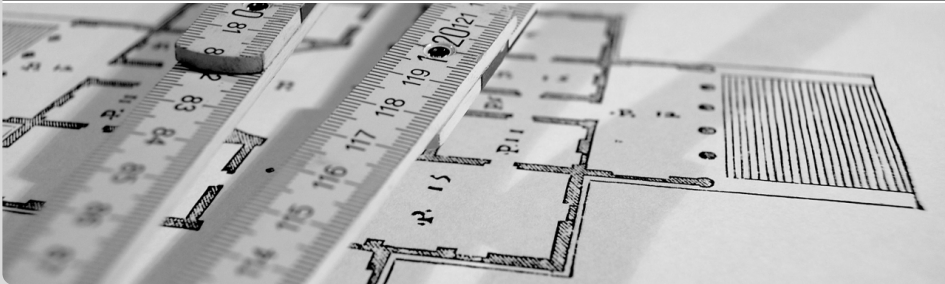


# Grundbegriffe der Informatik

## Tutorium 36

Termin 6 | 02.12.2016  
Thassilo Helmold

KIT – Karlsruher Institut für Technologie

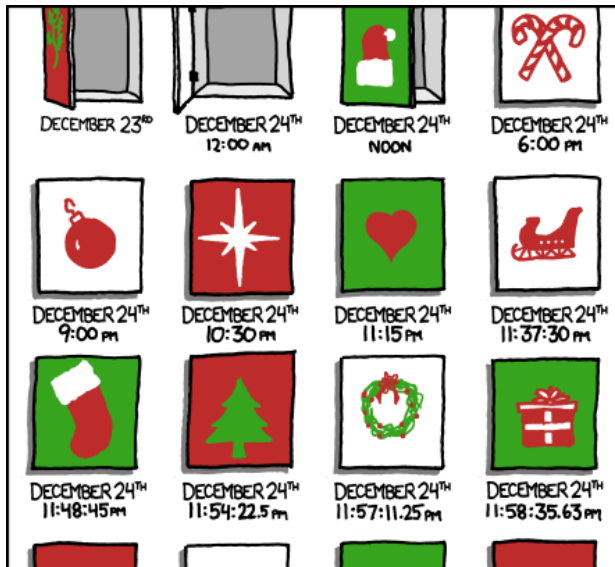


# Inhalt

Funktionen von Funktionen

Speicher

MIMA



## ZENO'S ADVENT CALENDAR

Abbildung: <https://www.xkcd.com/994/>

# Zum letzten Übungsblatt

- Vollständige Induktion: Die Induktionsvoraussetzung lautet „für **ein**  $i$  gilt...“.  
Für alle  $i$  zeigt ihr es erst in der Induktion!
- Kaum jemand hat das richtig gemacht, es ist aber wirklich essentiell. Wer das bei der Klausur falsch macht, verschenkt einfache Punkte!
- Folgendes ist **nicht** korrekt, um  $A$  zu zeigen:  
 $A \implies B$ ,  $B$  ist wahr. Also ist auch  $A$  wahr.  
Hier bedarf es Äquivalenz, damit aus  $B$  auch  $A$  folgt!

**In the previous episode of GBI...**

# Rückblick: Codierungen

- Codierungen: Injektive Abbildungen (meist Homomorphismen)
- $\varepsilon$ -Freiheit, Präfixfreiheit
- Einfaches decodieren für präfixfreie Homomorphismen
- Huffman-Codierungen: Präfixfreie Codes mit optimaler Länge

Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen

Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W



Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei

Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F

Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F
- Jede Huffman-Codierung ist präfixfrei

Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F
- Jede Huffman-Codierung ist präfixfrei W

Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F
- Jede Huffman-Codierung ist präfixfrei W
- Jeder  $\varepsilon$ -freie Homomorphismus ist präfixfrei

## Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F
- Jede Huffman-Codierung ist präfixfrei W
- Jeder  $\varepsilon$ -freie Homomorphismus ist präfixfrei F Aber:  
Jeder präfixfreie Homomorphismus ist  $\varepsilon$ -frei

## Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F
- Jede Huffman-Codierung ist präfixfrei W
- Jeder  $\varepsilon$ -freie Homomorphismus ist präfixfrei F Aber:  
Jeder präfixfreie Homomorphismus ist  $\varepsilon$ -frei
- Präfixfreie Codes sind einfach zu decodieren.

