

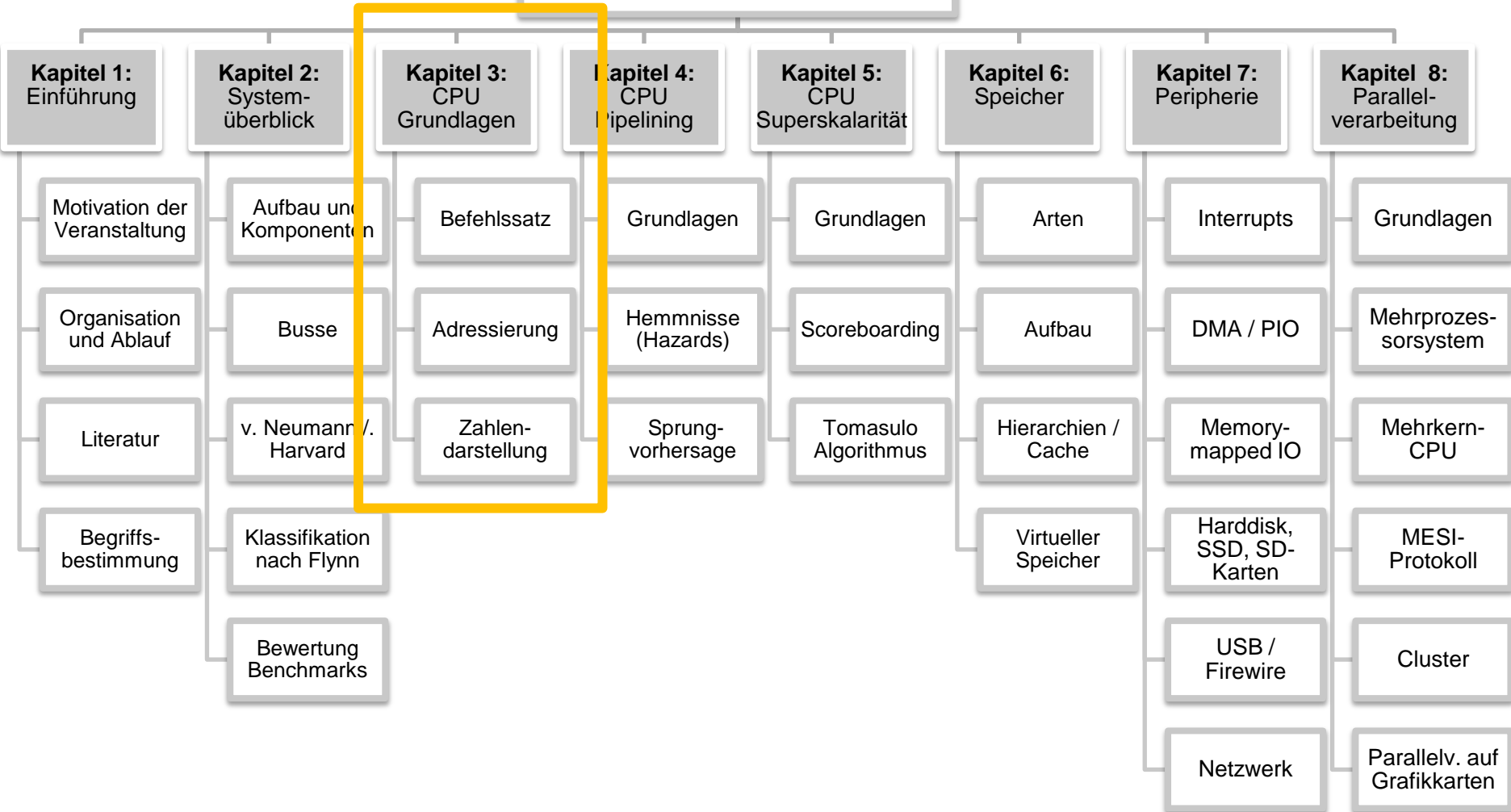
RECHNERARCHITEKTUR

Kapitel 3 – CPU: Grundlagen

Prof. Dr. L. Wischhof <wischhof@hm.edu>

Fakultät 07 – Hochschule München

Rechnerarchitektur



CPU: Grundlagen

Motivation

Typische Fragestellungen:

- Was kennzeichnet die Befehlssatzarchitektur einer CPU?
 - Welche unterschiedlichen Arten von Befehlen gibt es?
- Wie ist eine CPU aufgebaut und wie arbeitet sie?
- Wie werden Informationen adressiert? Wie werden sie gespeichert?



Technologie

Komplexität von Schaltungen

Schaltung	Aufgabe	Komplexität
Gatter	Verknüpfung zweier Binärwerte	4 Transistoren
Flipflop	Speichert ein Bit	6 Transistoren
Addierer	Addiert 64 Bit	>400 Transistoren
Datenpfad	Komplexes Rechenwerk mit Puffern	>200.000 Transistoren
Pentium II	Prozessor	4,5 Mio. Transistoren
Pentium 4	Prozessor	42 Mio. Transistoren
Xeon-Dunnington	6-Kern-Prozessor	1900 Mio. Transistoren

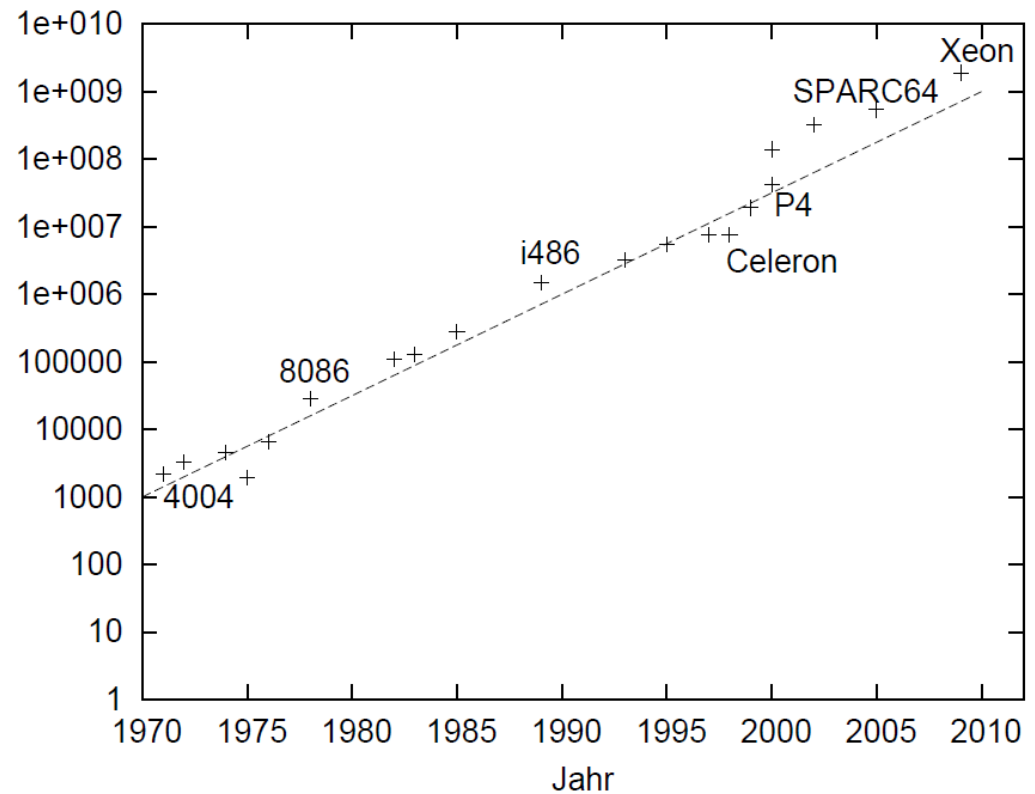


Technologie

Moore'sches Gesetz

**„Anzahl der Transistoren pro Chip verdoppelt sich
alle 18-24 Monate.“**

(G. Moore, 1965)



