Datenverarbeitungssysteme

Technische Informatik
Sommersemester 2016

Prof. Dr. Wolfgang Mauerer
wolfgang.mauerer@oth-regensburg.de

Inhalt

Dozent

OTH Regensburg

- Professor für theoretische Informatik
- Forschungsinteressen
  - Innovation durch Theorie (Zufallserzeugung, stochastische Methoden)
  - Industrie 4.0 (*Cyber Physical* und *Smart Embedded*)
  - Statistik und maschinelles Lernen in der SW-Entwicklung
- Hacken *und* Beweisen

Siemens Corporate Research

- Linux (Kernel, Low-Level)
- Harte Echtzeit (MRT, Simatic, Àndroit, …)
- Statistik und maschinelles Lernen in der SW-Architektur `siemens.github.com/codeface`
- Früher: MPL (QIT, QED)

Scheinkriterien/Übungen

Scheinkriterien

- Klausur (90min) am Ende des Semesters
- Keine Hilfsmittel
- Material der gesamten Vorlesung relevant (orientiert sich an Übungsaufgaben).

Übungen

- Gruppe 1: Fr, 11:45-13:15, U514 (Herr Kölbl)
- Gruppe 2: Fr, 11:45-13:15, U521 (Herr Welker)
- Gruppe 3: Fr, 13:30-15:00, U514 (Herr Kölbl)

Aufwand

- 4 SWS Vorlesung, 2 SWS Übungen
- ≈ 90h Vorlesung und Übungen, ≈ 150h Eigenstudium

Informationsquellen

Informationsquellen

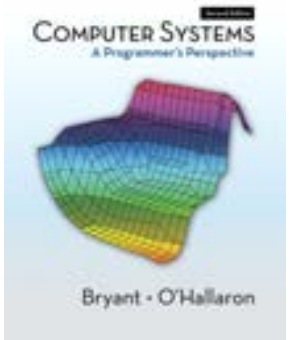
- Angegebene Literatur
- Herstellerdokumentation von ARM
- Weitere im G.R.I.P.S. verlinkte Dokumente

Technische Details

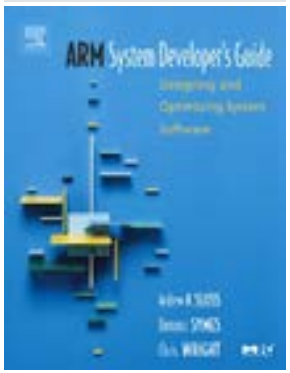
- …werden/können nicht vollständig in der Vorlesung abgehandelt werden
- Eigeninitiative!
- *Notwendige Fähigkeit* für industrielle Tätigkeiten

Literatur

- R. E. Bryant und D. O'Hallaron, *Computer Systems*, 2ed, Pearson, 2010.



- A. N. Sloss, D. Symes, C. Wright, *ARM System Developer's Guide*, Morgan Kaufmann, 2004.



- W. Mauerer, *Professional Linux Kernel Architecture*, Wiley/Wrox, 2009.

Lernziele

- Struktur von Datenverarbeitungssystemen
- Funktionsweise und Implementierung von CPUs
- Ganzzahl- und Gleitkommazahldarstellung und -arithmetik
- Assemblerprogrammierung (32-Bit ARM)
- Speicherhierarchie; RAM und Caches
- Massenspeichergeräte
- Arbeitsweise von Compilern, Bindern und Ladern
- Grundlagen von Betriebssystemen
- Grundlagen von Netzwerken

Spezialvorlesungen

- Ziel: *Technik-Stapel* verstehen!
- Vertiefung und Details: Spezialvorlesungen

DV-Systeme: Theorie und Praxis

Theorie

- Abtraktes Maschinenmodell
- Details bewusst ignoriert/abwesend
- Modellbildung

Reale Systeme

- *Nicht* das Gegenteil der Theorie
- Abstraktion nicht immer möglich
- Einzelschichten einfach bis trivial, Kombination hochkomplex

Bugs und Optimierungen

- Häufig querschneidend
- Verständnis Gesamtsystem (nicht: ausschalten–einschalten)

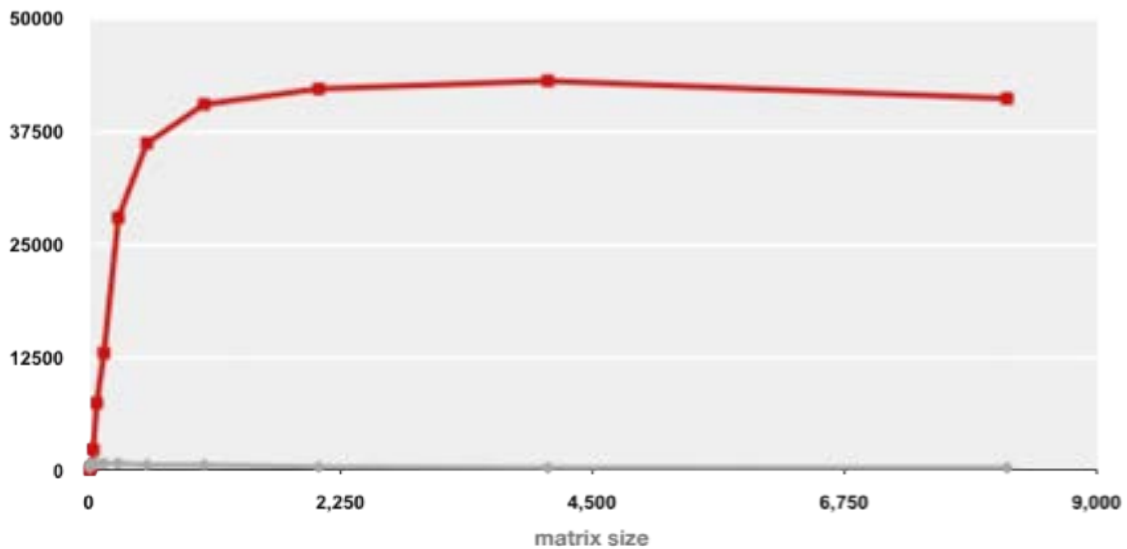
Reale Welt I

- Low-Level-Software, Gerätetreiber etc.: Entkopplung von HW unmöglich
- Konstanten und Faktoren in asypmtotischer Notation: *Praktische Umsetzbarkeit*
- Effizienz: 50 Cent weniger bei 10^6 Geräten ☞ Substantieller Unterschied
- Random Access Memory: Unrealistische Annahme

Reale Welt II

Algorithmus und Implementierung

- Optimaler Algorithmus bedeutet nicht optimale Leistung!
- Implementierung ist entscheidender Faktor
- Systemeigenschaften müssen genutzt werden



Reale Welt III

Computer kommunizieren

- Ein- und Ausgabe theoretisch häufig »wegabstrahierbar«
- Praktisch: Zahlreiche Sonderfälle
  - Interne und externe Kommunikation
  - Zuverlässigkeit und Leistung
  - (Teilweise) defekte Medien
  - Performance: Vielschichtiges Zusammenspiel

Lernziel II

Zerlegen statt neu bauen

- Seltene Fälle
  - Neue CPU entwerfen
  - Neue Busarchitektur entwerfen
  - Neues Betriebssystem schreiben
  - Neues Kommunikationsprotokoll entwerfen
  - Assembler schreiben
- Häufige (industrielle) Fälle
  - Riesige Mengen an Code verstehen
  - Betriebssysteme erweitern
  - CPUs »bis zum letzten Bit« ausnutzen
  - Assembler lesen
  - Systemumspannende Software verstehen und erweitern
- Hacken: Kreative und technisch tiefe Operationen am System
- Hacken ≠ Cracken!

Hardware/BYOD

Zielarchitektur

- ARM-Prozessoren: Raspberry Pi
- …zahllose Handys und Tablets
- Typischer Embedded-Prozessor
- Marktanteil typischer Desktop-Prozessoren (x86) verschwindend gering

Plattformen

- Physikalischer Raspberry Pi
- Virtueller rpi (auf qemu-Basis) – CIP-Pool
- Eigenes Gerät: Android/Cyanogenmod (eigene Verantwortung!)

Inhalt

Zahlenkodierung I

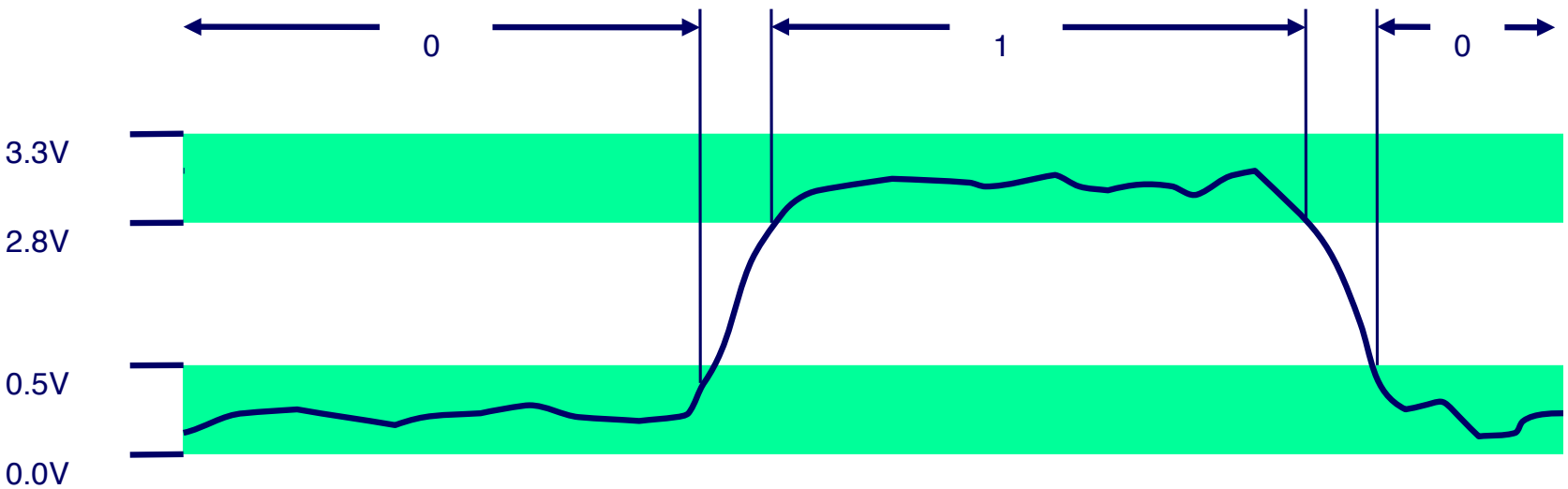


Abbildung basiert auf Bryant & O'Hallaron

Datenrepräsentation und -manipulation

Informationskodierung

- 10 Finger: Dezimalsystem (10 Ziffern)
- Computer: Systematischer
- Elektrisches Signal (ein/aus, geladen/nicht geladen, …)
- 2 Zustände als Basis: Bit (0/1)
- Bitfolgen zur Kodierung von Zahlen, Zeichen, Bildern, …

Zahlendarstellung

- Speicheraufteilung in Blöcke: Bytes (8 Bits), Words (16/32/64 Bits)
- Zugriffe auf Speicher erfolgt immer in Byte- oder größeren Schritten (bis auf pathologische Ausnahmen, bsp. TI TMS34010)
- Folge von Bits: Natürliche Zahl $x \in \mathbb{N}$
- Interpretation nach Stellenwertsystem

Zahlenkodierung II

- Dezimalsystem: Basis $b = 10$

$$\begin{aligned} 1026_{10} &= 1 \cdot 1000 + 0 \cdot 100 + 2 \cdot 10 + 6 \cdot 1 \\ &= 1 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 6 \cdot 10^0 \\ &= \sum_{i=0}^{N} x_{(i)} \cdot 10^i \end{aligned}$$

- Binärsystem: Basis $b = 2$.

$$\begin{aligned} 10110_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= \sum_{i=0}^{N} x_{(i)} \cdot 2^i \\ &= 22_{10} \end{aligned}$$

- Ist $b = 1$ (unäre Kodierung) sinnvoll?

Zahlenkodierung III

Kodierung

- Gebräuchliche Basen: 2 (Binär), 8 (Oktal), 10 (Dezimal), 16 (Hexadezimal)
- Jedes $x \in \mathbb{N}$ ist in jeder Basis eineindeutig darstellbar (ggf. Nullen von links entfernen: $00101_2 = 101_2$). Voraussetzung: Unendlich viele Bits verfügbar.

Hexadezimalzahlen

- Basis $b = 16$: 10 Ziffernsymbole nicht ausreichend
- Erweiterung: Buchstaben als zusätzliche Ziffernsymbole
- $A_{16} \equiv 10_{10}$, $B_{16} = 11_{10}$, …, $F_{16} = 15_{10}$.
- Einzelne Ziffer deckt vier Bits ab. Kompakte Schreibweise für Kombinationen mehrerer Bytes

Zahlenkodierung IV

Beispiel

$$\begin{aligned} \mathrm{BEEF}_{16} &= \mathrm{B} \cdot 16^3 + \mathrm{E} \cdot 16^2 + \mathrm{E} \cdot 16^1 + \mathrm{F} \cdot 16^0 \\ &= 11 \cdot 16^3 + 14 \cdot 16^2 + 14 \cdot 16^1 + 15 \cdot 16^0 \\ &= 48897_{10} \end{aligned}$$

Rechnen mit Binärzahlen I

Addition

- Binär: $0 + 0 = 0$,
  $0 + 1 = 1 + 0 = 1$, $1 + 1 = 10$
- Stellenweise Addition mit Übertrag (Beispiel: Siehe Tafel)
- Problem: Endliche Bit-Anzahl.
  Beispiel 4 Bits:
  $1000_2 + 1000_2 = 10000_2$ wird auf 0000_2 »gestutzt«.
- Überlauf!

speed.c

```
uint8_t speed = 240;
for (uint8_t i = 0; i < 30;
     i++) {
     // Beschleunigen
     speed += 1;
     printf("Auto faehrt mit
     %u km/h\n", speed);
}
```

Rechnen mit Binärzahlen II

Probleme

- Begrenzter Wertebereich. Maximale Größen der verwendeten Daten sind essentiell!
- Subtraktion: Wie negative Resultate repräsentieren?
- Division: Resultat nicht notwendigerweise $\in \mathbb{N}$.

Multiplikation

Schulverfahren – siehe Tafel.

Zahlendarstellung I: Zweierkomplement

Negative Ganzzahlen repräsentieren

- Naïv (bzw. *Sign-Magnitude*): Höchstes Bit als Vorzeichen verwenden (0 ≡“+”, 1 ≡ “-”):

$$1101 = -5$$
$$0101 = +5$$

- Problem: Zwei Kodierungen für die Zahl 0.
  - $1000_2 = -0$
  - $0000_2 = +0$
- Mathematisch: $+0 = -0$. ☞ Bitverschwendung!

Zahlendarstellung II: Zweierkomplement

Zweierkomplementdarstellung

- Höchstwertiges *Bit* wird mit *negativem Gewicht* interpretiert
- Sei $\vec{x}$ eine Folge von 8 Bits (als Vektor interpretiert):
  - Ganzzahliger Wert: $\mathrm{b2u}_8(\vec{x}) = \sum_{i=0}^{7} x_i 2^i$
  - Zweierkomplement: $\mathrm{b2t}_8(\vec{x}) = \sum_{i=0}^{6} x_i 2^i - x_7 2^7$
- Jede darstellbare Zahl $x \in [-2^7, 2^7 - 1]$ eineindeutig (bijektiv) identifiziert
- Achtung: Asymmetrisches Intervall

Zahlendarstellung II

Zweierkomplement bei 8 Bits

Binär	Zweierkomplement	Vorzeichenlos
00000000	0	0
00000001	1	1
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
10000010	-126	130
11111110	-2	254
11111111	-1	255

Wert des Speicherinhalts ist immer interpretationsabhängig!

Zahlendarstellung III

Zweierkomplement – allgemein

- Formal:

$$\mathrm{b2t}_w : \mathbb{F}_2^w \to \mathbb{Z}$$

$$\mathrm{b2t}_w(\vec{x}) = \sum_{i=0}^{w-2} x_i 2^i - x_{w-1} 2^{w-1}$$

- Umkehrung: $\mathrm{t2b}_w : \mathbb{Z} \to \mathbb{F}_2^w$ mit $\mathrm{t2b} = \mathrm{b2t}^{-1}$
- Wohldefiniert, da b2t bijektiv

Zahlendarstellung IV

Typumwandlung

- Positive Werte: Gleiches Bitmuster
- Allgemein
  - Signed → Unsigned: $\mathrm{t2u}_w(x) \equiv \mathrm{b2u}_w(\mathrm{t2b}_w(x))$
  - Unsigned → Signed: $\mathrm{u2t}_w(x) \equiv \mathrm{b2t}_w(\mathrm{u2b}_w(x))$
- Bitmuster beibehalten, Inhalt neu interpretieren
- CPUs verwenden üblicherweise gleiche Elementarinstruktionen für signed und unsigned-Arithmetik

Zahlendarstellung V

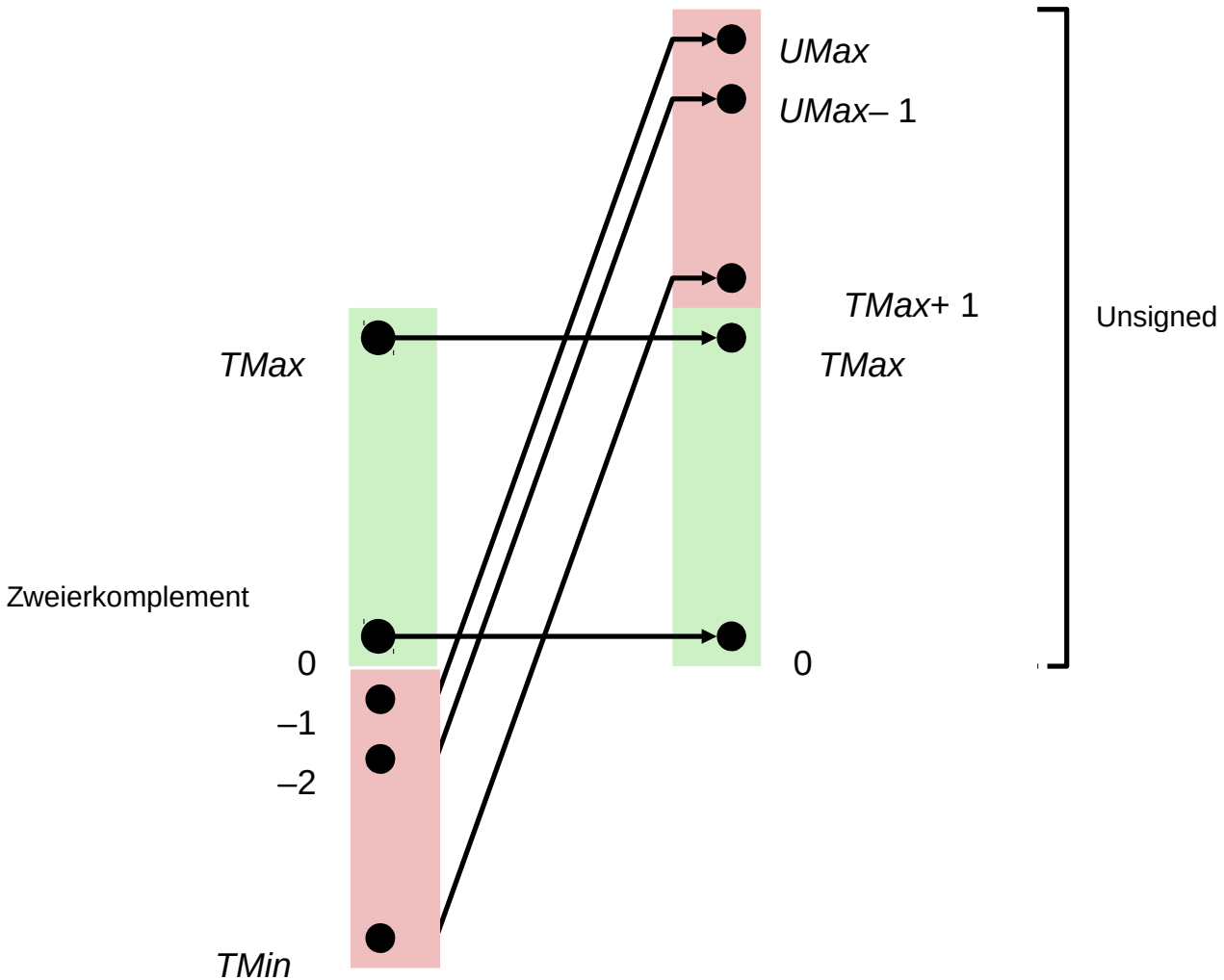


Abbildung basiert auf Bryant & O'Hallaron

Sign Extension I

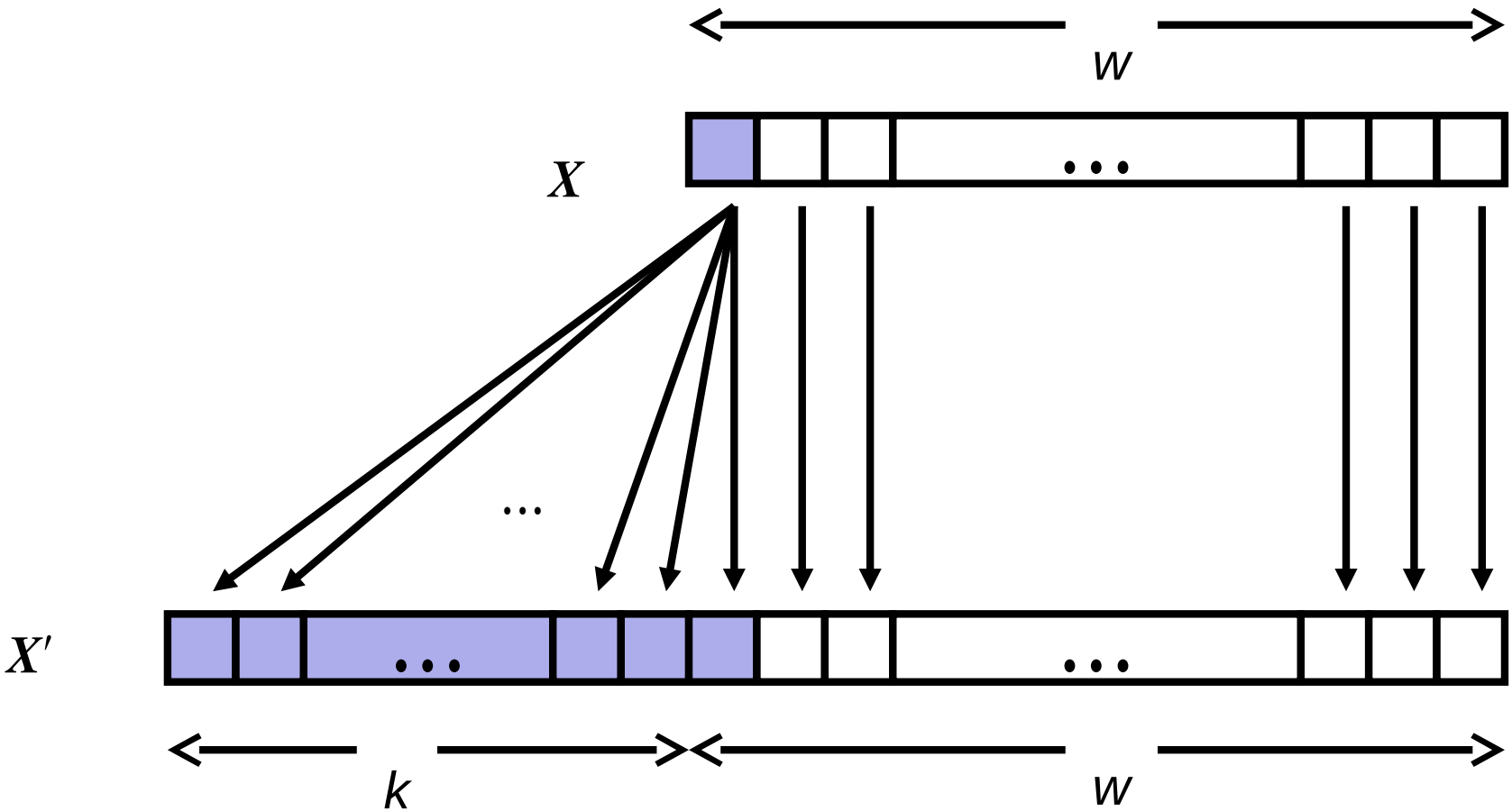
Aufgabe

- Gegeben: Vorzeichenbehaftete w-Bit-Ganzzahl
- Gesucht: Zahl mit $w + k$ Bits und gleichem numerischen Wert
- Trivial für vorzeichenlose Zahlen!

Vorgehensweise

- Vorzeichenbit an k neuen (höchstwertige) Positionen kopieren
- w Bits für Wert übernehmen

Sign Extension II



Sign Extension III

Sign Extension

```
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

	Dezimal	Hexadezimal	Binär
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Abbildung basiert auf Bryant & O'Hallaron

Endianess I

Mehrere Bytes pro Zahl

- Betrachte Zahl $0x12345678$ im Speicher an Position $0x1000$
- Mehrere Bytes erforderlich ⇒ Unterschiedliche Anordnungen möglich

Speicheradresse	$0x1000$	$0x1001$	$0x1002$	$0x1003$
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Konventionsfrage

- Little Endian: Intel x86, AMD64, ARM (erste Versionen)
- Big Endian: Motorola m68k, SPARC, MIPS
- Bi-Endian (wählbar): Alpha, ARM, (MIPS), (POWERPC)
- Arabische Zahlen, Netzwerk-Byteorder (Internet): Big Endian
- Mischformen möglich, aber nicht sonderlich verbreitet!

Endianess I

Mehrere Bytes pro Zahl

- Betrachte Zahl $0x12345678$ im Speicher an Position $0x1000$
- Mehrere Bytes erforderlich $\Rightarrow$ Unterschiedliche Anordnungen möglich

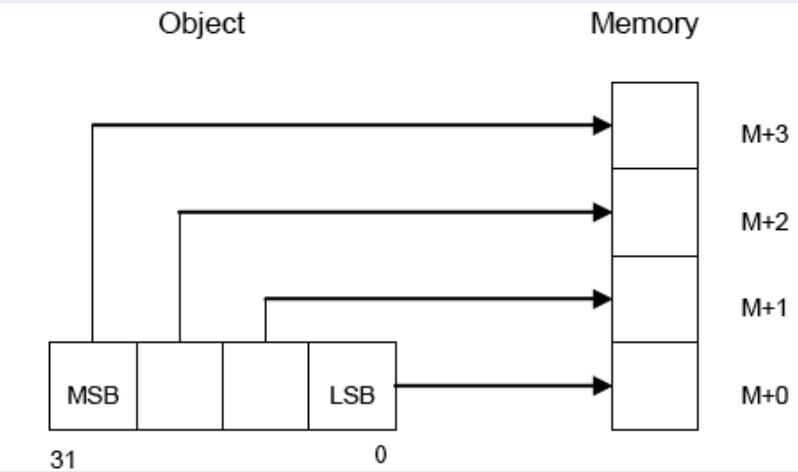
Speicheradresse	$0x1000$	$0x1001$	$0x1002$	$0x1003$
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Vorteile

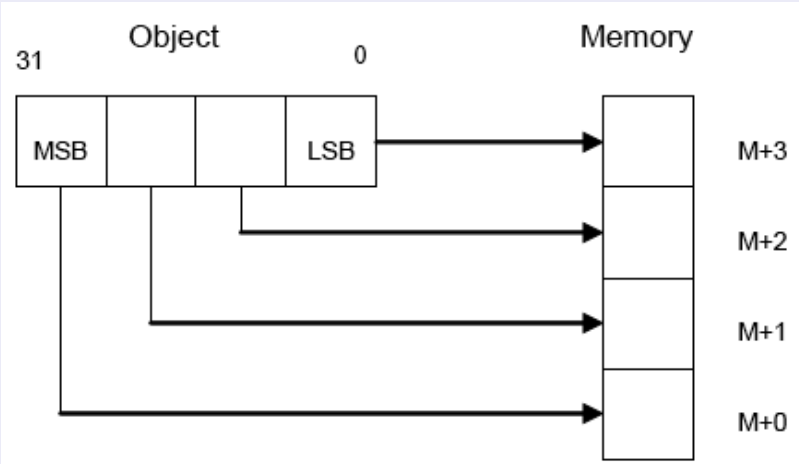
- Big Endian: (Speicher) Dumps von Zahlen leichter zu lesen; Standard für Datenübertragung im Internet
- Little Endian: Triviale Casts zwischen verschiedenen Bitbreiten

Endianess II

Little Endian



Big Endian



Abbildungsquelle: Procedure Call Standard for the ARM Architecture

Endianess III

Standardtest aus GNU Autoconf

```
// 0: big endian, 1: little endian
int endianess(){
    uint32_t i=0x01234567;

    return (*((uint8_t*)(&i))) == 0x67;
}
```

Endianess IV

- Endianess häufig transparent für Programmierer (bsp. beim *logischen* Links-Shift):

```
uint32_t value = 0x0000F000; // Little Endian: 00 F0 00 00
printf("Wert vor Shift : %#.8x\n", value); // Gibt immer
    0x0000F000 aus
value = value << 1; // Logischer Links-Shift
printf("Wert nach Shift: %#.8x\n", value); // Gibt immer
    0x0001E000 aus
// Speicherbelegung:
// - auf Little Endian (x86): 00 E0 01 00
// - auf Big Endian (MIPS): 00 01 E0 00
```

- Unterschiedliche Bit-Operation, gleiches Resultat
- Relevant für Systemprogrammierung: Kommunikation mit Geräten, Low-Level-Netzwerkprotokolle, Datenträgeraustausch zwischen Maschinen, …

Daten untersuchen

```
void print_bytes(void *val, int len) {
  uint8_t *ptr = (uint8_t*)val;
  int count;

  for (count = 0; count < len; count++) {
    printf("0x%x: %.2x", &ptr[count], ptr[count]);
  }
}
```

Zeichendarstellung I

Zeichendarstellung

- Buchstaben und Zeichenketten: Darstellung durch Bitstring, d.h. anderes Alphabet
- Code: Konvention zur Abbildung zwischen Alphabeten:
  - ASCII-Code: Abbildung 7/8 Bits (fixe Anzahl!) auf Buchstaben
  - Morse-Code: Abbildung variabler Anzahl ($\{\cdot, -\}$ auf Buchstaben
- Zeichenketten durch Aneinanderreihung von Bits erzeugen
- Achtung: Code ≠ Font (Zeichen vs. Glyphen)

Zeichendarstellung II

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Bildquelle: Wikipedia

Zeichendarstellung III

Unicode

- Weltweit verwendete Schriftsysteme/Zeichen: Deutlich mehr als 2^8 Zeichen
- Unicode: Soll *alle* früher und heute verwendeten Zeichen enthalten. Organisiert in 17 *Ebenen* mit je 2^{16} Zeichen. Unterschiedliche Kodierungen:
  - UTF-8 (Linux, Plan 9, Internet): 128 Zeichen deckungsgleich mit ASCII; zwei Bytes für (u.A.) deutsche Umlaute, 4 Bytes für allgemeine Zeichen
  - UTF-16 (Windows, Java, Mac OS X): Zeichen dargestellt durch zwei oder vier Bytes
  - UTF-32: 4 Bytes für jedes Zeichen
  - Kodierung mit variabler Länge: Mehr Rechenaufwand; mit fester Länge: Mehr Speicherplatz
  - UTF-16 und UTF-32: Endianess der Kodierung relevant!

Zeichendarstellung IV

Bereich	Binärwerte
$0x0 - 0x007F$	$0xxxxxxx$
$0x0080 - 0x07FF$	$110xxxxx\ 10xxxxxx$
$0x0800 - 0000FFFF$	$1110xxxx\ 10xxxxxx\ 10xxxxxx$
$0x10000 - 0x0010FFFF$	$11110xxx\ 10xxxxxx\ 10xxxxxx\ 10xxxxxx$

Gleitkommazahlen I

Zahlmengen

- $\mathbb{N}$: Abzählbar unendlich, geschlossenes Intervall: endlich viele Werte
- $\mathbb{Q}$: Abzählbar unendlich, geschlossenes Invervall: Unendlich viele Werte
- $\mathbb{R}$: Überabzählbar unendlich
- Rationale und reelle Zahlen nur approximativ darstellbar

Kodierung

- Zahlendarstellung basierend auf Bitstring
- Interpretation komplizierter als bei ganzen Zahlen
- Darstellbares Intervall wichtiger als genauere Approximation von Zahlen

Gleitkommazahlen II

Darstellung rationaler/reeller Zahlen

- Unterschiedliche äquivalente Schreibweisen:

$$\begin{aligned} c_0 &= 299792458 \ \frac{\mathrm{m}}{\mathrm{s}} \\ &= 299792,458 \cdot 10^3 \ \frac{\mathrm{m}}{\mathrm{s}} \\ &= 0,299792458 \cdot 10^9 \ \frac{\mathrm{m}}{\mathrm{s}} \\ &= 2,99792458 \cdot 10^8 \ \frac{\mathrm{m}}{\mathrm{s}} \end{aligned}$$

- Normalisierung: Darstellung mir nur einer Ziffer vor Dezimaltrenner. Basis 2: Entweder 0 oder 1. Erster Fall trivial, daher immer 1.
- Zuätzliches (Gratis-)Bit an Präzision

Gleitkommazahlen III

Allgemeine Repräsentation

Generelle Form: $x = s \cdot m \cdot b^e$

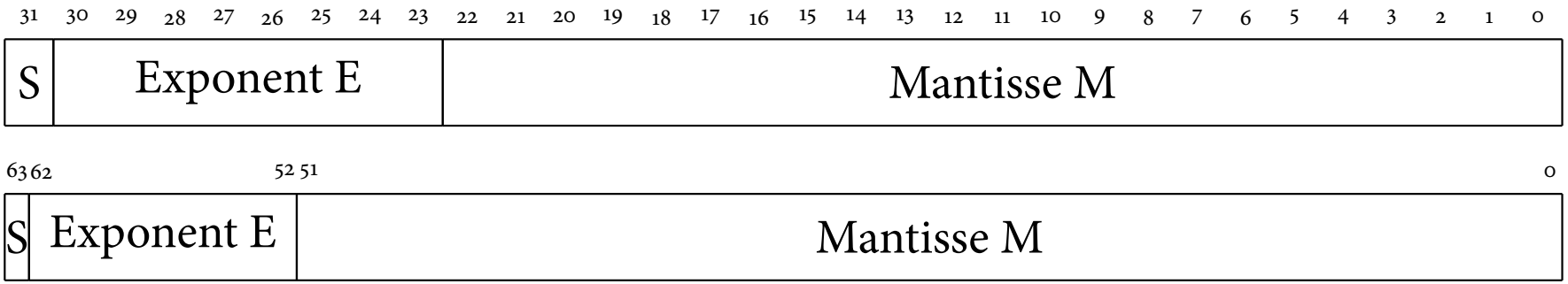
- Vorzeichen s: ± 1
- Mantisse m
- Basis b: Bei Menschen 10, bei Computern 2
- Exponent e

Einflussfaktoren

- Mantisse: Genauigkeit
- Exponent: Zahlenumfang

Gleitkommazahlen III

IEEE754: Definiert Standardformate mit ≈ 7 bzw. 15 dezimalen Stellen relativer Genauigkeit:



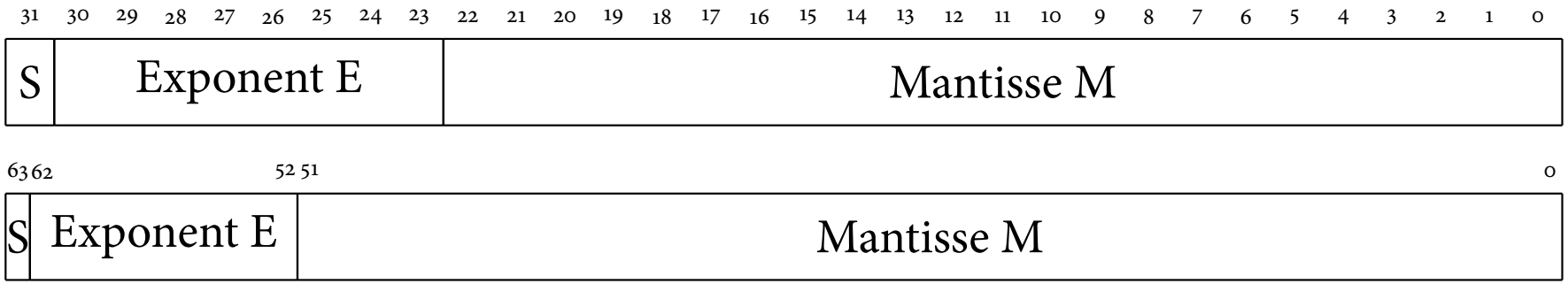
- Einfache Genauigkeit: $|S| = 1, |E| = r = 8, |M| = p = 23$
- Doppelte Genauigkeit: $|S| = 1, |E| = 11, |M| = 52$

Interpretation Bitmuster zu $x = s \cdot m \cdot b^e$

- $s = (-1)^S$
- $m = 1, M$ (Kommazahl!) respektive $m = 1 + \frac{M}{2^p}$
- $e = E - B$ (*Biaswert* $B = 2^{r-1} - 1$ zur Darstellung negativer Exponenten)
- Beispiel: Siehe Tafel

Gleitkommazahlen III

IEEE754: Definiert Standardformate mit ≈ 7 bzw. 15 dezimalen Stellen relativer Genauigkeit:



- Einfache Genauigkeit: $|S| = 1, |E| = r = 8, |M| = p = 23$
- Doppelte Genauigkeit: $|S| = 1, |E| = 11, |M| = 52$

Extremale Zahlen: Einfache Genauigkeit

- Kleinste normalisierte Zahl: $\pm 1,0 \cdots 0 \times 2^{-126} \approx \pm 1,18 \cdot 10^{-38}$
- Größte Zahl: $\pm 1,1 \cdots 1 \times 2^{127} \approx \pm 3,4 \cdot 10^{38}$

Gleitkommazahlen IV

Sonderfälle

- NaN (*not a number*), beispielsweise $\frac{1}{0}$
- Singularitäten ($\pm\infty$)
- Denormalisierte Zahlen: Nahe an 0

Denormalisierte Zahlen

- Exponent kodiert als $0x0$
- Interpretation: $1 - B$ (*nicht* $0 - B$!)
- Mantisse: Implizite führende 0 (*nicht* 1)
- Präzisionsverlust für $\lim_{x \to 0} x$
- Zwei Nullwerte: ± 0 (Limes!)

Abbildung basiert auf Bryant & O'Hallaron

Gleitkommazahlen IV

Sonderfälle

- NaN (*not a number*), beispielsweise $\frac{1}{0}$
- Singularitäten ($\pm\infty$)
- Denormalisierte Zahlen: Nahe an 0

Denormalisierte Zahlen

Verkürztes IEEE-Format: $|E| = 3$ (Bias $= 2^{3-1} - 1 = 3$), $|M| = 2$

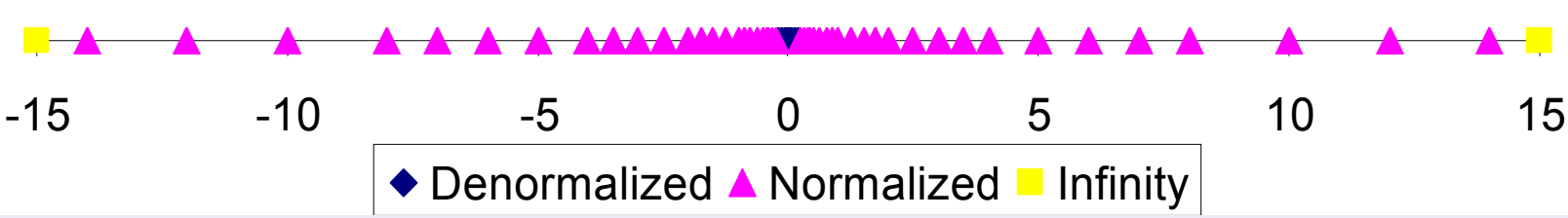


Abbildung basiert auf Bryant & O'Hallaron

Gleitkommazahlen IV

Sonderfälle

- NaN (*not a number*), beispielsweise $\frac{1}{0}$
- Singularitäten ($\pm\infty$)
- Denormalisierte Zahlen: Nahe an 0

Denormalisierte Zahlen

Verkürztes IEEE-Format: $|E| = 3$ (Bias $= 2^{3-1} - 1 = 3$), $|M| = 2$

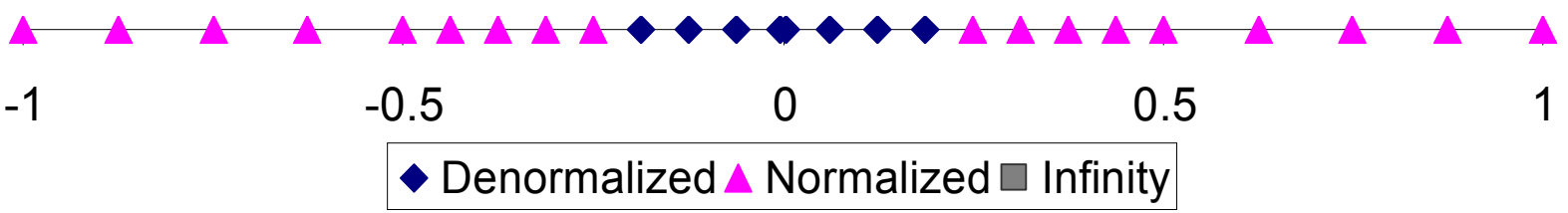


Abbildung basiert auf Bryant & O'Hallaron

Gleitkommazahlen V

NaN und $\pm\infty$

- Exponent in beiden Fällen 111…1
- Mantisse 000…0: $\pm\infty$
- Not a Number (NaN): Mantisse $\neq$ 000…0

Zusammenfassung (Single Precision)

Exponent	Mantisse	Objekt
0	0	Null
0	$\neq 0$	Denormalisierte Zahl
1..254	beliebig	Normalisierte Zahl
255	0	$\pm\infty$
255	$\neq 0$	NaN

Gleitkommazahlen VI

Rechen

- Prinzip: $a \odot_{\text{fp}} b = \text{round}(a \odot b)$
- Unabhängig von konkreter Implementierung

Eigenschaften

- Kommutativgesetz gilt: $a \odot b = b \odot a$
- Keine Assoziativität: $a \odot (b \odot c) \neq (a \odot b) \odot c$
- Beispiel: $(3.141 + 10^{20}) - 10^{20} = 0$, $3.141 + (10^{20} - 10^{20}) = 3.141$
- Keine Distributivität: $a \cdot (b + c) \neq a \cdot b + a \cdot c$
- Compileroptimierungen problematisch

Gleitkommazahlen VII

Beispiel: Multiplikation

$$(-1)^{S_1} \cdot M_1 \cdot 2^{E_1} \times (-1)^{S_2} \cdot M_2 \cdot 2^{E_2} = (-1)^{S} \cdot M \cdot 2^{E}$$

- $S = S_1 \oplus S_2$, $M = M_1 \cdot M_2$, $E = E_1 + E_2$
- Wenn $M \geq 2$ ☞ Rechtsshift und E erhöhen
- Wenn E zu groß ☞ Overflow
- M auf passende Bitanzahl runden

Bitoperationen

Boole'sche Algebra

- Formale Grundlage vieler Bitoperationen
- Algebraische Darstellung der Logik
- »Wahr« = 1
- »Falsch« = 0
- Auch für Bitvektoren definiert!

Wahrheitstabellen

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

∧	0	1
0	0	1
1	1	0

~	0
0	1
1	0

Bitoperationen II

Anwendung auf Bitvektoren: Siehe Tafel

Bitoperationen III: Mengen

Darstellung von Mengen

- Bitvektoren repräsentieren Mengen
- Nützlich bei Statusflags (μController, Betriebssysteme, …)
- Elemente der Menge durch Bits kodiert

Kodierung

- Bitvektor mit w Bits: Teilmenge $\{0, \ldots, w-1\}$
- Interpretation als Flags: $\{A, B, C, D, E, F, \ldots\}$
- $w_j = 1$ wenn $j \in w$
- Beispiel:
  - 01101001 = {G, F, D, A} (HGFEDBCA)
  - 01010101 = {G, E, C, A} (HGFEDCBA)

Bitoperationen III: Mengen

Darstellung von Mengen

- Bitvektoren repräsentieren Mengen
- Nützlich bei Statusflags (μController, Betriebssysteme, …)
- Elemente der Menge durch Bits kodiert

Operationen

- &: Schnittmenge ☞ 01000001 = {A, G}
- |: Vereinigungsmenge ☞ 01111101 = {A, C, D, E, F, G}
- ^: Symmetrische Differenz ☞ 00111100 = {C, D, E, F}
- ~: Komplement ☞ 10101010 = {B, D, F, H}

Bitoperationen IV

Trick: Komplement und Inkrement

- `~x + 1 == -x`
- Gilt, da `~x + x == -1`
- Veranschaulichung: Siehe Tafel

Trick: Inverses eines Wertes

- Komplementbildung: $a \& \neg a = 0 \; \forall a$
- »Ausnullen« von Registern

Trick: Flags testen

- Idempotenz: $a | a = a \; \forall a$
- Testen auf *exakten* Satz an Flags: `(var | FLAGS) == FLAGS`

Bitoperationen V

Bitoperationen in C

- Operatoren: `&`, `|`, `~`, `^`
- Definiert auf allen integralen Datentypen (`short`, `int`, `long`, `unsigned`, `char`)
- Bitweise Bearbeitung der Argumente

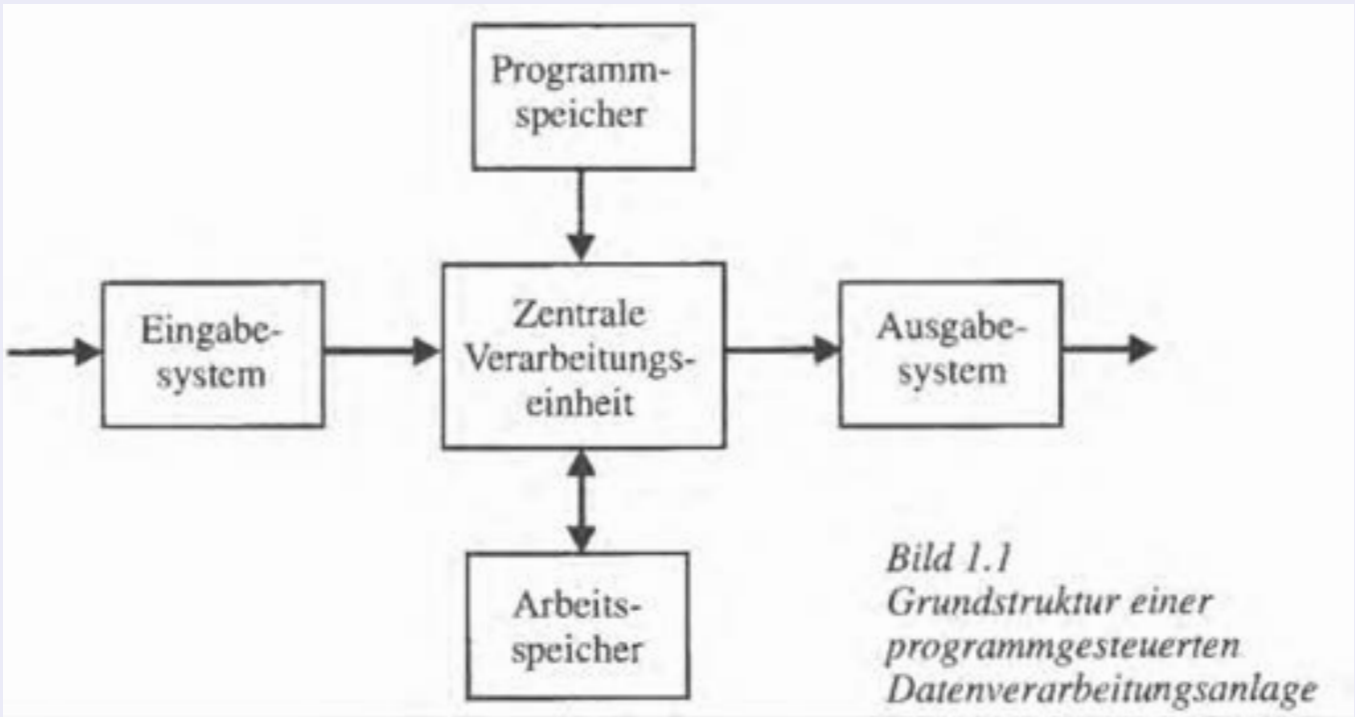
Logische Operationen

- Operatoren: `&&`, `||`, `!`
- Unterschiedliche Wahrheitstabelle
  - 0 = »Falsch«
  - Alles andere: »Wahr«
- Bit als Rückgabewert
- Partielle Auswertung!

Inhalt

Datenverarbeitungssystem

Grundstruktur DV-System



Kernelement: Zentrale Verarbeitungseinheit (*Central Processing Unit, CPU*) – (Mikro)Prozessor

Abbildung: Taschenbuch der Mikroprozessortechnik, Fachbuchverlag Leipzig

Prozessorarchitektur I

Komponenten eines Prozessors

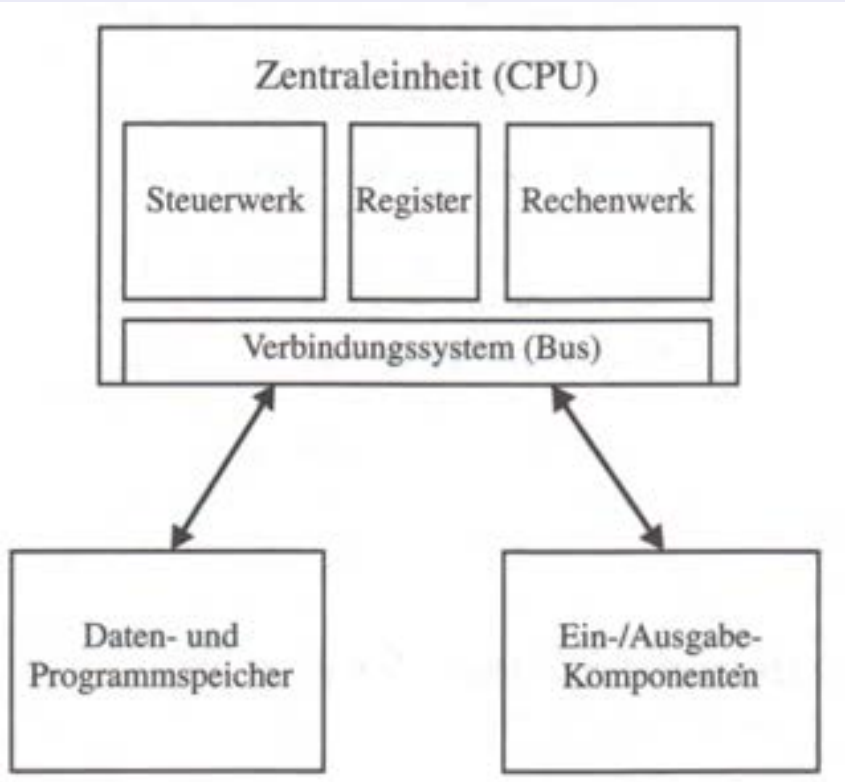


Abbildung: Taschenbuch der Mikroprozessortechnik, Fachbuchverlag Leipzig

Prozessorarchitektur II

Halbleiterschaltungen

- ASIC (*Application Specific Integrated Circuit*): In Silizium »gegossene« elektronische Spezialschaltung
- FPGA (*Field Programmable Gate Array*): Rekonfigurierbare Schaltung
- IC: Vielzweckprozessor

Prozessortypen

- Vielzweckprozessoren
- Hochleistungsprozessoren
- Vektorprozessoren
- Signalprozessoren (*Digital Signal Processors – DSP*)

Prozessorarchitektur III

Eigenschaften von Vielzweckprozessoren

- Keine Spezialisierung
- Einsatzweck a priori unbekannt
- Universelle, dynamische Programmierbarkeit
- Einfache Elementarbefehle
- Schnelle Ausführung von Elementarbefehlen

Prozessorarchitektur IV

Aufbau einer CPU

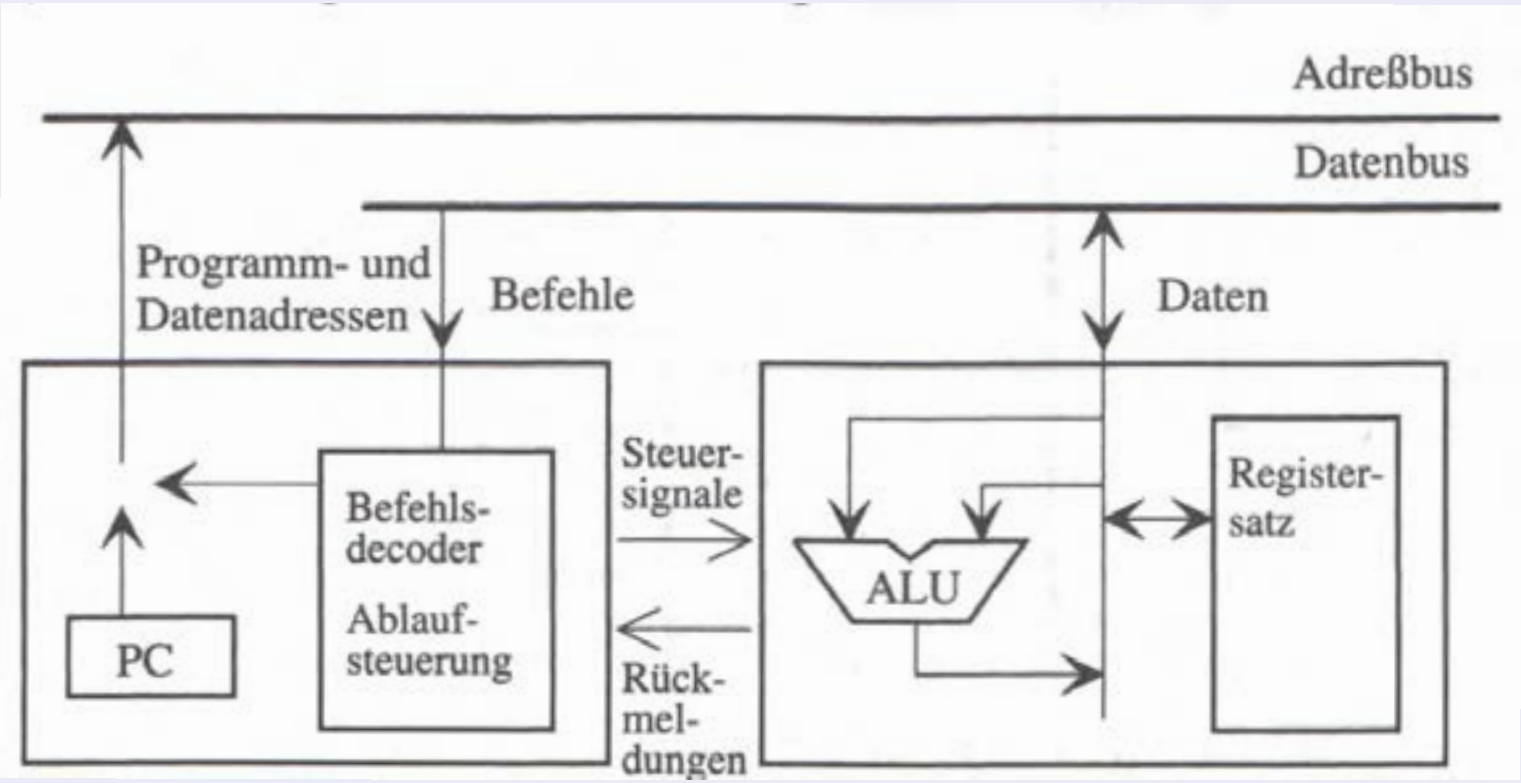


Abbildung: Taschenbuch der Mikroprozessortechnik, Fachbuchverlag Leipzig

Grundoperationen einer CPU I

- Arithmetische Funktionen
  - Ganzzahl- und Gleitkommaarithmetik, Formatkonvertierung, Sign Extension, …
- Logische Funktionen und Bitoperationen
  - Vergleiche, And, Or, Bit-Shifting, Negation, …
- Speicherzugriff
  - Lesen und Schreiben von Speicherwerten
- Bedingungen und Sprünge
- Unterbrechungen
- Speicherverwaltung und Betriebssystemunterstützung
- Spezialbefehle
  - Verschlüsselung, Hashing, FFT, Performancemessung, Debugging, Diagnose …

Grundoperationen einer CPU II

Gegeben

- Programm im Speicher
  - Grundfunktionen in *Instruktionen* kodiert
  - Interpretation binärer Daten
  - Herstellerabhängige Kodierung
- Daten im Speicher

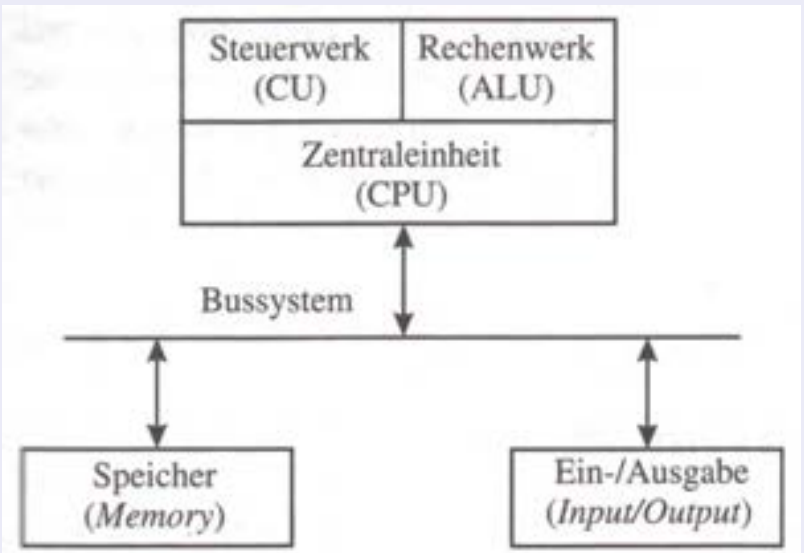
Arbeitsweise: Phasen

1. Neue Instruktion aus Speicher ins Steuerwerk laden (*fetch*)
2. Instruktion auswerten (*decode*)
3. Benötigte Daten aus Speicher ins Rechenwerk lesen, Daten gemäß Instruktion verarbeiten und Ergebnis in Speicher schreiben (*execute*)
4. Weiter bei 1

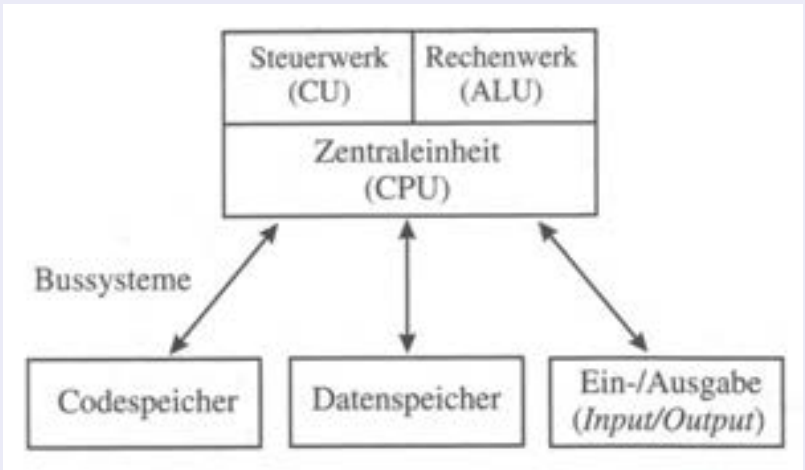
Aktuelle Position im Programm: *Befehlszähler (Program Counter)*

von Neumann/Harvard-Architektur

von Neumann



Harvard



Moderne Rechner

- Mischform aus beiden Konzepten
- Daten und Code gemeinsam im RAM, aber Aufteilung bei Caches

Abbildung: Taschenbuch der Mikroprozessortechnik, Fachbuchverlag Leipzig

Intermezzo: Zeiger

Zeiger (*Pointer*)

- Direkter Zugriff auf den Speicher
- Systemsoftware, Simulatoren, High Performance
- Ähnliche Sicht wie aus Assembler-Ebene
- Leistungsfähig, aber gefährlich

Bjarne Stroustrup

C makes it easy to shoot yourself in the foot.

Intermezzo: Zeiger

Zeiger (*Pointer*)

- Direkter Zugriff auf den Speicher
- Systemsoftware, Simulatoren, High Performance
- Ähnliche Sicht wie aus Assembler-Ebene
- Leistungsfähig, aber gefährlich

Bjarne Stroustrup

C makes it easy to shoot yourself in the foot; C++ makes it harder.

Intermezzo: Zeiger

Zeiger (*Pointer*)

- Direkter Zugriff auf den Speicher
- Systemsoftware, Simulatoren, High Performance
- Ähnliche Sicht wie aus Assembler-Ebene
- Leistungsfähig, aber gefährlich

Bjarne Stroustrup

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

Intermezzo: Zeiger

Zeiger-Beispiel 0

