

4.4 RISC & CISC

CISC

Complex Instruction Set Computers

RISC

Reduced Instruction Set Computers



Programmiermodell: Intel 80x86

Allgemeine Register

EAX	Arithm. Ergeb.	AX
EDX	& Ein/Ausgabe	DX
ECX	Zählregister	CX
EBX	Basis-Register	BX
EBP	Basis-Register	BP
ESI	Index-Register	SI
EDI	Index-Register	DI
ESP	Stack pointer	SP

Spezialregister

PC

Instruction pointer

EFR

...

sign

zero

aux carry

parity

carry

Maschinen-Status-Wort

Segmentregister

CS

Code-Segment

SS

Stack-Segment

DS

Daten-Segment

ES

Daten-Segment

FS

Daten-Segment

GS

Daten-Segment

In der Übung:
MIPS-Programmiermodell



Befehlsaufbau der Intel-x-86-Prozessoren



Befehlsaufbau der Intel-x-86-Prozessoren

- ❑ Ein Befehl besteht aus maximal 12 Bytes.
- ❑ Der Opcode belegt je nach Befehlstyp 1 oder 2 Bytes
 - Das WL-Bit (*Word Length*) bestimmt die Länge eines im Befehl „unmittelbar“ angegebenen Datum oder eines Offsets
- ❑ Die folgenden beiden Bytes dienen zur Auswahl der Adressierungsart und der dabei benutzten Register
- ❑ Falls ein Offset für die Adressberechnung erforderlich ist, wird dieser in den folgenden 1 bis 4 Bytes angegeben
- ❑ In den nächsten Bytes kann ein „unmittelbar adressiertes“ Datum mit einer Länge von 1, 2, 4 byte auftreten.

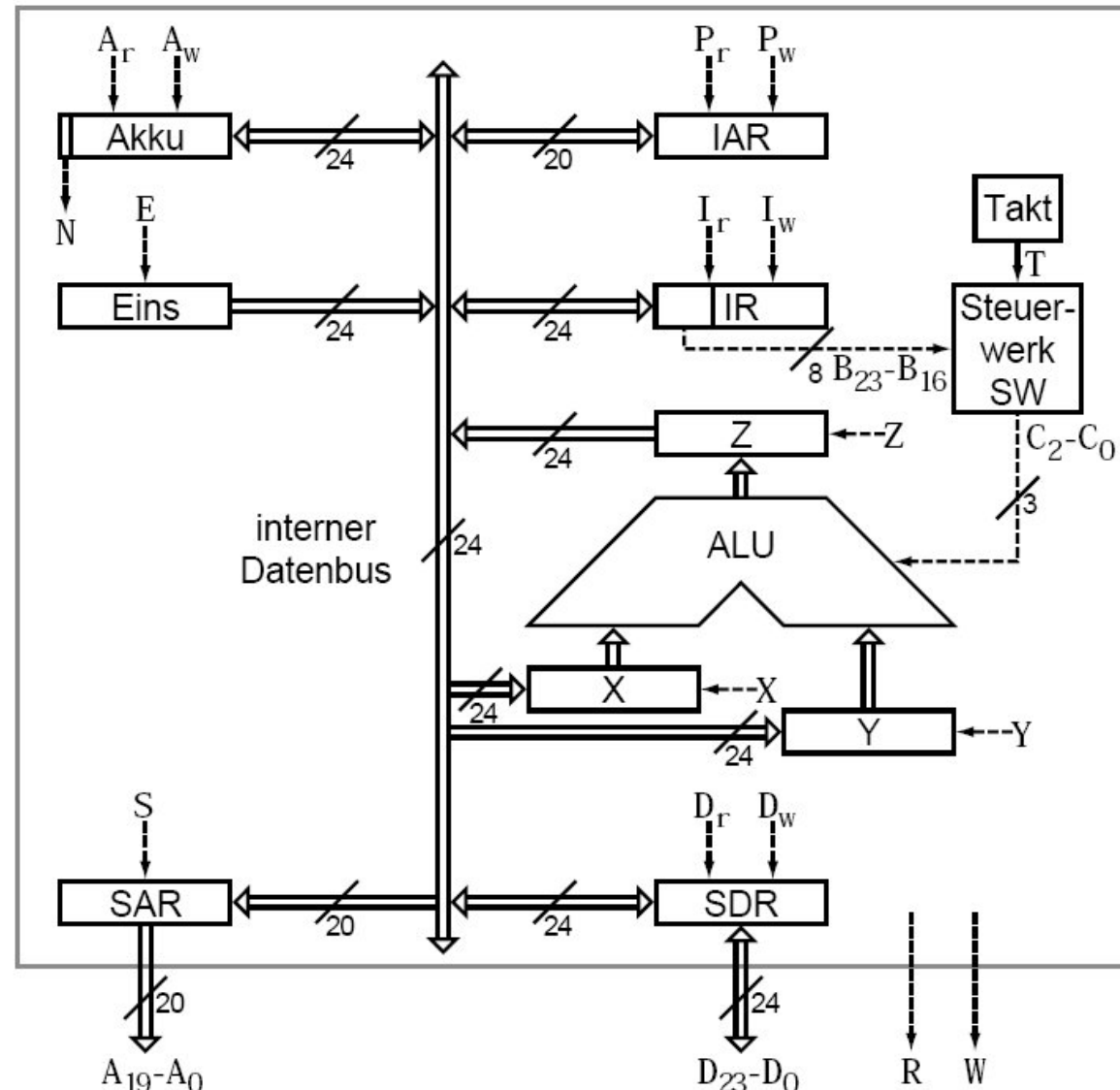
CISC & RISC

Zwei Techniken zur Implementierung von Befehlen im Rechner

- Direkt durch Hardware → *„fest verdrahtet“*
 - aufwendige Schaltnetze und Schaltwerke bei großer Anzahl von Befehlen (200-300)
- Mikroprogramme als Befehlsinterpreter
 - Mikroprogrammspeicher im Steuerwerk, der neu geladen werden kann
 - verschiedene Befehlssätze können implementiert werden

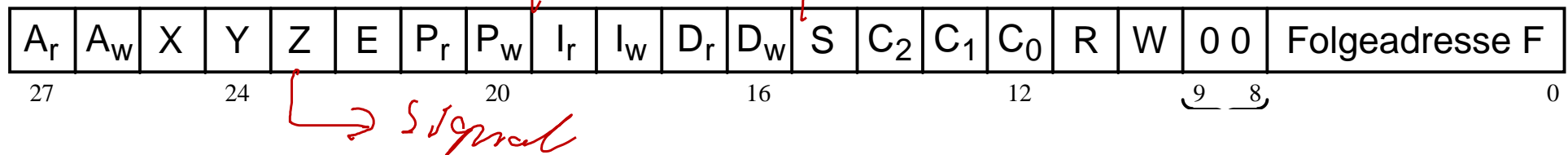
Mima-Architektur (Übungsblatt 2)