## Wahr oder falsch?

- Das Zweierkomplement ist gut zum Rechnen W
- Jeder Homomorphismus ist präfixfrei F
- Jede Huffman-Codierung ist präfixfrei W
- Jeder  $\varepsilon$ -freie Homomorphismus ist präfixfrei F Aber:  
Jeder präfixfreie Homomorphismus ist  $\varepsilon$ -frei
- Präfixfreie Codes sind einfach zu decodieren. W



# Zum Aufwärmen: Vogelfarben

Wir zeigen nun:

**Alle Vögel haben die gleiche Farbe!**

Achtung: Dieser Beweis enthält selbstverständlich einen Fehler!

# Zum Aufwärmen: Vogelfarben

Dazu verwenden wir **vollständige Induktion** und zeigen die folgende, äquivalente Aussage:

$\forall n \in \mathbb{N}_+ : \text{In jeder Menge, die genau } n \text{ Vögel enthält,}$   
haben alle Vögel die gleiche Farbe.

# Zum Aufwärmen: Vogelfarben

$\forall n \in \mathbb{N}_+ :$  In jeder Menge, die genau  $n$  Vögel enthält,  
haben alle Vögel die gleiche Farbe.

## Induktionsanfang

$n = 1$ : Wenn eine Menge genau 1 Vogel enthält, dann haben offensichtlich alle Vögel die gleiche Farbe.

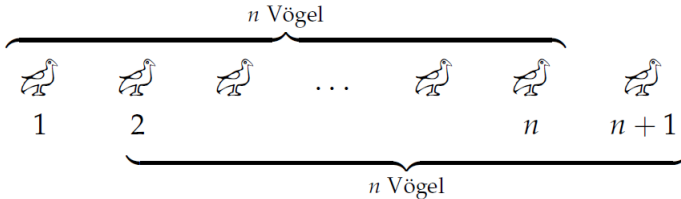
## Induktionsvoraussetzung

Für ein beliebiges aber festes  $n$  gelte: In jeder Menge, die genau  $n$  Vögel enthält, haben alle Vögel die gleiche Farbe.

# Zum Aufwärmen: Vogelfarben

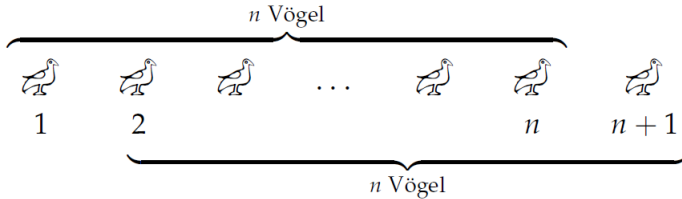
## Induktionsschluss

Man zeige die Aussage für  $n + 1$ : Sei also  $M$  eine Menge, die genau  $n + 1$  Vögel enthalte. Man stelle sich vor, dass die Vögel alle nebeneinander sitzen:



# Zum Aufwärmen: Vogelfarben

## Induktionsschluss

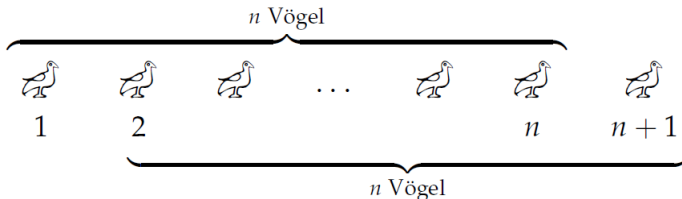


Die Vögel 1, 2, ...,  $n$  bilden eine Menge mit genau  $n$  Vögeln. Also haben sie nach Induktionsvoraussetzung alle die gleiche Farbe.

Die Vögel 2, 3, ...,  $n+1$  bilden auch eine Menge mit genau  $n$  Vögeln. Also haben nach Induktionsvoraussetzung auch diese alle die gleiche Farbe.

Folglich haben auch die Vögel 1 und  $n+1$  die gleiche Farbe, also haben alle Vögel die gleiche Farbe.

# Vogelfarben: Auflösung



Das Bild ist zwar außerordentlich hübsch, suggeriert aber leider etwas, was nicht immer stimmt: Für  $n = 2$  überlappen sich die Teilmengen „ohne den ersten“ und „ohne den letzten“ Vogel nicht. Es ist also nicht erzwungen, dass beide Vögel die gleiche Farbe haben. (Und das macht „alles weitere“ auch kaputt: Wenn nicht immer 2 Vögel die gleiche Farbe haben, dann auch nicht immer 3 Vögel, usw.)