```
1   int i[2] = { 42, 23 };
2   int x = 4711;
3
4   int *xptr = &x;
5   int *iptr = &i[0];
6
7   printf("x vor Zeiger-Operation: %d\n", x);
8   *xptr = 1147;
9   printf("x nach Zeiger-Operation: %d\n\n", x);
10
11  printf("&i[0]: %p, &i[1]: %p\n", &i[0], &i[1]);
12  printf("Differenz &i[1] - &i[0]: %lu\n", &i[1]-&i[0]);
13  printf("Void-Differenz &i[1] - &i[0]: %lu\n\n",
      (void*)&i[1]-(void*)&i[0]);
14
15  printf("iptr: %p, iptr+1: %p\n", iptr, iptr+1);
```

Intermezzo: Zeiger

Zeiger-Beispiel 1

```
1   unsigned long l = ULONG_MAX;
2   unsigned int i[2] = { 0, 1 };
3   unsigned int *iptr = &i[0];
4   unsigned long *iptr2 = (unsigned long*)&i[0];
5
6   printf("i[0]: %u, i[1]: %u\n", i[0], i[1]);
7
8   *iptr = l;
9   printf("i[0]: %u, i[1]: %u\n", i[0], i[1]);
10
11  *iptr2 = l;
12  printf("i[0]: %u, i[1]: %u\n", i[0], i[1]);
```

Intermezzo: Zeiger

Zeiger-Beispiel 2

```
1   struct collection {
2     int a, b, c;
3     unsigned long d;
4   };
5
6   struct collection c[2];
7   struct collection *cptr = &c[0];
8
9   printf("sizeof(int): %lu, sizeof(long): %lu\n", sizeof(int),
      sizeof(long));
10  printf("sizeof(collection): %lu\n\n", sizeof(struct
      collection));
11
12  printf("&c[0]: %p, &c[1]: %p\n\n", &c[0], &c[1]);
13  printf("Differenz &c[1] - &c[0]: %lu\n", &c[1]-&c[0]);
14  printf("Differenz (void*)&c[1] - (void*)&c[0]: %lu\n",
      (void*)&c[1]-(void*)&c[0]);
15
16  printf("cptr: %p, cptr+1: %p\n", cptr, cptr+1);
```

Intermezzo: Zeiger

Zeiger-Beispiel 3

```
1   #define offsetof(type, element) \
2        ((size_t)&(((type *)0)->element))
3
4   struct collection {
5     int a;
6     int b;
7     int c;
8     unsigned long d;
9   };
10
11  printf("offset(a): %lu\n", offsetof(struct collection, a));
12  printf("offset(b): %lu\n", offsetof(struct collection, b));
13  printf("offset(c): %lu\n", offsetof(struct collection, c));
14  printf("offset(d): %lu\n", offsetof(struct collection, d));
```

Befehlsausführung I

Befehlsphasen

Fetch → Decode → Execute

Register

- Rechnungsoperanden, Speicheradressen etc. in schnellen, winzigen Speichern (Register) im Prozessor vorgehalten
- Typischerweise 32 oder 64 Bits *Wortbreite* (μController: 8 und 16 Bits)

Befehlsausführung II: iSmpl

Beispielarchitektur iSmpl

Mischung aus ARM und x86:

- 8 Register: $\langle R_0 \rangle$, …, $\langle R_5 \rangle$, SP (Stack Pointer), PC (Program Counter)
- 32 Bit Wortbreite
- Befehlslänge: 8–44 Bits
- 4 Bits zur Kodierung Opcode
- Statusregister: Zero, Less, Greater

Instruktionen (Mnemonics)

- `nop`: No Operation
- Speicherzugriff: `load`, `store` (Speicher/Register)
- Arithmetik: `add`, `sub`, `cmp` (Compare)
- Bitoperationen: `and`, `or`, `xor`
- Sprünge: `jmp` (unkonditional), `jlt`, `jgt` (less/greater than), `jeq` (equal), `jne` (not equal)

Befehlsausführung III

Fetch: Instruktion laden

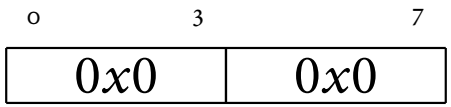
- Speicherinhalt an Position des Befehlszählers lesen (Dereferenzieren!)
- Befehlszähler erhöhen

Decode: Instruktion auswerten

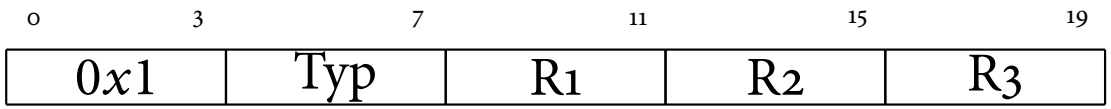
- Instruktion: Operationscode (*Opcode*) und *Argumente*
- Argumente: Speicheradressen oder Registernamen
- ☞ Komponenten auswerten und an Rechenwerk übergeben

Befehlsausführung IV: iSmpl-Kodierung

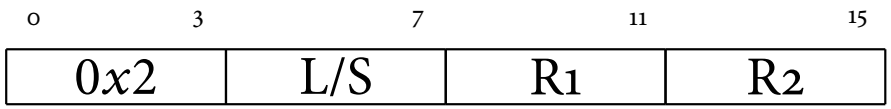
- Nop:

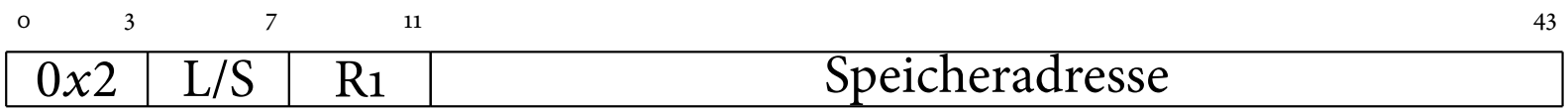


- Arithmetik + Logik:

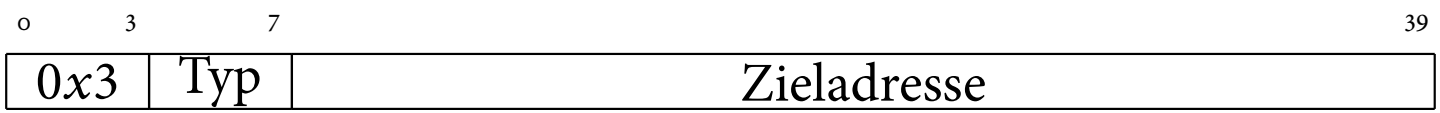


- Speicher:





- Sprünge:



Befehlsausführung V

Execute: Befehlsausführung

Typischerweise aufgespalten in mehrere Aktionen (Anzahl Hersteller- und CPU-abhängig):

- Operanden aus Speicher holen (*operand fetch*)
- Operation ausführen (*execute*), Wirkung je nach Opcode)
- Ergebnis schreiben (*writeback*), Speicher/Register

Befehlsausführung VI: Beispiele iSmpl

Addition zweier Zahlen

```
load r1, 0x0
load r2, 0x4
add r1, r1, r2
store r1, 0x8
```

- Speicher byteweise adressiert ☞ 32-Bit-Werte im Abstand 0x4
- Register $\langle R_1 \rangle$ »wiederverwendet«

Zeiger dereferenzieren

```
load r1, 0x0
load r2, r1
```

- Speicherinhalt an Position $0x0 : 0x100$
- Inhalt Register $\langle R_2 \rangle$: Inhalt von Speicherzellen

$$\langle 0x100, 0x101, 0x102, 0x103 \rangle$$

Befehlsausführung VII

Bedingungen und Sprünge

- Bislang: Lineare Ausführung, keine Sprünge
- Probleme:
  - Nach welche Kriterien springen?
  - Keine »Kommunikation« zwischen den Befehlen
- Lösung: Statusregister mit Flags, bsp.
  - Overflow
  - Less (equal), Greater (equal)
  - Carry
  - Zero

Ablaufdiagramm: Siehe Tafel

Beispiel

```
load r1, 0x0
load r2, 0x4
cmp r1, r2
jne L1 // r1 != r2?
<Anweisungen fuer Ergebnis r1 == r2>
...
jmp L2

L1:
<Anweisungen fuer Ergebnis r1 != r2>
...
L2:
<Gemeinsamer Pfad>
```

Sprungimplementierung

Sprung = Veränderung Befehlszähler

Befehlsausführung VIII

Schleife

```
load r1, 0x0
load r2, 0x4
load r3, 0x8
L1:
// Schleifenkoerper: Aktion durchfuehren
sub r1, r1, r3
cmp r1, r2 // Schleifenzähler == 0?
jne L1     // Nein, Schleife nochmals durchlaufen
L2:
// Code nach Schleife
```

Speicherinhalt:

- $0x0$: 10
- $0x4$: 0
- $0x8$: 1

RISC versus CISC

CISC (bsp. x86)

- *Complete Instruction Set Computer*
- Viele, teilweise hochkomplexe Instruktionen (`pclmullqlqdq`...)
- Potentiell lange Instruktionsdauer (z.B. Speicherblöcke kopieren)
- Variable Instruktionslänge
- Operationen direkt im Speicher möglich
- Funktionsparameter: Stack

RISC (bsp. MIPS, ARM)

- *Reduced Instruction Set Computer*
- Wenige, sehr elementare Instruktionen
- Schnelle Instruktionsausführung
- Feste Instruktionslänge
- Load/Store-Prinzip
- Funktionsparameter: Register

Befehlszyklen I

Gesamtablauf

- Assembler-Befehl: Mehrere Phasen
- Synchronisation notwendig:
  - Fetch beendet → Decode beginnen,
  - Decode beendet → Execute beginnen, …
- Gemeinsamer *Takt* in allen CPU-Komponenten verfügbar

Taktfrequenz

- Taktung: CPU-Frequenz, z.B. 700 MHz oder 2.2 GHz:

$$f = 700\ \text{MHz} = 700 \cdot 10^6 \frac{1}{\text{s}} = 7 \cdot 10^8 \frac{1}{\text{s}} = \frac{1}{\Delta t}$$

$$\Delta t = \frac{1}{f} = 1/7 \cdot 10^{-8} \approx 1.43\ \text{ns}$$

- Höhere Frequenz → Schnellere Verarbeitung

Befehlszyklen I

Gesamtablauf

- Assembler-Befehl: Mehrere Phasen
- Synchronisation notwendig:
  - Fetch beendet → Decode beginnen,
  - Decode beendet → Execute beginnen, …
- Gemeinsamer *Takt* in allen CPU-Komponenten verfügbar

Taktfrequenz

- Taktung: CPU-Frequenz, z.B. 700 MHz oder 2.2 GHz:

$$f = 700\ \text{MHz} = 700 \cdot 10^6 \frac{1}{\text{s}} = 7 \cdot 10^8 \frac{1}{\text{s}} = \frac{1}{\Delta t}$$

$$\Delta t = \frac{1}{f} = 1/7 \cdot 10^{-8} \approx 1.43\ \text{ns}$$

- *Aber:* Höhere Frequenz → Höherer Stromverbrauch

Befehlszyklen II

Zyklentypen

- *Befehlszyklus*: Abarbeitung einer Assembler-Instruktion
- *Maschinenzyklus*: Abarbeitung einer Befehlsphase (Fetch etc.)
- *Taktzyklus*: Elementare Zeiteinheit der CPU

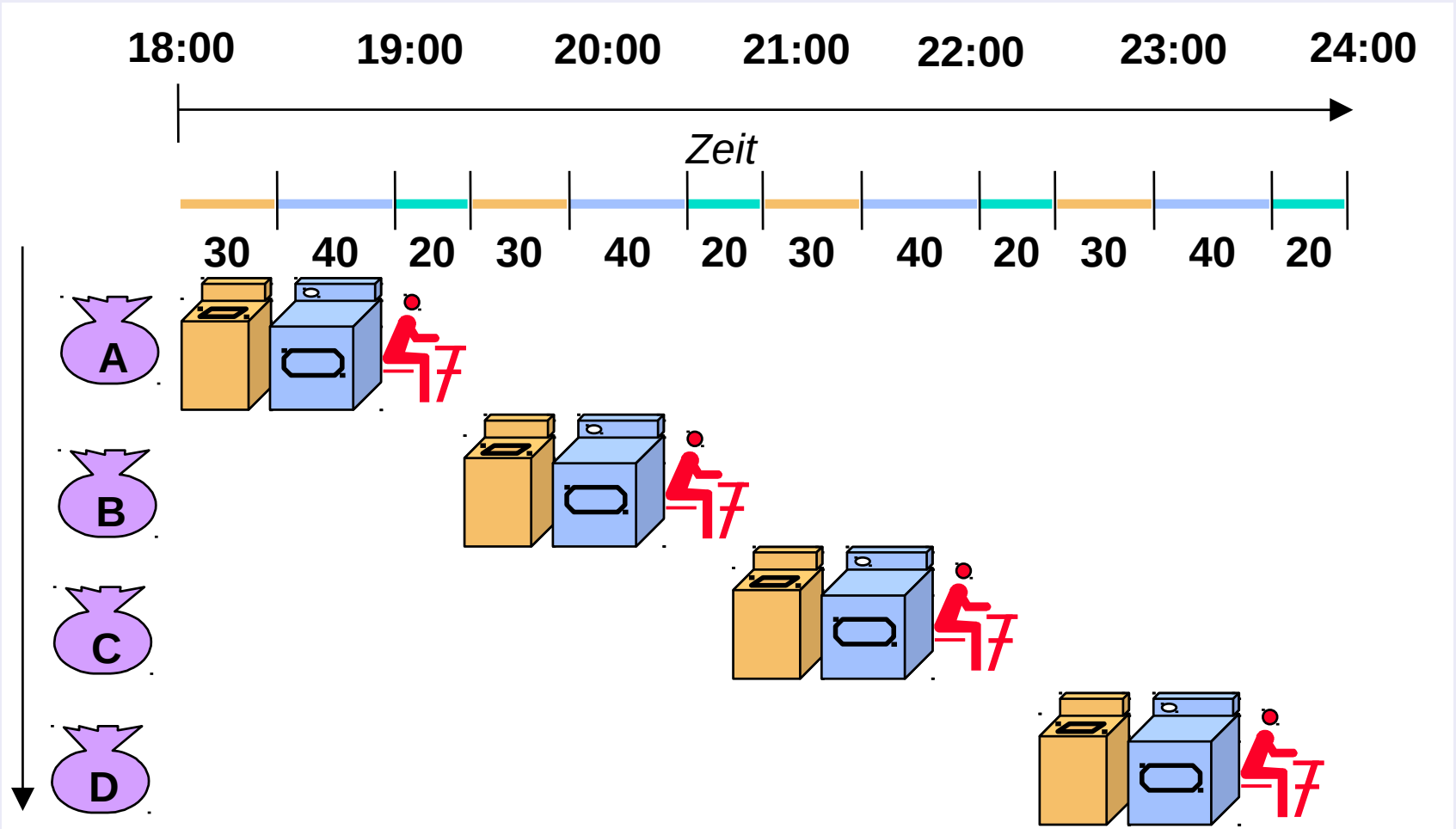
Herausforderungen

- Anzahl Takte pro Zyklus variiert
  - Prinzipbedingt: Unterschiedlich lang kodierte Instruktionen
  - Variabel: Dauer Speicherzugriff (Caches, NUMA-Systeme, …)
- Nicht alle Maschinenzyklen für alle Befehlstypen notwendig (Beispiel: Kein Writeback bei Sprung)
- Komplexe Ablaufsteuerung (endlicher Automat!)

Illustration: Siehe Tafel

Pipelining I: Waschtag

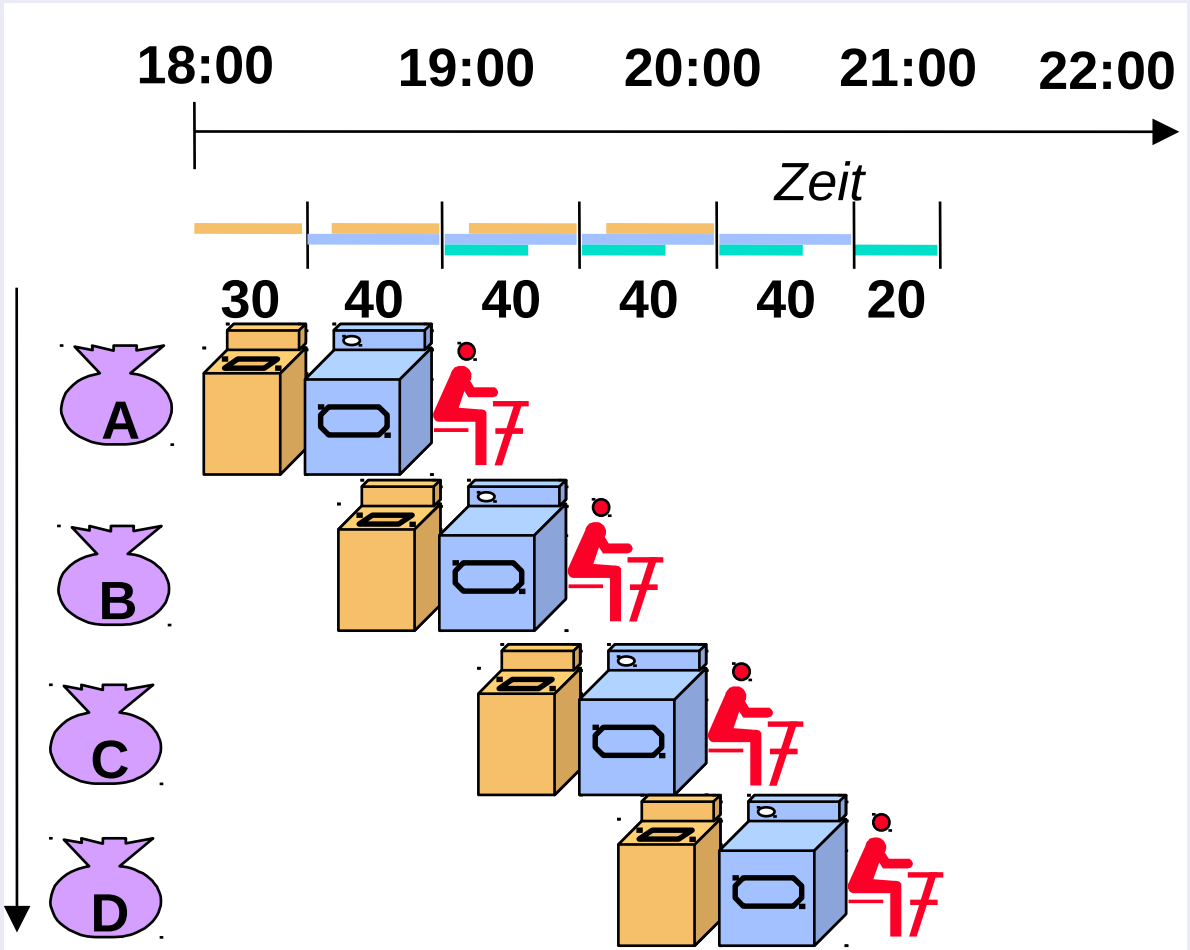
Waschtag in der WG



Bildquelle: Hennessy & Patterson, *Computer Architecture*

Pipelining I: Waschtag

Waschtag in der WG



Bildquelle: Hennessy & Patterson, *Computer Architecture*

Pipelining II

Pipelining auf CPUs

IF ID OF IE WB IF ID OF IE WB

Grenzen des Pipelinings

- Komplexität (gleich lange Stages konstruieren)
- Nicht verwendete Stages
- Stalls durch Konflikte

Pipelining II

Pipelining auf CPUs

1	IF	ID	OF	IE	WB				
2		IF	ID	OF	IE	WB			
3			IF	ID	OF	IE	WB		
4				IF	ID	OF	IE	WB	
5					IF	ID	OF	IE	WB
6						IF	ID	OF	IE

Grenzen des Pipelinings

- Komplexität (gleich lange Stages konstruieren)
- Nicht verwendete Stages
- Stalls durch Konflikte

Pipelining II

Pipelining auf CPUs

1	IF	ID	OF	IE	WB				
2		IF	ID	OF	IE	WB			
3			IF	ID	OF	IE	WB		
4				IF	ID	OF	IE	WB	
5					IF	ID	OF	IE	WB
6						IF	ID	OF	IE

Grenzen des Pipelinings

- Komplexität (gleich lange Stages konstruieren)
- Nicht verwendete Stages
- Stalls durch Konflikte

Pipelining III: Konflikte

Konflikte

- *Datenkonflikt*: Instrukion $n + j$ benötigt Ergebnis von Instruktion n (Beispiel: Rechenoperand)
- *Ressourcenkonflikt*: Aufeinanderfolgende Instruktionen benötigen gleiche HW-Ressource
- *Kontrollkonflikt*: Adresse von Instruktion $n + j$ hängt ab von Instruktion n (Beispiel: Sprung)

Pipelining IV: Konflikte II

Datenkonflikt: Instruktion 4 benötigt Datum von Inst. 3

1	IF	ID	OF	IE	WB						
2		IF	ID	OF	IE	WB					
3			IF	ID	OF	IE	WB				
4				IF	ID	←Stall→	OF	IE	WB		
5					IF	←Stall→	ID	OF	IE	WB	
6							IF	ID	OF	IE	WB
7								IF	ID	OF	IE

Pipelining V: Konflikte III

Kontrollkonflikt: Sprung von Instruktion 3 auf Instruktion 42

1	IF	ID	OF	IE	WB								
2		IF	ID	OF	IE	WB							
3			IF	ID	OF	IE	WB						
4				IF	ID	OF		←	Verlust		→		
5					IF	ID							
6						IF							
42								IF	ID	OF	IE	WB	
43									IF	ID	OF	IE	WB

Optimierungsmechanismen I

Superskalarität

- *Statisches Scheduling:* Befehlsfolge statisch; Anzahl paralleler Befehle dynamisch von CPU bestimmt
- *Dynamisches Scheduling:* CPU bestimmt Reihenfolge paralleler Befehle durch Umordnung (*out-of-order*)
- *Very Long Instruction Word* (VLIW): Mehrere Befehle pro Instruktion

Illustration: Zwei parallele Pipelines

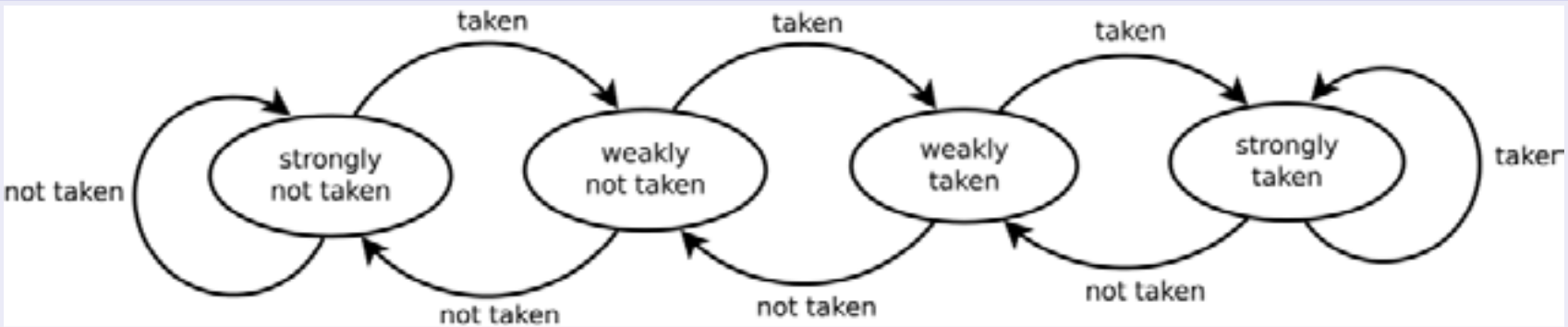
1	IF	ID	OF	IE	WB		
2	IF	ID	OF	IE	WB		
3		IF	ID	OF	IE	WB	
4		IF	ID	OF	IE	WB	
5			IF	ID	OF	IE	WB
6			IF	ID	OF	IE	WB

Optimierungsmechanismen II

Sprungvorhersage

- Vermeiden von Pipeline-Stalls bei Verzweigungen
- Sprungvorhersage (*Branch Prediction*):
  - Statische Vorhersage (Beispiel: Rückwärts ja, vorwärts nein)
  - Saturating Counter (Zustandsmaschine mit zwei Bits)
  - Maschinelles Lernen zur Vorhersage aus Analyse der (globalen/lokalen)Historie

Illustration: Saturating Counter



Bildquelle: Wikipedia

Optimierungsmechanismen III

Multi- und Many-Core

- Mehrere CPUs pro System
- Übliche Konfiguration (bis auf kleinste Systeme)
- CPUs: 2-16 Cores üblich; GPUs (Grafikkarten): Tausende
- Probleme: Synchronisierung, Arbeitsverteilung

Multithreading

- Physikalischer Prozessorkern als mehrere *virtuelle* Kerne sichtbar
- Bessere Ausnutzung der statischen Ressourcen

Arbeitsaufteilung

- Speedups nicht automatisch erzielbar
- Muss in Algorithmen, Systemsoftware etc. berücksichtigt werden!

Hochsprachen und CPUs

Compiler

- *Compiler*: Konverter Hochsprachenprogramm → Maschinenformat
- Spezifische Übersetzung je Maschinentyp (und Betriebssystem) erforderlich

C-Beispiel

```
int x, y, z;
...
x = y + z
```

Assembler-Code

```
load r1, &y
load r2, &z
add r0, r1, r2  // r0 <- r1 + r2
store r0, &x
```

Inhalt

Hochsprachen und Assembler

System und ISA

- System: Hardware *und* Software
- »Grenzschicht«: Assembler.
- *Instruction Set Architecture* (ISA): Register, grundlegende Befehle, Speicherzugriff, Unterbrechungen, …
- Kompatibilität wichtig: ISAs typischerweise abwärtskompatibel (Bsp. ARMv1, ⊆ …⊆ ARMv8, IA32 ⊆ AMD64/EMT64)

Systemdetails

- Coprozessoren, GPUs, Floating Point-Mechanismen
- Zubehörgeräte (Grafik, Sound, Netwerk, …)
- Prozessor-Caches, Betriebsfrequenz(en)
- Mikroarchitektur: Implementierung einer ISA

ISA ≠ Implementierung

Bsp. ARM: Intel, Qualcom, Freescale, …

ARM: Allgemeines und Historie I

ARM-Historie

- 8oer Jahre: Acorn Computers (Co-Prozessor für BBC Micro)
- Acorn RISC machine (ARM): Berkeley RISC-inspirierte CPU (Erster Simulator in Basic!)
- Archimedes: Erster ARM-Basierter Computer
- Transition zu Standard-Embedded-Prozessor (geringe Leistungsaufnahme, System-on-Chip (SoC))
- ARM Holdings: Kein HW-Produzent; Know-How-Verkauf

ARM: Allgemeines und Historie II

ARM: Allgemeines

- ISA: ARMvx, $x \in [1, 8]$
- Zahlreiche Hardware-Implementierungen (synthetisierte Cores!)
  - Parameter wählbar: Core-Anzahl, Cache-Größen, etc.
  - In »HW-Programmiersprache« beschrieben
- Momentan aktuell: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv6
  - T: *Thumb*
  - J: *Java*
  - E: DSP-Instruktionen
- ARMv7: *Cortex*-Reihe mit Profilen Cortex-y, $y \in \{A, R, M\}$
  - A: Anwendung (*Application*)
  - R: HaRte Echtzeit (*Real-Time*)
  - M: μController (*Microcontroller*)

Kompatibilität von ISAs

ISA-Kompatibilität

- Abwärtskompatibilität (*Jahrzehnte*!); neue Architekturen selten (Itanium…)
- Bsp.: ARMv4 ⊆ ARMv5 ⊆ ARMv6 ⊆ ARMv7 ⊆ ARMv8 ⊆ ⋯
- Trend: Optionale Erweiterungen

Vorteile

- Programme »für immer« (Branchenspezifische Software!)
- Prozessoren (relativ) direkt austauschbar
- ABI-Stabilität (*Application Binary Interface*) möglich

Nachteile

- Neue Programmierstile (Assembler → Hochsprachen): ISA-Möglichkeiten obsolet
- HW-Leistung nicht optimal verwendet (bsp. Multicore)

System aus Sicht Assembler-Programmier/in

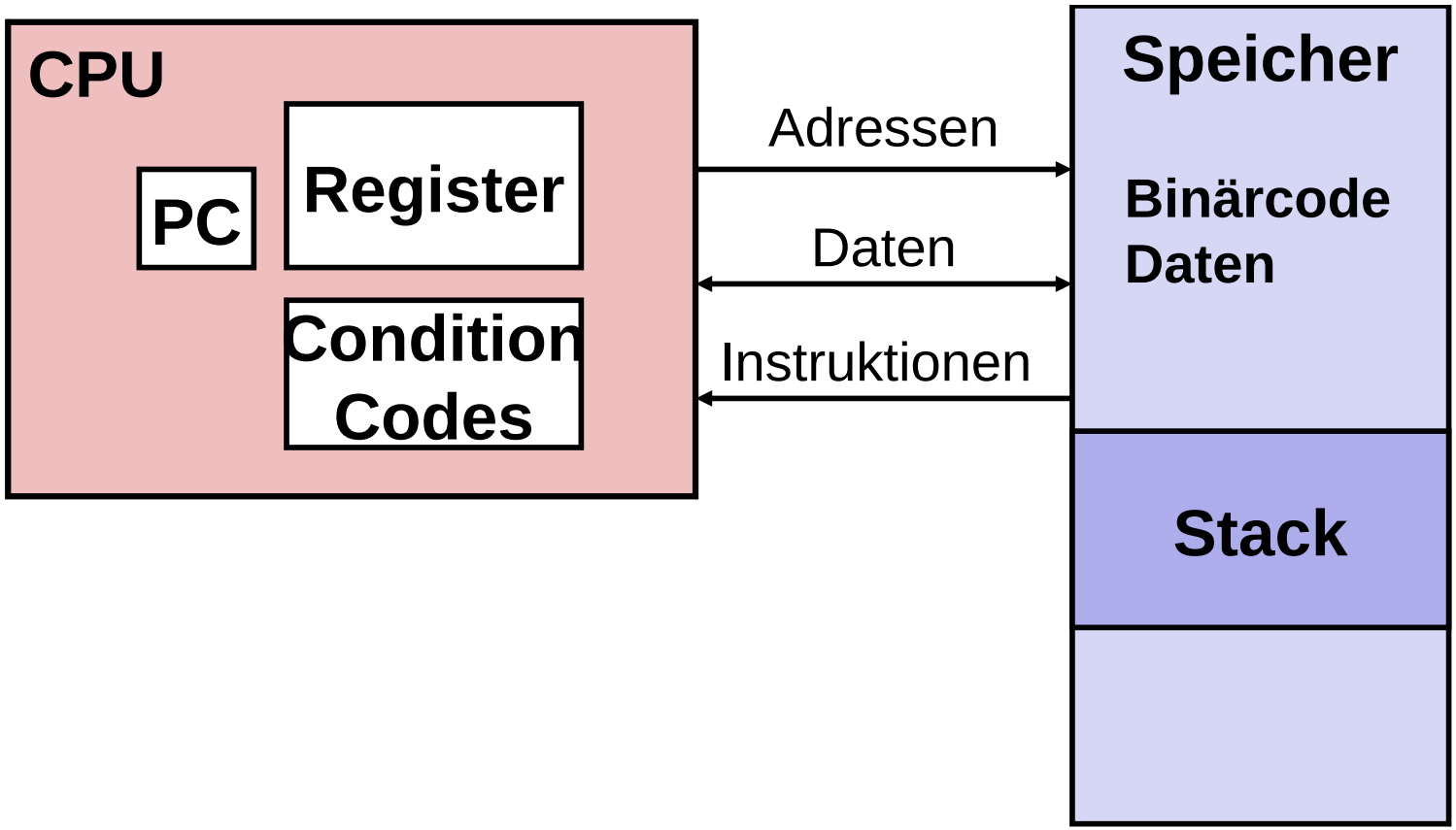
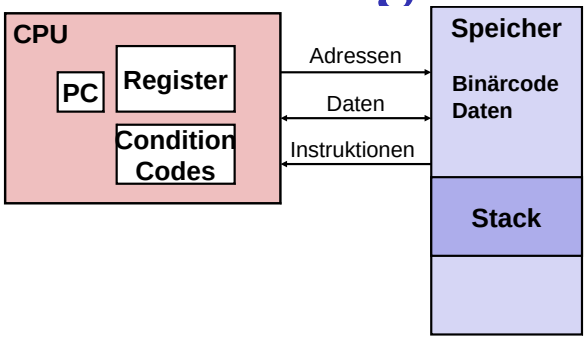


Abbildung basiert auf Bryant & O'Hallaron

System aus Sicht Assembler-Programmier/in



Sichtbarer Zustand

- *Program Counter* (PC): Aktuelle Stelle im Code
- Register (file): Werte zur direkten Bearbeitung, Funktionsparameter
- Condition Codes: Statusinformationen, Verzweigungen

Speicher

- Byte-adressierbares Array
- Programm- und Nutzdaten
- Stack: Funktionsaufrufe und lokale Variablen; per *Konvention* besonders ausgezeichneter Speicherbereich

Instruktionssets

ARM und x86: Unterschiedliche, inkompatible ISAs

- x86: CISC, vorwiegend Desktop und Cloud
- ARM: RISC, vorwiegend mobile Geräte (Handy, Tablet)

Quellcode

```
int sum(int x, int y) {
    int res;
    res = x + y;

    return (res);
}
```

Instruktionssets

ARM und x86: Unterschiedliche, inkompatible ISAs

- x86: CISC, vorwiegend Desktop und Cloud
- ARM: RISC, vorwiegend mobile Geräte (Handy, Tablet)

ARM

```
sum:
  sub  sp, sp, #20
  str  r0, [sp, #16]
  str  r1, [sp, #12]
  ldr  r2, [sp, #16]
  add  r2, r2, r1
  str  r2, [sp, #8]
  str  r0, [sp, #4]
  mov  r0, r2
  str  r1, [sp]
  add  sp, sp, #20
  bx   lr
```

x86 (AT&T-Syntax)

```
sum:
  pushq %rbp
  movq  %rsp, %rbp
  movl  %edi, -20(%rbp)
  movl  %esi, -24(%rbp)
  movl  -24(%rbp), %eax
  movl  -20(%rbp), %edx
  addl  %edx, %eax
  movl  %eax, -4(%rbp)
  movl  -4(%rbp), %eax
  popq  %rbp
  ret
```

Instruktionssets

ARM und x86: Unterschiedliche, inkompatible ISAs

- x86: CISC, vorwiegend Desktop und Cloud
- ARM: RISC, vorwiegend mobile Geräte (Handy, Tablet)

ARM

```
sum:
  sub   sp, sp, #20
  str   r0, [sp, #16]
  str   r1, [sp, #12]
  ldr   r2, [sp, #16]
  add   r2, r2, r1
  str   r2, [sp, #8]
  str   r0, [sp, #4]
  mov   r0, r2
  str   r1, [sp]
  add   sp, sp, #20
  bx    lr
```

x86 (Intel-Syntax)

```
sum:
  push   rbp
  mov    rbp, rsp
  mov    DWORD PTR [rbp-20], edi
  mov    DWORD PTR [rbp-24], esi
  mov    eax, DWORD PTR [rbp-24]
  mov    edx, DWORD PTR [rbp-20]
  add    eax, edx
  mov    DWORD PTR [rbp-4], eax
  mov    eax, DWORD PTR [rbp-4]
  pop    rbp
  ret
```

Objektcode

ARM

```
0xe5 0x2d 0xd0 0x04 0xe2 0x8d
0xb0 0x00 0xe2 0x4d 0xd0 0x14
0xe5 0x0b 0x00 0x10 0xe5 0x0b
0x10 0x14 0xe5 0x1b 0x20 0x10
0xe5 0x1b 0x30 0x14 0xe0 0x82
0x30 0x03 0xe5 0x0b 0x30 0x08
0xe5 0x1b 0x30 0x08 0xe1 0xa0
0x00 0x03 0xe2 0x8b 0xd0 0x00
0xe8 0xbd 0x08 0x00 0xe1 0x2f
0xff 0x1e
```

Intel

```
0x55 0x48 0x89 0xe5 0x89 0x7d
0xec 0x89 0x75 0xe8 0x8b 0x45
0xe8 0x8b 0x55 0xec 0x01 0xd0
0x89 0x45 0xfc 0c8b 0x45 0xfc
0x5d 0xc3
```

Unterschiede und Ähnlichkeiten

- Beide Architekturen: Sequenz von Bytes (ARM umfangreicher!)
- Binärcode (offensichtlich) völlig unterschiedlich
- Grundlegende Prinzipien dennoch oft ähnlich
- Maschine/CPU: *Keine* Kenntnis über Hochsprachenquelltext!

Beispiel: Addition in Assembler I

ARM-Assemblercode

