

Grundbegriffe der Informatik

Kapitel 10: Prozessor

Mattias Ulbrich

(basierend auf Folien von Thomas Worsch)

KIT · Institut für Theoretische Informatik

Wintersemester 2023/2024

MIMA — die Minimalmaschine ist ein idealisierter Prozessor

- Original MIMA aus Vorlesung «Rechnerorganisation»
- erweiterte Version MIMA-x für die Vorlesungen
«Programmieren» und «Softwaretechnik» in Kap. 21
- alle fundamentalen Bestandteile und Konzepte
wie in realen Prozessoren

Einfache „Hardware“-„Bausteine“

Grobstruktur der MIMA

Maschinenbefehle der MIMA

Mikroprogrammsteuerung der MIMA

Ein Beispielprogramm

Wo sind wir?

Einfache „Hardware“-„Bausteine“

Grobstruktur der MIMA

Maschinenbefehle der MIMA

Mikroprogrammsteuerung der MIMA

Ein Beispielprogramm

Drähte verbinden „Erzeuger“ und „Verbraucher“

-
- für einen „Draht“ gibt es *drei* Möglichkeiten
 - es wird eine 0 übertragen
 - es wird eine 1 übertragen
 - es wird nichts übertragen
 - manchmal durch Z repräsentiert

Drähte verbinden „Erzeuger“ und „Verbraucher“

0

- für einen „Draht“ gibt es *drei* Möglichkeiten
 - es wird eine 0 übertragen
 - es wird eine 1 übertragen
 - es wird nichts übertragen
 - manchmal durch Z repräsentiert

Drähte verbinden „Erzeuger“ und „Verbraucher“

1

- für einen „Draht“ gibt es *drei* Möglichkeiten
 - es wird eine 0 übertragen
 - es wird eine 1 übertragen
 - es wird nichts übertragen
 - manchmal durch Z repräsentiert

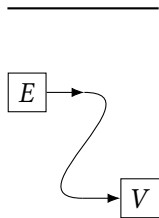
Drähte verbinden „Erzeuger“ und „Verbraucher“

- für einen „Draht“ gibt es *drei* Möglichkeiten
 - es wird eine 0 übertragen
 - es wird eine 1 übertragen
 - es wird nichts übertragen
 - manchmal durch Z repräsentiert

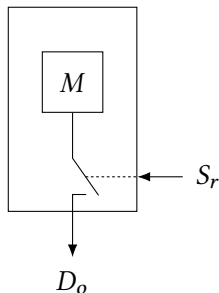
Z

Drähte verbinden „Erzeuger“ und „Verbraucher“

- für einen „Draht“ gibt es *drei* Möglichkeiten
 - es wird eine 0 übertragen
 - es wird eine 1 übertragen
 - es wird nichts übertragen
 - manchmal durch Z repräsentiert
- an jedem Drahtende
 - *Erzeuger*: kann Bit schreiben (oder nichts tun) oder
 - *Verbraucher*: kann Bit lesen (oder nichts tun)
- Draht kann weder 0 noch 1 speichern
 - wenn kein Erzeuger ein Bit „liefert“, wird nichts übertragen

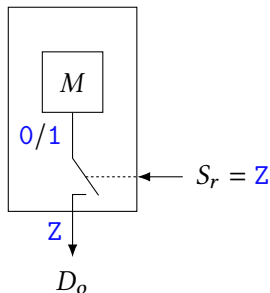


„Erzeuger“ für ein Bit



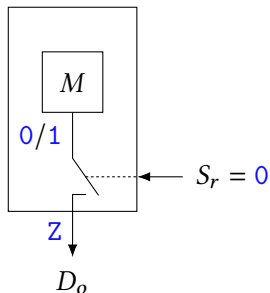
- Bestandteile
 - interne Schaltung M
 - u. U. Eingänge, weitere Steuerleitungen
- *Steuerleitung S_r*
- *Datenleitung D_o*

„Erzeuger“ für ein Bit



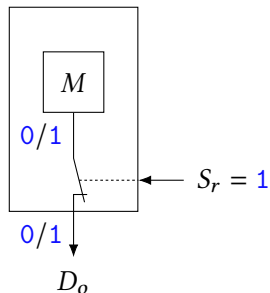
- Bestandteile
 - interne Schaltung M
 - u. U. Eingänge, weitere Steuerleitungen
- *Steuerleitung S_r*
 - liefert ggf. Befehl zur „Ausgabe“: $S_r = 1$
- *Datenleitung D_o*
 - liefert ein Bit, falls $S_r = 1$ ist
 - sonst nichts — „ Z “

„Erzeuger“ für ein Bit

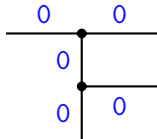


- Bestandteile
 - interne Schaltung M
 - u. U. Eingänge, weitere Steuerleitungen
- *Steuerleitung S_r*
 - liefert ggf. Befehl zur „Ausgabe“: $S_r = 1$
- *Datenleitung D_o*
 - liefert ein Bit, falls $S_r = 1$ ist
 - sonst nichts — „ Z “

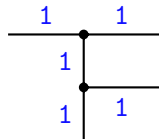
„Erzeuger“ für ein Bit



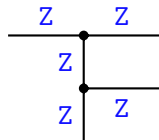
- Bestandteile
 - interne Schaltung M
 - u. U. Eingänge, weitere Steuerleitungen
- *Steuerleitung S_r*
 - liefert ggf. Befehl zur „Ausgabe“: $S_r = 1$
- *Datenleitung D_o*
 - liefert ein Bit, falls $S_r = 1$ ist
 - sonst nichts – „Z“



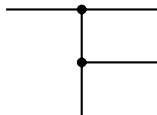
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen



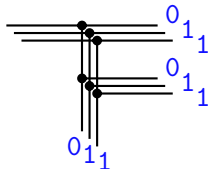
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen



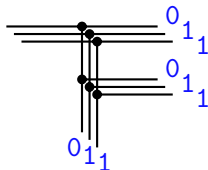
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen



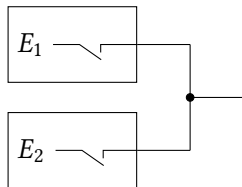
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen

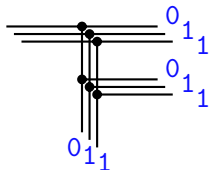


- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“

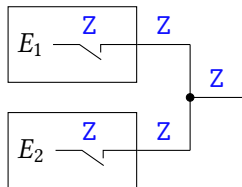


- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“
- Ausgänge mehrerer Erzeuger können miteinander verbunden sein

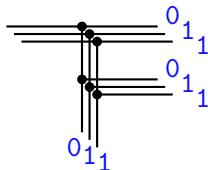




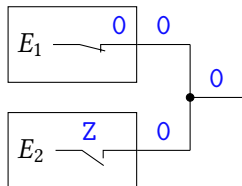
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen **Bus**
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“



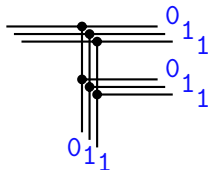
- Ausgänge mehrerer Erzeuger können miteinander verbunden sein
- **erlaubt:** maximal einer liefert ein Bit