- Mikroprogrammierte Minimalmaschine (von-Neumann-Prinzip)
- SW mit 10 Meldesignale, 18 Steuersignale und Mikroprogrammspeicher für maximal 256 Mikrobefehle
- Befehlsabarbeitung:
 - Lese-Phase
 - Dekodierphase
 - Ausführungsphase
- 3 Taktzyklen für Lese- und Schreibzugriffe



Mikroprogrammsteuerwerke

Mikrobefehlsformat:



Jedes Bit des Mikrobefehls entspricht einem Steuersignal

- **Horizontale Mikroprogrammierung:**
Steuerungsfeld im Mikrobefehl für jedes Steuersignal im Rechner → Kein Dekoder
- **Vertikale Mikroprogrammierung:**
Zusammenfassung von Steuersignalen zu Feldern.

→ Decoder benötigt!

Mikroprogrammsteuerwerke

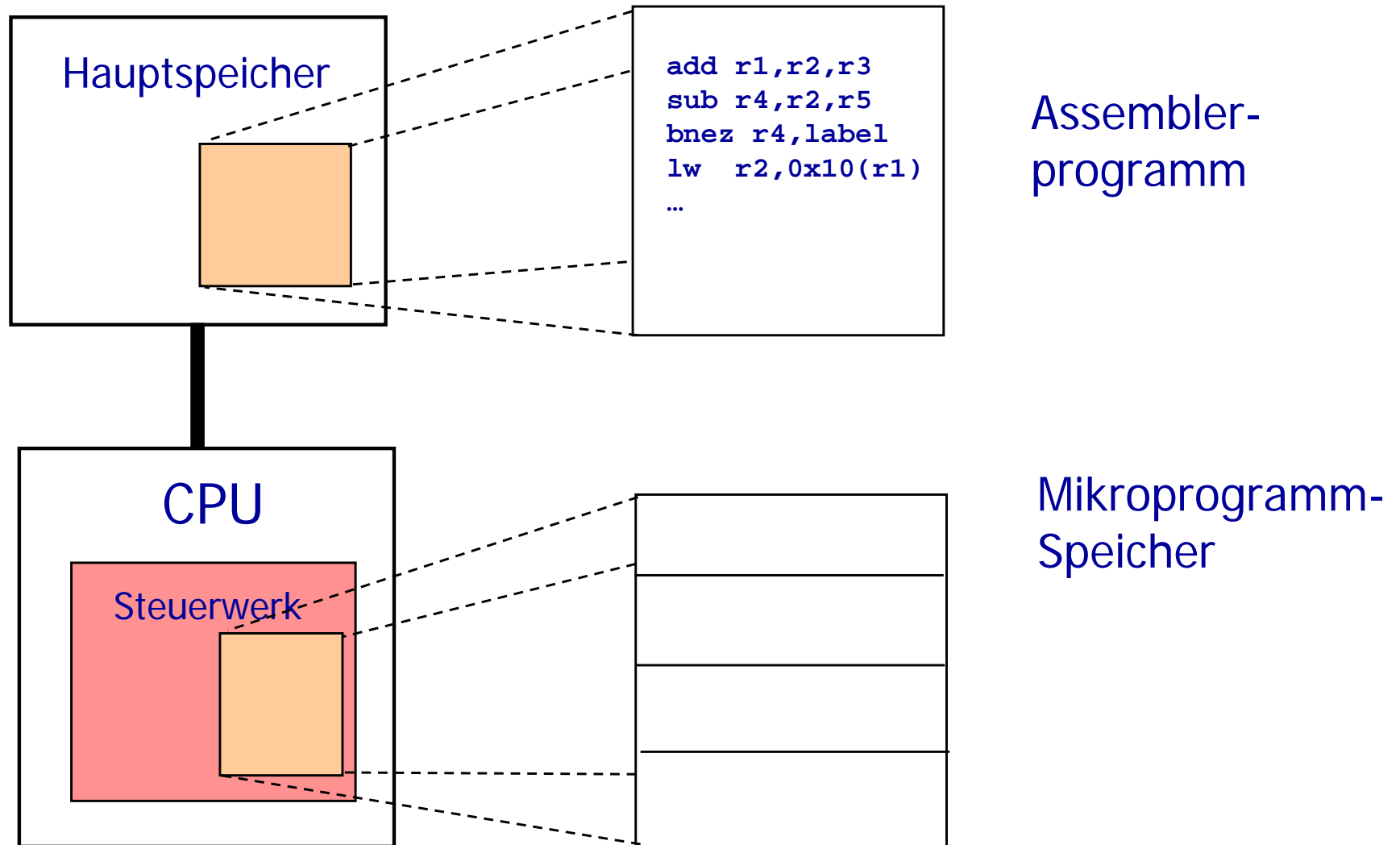
Fetch-Phase bei der MIMA:

1. Takt: IAR -> SAR; IAR -> X; R = 1
2. Takt: Eins -> Y; ALU auf addieren; R = 1
3. Takt: ALU auf addieren; R = 1
4. Takt: Z -> IAR
5. Takt: SDR -> IR