```
ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
add r3, r2, r3
str r3, [fp, #-8]
```

Illustration: Siehe Tafel

C-Äquivalent

```
int r2, r3, *fp;

// Operanden aus Array
// (Speicher) holen
r2 = fp[-16];
r3 = fp[-20];

r2 = r2 + r3; // Addition

// Resultat in Array schreiben
fp[-8] = r2;
```

- Kurios: Array mit negativen Indizes
- Ansonsten: Sehr einfacher C-Code mit elementaren Operationen

Beispiel: Addition in Assembler I

x86-Assemblercode

```
mov     eax,DWORD PTR [rbp-0x18]
mov     edx,DWORD PTR [rbp-0x14]

add     eax,edx

mov     DWORD PTR [rbp-0x4],eax
```

C-Äquivalent

```
int eax, edx, *rbp;

// Operanden aus Array
// (Speicher) holen
eax = rbp[-0x18];
edx = rbp[-0x14];

eax = eax + edx; // Addition

// Ergebnis in Array schreiben
rbp[-0x4] = eax;
```

- Kurios: Array mit negativen Indizes
- Ansonsten: Sehr einfacher C-Code mit einfachsten Elementen

Eigenschaften von Assembler I

Daten

- Daten: Eines oder mehrere Bytes
- Zeiger zur Speicheradressierung
- Datenverarbeitung in Registern (RISC) oder gemischt Register/Speicher (CISC)
- Speicher und Register untypisiert
- Keine zusammengesetzten Datentypen; kontinuierliche Speicherbereiche

Operationen

- Datentransfer Speicher ↔ Register
- Arithmetische und logische Funktionen auf Registern (CISC: auch Speicher) ausführen
- Programmfluss durch (bedingte) Sprünge und Unterroutinen
- CISC: Mächtige Spezialinstruktionen (AES, SHA, FFT, …), RISC: Kombination elementarer Operationen

Eigenschaften von Assembler II

Stapelspeicher

- Jede Funktion: Arbeitsbereich auf Stack
- Lokale Variablen, Parameterübertragung (neben Registern, je nach Konvention – definiert in *ABI*-Dokument)

Struktur

- *Frame Pointer*: Stapelanfang
- *Stack Pointer:* Aktuelles Stapelende
- Stack wächst von oben nach unten
- Daten auf Stack relativ zu FP oder SP adressierbar

Vereinfachungen

- Funktionsaufrufe: Strukturen auf Stack komplexer
- Reale Speicheraufteilung deutlich komplizierter (Bibliotheken, Interaktionscode mit BS)
- `cat /proc/⟨pid⟩/maps`

Die Werkzeugkette I

Selbstverständnis Informatik

- Informatik ≠ Werkzeuge bedienen
- **Aber**: Ohne Werkzeuge kann man nichts erschaffen
- *Verständnis*: Prinzipien und Grundlagen
- *Tun*: Probleme der realen Welt lösen!

Die Werkzeugkette II

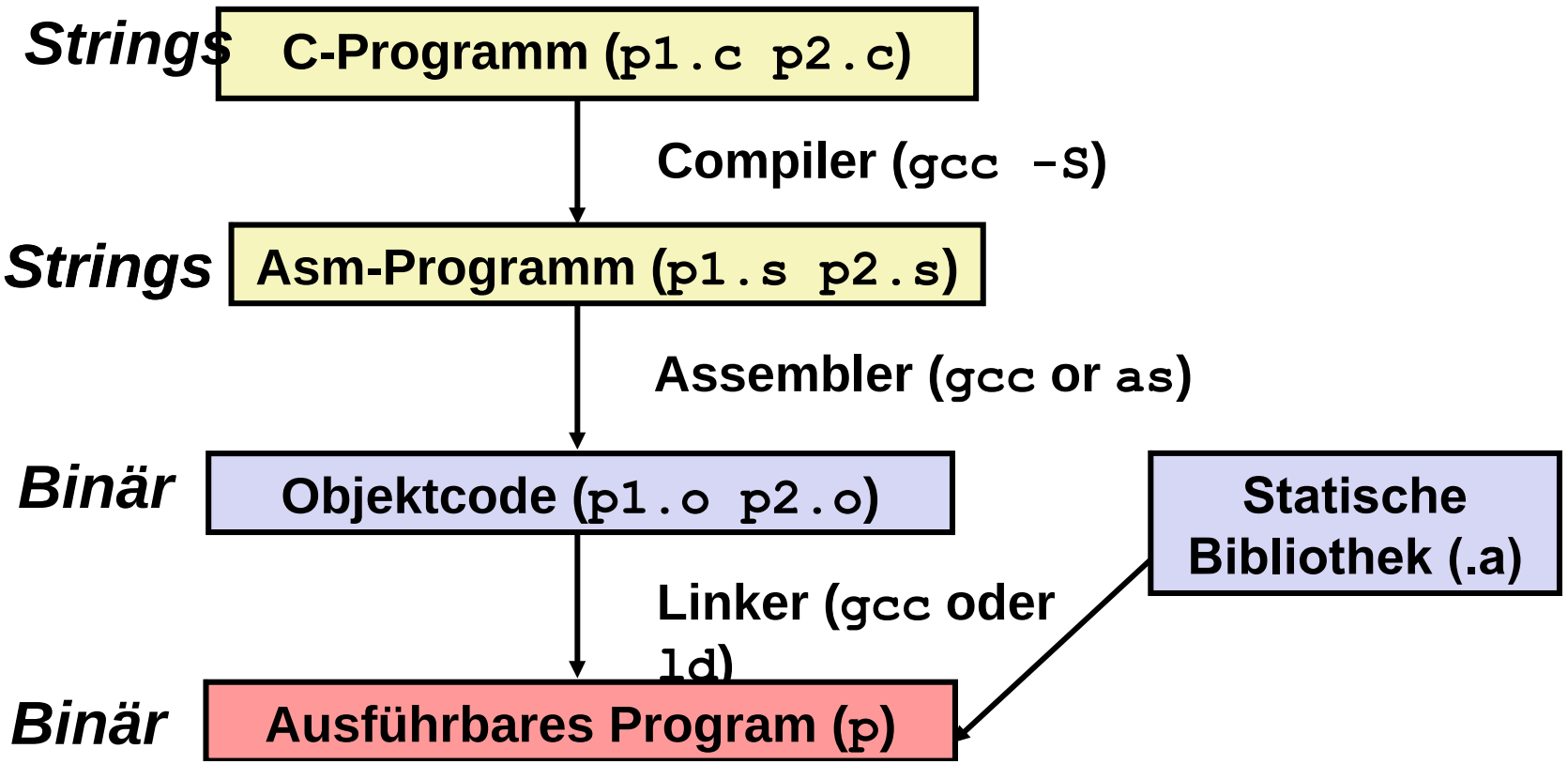


Abbildung basiert auf Bryant & O'Hallaron

Die Werkzeugkette III

Werkzeuge (*Tools*)

- `gcc`: C-Compiler; Dispatcher für andere Teile der Kompilationskette
- `gas`: Assembler
- `ld`, `ld.so`: (Dynamischer und statischer) Binder/Linker
- `objdump`, `readelf`: Analyse (und Disassemblierung) von Binärdateien
- `gdb`: Debugger (Live- und Post-Mortem-Analyse)
- `nm`: Symbole anzeigen (Funktionen etc.)
- `strings`: Zeichenkettenkonstanten anzeigen
- `addr2line`: Adressen auf Quelltextsymbole abbilden
- Aus *GNU Binutils*: `www.gnu.org/software/binutils`

Die Werkzeugkette IV

Anmerkungen

- ⟨`tool`⟩ für native Verwendung (host = target)
- Präfix für Cross-Tools (host ≠ target): `arm-linux-gnueabi-`⟨`tool`⟩, `mips-sgi-irix-`⟨`tool`⟩, …
- Komplexität: `gcc` >1700, `clang` > 200 Kommandozeilenoptionen → mehr als ⟨F5⟩ drücken.

Die Werkzeugkette V

```
sh> arm-linux-gnueabi-objdump -dS sum.o
int sum (int x, int y) {
   0: e52db004  push {fp} ; (str fp, [sp, #-4]!)
   4: e28db000  add fp, sp, #0
   8: e24dd014  sub sp, sp, #20
   c: e50b0010  str r0, [fp, #-16]
  10: e50b1014  str r1, [fp, #-20]
  int res;
  res = x+y;
  14: e51b2010  ldr r2, [fp, #-16]
  18: e51b3014  ldr r3, [fp, #-20]
  1c: e0823003  add r3, r2, r3
  20: e50b3008  str r3, [fp, #-8]

  return (res);
  24: e51b3008  ldr r3, [fp, #-8]
}
  28: e1a00003  mov r0, r3
  2c: e28bd000  add sp, fp, #0
  30: e8bd0800  ldmfd sp!, {fp}
  34: e12fff1e  bx lr
```

Die Werkzeugkette V

```
sh> objdump -dS sum.o

int sum (int x, int y) {
   0: 55                          push   rbp
   1: 48 89 e5                    mov    rbp,rsp
   4: 89 7d ec                    mov    DWORD PTR [rbp-0x14],edi
   7: 89 75 e8                    mov    DWORD PTR [rbp-0x18],esi
  int res;
  res = x+y;
   a: 8b 45 e8                    mov    eax,DWORD PTR [rbp-0x18]
   d: 8b 55 ec                    mov    edx,DWORD PTR [rbp-0x14]
  10: 01 d0                       add    eax,edx
  12: 89 45 fc                    mov    DWORD PTR [rbp-0x4],eax

  return (res);
  15: 8b 45 fc                    mov    eax,DWORD PTR [rbp-0x4]
}
  18: 5d                          pop    rbp
  19: c3                          ret
```

Die Werkzeugkette VI

gdb

- `gdb ⟨objekt⟩`: Funktioniert mit Binärdateien, Programmen, Core-Dumps
- Kann auch disassemblieren
- Statische Daten, aber auch laufende Programme analysieren und *manipulieren*

Cross-Verwendung

- Eingebettete Systeme, Tablets, Smartphones: Oft zu schwach/langsam, um darauf zu entwicklen
- Abbilfe: Cross-Debugging
  - Gerät (*Target*): `gdbserver`
  - Entwicklungsrechner (*Host*): `gdb` (mit minimalen Einschränkungen)

Assembler: Struktur von Anweisungen I

Generelle Aufgaben

- Transfer Register ↔ Speicher
- Verknüpfen von (mehreren) Registern: Logisch, Arithmetisch
- Datenverarbeitung
- Kontrollfluss, Stack
- Systemkontrolle und -verwaltung

Komponenten

- *Mnemonic* gibt an, was gemacht werden soll
- 0–4 Argumente als *Operanden*
  - Register: Wert direkt auslesen
  - Speicher: Zugriff über Zeiger (aus Register)
  - Unmittelbarer Wert (*Immediate*): Konstante in Anweisung einkodiert
  - Weitere Spezialmöglichkeiten (Details später)

Assembler: Struktur von Anweisungen II

Allgemeine Struktur

- Häufig: ⟨*mnemonic*⟩ ⟨R_d⟩, Op_n, Op_m
- Reihenfolge: Ziel (*destination*), *Argumente*
- Bei Registern: $n \in [0, 15]$; $n \in [0, 12]$ frei verwendbar

Beispiel: Speicher/Register-Transfer

- Format: ⟨*mnemonic*⟩ ⟨R_d⟩ ⟨R_n⟩
- `ldr r0, [r1]`
  - `ldr` = *Load RegisteR*
  - Transfer Speicher → Register
  - ⟨R_o⟩ ← memory[⟨R_1⟩]
  - `[r1]`: *Dereferenzierung*
  - Nebenbedingung: Ausrichtung der Daten an 32-Bit-Grenzen!
  - Adresse Vielfaches von 4 Bytes: 0, 4, 8, 12, …
- ARM (RISC): *Kein* Transfer Speicher ↔ Speicher

Assembler: Struktur von Anweisungen II

Allgemeine Struktur

- Häufig: $\langle mnemonic \rangle$ $\langle R_d \rangle$, Op_n, Op_m
- Reihenfolge: Ziel (*destination*), *Argumente*
- Bei Registern: $n \in [0, 15]$; $n \in [0, 12]$ frei verwendbar

Beispiel: Speicher/Register-Transfer

- Format: $\langle mnemonic \rangle$ $\langle R_d \rangle$ $\langle R_n \rangle$
- `str r0, [r1, #N]`
  - `str` = *Store RegisteR*
  - #N: Additive Konstante (Angabe generell möglich)
  - Transfer Register → Speicher
  - $\langle R_o \rangle \rightarrow \text{memory}[\langle R_1 \rangle + N]$
  - Gleiche *Alignment*-Anforderungen
- ARM (RISC): *Kein* Transfer Speicher ↔ Speicher

Struktur von Assembler-Anweisungen III

Variablen vertauschen

```
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Assembler-Code

```
ldr   r3, [fp, #-16] int t0 = *xp;
ldr   r3, [r3]
str   r3, [fp, #-12]

ldr   r3, [fp, #-20] int t1 = *yp;
ldr   r3, [r3]
str   r3, [fp, #-8]

ldr   r3, [fp, #-16] *xp = t1;
ldr   r2, [fp, #-8]
str   r2, [r3]

ldr   r3, [fp, #-20] *yp = t0;
ldr   r2, [fp, #-12]
str   r2, [r3]
```

Illustration: Siehe Tafel

Inhalt

ARM-Architektur I

Grundzüge der ARM-Architektur

- *Alle* Instruktionen: 32 Bit-Kodierung
- *Fast alle* Befehle in einem Zyklus ausführbar
  - RISC »ohne religiösen Eifer«
- Viele Befehle bedingt ausführbar
- Load/Store-Architektur
  - Operationen ausschließlich auf Registern
- ISA über Koprozessoren erweiterbar
  - Systemverwaltung (*System Control)*
  - Gleitkommaeinheit (*Floating Point)*
  - Multimedia
  - …

ARM-Architektur II

Registersatz

- 16 Register (im Userland) sichtbar
  - $\langle R_0 \rangle \ldots \langle R_{12} \rangle$ frei verwendbar
  - $\langle R_{13} \rangle$: Stack Pointer, SP
  - $\langle R_{14} \rangle$: Link Register, LR (für Prozeduraufrufe)
  - $\langle R_{15} \rangle$: Program Counter, PC (aktuelle Stelle im Code + 8 Bytes)
- Spezialregister: CPSR (*Current Program Status Register*)
  - Steuert bedingte Operationen
  - Über Spezialbefehle ansprechbar

ARM-Architektur III

(Normative) Informationsquellen

- ISA: *ARM Architecture Reference Manual* (ARM ARM)
  - Definiert allgemeine Architektur
  - *Keine* implementierungsspezifischen Details (aber: generelle Grenzen)
- Implementierung: *ARM Technical Reference Manual* (ARM TRM)
  - Spezifisch für eine Implementierungs*variante*
  - Generische ISA von allen Implementierungen umgesetzt
- Alle Dokumente verfügbar auf `http://infocenter.arm.com`

Details! Details! Details!

Viele Details! ☞ Manche Details (vorerst) ignorieren

ARM-Architektur III

(Normative) Informationsquellen

- ISA: *ARM Architecture Reference Manual* (ARM ARM)
  - Definiert allgemeine Architektur
  - *Keine* implementierungsspezifischen Details (aber: generelle Grenzen)
- Implementierung: *ARM Technical Reference Manual* (ARM TRM)
  - Spezifisch für eine Implementierungs*variante*
  - Generische ISA von allen Implementierungen umgesetzt
- Alle Dokumente verfügbar auf `http://infocenter.arm.com`

Details! Details! Details!

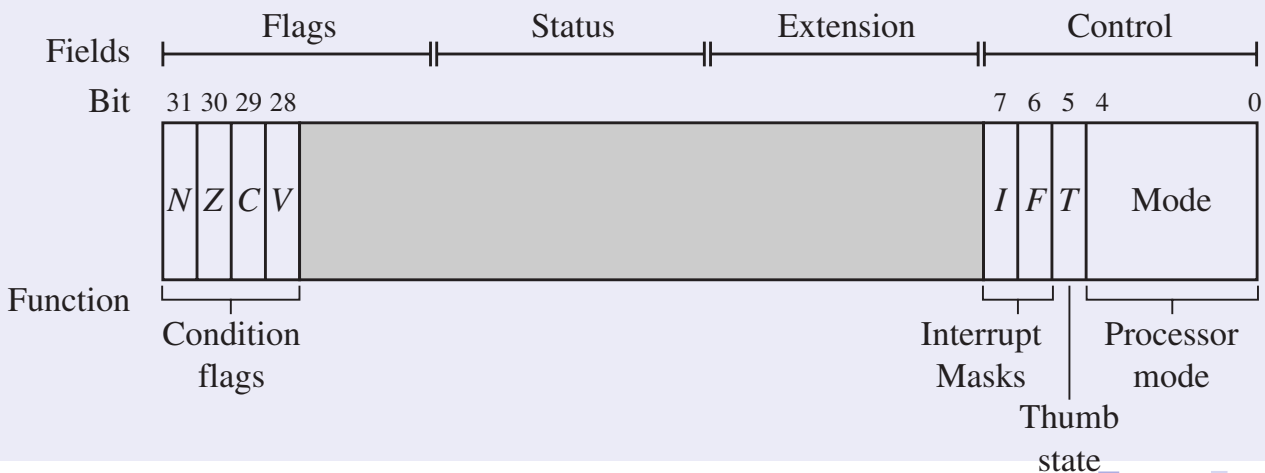
Viele Details! ☞ Manche Details (vorerst) ignorieren

Registeraufbau und CPU-Architektur I

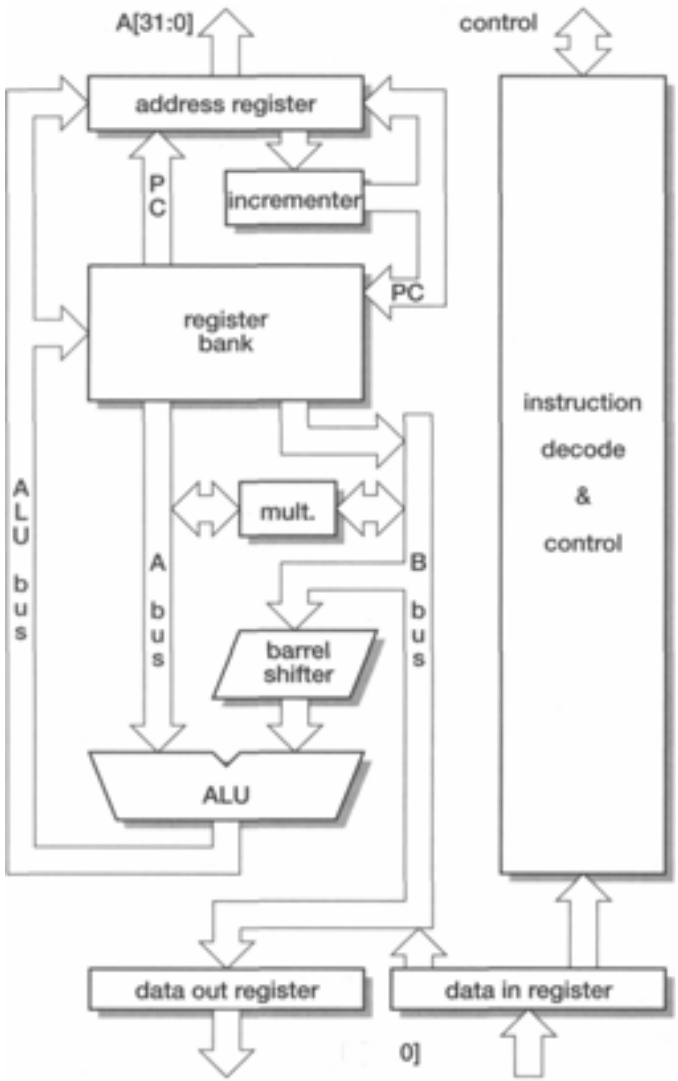
Register

- $\langle R_0 \rangle$–$\langle R_{12} \rangle$ ($\langle R_{13} \rangle$): *Orthogonal*
- Zugriff auf CPSR aus *User-Mode* nur lesend

Status-Register: Details



CPU-Architektur II: Datenfluss



Reale Welt…

- Komplexer
- Abhängig von HW-Implementierung
- Semantik immer identisch!

Inline-Assembler I

gcc: Inline-Assembler

```
void test_asm() {
    // Effektives nop
    asm("mov r3, r3")
}
```

Caveat Emptor

Low-Level-»Optimierungen« typischerweise schlechter als generierter Code!

Rationale

- Einfache Test/Experimentiermöglichkeit
- Reale Welt: Funktionen ohne Hochsprachen-Unterstützung (alternativ: Compiler-*Intrinsics*)
- Reine Assembler-Programme (natürlich) möglich

Inline-Assembler II

Allgemeine Syntax

- `asm(⟨code⟩ : ⟨Liste Ausgabeoperanden⟩ : ⟨Liste Eingabeoperanden⟩ : ⟨Clobber-Liste⟩);`
- *to clobber* = verprügeln; d.h. Modifikation eines Registers

```
int main() {
    uint32_t result;
    uint32_t value = 0x8;

    printf("Wert vor asm: 0x%x\n", value);   // 0x8 = 1000b
    asm("mov %0, %1, ror #1" : "=r" (result) : "r" (value));
    printf("Wert nach asm: 0x%x\n", result); // 0x4 = 0100b

    return(0);
}
```

Details Inline-Assembler: Siehe gcc-Dokumentation

Standalone-Assembler I

Beispiel: Eigenständiges Assembler-Programm

```
/* Eigenständiges Assembler-Programm */
.global main   /* main: Haupteintrittspunkt als globales Symbol */

.func main     /* main als Funktion kenntlich machen */
main:          /* Label für main*/
   mov r0, #42 /* r0: Per Konvention BS-Returncode */
   bx lr       /* Rücksprung */
.endfunc
```

HOWTO

- In Binärobjekt verwandeln (assemblieren): `arm-linux-gnueabi-as standalone.s -o standalone.o`
- Linken (in ausführbares Programm verwandeln): `arm-linux-gnueabi-gcc standalone.o -o standalone`
- Abkürzung: `arm-linux-gnueabi-gcc standalone.s -o`

Standalone-Assembler II

- Returncode ausgeben: `./standalone; echo $?`
- `file`: Unterschiedliche Binärobjekte
- `nm`: Unterschiedliche Symbole

Demo: Siehe Rechner

Standalone-Assembler II

Assembler-Syntax

- `.directive` ⟨*arguments*⟩: Meta-Anweisungen an den Assembler
- `label:` Funktionsnamen, Sprungziele
- #⟨*Zahl*⟩: Unmittelbarer Wert (*Immediate*)

Assembler-Syntax II

- Kommentarzeilen
  - `@ Comment`
  - C-Kommentare
- Kommentare *extrem* wichtig!
- Syntaktischer Zucker, Makros, etc: Siehe `gas`-Anleitung

Pipelining

Pipeline-Verarbeitung

- Instruktionen in mehreren Phasen ausgeführt
- Original-Pipeline: 3 Stufen
  1. Instruktion aus Speicher lesen
  2. Dekodieren verwendeter Register
  3. Register lesen, Operation ausführen, Register zurückschreiben
- Neuere CPUs: Längere Pipelines
- Struktur *Teil der ISA*

Konsequenz

- PC-»Versatz« um 8 Bytes
- Dumps, Relokation (siehe später) etc. zu berücksichtigen

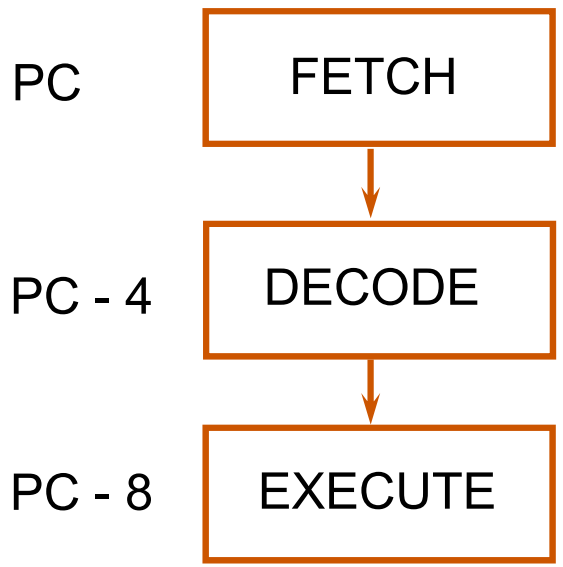


Abbildung: Embedded Systems Architecture, University of Texas

Überblick: Instruktionsformat I

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition	0	0	I	OPCODE							S	Rn				Rs				OPERAND-2												Data processing
Condition	0	0	0	0	0	0	A	S	Rd							Rn				Rs				1	0	0	1	Rm				Multiply
Condition	0	0	0	0	1	U	A	S	Rd HIGH							Rd LOW				Rs				1	0	0	1	Rm				Long Multiply
Condition	0	0	0	1	0	B	0	0	Rn							Rd				0	0	0	0	1	0	0	1	Rm				Swap
Condition	0	1	I	P	U	B	W	L	Rn							Rd				OFFSET												Load/Store - Byte/Word
Condition	1	0	0	P	U	B	W	L	Rn							REGISTER LIST																Load/Store Multiple
Condition	0	0	0	P	U	1	W	L	Rn							Rd				OFFSET 1				1	S	H	1	OFFSET 2				Halfword Transfer Imm Off
Condition	0	0	0	P	U	0	W	L	Rn							Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off
Condition	1	0	1	L	BRANCH OFFSET																												Branch
Condition	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch Exchange
Condition	1	1	0	P	U	N	W	L	Rn							CRd				CPNum				OFFSET								COPROCESSOR DATA XFER
Condition	1	1	1	0	Op-1				CRn							CRd				CPNum				OP-2			0	CRm				COPROCESSOR DATA OP
Condition					OP-1			L	CRn							Rd				CPNum				OP-2			1	CRm				COPROCESSOR REG XFER
Condition	1	1	1	1	SWI NUMBER																												Software Interrupt

Abbildung: Embedded Systems Architecture, University of Texas

Instruktionsformat II

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition	0	0	I	OPCODE							S	Rn				Rs				OPERAND-2												Data processing
Condition	0	0	0	0	0	0	A	S	Rd					Rn				Rs				1	0	0	1	Rm						Multiply
Condition	0	0	0	0	1	U	A	S	Rd HIGH					Rd LOW				Rs				1	0	0	1	Rm						Long Multiply
Condition	0	0	0	1	0	B	0	0	Rn					Rd				0	0	0	0	1	0	0	1	Rm						Swap
Condition	0	1	I	P	U	B	W	L	Rn					Rd				OFFSET														Load/Store - Byte/Word
Condition	1	0	0	P	U	B	W	L	Rn					REGISTER LIST																		Load/Store Multiple
Condition	0	0	0	P	U	1	W	L	Rn					Rd				OFFSET 1				1	S	H	1	OFFSET 2						Halfword Transfer Imm Off
Condition	0	0	0	P	U	0	W	L	Rn					Rd				0	0	0	0	1	S	H	1	Rm						Halfword Transfer Reg Off
Condition	1	0	1	L	BRANCH OFFSET																											Branch
Condition	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn						Branch Exchange
Condition	1	1	0	P	U	N	W	L	Rn					CRd				CPNum				OFFSET										COPROCESSOR DATA XFER
Condition	1	1	1	0	Op-1					CRn				CRd				CPNum				OP-2			0	CRm						COPROCESSOR DATA OP
Condition					OP-1			L	CRn					Rd				CPNum				OP-2			1	CRm						COPROCESSOR REG XFER
Condition	1	1	1	1	SWI NUMBER																											Software Interrupt

Immediates (*unmittelbare* Werte)

- In die Instruktion »einkodierte« Konstante
  - `OFFSET`
  - `BRANCH OFFSET`
  - `OPERAND`
- Erspart Leseoperationen aus Speicher
- Kein vollständiger Wertebereich (32 Bits prinzipiell nicht in Instruktion verfügbar)

Datentransfer I

ARM: Load/Store-Architektur

- *Keine* Speicher-Speicher-Operationen
- Werte *müssen* in Registern verarbeitet werden

Vorgehensweise

- Daten aus Speicher in Register laden
- Daten verarbeiten (verknüpfen)
- Resultat(e) in Speicher zurückschreiben

Mensch vs. Maschine

Kompliziert für Menschen, einfach für Compiler

Datentransfer II

Datentransfer

- Register → Speicher
- Speicher → Register

Typen von Datentransfer-Instruktionen

- Laden/Speichern eines Registers (*single register load/store*)
  - Flexibelste Form
  - Aufwendig für Block-Transfers
- Laden/Speichern *mehrerer* Register (*multiple register load/store*)
  - Weniger Flexibel
  - Nützlich bei Prozeduraufrufen (siehe später)
- Register-Speicher-Austausch (*swap*)
  - Atomare Tätigkeit (ununterbrechbares load/store)
  - Notwendig in Mehrprozessor-Systemen und für Systemsoftware

Datentransfer III

Datentransfer über Basisregister

- Speicherposition in Register
- Einfachste Positionierungsmöglichkeit
- Schreiben: `str r0, [r1]`; Laden: `ldr r2, [r1]`

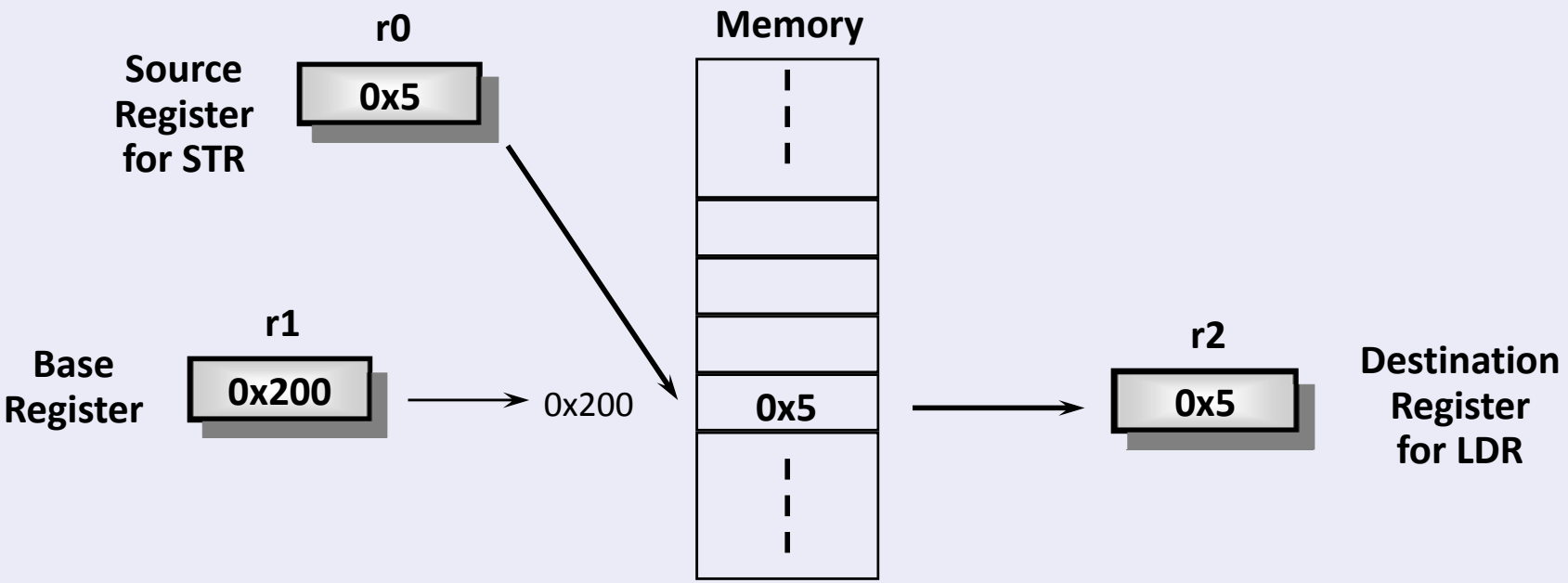
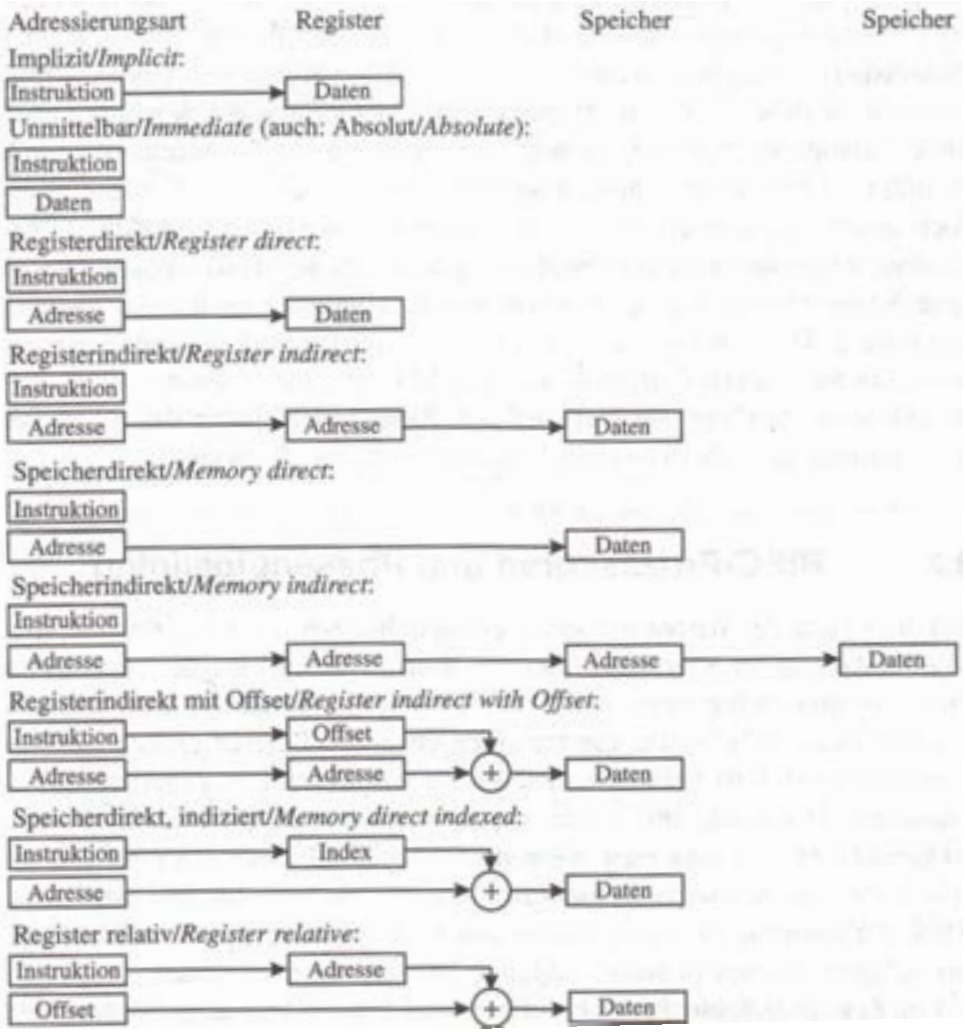


Abbildung: Embedded Systems Architecture, University of Texas

Adressierungsarten I



Adressierung

- Umwandlung: Information im Programm → tatsächlicher Ort der Information berechnen
- Strukturierte Datenbearbeitung
- Mehrteilige Operationen ☞ Evtl. mehrere Zyklen notwendig
- Busbelegung!

Abbildung: Taschenbuch der Mikroprozessortechnik, Fachbuchverlag Leipzig

Adressierungsarten II: ARM-Spezifika

Allgemeine Form: `ldr|str` ⟨R_d⟩, ⟨*adressierungsmodus*⟩

- Unmittelbarer Offset (*Immediate*):
  - `ldr` ⟨R_d⟩, `[`⟨R_n⟩, ±`offset_12]`
  - `ldr r0, [r1, #16]`
  - $\langle R_0 \rangle \leftarrow *(\langle R_1 \rangle + 16)$
- Register-Offset:
  - `ldr` ⟨R_d⟩, `[`⟨R_n⟩, ±⟨R_m⟩`]`
  - `ldr r0, [r1, -r2]`
  - $\langle R_0 \rangle \leftarrow *(\langle R_1 \rangle - \langle R_2 \rangle)$
- Skalierter Register-Offset:
  - `ldr` ⟨R_d⟩, `[`⟨R_n⟩, ±⟨R_m⟩, ⟨*Shift-Typ*⟩ `#shift_imm]`
  - `ldr r0, [r1, r2, lsl #4]`
  - $\langle R_0 \rangle \leftarrow *(\langle R_1 \rangle + (\langle R_2 \rangle << 4))$
- *Details:* Siehe ARM ARM, Kapitel A5.2 (Achtung: Datenverarbeitung ☞ Weitere Adressierungsarten!)

Adressierungsarten II: ARM-Spezifika

Auto- und Post-Indizierung

- Bisherige Adressierungen: *Pre*-Indizierung
- *Post*-Indizierung:
  - `ldr r0, [r1], #4`
  - $\langle R_0 \rangle \leftarrow *(\langle R_1 \rangle); \langle R_1 \rangle \leftarrow \langle R_1 \rangle + 4$
- *Auto*-Indizierung:
  - `ldr r0, [r1, #4]!`
  - $\langle R_0 \rangle \leftarrow *(\langle R_1 \rangle + 4); \langle R_1 \rangle \leftarrow \langle R_1 \rangle + 4$
- Analog für andere Adressierungstypen

Geschenkte Zyklen

Manuell:

```
ldr r0, [r1]
add r0, r0, #4
```

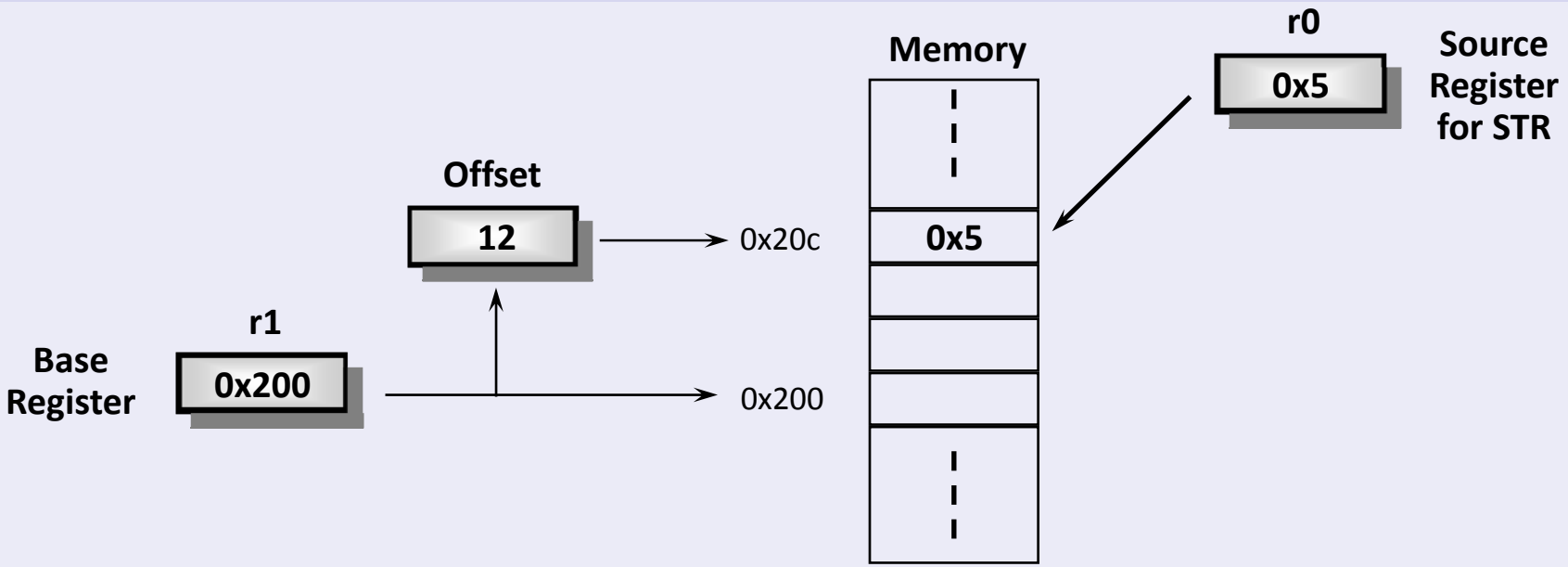
Automatisch:

```
ldr r0, [r1], #4
```

50% der Anweisungen gratis – Effekt in inneren Schleifen!

Datentransfer IV

Beispiel: Pre-Indizierung

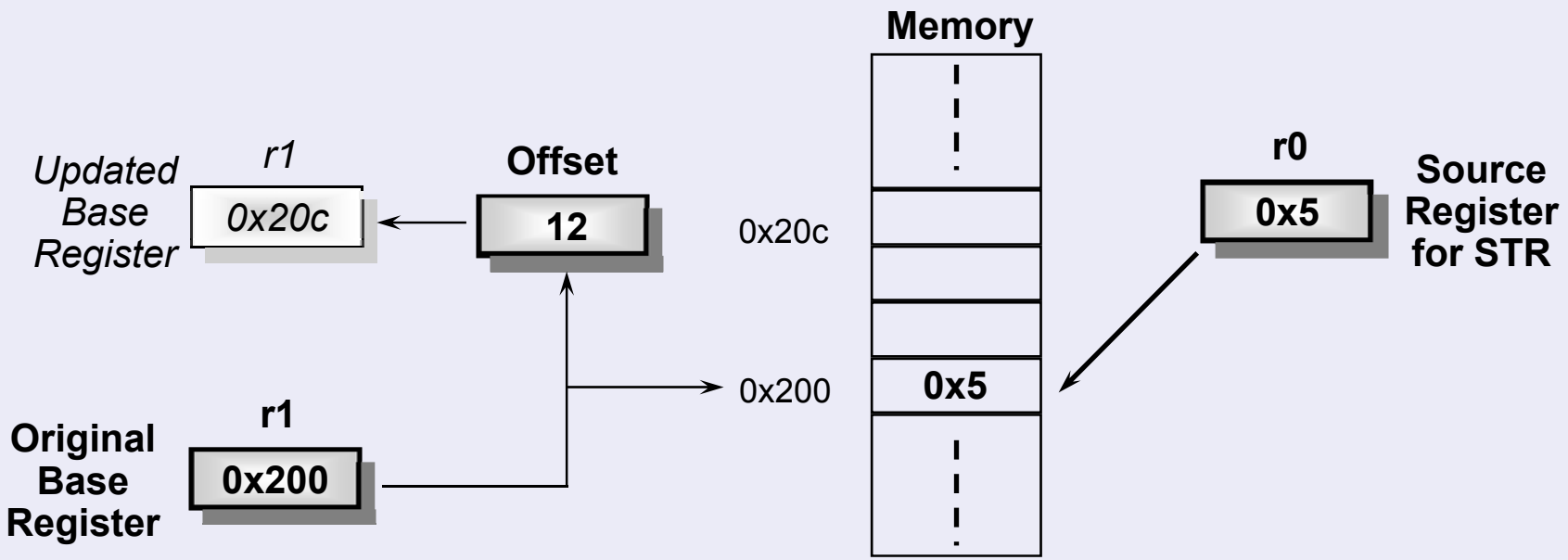


Variationen

- `str r0, [r1, #-12]`: Speichern nach $0x1f4$
- `str r0, [r1, #12]!`: $\langle R_1 \rangle$ auf $0x20c$ setzen
- `str r0, [r1, r2, lsl #2]`: Alternative Schreibweise ($\langle R_2 \rangle = 3$)

Datentransfer V

Beispiel: Post-Indizierung



Variationen

- `str r0, [r1], #-12`: $\langle R_1 \rangle$ auf $0x1f4$ setzen
- `str r0, [r1], r2, lsl #2`: Alternative Schreibweise ($\langle R_2 \rangle = 3$)

Abbildung: Embedded Systems Architecture, University of Texas

Datentransfer VI

Load/Store Multiple

- Konsekutive Speicherzellen in Register lesen: `ldm`
- Mehrere Register in konsekutive Speicherzellen schreiben: `stm`
- Relevant: Welche Register? Wachstumsrichtung im Speicher?
- Allgemeine Syntax: `ldm/stm`⟨*Modus*⟩ `Rn{!}, ⟨`*Register-Liste*⟩

Addressing mode	Description	Start address	End address	*Rn*!
`IA`	increment after	Rn	$Rn + 4*N - 4$	$Rn + 4*N$
`IB`	increment before	$Rn + 4$	$Rn + 4*N$	$Rn + 4*N$
`DA`	decrement after	$Rn - 4*N + 4$	Rn	$Rn - 4*N$
`DB`	decrement before	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

Bildquelle: Sloss et al., *ARM System Architecture*

Datentransfer VII

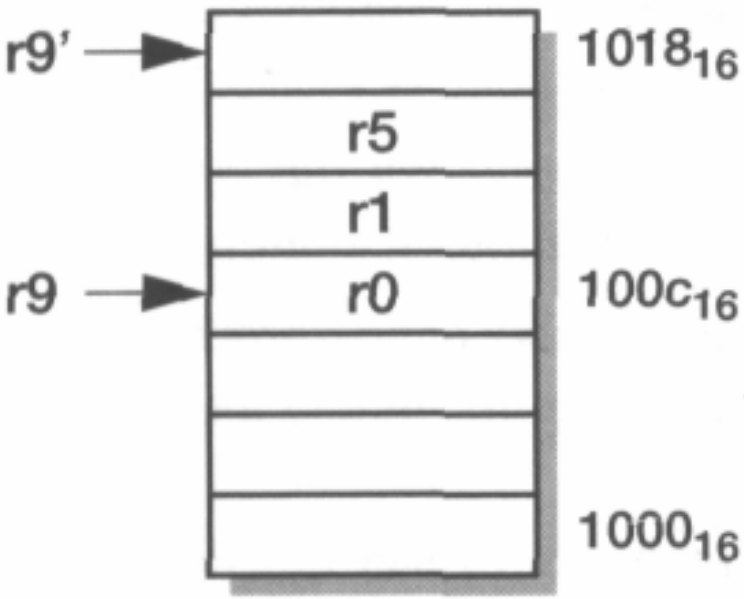
Alternative Mnemonics für Stack-Denkweise:

Synonyme

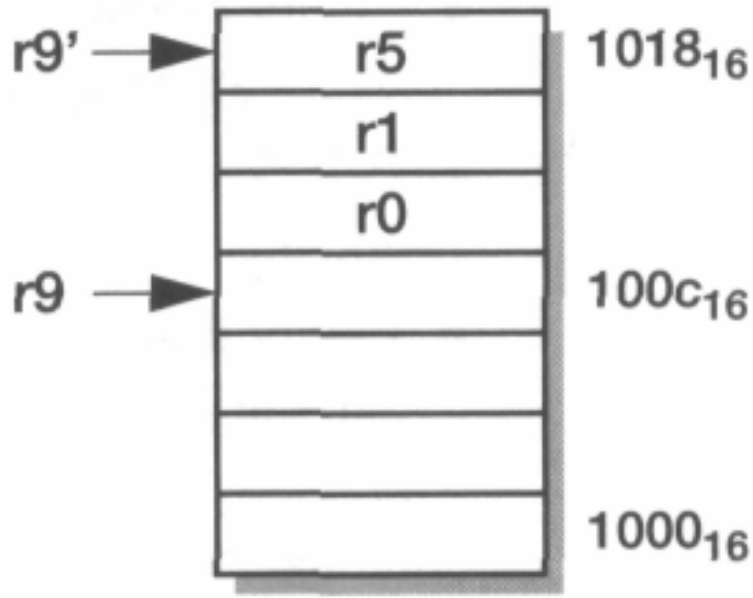
Typ	Push	Pop
Belegt, absteigend (*full descending*)	`stmfd` (`stmdb`)	`ldmfd` (`ldmia`)
Belegt, aufsteigend (*full ascending*)	`stmfa` (`stmib`)	`ldmfa` (`ldmda`)
Leer, absteigend (*empty descending*)	`stmed` (`stmda`)	`ldmed` (`ldmib`)
Leer, aufsteigend (*empty ascending*)	`stmea` (`stmia`)	`ldmea` (`ldmdb`)

Datentransfer VIII

Load/Store Multiple: Beispiele




```
STMIA r9!, {r0,r1,r5}
```

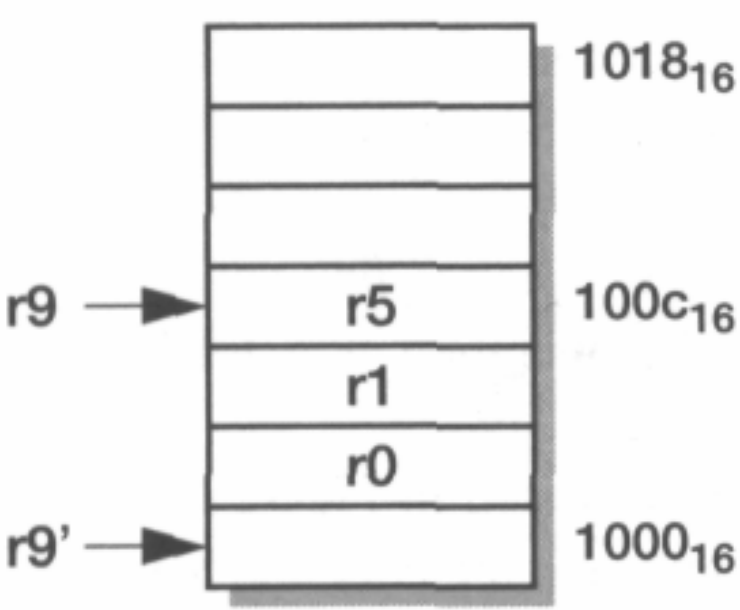



```
STMIB r9!, {r0,r1,r5}
```

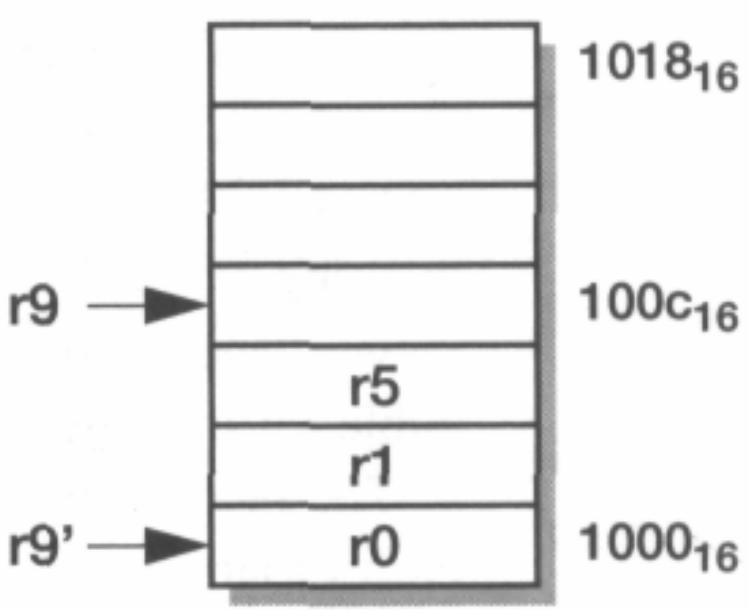
Bildquelle: Furber, *ARM system-on-chip architecture*

Datentransfer VIII

Load/Store Multiple: Beispiele




```
STMDA r9!, {r0,r1,r5}
```




```
STMDB r9!, {r0,r1,r5}
```

Bildquelle: Furber, *ARM system-on-chip architecture*

Datenausrichtung

Alignment-Anforderungen

- Alignment immer an `sizeof(`⟨*datum*⟩`)`
- Vereinfachung der Hardware-Implementierung
- Ausnahme: Container-Vektor mit 128 Bits (NEON; bsp. Cortex-A8): 8 Byte-Alignment
- *Unaligned Access*: Komplexe Möglichkeiten
  - Implementierungsabhängiges Verhalten, Shiften des Bitwerts, Compiler-Korrektur
  - Möglichst vermeiden!