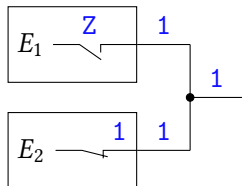
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“



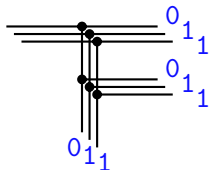
- Ausgänge mehrerer Erzeuger können miteinander verbunden sein
- *erlaubt*: maximal einer liefert ein Bit



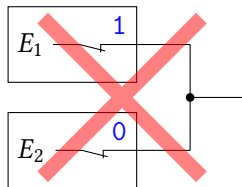
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen *Bus*
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“



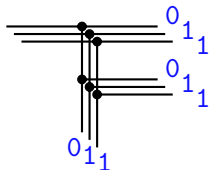
- Ausgänge mehrerer Erzeuger können miteinander verbunden sein
- *erlaubt*: maximal einer liefert ein Bit



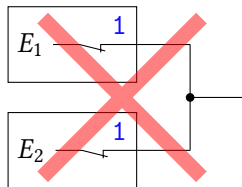
- Drähte kann man miteinander verbinden, so etwas nennt man auch einen **Bus**
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“



- Ausgänge mehrerer Erzeuger können miteinander verbunden sein
 - **erlaubt:** maximal einer liefert ein Bit
 - **verboten:** mehrere Erzeuger liefern gleichzeitig Bit auf gleichen Bus

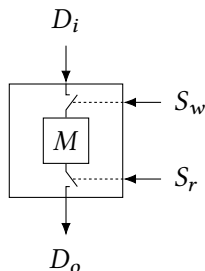


- Drähte kann man miteinander verbinden, so etwas nennt man auch einen **Bus**
- überall wird der gleiche Wert übertragen
- oder überall wird nichts übertragen
- oft auch mehrere Leitungen „parallel“

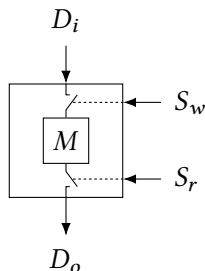


- Ausgänge mehrerer Erzeuger können miteinander verbunden sein
 - **erlaubt:** maximal einer liefert ein Bit
 - **verboten:** mehrere Erzeuger liefern gleichzeitig Bit auf gleichen Bus

Einfaches „Speicher-Element“ für ein Bit

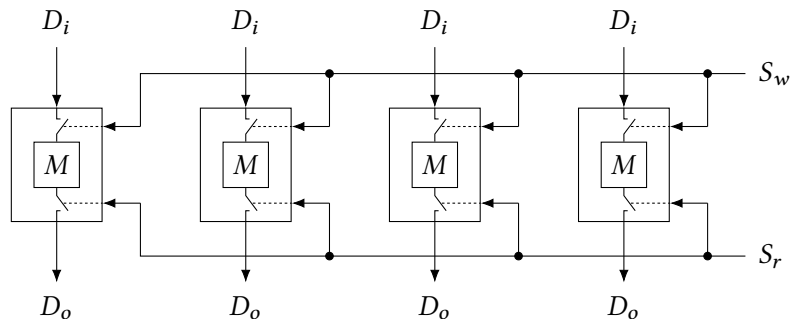


- Bestandteile
 - *interner Speicher M*
 - immer entweder in Zustand 0 oder in Zustand 1
 - *Steuerleitungen*
 - auf S_w wird Schreibbefehl geliefert
 - auf S_r wird Lesebefehl geliefert
 - *Datenleitungen*
 - auf D_i wird neu zu speicherndes Bit geliefert
 - auf D_o wird gelesenes Bit geliefert

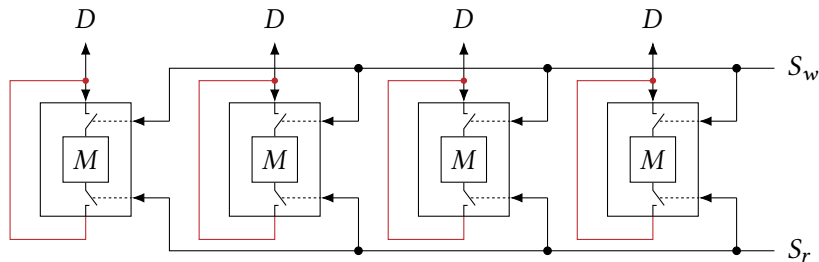


- zwei mögliche Aktionen
 - *ein Bit speichern: $S_w = 1$*
 - explizit auf D_i zur Verfügung gestellt
 - ersetzt bisher gespeichert Bit
 - *ein Bit lesen: $S_r = 1$*
 - wird auf D_o geliefert
 - gespeichertes Bit bleibt unverändert
- **verboten:** beide Aktionen gleichzeitig
- zwischen den Aktionen
 - gespeichertes Bit unverändert
 - D_o liefert „nichts“

Register — mehrere Ein-Bit-Speicher nebeneinander



Register — mehrere Ein-Bit-Speicher nebeneinander



manchmal D_i und D_o miteinander verbunden

Wo sind wir?

Einfache „Hardware“-„Bausteine“

Grobstruktur der MIMA

Maschinenbefehle der MIMA

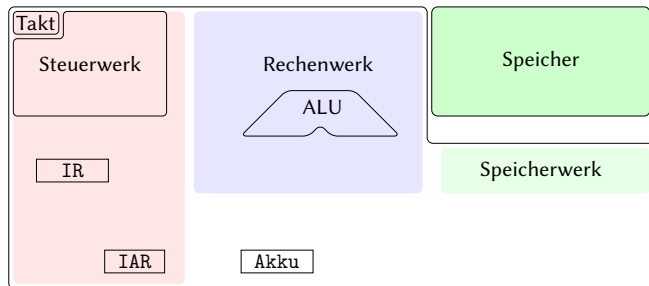
Mikroprogrammsteuerung der MIMA

Ein Beispielprogramm

[https://commons.wikimedia.org/wiki/vonNeumann_\(public_domain\)](https://commons.wikimedia.org/wiki/vonNeumann_(public_domain))



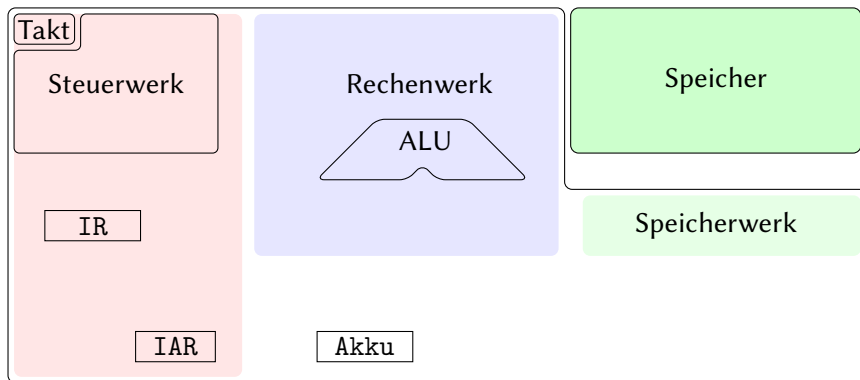
- geboren 28. Dezember 1903 in Budapest
gestorben 8. Februar 1957 in Washington, D.C.
- Mathematiker mit wesentlichen Beiträgen zu
 - mathematische Logik
 - Funktionalanalysis
 - Quantenmechanik
 - Spieltheorie
- einer der Väter der Informatik