# Funktionen von Funktionen

Speicher

MIMA

# Funktionen

## Definition

Seien  $A$  und  $B$  Mengen. Mit  $B^A$  bezeichnen wir die Menge aller Abbildungen von  $A$  nach  $B$ , also

$$B^A = \{f \mid f : A \rightarrow B\}$$

Man kann sich das wie eine Tabelle vorstellen. Wir wählen für jedes  $a \in A$  ein  $b \in B$ .

## Beobachtung

Für endliche Mengen  $A$  und  $B$  gilt:

$$|B^A| = |B|^{|A|}$$



## Problem: Ziffern filtern

Wir wollen aus einer Ziffernfolge (Wort aus  $Z_{10}^*$ ) Ziffern herausfiltern, die eine bestimmte Eigenschaft haben:

Gerade

$$\text{filter}_{\text{even}} : Z_{10}^* \rightarrow Z_{10}^*$$

$$\varepsilon \mapsto \varepsilon$$

$$z \cdot v \mapsto \begin{cases} z \cdot \text{filter}_{\text{even}}(v) & \text{falls } z \in \{2, 4, 6, 8, 0\} \\ \text{filter}_{\text{even}}(v) & \text{sonst} \end{cases}$$

$$\text{wobei } z \in Z_{10} \text{ und } v \in Z_{10}^*$$

## Problem: Ziffern filtern

Wir wollen aus einer Ziffernfolge (Wort aus  $Z_{10}^*$ ) Ziffern herausfiltern, die eine bestimmte Eigenschaft haben:

Ungerade

$$\text{filter}_{\text{odd}} : Z_{10}^* \rightarrow Z_{10}^*$$

$$\varepsilon \mapsto \varepsilon$$

$$z \cdot v \mapsto \begin{cases} z \cdot \text{filter}_{\text{odd}}(v) & \text{falls } z \in \{1, 3, 5, 7, 9\} \\ \text{filter}_{\text{odd}}(v) & \text{sonst} \end{cases}$$

$$\text{wobei } z \in Z_{10} \text{ und } v \in Z_{10}^*$$

## Problem: Ziffern filtern

Wir wollen aus einer Ziffernfolge (Wort aus  $Z_{10}^*$ ) Ziffern herausfiltern, die eine bestimmte Eigenschaft haben:

Löst die Ungleichung  $x^2 - 3 * x < 20$

$$filter_{solves} : Z_{10}^* \rightarrow Z_{10}^*$$

$$\varepsilon \mapsto \varepsilon$$

$$z \cdot v \mapsto \begin{cases} z \cdot filter_{solves}(v) & \text{falls } z^2 - 3 * z < 20 \\ filter_{solves}(v) & \text{sonst} \end{cases}$$

$$\text{wobei } z \in Z_{10} \text{ und } v \in Z_{10}^*$$

# Problem: Ziffern filtern

## Aufgabe

Schreibt die Funktionen zum Filtern aller Ziffern...

- $> 5$
- $< 5$
- $< 2$  oder  $> 7$
- ... die perfekt sind
- ... die sich ausmalen lassen

Nein, das wäre natürlich Unsinn!

Alle Funktionen unterscheiden sich nur an einer Stelle: Der Bedingung, die erfüllt sein muss, damit die Ziffer in das Ergebnis mit aufgenommen wird.

# Filter

Definiere stattdessen eine universelle Filter-Funktion:

$$\text{filter}_p : Z_{10}^* \rightarrow Z_{10}^*$$

$$\varepsilon \mapsto \varepsilon$$

$$z \cdot v \mapsto \begin{cases} z \cdot \text{filter}_p(v) & \text{falls } p(v) = \mathbf{w} \\ \text{filter}_p(v) & \text{sonst} \end{cases}$$

wobei  $z \in Z_{10}$  und  $v \in Z_{10}^*$

Für alle

$$p : Z_{10} \rightarrow \mathbb{B}$$

# Filter

Definiere beispielsweise nun:

$$even : Z_{10} \rightarrow \mathbb{B}, z \mapsto \begin{cases} \mathbf{w} & \text{falls } z \in \{2, 4, 6, 8, 0\} \\ \mathbf{f} & \text{sonst} \end{cases}$$

Dann ist  $filter_{even}(123456) = 135$

Auch für alle anderen Filter müssen wir nur noch die  $p$ -Funktionen definieren, was viel einfacher und kürzer ist als jedes Mal die ganze Filter-Funktion neu zu definieren.

