

# Modul Informatik-II

## Kurs Informatik-3: Teil-2

[www.engineering.zhaw.ch/de/engineering/studium/bachelor/informatik/studium-zurich.html](http://www.engineering.zhaw.ch/de/engineering/studium/bachelor/informatik/studium-zurich.html)

**Prof. Dr. Olaf Stern**  
**Leiter Studiengang Informatik**  
**+41 58 934 82 51**  
[olaf.stern@zhaw.ch](mailto:olaf.stern@zhaw.ch)

# Lernziele: (Allgemein)

- Die Studierenden kennen die *grundlegende Architektur von Rechnern* und die wichtigsten *Architekturelemente*.
- Sie sind vertraut mit der *elementaren Arbeitsweise eines Computers* und der *hardwarenahen Programmierung*. Sie können diese an einfachen Beispiel erläutern.
- Die Studierenden kennen die grundsätzlichen Aufgaben eines *Betriebssystems*. Sie können die *typischen* Verfahren und *Algorithmen*, die bei der *Entwicklung* von *Betriebssystemen* zur Anwendung gelangen, beschreiben.

# Lernziele: (Allgemein)

- Die Kurse der Module Informatik I und Informatik II (der Modulgruppen "Grundlagen der Informatik I+II") vermitteln den Studierenden die *Grundlagen der Informatik, die jede / jeder Studierende unabhängig von der Wahl der Wahlpflichtmodule im Fachstudium erlangen sollte.*
- Die vermittelten Grundlagen *werden in den Modulen im Fachstudium vorausgesetzt.*

## Lernziele: Spezifisch Teil 2

- Die Studierenden kennen den grundsätzlichen Aufbau und die Arbeitsweise eines **Prozessors**.
- Sie können die Funktionsweise der wichtigsten **Bauelemente** eines **Prozessors** erläutern und den schematischen Ablauf einer **Befehlsabarbeitung** demonstrieren.
- Die Studierenden kennen die Bedeutung der **Zykluszeit** eines Prozessors und können **Methoden** zur Optimierung der **Zykluszeit** beschreiben; in diesem Zusammenhang sind sie insbesondere mit dem grundsätzlichen Ansatz des **Pipelining** vertraut und können diesen an Beispielen erklären.

# Themenüberblick:

## Technische Informatik / Rechnerarchitektur

- Einführung / Übersicht
- Grundlegende Rechnerarchitektur
- **Prozessoren**
  - Leistungsmessung
  - Schematischer Aufbau
  - Prinzipielle Arbeitsweise (vereinfacht)
  - Pipelining
- Befehle – die „Wörter“ des Rechners
- „Mini-Power-PC“
- Speicher
- „Mini-Power-PC“ (Fortsetzung)

# Lerninhalte Teil 2

## ■ Prozessor (CPU)

### – Leistungsparameter

➤ *Befehlszahl, Zyklus(zeit) und **CPI***

### – Aufbau und Implementierung (vereinfachte Darstellung)

- Umsetzung arithmetisch-logischer Funktionen
- Schaltnetze und -werke
- Lade- und Speicherbefehle
- Sprungbefehle

### – Zykluszeit

- Optimierungstechniken
  - 1-Takt System, Mehrtakt-System
  - Pipelining

## Prozessorleistung

- Die Leistung eines Rechners kann allgemein über folgende Parameter charakterisiert werden:
  - Befehlsanzahl / Befehlssatz
    - Je mehr **Befehle** ein **Prozessor umsetzen** kann, desto **kompakter** und **kürzer**, d. h. mit einer geringeren Anzahl von Befehlen, können **Programme realisiert** und **ausgeführt** werden.
- => Die **Anzahl** der **Befehle** für ein **konkretes Programm** hängt vom **Compiler** und dem **Befehlssatz** eines Prozessors ab („**Befehlsarchitektur**“).

## Prozessorleistung

- Die Leistung eines Rechners kann allgemein über folgende Parameter charakterisiert werden:
    - Befehlsanzahl / Befehlssatz
    - Taktzyklus(zeit)
      - **Zeitangabe**, typischerweise als **Frequenzangabe** angegeben (in MHz, GHz, ...); dabei entspricht z. B. **4 GHz** einem **Taktzyklus** von **0.25 ns**.
      - Je **kürzer** die **Taktzykluszeit** ist, desto **mehr Befehle** kann ein **Prozessor pro Zeit ausführen** (bei identischer **CPI-Zahl**!).
- => Hängt von der Implementierung des Prozessors ab.**



## Prozessorleistung

- Die Leistung eines Rechners kann allgemein über folgende Parameter charakterisiert werden:
    - Befehlsanzahl / Befehlssatz
    - Taktzyklus(zeit)
    - Taktzyklen pro Befehl (engl. „*clock cycles per instruction*“ – **CPI**)
      - Gibt die **durchschnittliche Anzahl** von **Taktzyklen** an, die ein **Befehl** zur **Ausführung** erfordert.
- => Hängt von der Implementierung des Prozessors ab.**

# Prozessor (CPU)

## Aufbau / Implementierung

- Wie bereits bei der Beschreibung eines *Von-Neumann-Rechners* angegeben, besteht ein Prozessor im Wesentlichen aus:

- dem *Befehlszähler*, dem *Befehlsregister*
- dem *Steuerwerk*
- weiteren *Registern* und
- der *ALU* (dem *Rechenwerk*)

inkl. der zugehörigen **Steuer-, Daten- und Adressleitungen** sowie **Datenlogik** und greift für die **Bearbeitung der Befehle** auf den **Speicher** zu (**Laden von Befehlen** und **Daten** sowie **Speichern von Daten**).