Bitoptimierung

- Datenzeiger 4-Byte-aligned
- 2 LSB-Bits »frei« nutzbar
- `*(ptr & ~0x3)` zur Dereferenzierung verwenden

Verzweigungen I

Assembler: Verzweigungen

- Grundprimitiv: Sprünge
- Notwendig bei Assembler, nicht akzeptabel in höheren Sprachen
- Funktionsaufrufe: Sprung mit Rücksprung
- Verschachtelung: Buchhaltung notwendig

ARM-Besonderheiten

- Fast alle Befehle konditional ausführbar
- Statusregister (CPSR) gibt Bedingung (*Condition Code*) an

Statuscodes

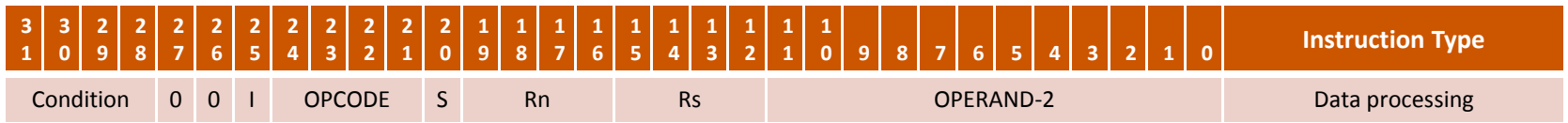
- CPSR[31:28]: NZCV
- *N*: Negatives ALU-Resultat
- *Z*: Null-Resultat
- *C*: Carry-Bit
- *V*: Überlauf (signed)

Condition Flags I

Interpration Statuscodes

Flag	Logische Anweisung	Arithmetische Anweisung
Negative	N/A	Bit 31 des Ergebnisses gesetzt (negative Zahl bei signed-Arithmetik)
Zero	Nur 0-Bits	Arithmetisches Resultat 0
Carry	1 im Carry nach Shift	Resultat größer als 32 Bits
oVerflow	N/A	Resultat größer als 31 Bits – mögliche Korruption des Sign-Bits

Condition Flags I



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Abbildung: Embedded Systems Architecture, University of Texas

Condition Flags II

- `add r0, r1, r2`
  - $\langle R_0 \rangle = \langle R_1 \rangle + \langle R_2 \rangle$
  - `add` entspricht implizit `addal` (*always*)
- `addeq r0, r1, r2`
  - `if (zero_flag_set()) {` $\langle R_0 \rangle = \langle R_1 \rangle + \langle R_2 \rangle$ `}`
  - Vorhergehende Operation setzt Flags!
  - »Kommunikation« zwischen ansonsten sequentiellen Operationen

Datenverarbeitung und Flags

- Datenverarbeitungs-Instruktionen setzen standardmäßig *keine* Flags
- Angehängtes `s` an Mnemonic notwendig!

Verzweigungen II

Explizite Verzweigung

```
    cmp r3, #0
    beq skip
    add r0, r1, r2
skip:
    ... Assembler-Code ...
```

Schleifen

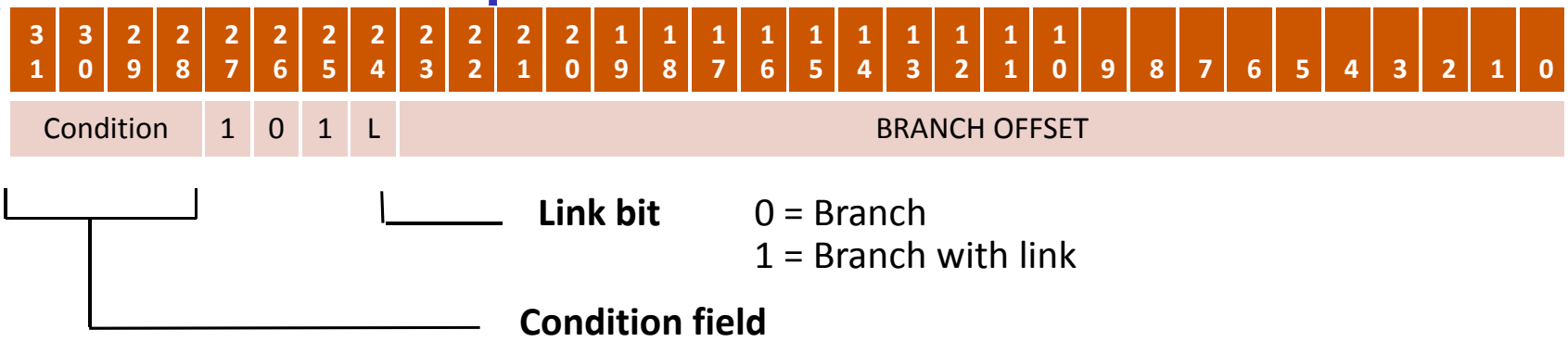
```
   @ Schleifenzähler
   @ initialisieren
   mov r0, #42
loop:

   ... Assembler-Code ...

   @ Schleifenzähler
   @ dekrementieren und
   @ Flags setzen (s-Suffix!)
   subs r0, r0, #1

   @ Wenn Ergebnis ungleich 0:
   @ Weiterer Schleifen-
   @ durchlauf
   bnz loop
```

Explizite Branch-Operationen I



Branch-Operationen

- `B label`: Verzweigung an Speicheradresse
  - Sprungziel als Immediate kodiert
  - Linksshift: Immediate << 2 (Alignment!) + Sign Extension
  - Sprung im Bereich ± 32 MiB relativ zum PC direkt abbildbar
  - Weitere Sprünge: Linker muss zusätzlichen Code einfügen
- `BL label` (*Branch with Link*)
  - Wie `B`; zusätzlich wird Adresse der nächsten Instruktion (`PC-8+4=PC-4`) in $\langle R_{14} \rangle$ (LR) gespeichert
- Direktes Schreiben in den Programmzähler $\langle R_{14} \rangle$

Speichermanipulation I

```
.data     @ Datendefinitionen
.balign 4 @ Ausrichtung an 4 Byte-Grenzen

varx:
.word 4
vary:
.word 3

.text
.balign 4 @ Programcode wird im .text-Abschnitt gespeichert
.global main
.func main
main:
@ Lokale Register: x=r0, y=r1, tmp=r2
ldr r0, x_ptr          @ Pseudo-Instruktion: &a in r0 laden
ldr r0, [r0]           @ r0 = *r0
ldr r1, y_ptr
ldr r1, [r1]           @ r2 = *r2

@ x = x+y berechnen
@ r0 ist gleichzeitig Rückgabevariable
add r0, r0, r1

bx lr
.endfunc

// Zeiger auf die Variablen
x_ptr : .word varx
y_ptr : .word vary
```

Speichermanipulation II

Pseudo-Instruktion: `ldr` ⟨R_n⟩`, pointer`

- Adresse der Konstanten zur Compile-Zeit nicht bekannt
- Linker ersetzt Platzhalter durch korrekte Adresse
- *Relokation* (Details siehe später)

Speichermanipulation III

Programmkomponenten

- `.text`: Ausführbarer Code
- `.rodata`: Statische nur-Lesen-Daten, z.B. Zeichenketten
- `.data`: Veränderliche Daten
  - Kann weiter aufgeteilt werden:
  - Initialisiert, Null-Initialisiert, Uninitialisiert

Reale Welt

- Binäraufbau deutlich komplexer
- Unterschied zwischen Layout im Speicher und Layout in ausführbarer Datei!
- Debug-Symbole, Relokationen, ABI-Informationen, Abhängigkeitsverwaltung, …
- (Einige) Details: Siehe Linker-Teil

Funktionsaufrufe I

`bl`: Branch with Link

- PC-4 in LR (*link register*) schreiben
- Sprung ausführen
- Rücksprung: `bx lr` kehrt auf darauffolgende Assembler-Anweisung zurück
  - Ebenfalls anzutreffen: `mov pc, lr`
  - Funktioniert nicht immer (seltene Fällen)
  - Verwendung überholt (*deprecated*)
- (Geschachtelte) Funktionen: Registerkonventionen notwendig!

Funktionsaufrufe II

Grundaufgabe

- Parameter an *aufgerufene* Funktion (Callee) übergeben
- Rückgabewert an aufrufende Funktion (Caller) übermitteln

Probleme

- Mehrere Rückgabetypen (C: nicht relevant)
- Gleitkommavariablen
- Umfangreiche Daten (Arrays etc.)

Lösungsmöglichkeiten

- Verwendung von Registern
- Verwendung des Stapelspeichers
- Mischform: Teils Register, teils Stack

Funktionsaufrufe III

ARM-Aufrufkonvention (genauer: *GNU EABI*)

- $\langle R_0 \rangle$–$\langle R_3 \rangle$: Prozedurargumente, *Scratch*-Register. Über Prozedurgrenzen vom Aufrufer zu sichern, wenn gewünscht (*caller-save*). Zusätzlich: $\langle R_{12} \rangle$
- Rücksprungadresse: $\langle R_{14} \rangle$ (LR)
- $\langle R_0 \rangle$, $\langle R_1 \rangle$: Rückgabewert
- $\langle R_4 \rangle$–$\langle R_{10} \rangle$, LR ($\langle R_{14} \rangle$) und FP ($\langle R_{11} \rangle$) von *aufgerufener* Funktion zu sichern (*callee-save*)
- Weitere Details: Siehe *ARM Procedure Call Standard*
  - Mehr als 4 Parameter
  - Gleitkommaargumente
  - Große `structs` (mehr als 32 Bits)
  - Variable Anzahl von Argumenten
  - Mehrere Rückgabewerte
  - …

Funktionsaufrufe III

Muster: Funktionsdefinition

```
callee:
        @ Register auf Stack sichern
        stmfd sp!, {r4-r10, fp, lr}

        @ Argumente sind in r0..r3
        ... Assembler-Code ...

        @ Modifizierte Register wiederherstellen
        ldmfd sp!, {r4-r10, fp, lr}

        mov r0, #0   @ Rueckgabewert 0 in r0
        bx lr
```

Illustration: Siehe Tafel

Funktionsaufrufe IV

```
        int test2(int a, int b, int c, int d, int e, int f) {
0x0000844c <+0>: push {r11, lr}
0x00008450 <+4>: add r11, sp, #4
0x00008454 <+8>: sub sp, sp, #16
0x00008458 <+12>: str r0, [r11, #-8]
0x0000845c <+16>: str r1, [r11, #-12]
0x00008460 <+20>: str r2, [r11, #-16]
0x00008464 <+24>: str r3, [r11, #-20]

        func1();
0x00008468 <+28>: bl 0x8578 <func1>

        return(e * f);
0x0000846c <+32>: ldr r3, [r11, #4]
0x00008470 <+36>: ldr r2, [r11, #8]
0x00008474 <+40>: mul r3, r2, r3

        }
0x00008478 <+44>: mov r0, r3
0x0000847c <+48>: sub sp, r11, #4
0x00008480 <+52>: pop {r11, pc}
```

Funktionsaufrufe V

Rekursion

```
int fak(int n) {
  if (n == 0)
     return (1);

  return(n*fak(n-1));
}
```

Vorgehensweise

- Pro Aufruf wird *ein* Activation Record angelegt
- Achtung: Stackaufbau kann »wegoptimiert« werden
  - Beispiel: Funktion verwendet keine lokalen Variablen und hat weniger als 5 Parameter

Live-Inspektion I

gdb

- `gdb` kann zur Stapelinspektion verwendet werden
- Eigentliche Intention: Debugging
- Ebenfalls nützlich für Verständnis der Mechanismen

Kommandos I

- `bt`: Backtrace
- `info frame` ⟨*n*⟩: Information (über *n*-ten Frame) anzeigen
  - Stack- und Basepointer
  - Funktionsargumente
  - Lokale Variablen
  - Verweis auf vorhergehenden Stack-Frame
  - Gespeicherte Register
- Generisch: Zugriff über `$sp`

Live-Inspektion I

gdb

- `gdb` kann zur Stapelinspektion verwendet werden
- Eigentliche Intention: Debugging
- Ebenfalls nützlich für Verständnis der Mechanismen

Kommandos II

- `break`: Breakpoint setzen
  - C-Funktionssymbol: Nach Konstruktion eines kompletten *Activation Records*
  - Speicheradresse: Assembler-Anweisung
- `disas`⟨`/m`⟩ ⟨*`symbol`*⟩: Assembler-Quelltext von `symbol` anzeigen; optional mit C-Quelltest vermischt
- `print *(unsigned int*)$fp[+k*4]`: k-ten Stackeintrag anzeigen

Live-Inspektion I

```
(gdb) info frame
Stack level 0, frame at 0xbefff6e8:
 pc = 0x8458 in test2 (call.c:13); saved pc 0x8504
 called by frame at 0xbefff718
 source language c.
 Arglist at 0xbefff6e4, args: a=1, b=2, c=3, d=4, e=23,
 Locals at 0xbefff6e4, Previous frame's sp is 0xbefff6e8
 Saved registers:
  r11 at 0xbefff6e0, lr at 0xbefff6e4
```

Bedingte Ausführung I

Befehlserweiterung

- `opcode` ☞ `opcode⟨cond⟩S`
- `S`: Condition Flag nach Ausführung setzen
- ⟨cond⟩: Bedingung, unter der Instruktion ausgeführt wird (siehe *Condition Flags I*)

Bedingte Ausführung II

```
uint32_t cpsr = 0;
volatile int val = INT_MAX;

printf("val vorher: %d\n", val);

asm("mrs %0, CPSR" : "=r" (cpsr) : );
printf("Wert von CPSR: 0x%x\n", cpsr);

// Overflow provozieren und CPSR lesen
// -> Negative Result und ALU overflowed (Bits 31 und 28)
asm("adds %0, %0, #1" : "=r" (val) : "r" (val));
asm("mrs %0, CPSR" : "=r" (cpsr) : );

// Overflow-Bit (V) und Negative Result-Bit (N) gesetzt
printf("Wert von CPSR: 0x%x, val: %d\n", cpsr, val);
```

Bedingte Ausführung II

Resultat

```
val vorher: 2147483647
Wert von CPSR: 0x60000010
Wert von CPSR: 0x90000010, val: -2147483648
```

Explizite Branch-Operationen II

Kombination

- Branch-Anweisung mit bedingter Ausführung kombiniert
- Entspricht »klassischen« Branch-Anweisungen anderer ISAs
- ARM: »Im Vorbeigehen« implementiert

Explizite Branch-Operationen III

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Abbildung: Embedded Systems Architecture, University of Texas

Ganzzahlarithmetik I

Arithmetische Operationen

⟨*operation*⟩{⟨*cond*⟩}{S} ⟨R_d⟩, ⟨R_n⟩, Operand2

- `add`, `adc`: Addition (ohne/mit Carry)
- `sub`, `sbc`: Subtraktion (ohne/mit Carry)
- `rsb`, `rsc`: Inverse Subtraktion (ohne/mit Carry): Reihenfolge verdreht

Kodierung Operand2

- Immediate
- Register
- Logischer/Arithmetischer Shift nach Immediate/Register
- Rotation (mit und ohne Erweiterung) nach Immediate/Register
- Details: $(\text{ARM})^2$, Kapitel A5.1

Ganzzahlarithmetik II

Multiplikation

- Multiplikation: `mul{`⟨*cond*⟩`}{S}` $\langle R_d \rangle, \langle R_m \rangle, \langle R_s \rangle$
  - $\langle R_d \rangle = \langle R_m \rangle \times \langle R_s \rangle$
  - Ältere ISAs: $\langle R_d \rangle$ und $\langle R_m \rangle$ dürfen nicht identisch sein
- Multiply-Accumulate: `mla{`⟨*cond*⟩`}{S}` $\langle R_d \rangle, \langle R_m \rangle, \langle R_s \rangle, \langle R_n \rangle$
  - $\langle R_d \rangle = (\langle R_m \rangle \times \langle R_s \rangle) + \langle R_n \rangle$
- *Keine* Division! (in Software zu erledigen)

Erweiterungen: Siehe ARM ARM

- Multiplikation $32 \times 32 \rightarrow 64$ Bits (*long multiply*)
- Multiplikation auf Halfwords und Word/Halfword-Kombinationen
- Parallele Arithmetik
- Gleitkommaarithmetik: VFP (später), NEON (Cortex-Ax)

Bitschieben I

Bits schieben

- ARM besitzt *keine* expliziten Bit-Shift-Anweisungen
- Alternative: *Barrel Shifter*
- In Ausführungszyklus integrierte Schiebeoperation »im Vorbeigehen«
- Shifts werden als Teil anderer Instruktionen ausgeführt

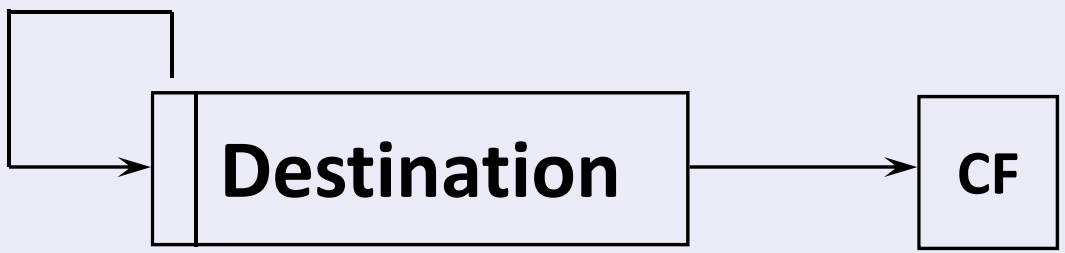
Schiebeoperationen

- Linksshift (identisch: arithmetischer und logischer Linksshift)
- Rechtsshift: Arithmetisch und logisch
- Rotieren/Rotieren mit Erweiterung

Bitschieben II

Linksshift: LSL, ASL

Arithmetischer Rechtshift: ASR



Sign bit shifted in

Bitschieben III

Logischer Rechtsshift: `LSR`

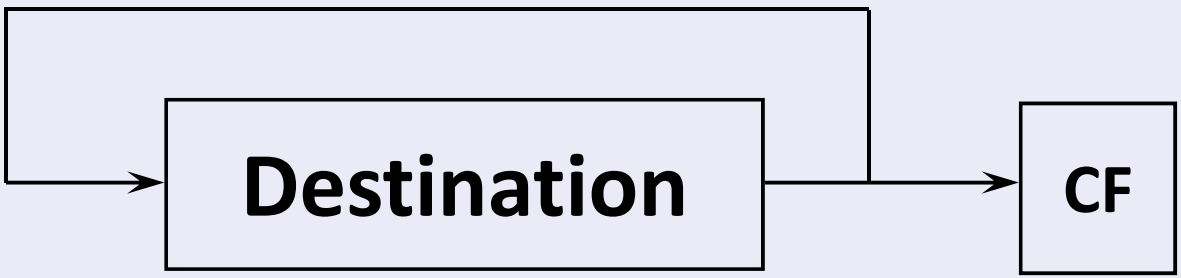
C-Standard

- Faustregel: Shifts sind arithmetisch
- Standard: Einige *undefinierte* Kombinationen!
- Compiler (GCC, clang, Visual C): Durch arithmetische Shifts implementiert

Abbildung: Embedded Systems Architecture, University of Texas

Bitschieben IV

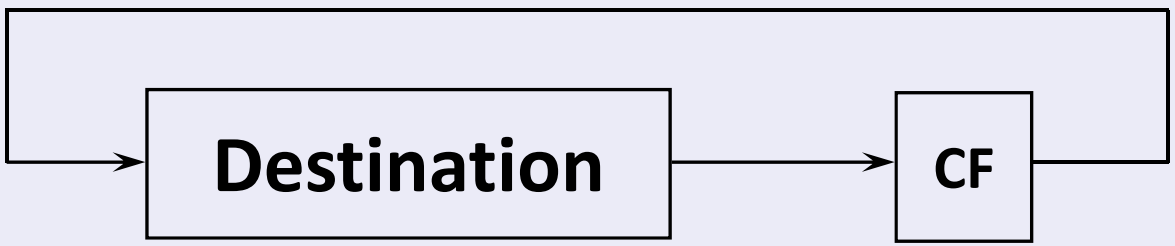
Rechts rotieren: ROR



- Bits, die aus LSB »herausfallen«, werden bei MSB angefügt
- Nicht direkt auf C-Operator abgebildet!

Bitschieben V

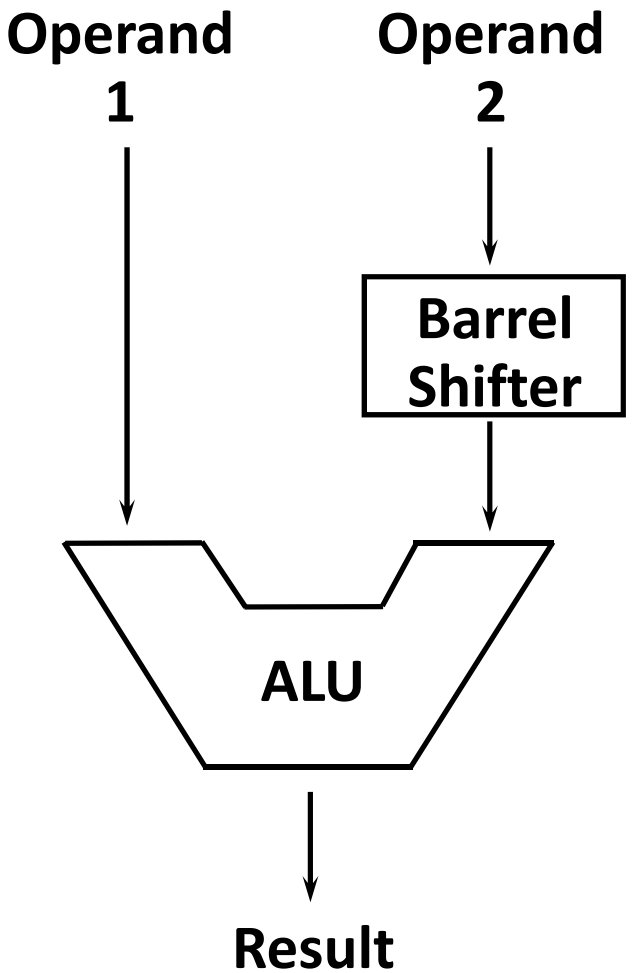
Recht rotieren, erweitert: RRX



- Carry wird als 33. Bit verwendet
- Rotation um 1 Bit nach rechts
- Kodierung: `ROR #0`
- Nicht direkt auf C-Operator abgebildet!

Abbildung: Embedded Systems Architecture, University of Texas

Bitschieben VI



Zweiter Operand

- Register mit Shift durch
  - 5 Bit Unsigned-Zahl
  - Unterstes Byte eines anderen Registers
- Immediate
  - 8 Bit-Zahl in 32 Bits eingebettet
  - Um *gerade* Anzahl von Bits nach rechts rotiert
  - Nur bestimmte Immediates darstellbar (Assembler wählt passende Kodierung)

Abbildung: Embedded Systems Architecture, University of Texas

Bitschieben VII

Beispiel

```
int mul12(int x) {
  return x*12;
}
```

Unoptimierter Code

Argument x ist in $\langle R_0 \rangle$

```
mov   r3, r0
lsl   r3, r3, #1
add   r3, r3, r0
lsl   r3, r3, #2
```

Optimierter Code

```
add   r0, r0, r0, lsl #1
lsl   r0, r0, #2
```

Rechnung: Siehe Tafel

Sonstiges

Halfwords und Bytes

- Load und Store für Werte < 32 Bits
- Halfwords (16 Bits): `ldrh`, `strh`
- Bytes (8 Bits): `ldrb`, `strb`
- Signed: `ldrsb` etc.

Logische Operationen

- `and`, `orr`, `eor`
- `orn` (or not), `bic` (bit clear/and not)

Sign Extension

- `sxth` (Halfword nach Word), `sxtb` (Byte nach Word)
- Weitere Spezialfälle: Siehe ARM ARM

Vollständiges Assembler-Beispiel I.1

Skalarprodukt

- Vektoren $\vec{a}, \vec{b} \in \mathbb{R}^n$ (d.h. mit Länge n)
- Skalarprodukt: $\vec{a} \cdot \vec{b} = |\vec{a}| \times |\vec{b}| \times \cos(\vec{a}, \vec{b})$

$$s = \sum_{i=1}^{n} a_i \cdot b_i$$

C-Äquivalent

```
for (int i=0, s=0; i < n; i++) {
      s = s + a[i]*b[i]
}
```

Vollständiges Assembler-Beispiel I.2

```
.globl do_mul      @ Parameter: r0 = N, r1 = &a, r2 = &b
.func do_mul

do_mul:
        push {r4, r5, r6, r8}
        mov r3, #0              @ Schleifenindex i
        mov r8, #0              @ ⟨R8⟩ = Array-Index idx
        mov r5, #0              @ Zwischensumme
.Lloop:
        ldr r4, [r1, r8]        @ a[i] in ⟨R4⟩ laden
        ldr r6, [r2, r8]        @ b[i] in ⟨R6⟩ laden
        mul r4, r4, r6          @ ⟨R4⟩ = a[i] * b[i]
        add r5, r5, r4          @ s = s + ⟨R4⟩
        add r3, r3, #1          @ i = i + 1
        add r8, r8, #4          @ idx = idx + 4 (wg. sizeof(c) = 4)
        cmp r3, r0              @ Schleife beenden
        blt .Lloop              @ i < N: Continue
        mov r0, r5
        pop {r4, r5, r6, r8}
        bx lr
.endfunc
```

Vollständiges Assembler-Beispiel I.3

```
#include<stdio.h>

extern int do_mul(int N, int c[], int x[]);

int main() {
  int c[10], int x[10];
  int count, res;

  // ... Arrays initialisieren ...

  res = do_mul(10, c, x);

  printf("Result: %d\n", res);

  return(0);
}
```

Vollständiges Assembler-Beispiel II.1

Stringlängen bestimmen

- C-String: Null-Terminiert
- `unsigned strnlen(char* string, unsigned n)`
  - Auch in stdlibc vorhanden
  - Obergrenze für Stringlänge vorgegeben
  - Warum Obergrenze?

Vollständiges Assembler-Beispiel II.2

```
.data
.balign 4
string: .asciz "Das ist ein Teststring!"
```

Vollständiges Assembler-Beispiel II.3

```
.text
.balign 4
.global strnlen
.func strnlen
strnlen:
        @ Parameter: Pointer auf String (r0), maximale Laenge (r1)
        @ Rueckgabewert: String-Laenge in r0
        @ Lokale Variablen: Aktuelles Byte (r2), Laenge (r3)

        mov  r3, #-1       @ Laenge auf -1 initialisieren
.Liter:
        add  r3, r3, #1    @ Laenge weiterzaehlen
        ldrb r2, [r0], #1  @ Naechstes Byte laden, post-inkrement

        cmp   r3, r1       @ Maximale Laenge erreicht?
        cmpne r2, #0       @ Nein: Byte gleich Nullbyte?

        bne .Liter         @ Nein: Weiterzaehlen
        mov r0, r3         @ Ja: Laenge zurueckgeben
        bx lr
.endfunc
```

Koprozessoren I

Koprozessoren: Allgemeines

- Allgemeine Schnittstelle: Unterstützung bis zu 16 Koprozessoren
- Jeder Koprozessor: Bis zu 16 private Register
  - Architektur: »…*of any reasonable size*«
  - *Keine* Beschränkung auf 32 Bits
- Kann, muss aber nicht als separater Chip ausgeführt werden
- Load-Store-Architektur
  - Koprozessor-Register↔Speicher
  - ARM-Register↔Koprozessor-Register
  - Operationen auf Koprozessor-Registern

Koprozessoren II

Koprozessor-Befehle

- `ldc`, `stc`: Koprozessor-Register aus/in RAM laden/speichern
- `mcr`, `mcrr`: Von einem (zwei) ARM-Register(n) in Koprozessor-Register schreiben
- `mrc`, `mrrc`: Koprozessor-Register in ein (zwei) ARM-Register schreiben
- `cdp`: Datenoperation ausführen

Architekturerweiterung

- Alle Befehle mit angehängter `2` verfübgar (`ldr2`, `mrrc2`, ...)
- Instruktions-*Adressraumerweiterung* ab ARMv5/ARMv6
- Mehr Möglichkeiten für Koprozessoren
- Nur unkonditional ausführbar!

Koprozessoren III

Gleitkommarechnungen: VFPv1, VFPv2

- Koprozessoren 10 und 11: FPUs für einfache und doppelte Genauigkeit
- Optional für ARM-Maschinen – auf Raspberry Pi vorhanden!
- Vektoroperationen (SIMD) möglich
- Assembler: Syntaktischer Zucker
- Details: Siehe $(\text{ARM})^2$, Kapitel C

Beispiel

```
void do_float() {
  volatile float d1, d2, d3;
  d1 = 3.01;
  d2 = 17.99;

  d3 = d1*d2;
}
```

Koprozessoren IV

```
(...)
volatile float d1, d2, d3;
d1 = 3.01;
 4:    ldr      r3, [pc, #32]    ; 2c <do_float+0x2c>
 8:    str      r3, [sp, #4]
d2 = 17.99;
 c:    ldr      r3, [pc, #28]    ; 30 <do_float+0x30>
10:    str      r3, [sp, #8]

d3 = d1*d2;
14:    vldr     s14, [sp, #4]
18:    vldr     s15, [sp, #8]
1c:    vmul.f32 s15, s14, s15
20:    vstr     s15, [sp, #12]

(...)
2c:    .word    0x4040a3d7
30:    .word    0x418feb85
```

Virtueller Adressraum: Illusionen

Illusion 1

- Mehrere Prozesse laufen »gleichzeitig« auf dem System
  - Mehrprozessor: Maximal N möglich bei N Cores
  - $n > N$ Tasks: In schneller Folge zwischen Prozessen wechseln

Illusion 2

- Jedem Prozess steht der komplette adressierbare Adressraum zur Verfügung (ARM: 4 GiB)
  - *Virtuelle* vs. *physikalische* Adressen
  - CPU/BS erledigen Umwandlung aus Anwendungssicht transparent
- Einfache eingebettete Systeme: Nicht notwendigerweise der Fall!

Virtueller Adressraum II

Komplizierende Faktoren

- Randomisierung
- Betriebssystem-Konventionen
- Hohe Komplexität durch dynamische Bibliotheken und technische Details

Inhalt

Bibliotheken, Linken & ELF-Binärformat

- Bibliotheken: Sammlungen haufig gebrauchter Funktionen
- Binden (*Linking*): Kombination von Objektdateien und Bibliotheken zu ausführbarem Objekt
- ELF-Binärformat: Container für Binärobjekte und Metainformationen

Binden: Notwendigkeit

Modularität

- Ermöglichst separates Übersetzen von Programmteilen (*compilation units*)
- Ermöglicht *Bibliotheken* als Funktionensammlungen (libc, libm, QT, …)

Effizienz

- Änderungen in einer Datei machen keine komplette Neuübersetzung notwendig
- Nur tatsächlich benutzte Funktionen eingebunden

Binden: Beispiel

main.c

```
int buf[2] = {0xdeadbeef,
    0xaffe}

int main() {
  swap();
  return 0;
}
```

- Global: `buf`, `main`
- Extern: `swap`

swap.c

```
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void swap() {
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

- Global: `bufp0`, `swap`
- Lokal: `bufp1`
- Extern: `buf`

Binden: Arbeitsweise

Zwei Phasen notwendig

Symbolauflösung

- Code *definiert* und *referenziert* Symbole:
  - `void func(int a) {…}`
  - `func(17)`
  - `int *ptr; int var; ptr = &var;`
- Symboldefinitionen: Symboltabelle
- Binder: Referenzen mit *genau einer* Definition verbinden

Relokation

- *Code* und *Daten* aus unterschiedlichen Objekten vereinigen
- Symbole aus *relativen* Adressen im Objekt an *absolute* Adressen im Speicher verschieben
- Referenzen auf Symbole mit korrekt auflösen

Binden: Typen von Objektdateien

- Relozierbare Objektdatei (`file.o`)
  - Enthält Code und Daten
  - Kombinierbar mit anderen Objektdateien
  - Nicht direkt ausführbar
  - C: Objektdatei aus genau einer Quelldatei (aber potentiell vielen `.h`-Headern) erzeugt
- Ausführbare Datei (`a.out`)
  - Enthält Code und Daten
  - Direkt ausführbar: kopieren, Umgebung aufsetzen (Argumente), an Startadresse (`_start`) springen
- Bibliothek (*shared object*; `file.so`/`file.dll`)
  - Variante relozierbarer Objektdatei
  - Wird über *dynamischen Binder* mit Programm verbunden
  - Zur Ladezeit (`ld.so`) oder zur Laufzeit (`dlopen(), dlsym()`)

Relokationen durchführen I

Demonstration: Siehe Rechner

- `nm main.o; nm swap.o`
  - `D`: Datensektion
  - `T`: Textsektion
  - `U`: Undefiniertes Symbol
  - `B`: BSS (nicht initialisiert)
  - Kleinbuchstaben: Lokales Symbol
- `objdump -r -t -d -z -S main.o -j.text -j.data`
- `objdump -r -t -d -z -S swap.o -j.text -j.data`
- `objdump -r -t -d -z -S main -j.text -j.data`

Relokationen durchführen II

Konvention Initialwerte

Relokation verändert Immediates (teilweise Instruktion selbst, bsp. `add` statt `sub`):

- Relokation Datum: Initialwert sign-extended auf 32 Bits
- Relokation Instruktion: Immediate wird als Addendum verwendet
- Abweichungen/Details: Siehe *ELF for the ARM(R) Architecture*

Relokation `R_ARM_CALL`

Zielwert: $S + A - P$ (siehe ARM ELF S. 26)

- S: Adresse Symbol (d.h. Ziel)
- A: Addendum
- P: Adresse Relokation (d.h. veränderter Speicherbereich)

Relokationen durchführen II

C-Code für `R_ARM_CALL`-Relokation

```
// S sei bereits bekannt
A = *P; // Instruktion laden

// Sign Extension 24->30 Bit
if (A & 0x00800000)
    A |= 0x3F000000;
A = A << 2; // Konvention fuer Branch-Instruktion; jetzt 32 Bit

tmp = S + A - P; // Relokation R_ARM_CALL

// Skalieren und untere 24 Bits auswaehlen
tmp = (tmp >> 2) & 0x00FFFFFF;
*P = (*P & 0xFF000000) | tmp;  // Immediate zurueckschreiben
```

Beispiel: Siehe Tafel/Rechner; *ARM ARM* S. A4-10

Statische Bibliotheken I

Funktionssammlungen: Schlechte Ansätze

- Alle Bibliotheksfunktionen in großer Datei sammeln
- Jede Funktion in separater Quelldatei definieren (ala Matlab)

Alternative: Statische Bibliotheken

- Jede Funktion in eine Objektdatei übersetzen
- Objektdateien zu *Archiv* verbinden
- Linker vereinfacht Selektion:
  - Über alle Objektdateien (und Archive) iterieren
  - Binärobjekt nach undefinierten Symbolen durchsuchen
  - Nächste Objektdatei: Bislang nicht aufgelöste Symbole auflösen, wenn definiert
- Nur notwendige Funktionen in endgültiger Binärdatei

Statische Bibliotheken II

Demonstration

- Erstellen & Inhalt von Archiven: `ar`
- Binden & Reihenfolge: `ld`, `collect2`

Anwendungen

- Komplett eigenständige Binaries
- Schnellere Ladezeiten, evtl. kleinere Binärobjekte
- Eingebettete Systeme mit unveränderlichen Aufgaben
- Geringe technische Komplexität

Dymamische Bibliotheken I

Nachteile statischer Bibliotheken

- Binärobjekte benötigen Platz für duplizierte Funktionen
  - auf der Festplatte
  - im Speicher
- Updates signifikant erschwert: N Bibliotheksnutzer ☞ N Updates erforderlich

Moderner: Dynamische Bibliotheken

- Dynamische Bibliotheken: `lib.so`, `lib.dll`
- Zur Lade- oder Laufzeit gebunden
- Können zwischen Programmen geteilt werden (notwendig: Virtueller Speicher, siehe später)

Dynamische Bibliotheken II

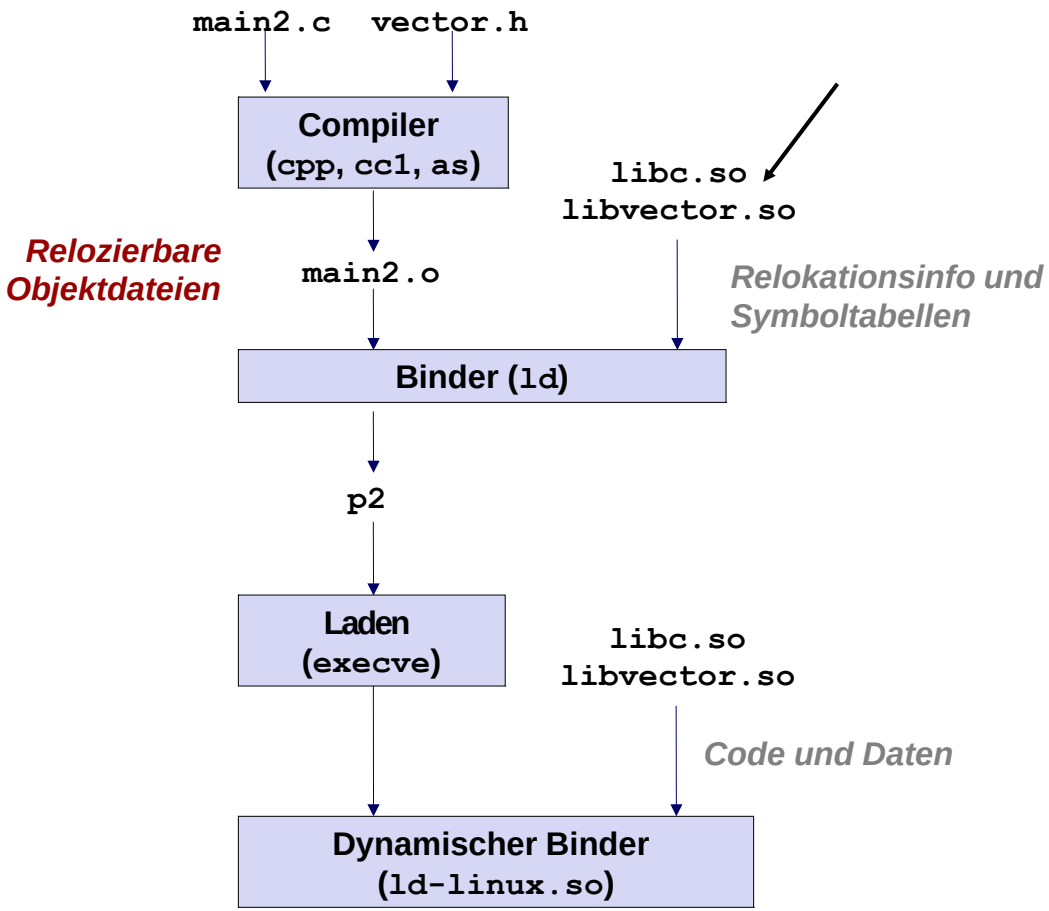


Abbildung basiert auf Bryant & O'Hallaron

Dynamische Bibliotheken III

Beispiel für `dlopen()`, `dlsym`

```
void *handle;
void (*fun)(int, int, int*);

// Keine Fehlerbehandlung - siehe man-pages fuer die
// entsprechenden Details
handle = dlopen("libsepp.so", RTLD_LAZY); // Shared Objekt laden
fun = dlsym(handle, "hypertrichter");     // Funktion suchen

int res;
fun(1,7, &res); // Aufgerufen wird hypertricher()
```

Starke und schwache Symbole I

Unterschiedliche Symbolarten

- *Starke* Symbole: Funktionen, initialisierte globale Variablen
- *Schwache* Symbole: Nicht-initialisierte globale Variablen

Binder-Regeln

- Mehrere gleichnamige starke Symbole sind nicht möglich
- Referenze auf schwache Symbole werden durch starke Symbole befriedigt (d.h. starke überschreiben schwache Symbole)

Starke und schwache Symbole II

eins.c	zwei.c	Kommentar
`func() {...}`	`func() {...}`	Linkerfehler
`int x;`	`int x;`	Globales `x` *ohne* Initialwert
`u16 x; u16 y;`	`u32 x;`	Zugriff auf `zwei.c:x` *könnte* `eins.c:y` überschreiben
`u16 x=0; u16 y=0;`	`u32 x;`	Zugriff auf `zwei.c:x` wird `eins.c:y` überschreiben

ELF-Binärformat I

Executable and Linkable Format

- Sektionen, Relokations- und Metainformationen: Containerformat notwendig
- Probleme essentiell auf allen Architekturen identisch
- Universelle Lösung: ELF
- *Keine* Möglichkeit zur Übertragung von Binaries zwischen Architekturen und Systemen!
  - Architekturen: Völlig unterschiedliche Assembler-Binärobjekte in gleicher Verpackung
  - Systeme: Andere Systemaufrufmechanismen, Semantik der Aufrufe, vorhandene Bibliotheken, …

ELF-Binärformat II

`readelf`

- `readelf -h`: Allgemeine Informationen
- `readelf -S`: Vorhandene Sektionen
- `readelf -r`: Relokationsinformationen
- `readelf -l`: Program header table (nur für ausführbare Dateien)
- Symboltabelle: Drei Sektionen
  - `.symtab`: Abbildung Symbol/Wert (*keine* Debug-Info!)
  - `.strtab`: Zuordnung Symbolnamen zu numerischen Identifiern
  - `.hash`: Hashtabelle zum schnellen Finden von Symbolen
- Details: Siehe Demo

Details: Siehe PLKA Kapitel E, ELF-Standard

ELF-Binärformat III

Beispiel: Symbolinformationen

