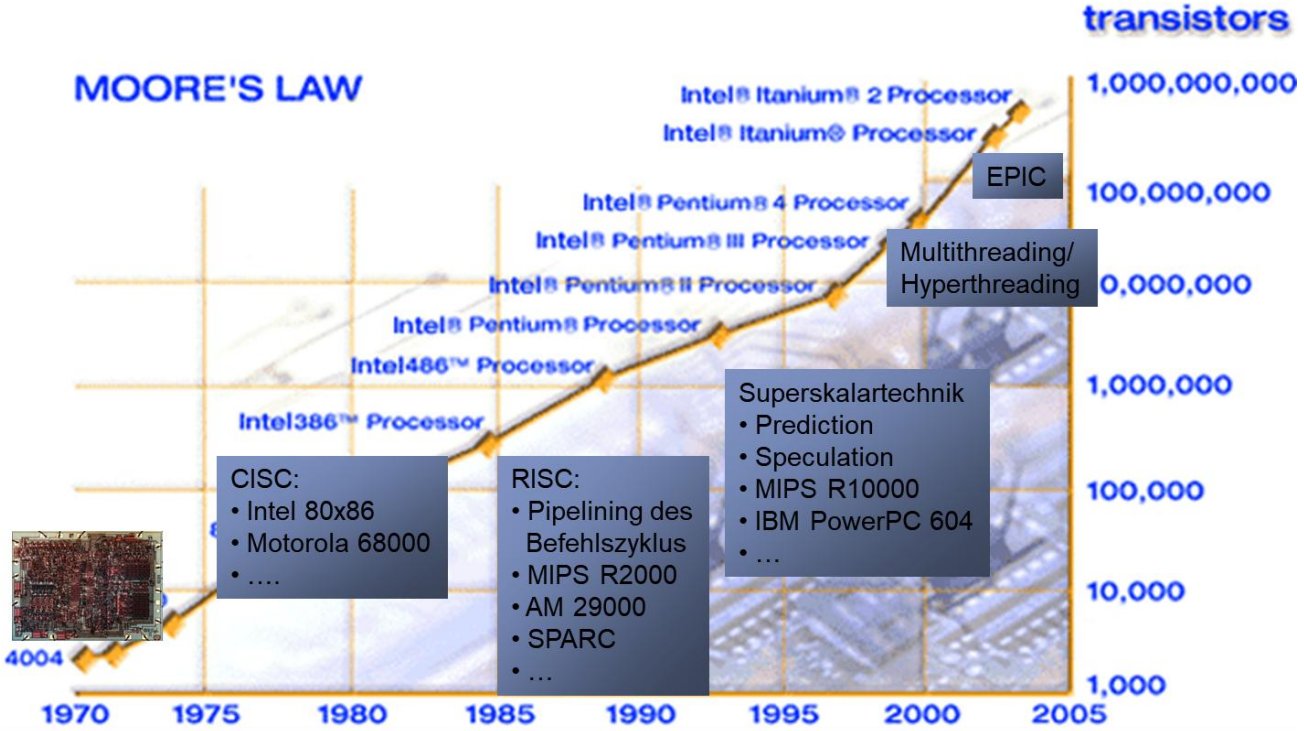
- **Entwicklung der Mikroprozessoren**



Quelle: J. Hennessy, D. Patterson: Computer Architecture –
A Quatative Approach, Morgan Kaufmann Publishers, 6th Ed., 2019