# Prozessor (CPU)

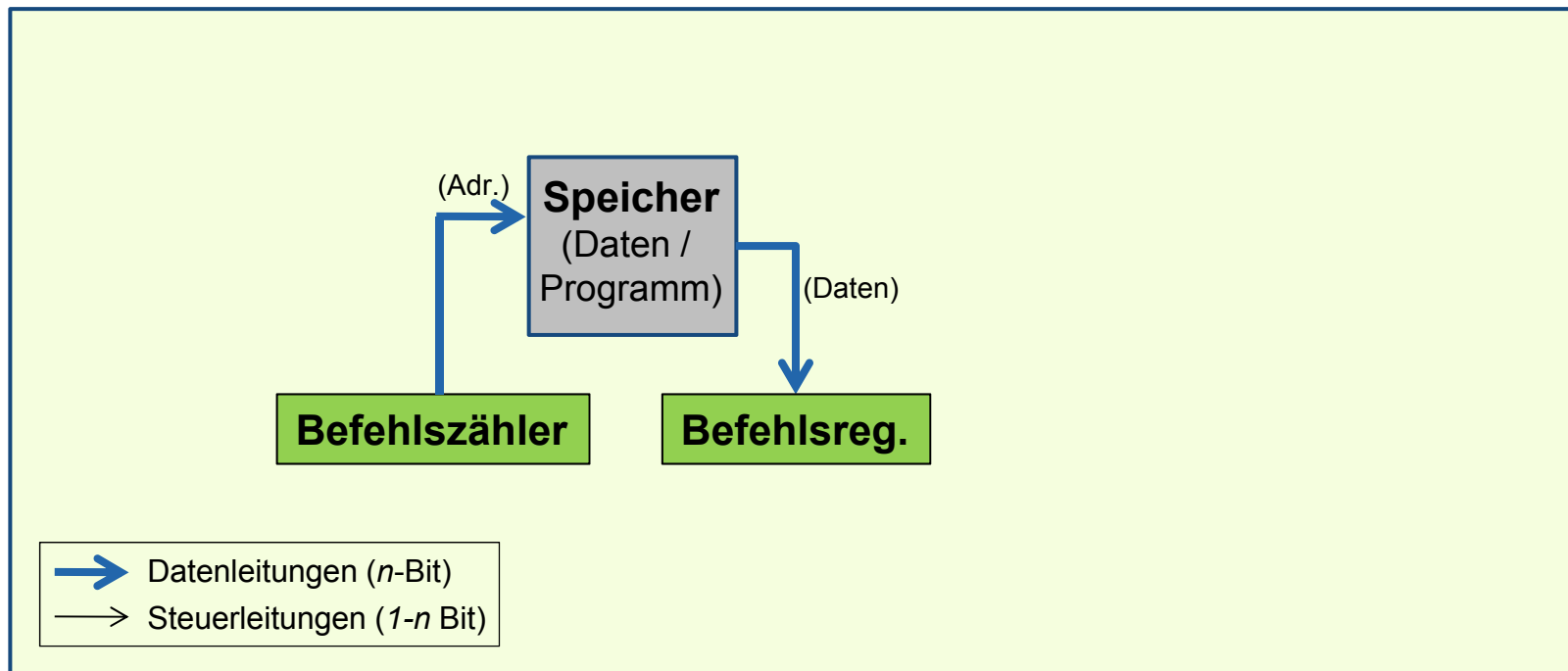
## Aufbau / Implementierung (Schema)

- Welche Bauelemente grundsätzlich erforderlich sind und wie diese miteinander kommunizieren, d. h. über Daten- und Steuerleitungen verbunden sind, lässt sich am besten an einer schematischen Skizze für die Bearbeitung eines Befehls skizzieren (→ folgt).
- Da mit einem Rechner „gerechnet“ werden soll, müssen Befehle **arithmetisch-logische** Funktionen ausführen  
(wie bereits erwähnt, existiert für diese Berechnung eine **Arithmetische-Logische Einheit**, kurz **ALU**)

! Aber wie wirken die einzelnen Komponenten bei der Abarbeitung eines Befehls zusammen?

## Aufbau / Implementierung (Schema – vereinfacht)

**Schritt 1:** Der Inhalt des *Befehlszählers* (Register) adressiert eine *Speicherzelle*, dessen Inhalt in das *Befehlsregister* übertragen wird



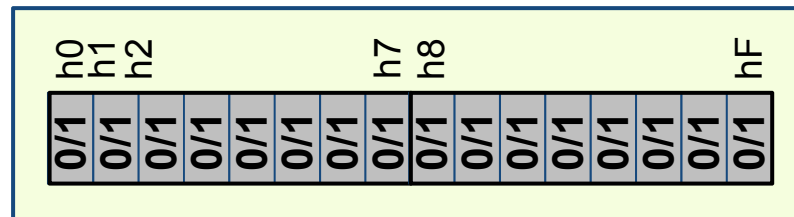
## Aufbau / Implementierung

- Welche Art von Funktionen werden damit abgedeckt?
  - => Alle *arithmetisch-logischen Funktionen*, welche die **ALU** (das *Rechenwerk*) berechnen kann
    - **Addition**, (Inkrement), (Subtraktion), Multiplikation,
    - **1- und 2-stellige boolesche Funktionen** (wort- oder bitweise), wie z. B. **UND**, **ODER**, **EXOR**

**Anmerkung:** Komplexe Funktionen, wie z. B. *Fliesskomma-Operationen* oder die *Division*, werden heute von speziellen **optimierten Rechenwerken** verarbeitet! (früher auch in Software - „*Mikroprogramme*“)

## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?
  - *Register: Bsp. für 16-Bit-Register*



- Auf jedes einzelne Bit kann lesend und schreibend direkt zugegriffen werden.