# Charakteristische Funktion

Sei  $M$  eine abzählbar unendliche Menge und  $L \subseteq M$ .

## Definition

Die Charakteristische Funktion einer (Teil-)Menge  $L$  ist die Funktion

$$C_L : M \rightarrow \{0, 1\}$$
$$x \mapsto \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

Also gilt  $C_L \in \{0, 1\}^M$

Klar ist außerdem: Es gibt eine Bijektion  $C$  zwischen Mengen und Charakteristischen Funktionen:

$$C : 2^M \rightarrow \{0, 1\}^M, L \mapsto C_L$$

# Charakteristische Funktion

Jetzt können wir auf diesen Funktionen äquivalente Operationen wie auf Mengen definieren...

Vereinigung:  $V: \{0, 1\}^M \times \{0, 1\}^M \rightarrow \{0, 1\}^M$

Beispielbild für  $L_1 = \{a, c, d\}$  und  $L_2 = \{b, c\}$

	$L_1$	$L_2$	$L_1 \cup L_2$
$x$	$f_1(x)$	$f_2(x)$	$V(f_1, f_2)$
$a$	1	0	1
$b$	0	1	1
$c$	1	1	1
$d$	1	0	1
$e$	0	0	0



# Charakteristische Funktion

Jetzt können wir auf diesen Funktionen äquivalente Operationen wie auf Mengen definieren...

Vereinigung:  $V: \{0, 1\}^M \times \{0, 1\}^M \rightarrow \{0, 1\}^M$

Wie definiert man  $V(f_1, f_2)$ ? Zum Beispiel so:

$$\begin{aligned} V: \{0, 1\}^M \times \{0, 1\}^M &\rightarrow \{0, 1\}^M \\ (f_1, f_2) &\mapsto (x \mapsto \max(f_1(x), f_2(x))) \end{aligned}$$

Oder so:  $V(f_1, f_2)(x) = \max(f_1(x), f_2(x))$

Wir haben gesehen:

- Eine Abbildung, die eine Funktion auf einen Wert abbildet
- Eine Abbildung, die einen Wert auf eine Funktion abbildet

Gleich werden wir das kombinieren!

(Hinweis: Abbildung und Funktion sind hier synonyme Begriffe!)

Funktionen von Funktionen

Speicher

MIMA

# Bit vs Byte

## Definition

Eine **Bit** ist ein Zeichen des Alphabets  $\{0, 1\}$

Ein Wort aus 8 *Bits* wird *Byte* genannt.

## Definition

Ein *Speicher*  $m$  bildet Adressen ( $Adr$ ) auf Werte ( $Val$ ) ab.

Also  $m \in Val^{Adr}$

Hier interessiert uns nicht die konkrete Realisierung der Speicherung, sondern nur die abstrakte Funktionsweise.

## Hinweis

Im Umgang mit Speicher benutzen wir hier nur Zahlen im Binärsystem.

D.h.  $Adr = 2^k$ ,  $Val = 2^l$   $k, l \in \mathbb{N}_+$ .

# Speicher

Methoden:

*memread* (*m*, *adr*) um eine Zelle zu lesen.

*memwrite* (*m*, *adr*, *val*) um in eine Zelle zu schreiben.

$$\begin{aligned} \text{memread} : \text{Val}^{\text{Adr}} \times \text{Adr} &\rightarrow \text{Val} \\ (m, a) &\mapsto m(a) \end{aligned}$$

$$\begin{aligned} \text{memwrite} : \text{Val}^{\text{Adr}} \times \text{Adr} \times \text{Val} &\rightarrow \text{Val}^{\text{Adr}} \\ (m, a, v) &\mapsto m' \end{aligned}$$

Mit

$$m'(a') = \begin{cases} v & \text{falls } a' = a \\ m(a') & \text{falls } a' \neq a \end{cases}$$