Technologie

Wichtige Begriffe (1/2)

Hardware-Beschreibungssprache

(Hardware Description Language, HDL)

- „Programmiersprache“ zur Beschreibung von Hardware, wichtigste Vertreter: **VHDL** (Very High Speed Integrated Circuit Hardware Description Language) und **Verilog**
- Entwicklungsumgebung simuliert und generiert Hardware, z.B. Bitfile für FPGA (Field Programmable Gate Array) oder ASIC (Application Specific Integrated Circuit)

Synthetisierbarer Kern

(Softcore)

- Prozessorkern, der in HDL vorliegt und von Lizenznehmern in eigenen Designs verwendet werden kann
- Beispiele: Cortex-M (ARM), MicroBlaze (Xilinx), Nios (Altera), OpenRISC



Technologie

Wichtige Begriffe (2/2)

System-on-a-Chip (SoC)

- Auf einem Chip: Prozessorkern, Speicher, Schnittstellen, Interrupt-, Grafikcontroller, etc.
- Daher kein Chipsatz „außen herum“ erforderlich
- Beispiele: A5 (Apple), XScale (Intel), ORSoC (OpenRISC SoC)

Eingebettete Systeme (Embedded Systems)

- (kleine) Computer, die in Produkten genutzt („eingebettet“) werden
- Beispiele: Kraftfahrzeug, Unterhaltungselektronik, Haushaltsgeräte, ...
- Anzahl übersteigt inzwischen die Zahl klassischer Computer bei weitem!



Technologie

RISC vs. CISC

Complex Instruction Set Computer (CISC)

- Umfangreiche Befehlssätze und Adressierungsarten
- Mächtige Befehle: komplexe Schritte in nur einem Befehl
- Variable Befehlslänge
- Bekannteste Vertreter: x86 CPUs
(diese bilden intern jedoch ab Pentium Pro auf RISC Befehle ab)

Reduced Instruction Set Computer (RISC)

- Wenige (einfache) Befehle und Adressierungsarten
- Feste Befehlslänge und –format
- Load/Store Architektur: Nur zwei Befehle die auf Speicher zugreifen.
- Große Registersätze
- Enge Kopplung zwischen Prozessor und Compiler: Compiler muss komplexe Aufgaben auf Elementarbefehle abbilden.
- Beispiele: PowerPC, SPARC, MIPS, ARM, MMIX, OpenRISC



Technologie

Landschaft der Prozessoren – Überblick (1/2)

Prozessortyp	Einsatzgebiet	Bemerkung
Intel Core i3, i5, i7	Desktop	Mainstream
Intel Xeon	Server/Desktop	
AMD Athlon, Sempron, Phenom	Desktop	Mainstream
AMD Opteron	Server/Desktop	
IBM PowerPC	MAC (bis 2006), Automotive, Settop	
HP/DEC Alpha	Alpha Systeme (bis 2007), Desktop, Supercomputer	Entwicklung eingestellt, Alpha Systeme auf Itanium umgestellt
SUN SPARC	Workstations	Ultra SPARC Open-Source: OpenSPARC



Technologie

Landschaft der Prozessoren – Überblick (2/2)

Prozessortyp	Einsatzgebiet	Bemerkung
Intel Atom	Mobile (Netbooks, Nettops, etc.)	Ultra-low voltage (ULV) x86 CPU
AMD Fusion / APU (z.B. C30, C50)	Mobile	Accelerated Processing Unit (APU): CPU+GPU
Transmeta Crusoe / Efficeon	Mobile	Code Morphing, 128-bit / 256-bit, VLIW CPU
Atmel (z.B. AVR, AVR32)	Diverse Controller	RISC CPU
Infineon TriCore, Automotive unified processor (AUDO)	Embedded/Automotive	TriCore: RISC CPU + MCU + DSP (Digitaler Signalprozessor)



CPU: Grundlagen

Befehlssatzarchitektur

Befehlssatz: Für (Assembler-)Programmierer „sichtbarer“ Teil der CPU

→ Beschreibung der **Befehlssatzarchitektur** umfasst

- Maschinenbefehlssatz
- Registerstruktur
- Adressierungsarten
- Interruptbehandlung

NICHT jedoch den internen Aufbau der CPU (Mikroarchitektur)



Befehlssatzarchitektur

Register und Registersätze

- Register sind schnellste speichernde Elemente eines Prozessors (wesentlich schneller als Speicherzugriff!)
- Unterscheidung von
 - Allgemein verwendbaren Registern (General Purpose Register, GPR)
 - Spezialregister (Special Purpose Register, SPR), z.B.:
 - Befehlszähler (Instruction Pointer)
 - Stackpointer
 - Statusregister (Flags)
 - Indexregister (für Adressberechnungen)

Programmiermodell: Befehlssatz + verfügbare Register



Befehlssatzarchitektur

Unterscheidungskriterien

1. Aufbau/**Anzahl der Operanden** bei typischen ALU-Instruktionen

Historisch:

- Ein-Adress**maschine**
- Zwei-Adressmaschine
- Drei-Adressmaschine

Heute üblicher ist Unterscheidung:

- Ein-Adress**befehle**
- Zwei-Adressbefehle
- Drei-Adressbefehle

2. Anzahl der Operanden, die **Speicheradressen sein dürfen**

0: Register-Register / Load-Store

1: Register-Memory

2 oder 3: Memory-Memory (heute nicht mehr genutzt)

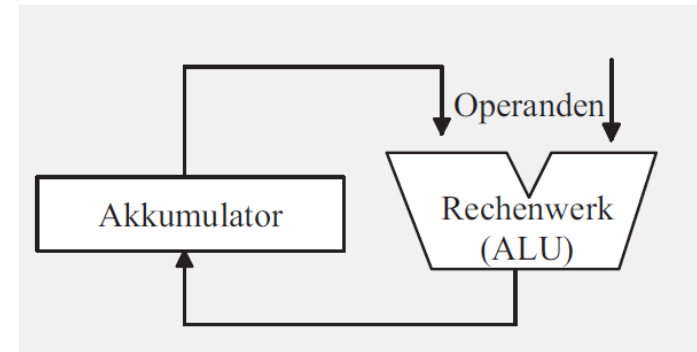
Beide Kriterien werden in den folgenden Folien näher betrachtet.



Befehlssatzarchitektur

Ein-Adress-Maschine / Akkumulatormaschine

- Nur ein Operand kann spezifiziert werden
- Linker Operand und Zielregister für Ergebnis implizit vorgegeben:
Akkumulator



- Einfache Hardware
→ schnelle Ausführung
- Geringer Speicherbedarf für einen Befehl
- Einheitliche Befehlslänge

- Programmierung umständlich / erfordert Übung
- Programme lang (Code-Größe) durch notwendiges Zwischenspeichern von Hilfsgrößen



Befehlssatzarchitektur: Ein-Adress-Maschine

Beispiel

Wie würde der Wert

$$y = \frac{x_1 + x_2 x_3}{x_1 - x_2}$$

berechnet, wenn MMIX eine Ein-Adress-Maschine wäre?