**Hinweis:** Wie die einzelnen Bauelemente (*Register*, *ALU*, *Addierer*, *Multiplexer* usw.) konkret implementiert werden, ist u. a. Inhalt des Kurses Elektronik (im 4. Semester)

## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?
  - „**Schaltnetz**“ („*Kombinatorische Elemente*“)
    - Ein **Schaltnetz** besteht ausschliesslich aus **logischen Bauelementen** und **keinen Speicherbausteinen**.  
(vgl. *Boolesche Funktionen*, → Kurs Informatik-1)
    - Der/die **Ausgangswert(e)** ergeben sich auf Grund der **Werte** an den **Eingängen**.
    - „**Bausteine, die Datenwerte verarbeiten, sind ein Schaltnetz**“.  
[Patterson, Hennessy]
    - Eine **Kombination** von **Schaltnetzen** ist wieder ein **Schaltnetz**.

# Prozessor (CPU)

## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?

- „**Schaltnetz**“: Beispiel **3-Bit-Multiplexer**

- Von den **8 Eingängen** ( $2^3$ ) wird mit Hilfe der **drei Steuereingänge**  $s_0$ ,  $s_1$  und  $s_2$  **genau einer** selektiert und auf den **Ausgang a** durchgeschaltet.
    - **Allgemein:**  $n$  Steuereingänge,  $2^n$  Eingänge, 1 Ausgang

**Hinweis:** Wie die einzelnen Bauelemente (**Register**, **ALU**, **Addierer**, **Multiplexer** usw.) konkret implementiert werden, ist u. a. Inhalt des Kurses Elektronik (im 4. Semester).



## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?
  - „**Schaltnetz**“: Beispiel **3-Bit-Demultiplexer**
    - Ein **Eingang e** wird mit Hilfe der **Steuereingänge**  $s_0$ ,  $s_1$  und  $s_2$  auf **genau einen Ausgang der 8 Ausgänge** ( $2^3$ ) durchgeschaltet.
    - **Allgemein:**  $n$  Steuereingänge,  
1 Eingang,  $2^n$  Ausgänge

**Hinweis:** Wie die einzelnen Bauelemente (**Register**, **ALU**, **Addierer**, **Multi-plexer** usw.) konkret implementiert werden, ist u. a. Inhalt des Kurses Elektronik (im 4. Semester).

# Prozessor (CPU)

## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?
  - „**Schaltnetz**“: Bsp. **Addierer** (Funkt.: Addition von Zahlen)
    - Wie andere **logische Schaltungen** kann ein **Addierer** über **boolesche Schaltungen** realisiert werden.
      - => In der Praxis sind „**Addierwerke**“ aber spezielle (bzgl. Geschwindigkeit) **optimierte Schaltungen**.
    - Vielzahl von (**optimierten**) Typen: **Halb-** und **Volladdierer**, **Carry-Ripple**, **Carry-Skip**, **Carry-Look-Ahead**, **Conditional Sum Addition**, **Carry-Select**,

## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?
  - „**Schaltwerk**“ („*Sequentielle Elemente*“)
    - **Schaltwerke** können *intern Datenwerte speichern*.
    - Ein **Schaltwerk** hat mindestens einen Dateneingang, einen Takteingang und einen Datenausgang.
    - Der Takteingang bestimmt, wann ein an einem **Speicherelement** anliegender Wert übernommen wird, d. h. geschrieben wird; gelesen werden können die Werte jederzeit.

## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?

- „**Schaltwerk**“



- Es wird prinzipiell zwischen **pegel-** und **flankengesteuerten** (engl. „**edge triggered**“) **Schaltwerken** unterschieden.

**=> Heute werden überwiegend flankengesteuerte Schaltwerke genutzt**

(Hinweis: „*Hazard*“, „*Glitch*“, „*race-condition*“)

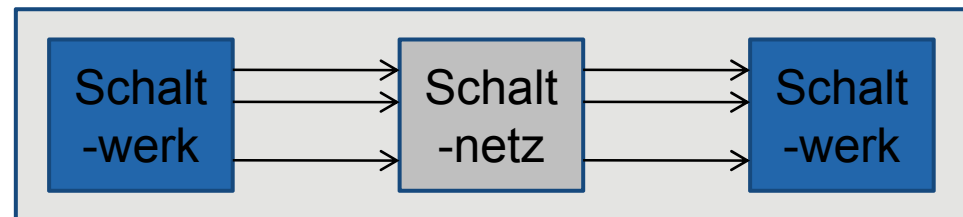
## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?

— „*Schaltwerk*“

=> Da nur *Schaltwerke* Daten speichern, müssen alle Eingaben von *Schaltnetzen* aus *Schaltwerken* stammen und wieder in *Schaltwerken* abgespeichert werden.

Schema: (allgemein)



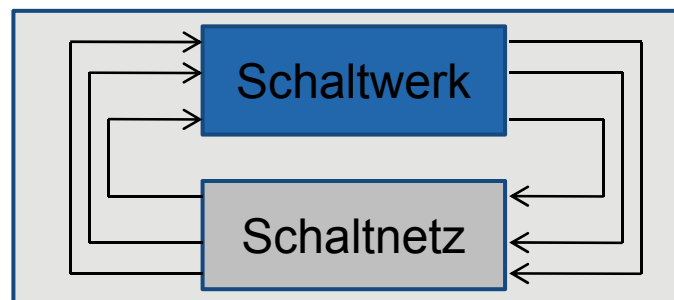
## Aufbau / Implementierung

- Welche Bauelemente sind dafür erforderlich?

- „**Schaltwerk**“

- Werden **flankengesteuerte Speicherelemente** in den **Schaltwerken** verwendet, können die **Daten** aus einem **Schaltwerk** **ausgelesen** werden und in dasselbe **Schaltwerk** wieder **abgespeichert** werden.