# Speicher

## Beispiel

$\text{memread}(m, 01) = 00000111$

Speicher $m$	
00	00101000
01	00000111
10	10010110
11	00100101

Zustand bei  $t = 0$

# Speicher

## Beispiel

$\text{memread}(m, 01) = 00000111$

$\text{memwrite}(m, 01, 11111100)$

$\text{memread}(\text{memwrite}(m, 01, 11111100), 01) = 11111100$

Speicher $m$	
00	00101000
01	11111100
10	10010110
11	00100101

Zustand bei  $t = 1$

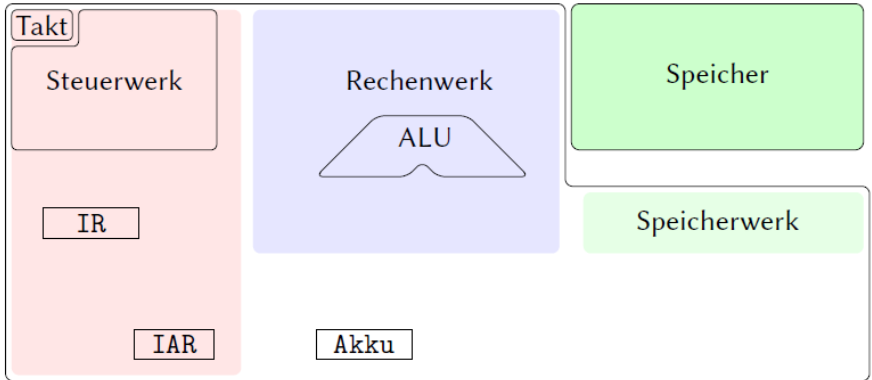
Funktionen von Funktionen

Speicher

MIMA



# MIMA

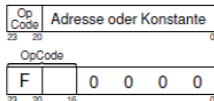


Die *MIMA* ist ein idealisierter Prozessor.

# Eigenschaften

- Adressen sind 20 Bit lang
- „Werte“ sind 24 Bit lang
- Befehlskodierungen:
  - 4 Bit für den OpCode und 20 Bit für einen Parameter (Adresse / Konstante)
  - 8 Bit Befehl (Rest irrelevant)

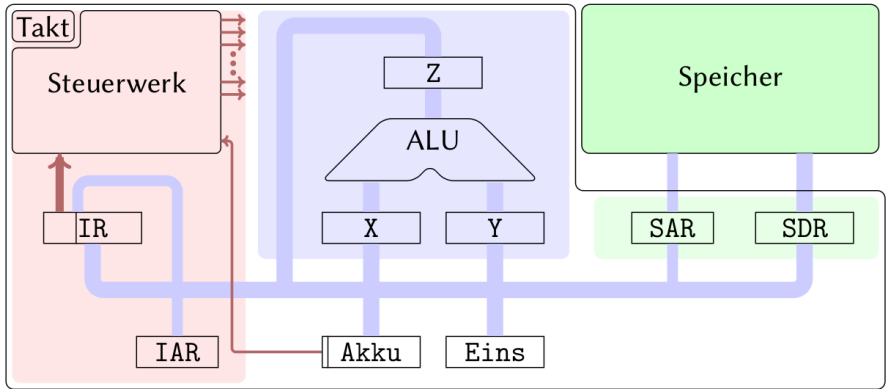
## *Befehlsformate*



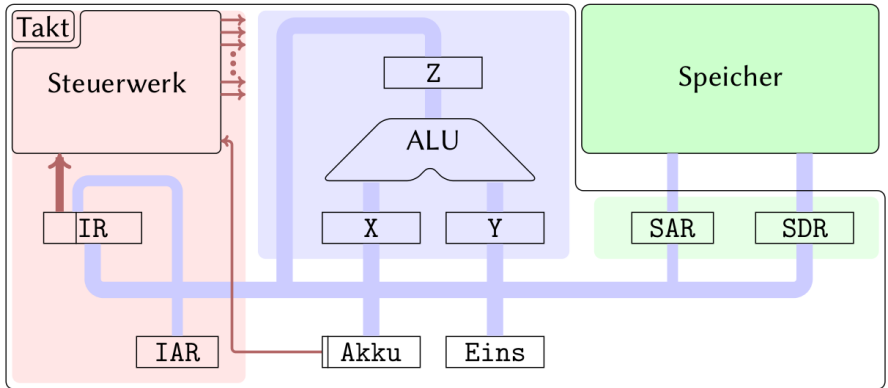
# Wichtige Register

- *IAR* : InstruktionsAdressRegister : Speichert Adresse des aktuell auszuführenden Befehls.
- *IR*: InstruktionsRegister : Speichert den auszuführenden Befehl.
- *SAR*: SpeicherAdressRegister : Enthält die Adresse eines Wertes, der aus dem Speicher gelesen werden soll.
- *SDR*: SpeicherDatenRegister : Enthält einen Wert, der aus dem Speicher geladen wurde.