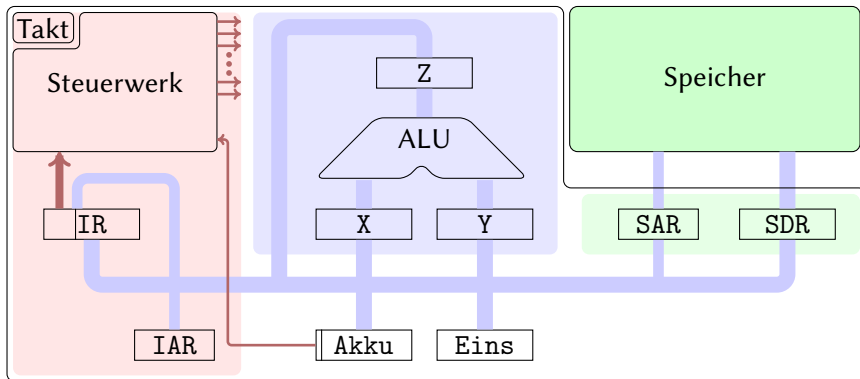
- Hauptspeicher
- Prozessor
 - Register
 - Steuerwerk
 - Rechenwerk
 - Speicherwerk
- wir machen es uns bequem
reden z. B. von
«Zahlen», «Addition» ...

- Größen
 - Adressen: 20 bit
 - Werte: 24 bit
 - sogenannte (Speicher-)Worte
- Programmspeicher
 - ein Maschinenbefehl je Wort
 - 4 bit Befehlskodierung und 20 bit Adresse/Wert oder
 - 8 bit Befehlskodierung, Rest irrelevant
- Datenspeicher
 - Eingaben, Zwischenergebnisse, Ausgaben
 - Zweierkomplementdarstellung
- Trennung von Programm und Daten
 - nur per Konvention, nicht erzwungen

- Register
 - Speicher für je ein Datenwort bzw. eine Adresse
 - symbolische Namen, z. B. **Akkumulator**
- Rechenwerk
 - **ALU** *arithmetic logic unit*
 - verschiedene Funktionen
 - Argumente aus **Akku** bzw. Befehl/Speicher
- Steuerwerk
 - **IR** *instruction register*
 - **IAR** *instruction address register*
- Speicherwerk
 - **SAR** *storage address register*
 - **SDR** *storage data register*



Verbindungen zwischen den Teilen durch einen **Daten- und Adressbus**



Steuersignale veranlassen Register und ALU,
auf einem **Bus** zu lesen bzw. zu schreiben
Meldesignale beeinflussen Arbeitsweise des Steuerwerks

- Mima hat **Von-Neumann-Architektur**
 - gemeinsamer Speicher für Befehle und Daten
- **Harvard-Architektur**
 - physisch getrennte Speicher für Befehle und Daten
 - gleichzeitiges Laden von Befehlen und Daten
 - besserer Speicherschutz
- heutige Rechner haben eine Mischung

Wo sind wir?

Einfache „Hardware“-„Bausteine“

Grobstruktur der MIMA

Maschinenbefehle der MIMA

Mikroprogrammsteuerung der MIMA

Ein Beispielprogramm

Maschinenbefehle werden aus dem Speicher geladen und ausgeführt

- verschiedene Typen von Befehlen
 - Transport von Daten
 - Verarbeitung von Daten
 - Beeinflussung der Befehlsreihenfolge
- verschiedene Notationsmöglichkeiten
 - Bitfolgen
 - symbolische Namen
 - für Befehle
 - für Adressen

- Notation
 - MIMA: Folge von 24 Bits
 - z. B. 0010000000000000000101010
 - Mensch: symbolisch
 - STV 101010
 - STV 42
 - $M(adr)$: Hauptspeicherinhalt an Adresse adr
- MIMA-Befehle z. B.
 - LDC *const* «load constant»
 - Akku $\leftarrow const$ als 24-Bit-Muster
 - LDV *adr* «load value from address»
 - Akku $\leftarrow M(adr)$
 - STV *adr* «store value at address»
 - $M(adr) \leftarrow \text{Akku}$

$$0 \leq const \leq 2^{20} - 1$$

$$0 \leq adr \leq 2^{20} - 1$$

	Akku	...	Hauptspeicher		...
			M(46)	M(47)	
LDC 13	13				
STV 46					
	13		13		
LDC 25					
	25		13		
STV 47					
	25		13	25	
LDV 46					
	13		13	25	

	Hauptspeicher				
	Akku	...	M(46)	M(47)	...
LDC 13	13				
STV 46	13		13		
LDC 25	25		13		
STV 47	25		13	25	
LDV 46	13		13	25	

	Hauptspeicher				
	Akku	...	M(46)	M(47)	...
LDC 13					
	13				
STV 46					
	13		13		
LDC 25					
	25		13		
STV 47					
	25		13	25	
LDV 46					
	13		13	25	

	Hauptspeicher				
	Akku	...	M(46)	M(47)	...
LDC 13					
	13				
STV 46					
	13		13		
LDC 25					
	25		13		
STV 47					
	25		13	25	
LDV 46					
	13		13	25	

	Akku	...	Hauptspeicher		...
			M(46)	M(47)	
LDC 13	13				
STV 46					
	13		13		
LDC 25					
	25		13		
STV 47					
	25		13	25	
LDV 46	13		13	25	

- MIMA-Befehle
 - **LDIV** *adr* «load value indirect from address»
 - $\text{Akku} \leftarrow M(M(\text{adr}))$
 - **STIV** *adr* «store value indirect at address»
 - $M(M(\text{adr})) \leftarrow \text{Akku}$
-  dabei werden von $M(\text{adr})$ nur die 20 niedrigstwertigen Bits berücksichtigt
- vergleiche Vorlesung «Programmieren»
 - Objekte
 - Referenzen

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

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

arithmetisch/logische Befehle

■ ADD *adr*

- $\text{Akku} \leftarrow \text{Akku} \text{ „+“ } M(\text{adr})$
- für eine gewisse Interpretation von „+“
(Zweierkomplement!)