**Schema:** (unter Nutzung von *flankengesteuerten* Speichern)



## Aufbau / Implementierung

- Welche Art von Funktionen werden damit abgedeckt?
  - => Alle *arithmetisch-logischen Funktionen*, welche die *ALU* (das *Rechenwerk*) berechnen kann.
    - **Addition**, (Inkrement), (Subtraktion), Multiplikation,
    - **1- und 2-stellige boolesche Funktionen** (wort- oder bitweise), wie z. B. *UND*, *ODER*, *EXOR*

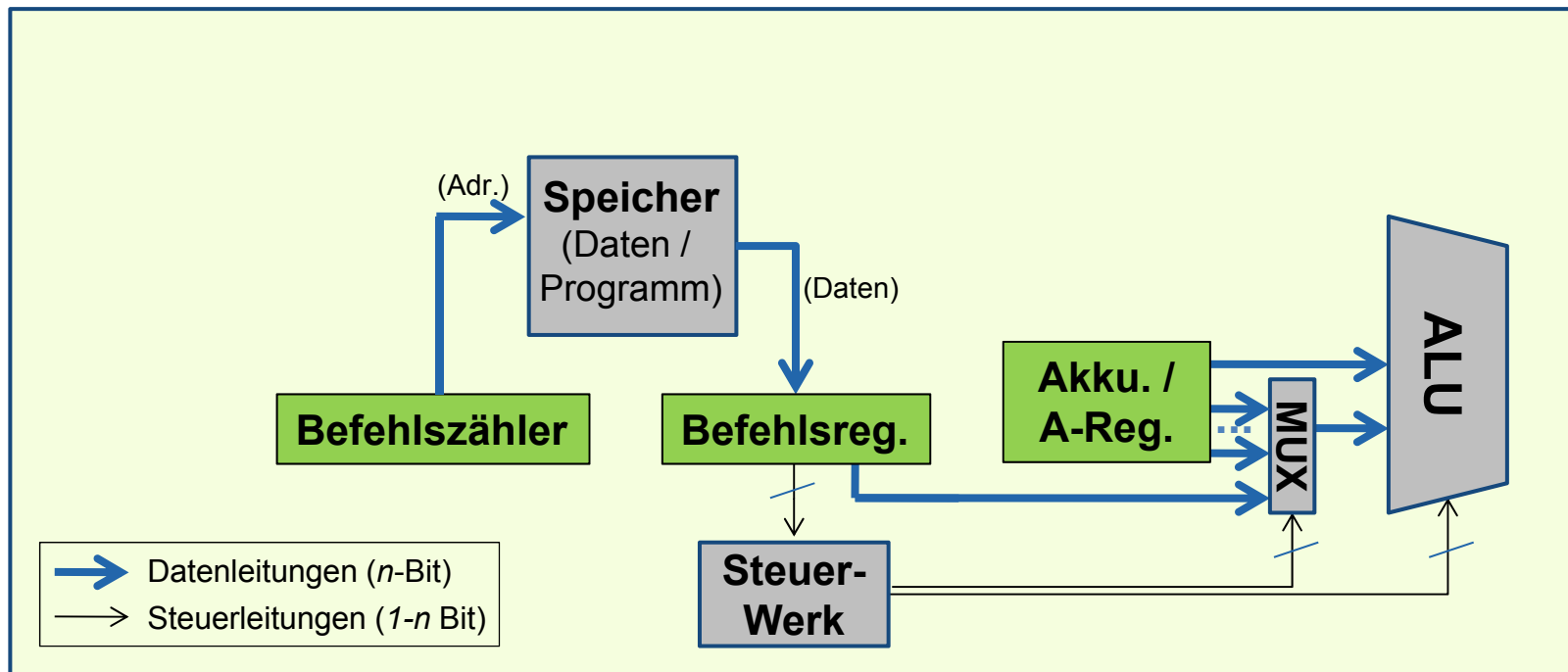
**Was fehlt nun noch?**

# Prozessor (CPU)

## Aufbau / Implementierung (Schema – vereinfacht)

### Schritte 1+2: Identisch

*Store + Load  