```
struct Elf32_Sym {
    Elf32_Word    st_name;  // Symbol name (index into string
    table)
    Elf32_Addr    st_value; // Value or address associated with
                            // the symbol
    Elf32_Word    st_size;  // Size of the symbol
    unsigned char st_info;  // Symbol's type and binding
    attributes
    unsigned char st_other; // Must be zero; reserved
    Elf32_Half    st_shndx; // Which section (header table
    index)
                            // it's defined in
}
```

Inhalt

Hochsprachen & Compiler I

Compiler: Aufgaben I

- Makroersetzungen durchführen (syntaktische Transformation)
- Syntaxprüfung (parsen)
  - Zeichen-Eingabestrom in *Tokens* zerlegen: Bezeichner, Zahl, Schlüsselwort, …
  - Grammatikalische Prüfung (Grammatik, typisch: LR(k))
    - Theoretische Informatik: Deterministischer Kellerautomat
    - Tools: Yacc, Bison, ANTLR, CoCo, XText, …
    - Abstrakten Syntaxbaum (AST) aufbauen
- Semantische Analyse: Verwendung von Typen, Anzahl Prozedurparameter, …

Hochsprachen & Compiler II

Korrektes Programm

Benutzerfehler (inkorrekte Syntax, falsche Funktionsaufrufe etc.) in Folgeschritten ausgeschlossen

Compiler: Aufgaben II

- Optimierung (Größe, Geschwindigkeit)
- Transformation in Assembler-Code (Architektur- und CPU-spezifische Optimierungen)
- In Binärcode übersetzen
  - Assembler → Objektdatei
  - Linken (Relokation, Einbinden Bibliotheken etc.)

Syntaxanalyse I

Praktische Sprachklassen

- Kontextfreie Sprachen
  - Automatenmodell: Nicht-deterministischer Kellerautomat
  - Wortproblem in $\mathcal{O}(n^3)$ entscheidbar: CYK-Algorithmus
- LR(k)
  - Eingabe von *links* nach rechts gelesen
  - Syntaxbaum ist *rechts*ableitung
  - k Tokens »Vorausschau« (*Lookahead*) möglich, um Entscheidung über Regelanwendung zu treffen
  - Teilmenge deterministisch kontextfreier Sprachen (det. Kellerautomat)
  - Wortproblem in $\mathcal{O}(n)$ entscheidbar

Syntaxanalyse II

Beispiel: Bison

- Compiler-Compiler bzw. Compiler-Generator
- Generiert C-Routinen aus Grammatik-Regeln
- Ausführen der C-Routinen ☞ Syntaxbaum (Datenstruktur)

Beispiel: Flex

- *Lexical analyser*: Erkennt Schlüsselwörter anhand regulärer Ausdrücke
- Vorgeschaltet zu Bison ☞ Erleichterung Parsing
- Beispiel: String »int« ☞ Token `INT`, String »char« ☞ Token `CHAR`, …
- Beispiel: String »1.234« ☞ Token `NUM` mit Wert `1.234`

Syntaxanalyse III

Beispiel: Arithmetische Ausdrücke

$$
\begin{aligned}
\langle Exp \rangle \quad \rightarrow \quad & \langle Exp \rangle \texttt{ + } \langle Exp \rangle \mid \langle Exp \rangle \texttt{ - } \langle Exp \rangle \\
& \mid \langle Exp \rangle \texttt{ * } \langle Exp \rangle \mid \langle Exp \rangle \texttt{ / } \langle Exp \rangle \\
& \mid \langle Exp \rangle \texttt{ ** } \langle Exp \rangle \\
& \mid \texttt{(} \langle Exp \rangle \texttt{)} \\
& \mid \langle Number \rangle
\end{aligned}
$$

Probleme

- Operator-Prioritäten: Punkt vor Strich:
  `a*b+c`$= (a * b) + c$, nicht $a * (b + c)$
- Assoziativität: `a**b**c`$= a^{(b^c)}$ oder `a**b**c`$= (a^b)^c$?

Syntaxanalyse III

Beispiel: Arithmetische Ausdrücke

$$\begin{aligned}\langle Exp\rangle \quad \rightarrow \quad & \langle Exp\rangle + \langle Exp\rangle \mid \langle Exp\rangle \text{ - } \langle Exp\rangle \\ & \mid \langle Exp\rangle * \langle Exp\rangle \mid \langle Exp\rangle / \langle Exp\rangle \\ & \mid \langle Exp\rangle ** \langle Exp\rangle \\ & \mid (\langle Exp\rangle) \\ & \mid \langle Number\rangle\end{aligned}$$

Probleme

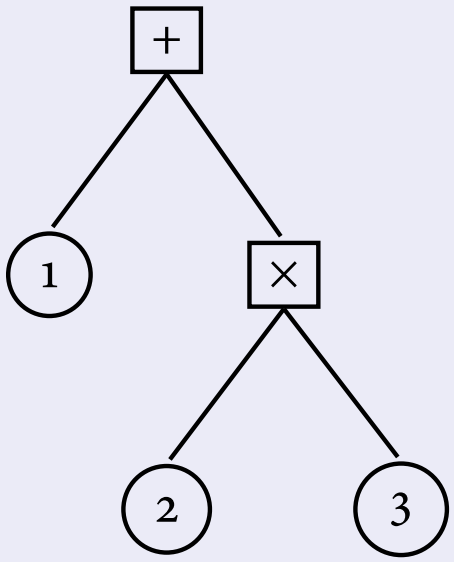
- Operator-Prioritäten: Punkt vor Strich:
  `a*b+c`$= (a * b) + c$, nicht $a * (b + c)$
- Assoziativität: `a**b**c`$= a^{(b^c)}$ oder `a**b**c`$= (a^b)^c$?

Syntaxanalyse IV

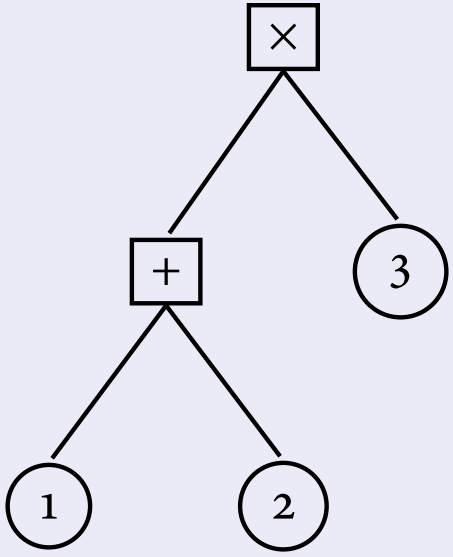
Konkrete und abstrakte Syntax

- *Konkrete Syntax*: Lineare Notation, Zusatzsymbole zur Eindeutigkeit/Überschreiben von Prioritätsvorgaben
- *Abstrakte Syntax*: Datenstruktur, keine Zusatzsymbole

$1 + 2 \times 3$



$(1 + 2) \times 3$



Syntaxanalyse V

Definition: Baum

Ein Baum $B = (V, E)$ besteht aus einer endlichen Menge von Knoten V und einer Menge geordneter Paare $E \subseteq V \times V$ mit den Eigenschaften

- Es gibt genau einen *Wurzelknoten* v_r, der keinen Vorgänger hat:

$$\exists v_r \in V, \forall v \in V : (v, v_r \notin E).$$

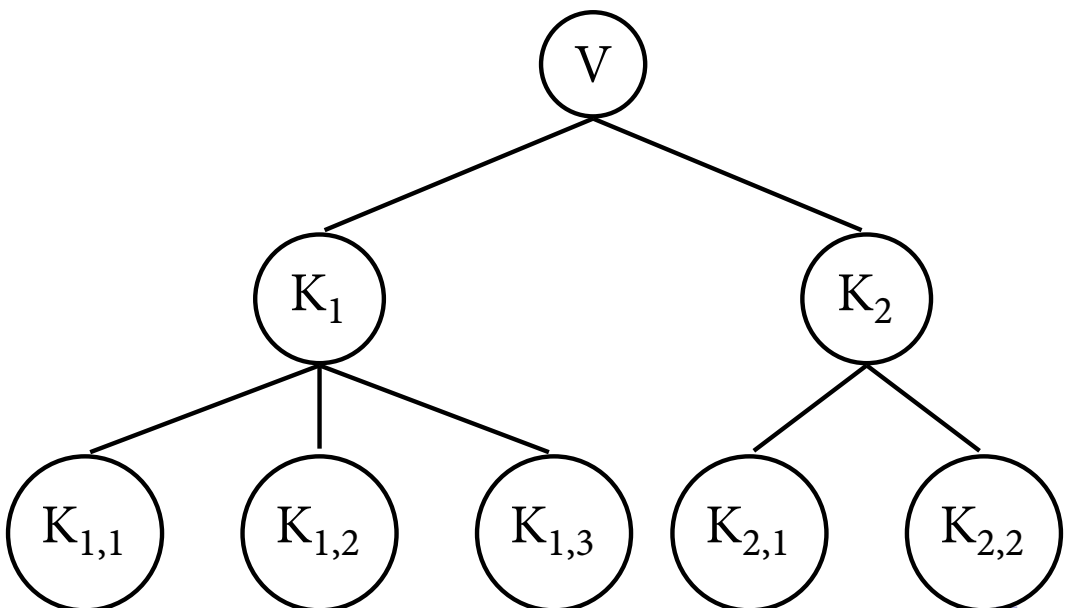
- Jeder Knoten (bis auf den Wurzelknoten) hat einen eindeutigen Vorgänger:

$$\forall v_c \in V \backslash \{v_r\}, \exists v_f \in V : (v_f, v_c) \in E \wedge \nexists v \in V \backslash \{v_f\} : (v, v_c) \in E.$$

Syntaxanalyse VI

Bäume: Datenstruktur für AST

- *Kante*: Beziehung Vorgänger/Nachfolger
- *Grad*: Anzahl unmittelbarer Nachfolger eines Knotens
- *Blatt*: Knoten *ohne* Nachfolger
- *Geschwister* (Siblings): Unmittelbare Nachfolger eines Knotens



Syntaxanalyse VII

Rechenoperationen

```
enum arith_op {
  TIMES, DIV, PLUS,
  MINUS, EXP, NUMBER
};
```

Knoten-Datentyp

```
typedef struct expression {
  enum arith_op op;

  union {
    int value;

    // Rechenoperation
    struct {
      struct expression *left;
      struct expression *right;
    } operands;
  };
} expression_t;
```

Semantische Analyse I

Unterschiedliche Datentypen

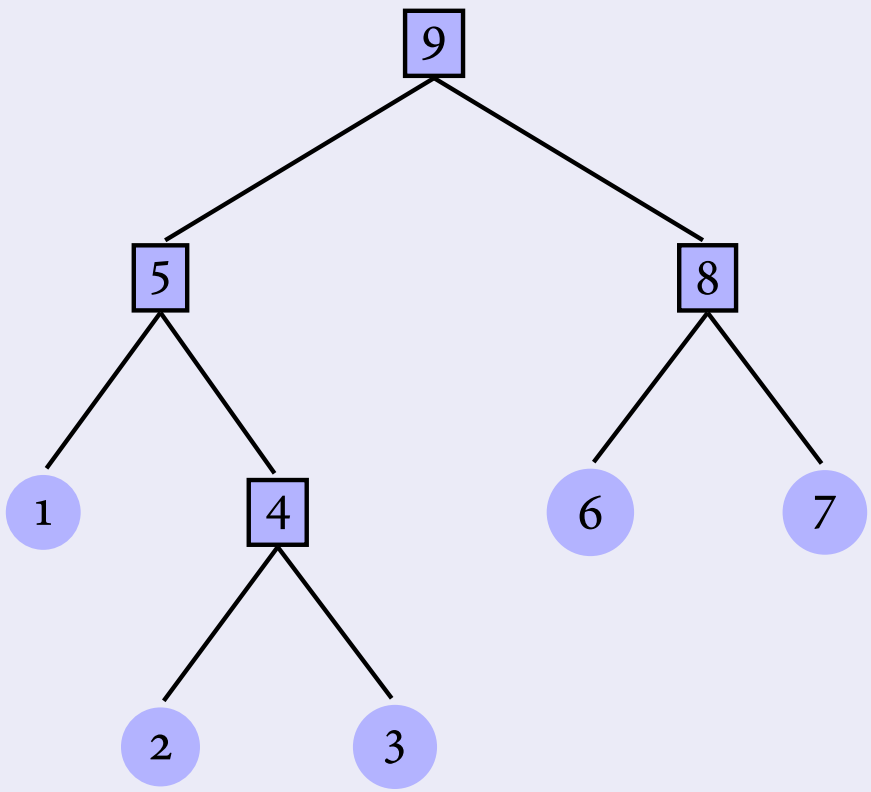
- Rechenoperation auf unterschiedlichen Datentypen: *Typkonversion*
  - $1 + 2 = 3$: int $\times$ int $\rightarrow$ int
  - $1 + 3.14 = 4.14$: int $\times$ float $\rightarrow$ float
  - $(1 + 2) * 3.13$: Zunächst int $\times$ int $\rightarrow$ int, dann int $\times$ float $\rightarrow$ float
- Konvertierung entsprechend Hochsprachen-Semantik
- *Annotation* AST erforderlich

Algorithmus

- Linksrekursiver Durchlauf des Baums
- Annotation beginnend bei Blättern
- Vaterknoten: Typ aus Blättern ermitteln

Semantische Analyse II: Baumdurchlauf

Linksrekursiver Durchlauf



Semantische Analyse III: Datenstruktur

Rechenoperationen

```
enum arith_op {
  TIMES, DIV, PLUS,
  MINUS, EXP, NUMBER
};

enum arith_type {
  INTEGER, FLOAT,
  UNDETERMINED
};
```

Knoten-Datentyp

```
typedef struct expression {
  enum arith_op op;
  enum arith_type type;

  union {
    // Zahl (Konstante)
    union {
      int ival;
      float fval;
    } value;

    // Rechenoperation
    struct {
      struct expression *left;
      struct expression *right;
    } operands;
  };
} expression_t;
```

Semantische Analyse V: Annotation

```
enum arith_type annotate_tree(expression_t *e) {
  if (e->type == UNDETERMINED) {
    enum arith_type lhs, rhs;
    lhs = annotate_tree(e->operands.left);
    rhs = annotate_tree(e->operands.right);

    // Beide Unterausdruecke Ganzzahlen: Gemeinsamer
    // Ausdruck wird Ganzzahl, anderenfalls Gleitkommazahl
    if (lhs == rhs && lhs == INTEGER) {
      e->type = INTEGER;
    } else {
      e->type = FLOAT;
    }
  }

  // Knoten jetzt sicher mit einem Typ annotiert
  return(e->type);
}
```

Syntaxanalyse VIII

Flex: Lexikalische Analyse

- Eingabe Zeichen für Zeichen lesen
- Mit RegExps in höherwertiges *Token* umwandeln

```
NUMBER          [0-9]+
FLOAT           [0-9]*\.[0-9]+
DOUBLE          [0-9]*\.[0-9]+d
WS              [ \r\n\t]*
%%
{WS}            { /* Leerzeichen ignorieren */ }
{NUMBER}        { sscanf(yytext, "%d", &yylval->value_int);
                  return TOKEN_NUMBER; }
{FLOAT}         { sscanf(yytext, "%f", &yylval->value_float);
                  return TOKEN_FLOAT; }
{TIMES}         { return TOKEN_TIMES; }
{DIV}           { return TOKEN_DIV; }
(...)
{RPAREN}        { return TOKEN_RPAREN; }
```

Syntaxanalyse IX

Bison

- Bison: Grammatik ☞ C-Code
- Bei Ausführung: C-Schnipsel erzeugt Eintrag in Datenstruktur
- Resultat/Komponenten über Meta-Symbole `$$`, `$`⟨n⟩ repräsentiert

```
expr:
      expr TOKEN_PLUS  expr { $$ = create_ast_node(PLUS, $1, $3); }
    | expr TOKEN_MINUS expr { $$ = create_ast_node(MINUS, $1, $3); }
    | expr TOKEN_TIMES expr { $$ = create_ast_node(TIMES, $1, $3); }
    | expr TOKEN_DIV   expr { $$ = create_ast_node(DIV, $1, $3); }
    | expr TOKEN_EXP   expr { $$ = create_ast_node(EXP, $1, $3); }
    | TOKEN_LPAREN expr TOKEN_RPAREN { $$ = $2; }
    | TOKEN_NUMBER { $$ = create_integer($1); }
    | TOKEN_FLOAT  { $$ = create_float($1); }
    ;
```

Syntaxanalyse X

Knoten erzeugen

```
expression_t *create_ast_node(enum arith_op t,
                              expression_t *lop,
                              expression_t *rop) {
  expression_t *e = (expression_t*)malloc(sizeof(expression_t));

  e->op = t;
  e->type = UNDETERMINED;
  e->operands.left = lop;
  e->operands.right = rop;
  return(e);
}

expression_t *create_integer(int number) {
  expression_t *e = (expression_t*)malloc(sizeof(expression_t));

  e->op = NUMBER; e->type = INTEGER;
  e->value.ival = number;
  return(e);
}
```

Codegenerierung I

Varianten

- Stapelbasiert: Werte auf Stack abgelegt, Register zur unmittelbaren Verknüpfung verwendet
  - Struktur vieler virtueller Maschinen
  - Direkte Umsetzung: Umgekehrte polnische Notation
  - Einfacher Code, viele Speicherzugriffe
- Registerbasiert: Werte soweit möglich in Registern halten
  - Typischerweise leistungsfähiger
  - Komplexere Codegenerierung
  - Geeignet für ARM (viele Register)

Codegenerierung II: Registerbasiert

Zutaten

- Zielregister $\langle R_i \rangle$ für Resultat
- Hilfsregister $H \equiv \{\langle R_{i+1} \rangle, \ldots, \langle R_N \rangle\}$ für Berechnung

Rekursiver Algorithmus

- Konstanten in Blättern: Konstante in Zielregister $\langle R_i \rangle$ laden
- Rechenoperation:
  - Linken Teilbaum mit Hilfsregistern H berechnen, Resultat in $\langle R_i \rangle$
  - Rechten Teilbaum mit Hilfsregistern $H' = \{H - \langle R_i \rangle\}$ berechnen, Resultat in $\langle R_{i+1} \rangle$
  - $\langle R_i \rangle$ und $\langle R_{i+1} \rangle$ verknüpfen (`add`, `mul` etc.), Resultat in $\langle R_i \rangle$ speichern

Codegenerierung III: Pseudocode

```
1: procedure CodeGeneration(Expression, ⟨R_i⟩, H)
2:     if Expression.type == NUMBER then
3:         Konstante in Register ⟨R_i⟩ laden
4:     else
5:         CodeGeneration(Expression.lhs, ⟨R_i⟩, H)
6:         CodeGeneration(Expression.rhs, ⟨R_{i+1}⟩, H \ ⟨R_i⟩)

7:         EmitExpression(Expression.op, ⟨R_i⟩, ⟨R_{i+1}⟩)
8:     end if
9: end procedure
```

Codegenerierung IV: Umsetzung I

```
void emit_expression(enum arith_op op, unsigned reg1,
                     unsigned reg2) {
  switch(op) {
    case PLUS:
      printf("add r%u, r%u, r%u\n", reg1, reg1, reg2);
      break;
    case MINUS:
      printf("sub r%u, r%u, r%u\n", reg1, reg1, reg2);
      break;
    ...
  }
}
```

Codegenerierung VI: Beispiele I

$1 + 2 + 3$

```
mov r0, #1
mov r1, #2
add r0, r0, r1

mov r1, #3
add r0, r0, r1
```

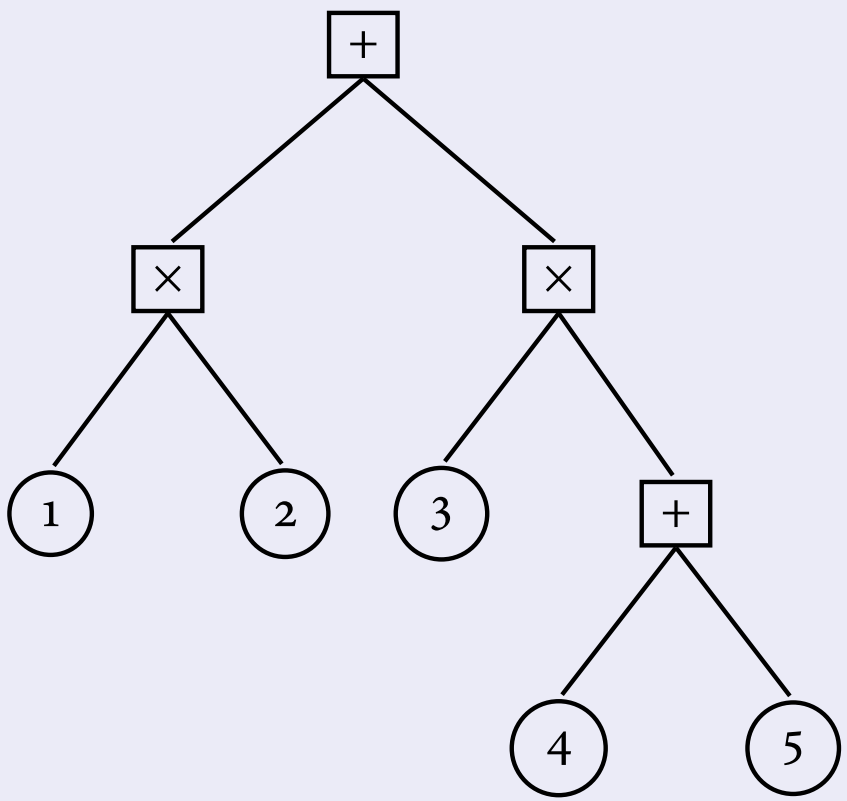
$1 + 2 \times 3$

```
mov r0, #1

mov r1, #2
mov r2, #3
mul r1, r1, r2
add r0, r0, r1
```

Codegenerierung VII: Beispiele II

$1 * 2 + 3 * (4 + 5)$



Code

```
mov r0, #1
mov r1, #2
mul r0, r0, r1

mov r1, #3

mov r2, #4
mov r3, #5
add r2, r2, r3

mul r1, r1, r2
add r0, r0, r1
```

Codegenerierung VII: Register-Spilling

Endliche Ressourcen

Mehr Rechenoperationen als Register

Stapelspeicher als Zwischenablage

- Register verbraucht: *Register Spilling*
  - Register auf Stapel legen
  - Verbleibenden Teilbaum mit »frischer« Hilfsregister-Liste berechnen
- Rückwärtsoperation: *Register Filling*
  - Register mit Stapelinhalt befüllen
  - Letztes Zwischenergebnis bewahren!

Codegenerierung VIII: Spilling-Algorithmus

```
 1: procedure CodeGeneration(Expression, ⟨R_i⟩, H)
 2:     spilled ← false
 3:     if Expression.type = NUMBER then
 4:         Konstante in Register ⟨R_i⟩ laden
 5:     else
 6:         if i = r_max then SpillRegisters(⟨R_0⟩, ⟨R_1⟩, ..., ⟨R_{r_max−1}⟩)
 7:             spilled ← true
 8:             i ← 0
 9:         end if

10:         CodeGeneration(Expression.lhs, ⟨R_i⟩, H)
11:         CodeGeneration(Expression.rhs, ⟨R_{i+1}⟩, H \ ⟨R_i⟩)

12:         EmitExpression(Expression.op, ⟨R_i⟩, ⟨R_{i+1}⟩)

13:         if spilled = true then
14:             ⟨R_{r_max}⟩ ← ⟨R_0⟩
15:             FillRegisters(⟨R_0⟩, ⟨R_1⟩, ..., ⟨R_{r_max−1}⟩)
16:         end if
17:     end if
18: end procedure
```

Codegenerierung IX: Umsetzung Spilling

Implementierung

- Spilling: `push {r0, r1, ..., r`$\langle r_{max-1} \rangle$`}`
- Filling: Äquivalente `pop`-Anweisung

Suboptimale Algorithmen

- Industriell taugliche Compiler: *Deutlich* komplexere Mechanismen
- Siehe Vorlesung *Compilerbau*

Codegenerierung IX: Umsetzung Spilling

Implementierung

- Spilling: `push {r0, r1, ..., r`$\langle r_{max-1} \rangle$`}`
- Filling: Äquivalente `pop`-Anweisung

Suboptimale Algorithmen

- Industriell taugliche Compiler: *Deutlich* komplexere Mechanismen
- Siehe Vorlesung *Compilerbau*

Optimierungen I

Optimierungen (eigentlich: Verbesserungen)

- Priorität Programmierer: *Les- und wartbarer* Code
- Priorität Compiler: *Schneller* und *korrekter* Code
- Optimierung: Leistungs- oder Speicherbedarfs-Verbesserung
- Probleme: Debugging
  - Generierter Code und Ausgangsprogramm möglicherweise drastisch unterschiedlich
  - Variablen »wegoptimiert«, Kontrollfluss unterschiedlich
  - GCC und Clang mittlerweile sehr gut im Verbergen dieser Effekte
- Gute Algorithmen (meistens) wichtiger als aggressive Optimierungen!
- Albtraum: Compilerfehler

Optimierung II

Optimierungsmöglichkeiten: Performance

- Registerallokation
  - Am häufigsten benutzte Werte in Registern vorhalten
  - Endliche Registermenge ☞ Priorisierung notwendig
- Umordnung von Ausdrücken: Abhängigkeiten vermeiden
- Funktionsinlining: Code kopieren statt aufrufen
- Schleifen ausrollen (*Loop unrolling*)
- Prozessor- und Architekturfeatures ausnutzen
  - Kombinierte Multiplikation/Addition
  - Datenstrukturen Cache-gerecht anordnen
  - Instruktionsabhängigkeiten vermeiden

Optimierungen III

Optimierungsmöglichkeiten: Speicherplatz

- Unbenutzten (toten) Code eliminieren
- Gemeinsame Teilausdrücke finden
- Tail Call-Optimierung
  - Rekursiver Aufruf von `f()` am Ende von `f()`
  - Prolog/Epilog-Overhead durch Sprung vermeiden
- Komprimierte Befehlsformate (Thumb)

Doppelter Effekt

Speicheroptimierungen häufig auch Geschwindigkeitsoptimierungen!

Optimierungen IV

C-Code: Tail Call-Optimierung

```
int f(int n) {
  if (n == 0)
    return(1);
  else
    return(n*f(n-1));
}
```

Unoptimiert

```
f:    stmfd sp!, {r4, lr}
      subs  r4, r0, #0
      beq   .L3
      sub   r0, r4, #1
      bl    f
      mul   r0, r4, r0
      ldmfd sp!, {r4, pc}
.L3:  mov   r0, #1
      ldmfd sp!, {r4, pc}
```

Optimiert

```
f:    subs r3, r0, #0
      mov  r0, #1
      beq  .L4
.L3:  mul  r0, r3, r0
      subs r3, r3, #1
      bne  .L3
      bx   lr
.L4:  bx   lr
```

Inhalt

Speicherhierarchie

Grenzen der Beschleunigung

- Taktfrequenz und CPU-Architekturen: Kontinuierliche Steigerung
- Höherer Speicherbedarf ☞ höhere Buslast
- Busse und RAM bremsen CPUs aus!
  - Häufig gebrauchte Daten lokal vorrätig halten (Caches)
  - Speicherhierarchie einführen
- Register ☞ Caches ☞ RAM ☞ Flash-Speicher ☞ Festplatten ☞ Magnetbänder

Genauere Untersuchung

- RAM & nicht-flüchtiger Speicher
- Festplatten & SSDs
- Caches & ihre Nutzung

RAM-Speicher

Grundlegendes

- RAM = *Random Acccess Memory*: wahlfreier Zugriff
- Programmierersicht: Lineares Array schreib- und lesbarer Daten
- Kleinste Einheit: Bit; RAM-Chips werden zu Hauptspeicher verbunden

RAM-Speicher

Statisches RAM

- Schaltung mit vier/sechs Transistoren pro Bit
- Stromversorgung ☞ Wert wird behalten
- Relativ unempfindlich gegenüber EMI, kosmischer Strahlung,…
- Relative Zugriffszeit: 1, Relative Kosten: 100
- Anwendung: Caches

Dynamisches RAM

- Kondensator speichert Wert; ein Transistor für Zugriff notwendig
- Werte müssen zyklisch (10..100 ms) *aufgefrischt* werden (refresh)
- Langsamer, empfindlicher und billiger als SRAM
- Relative Zugriffszeit: 10, Relative Kosten: 1
- Anwendung: Hauptspeicher, Frame Buffer, GPUs

Nichtflüchtiger Speicher

- Speicherinhalt auch ohne Strom persistent
- Firmware (ROM, Initalisierungscode, Identifizierung)
- Solid State Disks: Tablets, Smartphones, USB-Sticks, SD-Cards,…

Typen nichtflüchtigen Speichers

- ROM (*Read only-Memory*): Beschreiben während Produktion
- PROM (*Programmable ROM*): Einmaliges Beschreiben
- EPROM (*Eraseable PROM*): Vollständiges Löschen (UV, Röntgen) möglich
- EEPROM (*Electrically EPROM*): (Byte-granulares) Löschen mit elektronischen Mitteln
- Flash-Speicher: Sektorweise löschbares EEPROM (Technologien: NAND/NOR); begrenzte Anzahl Löschzyzklen ($\approx 100\,000$)

Verbindung CPU/Speicher

Busse und Datentransport

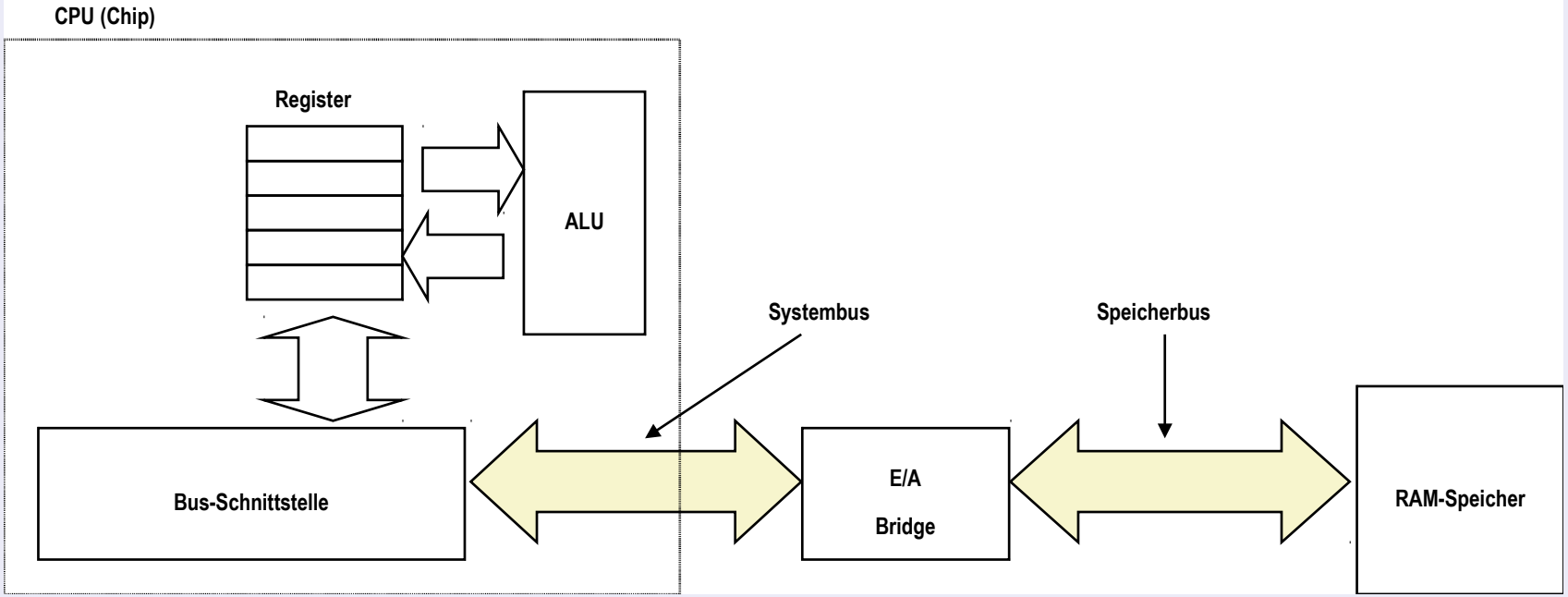


Abbildung basiert auf Bryant & O'Hallaron

Verbindung CPU/Speicher

Busse und Datentransport

- Traditionell: Parallele Drähte zur Übertragung von Adressen und Daten
- Trend zu *seriellen* Bussen
  - Weniger Drähte
  - Kein Signalversatz
  - Kein Nebensprechen (Cross-Talk)
- Mehrere Geräte kommunizieren über Bus
- Speicherzugriff über *Systembus*

Speicheroperationen

Lesetransaktion

- `ldr r1, [#A]`
- CPU schreibt Adresse A auf Speicherbus
- Speicher liest Wert x an Adresse A aus und legt x auf Speicherbus
- CPU liest Wert x auf Speicherbus und legt ihn in einem Register ab

Schreibtransaktion

- `str r1, [#A]`
- CPU schreibt Adresse A auf Speicherbus
- Hauptspeicher liest Adresse und wartet auf Datenwort
- CPU schreibt Datenwort aus Register auf Speicherbus
- Hauptspeicher liest Datenwort und schreibt es an Adresse A

Festplatten I

- Nichtflüchtig, hohe Informationsdichte
- Preisgünstig und langsam (Mechanik!)
- Rotierender Rost…

Komponenten

- Magnetplatten
- Drehachse
- Schreib/Lesekopf
- Aktuator zur Kopfpositionierung
- Steuerungselektronik (inkl. Prozessor und Speicher)
- Cache-Speicher (transparent)

Bildquelle: chip.de

Festplatten II

- Magnetplatte: zwei Oberflächen
- Oberfläche: In konzentrische Ringe (*Tracks*) aufgeteilt
- Tracks: Durch Freiräume (*Gaps*) getrennte *Sektoren* (Achtung: Nomenklatur Block/Sektor oft nicht eindeutig!)
- Vertikal benachbarte Tracks in *Zylinder* zusammengefasst

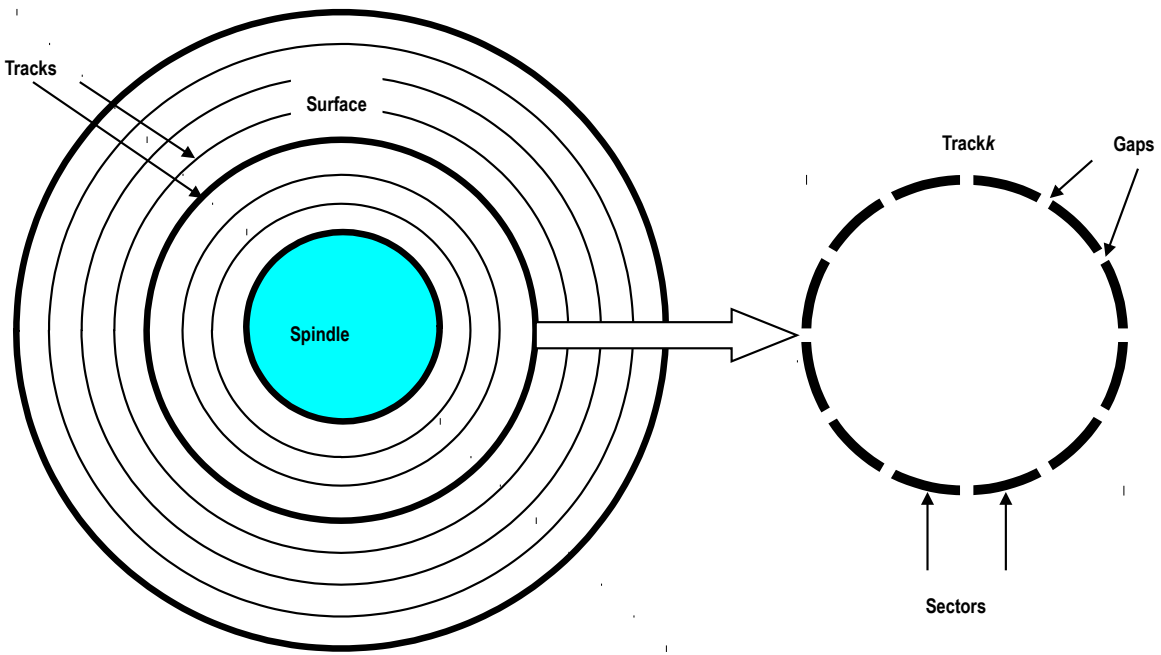
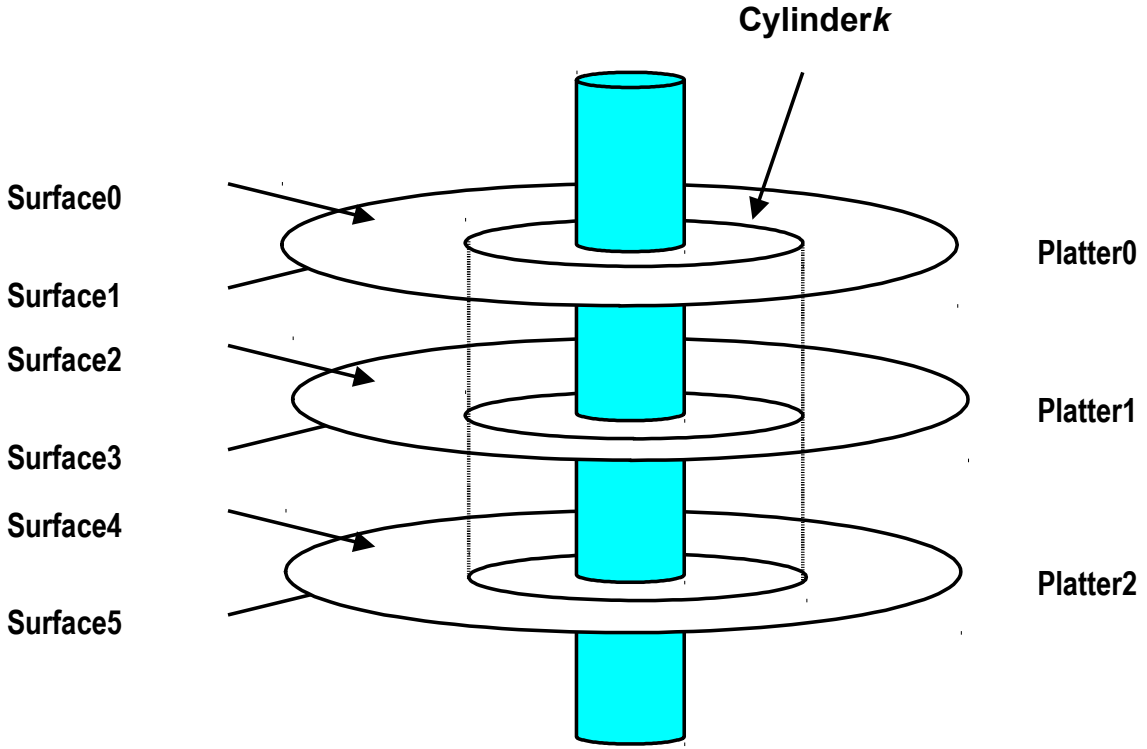


Abbildung basiert auf Bryant & O'Hallaron

Festplatten II

- Magnetplatte: zwei Oberflächen
- Oberfläche: In konzentrische Ringe (*Tracks*) aufgeteilt
- Tracks: Durch Freiräume (*Gaps*) getrennte *Sektoren* (Achtung: Nomenklatur Block/Sektor oft nicht eindeutig!)
- Vertikal benachbarte Tracks in *Zylinder* zusammengefasst



Festplatten III

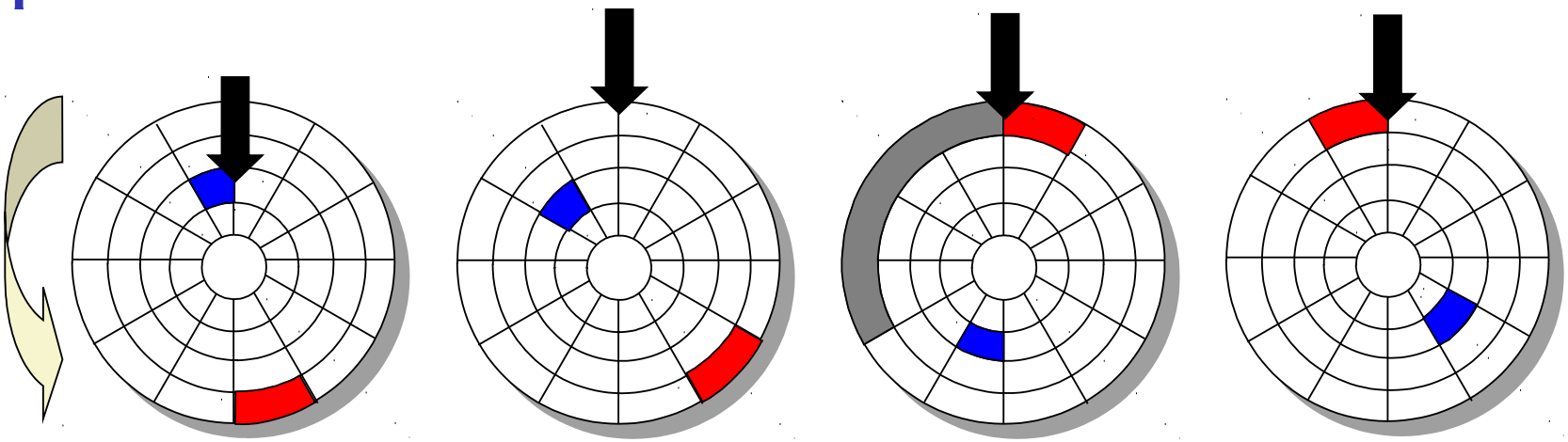
Kapazität

- Hersteller/BWLer: 1 GB = 10^9 Bytes
- Technikverständige: 1 GiB = 2^{30} Bytes = 1 073 741 824 Bytes

Dichten

- Aufnahmedichte: Bits/Inch (Bits auf einer 1 Inch langen Spur des Tracks)
- Track-Dichte: Tracks/Inch (Anzahl Tracks in einem 1 Inch langen Redialsegment)
- Speicherdichte: Bits/Inch^2 (Produkt obiger Dichten)
- Festplatte: 300 GiBit/Inch^2 (DVD: 300 GiBit/Inch^2, BlueRay: 30 GiBit/Inch^2)
- Nicht-SI-Einheit »Inch« historisch bedingt

Festplatten IV



Nach blauer Leseoperation | Seek nach rot | Rotationslatenz | Nach roter Leseoperation

Latenzen und Zugriffszeit

- Verzögerungen: *Rotations*latenz und *Seek-Zeit*
- Bits im gewünschten Sektor lesen (vernachlässigbar gegenüber obigen Größen)
- Zeit für Kommandoverarbeitung
- Zugriffszeit Festplatte/SRAM = 40000, Festplatte/DRAM = 2500

Festplatten V: Rechnung

Plattencharakteristik

- Rotationsgeschwindigkeit: 7200 $\frac{1}{\text{min}}$
- Durchschnittliche Seek-Zeit: 9 ms
- Durchschnittliche # Sektoren/Track: 400

Lesen eines Sektors

Sektor gelesen in $\bar{t}_{\text{acc}} = \bar{t}_{\text{seek}} + \bar{t}_{\text{rot}} + \bar{t}_{\text{trans}}$

- Rotationslatenz: $\bar{t}_{\text{rot}} = \frac{1}{2} \times 60\frac{\text{s}}{\text{min}} / 7200\frac{1}{\text{min}} = 4$ ms
- Transferzeit: $\bar{t}_{\text{trans}} = 60\ \frac{\text{S}}{\text{min}} / 7200\ \frac{1}{\text{min}} \times 1/400 = 0.02$ ms
- $\bar{t}_{\text{acc}} = 9$ ms $+ 4$ ms $+ 0.02$ ms

Moral

- Zugriffszeit dominiert von Seek-Zeit und Rotationslatenz
- Erstes Bit teuer, restliche Bits gratis

Festplatten V: Optimierungen I

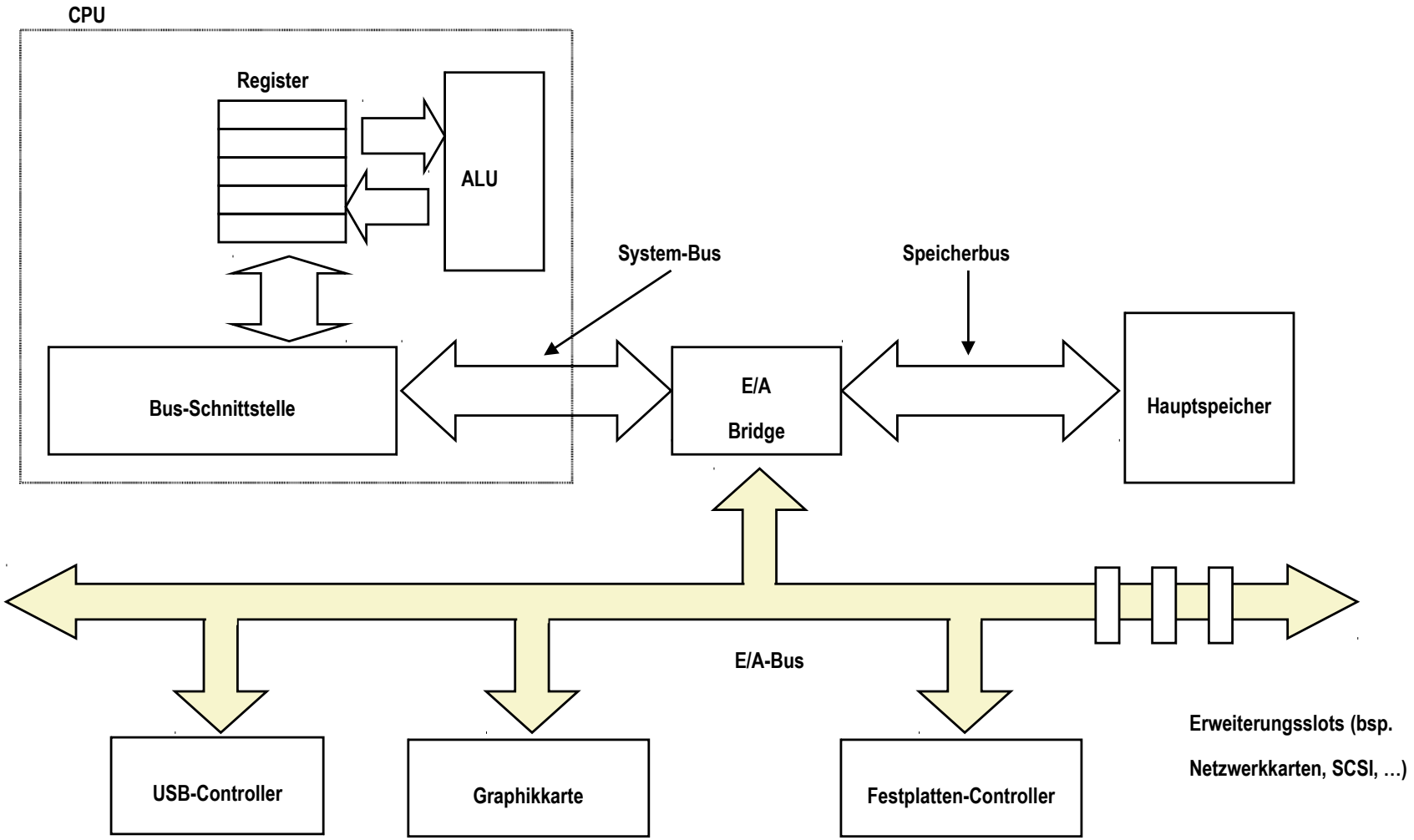
Logische Datensicht

- Aufteilung in Zylinder etc. eigentlich nicht von Interesse
- CHS nicht anwendbar auf Magnetbänder, SSDs, etc.
- Moderner: LBA (*logical block addressing*)
- Lineare Sequenz von Blocks (bsp. 4 KiB)
- Um/Übersetzung Adressen: DSP in Festplatte
- Defekte ☞ Umblenden auf Reservespeicher (S.M.A.R.T.)

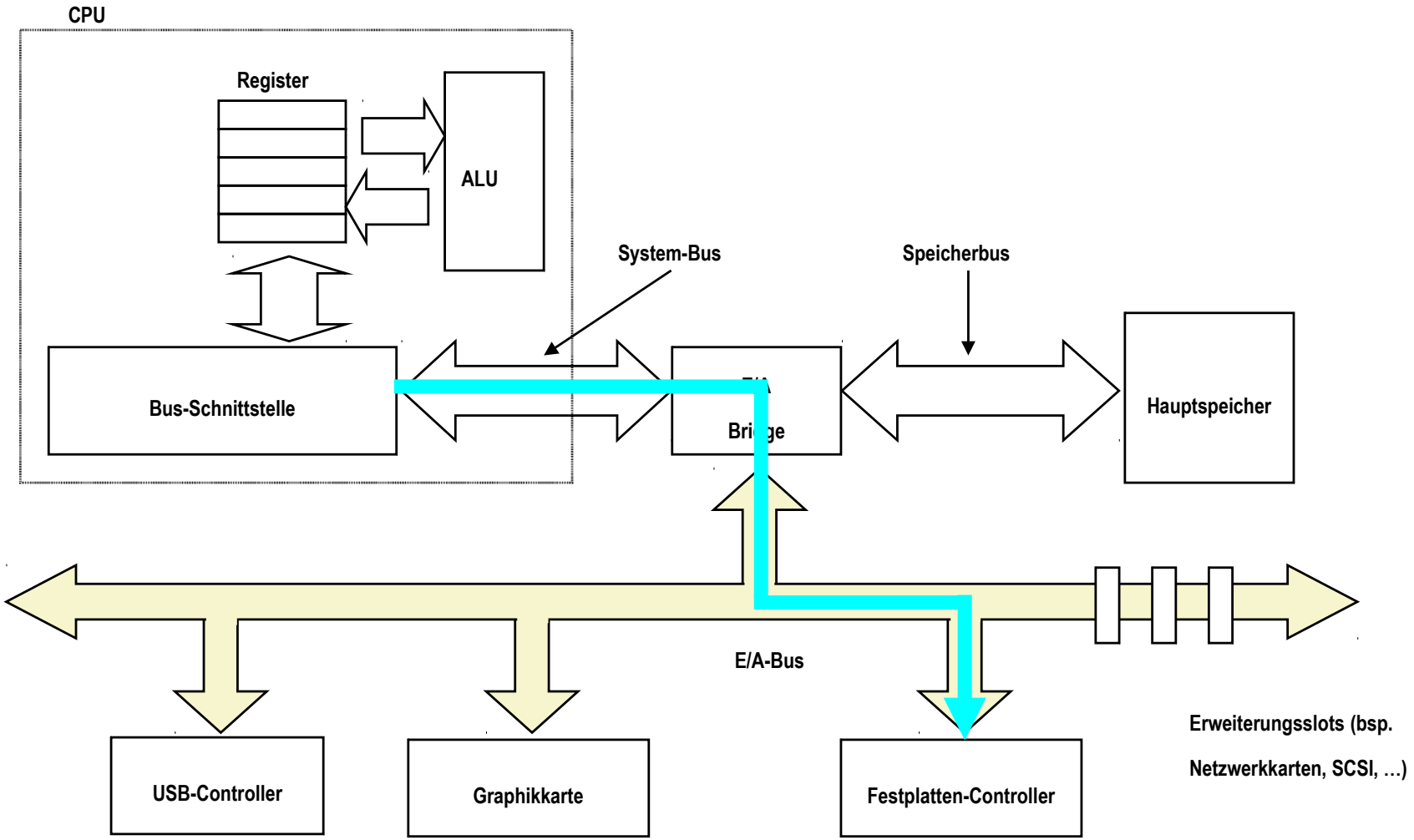
DMA

- DMA: *Direct Memory Access*
- CPU: andere Aufgaben während Lesevorgang ☞ direkter Transfer Platte ⇔ RAM
- Daten direkt zwischen Platte und Speicher transportiert
- Signalisierung Übertragungsende durch Unterbrechung (Interrupt)

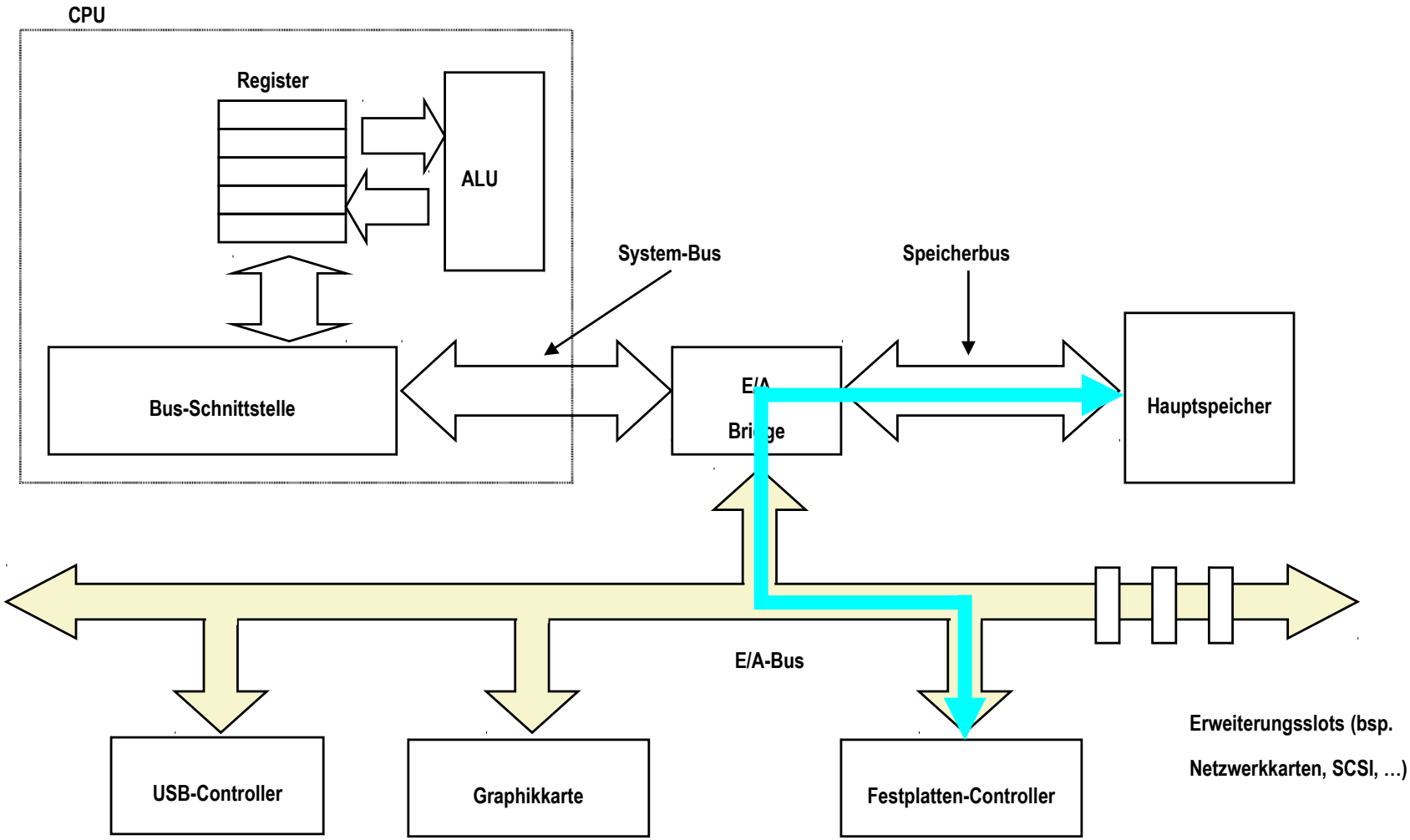
Festplatten VI: Optimierungen II



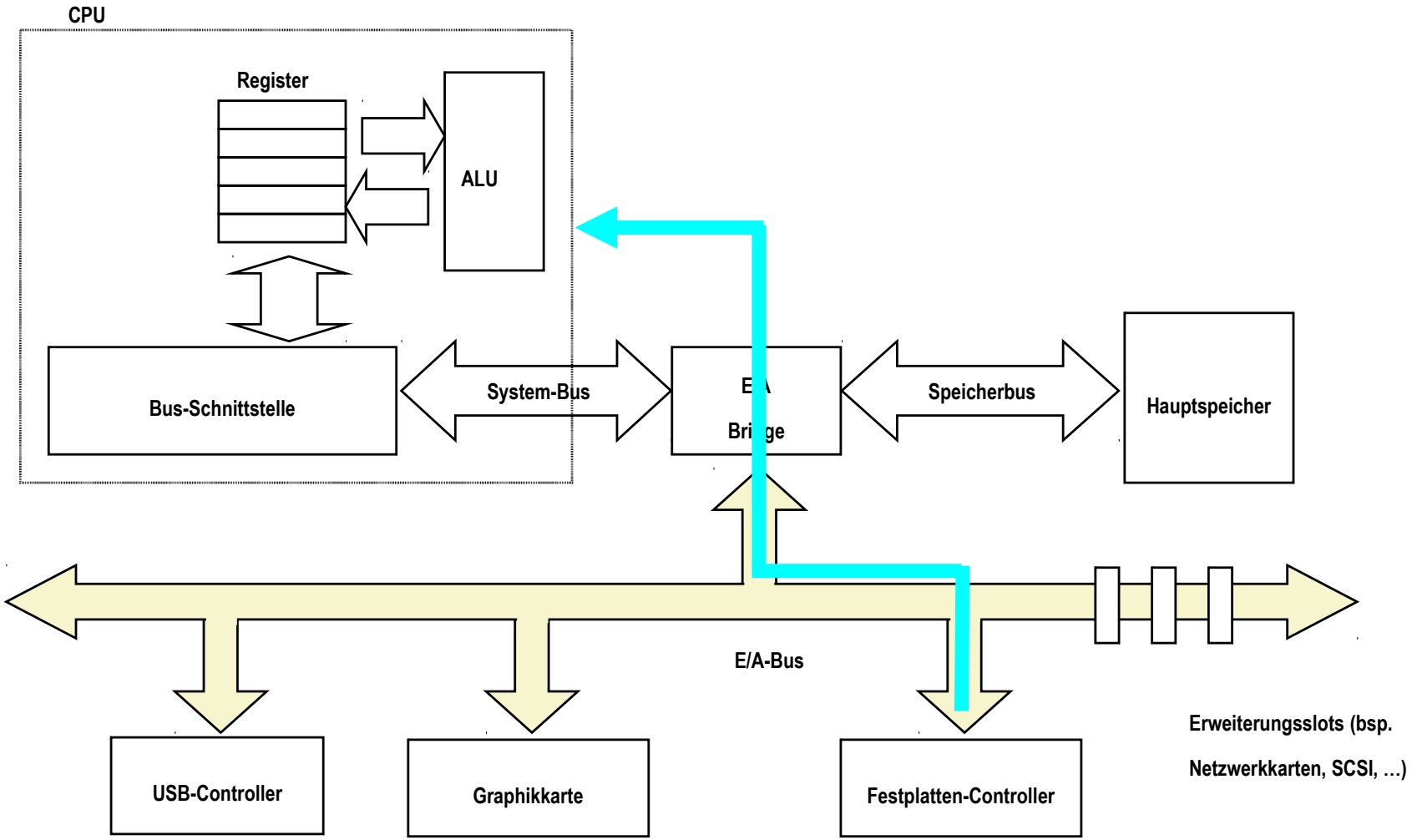
Festplatten VI: Optimierungen II



Festplatten VI: Optimierungen II



Festplatten VI: Optimierungen II



Festplatten VII: Optimierungen III

Weitere Optimierungen

- HW: Mehrere unabhängige Köpfe
- HW: Schnelle Busübertragung
- BS: Minimierung Anzahl Seek-Operationen
- BS: Lesen/Schreiben aufeinanderfolgender Daten
- ☞ Scheduling erforderlich!

Solid State Disks I

- Programmierschnittstelle wie bei Festplatten
- Völlig unterschiedliche physikalische Implementierung

Flash: Funktionsmechanismus

- Seiten: 512 Bytes bis 4 KiB; Blocks: 32...128 Seiten
- Daten können in Seiten-Granularität gelesen/geschrieben werden
- Schreiben nur möglich, wenn zugehöriger Block gelöscht wurde
- Blöcke gehen nach ≈ 100.000 Zugriffen kaputt (wearing)

Solid State Disks II

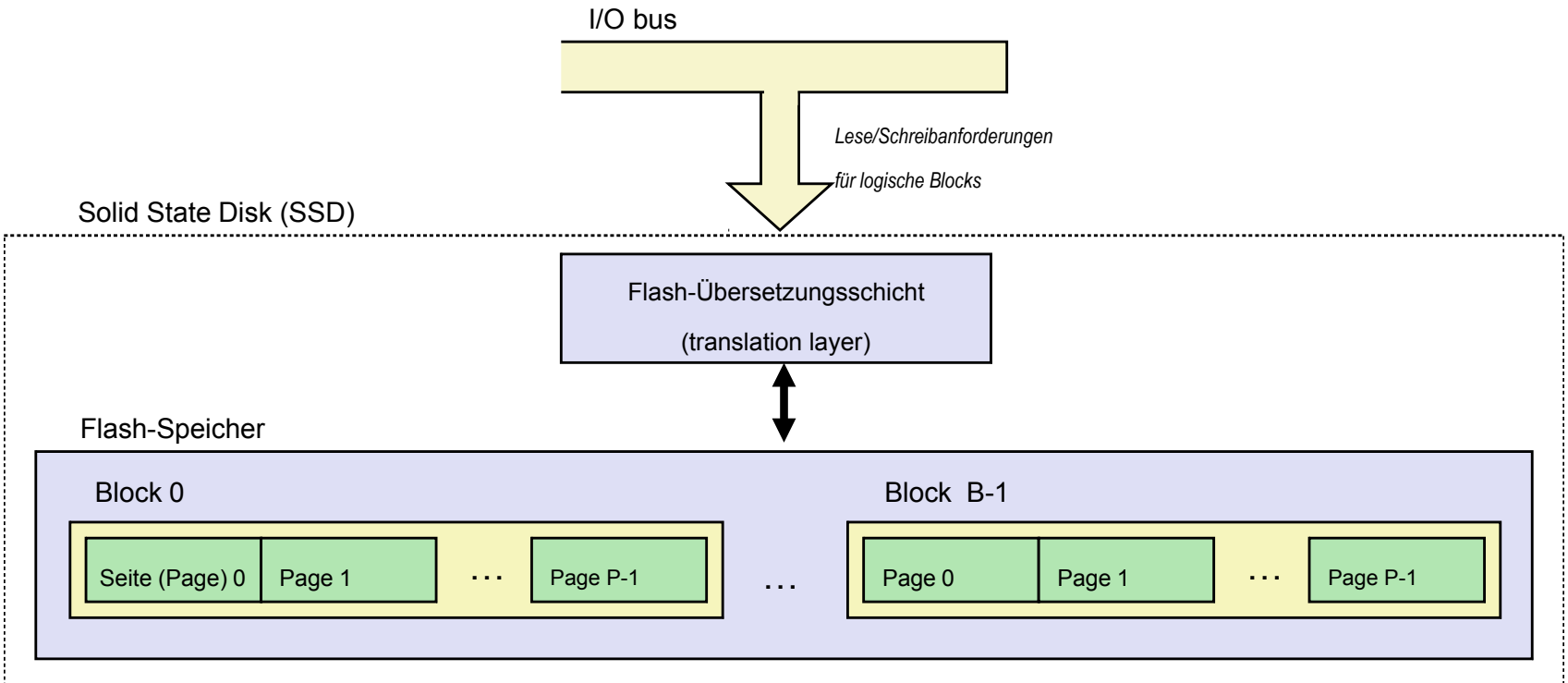


Abbildung basiert auf Bryant & O'Hallaron

CPU-Speicher-Kluft

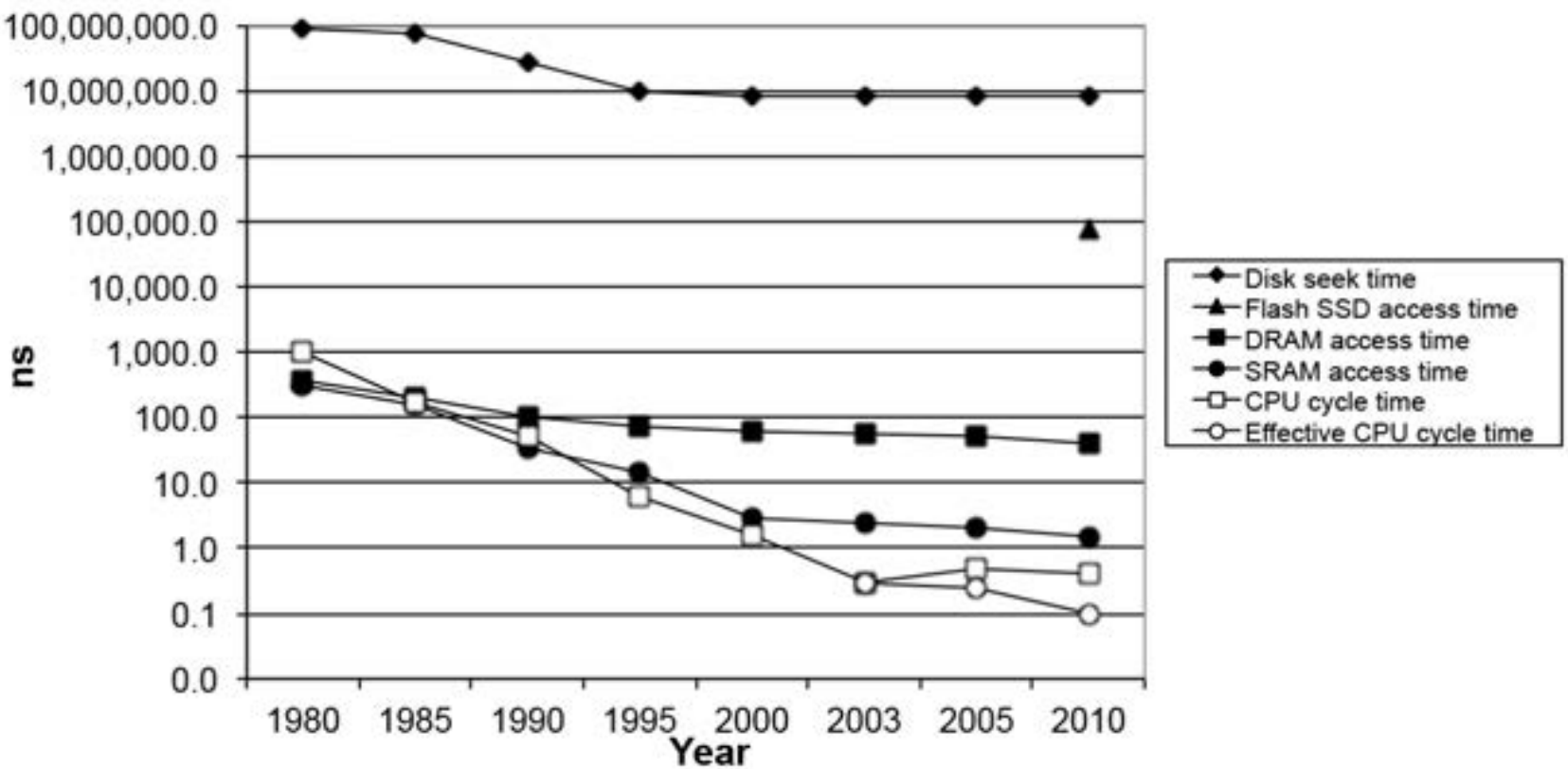


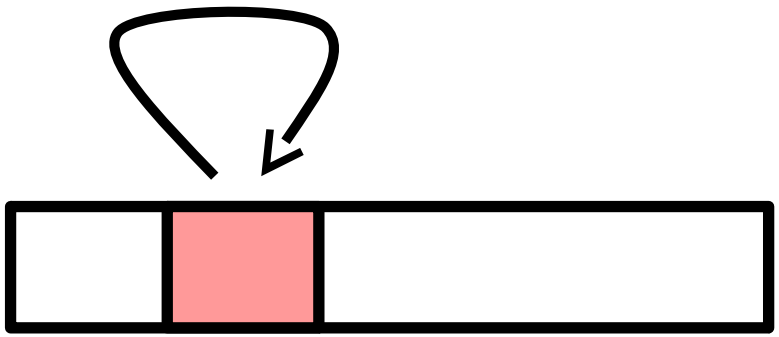
Abbildung basiert auf Bryant & O'Hallaron

Lokalität I

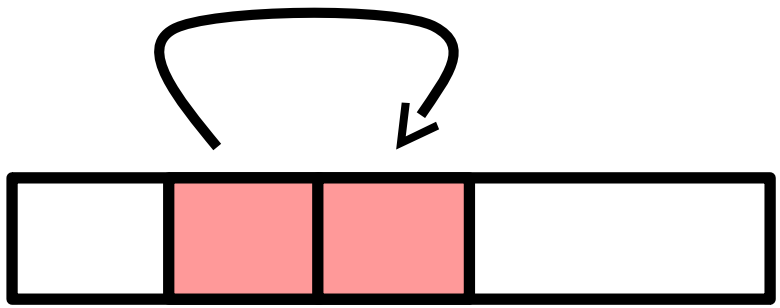
Lokalität (*locality of reference*)

Programme verwenden häufig Daten und Instruktionen, deren Adressen nahe an vorher benutzten liegen.

Zeitlich



Räumlich



Lokalität II

Beispiel