■ «logische» Operationen

- $\text{Akku} \leftarrow \text{Akku} \text{ „op“ } M(\text{adr})$
- eine 1 entspricht «wahr»
- eine 0 entspricht «falsch»
- Anwendung an allen 24 Stellen der Operanden separat

x_1	x_2	and	or	xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

■ AND *adr*

■ OR *adr*

■ XOR *adr*

exklusives Oder, d. h. «Addition modulo 2»

			Hauptspeicher				
	Akku	...	M(46)	M(47)	M(48)	M(49)	...
initial	?		11	22	33	44	
LDC 0							
	0		11	22	33	44	
ADD 46							
	11		11	22	33	44	
ADD 47							
	33		11	22	33	44	
ADD 48							
	66		11	22	33	44	
ADD 49							
	110		11	22	33	44	

	Hauptspeicher						
	Akku	...	M(46)	M(47)	M(48)	M(49)	...
initial	?		11	22	33	44	
LDC 0							
	0		11	22	33	44	
ADD 46							
	11		11	22	33	44	
ADD 47							
	33		11	22	33	44	
ADD 48							
	66		11	22	33	44	
ADD 49							
	110		11	22	33	44	

	Hauptspeicher						
	Akku	...	M(46)	M(47)	M(48)	M(49)	...
initial	?		11	22	33	44	
LDC 0							
	0		11	22	33	44	
ADD 46							
	11		11	22	33	44	
ADD 47							
	33		11	22	33	44	
ADD 48							
	66		11	22	33	44	
ADD 49							
	110		11	22	33	44	

	Hauptspeicher						
	Akku	...	M(46)	M(47)	M(48)	M(49)	...
initial	?		11	22	33	44	
LDC 0							
	0		11	22	33	44	
ADD 46							
	11		11	22	33	44	
ADD 47							
	33		11	22	33	44	
ADD 48							
	66		11	22	33	44	
ADD 49							
	110		11	22	33	44	

	Hauptspeicher						
	Akku	...	M(46)	M(47)	M(48)	M(49)	...
initial	?		11	22	33	44	
LDC 0							
	0		11	22	33	44	
ADD 46							
	11		11	22	33	44	
ADD 47							
	33		11	22	33	44	
ADD 48							
	66		11	22	33	44	
ADD 49							
	110		11	22	33	44	

- einstellige Operationen auf Akku

- NOT

- Invertierung der Akku-Bits — wie exklusives Oder mit $11 \dots 11$

- RAR «rotate accumulator right»

- Rotation der Akku-Bits nach rechts
aus $x_{23}x_{22}x_{21} \dots x_2x_1x_0$ wird $x_0x_{23}x_{22}x_{21} \dots x_2x_1$

- Vergleichsoperation

- EQL *adr* «equal?»

- $$\text{Akku} \leftarrow \begin{cases} 11 \dots 11 & \text{falls Akku} = M(\text{adr}) \\ 00 \dots 00 & \text{sonst} \end{cases}$$
$$= \begin{cases} -1 & \text{falls Akku} = M(\text{adr}) \\ 0 & \text{sonst} \end{cases}$$

- «normale» Reihenfolge
 - Befehle aus aufeinander folgenden Speicherstellen geholt

Adresse	Befehl	
000101	LDC 111	«lädt 7 in den Akku»
000110	STV 111100	«speichert die 7 an Adresse 60»
000111	LDC 101010	«lädt 42 in den Akku»
001000	ADD 111100	«addiert Wert von Adresse 60» «jetzt steht 49 im Akku»
- Befehlsadressen irrelevant
 - weggelassen, solange nicht in Befehlen benutzt
- Stelle des nächsten Befehls durch
Instruktionsadressregister IAR festgelegt
 - muss also «regelmäßig» erhöht werden

Sprünge ändern die normale Reihenfolge der Programmabarbeitung

- unbedingter Sprung
 - **JMP** *adr* «*jump*»
 - setze fort mit Befehl in Adresse *adr*
 - Fortsetzung der Programmausführung an explizit angegebener Stelle

000101	LDC 111	«lädt 7 in den Akku»
000110	STV 111101	«speichert die 7 an Adresse 61»
000111	JMP 011000	«springt zum Befehl an Adresse 56»
	⋮	
011000	LDC 101011	«lädt 43 in den Akku»
011001	ADD 111101	«addiert Wert von Adresse 61»
		«jetzt steht 50 im Akku»

Rückwärtssprünge gehen natürlich auch ...

000101	LDC 1	«lädt 1 in den Akku»
000110	ADD 111101	«addiert Wert von Adresse 61»
000111	STV 111101	«speichert neuen Wert an Adresse 61»
001000	JMP 000101	«springt zum Befehl an Adresse 5»

- Dieses Programm hält nie an.
- Will man so etwas?

Bedingte Sprünge — Ausführungen hängen vom Inhalt des Akku ab

- Zweierkomplement
 - die *negativen* Zahlen haben höchstwertiges Bit gesetzt
- bedingter Sprung
 - **JMN** *adr* «*jump if negative*»
 - setze fort mit Befehl in Adresse *adr*, falls in Akku
 - höchstwertiges Bit auf 1 ist, d. h.
 - eine negative Zahl repräsentiert ist

000101	LDV 12	lädt $M(12)$ in den Akku
000110	EQL 13	vergleicht Akku mit $M(13)$
000111	JMN 011101	Sprung, <i>wenn</i> Vergleich erfolgreich
001000		hier weiter falls $M(12) \neq M(13)$
001001		
⋮		
011101		hier weiter falls $M(12) = M(13)$

- es gibt übrigens den Befehl **HALT**
(beeinflusst auch die Befehlsabarbeitung)

Wo sind wir?

Einfache „Hardware“-„Bausteine“

Grobstruktur der MIMA

Maschinenbefehle der MIMA

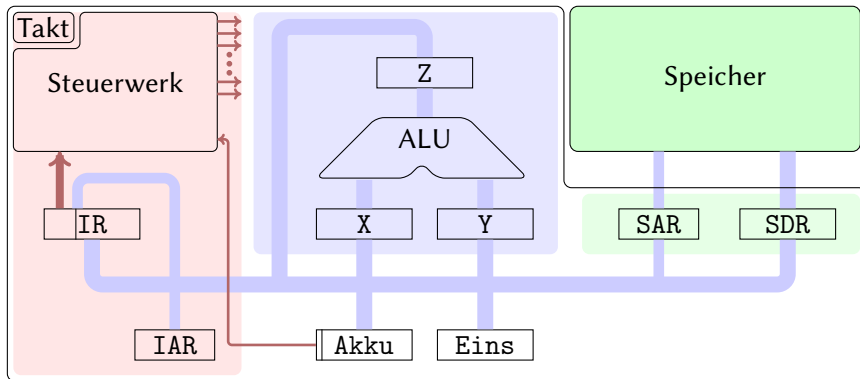
Mikroprogrammsteuerung der MIMA

Ein Beispielprogramm

Arbeitsweise der MIMA— für jeden Maschinenbefehl ein Mikroprogramm

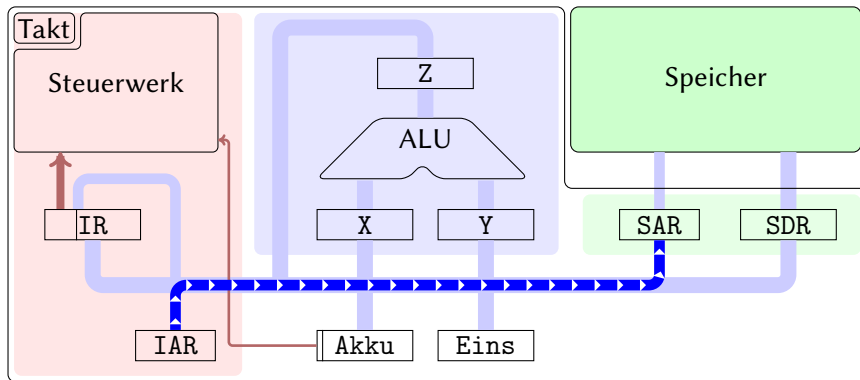
- **Maschinenbefehle**
- Durchführung in drei Phasen
 - Holphase
 - Decodierphase
 - Ausführungsphase
- **Mikroprogramme**
- in jeder Phase werden einige Mikrobefehle ausgeführt