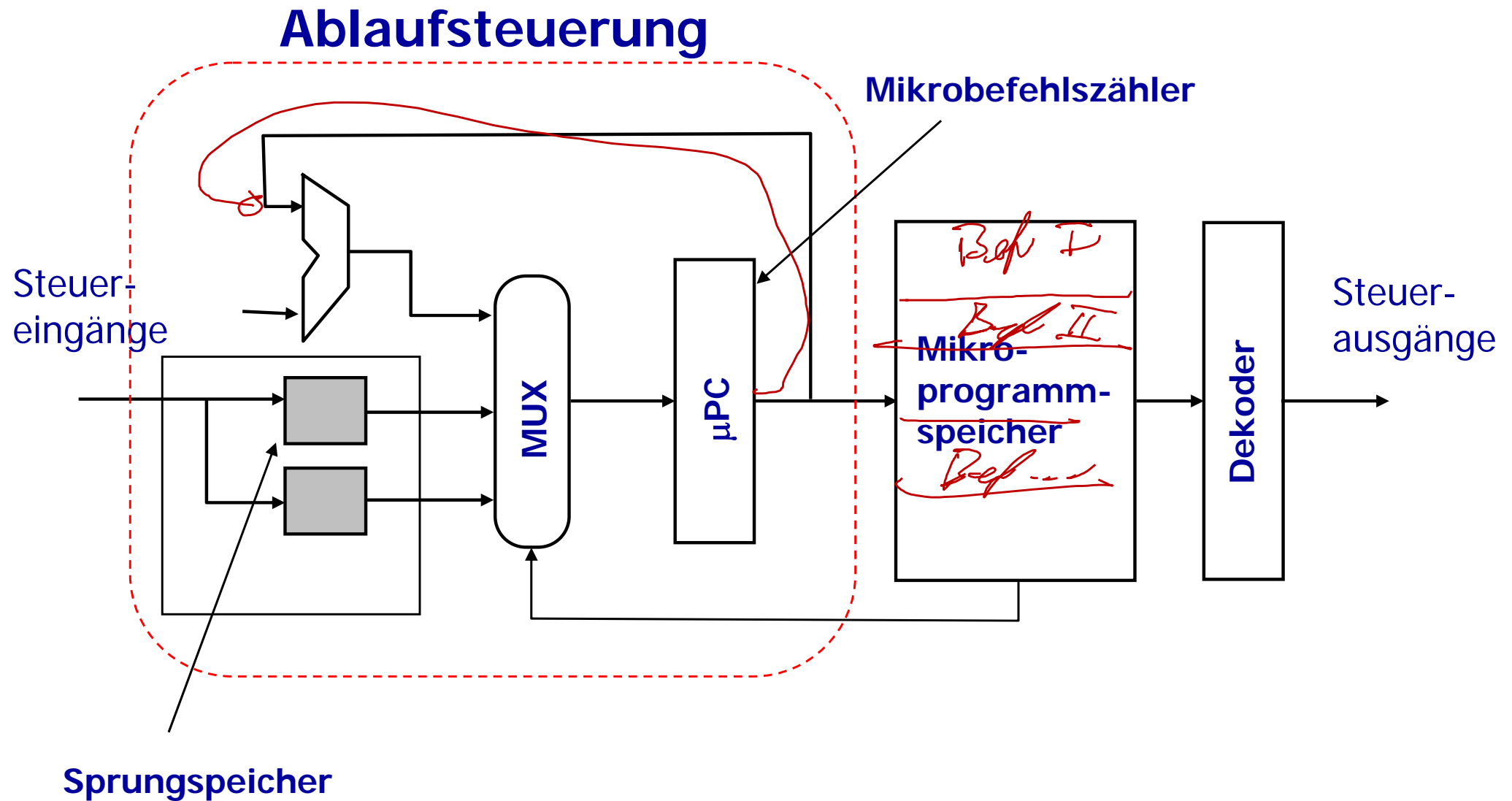
Mikroprogramm der Fetch Phase

0010	0001	0000	1000	1000	0000	0001
0001	0100	0000	0000	1000	0000	0010
0000	0000	0000	0001	1000	0000	0011
0000	1010	0000	0000	0000	0000	0100
0000	0000	1001	0000	0000	0000	0101

Prinzip der Mikroprogrammierung



Implementierung des Steuerwerks



Vorteile und Nachteile

➤ Vorteile der Mikroprogrammierung:

- Mehrere Befehlssätze auf einem Rechner ➔ Anpassung des Befehlssatz an der Anwendung
- Mehrere Rechnertypen mit dem gleichen Befehlssatz

flexibel

➤ Nachteile der Mikroprogrammierung:

- Aufwendig
- Langsam

CISC (complex instruction set computers)

Fuer CISC spricht:

- Ausführung komplexer Befehle ist schneller als die Ausführung von Programmen gleicher Funktion
- Mikroprogrammierung begünstigt komplexe Befehle
- komplexe Befehle führen zu kurzen Programmen
- Unterstützung höherer Programmiersprachen durch komplexe Befehle (Direkte Abbildung:
Sprachkonstrukt → *Befehl*)

CISC (complex instruction set computers)

Fazit:

Entwicklung von Hardware,
Programmiersprachen und Einsatzgebieten
begünstigt „komplexe“ Befehle.

CISC (complex instruction set computers)

Gründe dagegen:

- Schnellere Hauptspeicher und die Verwendung von Cache-Speichern beschleunigen die Programmausführung
- Mikroprogramme wurden immer umfangreicher; Verlängerte Entwurfszeit, Komplexe Steuerwerke (> 50% der Chipfläche)
- Nur relativ kleine Teile des großen Befehlssatzes werden häufig benutzt
- Größere Fehlerhäufigkeit auf der Mikroprogrammenebene
- Schwieriger Compilerbau

CISC (complex instruction set computers)

- **Systemprogramme in XPL auf IBM/360:**

90 % aller ausgeführten Befehle: 10 verschiedene Befehle

95 % aller ausgeführten Befehle: 21 verschiedene Befehle

99 % aller ausgeführten Befehle: 30 verschiedene Befehle

- **COBOL-Programme auf IBM/370:**

90,28 % aller ausgeführten Befehle: 26 verschiedene Befehle

99,08 % aller ausgeführten Befehle: 48 verschiedene Befehle

(nur 84 verschiedene Befehle wurden überhaupt benutzt)

Limitationen der CISC Architekturen

- **Befehlsausnutzung (80/20 Regel):**

viele mächtige Befehle, komplexes Befehlsformat, Mikroprogrammierung, nur 20 % der Befehle werden überwiegend benutzt

- **Kritisches Problem: Anzahl der Zyklen pro Instruktion (CPI)**

bei den meisten heutigen CISC Architekturen ist $CPI > 2$

Prozentualer Anteil von Anweisungen in Hochsprachenprogrammen

Großteil der in Hochsprachen verwendeten Anweisungen ist sehr einfach:

Anweisung	Mittlerer zeitlicher Anteil
Zuweisung	47 %
if	23%
call	15 %
loop	6 %
goto	3 %
Andere	7 %

RISC (reduced instruction set computers)

Grundprinzipien:

- Viel benutzte einfache Befehle so schnell wie möglich machen (Ausführung möglichst in einer Taktphase. Keine Mikroprogrammierung mehr, Befehls-Pipeline)
- Der größte Teil der Arbeit soll durch optimierende Compiler zur Übersetzungszeit erledigt werden
- Operanden werden nach Möglichkeit in großen Registersätzen gehalten → schneller Zugriff → schnelle Verarbeitung
- Einheitliche Befehlsformate → schnelle Decodierung
- Pipelining anwenden, so gut es geht

RISC (reduced instruction set computers)

Entwurfsziele:

- Ausführung jedes Befehls in einem Taktzyklus
(Befehl \approx bisheriger Mikrobefehl bei CISC)
- Alle Befehle gleich lang:
Decodierschaltung wird einfacher
Programme länger, aber Ausführungszeit kürzer
- Nur Load-Store und Register-Register-Befehle:
weniger Adressierungsarten \rightarrow schnelle Ausführung
- Koprozessorarchitektur für komplexe Befehle

separiert

RISC (reduced instruction set computers)