Nachahmen der Ein-Adress-Maschine unter MMIX:

- \$1 sei Akkumulator
- \$2 wird nur zum Holen der Operanden aus dem Speicher genutzt

1	x1	OCTA	3
2	x2	OCTA	7
3	x3	OCTA	11
4	Accu	IS	\$1
5	Main	LDO	Accu,x2
6		LDO	\$2,x3
7		MUL	Accu,Accu,\$2
8		LDO	\$2,x1
9		ADD	Accu,Accu,\$2
10		STO	Accu,x3
11		LDO	Accu,x1
12		LDO	\$2,x2
13		SUB	Accu,Accu,\$2
14		STO	Accu,x1
15		LDO	Accu,x3
16		LDO	\$2,x1
17		DIV	Accu,Accu,\$2
18		TRAP	0,Halt,0



Befehlssatzarchitektur

Zwei-Adress-Maschine

- Zwei Operanden können spezifiziert werden
- Linker Operand wird implizit als Ziel verwendet, d.h. Befehle sind von der Art

$$OP1 \leftarrow OP1 \otimes OP2$$

wobei \otimes für eine beliebige Verknüpfung steht.



- Geringerer Speicherbedarf für einen Befehl als bei Drei-Adress-Maschine
- Geringere Code-Größe als bei Ein-Adress-Maschine

- Linker Operand wird immer überschrieben → muss ggf. vorher gesichert werden



Befehlssatzarchitektur: Zwei-Adress-Maschine

Beispiel

Wie würde der Wert

$$y = \frac{x_1 + x_2 x_3}{x_1 - x_2}$$

berechnet, wenn MMIX eine Ein-Adress-Maschine wäre?

1	Main	MUL	x3,x3,x2
2		ADD	x3,x3,x1
3		SUB	x1,x1,x2
4		DIV	x3,x3,x1

Nachahmen der Zwei-Adress-Maschine unter MMIX:

- Linker Operand wird stets als Ziel benutzt



Befehlssatzarchitektur

Drei-Adress-Maschine

- Zwei Quellen und ein Ziel können angegeben werden



- Bequeme Programmierung
- Geringe Code-Größe

- In der Regel **keine Speicheradressen als Operanden** erlaubt.
Grund: Speicheradresse z.B. 64-Bit groß, bei 3 Operanden also Befehlsbreite > 192 Bit nötig

Beispiel: $y = \frac{x_1 + x_2 x_3}{x_1 - x_2}$

1	Main	SUB	h1,x1,x2
2		MUL	y,x2,x3
3		ADD	y,y,x1
4		DIV	y,y,h1



Befehlssatzarchitektur

Speicherbedarf von Zwei- und Drei-Adress-Befehlen

Ansätze um den Speicherbedarf zu reduzieren:

1. Flexibles Befehlsformat

- Befehlslänge abhängig vom Befehlscode (Op-Code)
- Auslesen von Befehlen aus dem Speicher in diesem Fall jedoch komplizierter
(Befehle nicht an Wortgrenzen ausgerichtet!)

Beispiele: CISC (Complex Instruction Set Computer) Prozessoren wie 80x86, 680x0, Transputer

2. Nur Register als Operanden

- Befehlslänge immer gleich, Operanden können jedoch nur aus Registern geholt und Ergebnisse nur in Register geschrieben werden
- Separate Befehle (Load/Store) für Transport zwischen Speicher und Registern

Beispiele: RISC (Reduced Instruction Set Computer) Prozessoren wie SPARC, PowerPC, MMIX, OpenRISC



Befehlssatzarchitektur

Stack-Maschine

- Stack (Stapelspeicher, Kellerspeicher) anstelle von Registern
- ALU Befehl ohne Operanden („Zero Operand Architecture“, also quasi eine „Null-Adress-Maschine“) – stattdessen liegen Operanden der ALU immer oben auf dem Stack



- Kurze Befehlslänge → kleine Programme
- Compiler einfach zu bauen
- Mit geringem Hardware-Aufwand realisierbar

- Langsame Ausführungsgeschwindigkeit, u.a. wegen weniger Flexibilität beim Pipelining (siehe nächstes Kapitel), großer Anzahl an Speicherzugriffen, Speicherung von Zwischenergebnissen

- Stack-Maschinen heute nur noch in virtuellen Maschinen (Java Virtual Machine, JVM), 80x86 Floating Point Unit



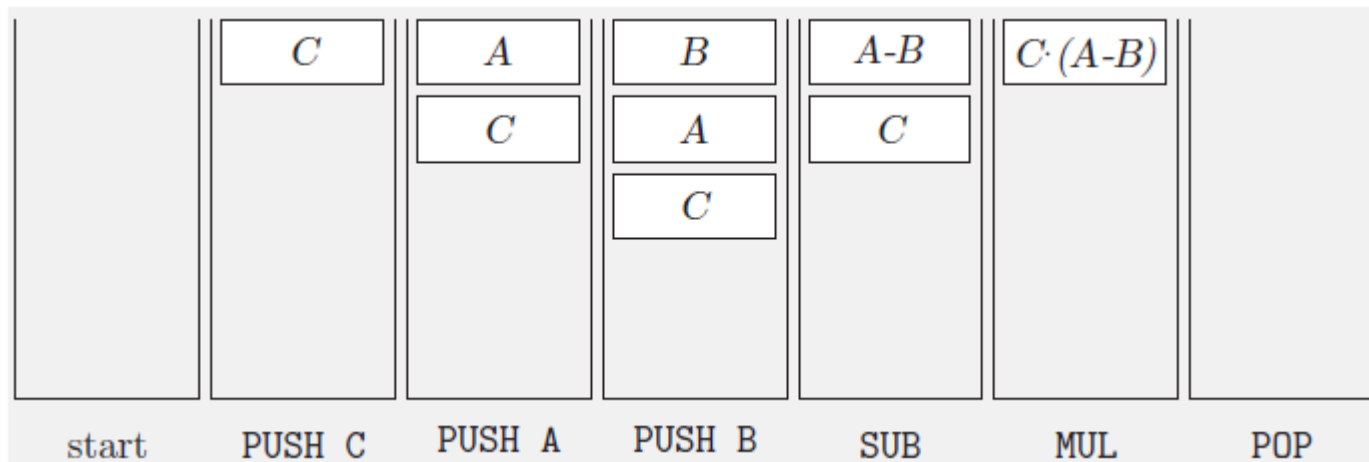
Befehlssatzarchitektur: Stack-Maschine

Beispiel

Wie würde der Wert

$$(A - B)C$$

auf einer Stack-Maschine berechnet und wie sieht der Stack nach jedem Teilschritt aus?