# Aufbau



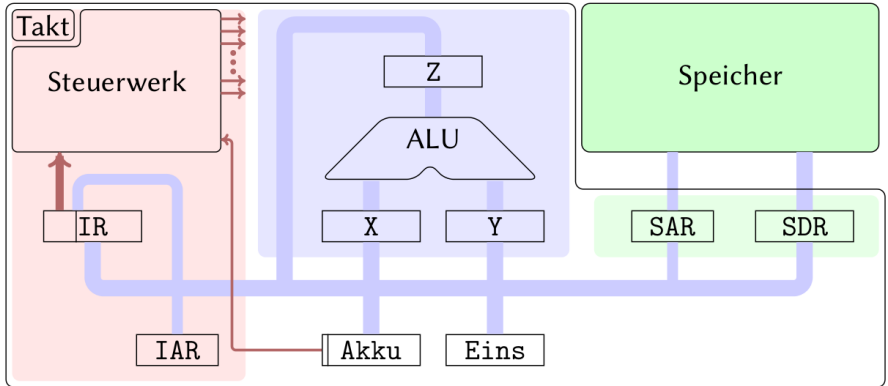
# Befehlsholphase



1  $IAR \rightarrow SAR$  und  $IAR \rightarrow X$

Befehlsadresse dem Speicher übergeben und Zähler zum Erhöhen an ALU geben.

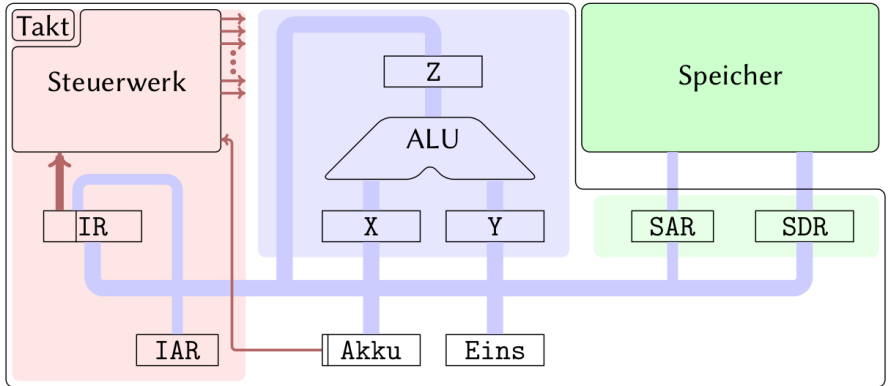
# Befehlsholphase



2 Eins  $\rightarrow$  Y

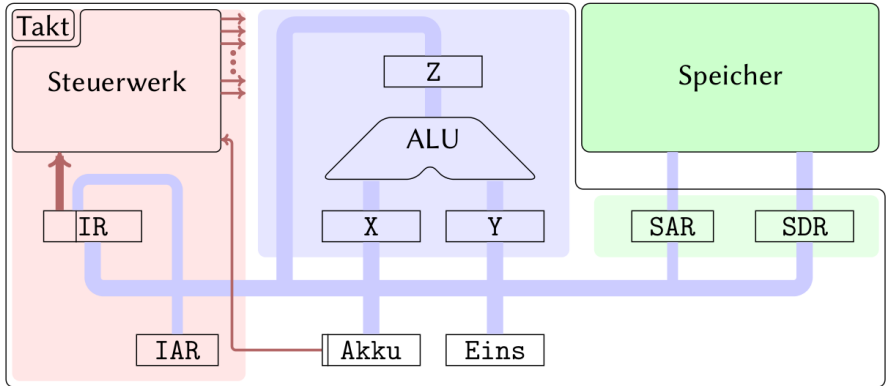
1-Wert für Erhöhung des Zählers an ALU geben.

# Befehlsholphase



- 3 ALU aufaddieren ( $Z=X+Y$ )  
Nächste Befehlsadresse berechnen.