Befehl*

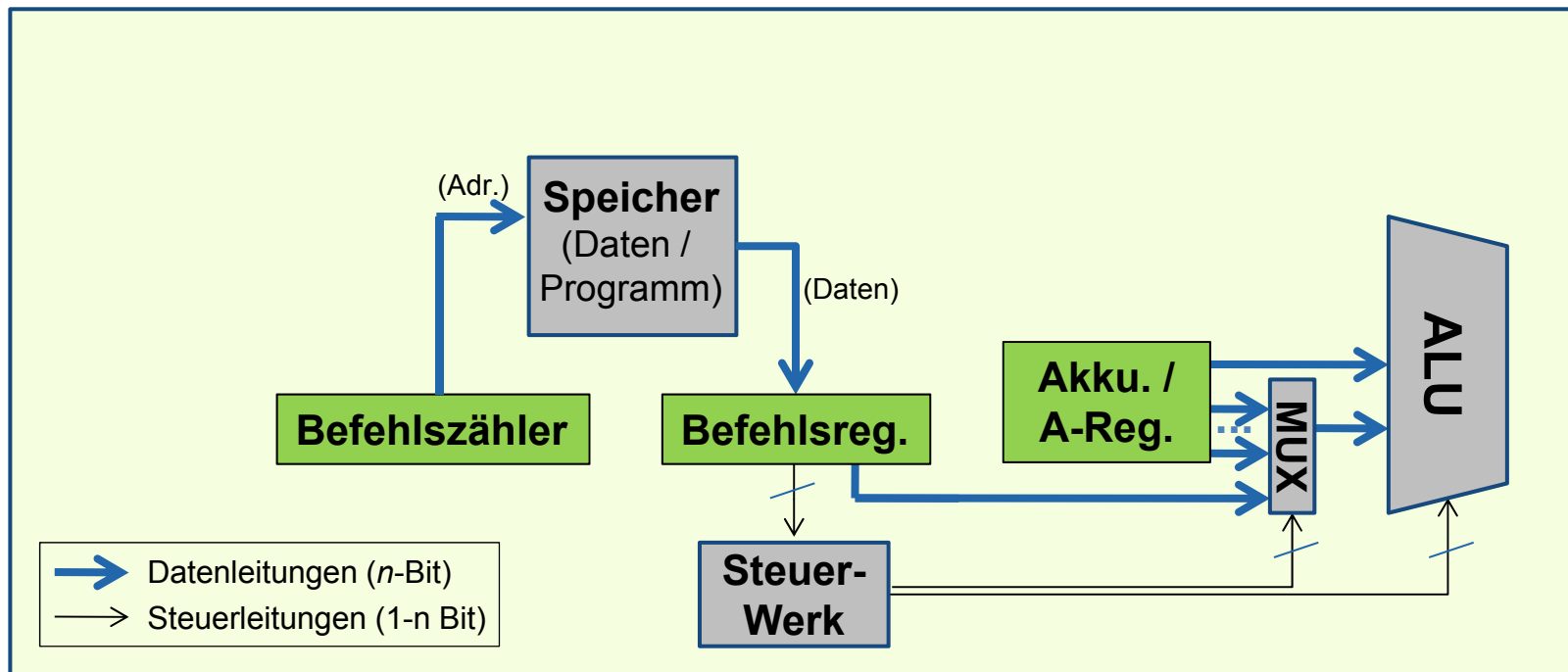




## Aufbau / Implementierung (Schema – vereinfacht)

**Schritt 3:** Die **ALU** berechnet die absolute Speicheradresse.

*Store + Load  
Befehl*



## Aufbau / Implementierung

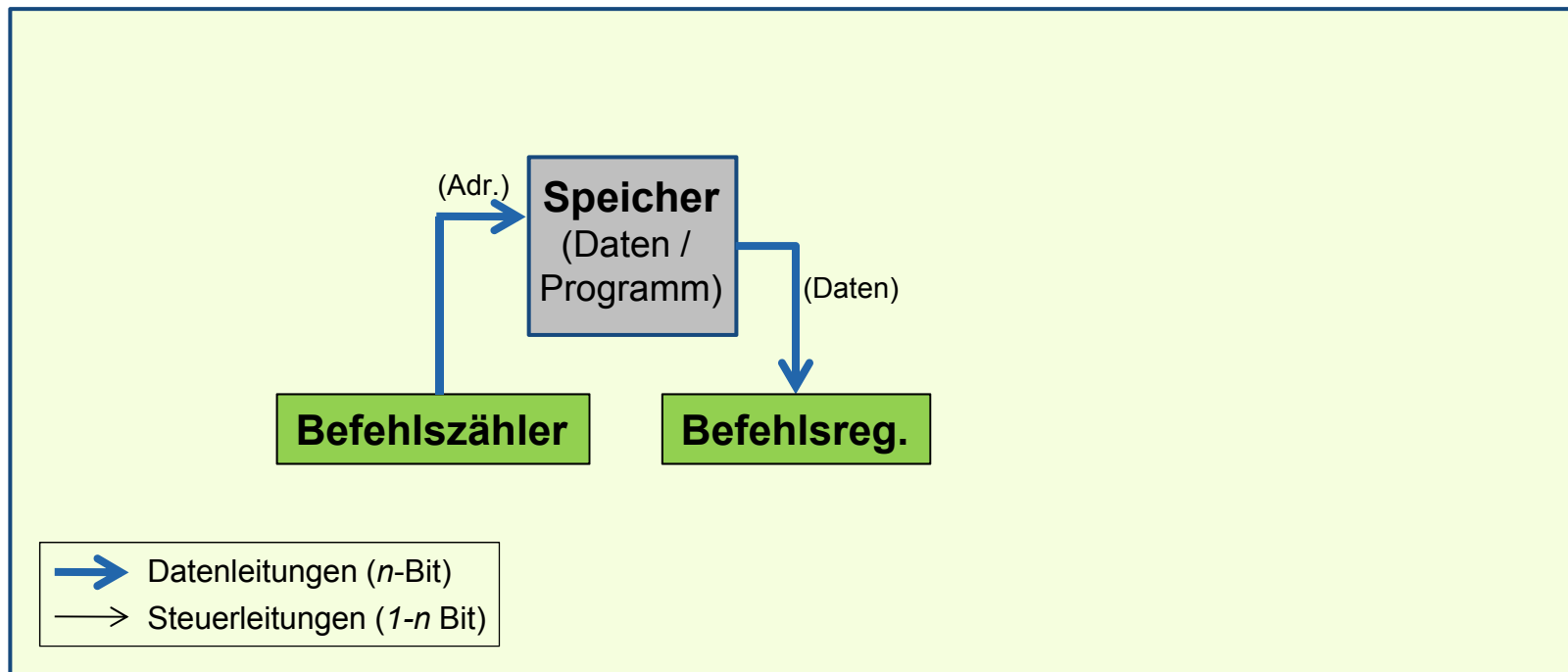
- Welche Art von Funktionen werden damit abgedeckt?
  - => Alle *arithmetisch-logischen Funktionen*, welche die *ALU* (das *Rechenwerk*) berechnen kann
  - => Speichern und *Laden* von *Daten* in oder aus einem Register in den bzw. aus dem Speicher

Was fehlt nun noch?