```
int do_sum(int *arr,
           unsigned int N) {
  int sum = 0;

  for (int i=0; i < N; i++) {
    sum += arr[i];
  }

  return sum;
}
```

Lokalität

- Datenreferenzen
  - Jedes Array-Element wird nacheinander verwendet (*räumlich*)
  - `sum` in jedem Durchlauf verwendet (*zeitlich*)
- Instruktionsreferenzen
  - Lineare Sequenz (keine Sprünge) (*räumlich*)
  - Schleife wiederholt durchlaufen (*zeitlich*)

Lokalität III

Welches Beispiel weist höhere Lokalität auf?

Variante A

```
int sum_all(int *a,
            unsigned M,
            unsigned N) {
  int i, j, sum = 0;
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
      sum += a[i][j];
    }
  }

  return sum;
}
```

Variante B

```
int sum_all(int *a,
            unsigned M,
            unsigned N) {
  int i, j, sum = 0;
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
      sum += a[j][i];
    }
  }

  return sum;
}
```

Erinnerung: Row-major storage, d.h. `a[i][j] = *a + i*M + j`

Speicherhierarchie I

Speicherhierarchie

- Fakt 1: Fast alle Programme weisen starke Lokalität auf
- Fakt 2: Je schneller der Speicher, desto teurer
- Fakt 3: Geschwindigkeit von CPU und Hauptspeicher divergiert
- Folgerung: Einführen von Speicherhierarchien

Speicherhierarchie II

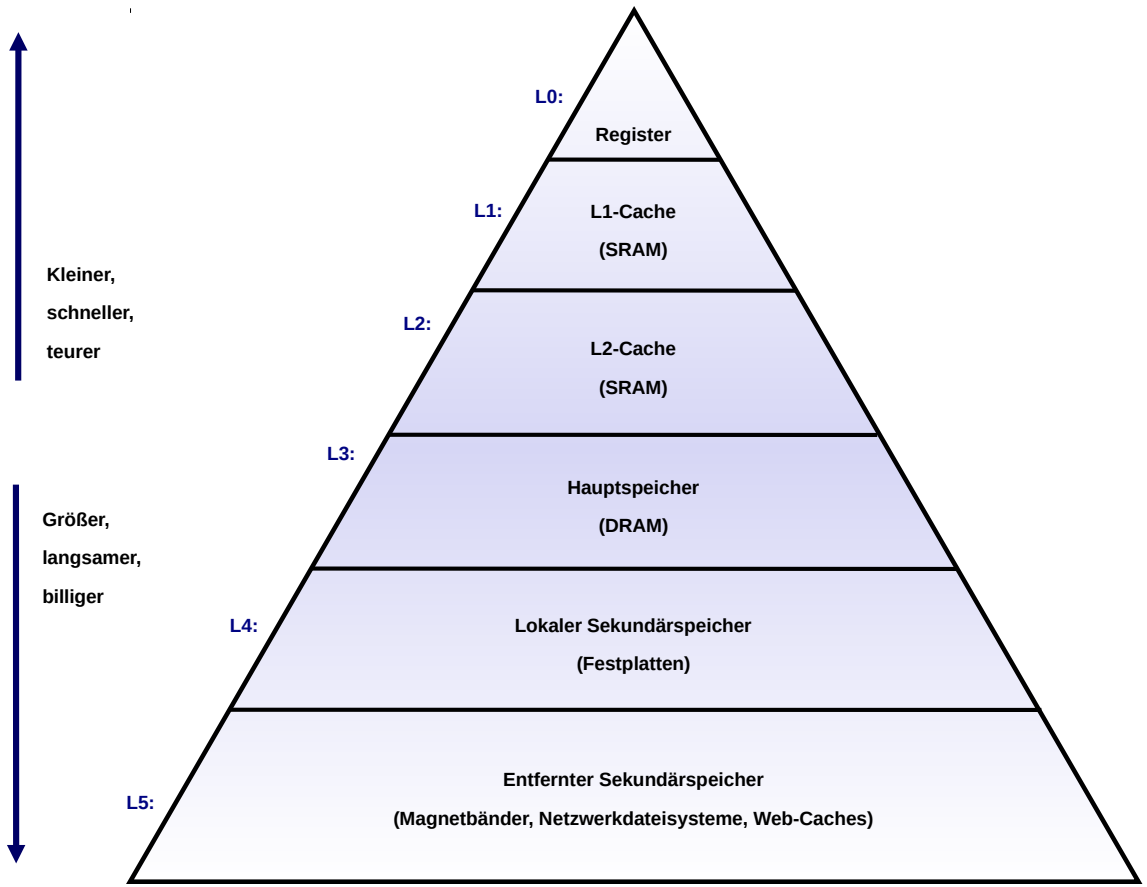


Abbildung basiert auf Bryant & O'Hallaron

Caches: Grundlagen I

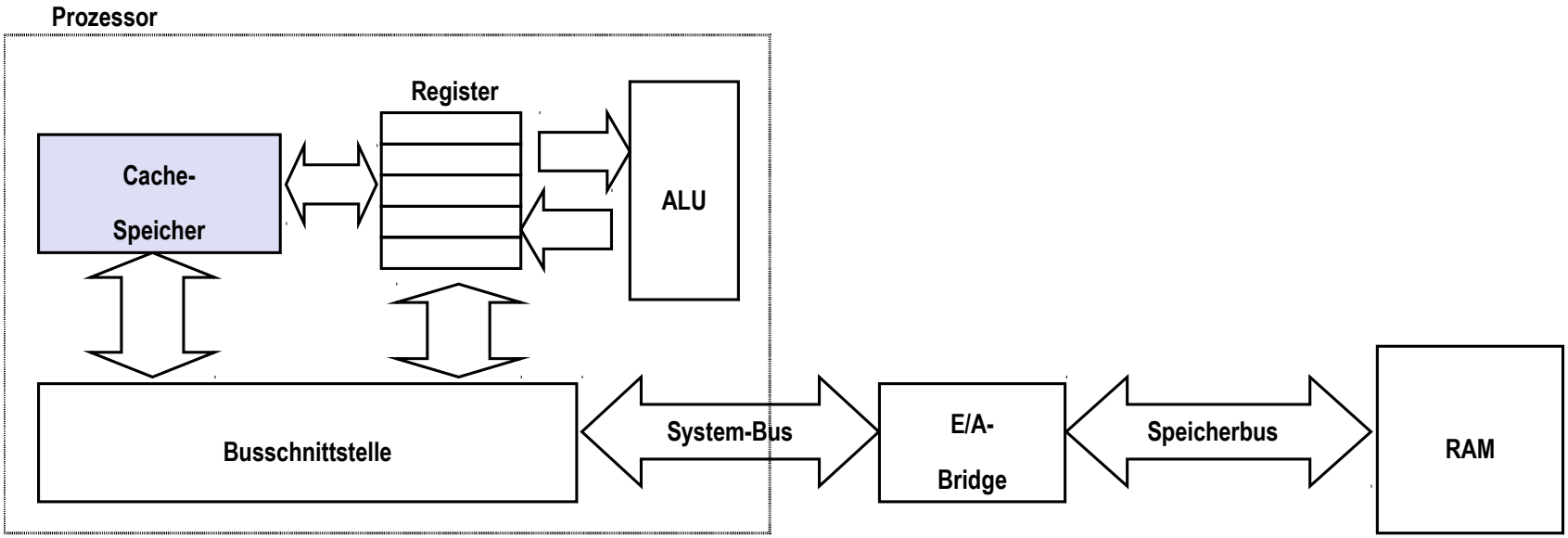


Abbildung basiert auf Bryant & O'Hallaron

Caches: Grundlagen II

Grundeigenschaften

- Caches: *Geheimspeicher*, schneller bis extrem schneller Speicher
- Halten stark genutzte Daten (und Code) vor
- Häufig transparent für Applikationsprogrammierer
- …bis auf Leistungssteigerungen
- Hierarchisch organisiert: $L_1, L_2, L_3, \ldots$
- ☞ Die kleinere, schnellere Stufe L_k dient als Cache für die langsamere, größere Stufe L_{k+1}
- Wünschenswerte Cache-Größe: 0 oder ∞ (Warum?)

Caches: Grundlagen III

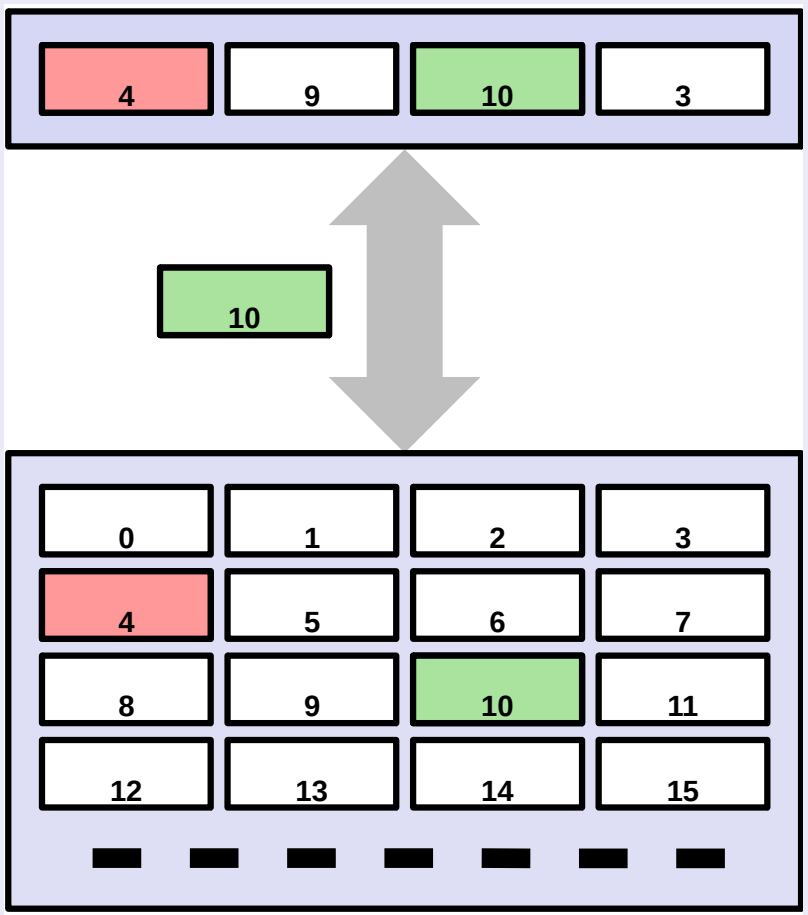
Prinzip

Die Speicherhierarchie soll

- im Mittel so wenig wie der langsamste Speicher auf niedrigster Stufe kosten
- im Mittel Daten so schnell wie der Speicher auf höchster Stufe bereitstellen

Caches: Technische Umsetzung

Grundaufbau



Wirkung

- Daten in Stufe $k + 1$ werden in Blöcke gruppiert
- Blockweises kopieren in Stufe k
- Offenbar können nicht alle Blöcke gleichzeitig im Cache sein ☞ Plazierungs- und Austauschmechanismen (*placement* und *replacement*) notwendig
- Zugriff: Cache Hits und Cache Misses

Abbildung basiert auf Bryant & O'Hallaron

Caches: Hits und Misses

Typen von Cache Misses

- Kalter Miss (*cold miss*): Cache ist leer, Daten wurden noch nie angefragt – unvermeidbar (befüllte Caches: *warm*)
- Cache-Konflikt
  - Unterschiedliche Blöcke aus L_{k+1} können auf gleiche Blöcke aus Stufe L_k abgebildet werden (notwendig, da $|L_k|/|L_{k+1}| \approx 0$)
  - Beispiel: $i \in L_{k+1} \rightarrow i \mod 4 = j \in L_K$
  - Cache kann dabei teilweise unbenutzt sein!
  - Lösung: Geschicktes plazieren der Daten in Stufe $k+1$; Wissen über Cache-Aufbau erforderlich
- Kapazitäts-Miss: Mehr häufig benutzte Daten (*working set*) vorhanden, als in L_k passen

Caches: Kapazitäten und Typen

Cache-Typ	Was	Wo	Latenz	Verwaltung
Register	Speicherwort	CPU-Kern	0 Zyklen	Compiler
TLB	MMU-Ergebnisse	CPU-Kern	0	Hardware
L1-Cache	D & I	CPU-Kern	1	Hardware
L2-Cache	D & I	CPU(-Kern)	10	Hardware
Buffer-Cache	Dateien	RAM	100	BS
Web-Proxy	Statisches HTML	Festplatten	1.000.000.000	Proxy

Lesen aus Caches

Grundproblem: Lesen

- Gewünschte Lese-Adresse im Speicher bekannt
- Adresse(n) im Cache ermitteln
- Testen, ob Eintrag vorhanden/gültig
- Wenn ja, Wert zurückliefern
- Wenn nein: Vorgehen in höherstufigen Caches wiederholen

Cache-Typen

Systematik vor Didaktik

Bedeutung der Bezeichnungen wird im weiteren Verlauf deutlich!

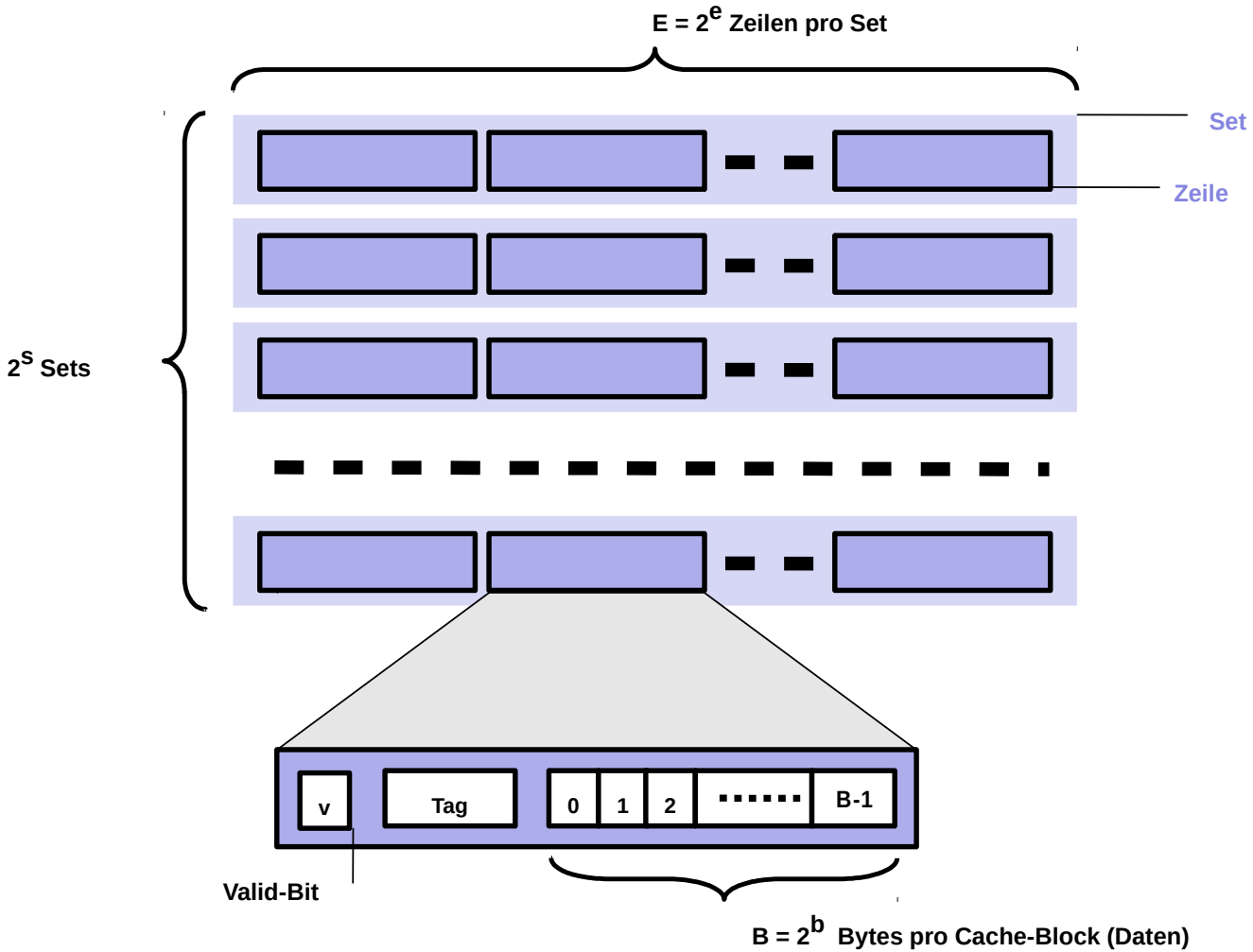
Implementierungen

- Direktes Mapping
- Set-Assoziativ
- Vollassoziativ
- *Alle Typen können auf eine gemeinsame Beschreibung abstrahiert werden!*

Adressierungsarten

- Physikalische Adressierung
- Virtuelle Adressierung
- Virtuelle Adressierung mit physikalischem Tagging
- …weitere Mischtypen

Cache-Aufbau I



Cache-Aufbau II

Vorgehensweise

- Gegeben: Speicheradresse
- Aufteilung in drei Komponenten (t, s, b) als »Indices«
- Schritt 1: Set s identifizieren
- Schritt 2: Prüfen, ob passender Tag t bei (irgendeiner) Zeile im Set
- Schritt 3: Wenn ja, Daten an Offset b lesen

Adressaufspaltung

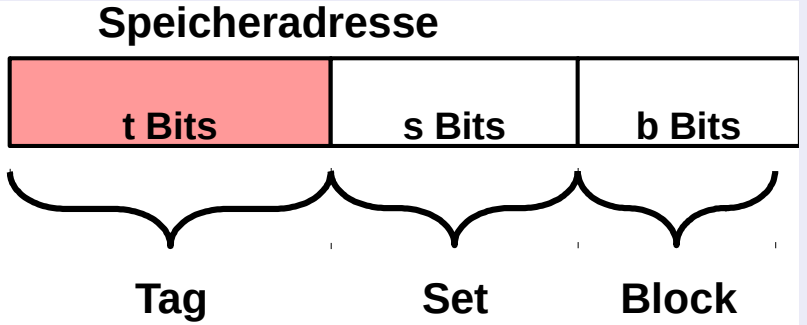
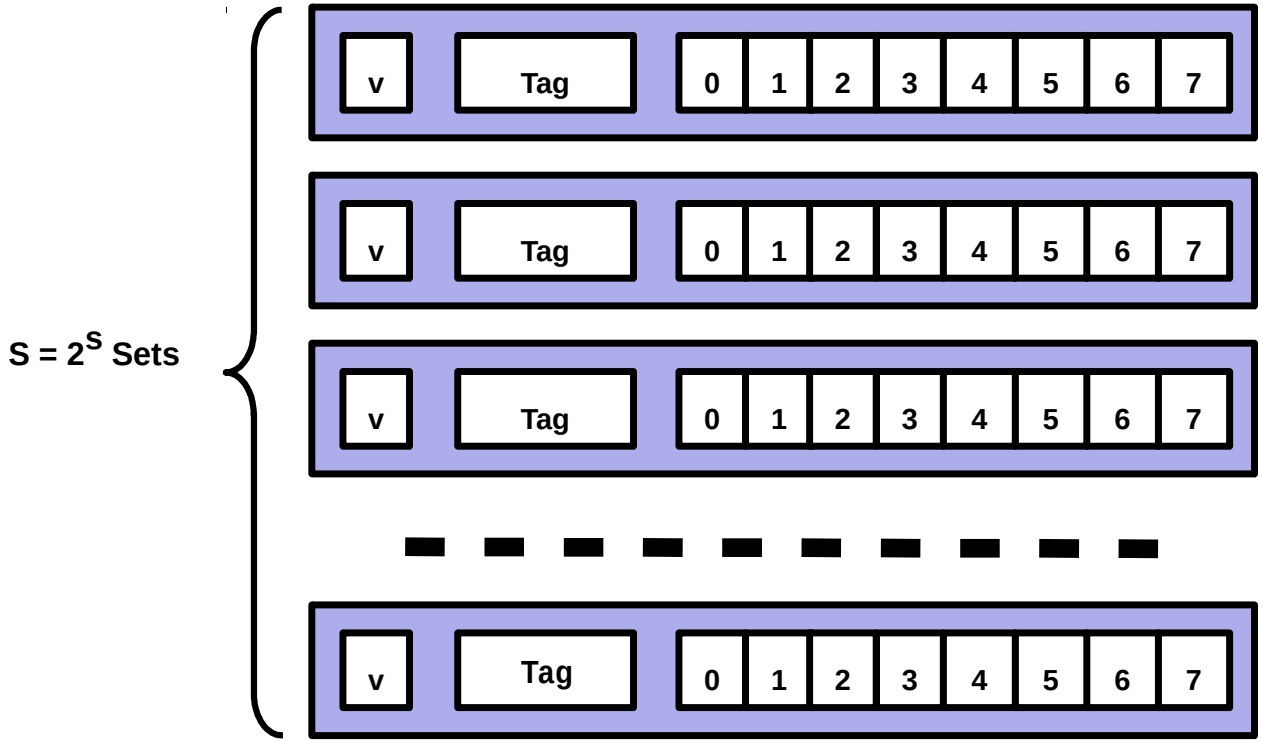


Abbildung basiert auf Bryant & O'Hallaron

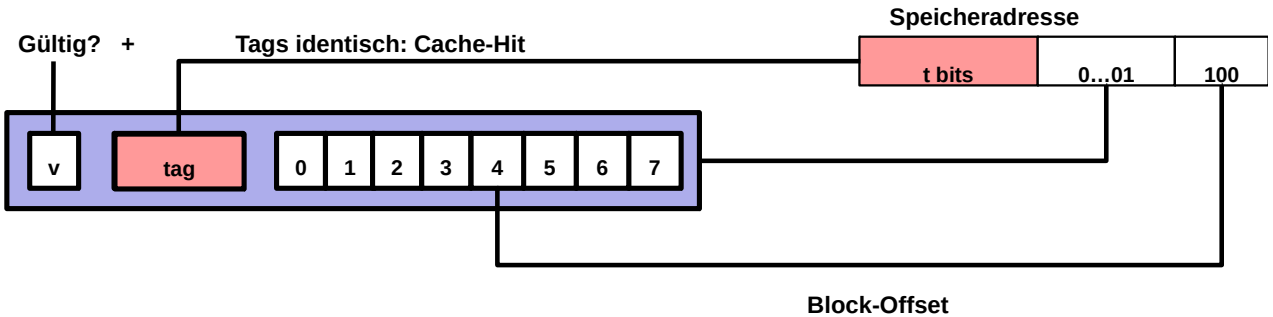
Cache-Aufbau III



Beispiel: Direct-Mapped Cache

- $E = 1$, d.h. eine Zeile pro Set
- Annahme: 2 Bytes pro Block

Cache-Aufbau III



Beispiel: Direct-Mapped Cache

- $E = 1$, d.h. eine Zeile pro Set
- Annahme: 2 Bytes pro Block

Abbildung basiert auf Bryant & O'Hallaron

Cache-Aufbau IV

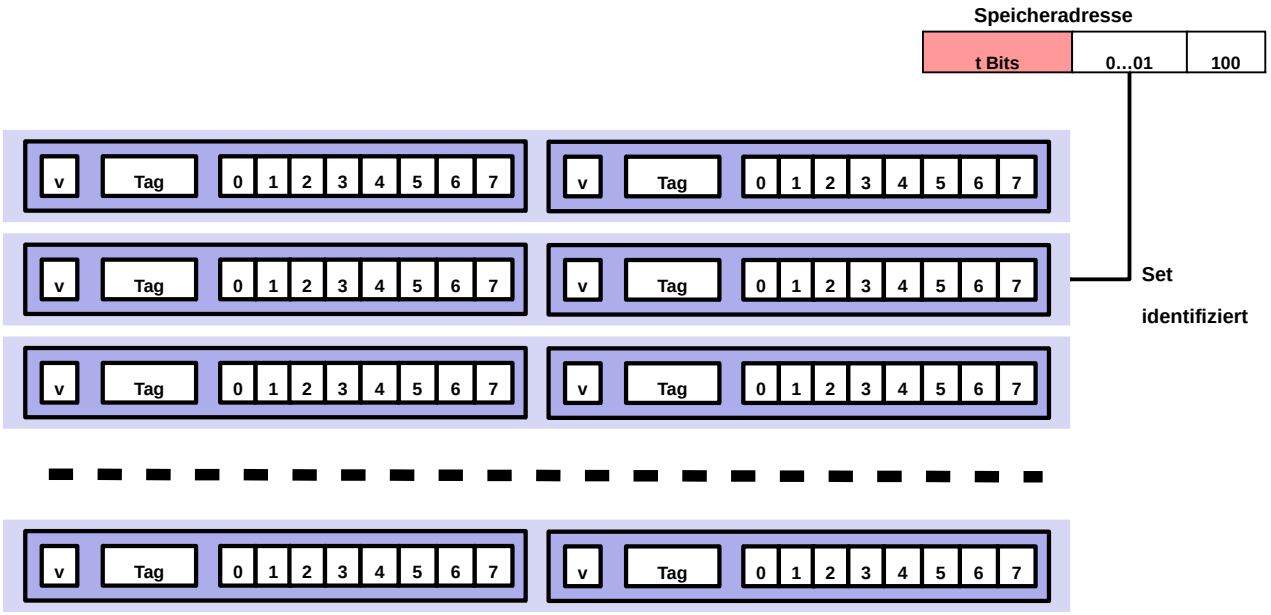
Simulation Direct-Mapped Cache: Siehe Tafel

Cache-Aufbau V

Direct-Mapped: Eigenschaften

- Jeder Hauptspeicherblock wird auf *genau eine* Cache-Zeile abgebildet
- Adresse mit gleichem Indexanteil: Konkurrenz
- HW-Anforderung: t-Bit-Vergleich ☞ Implementierung einfach
- Triviale Verdrängungsstrategie

Set-Assoziativer Cache I



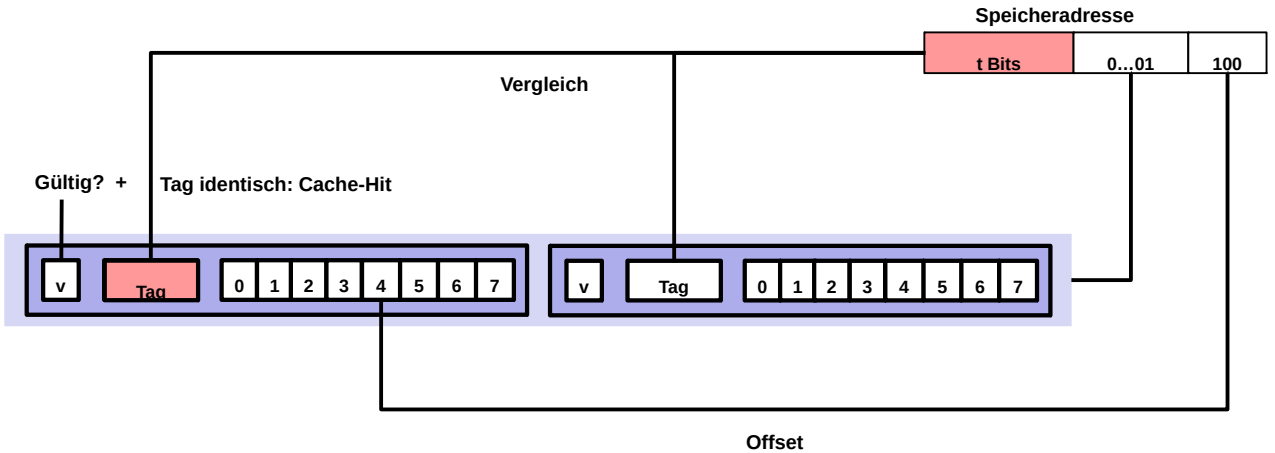
Eigenschaften

- Hier: $E = 2$, (zwei Zeilen/Set)
- Reale Welt: 4- oder 8-fach Set-Assoziative Caches
- Set: Speicheradresse
- Zeile: Auswahlmöglichkeiten!

Probleme

- Mehrere Tag-Vergleiche notwendig (hier: 2)
- Komplexere Hardware
- Verdrängungsstrategie notwendig

Set-Assoziativer Cache I



Eigenschaften

- Hier: $E = 2$, (zwei Zeilen/Set)
- Reale Welt: 4- oder 8-fach Set-Assoziative Caches
- Set: Speicheradresse
- Zeile: Auswahlmöglichkeiten!

Probleme

- Mehrere Tag-Vergleiche notwendig (hier: 2)
- Komplexere Hardware
- Verdrängungsstrategie notwendig

Abbildung basiert auf Bryant & O'Hallaron

Set-Assoziativer Cache II: Verdrängungsstrategien

Zufällig (*Random*)

- Irgendeine Zeile auswählen und ersetzen
- Pseudo-Zufall mangels echtem Zufall

Round Robin

- Zeilen der Reihe nach ersetzen: 0,1,2,3,0,1,...
- Simple Modulo-Operation
- Zuletzt ersetzte Zeile muss gespeichert werden

Least Recently Used (LRU)

- Strategie: Am seltensten benutzte Zeile ersetzen
- Erfordert Zugriffszähler

First In, First Out (FIFO)

- Warteschlangenprinzip

Vollassoziativer Cache I

Vollassoziativer Cache

- $S = 1$, d.h. nur *ein einziges* Set
- Jede Speicheradresse kann an jeder Cache-Position abgelegt werden
- Hoher Vergleichs/Suchaufwand für Tags
- Typischerweise Parallelvergleich
- Aber: Keine Conflict Misses
- Nur für sehr kleine Caches geeignet (TLB, L1)

Caches: Schreiboperationen I

Problem

- Mehrere »Instanzen« eines Datums
- Beispiel: L1, L2, RAM
- ☞ Vorgehensweise bei Schreib-Modifikationen?

Write-Hit

- *Write-through*: Unmittelbar in Speicher »weiterschreiben«
- *Write-back*: Transfer ins RAM erst nach Ersetzen der Zeile. Zusätzliches Dirty-Bit erforderlich

Write-Miss

- *Write-allocate*: Block in Cache laden (vorteilhaft bei weiteren Schreibzugriffen)
- *No-Write-allocate*: Unmittelbar in Speicher schreiben

Leistungsberechnung

Misses

- Miss Rate: $\frac{\#\text{ Misses}}{\#\text{ Zugriffe}}$
- Typisch: $\approx$ 5% für L_1-Cache
- Verzögerung bei Miss: 50–200 Zyklen, steigender Trend!

Hits

- Benötigte Zeit: Suche nach Eintrag + Transfer in CPU
- Typisch: 1–2 Zyklen (L_1), 5–2 Zyklen (L_2)

Miss-Rate entscheidend

- Annahme: Nachschlagen+Treffer 1 Zyklus, Miss 100 Zyklen
- Durchschnittlicher Speicherzugriff:
  - 3% Miss-Rate: $t_{avg} = 1 + 0.03 \cdot 100 = 4$ Zyklen
  - 1% Miss-Rate: $t_{avg} = 1 + 0.01 \cdot 100 = 2$ Zyklen
- Verdoppelung von t_{avg} bei 97% statt 99% Trefferquote!

Cache-Optimierungen I

Probleme

- Programme sollen Caches bestmöglich nutzen
- Cache-Aufbau und -Größen sind *nicht* Bestandteil der ISA
- Code soll auf möglich vielen Maschinen möglichst schnell laufen

Optimierungsmöglichkeiten

- Cache Misses in inneren Schleifen vermeiden (evtl. Schleifen umsortieren)
- Günstige Zugriffsmuster (»Stride 1«) anstreben (räumliche Lokalität)
- Wiederholte Zugriffe auf Variablen (zeitliche Lokalität)
- Ausrichtung (Alignment) von Datenstrukturen; blockweiser Zugriff

Cache-Optimierungen II

Matrixmultiplikation

- Häufig auftretendes Muster
- $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n \times n}$, $\mathbf{A} = \mathbf{B} \cdot \mathbf{C}$:

$$A_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

  i, k = (Zeile, Spalte)
- Naiver Algorithmus: Drei verschachtelte Schleifen

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    sum = 0;
    for (k=0; k<N; k++) {
      sum +=
        a[i][k] * b[k][j];
    }
    c[i][j] = sum;
  }
}
```

Cache-Optimierungen III

Annahmen

- Zeilengröße 32 Bytes (4× 64-Bit-Wort)
- $\frac{1}{N} \approx 0$
- Cache nicht groß genug für mehrere Matrix-Zeilen

Cache-Optimierungen III

Arrays in C

Row-Major-Konvention, d.h. jede Zeile kontinuierlich im Speicher

Zeilen durchlaufen

```
type sum;
for (i = 0; i < N; ++i) {
    sum += a[K][i];
}
```