Befehlssatzarchitektur: Stack-Maschine

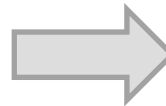
Beispiel Java

Java-Code: calc.java

```
public class calc {  
    public static void  
        main(String args[])  
        {  
            int a=3, b=4, c=5;  
            int y=(a-b)*c;  
  
            System.out.println("y="+y);  
        }  
}
```

Disassemblierter Java-Byte-Code:

javap -c calc



```
public static void main(java.lang.String[]);
```

Code:

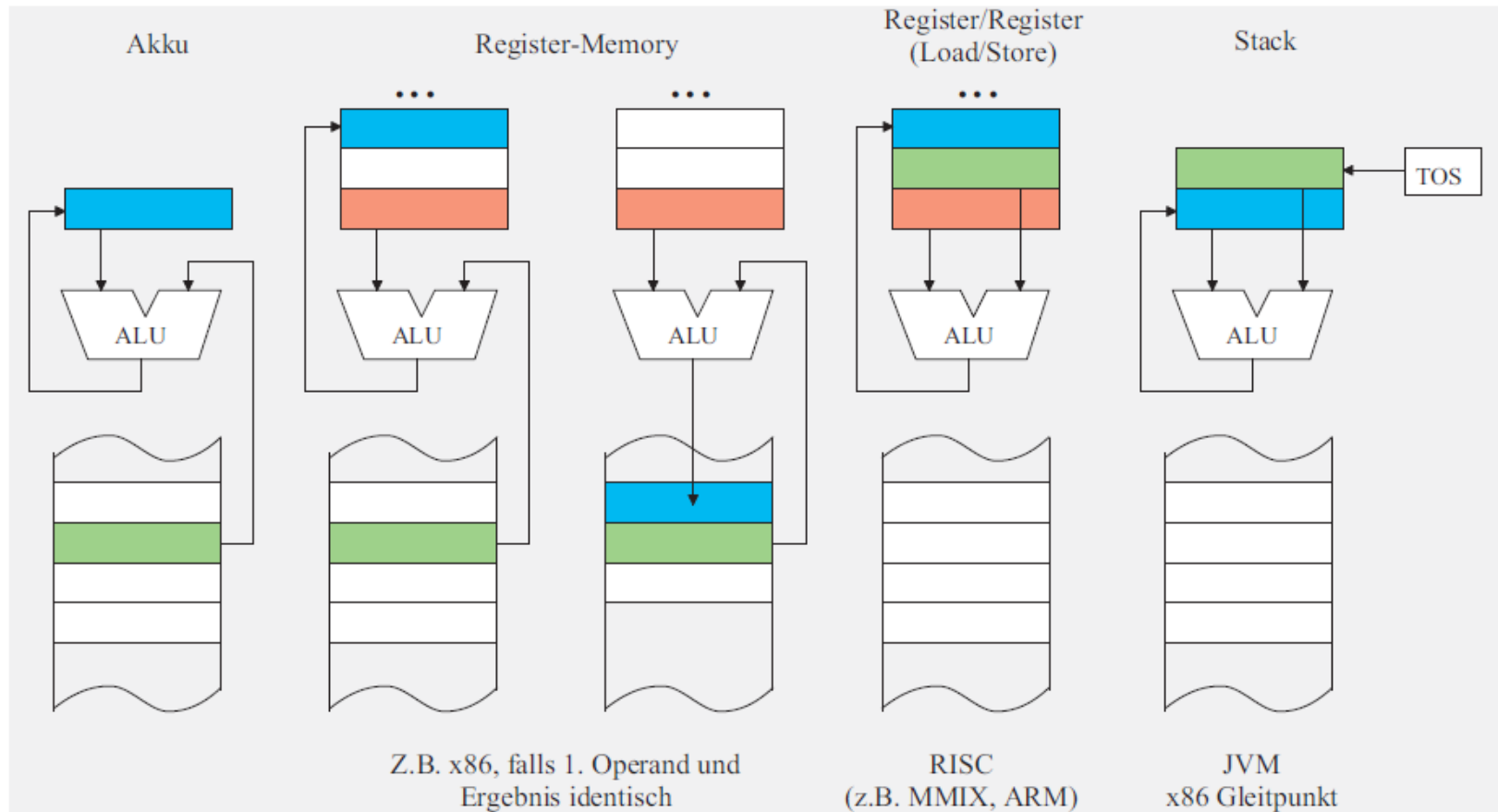
```
0: iconst_3  
1: istore_1  
2: iconst_4  
3: istore_2  
4: iconst_5  
5: istore_3  
6: iload_1  
7: iload_2  
8: isub  
9: iload_3  
10: imul  
11: istore          4  
13: getstatic       #2  
16: new             #3  
19: dup  
20: invokespecial   #4  
23: ldc             #5  
25: invokevirtual   #6  
28: iload          4  
30: invokevirtual   #7  
33: invokevirtual   #8  
36: invokevirtual   #9  
39: return
```

} entspricht
System.out.print...



Befehlssatzarchitektur

Register-Memory / Register-Register Architekturen



TOS: Top of Stack, JVM: Java Virtual Machine

Befehlssatzarchitektur

Befehlsarten

Grundsätzlich werden vier Arten von Befehlen unterschieden

1. Arithmetische und logische Befehle
2. Befehle zum Zugriff auf den Speicher (Load/Store)
3. Befehle zum Steuern des Programmablaufs
4. Ein- und Ausgabebefehle (I/O)

Aus den Vorlesungen der ersten Semester sind diese Großteils bekannt, wir konzentrieren uns daher auf Besonderheiten:

- SIMD-Befehle
- Bedingte Befehle
- Nicht-unterbrechbare (atomare) Befehle
- Spezielle Aspekte beim Speicherzugriff



Befehlsarten

SIMD-Befehle

Beobachtung: *Multimedia-Applikationen nutzen oft kürzere Datentypen (z.B. 16-Bit Audio Sample, 8 Bit pro Farbe im Bild) als die 32-Bit bzw. 64-Bit für die die Prozessoren optimiert sind.*

Idee: *Befehle, die mehrere kurze Datentypen (z.B. 8 Mal 8 Bit) gleichzeitig verarbeiten können.*

MMX Instruktionen (1996, Intel, x86 Architektur)

- Erste Umsetzung von SIMD in Standard-Prozessor
- Acht 64-Bit Register (logisch – nicht physisch – identisch mit den FP-Registern) mm0 bis mm7 enthalten jeweils
 - Acht unabhängige Bytes (Packed Bytes), oder
 - Vier unabhängige Wydes (Packed Words)
- Operationen: arithmetische Operationen (auch saturiert, d.h. ohne Überlauf!), Maskierung, bedingte Befehle, etc.



Befehlsarten

SIMD-Befehle

3DNow! (1998, AMD, K6-2)

- SIMD mit je zwei Gleitkommawerten

Streaming SIMD Extensions (SSE, 1999, Intel, x86)

- Separate, 128-Bit breite Register für SIMD
- Ganzzahl- und Gleitkommaarithmetik
- Erweiterungen: SSE2 (2001), SSE3 (2004), SSE4 (2007)

Advanced Vector Extensions (AVX, 2010, Intel, x86)

- Breite der SIMD Register von 128-Bit auf 256-Bit vergrößert (Erweiterung auf 512-Bit und 1024-Bit zukünftig möglich)

Generell: Nutzung der Erweiterungen zunächst nur über spezielle Bibliotheken, heute teilweise auch schon automatisch über Compiler.