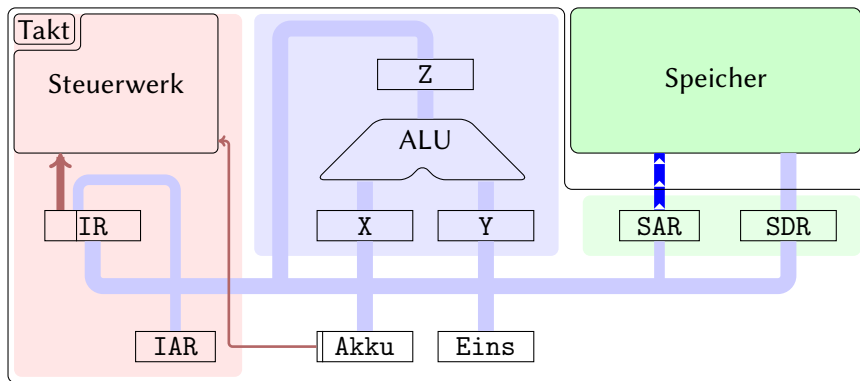
MIMA — die Minimalmaschine ist ein idealisierter Prozessor

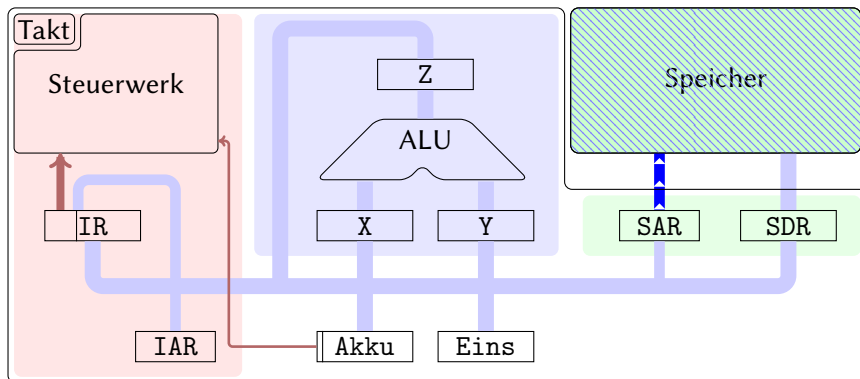


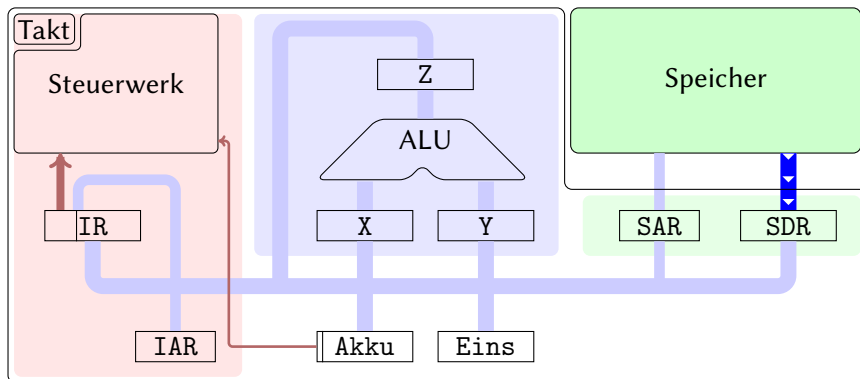
Steuersignale veranlassen Register und ALU, Werte zu lesen/liefern
Meldesignale beeinflussen Arbeitsweise des Steuerwerks

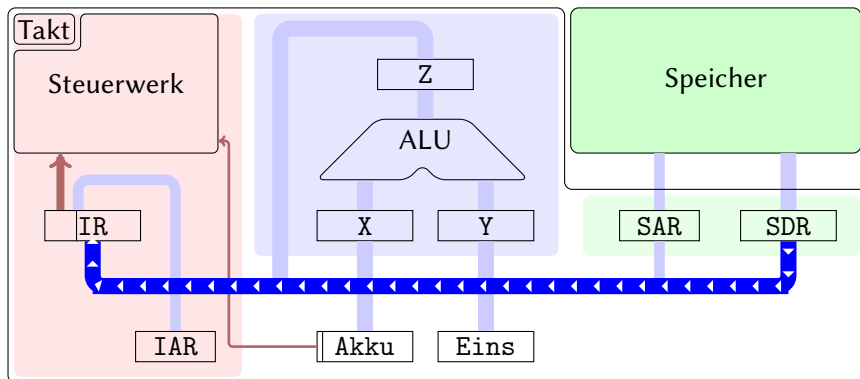
- zwei Aspekte
 - Holen des Befehls
 - Berechnung der Adresse des potenziell nächsten Befehls
- Steuersignale nicht eingezeichnet