- Zugriff auf sukzessive Elemente
- Annahme: Blockgröße B im Cache; $B >$ sizeof(type)
- Compulsory Miss-Rate: $\frac{\text{sizeof(type)}}{B}$

Spalten durchlaufen

```
type sum;
for (i = 0; i < N; ++i) {
    sum += a[i][K];
}
```

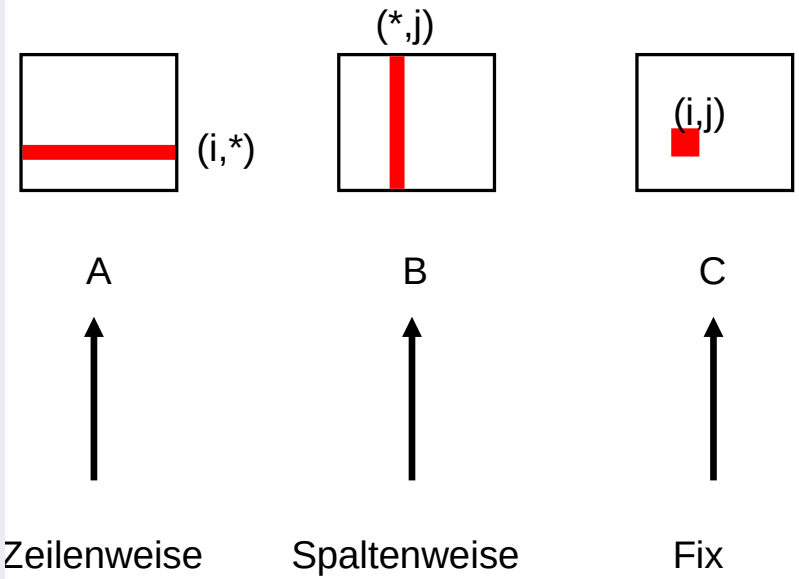
- Weit entfernte Speicherstellen
- Compulsory Miss-Rate: 100%

Cache-Optimierungen IV: Analyse I

Zugriffsschema: *ijk* und *jik*

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    sum = 0;
    for (k=0; k<N; k++) {
      sum +=
        a[i][k] * b[k][j];
    }
    c[i][j] = sum;
  }
}
```

Speichermuster



Cache-Misses innere Schleife

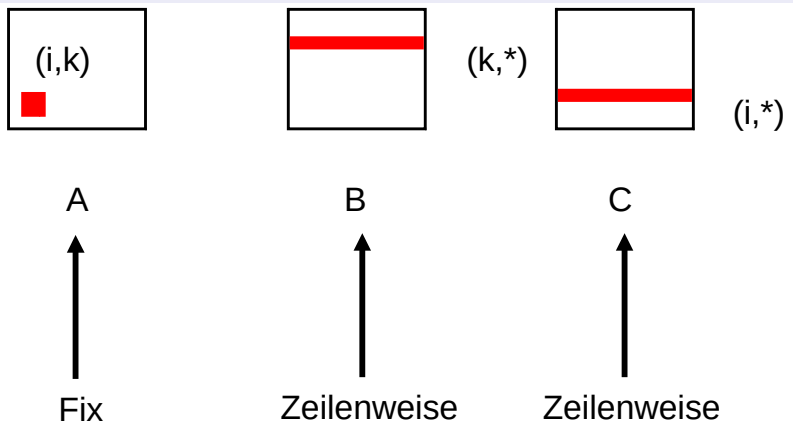
A : $\frac{\text{sizeof(type)}}{B}$, **B**: 100%, **C** : 0%

Cache-Optimierungen IV: Analyse II

Zugriffsschema: *kij* und *ikj*

```
for (k=0; k<N; k++) {
  for (i=0; i<N; i++) {
    r = a[i][k];
    for (j=0; j<N; j++) {
      c[i][j] += r * b[k][j];
    }
  }
}
```

Speichermuster



Cache-Misses innere Schleife

$$\mathbf{A}: o, \mathbf{B}: \frac{\text{sizeof(type)}}{B}, \mathbf{C}: \frac{\text{sizeof(type)}}{B}$$

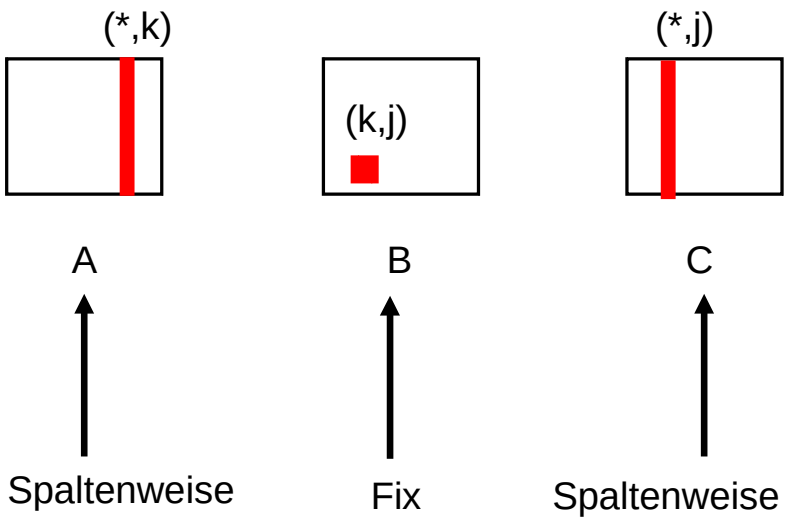
Abbildung basiert auf Bryant & O'Hallaron

Cache-Optimierungen IV: Analyse III

Zugriffsschema: *jki* und *kji*