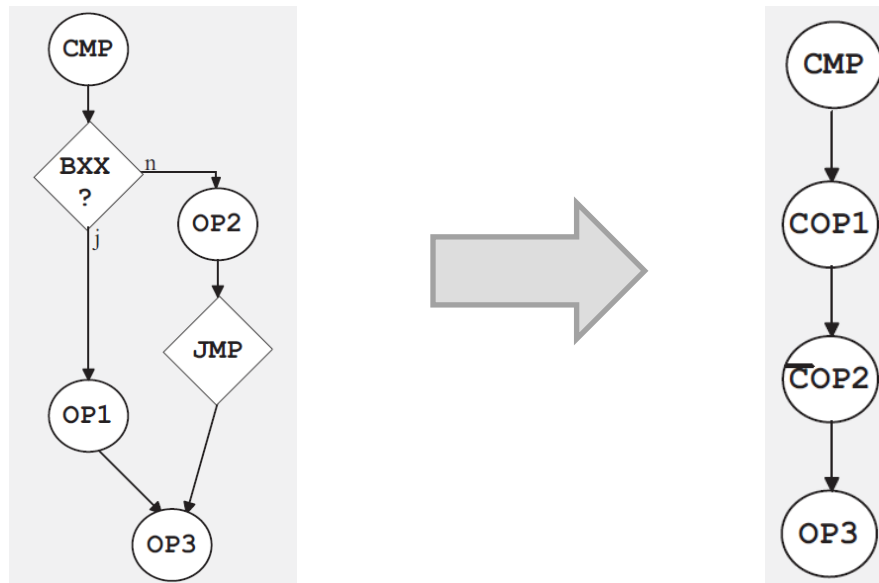


Befehlsarten

Bedingte Befehle

Problem: *Bedingte Sprünge erschweren die Verarbeitung in CPU*
(→ Pipelining, Kap. 4)

Idee: Vermeidung von Sprüngen durch Befehle, die nur unter bestimmten Bedingungen ausgeführt werden.



Befehlsarten

Bedingte Befehle – Beispiele

- **MMIX** hat die Befehle „Conditional Set“/“Zero or Set“
CSxx \$X, \$Y, \$Z $\$X \leftarrow \Z , falls \$Y die Bedingung xx erfüllt
ZSxx \$X, \$Y, \$Z $\$X \leftarrow \Z , falls \$Y die Bedingung xx erfüllt; 0 sonst
- **ARM**
Jede Instruktion hat 4-Bit Condition-Field: Gibt an, von welchen Flags die Ausführung des Befehls abhängt, z.B.:
MOV MI: „Move if Minus“
- **x86**
Seit Pentium Pro (P6): „Conditional Move“
CMOVxx Reg, Reg oder CMOVxx Reg, Mem
- **OpenRISC 1000**
„Conditional Move“
l.cmov rD, rA, rB



Befehlsarten

Nicht-unterbrechbare Befehle (atomare Operationen)

- Zur Thread-/Prozesssynchronisation, z.B. für Semaphore, werden atomare Operationen benötigt (Test-and-Set / Read-Modify-Write)
 1. Bedingung testen
 2. Variable setzen
- Können durch Interrupts nicht unterbrochen werden

Einschub: Sperrvariable und Semaphore

Sperrvariable (spin lock):

Sicherstellung, dass sich nur ein Thread in kritischem Bereich befindet. Nur wenn Wert 1, darf Bereich betreten werden und wird auf 0 gesetzt. Bei Verlassen wird Variable wieder auf 1 gesetzt.

Semaphore: Sicherstellung, dass Ressource/kritischer Bereich nur von maximal N Threads genutzt wird. Elementaroperationen: $P(s)$ prüft und verringert Ressourcenzähler, $V(s)$ erhöht nach Ressourcenzähler (Freigeben).



Befehlsarten

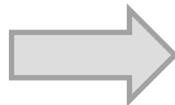
Nicht-unterbrechbare Befehle – Beispiele

MMIX hat „Compare and Swap“

CSWAP \$X, \$Y, \$Z

- Vergleicht Octabyte an Adresse \$Y+\$Z mit Prediction-Register rP.
- Falls beide Werte gleich sind, wird \$X an Adresse \$Y+\$Z gespeichert und zu 1 gesetzt.
- Andernfalls wird Speicherinhalt an Adresse \$Y+\$Z in rP geladen und \$X zu 0 gesetzt.

Anwendungs-
Beispiel:



1		LOC	Data_Segment	
2	SEMA	OCTA	1	Semaphorvariable
3				
4	semReg	IS	\$1	
5		LOC	#100	
6	Main	PUT	rP,1	SEMA soll eins sein
7		SET	semReg,0	Ziel: SEMA soll 0 werden
8		CSWAP	semReg,SEMA,0	
9		BZ	semReg,wait	
10		TRAP	0	



Befehlsarten

Nicht-unterbrechbare Befehle – weitere Beispiele

ARM

SWP Rd, Rm, [Rn]

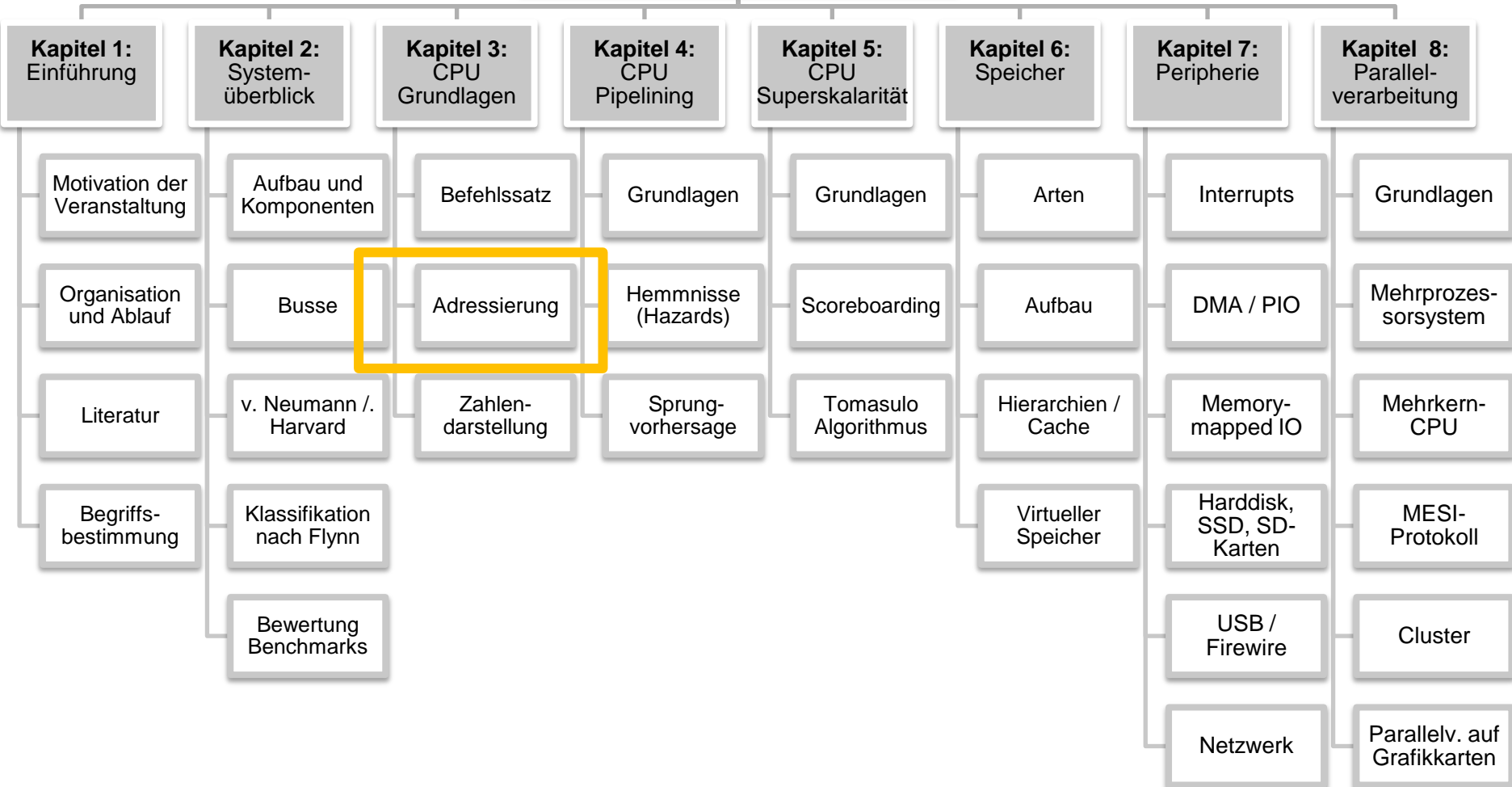
x86

CMPXCHG, XADD

BTC, BTS, BTR (auf Bitebene)