# Befehlsholphase

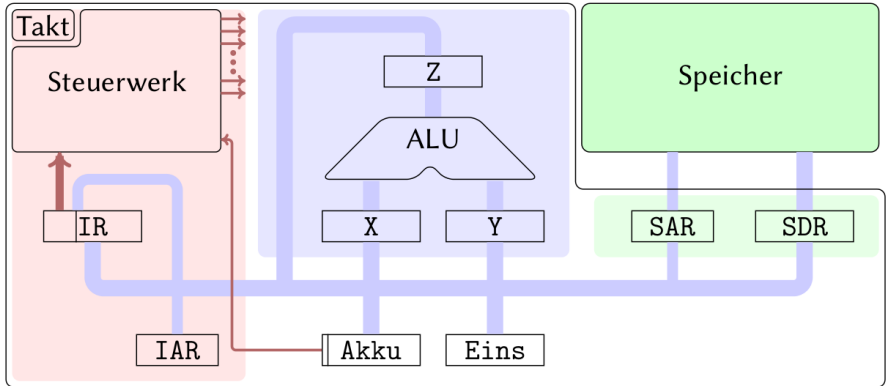


4  $Z \rightarrow \text{IAR}$

Adresse für nächste Runde speichern.



# Befehlsholphase



5 SDR → IR

Wert zur angefragten Adresse erhalten.

# Befehle

Die MIMA besitzt einen Befehlssatz mit möglichen Befehlen. Andere Befehle (oder Varianten) werden NICHT unterstützt und können daher nicht verwendet werden!

- Rechenoperationen
  - ADD adr
  - AND, OR, XOR adr
  - NOT, RAR (keine Parameter)
- Zugriffsoperationen
  - LDC const (const ist dabei eine **20-Bit** Konstante)
  - LDV, STV adr
  - LDIV, STIV adr
- Vergleichsoperation: EQL adr (-1 wenn gleich, 0 sonst)
- Sprünge
  - JMP adr
  - JMN adr (Jump if negative)
- HALT

# Bemerkungen

## Indirekte Adressierung

	Hauptspeicher					
	Akku	...	M(46)	...	M(81)	...
initial	?		81		25	
LDV 46						
	81		81		25	
LDIV 46						
	25		81		25	

## HALT

Jedes Programm muss mit HALT enden! Sonst läuft das Programm endlos weiter!

## Negative Konstanten

Negative Konstanten können nicht mit LDC geladen werden.

Warum? Unser Akku ist 24 Bit breit, aber wir können nur in die hinteren 20 Bit laden!

# Negative Zahlen

## Aufgabe

Schreibe ein Programm, das von einer an Adresse  $a_1$  gegebenen positiven Zahl das Zweierkomplement berechnet und an Adresse  $a_2$  ablegt.

# Negative Zahlen

## Aufgabe

Schreibe ein Programm, das von einer an Adresse  $a_1$  gegebenen positiven Zahl das Zweierkomplement berechnet und an Adresse  $a_2$  ablegt.

## Lösung

LDV  $a_1$

NOT

STV  $a_2$

LDC 1

ADD  $a_2$

STV  $a_2$

# Beispiele

Beispiele zur Umsetzung von Anweisungen aus Hochsprachen:  
Siehe Übung

# Übung: Modulo

Schreibe ein Programm, das eine an Speicheradresse  $a_1$  gegebenen Zahl Modulo  $R$  rechnet und an Adresse  $a_2$  ablegt.

- Modulo 2
- Modulo 3

# Lösung: Modulo 2

start: LDC 1 //000000000000000000000001

AND  $a_1$

STV  $a_1$

HALT



# Lösung: Modulo 3

start: LDC 1  
STV One  
LDC 3  
NOT  
ADD One  
STV MThree

while: LDV MThree  
ADD  $a_1$   
JMN abschluss  
STV  $a_1$   
JMP while

ende: LDC 3  
ADD  $a_1$   
HALT

## Was ihr nun wissen solltet

- Speicher - Abbildungen, die Abbildungen auf Abbildungen abbilden!
- Wie ein einfacher Prozessor aufgebaut ist
- Wie man die MIMA für einfache Aufgaben nutzt

## Was nächstes Mal kommt

- Grammatik ist schwer - aber nicht bei uns!

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.



I AM A GOD.

Abbildung: <https://www.xkcd.com/676/>

# Credits

Vorgänger dieses Foliensatzes wurden erstellt von:

Thassilo Helmold

Philipp Basler

Nils Braun

Dominik Doerner

Ou Yue