Keine Entwurfsziele sind z. B.:

- Unterstützung von Gleitkomma-Arithmetik
- Unterstützung von Betriebssystemfunktionen

Zielvorstellungen für RISC-Rechner

- Ein-Zyklus-Befehle
- Einheitliches Befehlsformat
- Wenige Maschinenbefehle
- Load/Store-Architektur
- Großer Registersatz
- Verzicht auf Mikroprogrammierung
- 32-Bit-Architektur
- Pipeline-gerechter Maschinenbefehlssatz (gleiche Befehlsausführzeiten)
- Keine Unterstützung für Betriebssystem und Gleitkomma-Arithmetik

Forderungen an RISC-Systeme

- ~~Mindestens 75% aller Befehle sind Ein-Zyklus-Befehle~~
- Einheitliche Länge aller Befehle entsprechend der Datenbusbreite
- Nicht mehr als 128 Befehle
- Nicht mehr als 4 Adressierungsarten
- Load/Store-Architektur
- Festverdrahtete Steuereinheit, keine Mikroprogrammierung
- Mindestens 32 allgemein verwendbare Register

RISC-Rechner aus heutiger Sicht

Geblieden ist von der RISC-Idee im wesentlichen:

- das Befehlspipelining
- die Load/Store-Architektur
- ein großer Registersatz: 32 allgemeine und 32 Gleitpunkt-Register
- ein einheitliches Befehlsformat
- die Verwendung weniger Adressierungsarten
- der Verzicht auf Mikroprogrammierung

RISC & CISC

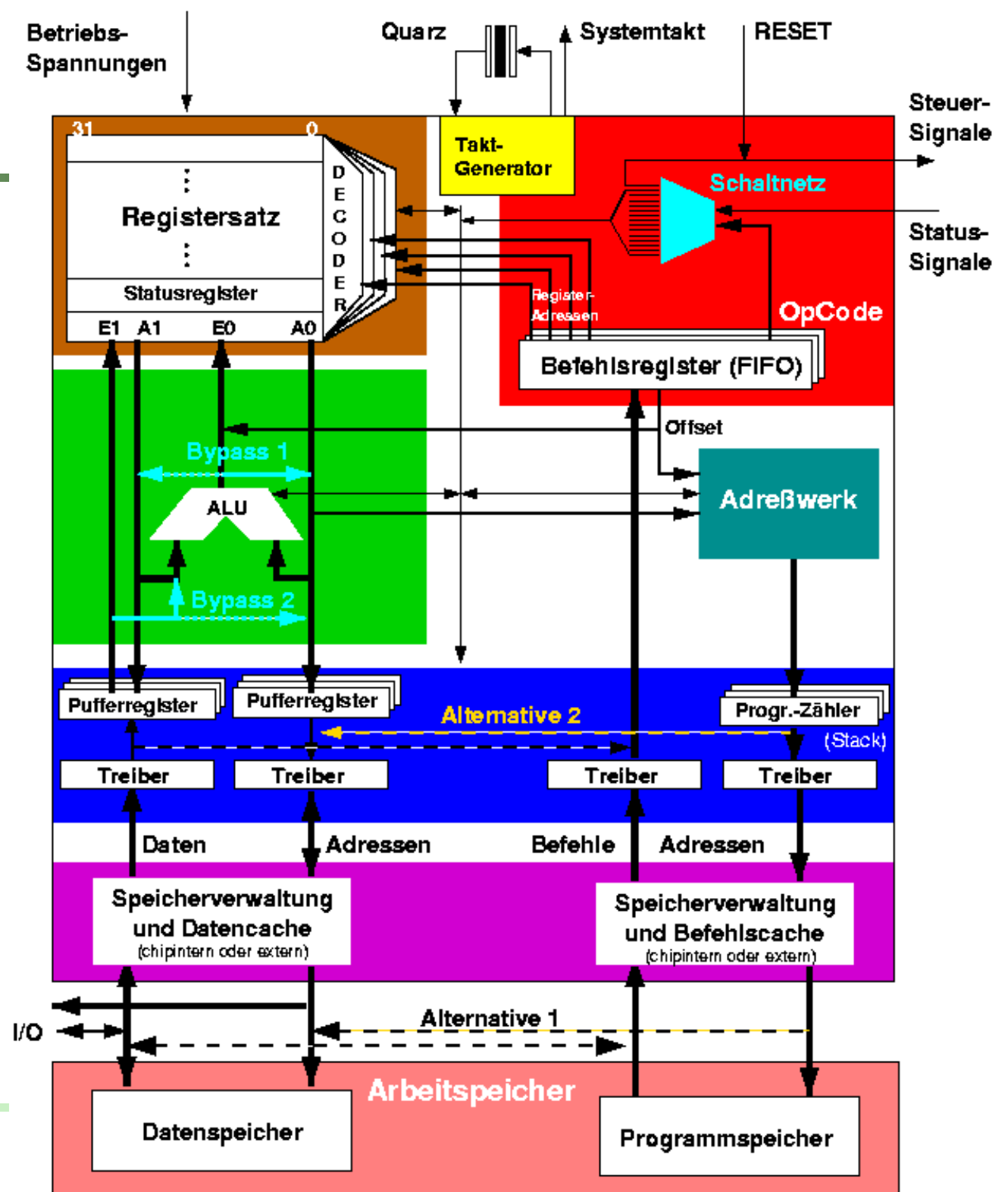
CISC	RISC
Komplexe Befehle, Ausführung in mehreren Taktzyklen	Einfache Befehle, Ausführung in einem Taktzyklen
Jeder Befehl kann auf den Speicher zugreifen	Nur Lade- und Speicherbefehle greifen auf den Speicher zu
Wenig Pipelining	Intensives Pipelining ^{hand-wired}
Befehle werden von einem Mikroprogramm interpretiert	Befehle werden durch festverdrahtete Hardware ausgeführt
Befehlsformat variabler Länge	Alle Befehle mit fester Länge
Die Komplexität liegt im Mikroprogramm	Die Komplexität liegt im Compiler
Einfacher Registersatz	Mehrere Registersätze

Vergleich CISC & RISC

Vergleich von drei typischen CISC-Rechnern mit den ersten drei RISC-Rechnern [Tanenbaum]:

	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC I	Stanford MIPS
Fertigstellungsjahr	1973	1978	1978	1980	1981	1983
Instruktionen	208	303	270	120	31	55
Mikrocodegröße	54k	61k	17k	0	0	0
Instruktionsgröße	2-6 Bytes	2-57 Bytes	1-3 Bytes	4 Bytes	4 Bytes	4 Bytes
Operationsmodell	Reg-Reg Reg-Mem Mem-Mem	Reg-Reg Reg-Mem Mem-Mem	Stack	Reg-Reg	Reg-Reg	Reg-Reg