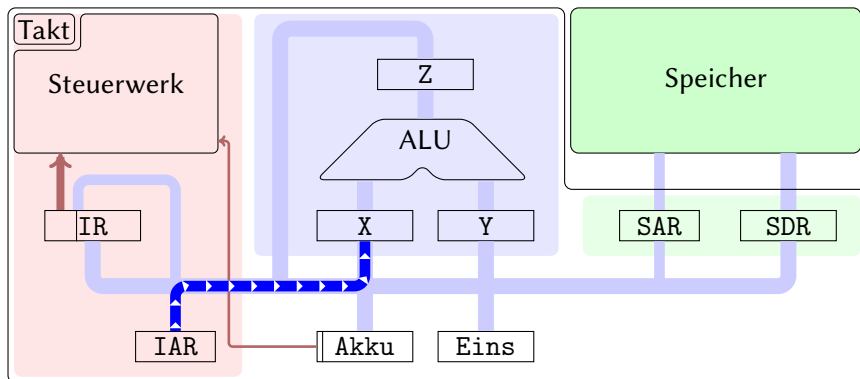


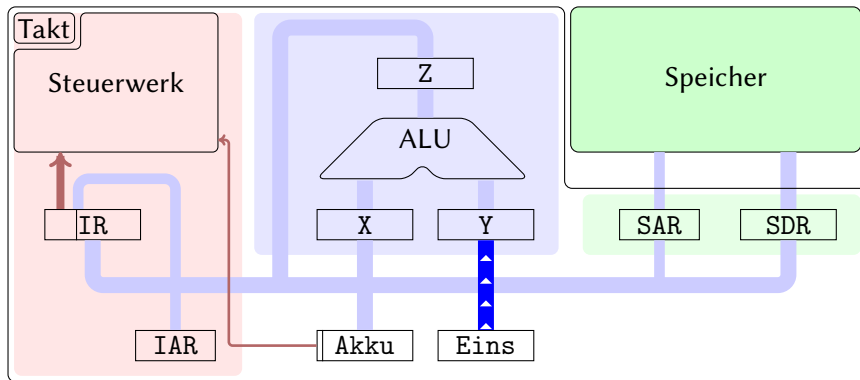


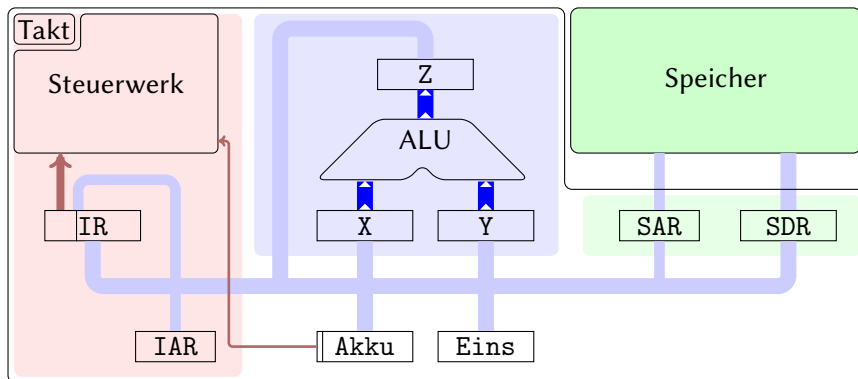


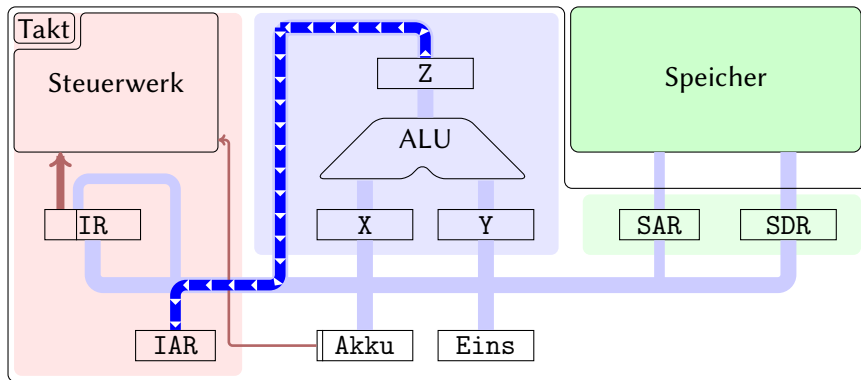


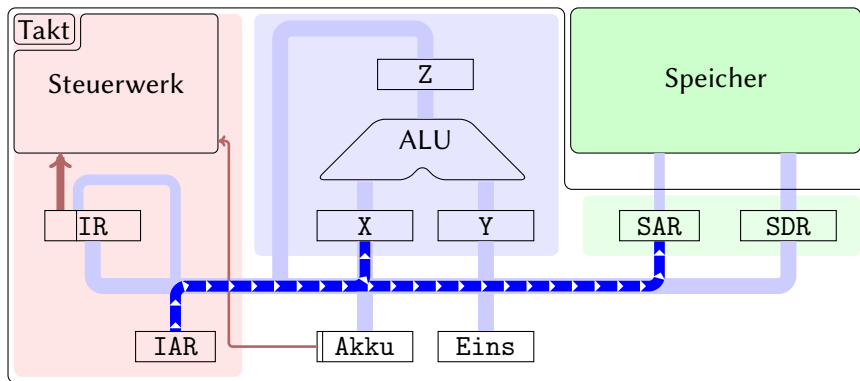


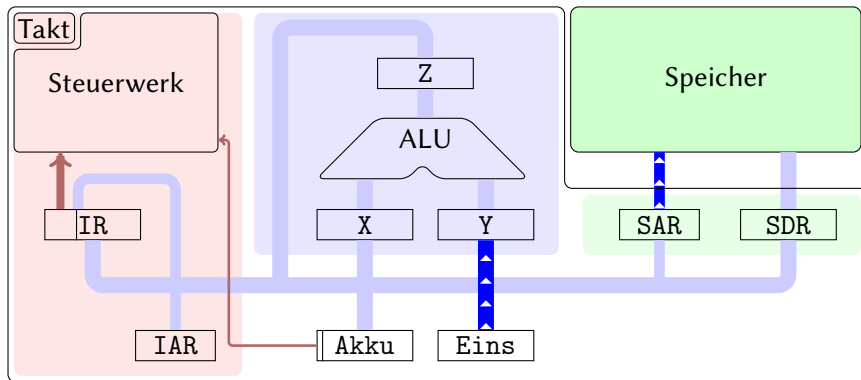


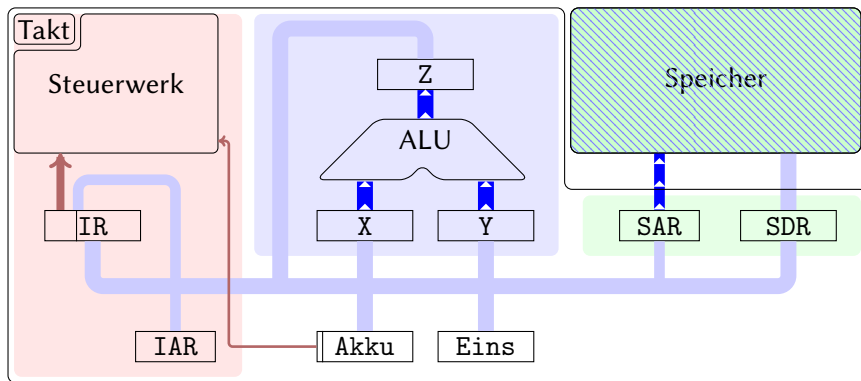


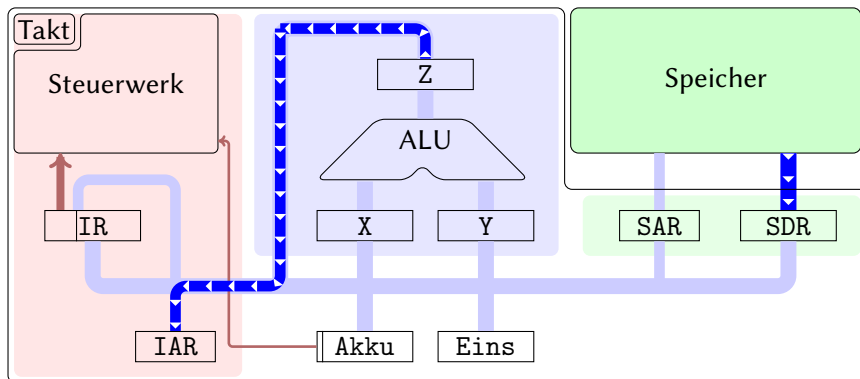


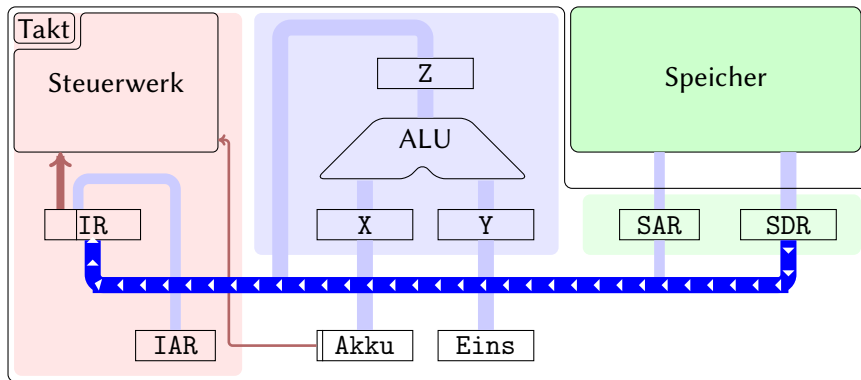




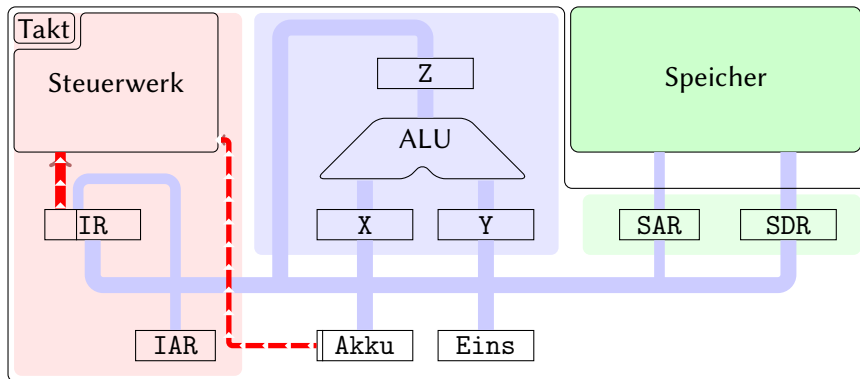






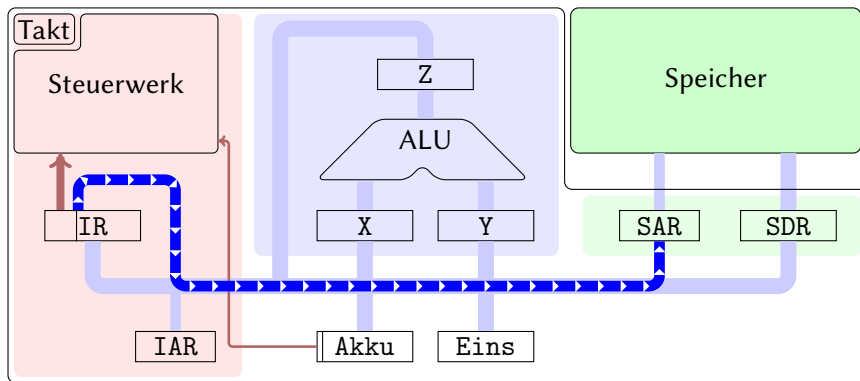


- Steuerwerk liest
 - die 8 Bits des IR,