Diskussion CISC / RISC

- **Entwicklung der Mikroprozessoren**



Diskussion CISC / RISC

- **Ausführungszeit eines Programms (CPU-Zeit) auf einer CPU (gemessen in Taktzyklen oder in Zeiteinheiten) wird bestimmt durch:**
  - Anzahl der ausgeführten Befehle des Programms (IC; instruction count)
  - Mittlere Anzahl der Taktzyklen pro Befehl (CPI, cycles per instruction)
  - Taktfrequenz oder Zyklendauer
- **Gleichung der Prozessorleistung (processor performance equation)**

$$\mathbf{CPU - Zeit = IC * CPI * Zyklendauer}$$

$$\mathbf{CPU - Zeit = \frac{IC * CPI}{Taktfrequenz}}$$

$$\mathbf{CPI = \frac{Anzahl\ der\ Taktzyklen}{IC}}$$

Diskussion CISC / RISC

- **CISC ISA**
  - Befehlssatz mit komplexen Maschinenbefehlen
    - Einfachere Programmierung in Assembler
      - Tendenz komplexe Funktionalität mit den Instruktionen bereitzustellen, um Assembler-Programmierer zu unterstützen
      - Entwurf der CISC-Befehlssatz-Architekturen in einer Zeit, in der viel in Assembler programmiert worden ist
    - Operationen einer Instruktion möglichst nahe an Anweisungen einer höheren Programmiersprache
    - Neben grundlegenden Datentypen auch Bitstreams, Arrays …
  - Mikroprogrammierte Implementierung des Steuerwerks
  - Variables Befehlsformat und Befehlslänge

Diskussion CISC / RISC

- **CISC ISA**
  - Betrachtung der Faktoren, welche die CPU-Leistung beeinflussen:

$$\text{CPU} - \text{Zeit} = \text{IC} * \text{CPI} * \text{Zyklendauer}$$

    - Reduzierung IC, also die mittlere Anzahl der auszuführenden Befehle für ein Programm
    - Je kompakter der Befehlsstrom, desto besser die Nutzung des Haupt- / Cache-Speichers, desto weniger Befehle müssen geholt werden, um ein Programm auszuführen

  - Beispiele:
  - IBM System 360, 30 (Familienkonzept)
  - Intel x86 ISA (IA-32)
  - Motorola 68000 Familie

Diskussion CISC / RISC

- **CISC ISA**
  - Die Realisierung eines großen Befehlssatzes mit vielen Adressierungsarten und Datentypen bedingt ein sehr komplexes Steuerwerk
  - Hohe Entwicklungskosten und lange Entwicklungsdauer
  - Dekodierung und Ausführung der Befehle benötigt relativ viel Zeit
  - Wegen der unterschiedlichen Komplexität der Befehle differiert ihre Ausführungszeit beträchtlich
  - Auch die Ausführung desselben Befehls benötigt wegen der unterschiedlichen Adressierungsarten mal mehr und mal weniger Taktzyklen
  - Komplexität des Befehlssatzes, der Adressierungsarten und Datentypen erschwert die Entwicklung von Compilern, die effizienten Code generieren

  - Mitte der 70er Jahr begann die Suche nach Alternativen

Diskussion CISC / RISC

- **RISC ISA**
  - Betrachtung der Faktoren, welche die CPU-Leistung beeinflussen:

$$CPU - Zeit = IC * CPI * Zyklendauer$$

    - Reduzierung des Faktors CPI, also der mittleren Anzahl der Zyklen pro Instruktion
      - Ziel: (CPI ~ 1)