## Aufbau / Implementierung (Schema – vereinfacht)

**Schritte 1-3: Wiederum identisch**

*Sprung-  
Befehl*



## Aufbau / Implementierung (Schema – vereinfacht)

„*All together*“:

- In dieser vereinfachten Darstellung fehlen noch einige **wichtige Komponenten**, wie z. B. *Interrupts*, *erweiterte Ein- und Ausgabefunktionen*, *Vorzeichen*, *Statusregister*
- Auch gibt es noch **weitere Arten** von **Befehlen**, die zwar die **Leistungsfähigkeit** (schneller, einfacher) eines **Rechners vergrössern**, **nicht aber die Mächtigkeit** (*TM-äquivalent*).
- Dennoch ist sie völlig ausreichend, um die **grundsätzliche Arbeitsweise** von (auch aktuellen) **Rechnern** darzustellen.

# Prozessor (CPU)

## Zykluszeit

- Zurück zu den Leistungsmerkmalen eines Prozessors:  
=> **Befehlszahl**, **Zyklus**(zeit) und **CPI**.
- Den Benutzer interessiert die **Ausführungsdauer**  $t_p$  seines Programms **P**: diese hängt offensichtlich von der **Anzahl** der erforderlichen Befehle und der **Zykluszeit** ab:

$$t_p = (\text{Anzahl Befehle}) * \text{Zykluszeit}$$

(wenn vereinfacht davon ausgegangen wird, dass die **Zykluszeit** für jeden Befehl ein **Zyklus** ist => **CPI = 1**)

# Prozessor (CPU)

## Zykluszeit

- Den Benutzer interessiert die *Ausführungsdauer*  $t_p$  seines Programms  $P$ : diese hängt offensichtlich von der *Anzahl* der erforderlichen Befehle und der *Zykluszeit* ab:

$$t_p = (\text{Anzahl Befehle}) * \text{Zykluszeit}$$

(wenn vereinfacht davon ausgegangen wird, dass die *Zykluszeit* für jeden Befehl ein *Zyklus* ist  $\Rightarrow \text{CPI} = 1$ )

**Beispiel für ein Programm  $A$  auf einem Prozessor  $P$ :**

# Prozessor (CPU)

## Zykluszeit

- Idealerweise wird ein Prozessor so entworfen, dass die **Zykluszeit** minimal ist.

**Aber was ist der „Minimalwert“ bzw. wodurch wird er bestimmt?**

**=> Reale Bauelemente** haben (heute zwar kleine aber immer noch) **endliche Laufzeiten:**

Der **Stromfluss** durch die **elektrischen Bauelemente** („**Schalten von Gattern**“) **erfordert eine gewisse Zeit**, die gewartet werden muss, bis die **Bauelemente die gewünschte Operation / Funktion ausgeführt haben.**

**Hinweis:** Details hierzu werden in den Kursen Physik (3. Semester) und Elektronik (4. Semester) vermittelt

## Zykluszeit

- Idealerweise wird ein Prozessor so entworfen, dass die **Zykluszeit** minimal ist.

Aber was ist der „*Minimalwert*“ bzw. wodurch wird er bestimmt?

- => Wenn **alle Befehle** in einem **Zyklus** abgearbeitet werden, **bestimmt der Befehl mit dem längsten möglichen Datenpfad die Zykluszeit** (*worst case*).
- => Dieses kann, wie an den vereinfachten Ablaufskizzen gezeigt wurde, **sehr komplex** sein und ist i. d. R. **nie für alle Befehle identisch** (auch unter Verwendung eines vereinfachten Befehlssatzes).
- => D. h., dass bei **vielen Befehlen umsonst gewartet wird!**



## Zykluszeit

- Idealerweise wird ein Prozessor so entworfen, dass die **Zykluszeit** minimal ist.

Aber was ist der „*Minimalwert*“ bzw. wodurch wird er bestimmt?

=>

=> Damit ist es **praktisch unmöglich**, einen Prozessor für die **Befehle** zu **optimieren**, die am **meisten verwendet** werden (zumindest ist die Wahrscheinlichkeit dafür sehr gering).

Die **Optimierung** erfolgt somit **zwangsläufig** für einen **Befehl**, der **ggf. nur selten** für ein Programm verwendet wird.

# Prozessor (CPU)

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

a) Architektur, Hardware und Befehlssatz eines Prozessors so entwerfen, dass **alle Befehle dieselbe Zeit** erfordern.  
(*Datenpfad* gleich lang)

=> **Dieses ist in der Realität kaum umsetzbar** (oder nur für sehr einfache Prozessoren).

Beispiel: **Load/Store** mit **Einbezug des Speichers** gegenüber einer **logischen Verknüpfung**, wie dem **ODER**, und Einbezug von **Registern** oder einem **festen Wert** (im Befehl kodiert).

(Oder ev. hat der Prozessor ein Rechenwerk für die **Division** von **Fließkommazahlen**!)

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

b) **Befehle** in **Gruppen** unterteilen, die in etwa die gleiche Zeit erfordern (*Datenpfad* gleich lang), und ausgehend von einer **Mindestzeit** (= ein **Zyklus / Takt**) die **Befehle** einer Gruppe einheitlich in einem oder mehreren **Zyklen** ausführen.

**Beispiel:** Einfache Vergleiche, logische Operationen in einem **Taktzyklus**, Lade- und Speicherbefehle in drei **Taktzyklen** usw.

und ...

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

b) ... Ausführung von Befehlen (einer Gruppe) in einem oder mehreren Taktzyklen.

- => Ansatz wird in vielen Prozessoren erfolgreich genutzt, da **Zeit sehr effizient genutzt werden** kann.
- => **Optimierung für Befehle**, die häufig genutzt werden, ist möglich.
- => Steuerung / Design ist **komplexer**.