die den Befehl codieren
 - das höchstwertige Bit des Akku
- verzweigt abhängig von diesen Bits
zum passenden Mikroprogramm



- abhängig vom konkreten Maschinenbefehl
- betrachte zwei Beispiele
 - *LDV adr*
 - *JMP adr*
- nach der Ausführungsphase wieder Holphase
 - für alle Maschinenbefehle gleich

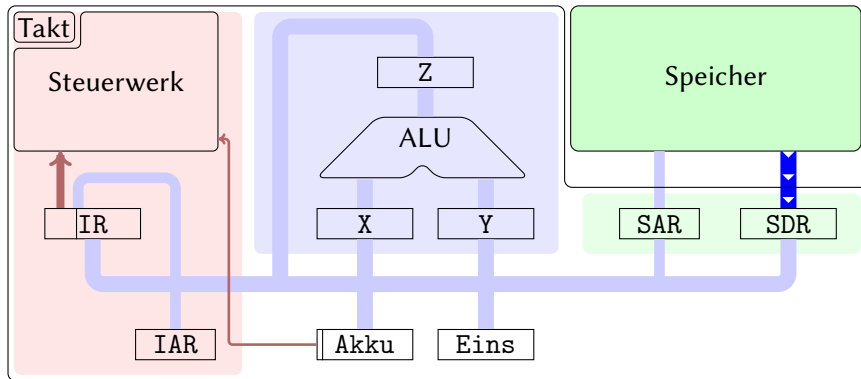
MIMA Befehlsausführungsphase für LDV *adr* (1)



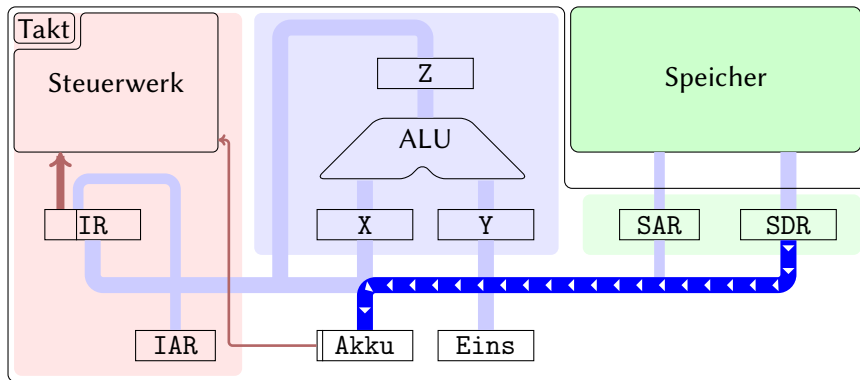




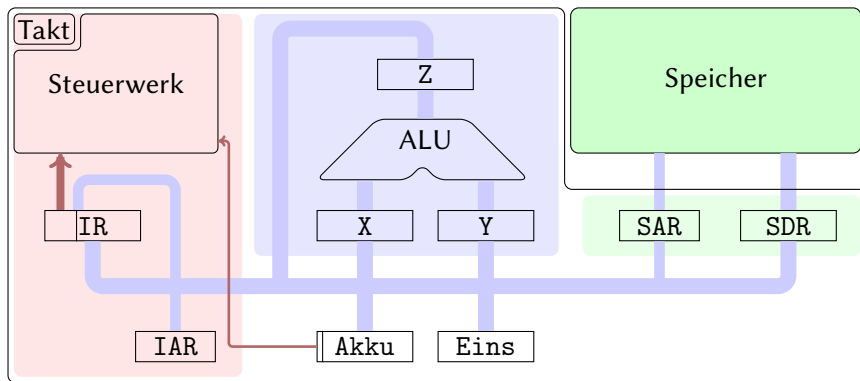
MIMA Befehlsausführungsphase für LDV *adr* (4)