Rechnerarchitektur



Speicheradressierung

Interpretation von Adressen: Endianness

Was gibt das folgende Programm auf der Console aus?

```
#include <stdio.h>
```

```
void main(){  
    long long ll = 0X123456789abcdef;  
    unsigned char* pc = (unsigned char*)&ll;  
    int i;  
  
    for (i=0; i<8; i++)  
        printf( "%02hx ", *(pc + i) );  
}
```



Speicheradressierung

Interpretation von Adressen: Endianness

- Aktuelle CPU erlaubt Zugriff auf Datentypen mit 8, 16, 32, 64 Bit

Frage: *In welcher Reihenfolge sind die einzelnen Bytes gespeichert?*

- Es gibt zwei Alternativen, beide werden in der Praxis verwendet!

Big Endian (verwendet u.a. bei MMIX, Motorola, Netzwerkübertragung)

- Byte mit höchstwertigsten Bits an kleinster Speicheradresse
- Ausgabe (Beispiel vorheriger Folie): 01 23 45 67 89 ab cd ef

Little Endian (verwendet u.a. bei Intel Prozessoren)

- Byte mit höchstwertigsten Bits an größter Speicheradresse
- Ausgabe (Beispiel vorheriger Folie): ef cd ab 89 67 45 23 01

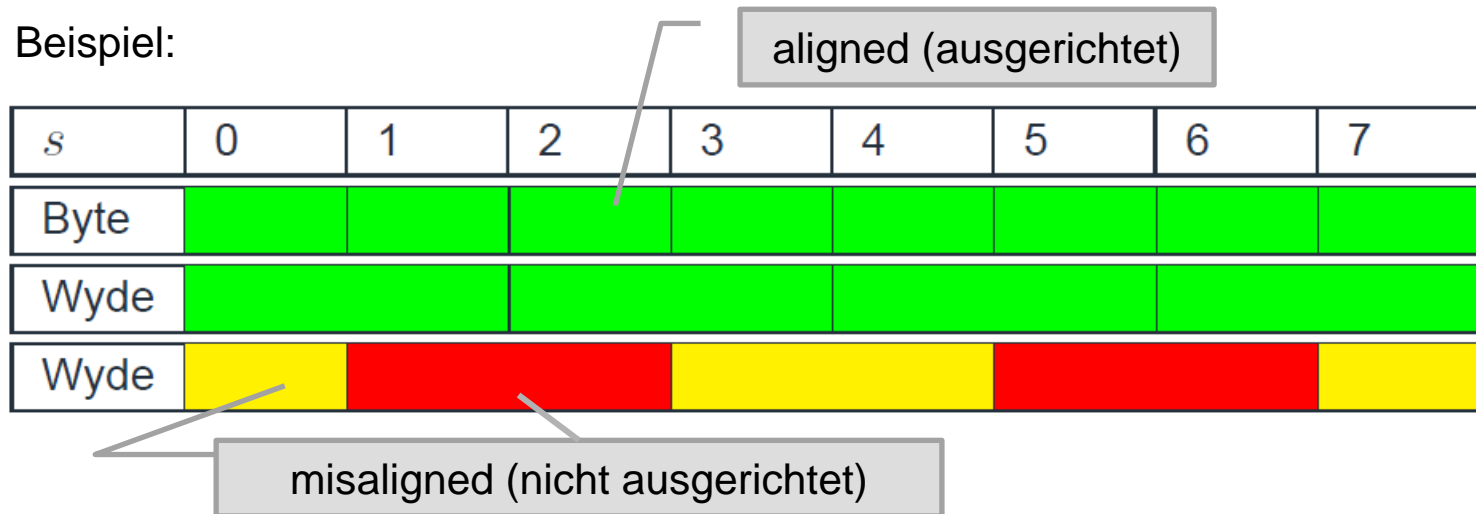


Speicheradressierung

Alignment

- Zugriff auf den Speicher erfolgt in Blöcken von z.B. 8 Bytes
→ Ablage im Speicher sollte an diesen Blöcken ausgerichtet sein
- **Alignment** (Speicherausrichtung):
Der Zugriff auf ein Speicherobjekt mit $s = 2^b$ Bytes ($b \geq 0$) an Adresse A ist ausgerichtet (aligned), falls $A \bmod s = 0$.

Beispiel:



Speicheradressierung

Alignment

Beispiel (Fortsetzung):

s	0	1	2	3	4	5	6	7				
Tetra												
Tetra												
Tetra												
Tetra												
Octa												
Octa												
Octa												
Octa												
Octa												
Octa												
Octa												



Speicheradressierung

Allgemeine Adressierungsarten

Grundsätzlich drei Möglichkeiten, **woher Operanden** stammen oder **wohin Ergebnis** eines Befehls geschrieben werden kann:

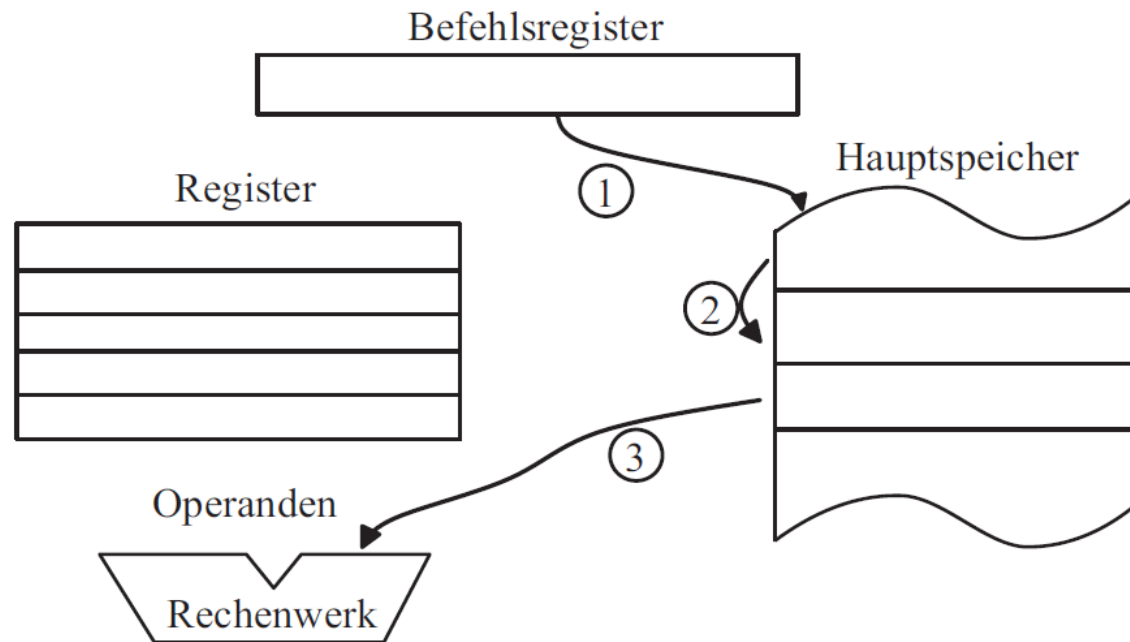
1. Aus dem **Befehlswort**
→ Konstantenadressierung / Immediate Adressierung
2. Aus einem **Register**
→ im Befehl direkt oder indirekt angegeben
3. Aus dem **Hauptspeicher**
→ Speicheradresse auf die letztendlich zugegriffen wird heißt *Effektive Adresse*