**Anmerkung:** Dieser Grundansatz wurde u. a. in der **gegenläufigen Zielsetzung** der **CISC-** und **RISC-Befehlssätze** für Rechner weiterentwickelt!

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

c) „**Pipelining**“

**Ansatz:** Mehrere Befehle werden *überlappend*, d. h. zeitgleich ausgeführt.

**Beispiel:** Nachdem der erste Befehl codiert ist und in der **ALU** verarbeitet wird, kann schon der nächste Befehl aus dem Speicher in den **Befehlszähler** geladen werden.

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

c) „**Pipelining**“

**Ansatz: Mehrere Befehle werden *überlappend*, d. h. zeitgleich ausgeführt.**

**Dazu wird die Abarbeitung eines Befehls in kleinere Teilaufgaben unterteilt, die unterschiedliche Ressourcen des Prozessors benötigen und ...**

**... diese Teilaufgaben parallel für aufeinanderfolgende Befehle durchgeführt.**

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:
  - c) „**Pipelining**“: Sequenzielle Befehlsabfolge (vereinfachtes Beispiel)
    - 1) Befehl laden (Speicher => Befehlsregister)
    - 2) Befehl decodieren (Steuerwerk)
    - 3) Operanden bereitstellen
    - 4) Rechenoperation durchführen (ALU) + Ergebnis Schreiben

**Anmerkung:** Wie viele Stufen eine Pipeline aufweist und welche Stufe welche Aufgabe(n) / Operation(en) durchführt, hängt von der konkreten Implementierung ab.

# Prozessor (CPU)

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der *Zykluszeit* bzw. Reduktion von „*Leerlaufzeiten*“:

### c) „*Pipelining*“: Sequenzielle Befehlsabfolge (Beispiel - Schema)

Schritt:	1	2	3	4	5	6	...
Befehl 1 (Load)	Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>			
Befehl 2 (Addition)		Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>		
Befehl 3 (Addition)			Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>	
Befehl 4 (Load)				Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>
Befehl 5 (ODER)					Befehl laden*	Befehl decode	Operanden bereitstel.
Befehl 6 (Store)						Befehl laden*	Befehl decode
...							

\* Für das Laden des Folgebefehls in der Pipeline muss der Befehlszähler jetzt bereits geeignet „*inkrementiert*“ werden.

+ Hier wird bei Sprungbefehlen erst abschliessend der Wert des Befehlszählers neu berechnet.

[Ist in der Realität erheblich komplexer - „*Branch Prediction*“.]



## Zykluszeit

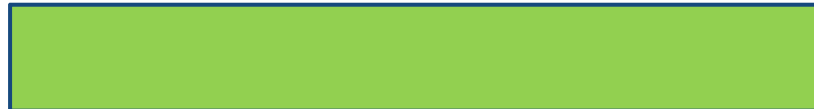
- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:
  - c) „**Pipelining**“: *Ideale Bedingungen*
    - Einzelne Teilaufgaben – **Pipeline-Stufen** – sind idealerweise gleich lang:
      - => Längste **Pipeline**-Stufe bestimmt Zykluszeit.
    - Anzahl der Befehle ist gross:
      - => **Pipeline**-Stufen sind „*ausgelastet*“.

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

c) „**Pipelining**“: Leistungssteigerungen (*Ideale Beding.*)

- Um wie viel schneller wird ein einzelner Befehl abgearbeitet?  
(„**Befehlsausführungszeit**“)



- Um wie viel schneller wird ein Programm abgearbeitet?  
(„**Gesamtausführungszeit**“)



## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:
  - c) „**Pipelining**“: *Reale Bedingungen*
    - Einzelne Teilaufgaben sind ungefähr gleich lang:
      - => Längste **Pipeline**-Stufe bestimmt **Zykluszeit** der **Pipeline**<sup>+</sup>.
      - => Es existieren **Wartezeiten**.
    - Es gibt eine Anlauf- und Auslaufphase:
      - => **Pipeline**-Stufen sind **nicht** immer voll „*ausgelastet*“.

<sup>+</sup>**Zykluszeit** = *max*(Zeit der Stufen) + Zusatzaufwand

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

- c) „**Pipelining**“: *Reale* Bedingungen (Forts.)

- Komplexität (Hardware) ist grösser:

=>

- Nicht alle Befehle nutzen auch alle Stufen der **Pipeline**

=>

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

c) „**Pipelining**“: Leistungssteigerungen (*Reale Beding.*)

- Um wie viel schneller wird ein einzelner Befehl abgearbeitet?  
(„**Befehlsausführungszeit**“)



- Um wie viel schneller wird ein Programm abgearbeitet?  
(„**Gesamtausführungszeit**“)



## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:
  - c) „**Pipelining**“: Leistungssteigerungen (Beispiel)
    - Ein mit **500 MHz** getakteter Prozessor (ohne **Pipeline**) benötigt für die Ausführung eines Programms mit a) **20** bzw. b) **2'500 Befehlen** **0.04 µs** bzw. **5 µs** (vereinfachte Annahme von **1 CPI**).
    - Nun wird der **Prozessor** des Rechners durch einen **Prozessor** mit einer **8-stufigen Pipeline** mit **3.2 GHz** aufgerüstet<sup>+</sup>.

**Welche Leistungssteigerung wird im Idealfall erzielt?**

(Unter vereinfachter Annahme, dass **keine Konflikte** vorkommen.)

<sup>+</sup>Unter realen Bedingungen kann jede einzelne Stufe **nicht 8-mal** so schnell sein (Taktrate entspräche dann **4 GHz**). **3.2 GHz** stellt einen durchaus realistischen Wert dar.