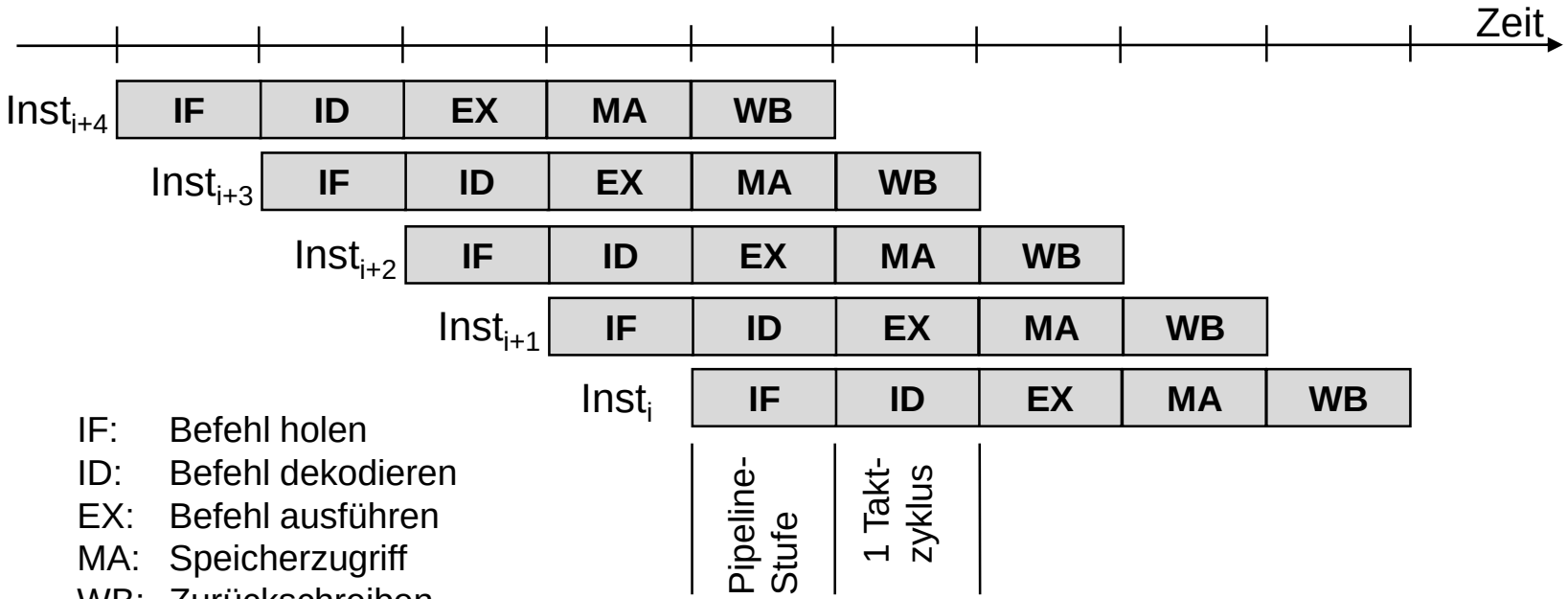
  - Beispiele:
    - Forschung bei IBM, Standford (MIPS) Berkeley (RISC)
    - AMD 29000
    - MIPS R2000, Nachfolger
    - SPARC

Diskussion CISC / RISC

- **RISC ISA**
  - Einfache Maschinenbefehle
    - Einheitliches und festes Befehlsformat
  - Load/Store Architektur
    - Befehle arbeiten auf Registeroperanden
    - Lade- und Speicherbefehle greifen auf Speicher zu
  - Einzyklus-Maschinenbefehle
    - Einheitliches Zeitverhalten der Maschinenbefehle, wovon nur Lade- und Speicherbefehle sowie die Verzweigungsbefehle abweichen
    - Effizientes Pipelining des Maschinenbefehlszyklus
  - Optimierende Compiler
    - Reduzierung der Befehle im Programm

Diskussion CISC / RISC

- **RISC ISA**
  - Effizientes Pipelining des Maschinenbefehlszyklus



IF: Befehl holen
ID: Befehl dekodieren
EX: Befehl ausführen
MA: Speicherzugriff
WB: Zurückschreiben

Diskussion CISC / RISC

- **RISC ISA**
  - Effizientes Pipelining des Maschinenbefehlszyklus
    - Alle Pipelinestufen benützen unterschiedliche Ressourcen
      - Keine Ressourcenkonflikte
    - Pipelining erhöht den Durchsatz
      - Mit jedem Takt wird unter Annahme idealer Verhältnisse ein Befehl geholt bzw. beendet.
      - Im eingeschwungenen Zustand der Pipeline gilt:
        - Durchsatz = 1 Befehl pro Taktzyklus (CPI ~ 1)
      - Aber, reduziert nicht die Ausführungszeit einer individuellen Instruktion
    - Dauer Zykluszeit, Taktzyklus:
      - Abhängig vom kritischen Pfad, der langsamsten Pipelinestufe