Speicheradressierung

Zweistufige Speicheradressierung

- Zur Bildung der effektiven Adresse wird Adresse aus Speicher geladen
→ Speicherindirekte Adressierung
- Als Adressierungsart nicht mehr direkt im Prozessor implementiert
→ muss von Programm übernommen werden



Speicheradressierung

Zweistufige Speicheradressierung – Beispiel

Einlesen von Zeichenketten mit MMIX:

1		LOC	Data_Segment	
2	Buffer	BYTE	0	
3		LOC	Buffer+80	Puffer anlegen
4	* Argumentbereich:			
5	Arg	OCTA	Buffer	Adresse des Puffers
6		OCTA	80	Puffergröße
7				
8		LOC	#100	
9	Main	LDA	\$255,Arg	Adresse Argumentbereich
10		TRAP	0,Fgets,StdIn	Einlesen
11		TRAP	0,Halt,0	



Speicheradressierung

Adressierungsarten (1/2)

Adressen können auf unterschiedliche Arten angegeben werden – diese Tabelle (aus [1]) fasst die gebräuchlichen Arten zusammen:

Bezeichnung	MMIX	Sonstige/Intel	Beschreibung
Konstanten-adressierung	ADD \$1,\$1,10	ADD R1,10	Direktooperand im Befehlsword
Registerdirekte Adressierung	ADD \$0,\$1,\$2	ADD R1,R2	Wert wird direkt aus Register gelesen
Registerindirekte Adressierung	≈ LDO \$0,\$255,0	MOV reg1,[reg2]	Effektive Adresse in einem Register
Absolute Adressierung	nicht verfügbar	MOVE reg1, Mem	Effektive Adresse Teil des Befehls
Speicherindirekte Adressierung	nicht verfügbar	MOVE reg1, @[reg2]	Effektive Adresse aus Speicher an im Register angegebener Position



Speicheradressierung

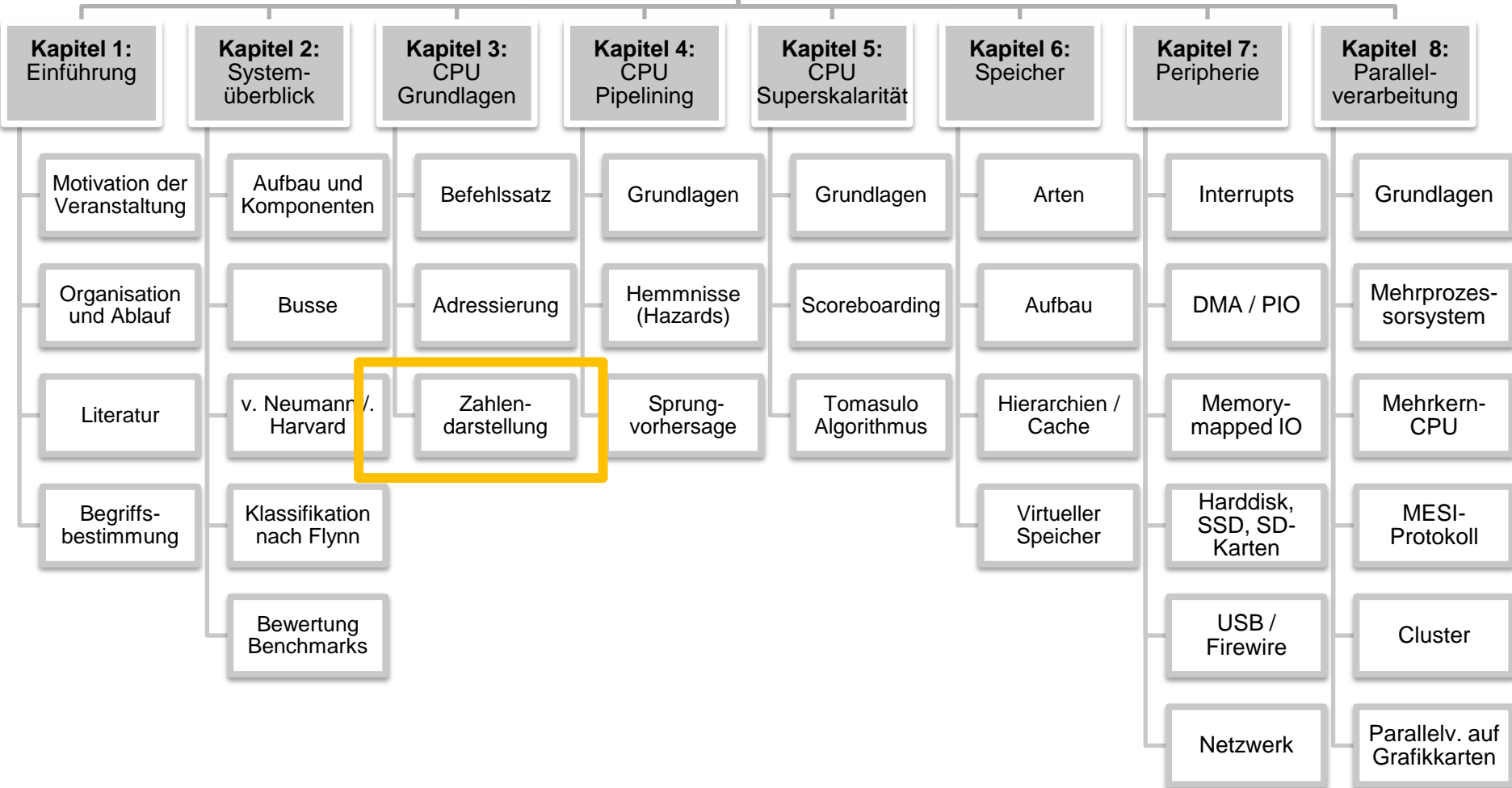
Adressierungsarten (2/2)

(Fortsetzung von vorheriger Folie)

Bezeichnung	MMIX	Sonstige/Intel	Beschreibung
Indiziert Register-relative Adress.	LDO \$0,\$255,\$1	MOVE reg1, [reg2+reg3]	Effektive Adresse Summe d. Register
Indiziert Register-relative Adressierung mit Index / Displacement	LDO \$0,\$255,8	MOVE reg1, [reg2+Off]	Effektive Adresse ist Summe Register + Konstante
Programmzähler relative Adressierung	JMP @+12	MOVE R1, [PC,offset]	Effektive Adresse ist Summe Konst. + Programmzähler



Rechnerarchitektur



Zahlendarstellung

Überblick

Ganze Zahl

- Ohne Vorzeichen → klassische Dualzahl, z.B.: 1111 1111 für 255 bei 8-Bit
- Mit Vorzeichen → Zweierkomplement: negative Zahl dargestellt durch Negation aller Bits und anschließende Addition von 1