Aufbau eines RISC-Prozessors



Aufbau eines RISC-Prozessors

Havard Architektur:

getrennter Programm- und Datenspeicher, deshalb
zwei Adress- und Datenbusse

→ paralleles Holen von Operanden und Instruktionen

Vereinfachende Varianten:

1. zwei getrennte Bussysteme bis zu den Cache-Speichern, jedoch nur ein Arbeitsspeicher (niedrigere Kosten)
2. nur ein Bussystem wie bei Standard-Mikroprozessoren

Aufbau eines RISC-Prozessors

Systembusschnittstelle:

enthält Registerblocks sowohl für Daten als auch für Adressen (gleichzeitiges Lesen eines Datums und Zwischenspeichern eines Ergebnisses)

Befehlszähler:

ist manchmal als Hardware-Stack ausgebildet (beschleunigt Unterprogrammaufrufe)

Aufbau eines RISC-Prozessors

Steuerwerk:

- festverdrahtet
- Das Befehlsregister als Warteschlange (FIFO) realisiert
- Für jede Pipeline-Stufe ist dort ein Register vorhanden
- Die OpCodes jeder Stufe können vom Schaltnetz des Steuerwerks ausgewertet werden

Registersatz:

- besteht aus einer großen Zahl von Registern
- erlaubt gleichzeitige Auswahl von 3 bis 4 Registern
(z. B. 4 Port Registersatz, gleichzeitiges Schreiben (E0, E1) und Lesen (A0, A1) von jeweils 2 Registern)

Aufbau eines RISC-Prozessors

Rechenwerk:

- Besitzt eine Load/Store-Architektur.
- Die Operanden werden über 2 Operandenbusse aus dem Registersatz herbeigeführt, das Ergebnis (noch im selben Taktzyklus) über den Ergebnisbus in den Registersatz geschrieben.
- Normalerweise gibt es keine direkte Verbindung zwischen ALU und Systemdatenbus
- Datentransfer läuft über die Register (Load/Store-Architektur)

Befehlsverarbeitung in RISC-Prozessoren

Einfacher Befehlssatz von RISC-Prozessoren

→ Maschinenprogramme sind länger als bei CISC Prozessoren

(komplexe Befehle und Adressierungsarten müssen aus den einfachen RISC-Befehlen zusammengesetzt werden)

Trotzdem arbeitet ein RISC-Prozessor meist schneller als ein CISC-Prozessor. Der Grund liegt in der nahezu **vollständigen Parallelarbeit aller Komponenten** eines RISC-Prozessors (extreme Pipeline-Verarbeitung)

Es wird mit großer Wahrscheinlichkeit **in jedem Taktzyklus ein Befehl** beendet

RISC - superskalar

- ❑ RISC-Prozessoren, die das Entwurfsziel von durchschnittlich einer Befehlsausführung pro Takt (CPI – *cycles per instruction* oder IPC – *instructions per cycle* von eins) erreichen, werden als **skalare RISC-Prozessoren** bezeichnet.
- ❑ Die Superskalar-Technik ermöglicht es, pro Takt mehrere Befehle den Ausführungseinheiten zuzuordnen und eine gleiche Anzahl von Befehlsausführungen pro Takt zu beenden.
- ❑ Solche Prozessoren werden als **superskalare (RISC)-Prozessoren** bezeichnet, da die oben definierten RISC-Charakteristika auch heute noch weitgehend beibehalten werden.
- ❑ Heutige Mikroprozessoren nutzen Befehlsebenenparallelität durch die **Pipelining-** und **Superskalartechnik**.

Kapitel 5

Pipeline-Verarbeitung



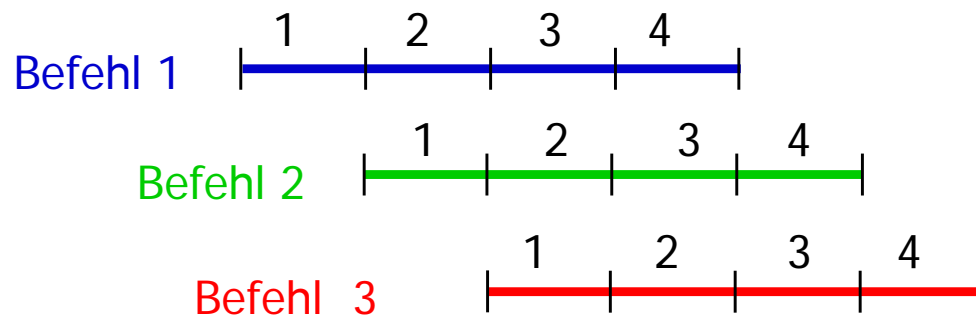
5. 1 Pipeline-Verarbeitung

Ausführung von 3 gleichartigen Verarbeitungsaufträgen in 4 Teilverarbeitungsschritten:

Serielle Verarbeitung:



Pipeline-Verarbeitung:



Pipelining „Fließband-Bearbeitung“

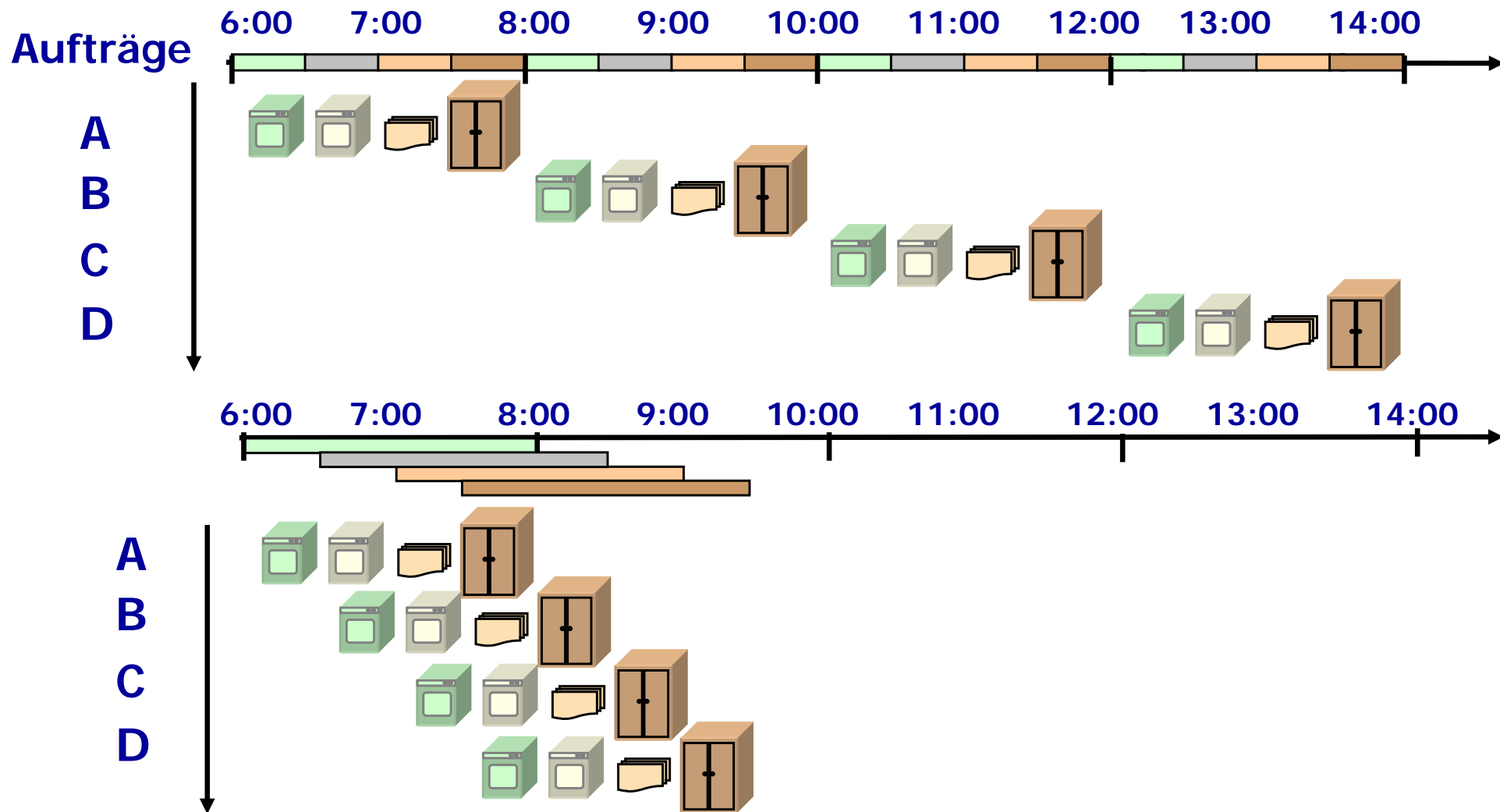
„Pipelines beschleunigen die Ausführungsgeschwindigkeit eines Rechners in gleicher Weise wie Henry Ford die Autoproduktion mit der Einführung des Fließbandes revolutionierte.“

(Peter Wayner 1992)

Wäsche-Pipelining

- Ein Wäsche-Vorgang kann in 4 Teilvorgänge unterteilt werden:
 - Schmutzige Wäsche in die Waschmaschine
 - Nasse Wäsche in den Trockner
 - Falten, Bügeln, ...
 - Kleider in den Schrank

Wäsche-Pipelining



Pipeline-Verarbeitung

Oft ablaufende Operationen werden in eine Folge von Teilprozessen zerlegt. Für jeden Teilprozess wird ein spezieller Prozessor (spezielle Ausführungseinheit) vorgesehen.

Beispiel: Befehlsverarbeitung wird aufgeteilt in:

- Befehl holen
- Befehl decodieren (interpretieren) und
- Operand(en) holen
- Operation ausführen
- Ergebnis speichern