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

c) „**Pipelining**“: Leistungssteigerungen (Beispiel)

Gesamtausführungszeit  **$T$**  für  **$n$**  Befehle mit  **$k$** -stufiger **Pipeline**:

Bespiel Programm a:

Bespiel Programm b:

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:
  - c) „**Pipelining**“: Optimierungen
    - Wenige Stufen, um Komplexität zu beherrschen (i.d.R. 4 bis 12)
    - Optimierung der Befehlssätze:
      - **Formate aller Befehle** sind **identisch** oder **sehr ähnlich**.
      - **Wenige Befehlsformate** (i. d. R. wenige Befehle)
      - **Bei Befehlen mit Speicherzugriff** nur **Load** exklusiv oder (xor) **Store**.
      - **Organisation der Daten im Speicher** (wortweise)
    - **Weitere ...**



## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:
  - c) „**Pipelining**“: Herausforderungen
    - „**Strukturkonflikte**“ (engl. „**structural hazard**“)
    - „**Datenkonflikte**“ (engl. „**data hazard**“)
    - „**Steuerkonflikte**“ (engl. „**control hazard**“)

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

- c) „**Pipelining**“: Herausforderungen

- „**Strukturkonflikte**“ (engl. „**structural hazard**“):

- Gleichzeitiger Zugriff auf Ressourcen durch aufeinanderfolgende Befehle:

- => I. d. R. durch **Pipeline**-Architektur und **abgestimmten Befehlssatz** gelöst.

- Z. B. durch **Befehlssatz sicherstellen**, dass nur in der **ersten Pipeline-Stufe** auf **Daten** für Befehle im **Speicher** zugegriffen werden muss.

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

- c) „**Pipelining**“: Herausforderungen (Forts.)

- „**Datenkonflikte**“ (engl. „*data hazard*“):

- Ein **Befehl** greift auf **Daten** eines **vorherigen Befehls** zu, der noch **nicht abgeschlossen** ist.

**Beispiel:** Zwei **verknüpfte** Additionen

1) **Akku := Akku + Register\_1**

2) **Akku := Akku + Register\_2**

(Wobei im Akku das Ergebnis der vorausgegangenen Addition verwendet werden soll.)

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

- c) „**Pipelining**“: Herausforderungen (Forts.)

- „**Datenkonflikte**“ (engl. „*data hazard*“):

- Ein **Befehl** greift auf **Daten** eines **vorherigen Befehls** zu, der noch **nicht abgeschlossen** ist.

**Beispiel:** Zwei **verknüpfte** Additionen (Forts.)

- 1) **Addition** des **Akkumulators** mit einem **Register** (das Ergebnis wird im **Akkumulator** abgelegt)
- 2) **Addition** der **Summe** der **vorherigen Addition** (Akkumulator) mit einer **weiteren Zahl**

=> In der **Pipeline** liegt das **Ergebnis** noch **nicht** im **Akkumulator** vor und Befehl 2 **muss warten**.

# Prozessor (CPU)

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

- c) „**Pipelining**“: Herausforderungen (Forts.)

- „**Datenkonflikte**“ (engl. „*data hazard*“):

- Ein **Befehl** greift auf **Daten** eines **vorherigen Befehls** zu, der noch **nicht abgeschlossen** ist.

**Lösungsmethoden:**

- **Umordnen des Codes** (=> erfolgt durch Compiler).
- „**Forwarding**“ (auch „**Bypassing**“ genannt):
  - => Durch **zusätzliche Hardware** wird das Ergebnis **früher bereitgestellt** (=> Wartezeit **reduzieren**).

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

- c) „**Pipelining**“: Herausforderungen (Forts.)

- „**Steuerkonflikte**“ (engl. „*control hazard*“):

- Immer, wenn **Entscheidungen** die **Programmausführung beeinflussen** (alle **Sprungbefehle**, bedingt oder unbedingt).

- Lösungsmethoden:**

- „**Vorhersagen**“ (engl. „*branch prediction*“)

- => Erfolgt durch **zusätzliche Hardware** (dynamisch).

- => **Sehr komplex**, aber **sehr erfolgreich** (> 90% richtig);  
**kurze Pipeline vorteilhaft**

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“:

### Zusammenfassung:

- Moderne Prozessoren arbeiten i. d. R. zur Optimierung des Gesamtdurchsatzes mit **Pipelining** und nutzen gleichzeitig die Gruppierung von Befehlen mit unterschiedlichen Taktzyklen.
- Die Prozessoren verfügen häufig über zusätzliche Hardware (z. B. mehrere Rechenwerke, spezielle Hardware für „komplexe“ Operationen, ).
- Erst die Abstimmung aller Komponenten miteinander führt zu den heute erreichten sehr hohen Leistungen.

## Zykluszeit

- Möglichkeiten zu „*Optimierung*“ der **Zykluszeit** bzw. Reduktion von „*Leerlaufzeiten*“

### Zusammenfassung:

- Die aktuellen Entwicklungen gehen zu **Multikern-Prozessoren** (mit **Pipeline** usw.), wodurch die Komplexität noch herausfordernder wird, wenn diese effizient genutzt werden sollen !  
**Z. B. Hardware** (hier insbesondere **Speicherarchitektur** und **Speichermanagement**), **Compiler**, **Software** usw. müssen aufeinander abgestimmt sein.



## Aufbau / Implementierung (Schema – vereinfacht)

### Weiter im Kurs:

- In den folgenden Lektionen werden noch genauer die **Befehle** (Struktur, Aufbau) sowie die Umsetzung von **Speichern** (und **Cache**) betrachtet.
- Die anderen erforderlichen Bauelemente (**Register**, **ALU**, **Addierer**, **Multiplexer** usw.) werden im Kurs Elektronik (im 4. Semester) ausführlich behandelt.

# Prozessor (CPU)

Zürcher Hochschule  
für Angewandte Wissenschaften

**zhaw** School of  
Engineering