```
for (j=0; j<N; j++) {
  for (k=0; k<N; k++) {
    r = b[k][j];
    for (i=0; i<N; i++) {
      c[i][j] += r * a[i][k];
    }
  }
}
```

Speichermuster



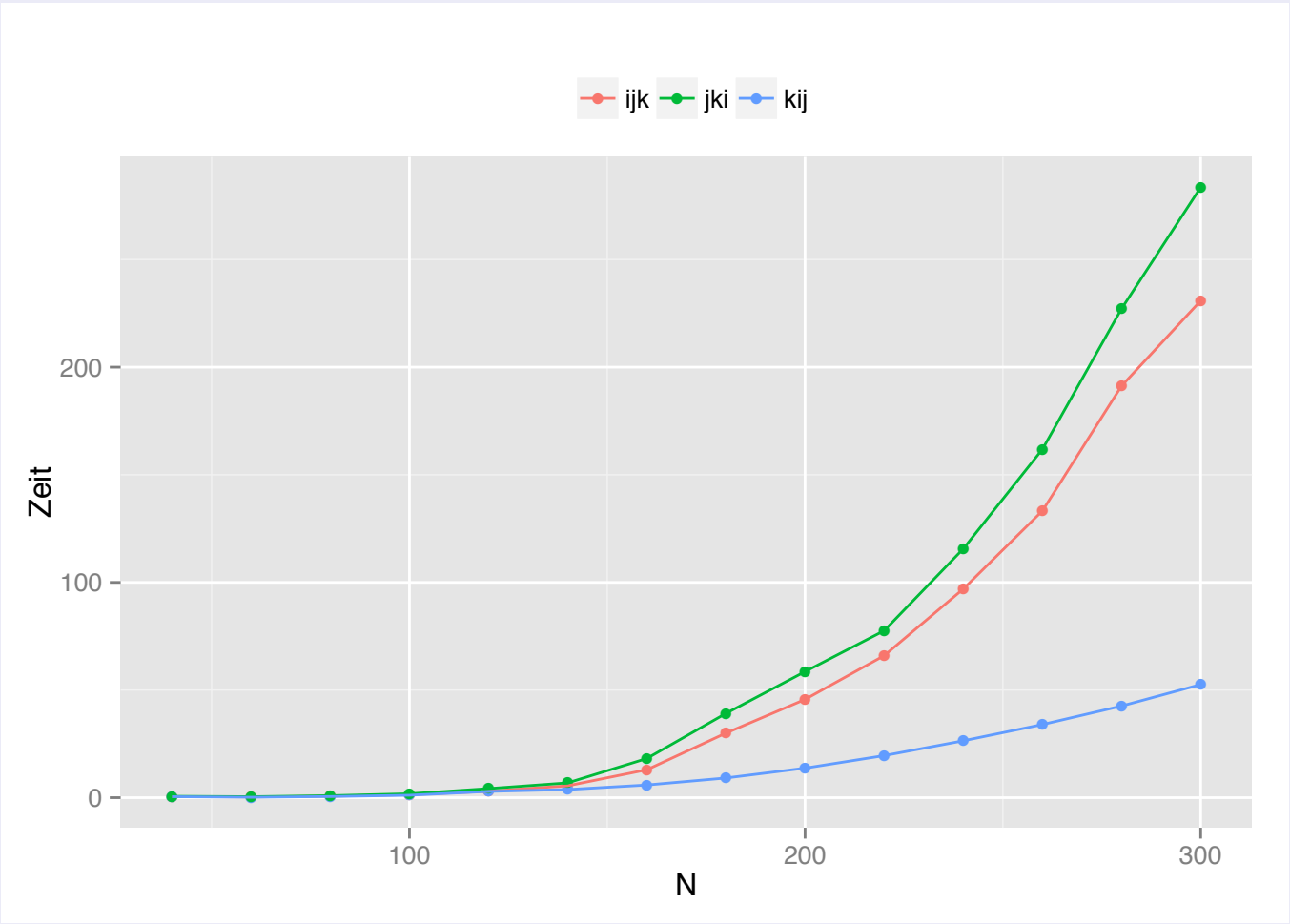
Cache-Misses innere Schleife

A : 100%, **B** : 0%, **C** : 100%

Abbildung basiert auf Bryant & O'Hallaron

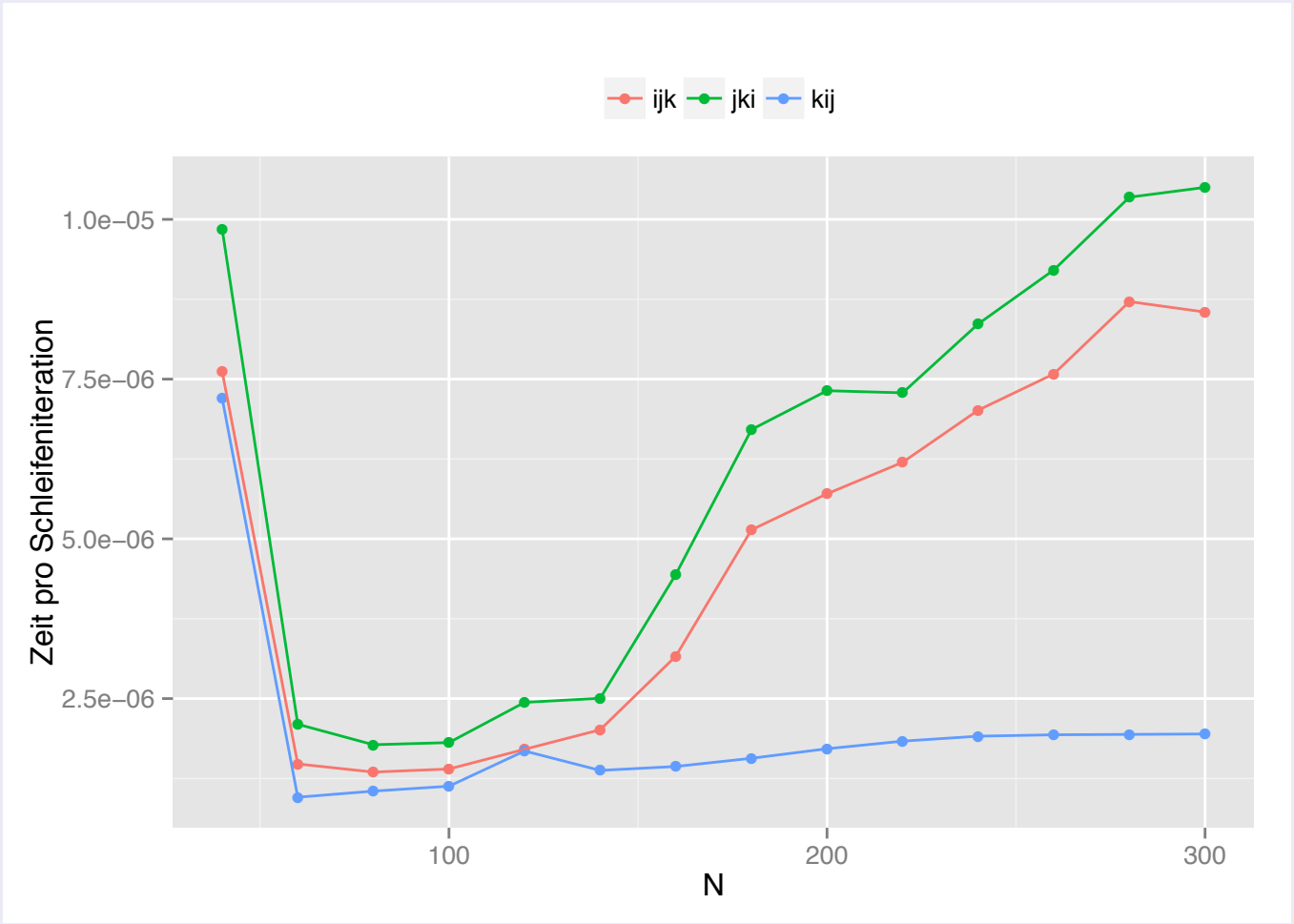
Cache-Optimierungen V: Messung

Messung auf Raspberry Pi



Cache-Optimierungen V: Messung

Messung auf Raspberry Pi



Inhalt

Betriebssysteme

Betriebssysteme

- Programm, das zusammen mit Eigenschaften eines Rechensystems die Abwicklung eines oder mehrerer Programme steuert
- Fundamentale Softwareschicht in einem Rechner
- Verwaltung für Betriebsmittel wie Speicher, E/A-Geräte, und Prozessorzeit

Beispiele

- CP/M, DOS, Windows
- Linux, Mac OS, iOS, SunOS (Solaris), HP/UX, FreeBSD, OpenBSD
- Sinix, Xenix, VMS, OS/2
- FreeRTOS, RTEMS, eCos, PSOS, vxWorks, QNX, Xenomai

Betriebssysteme II

Primäre Aufgaben

- Verteilung/Verwaltung von Betriebsmitteln
- Abstraktion von Betriebsmitteln
  - Uniformes Programmiermodell für unterschiedlich realisierte Ressourcen (Blockgeräte, Netzwerkarten, Busse, …)
  - Dateisysteme
- Zugriffskontrolle und Schutzmechanismen
  - Mehrbenutzerfähigkeit
  - Lokale und entfernte Dienste
- Kooperation zwischen Programmen

Sekundäre Aufgaben (Distribution vs. Kernel)

- Benutzerschnittstelle
- Werkzeuge

Betriebssysteme III

Drei Sichtweisen

- Erweiterte, virtuelle Maschine
  - Uniformes Programmiermodell: Programme unabhängig von HW (ARM, x86, …)
  - Gleichmäßige Fähigkeiten (Supercomputer vs. Mobiltelefon)
- Ressourcen-Manager
- Funktionsbibliothek mit Maschinenzugriff
  - Programme: Kein direkter Kontakt mit BS
  - *Systemaufrufe* einziges Mittel zur Kommunikation

Betriebssysteme: Grundstruktur I

Grundlegende Aufgaben

- Prozessverwaltung, Taskwechsel und Scheduling
- Adressraumverwaltung
- Systemaufrufe
- Gerätetreiber
  - Zeichen- und Blockgeräte
  - Netzwerkkarten
  - Busse
  - …
- Dateisysteme

Betriebssysteme: Grundstruktur II

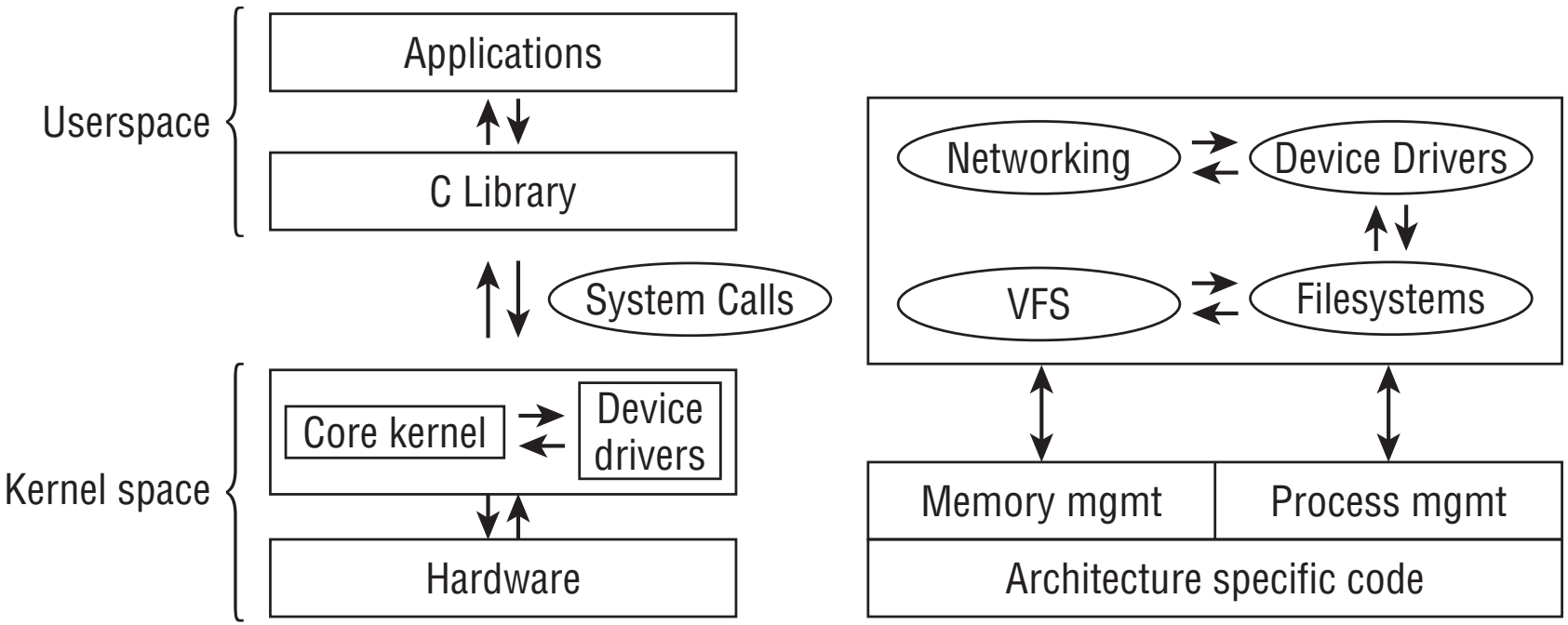
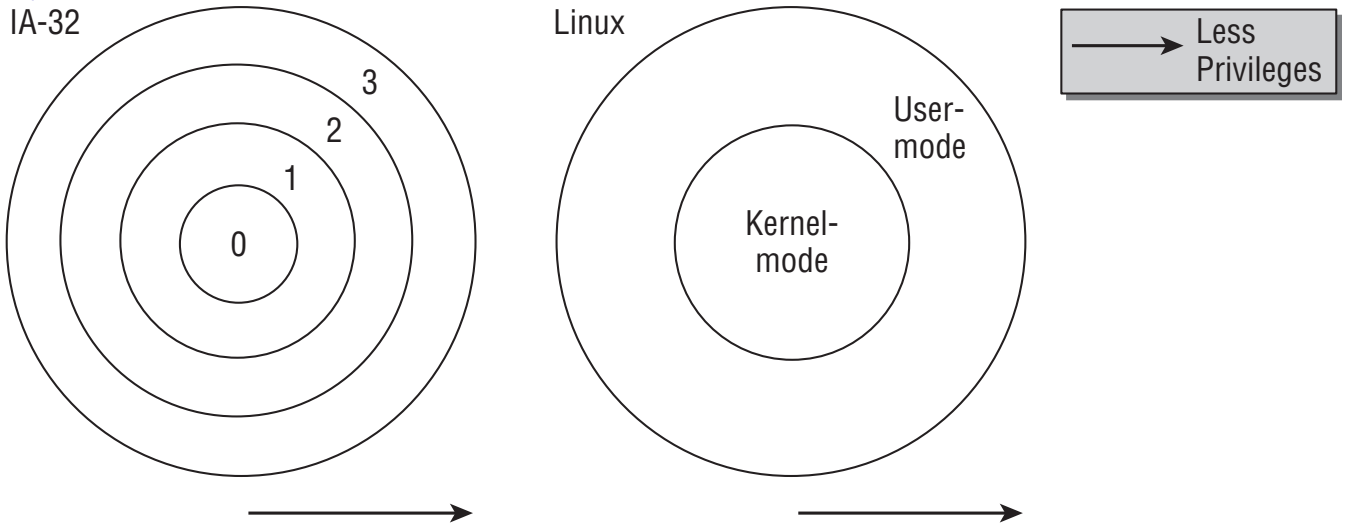


Abbildung basiert auf PLKA

Betriebssysteme: Grundstruktur III

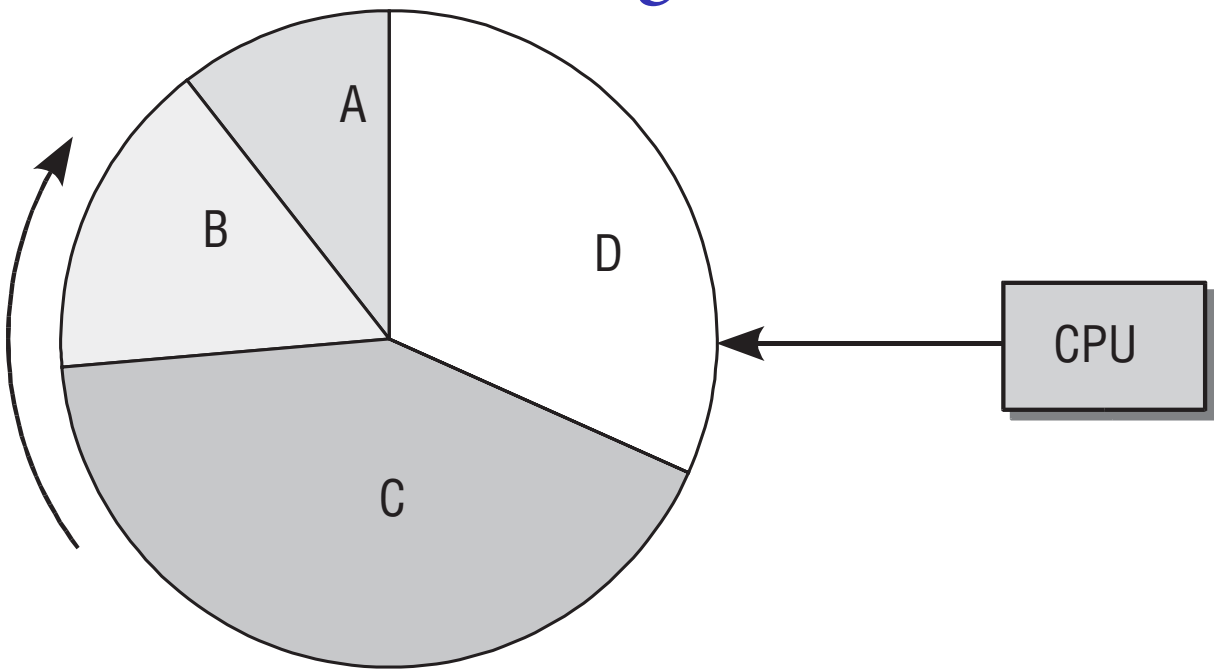


Privilegstufen

- Kernel-Mode: Unbeschränkter Betriebsmodus
- User-Mode: Nur Zugriff auf »ungefährliche« Ressourcen
- Manche Prozessoren: Zwischenstufen (historische Verwirrung…)
  - Typische BS: Kernel/User ausreichend
  - Achtung: Virtualisierung!

Abbildung basiert auf PLKA

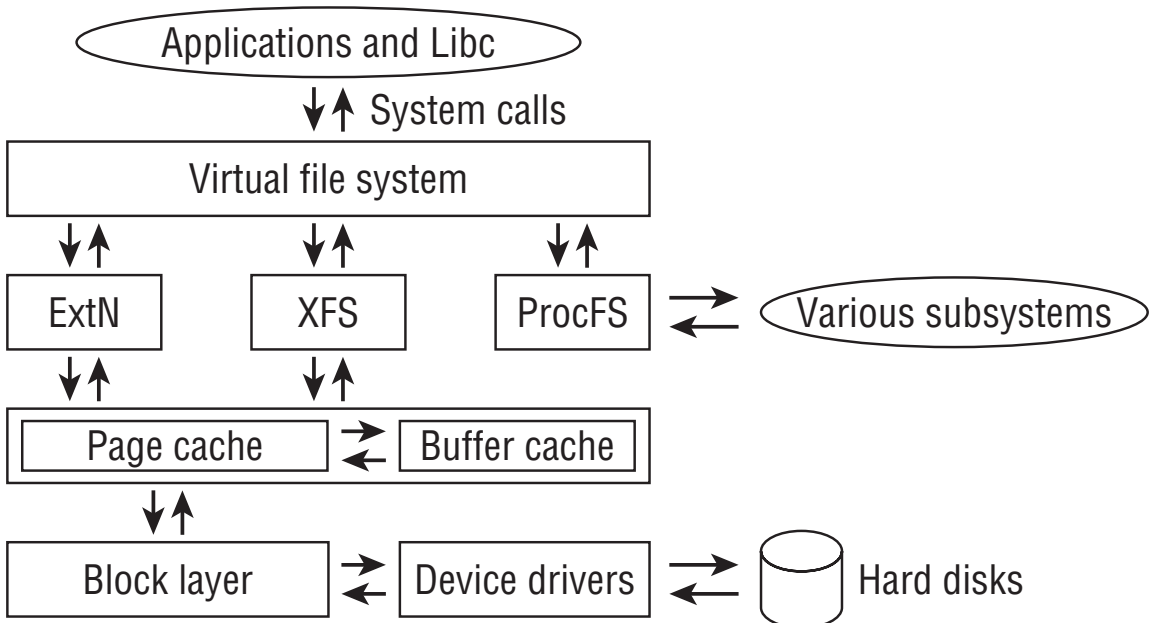
Grundstruktur IV: Multitasking



Multitasking: Pseudoparallele Ausführung

- Mehrere Prozesse quasi-gleichzeitig auf einer CPU
- Jeder Prozess: Zeitbudget je nach Wichtigkeit (typische Frequenz: Millisekunden)
- Kernel initiiert und kontrolliert Taskwechsel

Grundstruktur V: Dateisysteme



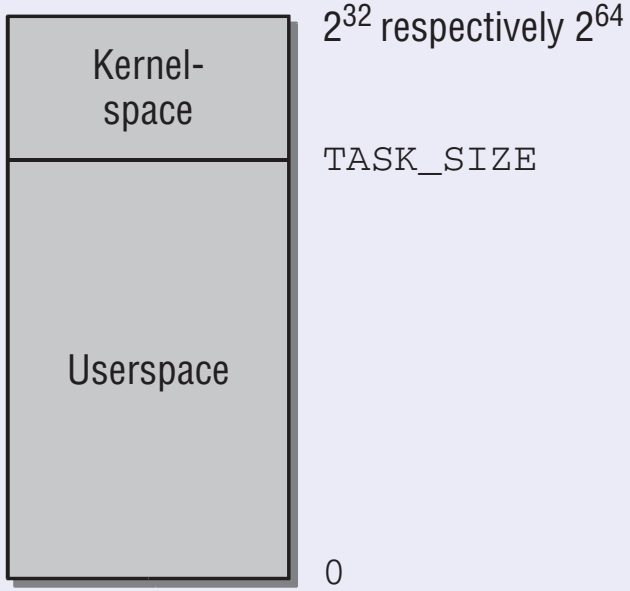
- Zugriff auf Dateien über *hierarchischen Namensraum*
  - `/path/to/file1.txt`, `/another/path/a.out`
  - Unix: Baum-Hierarchie; Windows: Laufwerksbuchstaben + Baum
- VFS: Generische Methoden (lesen, schreiben, …)
- Caches: RAM als Zwischenspeicher, BS-verwaltet

Betriebssysteme: Details

Betriebssysteme: Details

Prozessverwaltung I

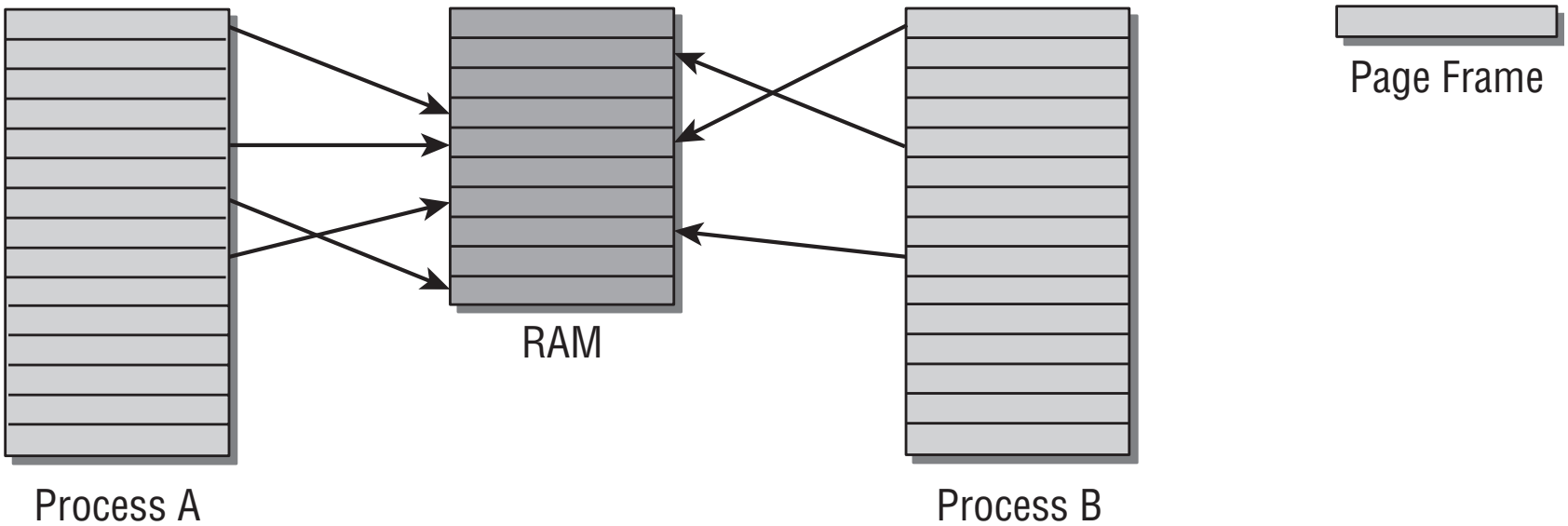
Adressraum



Eigenschaften

- Jeder Prozess: Identischer *virtueller* Adressraum
- Zugeordnet: Prozesskontrollblock mit Statusinformationen
  - Offene Dateien
  - Netzwerkverbindungen
  - Privilegien
  - Registerinhalte
  - Ablaufzustand (Laufen, Warten, …)
  - …

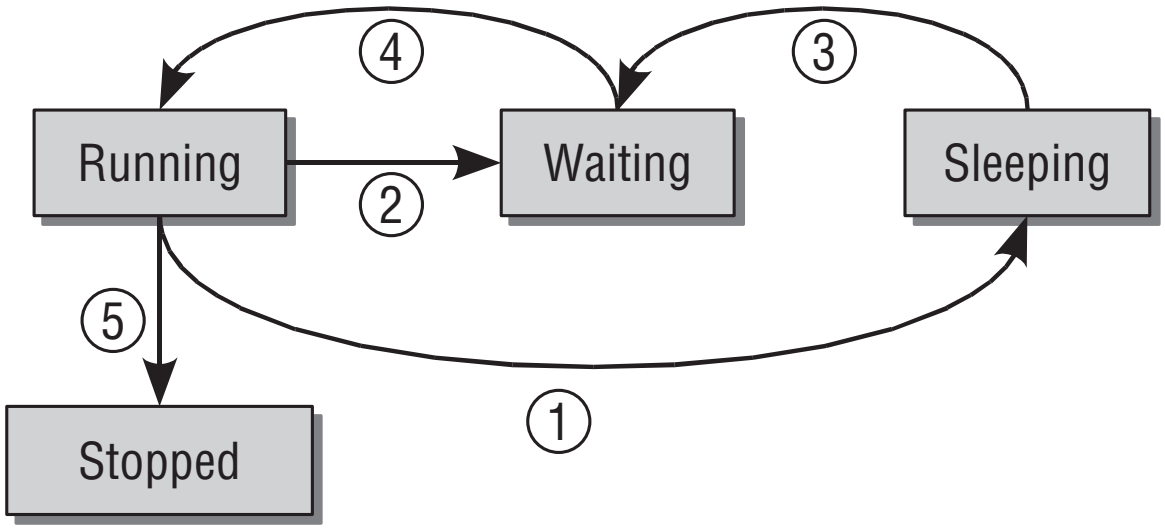
Prozessverwaltung II



Virtueller und physikalischer Speicher

- Gleiche Sichtweise für jeden Prozess ☞ (Deutlich) mehr virtueller als physikalischer Speicher
- Lösung: Dynamische Zuordnung über Memory Management Unit (MMU)

Prozessverwaltung III



Prozessübergänge

1. Warten auf Ereignis (bsp. Eingabe); Prozess gibt CPU frei
2. CPU (unfreiwillig) entzogen; anderer Prozess darf laufen
3. Erwartetes Ereignis eingetroffen
4. CPU wird an ablaufbereiten Prozess gegeben
5. Arbeit des Tasks beendet

Scheduling und Taskwechsel I

Ablaufplanung (Scheduling)

- *Policy:* Wann?
- *Mechanism:* Wie?
- Schedulingverfahren: Hohe Variantenvielfalt
  - Hier: Round Robin
  - Ansonsten: Siehe Vorlesung »Betriebssysteme«
- Timer-Interrupt: Regelmäßige Unterbrechung ☞ Scheduling
- Präemption: Vorzeitige Ablösung
  - Wichtigere Prozesse unterbrechen weniger wichtigere Prozesse
  - Kernel unterbricht Prozesse
  - Interrupts unterbrechen Kernel

Scheduling und Taskwechsel II

Scheduling: Mechanismus

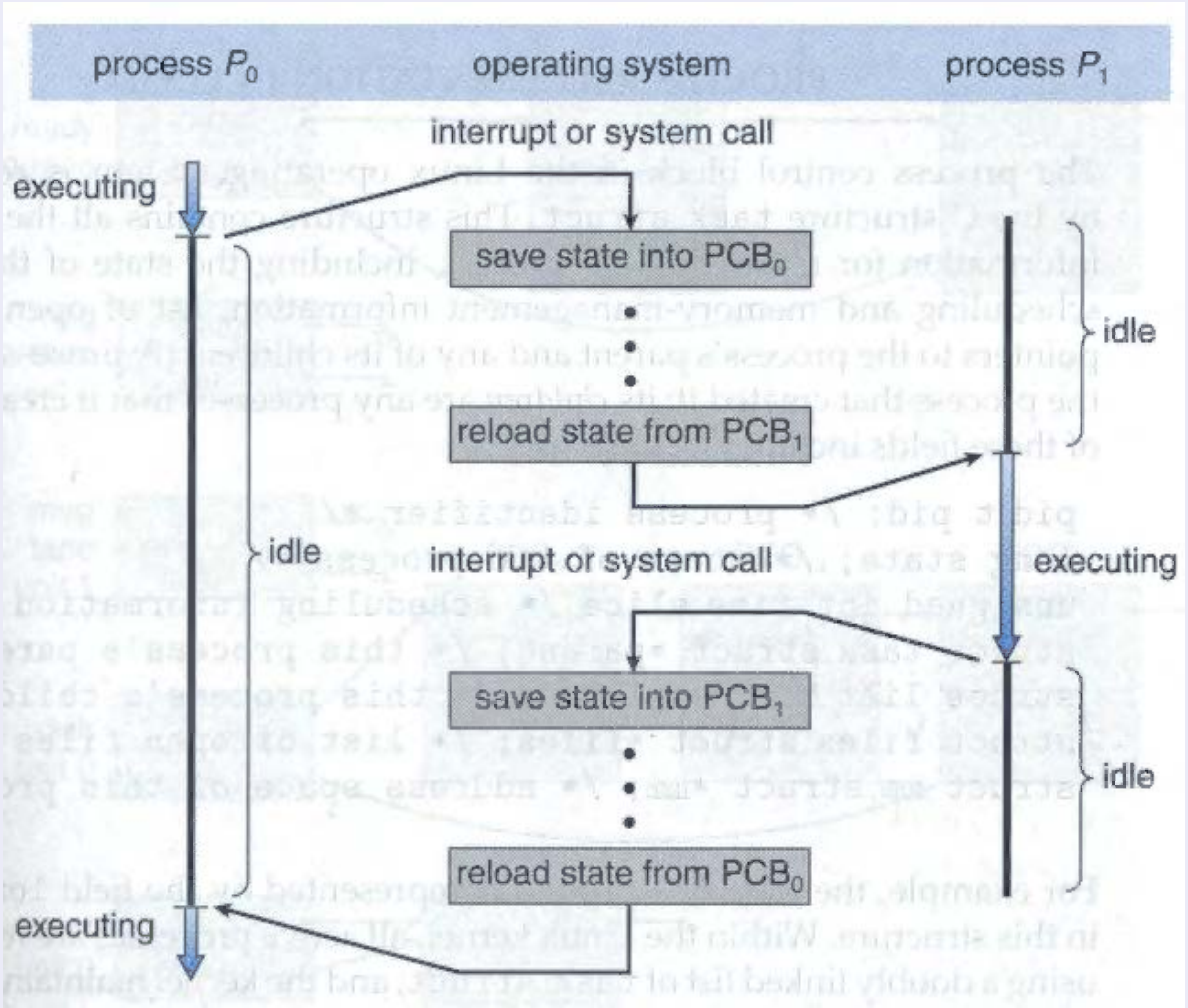
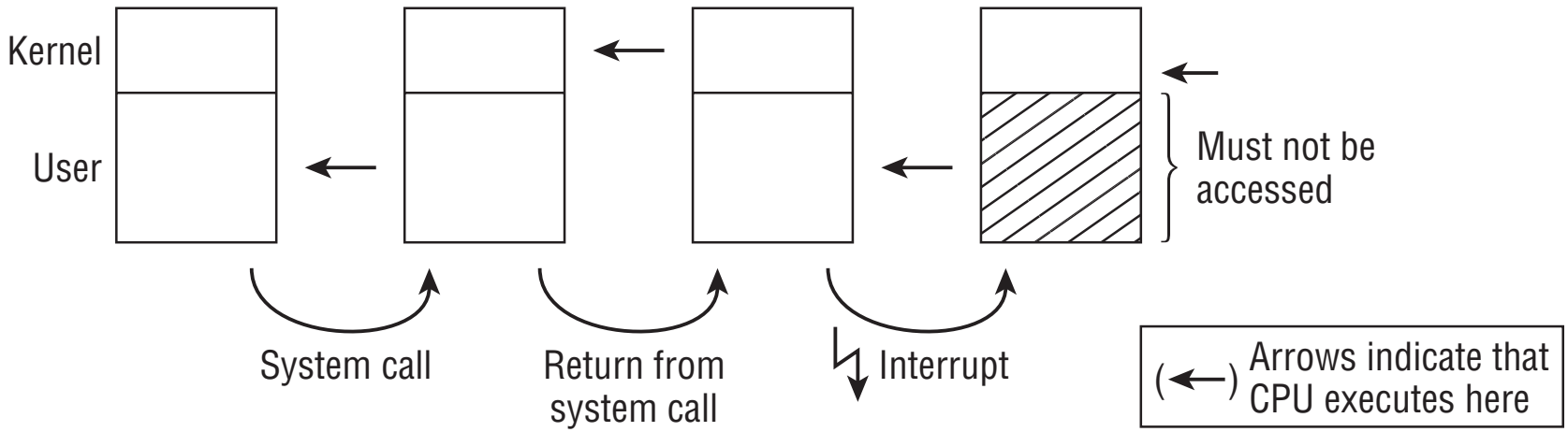


Abbildung basiert auf Silberschatz, Galvin & Gagne

Systemaufrufe I



Systemaufrufe

- Benötigt Parameter und Rückgabewert(e)
- Problem: Unterschiedliche Stacks
- Lösung: Register
  - Konvention BS- und Architekturspezifisch
  - Folgt nicht notwendigerweise Userland-ABI
- Wechsel User- nach Kernelmodus: Spezielle Assembler-Befehle

Systemaufrufe II: ARM-Spezifika

Systemaufrufe auf ARM

- Spezialanweisung: `svc` (*SuperVisor Call*)
  - Alte Nomenklatur: `swi` (Software Interrupt)
  - Beide Mnemonics verwendbar, gleiche Opcodes
- Details: Siehe Übung 13

Adressraumverwaltung I

Adressraum: Komponenten

- Virtuelle Adressen
- Physikalische Adresse
- Page Frames (Speicherseiten: konsekutive Speicherblöcke, typischerweise 4 KiB)

Aufgaben

- Zuordnung physikalischer aus virtueller Adresse
- Verwaltung Zuordnungstabellen
- Verwaltung Page Frames
  - Datenquellen (Blockspeicher oder dynamisch erzeugt)
  - Welche Daten am häufigsten gebraucht?
  - Entfernen und Laden von Page Frames

Adressraumverwaltung II

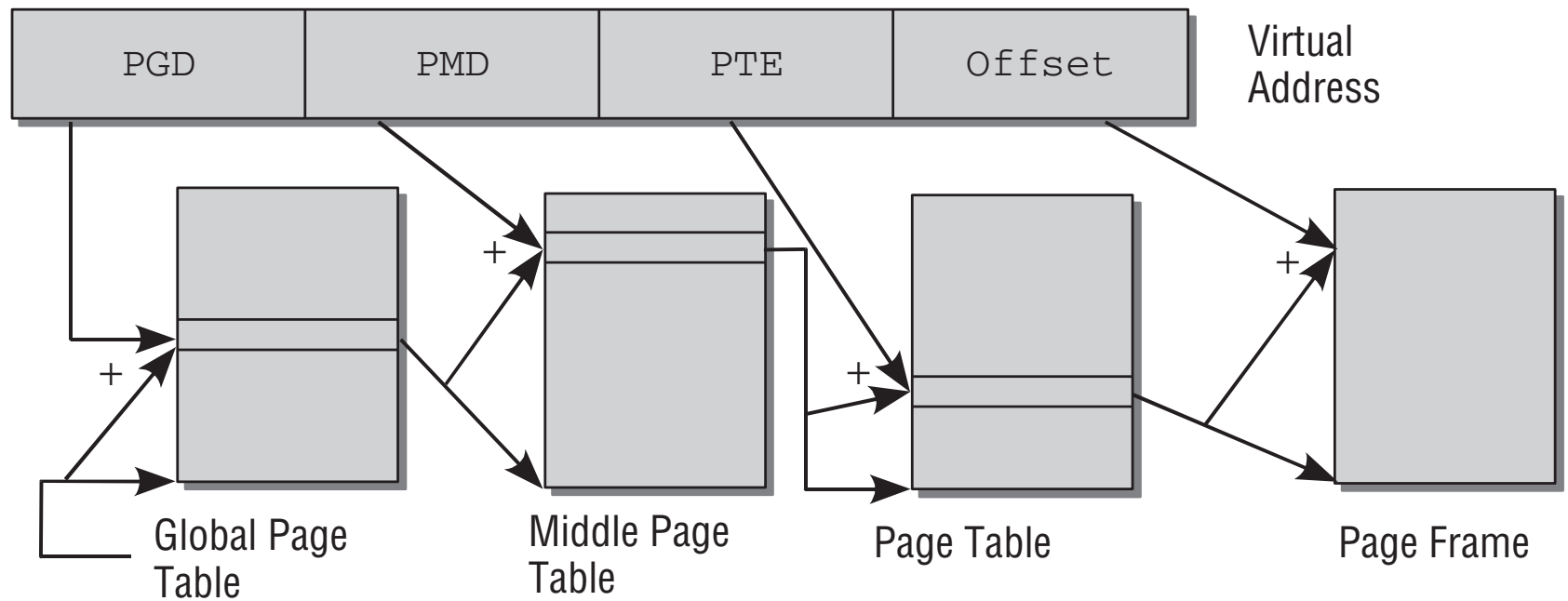
Grundproblem

- Gegeben: Virtuelle Adresse (Prozess-spezifisch)
- Gesucht: Physikalische Adresse (Systemglobal)
- Beobachtung: Virtueller Adressraum meist dünn besiedelt

Vorgehensweise

- Virtuelle Adresse in Teilkomponenten (»Schlüssel«) aufspalten
- Datenstruktur Virt→Phys mit Schlüsseln durchsuchen
- Ziel: Page Frame (enthält gesuchtes Byte)

Adressraumverwaltung III



Beispiel: Siehe Tafel

Adressraumverwaltung IV

MMU-Operation: Beschleunigung durch Caches

- Lookup-Operation: Speicherzugriffe und Zeiger verfolgen
- Vervielfachung Kosten Speicherzugriff
- Effizienter Cache notwendig: *Translation Lookaside Buffer* (TLB)
  - Typischerweise mehrstufig
  - Raspberry Pi (ARM):
    - Micro-TLB für Code und Daten mit je 10 vollassoziativen Einträgen
    - Haupt-TLB: 8 Vollassoziative, 64 2-fach assoziative Einträge
  - Core i5 (x86):
    - Daten-TLB: 4-fach assoziativ, 64 Einträge
    - Instruktions-TLB: 4-fach assoziativ, 128 Einträge
    - Geteilter TLB Stufe 2: 4-fach assoziativ, 512 Einträge

Adressraumverwaltung V

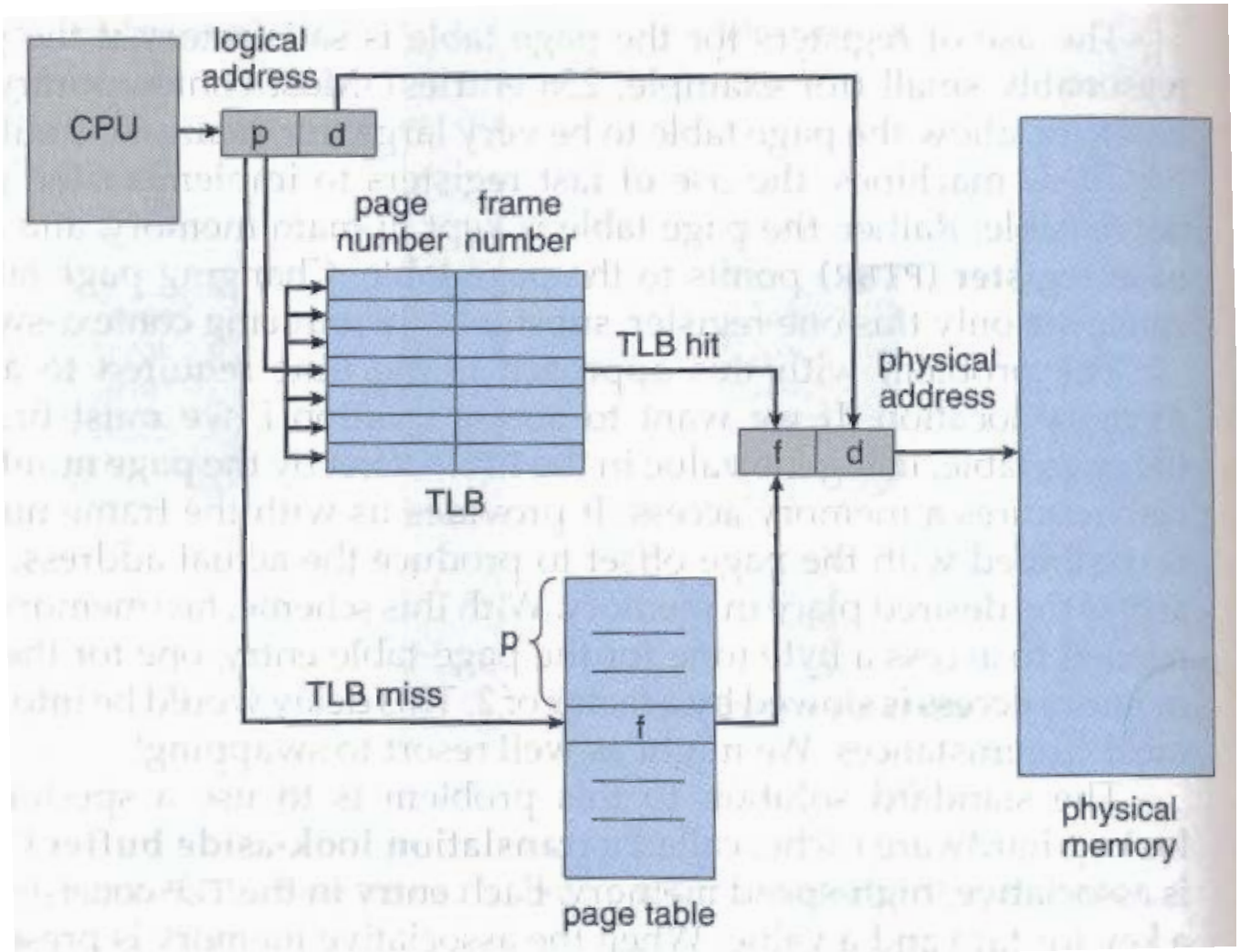


Abbildung: Silberschatz, Galvin & Gagne, *Modern Operating Systems*, Wiley

Adressraumverwaltung VI

MMU: Kooperation Kernel/CPU

- Aufgaben Kernel
  - Datenstrukturen aufsetzen
  - Mappings hinzunehmen und entfernen
  - Speicher für Seitentabellen bereitstellen
  - *Page faults* behandeln (nicht vorhandene Daten bereitstellen)
- Aufgaben CPU
  - Datenstrukturen durchlaufen (*page table walk*)
  - Caches nutzen und aktualisieren
  - Daten nicht vorhanden: *Page fault* auslösen

Page Faults

Page faults für Anwendungen *nicht* direkt sichtbar!

Adressraumverwaltung VI: ARM-Spezifika

Koprozessor 15

- *Keine* speziellen Assembler-Befehle zur Kontrolle MMU und TLB
  - Koprozessor 15 regelt Aufgaben
  - Untypischer Ansatz! (Alternative: Spezial-Assemblerbefehle)
- Kontrollregister: `c0` bis `c15`
- Manipulation mit Koprozessor-Befehlen (`mrc` etc., siehe VFP-Beschreibung)
- Funktionsgruppen:
  - System-Konfiguration (grundlegende Einstellungen)
  - MPU-Konfiguration (Speicherschutzeinheit)
  - MMU-Konfiguration (Virtueller Speicher)
  - Cache-Konfiguration
  - TCM-Konfiguration (tightly coupled memory)
  - Debugging & Performance-Messungen
- *Achung:* Chip-spezifische Sonderfunktionen!

Interrupts (Unterbrechungen) I

Ereignisse

- Extern: Tastaturklick, Datenpaket eingetroffen, …
- Intern: Timer abgelaufen, DMA-Transfer beendet, …

Reaktion auf Unterbrechung

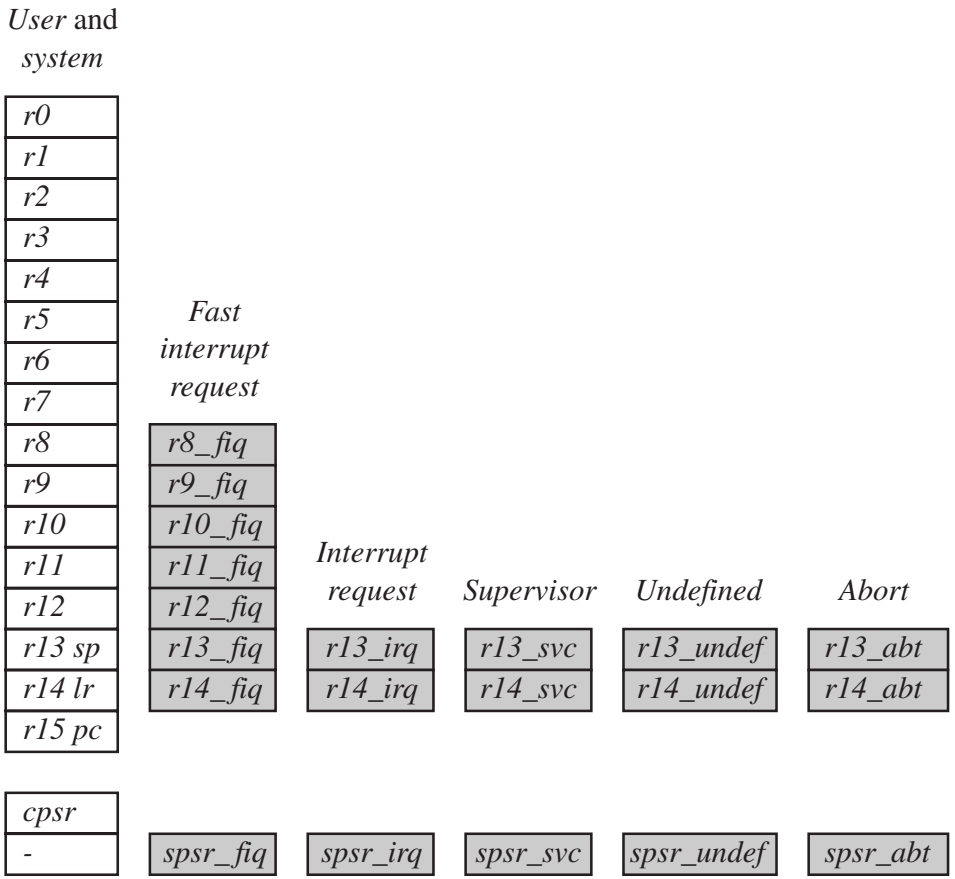
- CPU: Baldmöglichste Reaktion auf Interrupt
- Gerade abgearbeitetes Programm wird *präemptiert*
- Interrupts (je nach CPU) priorisierbar ☞ Interrupt-Präemption
- Interrupt-Handler bzw. Interrupt Service Routine:
  - Unterbrechungsgrund ermitteln (Statusprüfung)
  - Quittieren (wenn nötig – diverse Interrupt-Typen)
  - Notwendige Aktionen ausführen (z.B. Netzwerkpaket ins RAM kopieren)

Interrupts II: Zeitfluss

Interrupt: Zeitfluss

- CPU empfängt Unterbrechungsanforderung
- Moduswechsel CPU; Interrupts werden gesperrt
- Sprung in *Interrupt-Vektor-Tabelle*
  - IRQ-Kennzahl ☞ Tabellen-Eintrag
  - Eintrag enthält Sprung zu ISR (Interrupt Service Routine)
  - ISR verwendet (typischerweise) eigenen Stack
- Interrupts entsperren
- Rückkehr in normalen Ausführungsmodus

Interrupts III: ARM-Spezifika I



Mehr Register

- *Banked Registers*: Eigene physikalische Register während Unterbrechung aktiv
- Restliche Register: Save/Restore erforderlich
- FIQ: Schneller, da weniger Register zu sichern

Interrupts IV: ARM-Spezifika II

CPU-Unterstützung bei Interrupts

- CPSR in Modus-spezifisches Bank-Register speichern
- PC-Register in Modus-spezifisches LR-Register speichen
  - Rücksprung an unterbrochene Adresse
- Exception-Modus in CPSR setzen
- PC auf Adresse des Exception-Handlers setzen ☞ Handler wird aufgerufen

Interrupt-Stack

Modus-spezifischer Stackpointer manuell zu setzen, wenn eigener Stack gewünscht!

Interrupts V: ARM-Spezifika III

Vektortabelle

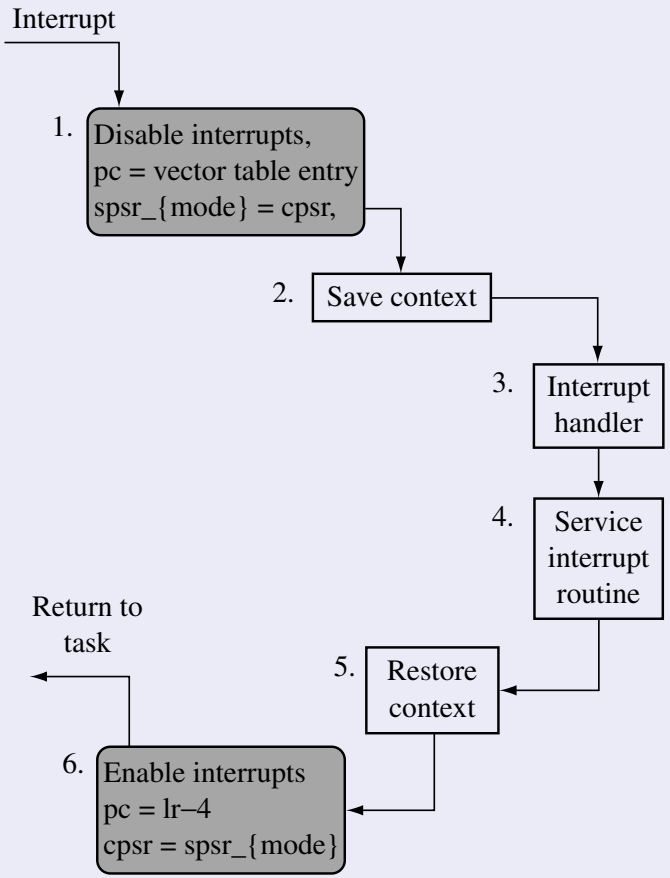
Ausnahme/Unterbrechung	Abkürzung	Adresse	Alternative
Reset	RESET	0x0	0xffff0000
Undefined instruction	UNDEF	0x4	0xffff0004
Software interrupt	SWI	0x8	0xffff0008
Prefetch abort	PABT	0xc	0xffff000c
Data abort	DABT	0x10	0xffff0010
Reserved	N/A	0x14	0xffff0014
Interrupt request	IRQ	0x18	0xffff0018
Fast interrupt request	FIRQ	0x1c	0xffff001c

Prioritäten

RESET > DABT > FIRQ > IRQ > PABT > SWI, UNDEF

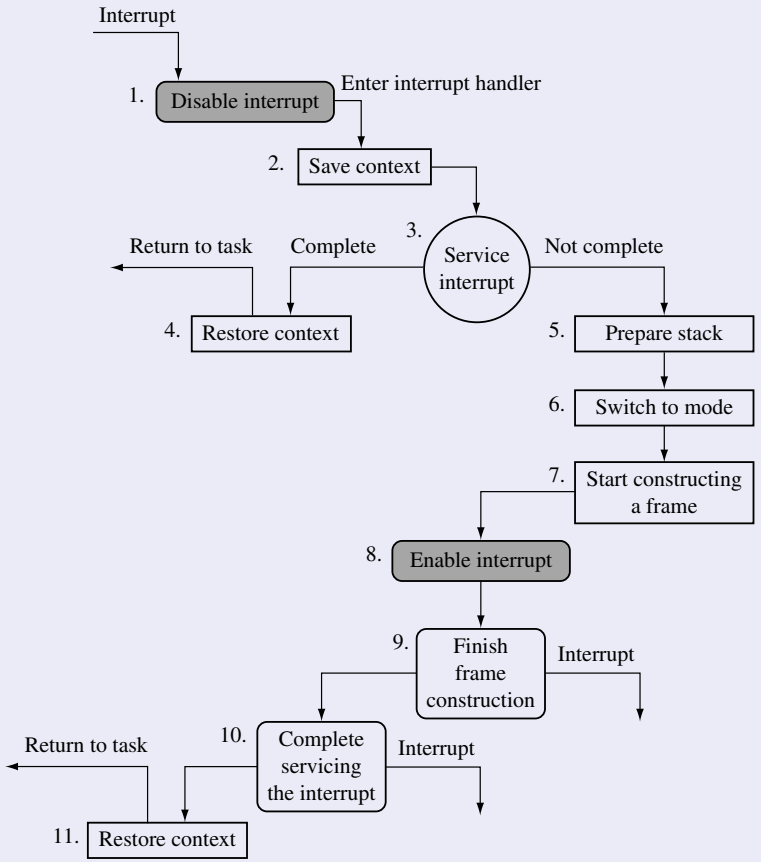
Interrupts IV: ARM-Spezifika

Nicht-verschachtelter (non-nested) Handler



Interrupts V: ARM-Spezifika

Verschachtelter (nested) Handler



Vorteil: Geringere Latenz, Nachteil: Komplexität

Ausprägungen von Betriebssystemen

Vielzweck

- Desktop, Server
- Multi-Architektur-Unterstüztung
- Features und Gerätetreiber essentiell

Echtzeit

- Reaktion auf Ereignisse
- Determinismus & Latenz essentiell
- Verspätung ☞ Katastrophe

Generiert

- BS-Instanz aus Quellcodebasis erzeugt
- Resultat konfigurierbar
- Beispiel: Fixe Anzahl Tasks, kein Netzwerk

Deeply Embedded

- Beschränkung auf einfachste Funktionen
- Beispiel: Scheduler ohne MMU
- Bare Metal: Weiche Grenze

Blockgeräte und Dateisysteme: Ansichten

Benutzer

- Daten in Dateien abgelegt
- Dateien hierarchisch in *Baum* organisiert
- Dateien besitzen Meta-Informationen (Letzte Modifikation, Zugriffsrechte, …)

System

- Blockgeräte: Lineare Kette von Blocks

Übersetzung

- *Dateisysteme*: Übersetzen zwischen Sichtweisen
- VFS-Schicht: Parallelbetrieb mehrerer Dateisysteme

Dateisysteme I

Beispiele

- *Vielzweck-Dateisysteme*
  - Traditionelle Struktur: FAT, ext2/3/4, UFS, …
  - Neuere Ansätze (vorwiegend B^+-Bäume): XFS, JFS, NTFS, XFS, Btrfs, HAMMER, ZFS …
- *(Raw) Flash-Dateisysteme*: UBIFS, JFFS2, YAFFS, …
- *Netzwerkdateisysteme*: NFS, CIFS (Samba), GlusterFS, AndrewFS, Coda, 9P, …
- *Read-Only-Dateisysteme*: ISO9660 (CD-ROM/DVD), romfs, …

Substantielle Komplexitätsunterschiede

- XFS, btrfs: ≈ 100 kLOC
- ext2: 9kLOC, iso9660: 4kLOC, MinixFS: 2.4 kLOC

FAT32 I

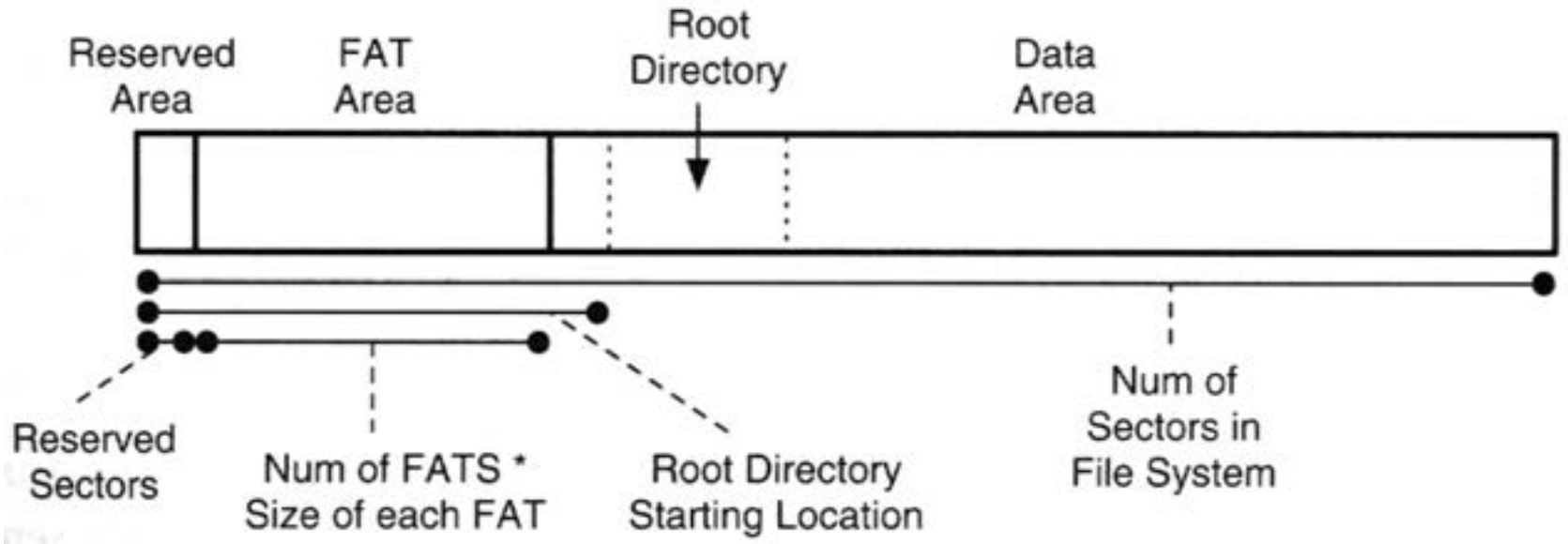
File Allocation Table (FAT32)

- Altes Dateisystem (> 30 Jahre, Ursprung: QDOS)
- Einfache Struktur, zahllose Erweiterungen
  - Modernen Anforderungen nicht gewachsen
  - Universelle Verfügbarkeit (einfacher Code): USB-Sticks, SD-Karten, …
  - Patentprobleme!
- FAT: 8.3, VFAT: 255 Buchstaben pro Dateiname

Technische Grundlagen

- Aufteilung Blockgerät in *Cluster*
  - Zwischen 512 Bytes (1 Sektor) und 64 KiB (128 Sektoren)
  - Kleinste zuordenbare Einheit
  - Sektoren im Cluster: Kontinuierlich
- Verkettete Liste: Zuordnung meherere Cluster ⇔ Datei

FAT32 II

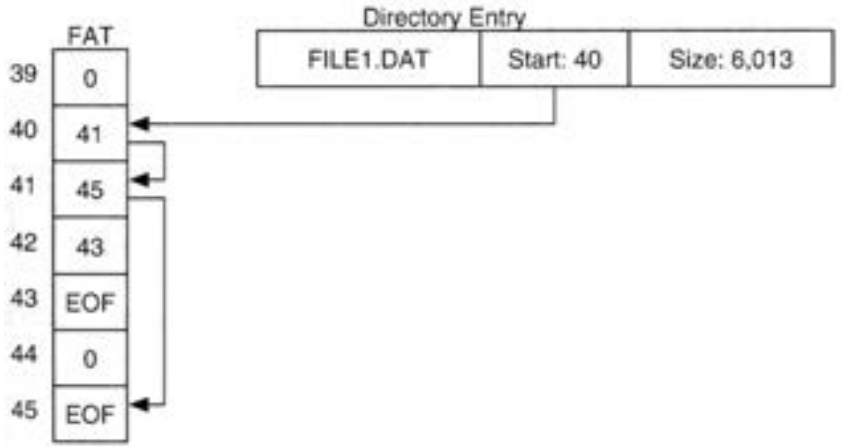


Struktur

- FAT-Bereich: Zuordnungstabelle Cluster zu Dateien
  - Ein FAT-Eintrag pro Cluster (benutzt,
  - Mehrere Kopien der FAT pro Dateisystem
- Datenbereich: Nutz- und Metadaten

Bildquelle: B. Carrier, *File System Forensics*, Addison-Wesley, 2005

FAT32 III

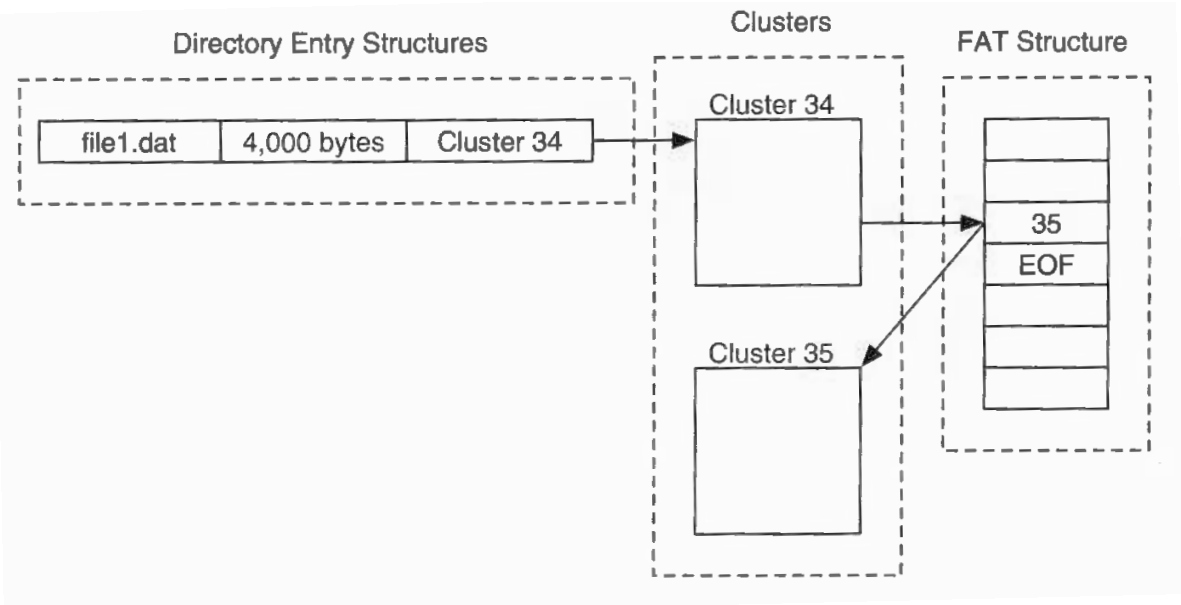


Verzeichnisse realisieren

- FAT: Verkettung mehrerer Cluster einer Datei
- Verzeichniseintrag: Dateiname und Startcluster

Bildquelle: B. Carrier, *File System Forensics*, Addison-Wesley, 2005

FAT32 III



Verzeichnisse realisieren

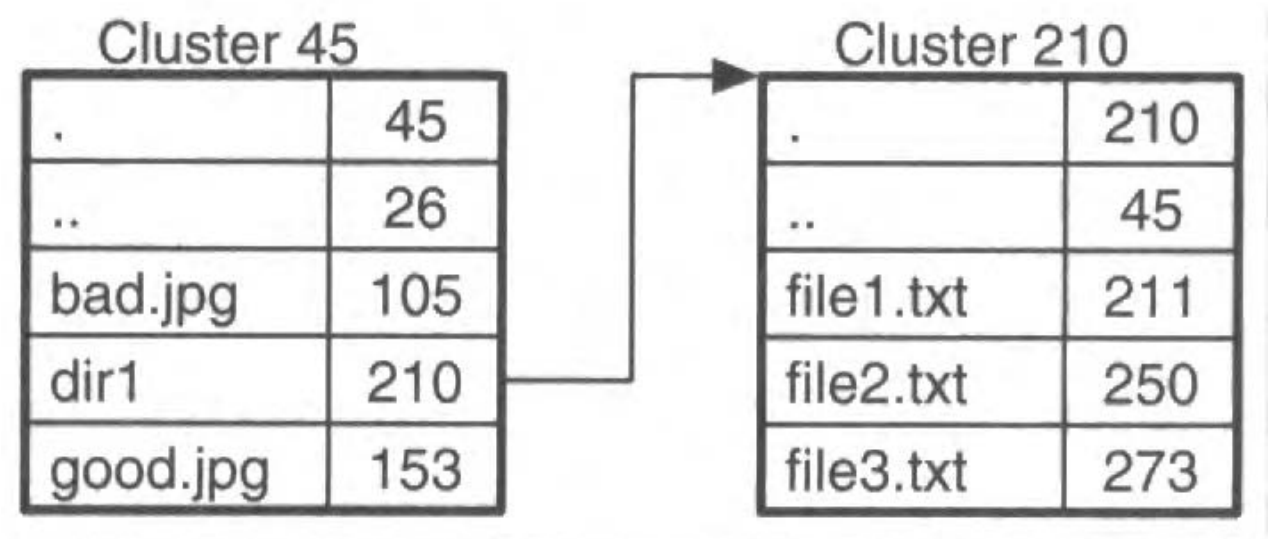
- FAT: Verkettung mehrerer Cluster einer Datei
- Verzeichniseintrag: Dateiname und Startcluster

Bildquelle: B. Carrier, *File System Forensics*, Addison-Wesley, 2005

FAT32 IV

Verzeichnisse (*Directories, Folders*)

- Verzeichnis wird durch Datei repräsentiert
- Inhalt: Verweise auf alle Dateien im Verzeichnis
- Rekursive Anwendung ☞ Verzeichnisbaum



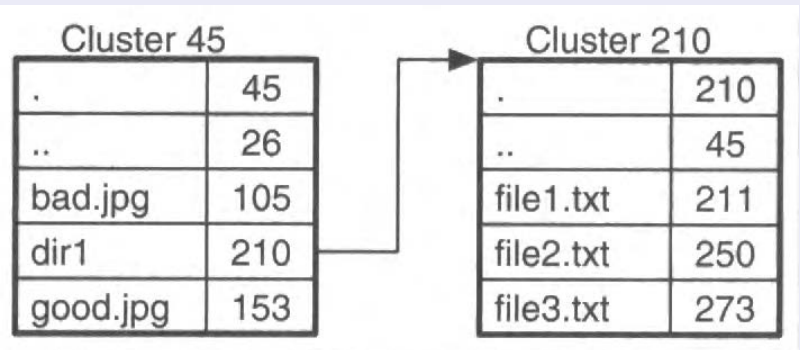
Bildquelle: B. Carrier, *File System Forensics*, Addison-Wesley, 2005

FAT32 V

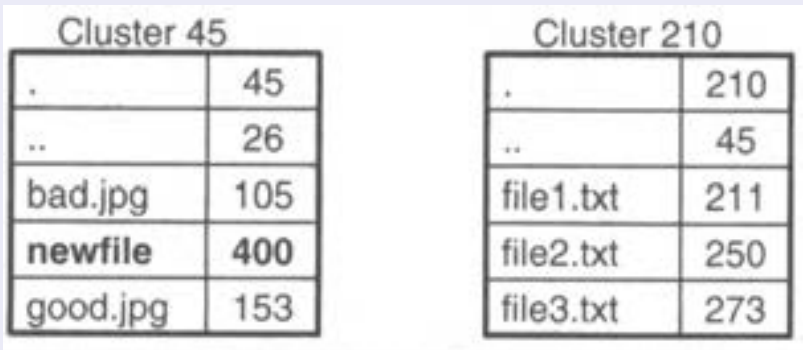
Löschen von Dateien/Verzeichnissen

- Eintrag aus Verwaltungsstruktur invalidieren
  - Kann später wiederverwendet werden
- Cluster als unbelegt markieren
  - *Kein* physikalisches Löschen/Überschreiben
  - Wiederherstellungsmöglichkeit + NSA

Vorher



Nachher



Index-basierte Dateisysteme I

Verkettung: Nachteile

- Wahlfreier Zugriff ☞ Lange Kettenverfolgung $\mathcal{O}(n)$
- Unbelegter Speicherplatz ☞ Hoher Verwaltungsaufwand
- Alternative: Index-basierte Allokation
  - Ein Block enthält Indizes aller Dateiblöcke
  - Wahlfreier Zugriff: $\mathcal{O}(1)$

Indizierte Allokation

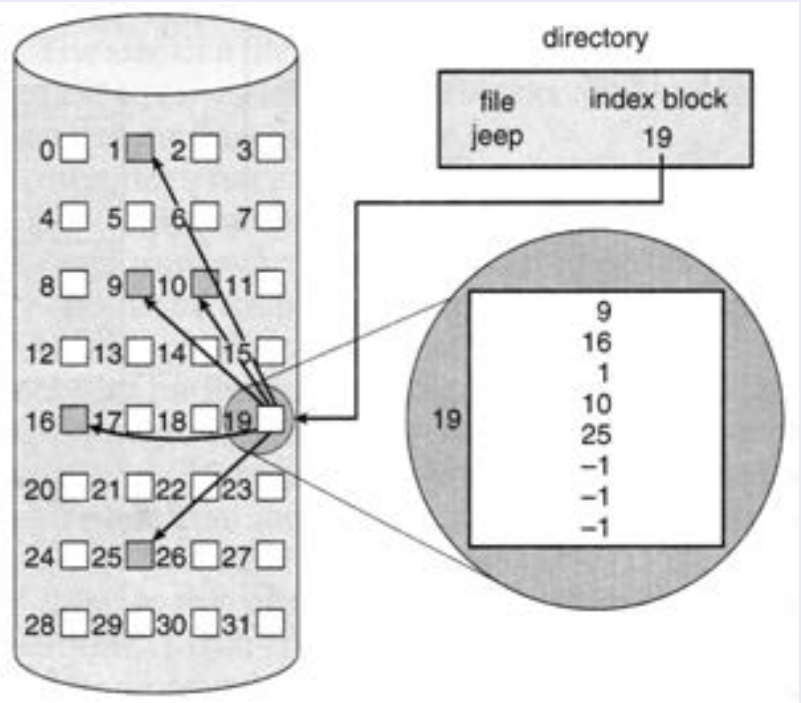


Abbildung: Silberschatz, Galvin & Gagne, *Modern Operating Systems*, Wiley

Index-basierte Dateisysteme II

Probleme: Optimale Indexblock-Größe?

- Dateien so groß wie möglich ☞ viele Einträge
- Jede Datei braucht Indexblock ☞ so klein wie möglich ☞ wenige Einträge

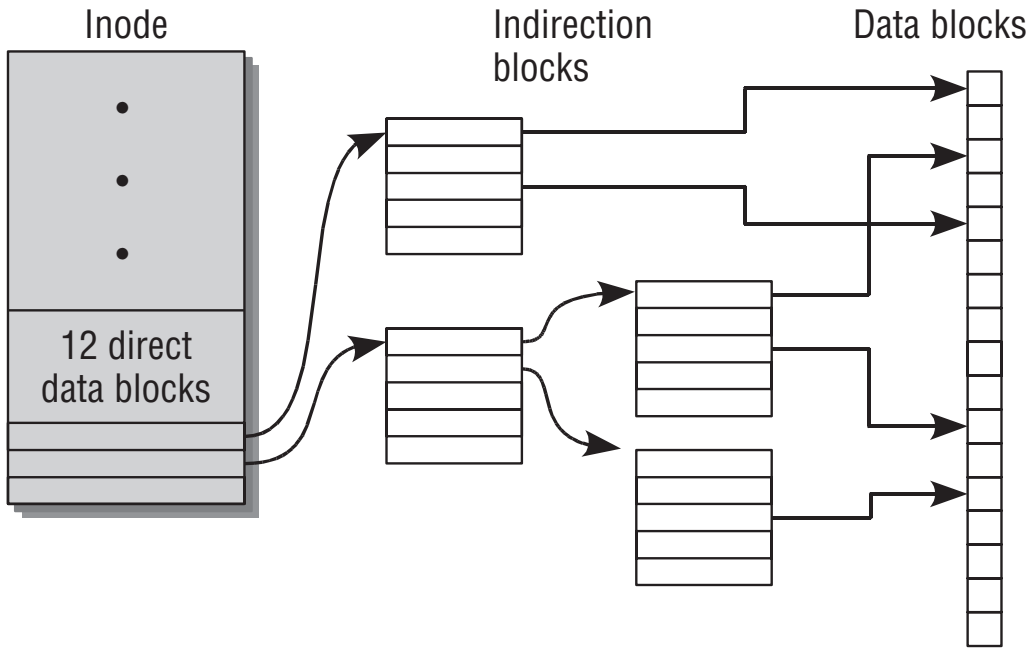
Alternativen

- Verkettete Indexblöcke
  - Wahlfreier Zugriff weiterhin $\mathcal{O}(n)$, aber weniger Durchläufe
  - Maximale Dateigröße unbeschränkt
- Mehrstufiger Index
  - Aufteilung in direkte und indirekte Indexblöcke
  - Analog für zweite, dritte, … Stufe
  - Aufwand bei wahlfreiem Zugriff $\mathcal{O}(\log(n))$
  - Maximale Dateigröße: Beschränkt

Index-basierte Dateisysteme II

Beispiel: ext2/3/4

- Bis zu drei Indirektionsstufen
- Maximale Dateigröße abhängig von Blockgröße:
  - 1024 KiB ☞ 16 GiB
  - 2048 KiB ☞ 256 GiB
  - 4096 KiB ☞ 2 TiB



Inode

- Eine Instanz pro offener Datei
- Direkte Blöcke: Kleine Dateien
- Große Dateien: Indirektion nutzen

Index-basierte Dateisysteme II

Beispiel: ext2/3/4

- Aufteilung Speicherplatz in *Blockgruppen*
- Freie Blöcke/Inodes verwaltet über *Bitmaps*

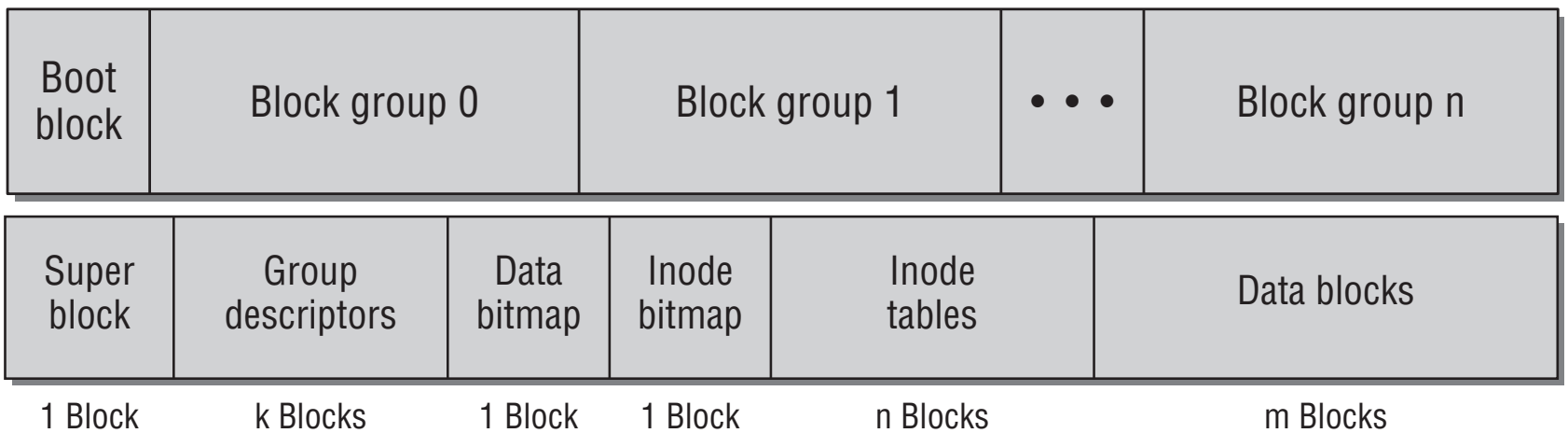


Abbildung basiert auf PLKA

Index-basierte Dateisysteme III

Beispiel: ext2/3/4

- Superblock: (Meta-)Informationen über Dateisystem
  - Magic Numer, Zeitstempel letzter Mount, Anzahl freier/belegter Blöcke, …
  - Redundant gespeichert in Blockgruppe 0, 1 und allen Blockgruppen mit Kennzahl Potenz 3^n, 5^n und 7^n
- Aufteilung Speicherplatz in *Blockgruppen*
  - Informationen verwaltet in Gruppendeskriptor
- Freie Blöcke und freie Inodes verwaltet über Bitmaps
  - Data/Inode Bitmap: Gültig für *lokale* Blockgruppe
  - Adresse *aller* Bitmaps ermittelbar: Größe und Anzahl Blockgruppen bekannt.

Fragmentierung & E/A-Ablaufplanung

Fragmentierung

- Dateien wachsen/schrumpfen
- Konequenz: *Keine* kontinuierliche Verteilung auf Blöcke
- Lineares Durchlaufen einer Datei ☞ Nichtlineare Zugriffe auf Blockgerät
- Simultane Prozesse ☞ Verschlimmerung Problem

Optimierung

- I/O-Scheduling versucht, Anzahl Kopfbewegungen zu minimieren
  - Zusätzlich: Lese- gegenüber Schreibanforderungen priorisieren
- SSDs, RAM-Disks: Optimierung kontraproduktiv
  - Seek-Zeiten nicht vorhanden
  - Rechenaufwand bleibt

E/A-Ablaufplanung II

Beispiel: Zwei Prozesse

- A: Lesen von Blöcken in den Zylindern 98, 122, 124, 67
- B: Lesen von Blöcken in den Zylindern 183, 37, 122, 14, 65
- Festplattenkopf auf Position 53
- Anfragen in Reihenfolge 98, 183, 37, 122, 14, 124, 65, 67 eingetroffen
- Welche Reihenfolge kann das Betriebssystem wählen?
  - Seek-Zeit dominiert Latenz! ☞Minimierungskriterium
  - Fairness soll erhalten bleiben (keine »verhungernden« Anfragen)
  - Approximation: FCFS, SSTF, (Circular) SCAN

E/A-Ablaufplanung III

First Come, First Serve (FCFS)

- Anforderungen in Reihenfolge des Eintreffens bearbeitet
- Fairer Algorithmus, einfache Implementierung

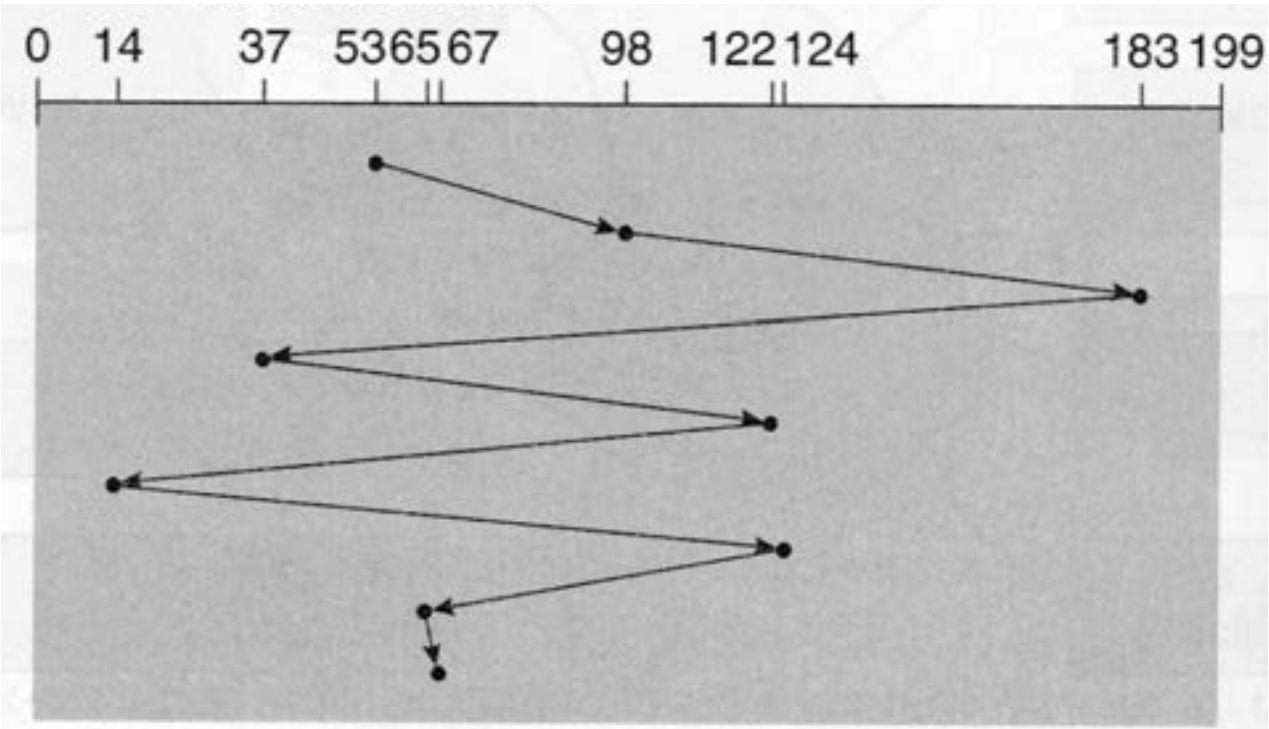


Abbildung: Silberschatz, Galvin & Gagne, *Modern Operating Systems*, Wiley

E/A-Ablaufplanung IV

Shortest Seek Time First (SSTF)

- Anforderung mit kürzester Kopf-Wegstrecke als nächstes bearbeitet
- Unfair: Anforderungen in der Mitte hungern Anforderungen an den Rändern aus

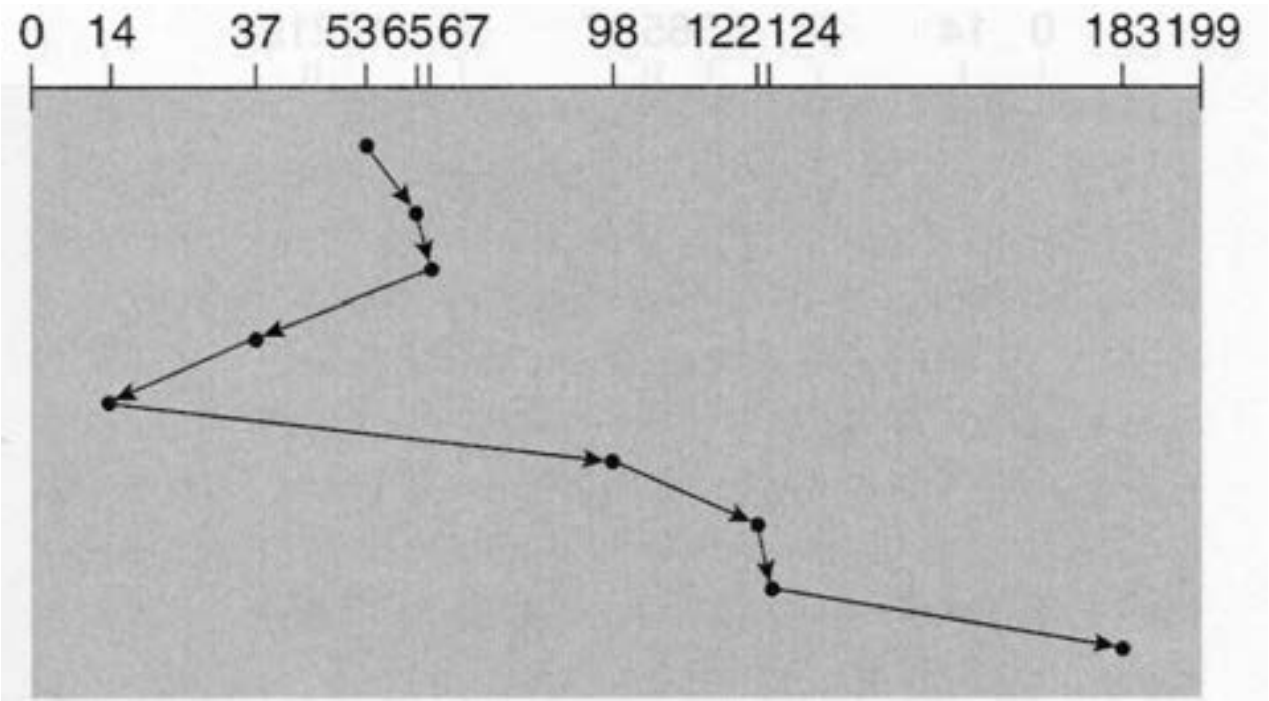


Abbildung: Silberschatz, Galvin & Gagne, *Modern Operating Systems*, Wiley

E/A-Ablaufplanung V

Scan (Elevator)

- Ähnlich wie Aufzug (*Elevator*):
  - Von innen nach außen bewegen und Anfragen bearbeiten
  - Richtung umdrehen, Anfragen bearbeiten, …
- Fairer Algorithmus (außer für neue Anfragen nahe Start)

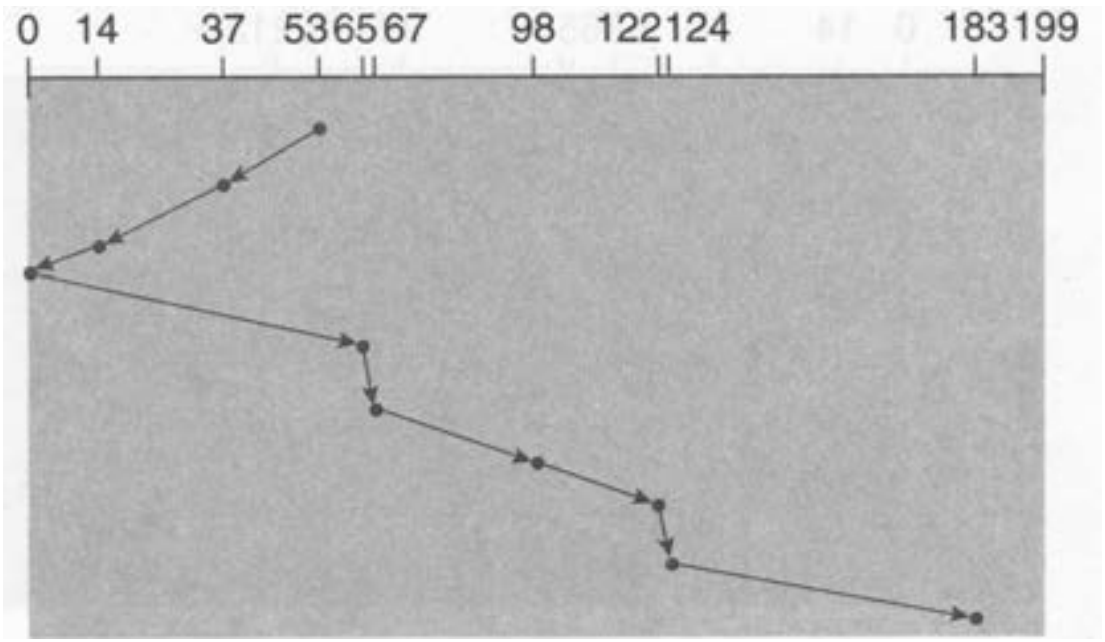
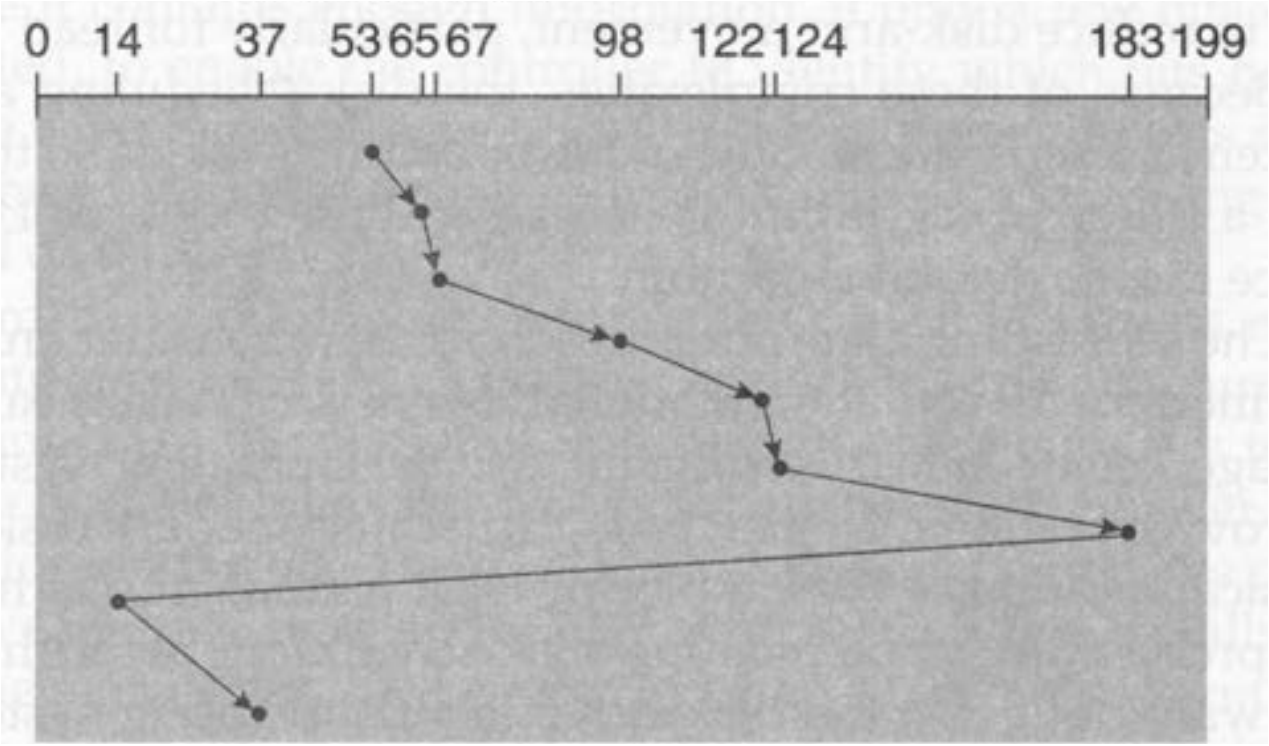


Abbildung: Silberschatz, Galvin & Gagne, *Modern Operating Systems*, Wiley

E/A-Ablaufplanung VI

Circular Scan (C-SCAN)

- Wie SCAN, aber *keine* Verarbeitung bei Rückwärtsbewegung
- Fairer Algorithmus
  - *Keine* Benachteiligung spät angekommener Anforderungen



E/A-Ablaufplanung VII

Praktische Probleme

- BS: Keine Detailkenntnis über Blockgeräteaufbau (LBA!)
- Zusätzlich: Anforderungen besitzen *Prioritäten*
- Blockgeräte: Eigenes Scheduling
- Priorisierung im Gerät: *Native Command Queueing* (NCQ)