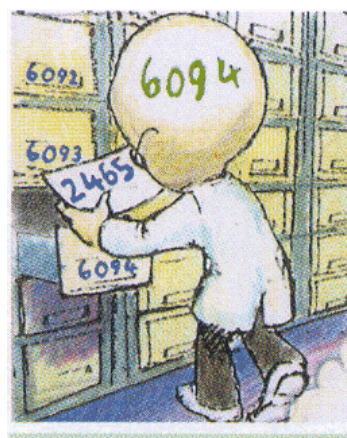
Beispiel



**Befehl
bereitstellen**



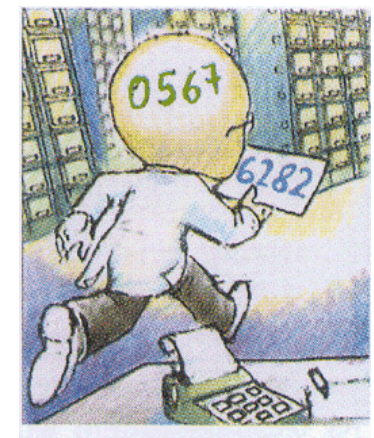
**Befehl
dekodieren**



**Operanden
holen**

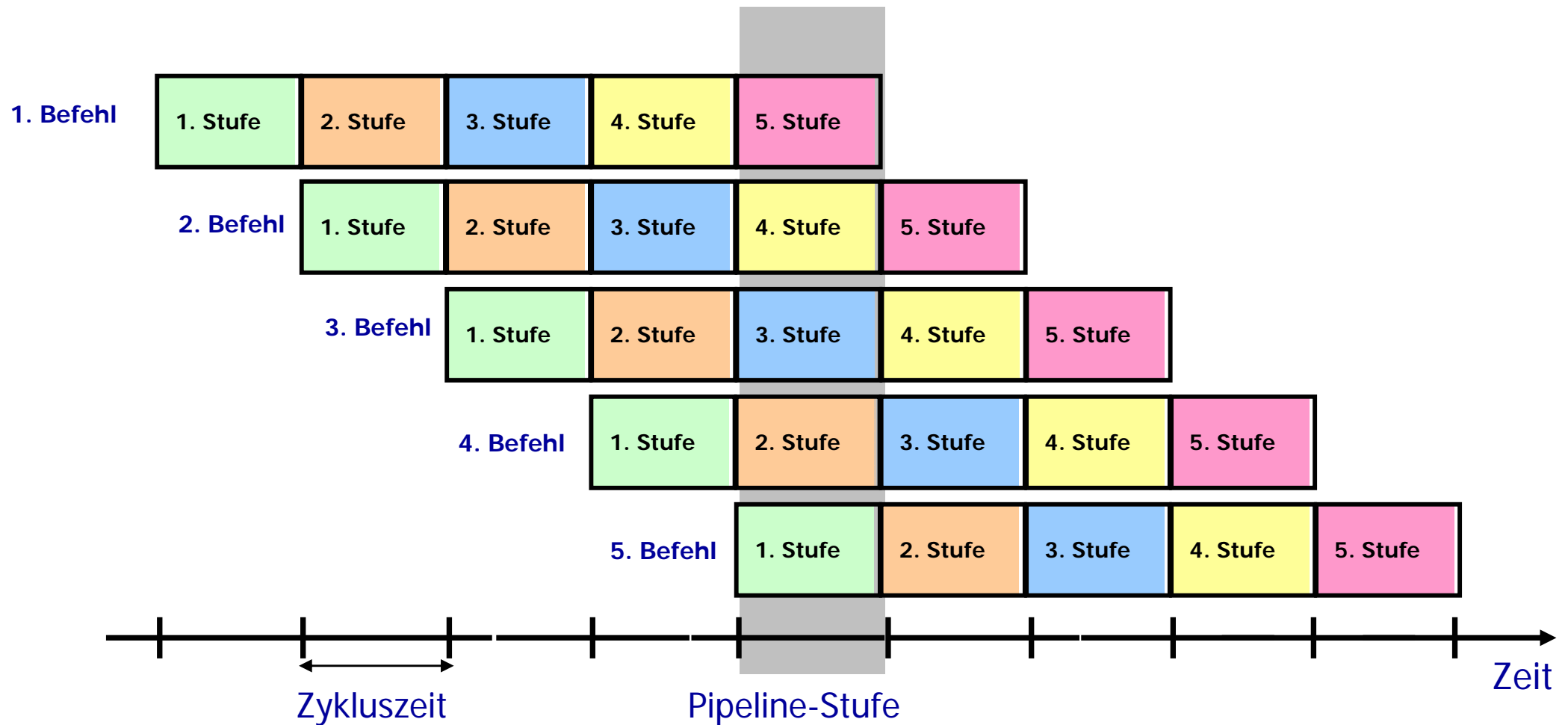


**Operation
ausführen**



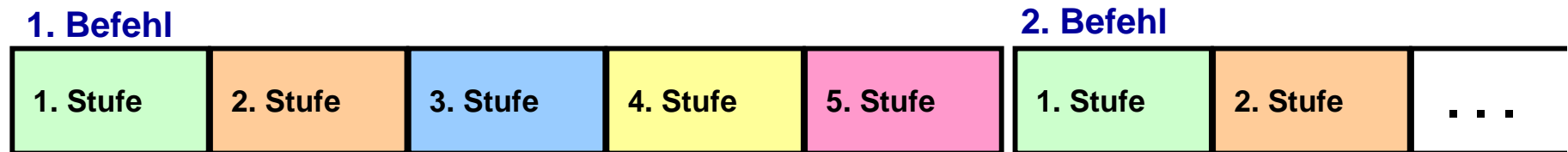
**Ergebnis
speichern**

Einfache fünfstufige Befehlspipeline



Pipelining

Sequentielle Ausführung:



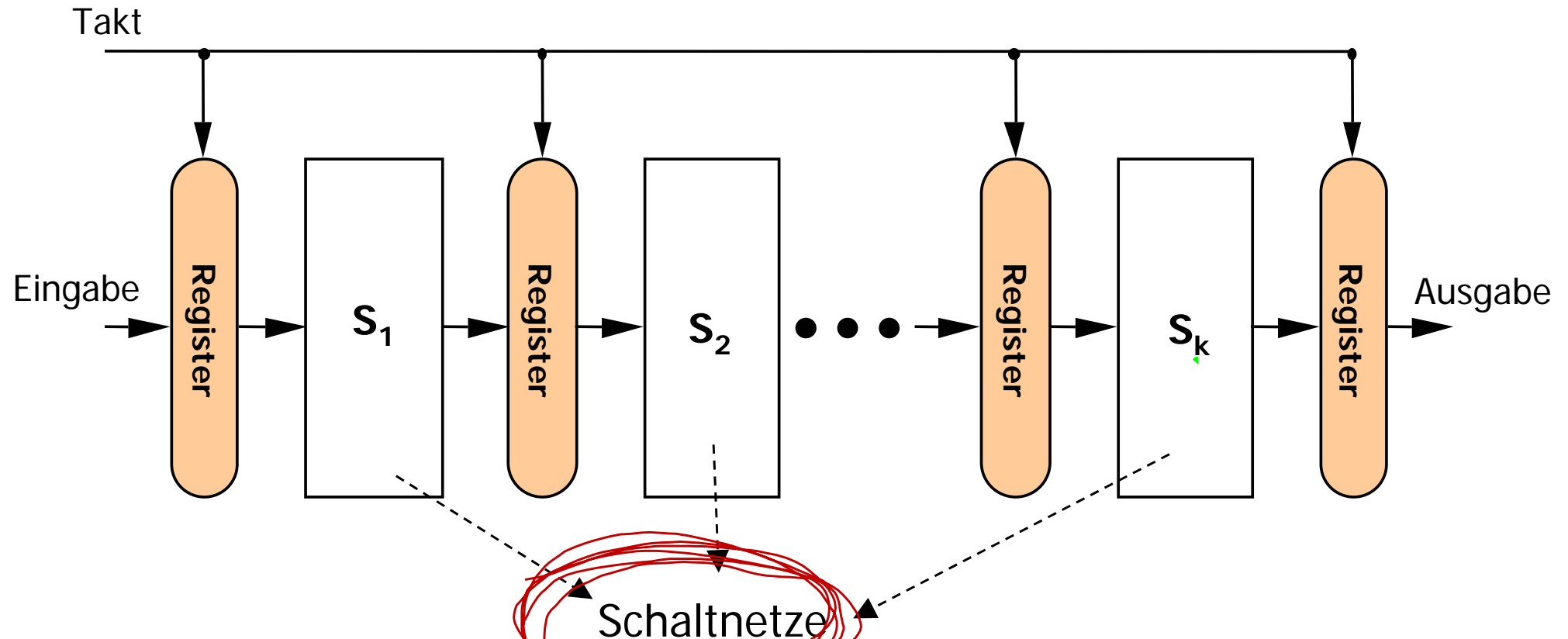
Pipelining:



Definitionen

- ❑ **Pipelining:** Zerlegung einer Maschinenoperation in mehrere Phasen oder Suboperationen, die dann von hintereinander geschalteten Verarbeitungseinheiten taktsynchron bearbeitet werden, wobei jede Verarbeitungseinheit genau eine spezielle Teiloperation ausführt
- ❑ Die Gesamtheit dieser Verarbeitungseinheiten nennt man eine **Pipeline**.
- ❑ Bei einer **Befehlspipeline** (Instruction Pipeline) wird die Ausführung eines Maschinenbefehls in verschiedene Phasen unterteilt, aufeinanderfolgende Maschinenbefehle werden jeweils um einen Taktzyklus versetzt ausgeführt

5.2 Pipeline-Stufen und Pipeline-Register



Verzögerungszeiten:

- der Schaltnetze: τ_i ($i = 1, \dots, k$)
- der Pipeline-Register: τ_{reg}

Länge eines Taktzyklus:

$$\tau = \max\{\tau_1, \tau_2, \dots, \tau_k\} + \tau_{reg}$$

Definitionen

- ❑ Jede Stufe der Pipeline heißt **Pipeline-Stufe** oder **Pipeline-Segment**.
- ❑ Pipeline-Stufen werden durch getaktete **Pipeline-Register** (auch *latches* genannt) getrennt.
- ❑ Ein Pipeline-Maschinentakt ist die Zeit, die benötigt wird, um einen Befehl eine Stufe weiter durch die Pipeline zu schieben.
- ❑ Idealerweise wird ein Befehl in einer **k-stufigen Pipeline** in **k** Takten von **k** Stufen ausgeführt.
- ❑ Wird in jedem Takt ein neuer Befehl geladen, dann werden zu jedem Zeitpunkt unter idealen Bedingungen **k** Befehle gleichzeitig behandelt und jeder Befehl benötigt **k** Takte, bis zum Verlassen der Pipeline.

Definitionen

- **Latenz:** die Zeit, die ein Befehl benötigt, um alle k Pipeline-Stufen zu durchlaufen.

Ideale Verhältnisse:

- Ausführung eines Befehls in k Takten.
 - Es werden gleichzeitig k Befehle bearbeitet.
-
- **Durchsatz einer Pipeline:** Anzahl der Befehle, die eine Pipeline pro Takt verlassen können. Dieser Wert spiegelt die Rechenleistung einer Pipeline wider.

Leistungssteigerung durch Pipelining

n Befehle mit einer Pipeline von k Stufen ausführen

- Hypothetischen Prozessor ohne Pipeline:
 $n * k$ Taktzyklen
- Pipeline-Prozessor mit einer k -stufigen Pipeline:
 $k + (n-1)$ Taktzyklen (unter der Annahme idealer Bedingungen mit einer Latenz von k Takten und einem Durchsatz von 1)
 - **k** Taktzyklen, um die Pipeline zu füllen
 - **$(n-1)$** Taktzyklen, um die restlichen **$(n-1)$** Befehle auszuführen.

→ Leistungssteigerung **S** (*speedup*):

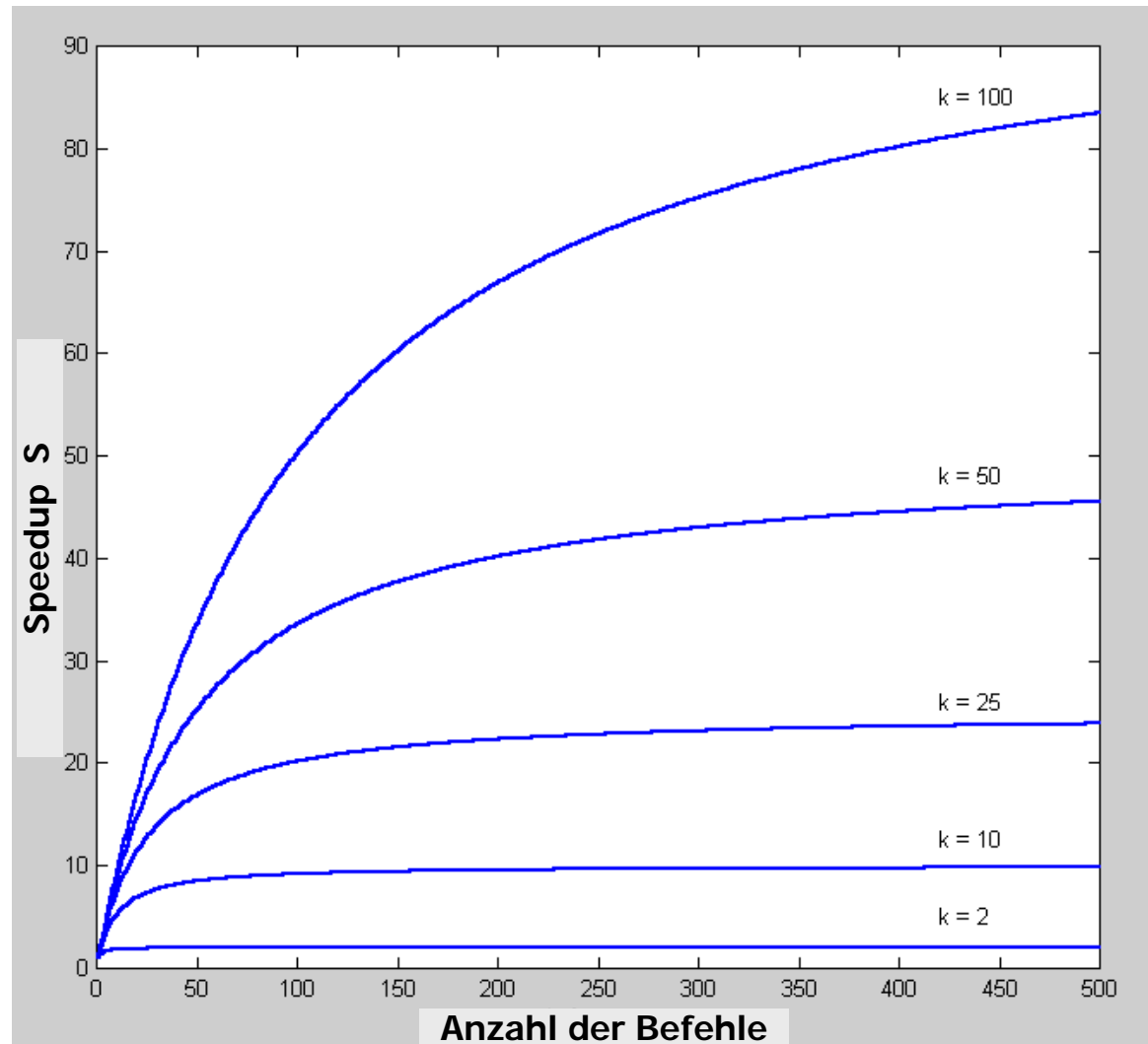
$$S = \frac{n * k}{k + (n-1)}$$

um früher die Pipeline

Leistungssteigerung durch Pipelining

$$S = \frac{n * k}{k + (n-1)}$$

$$\lim_{n \rightarrow \infty} S = k$$



Durchsatz

- Der **Durchsatz** gibt an, wie viele Befehle in einem Zeitraum $T_k * \tau$ ausgeführt werden.

$$D = \frac{n}{T_k * \tau}$$
$$= \frac{n}{(k + (n-1)) * \tau}$$

$$\lim_{n \rightarrow \infty} D = \frac{1}{\tau} = D_{max}$$