Fixkomma-Zahl

- Wie ganze Zahl jedoch mit fester Position eines Kommas (Beispiel siehe nächste Folie)

Gleitkomma-Zahl (Floating Point Number)

- Darstellung als Kombination von Mantisse und Exponent



Zahlendarstellung

Fixkomma-Zahl

- Feste Anzahl an Bits für den Teil vor dem Komma, restliche Bits für die Nachkommastellen
- Zweierkomplementdarstellung sinnvoll
- Mögliches Beispiel bei 8-Bit:
4-Bit vor dem Komma, 4-Bit danach → Multiplikation mit 1/16

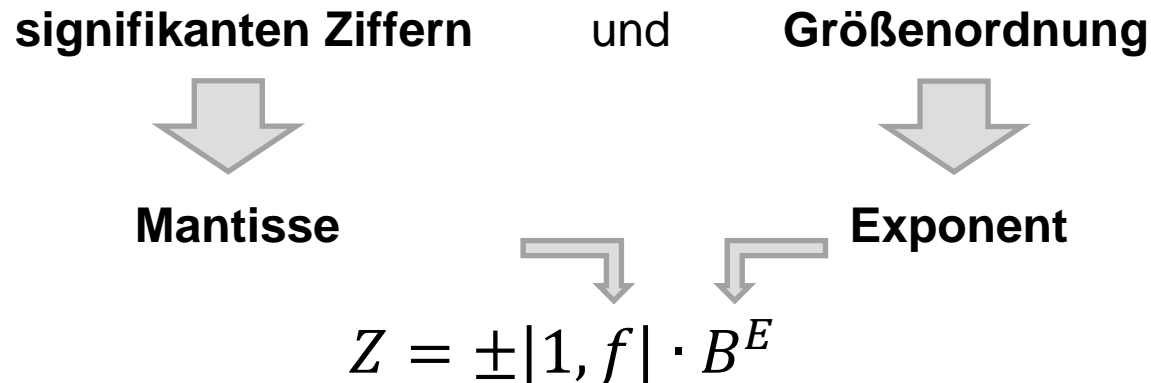
0000	0001	entspricht	$1/16$	$= 0,0625$
0000	1000	entspricht	$8/16$	$= 0,5$
0010	0000	entspricht	$32/16$	$= 2$
1111	0000	entspricht	$-16/16$	$= -1$



Gleitkommazahlen

Ansatz

Ziel: Abdeckung eines großen Zahlenraumes mit ausreichender **Genauigkeit** durch Speicherung von



- Eine Zahl wird also dargestellt durch das Vorzeichen, die Mantisse f mit $1 \leq 1, f < 2$ und einen Exponenten E .
- Die Basis $B = 2$ im Binärsystem sowie die bei f stets auftretende Eins vor dem Komma muss nicht gespeichert werden.



Gleitkommazahlen

Standard IEEE 754 für 64-Bit Zahlen (1/4)

1 Bit v für **Vorzeichen** (gesetztes Bit bedeutet: negatives Vorzeichen)

11 Bit für den **Exponenten**.

Exponent E wird in vorzeichenloser Form e gespeichert, indem man den Exzess $q = 2^{11-1} - 1 = 1023$ addiert: $e = E + q = E + 1023$

52 Bit für die **Nachkommastellen** des Betrags des gebrochenen Anteils f (Mantisse)



Gleitkommazahlen

Standard IEEE 754 für 64-Bit Zahlen (2/4)

Beispiel 1: $1,0 = \#3FF0\ 0000\ 0000\ 0000$


Beispiel 2:

Umwandlung der Dezimalzahl 8,25 zur Darstellung nach IEEE 754:


1. **Vorzeichen** positiv $\rightarrow v=0$

2. Umwandlung von 8,25 in **Binärdarstellung**

a. Anteil vor dem Dezimalkomma in Binärdarstellung:

$8 / 2 = 4$	Rest	0		(LSB)	$\rightarrow 1000$
$4 / 2 = 2$	Rest	0			
$2 / 2 = 1$	Rest	0			
$1 / 2 = 0$	Rest	1		(MSB)	

b. Anteil nach Dezimalkomma in Binärdarstellung

$0,25 * 2 = 0,5$	-	0		(MSB)	$\rightarrow 01$
$0,50 * 2 = 1$	-	1		(LSB)	

Insgesamt: 1000,01 (binär)



Gleitkommazahlen

Standard IEEE 754 für 64-Bit Zahlen (3/4)

Beispiel 2 (Fortsetzung):

Umwandlung der Dezimalzahl 8,25 zur Darstellung nach IEEE 754:

3. Umwandlung in **normalisierte Darstellung**

„Verschieben des Kommas“:

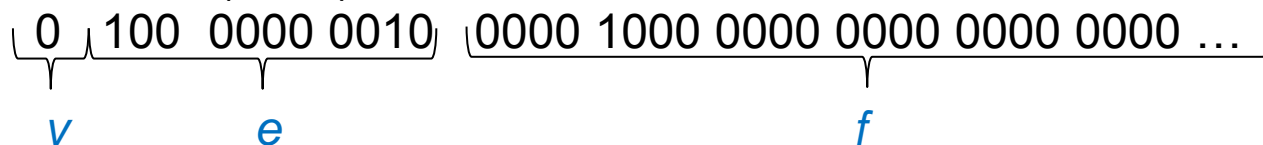
$$1000,01 \text{ (binär)} = 1,00001 \cdot 2^3$$

4. **Exponenten E** bestimmen und Exzess addieren:

$$E = 3$$

$$e = E + 1023 = 1026 = 100\ 0000\ 0010 \text{ (binär)}$$

5. Gesamt (binär)



6. Umwandlung in **Hexadezimaldarstellung** (4 Bit entsprechen jeweils einer Ziffer): #4020 8000 0000 0000



Gleitkommazahlen

Standard IEEE 754 für 64-Bit Zahlen (4/4)

Sonderfälle

- $e = 0$ und $f = 0$ steht für **Null**.
Es wird zwischen positiver und negativer Null unterschieden (je nach Vorzeichen v).
- $e = 0$ und $f > 0$ steht für einen **denormalisierten Wert**:
In diesem Fall wird vor der Mantisse nicht wie sonst eine „1,“ angenommen sondern „0,“. So kann die Lücke zwischen 0 und 1 in der normalisierten Darstellung geschlossen werden.
- $e = 2047$ und $f = 0$ steht für **unendlich**
(je nach v : $+\infty$ bzw. $-\infty$)
- $e = 2047$ und $f > 0$ bedeutet „**keine Zahl**“
NaN (Not a Number)

e	f	Bedeutung
0	0	Null
0	>0	Denormalisierter Wert
2047	0	Unendlich
2047	>0	Keine Zahl / Not a Number



Danksagung und Quellen

- Dieser Foliensatz basiert inhaltlich in großen Teilen auf einem älteren von Prof. Axel Böttcher, Hochschule München, entwickelten Foliensatz zur Rechnerarchitektur sowie dem entsprechenden Buch [1].
- Sämtliche Fehler im Foliensatz hingegen entstammen meiner Feder – falls Sie Fehler finden, bin ich Ihnen für einen kurzen Hinweis dankbar.
- Eine Liste weiterer Quellen finden Sie im Abschnitt „Empfohlene Literatur“ des Foliensatzes zu Kapitel 1.