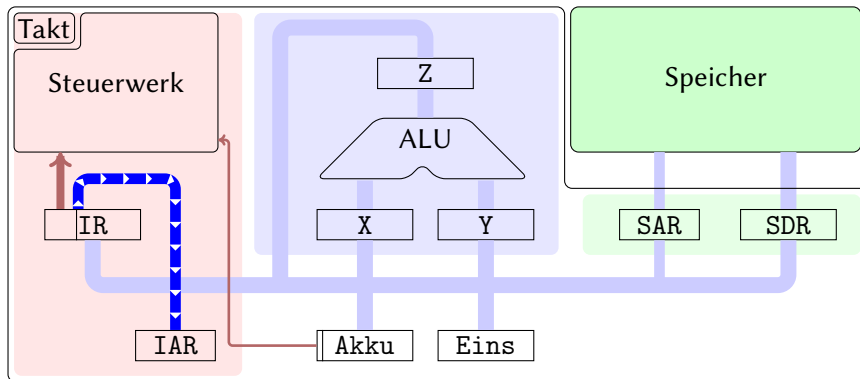
MIMA Befehlsausführungsphase für LDV *adr* (5)



MIMA Befehlsausführungsphase für JMP *adr*



MIMA Befehlsausführungsphase für JMP *adr*



Wo sind wir?

Einfache „Hardware“-„Bausteine“

Grobstruktur der MIMA

Maschinenbefehle der MIMA

Mikroprogrammsteuerung der MIMA

Ein Beispielprogramm

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“

cnt:
ref:
sum:

- Notation
 - symbolische Namen für Adressen
 - nur die, die man braucht
- Speicher vertikal dargestellt
 - von oben nach unten aufsteigende Adressen
- Problemstellung
 - gegeben:
 - Liste von Werten ab Adresse *list* und
 - ihre Länge, gespeichert an Adresse *len*
 - Ziel:

$$M(\textit{sum}) = \sum_{i=0}^{M(\textit{len})-1} M(\textit{list}[i])$$

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“

cnt:

ref:

sum:

Akku

Programmspeicher

start: LDC 0
STV *sum*
STV *cnt*
LDC *list*
STV *ref*

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“

cnt:

ref:

sum:

Akku 0

Programmspeicher

start: LDC 0
STV *sum*
STV *cnt*
LDC *list*
STV *ref*

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“

cnt:

ref:

sum: 0

Akku 0

Programmspeicher

start: LDC 0
STV *sum*
STV *cnt*
LDC *list*
STV *ref*

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 0
ref:
sum: 0

Akku 0

Programmspeicher

start: LDC 0
STV *sum*
STV *cnt*
LDC *list*
STV *ref*

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 0
ref:
sum: 0

Akku *list*

Programmspeicher

start: LDC 0
STV *sum*
STV *cnt*
LDC *list*
STV *ref*

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 0
ref: *list*
sum: 0

Akku *list*

Programmspeicher

start: LDC 0
STV *sum*
STV *cnt*
LDC *list*
STV *ref*

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 0
ref: *list*[0]
sum: 0

Akku

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 0
ref: list[0]
sum: 0

Akku 11

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 0
ref: list[0]
sum: 0

Akku 11

Programmspeicher

next: LDIV ref
ADD sum
STV sum

LDC 1
ADD cnt
STV cnt

LDC 1
ADD ref
STV ref

LDV cnt
EQL len
JMN done
JMP next

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 0
ref: list[0]
sum: 11
Akku 11

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 0
ref: *list*[0]
sum: 11

Akku 1

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 0
ref: *list*[0]
sum: 11

Akku 1

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[0]
sum: 11
Akku 1

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[0]
sum: 11
Akku 1

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 1
ref: *list*[0]
sum: 11
Akku *list*[1]

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: *list[1]*
sum: 11

Akku *list[1]*

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV ref

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 11

Akku 1

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 11

Akku 0

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 11

Akku 0

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 11
Akku 0

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 11
Akku 22

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 11
Akku 33

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 33
Akku 33

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 1
ref: list[1]
sum: 33

Akku 1

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 1
ref: *list*[1]
sum: 33
Akku 2

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[1]
sum: 33
Akku 2

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[1]
sum: 33
Akku 1

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 2
ref: *list*[1]
sum: 33
Akku *list*[2]

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 2
ref: *list*[2]
sum: 33
Akku *list*[2]

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV ref
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[2]
sum: 33

Akku 2

Programmspeicher

next: LDIV ref
ADD sum
STV sum

LDC 1
ADD cnt
STV cnt

LDC 1
ADD ref
STV ref

LDV cnt
EQL len
JMN done
JMP next

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[2]
sum: 33
Akku 0

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 2
ref: *list*[2]
sum: 33
Akku 0

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[2]
sum: 33
Akku 0

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[2]
sum: 33
Akku 33

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*
LDC 1
ADD *cnt*
STV *cnt*
LDC 1
ADD *ref*
STV *ref*
LDV *cnt*
EQL *len*
JMN *done*
JMP *next*
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 2
ref: list[2]
sum: 33
Akku 66

Programmspeicher

next: LDIV ref
ADD sum
STV sum
LDC 1
ADD cnt
STV cnt
LDC 1
ADD ref
STV ref
LDV cnt
EQL len
JMN done
JMP next
done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „list[0]“
22 „list[1]“
33 „list[2]“
44 „list[3]“
55 „list[4]“
cnt: 3
ref: list[3]
sum: 66

Akku 0

Programmspeicher

next: LDIV ref
ADD sum
STV sum

LDC 1
ADD cnt
STV cnt

LDC 1
ADD ref
STV ref

LDV cnt
EQL len
JMN done
JMP next

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 4
ref: *list*[4]
sum: 110

Akku 0

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 5
ref: *list*[5]
sum: 165

Akku -1

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

Aufsummieren — Iteration über die Elemente

Datenspeicher

len: 5
list: 11 „*list*[0]“
22 „*list*[1]“
33 „*list*[2]“
44 „*list*[3]“
55 „*list*[4]“
cnt: 5
ref: *list*[5]
sum: 165

Akku -1

Programmspeicher

next: LDIV *ref*
ADD *sum*
STV *sum*

LDC 1
ADD *cnt*
STV *cnt*

LDC 1
ADD *ref*
STV *ref*

LDV *cnt*
EQL *len*
JMN *done*
JMP *next*

done: HALT

- **Das sollten Sie mitnehmen:**
 - Grobstruktur
 - Steuerwerk, Rechenwerk, Speicherwerk
 - Maschinenbefehle
 - Mikroprogramme
- **Das sollten Sie üben:**
 - einfache ALU-Rechnungen
 - ganz einfache Maschinenprogramme