

4. Prozessorverhalten und -struktur

Inhalt

- 4.1 Befehlsablauf und Datenpfad der seriellen DLX**
- 4.2 Mehrfachnutzung und Auslastungsgrad**
- 4.3 Pipelining-Grundlagen**
- 4.4 Überlappung von Befehlshol- und -ausführungsphase**
- 4.5 Pipelining der DLX**
- 4.6 Unterbrechungen**

Lernziele von Kap. 4:

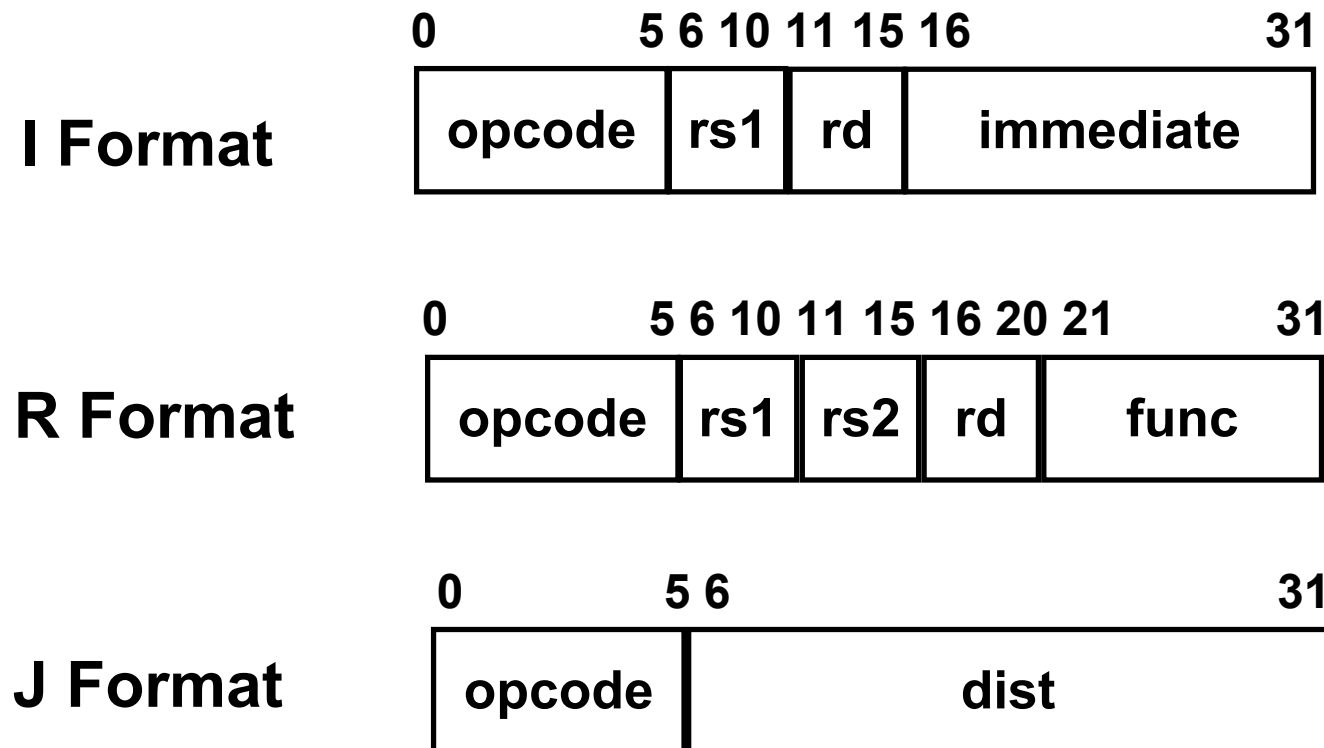
- **Beziehung zwischen Verhalten und Struktur bei einem Prozessor verstehen**
- **wissen, wie und mit welchen Hilfsmitteln die serielle und die Fließbandverarbeitung von Maschinenbefehlen realisiert werden kann**
- **Pipeliningkonflikte und ihre Lösungsmöglichkeiten kennen**
- **Techniken zur Sprungvoraussage kennen**
- **Konzepte: Auslastungsgrad, Pipelining, Pipeline-Register, Bernstein'sche Regeln, Forwarding, RAW/WAR/WAW-Hazards, *delayed branch*, Sprungvoraussage, interne und externe Unterbrechung**

4.1 Befehlsablauf und Datenpfad der seriellen DLX

- Die Maschinenbefehle der DLX werden in maximal 5 Schritten, den **Befehlsverarbeitungsphasen**, verarbeitet:
 - **IF** (*instruction fetch*): Holen des Befehls
 - **ID** (*instruction decode*): Dekodierung des Befehls
 - **EX** (*instruction execute*): Befehlsausführung
 - **MEM** (*memory*): Speicherzugriff (Lesen oder Schreiben)
 - **WB** (*write back*): Zurückschreiben in den Registersatz

4.1 Befehlsablauf und Datenpfad der DLX

- Nochmals die Befehlsformate der DLX:



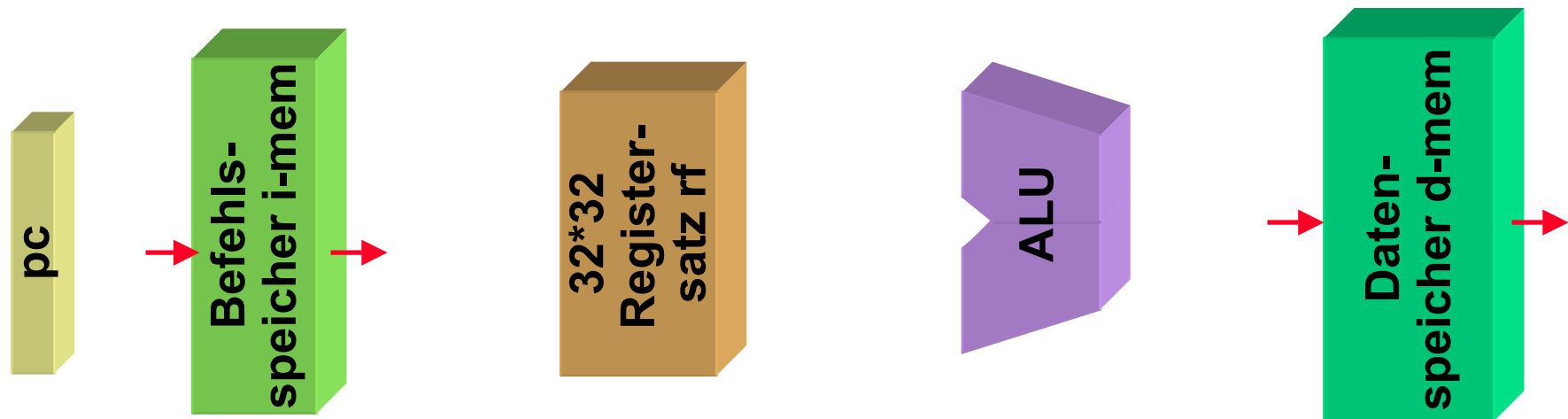
4.1 Befehlsablauf und Datenpfad der DLX

- Aufgabe ist es, die **Abläufe** der Befehlsverarbeitung und gleichzeitig den **Datenpfad** zu entwerfen
- für die Darstellung der Abläufe wird eine Tabellenform gewählt, bei der die Befehlsphasen von links nach rechts durchlaufen werden:

IF:	ID:	EX:	MEM:	WB:
-----	-----	-----	------	-----

4.1 Befehlsablauf und Datenpfad der DLX

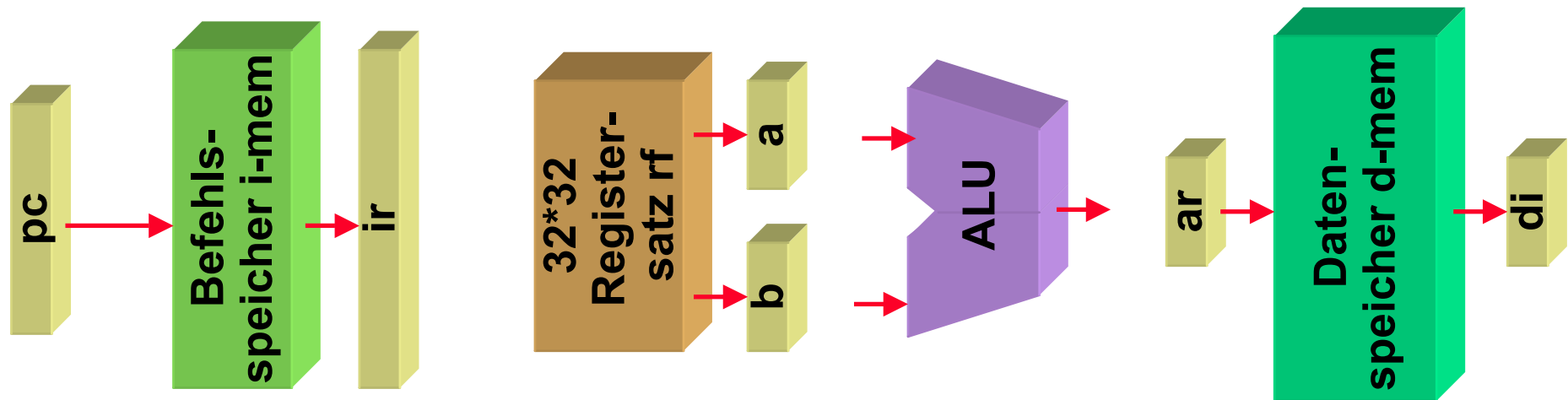
- Für die Darstellung des Datenpfads werden Blockschaltbilder benutzt
- die wesentlichen Bestandteile sind der Befehlszähler pc, der Befehlsspeicher i-mem, der Registersatz, die ALU, der Datenspeicher d-mem



4.1 Befehlsablauf und Datenpfad der DLX

- Zur Zwischenspeicherung werden zusätzliche Register eingeführt, u.a.:
 - das **ir**-Register für ausgelesene Befehle
 - zwei Register **a** und **b** für aus dem Registersatz ausgelesene Daten
 - ein Adressregister **ar** für die Adressierung des Datenspeichers
 - ein Pufferregister **di** für aus dem Datenspeicher ausgelesene Daten
- ➡ zusätzliche Register werden bei Bedarf eingeführt werden

4.1 Befehlsablauf und Datenpfad der DLX



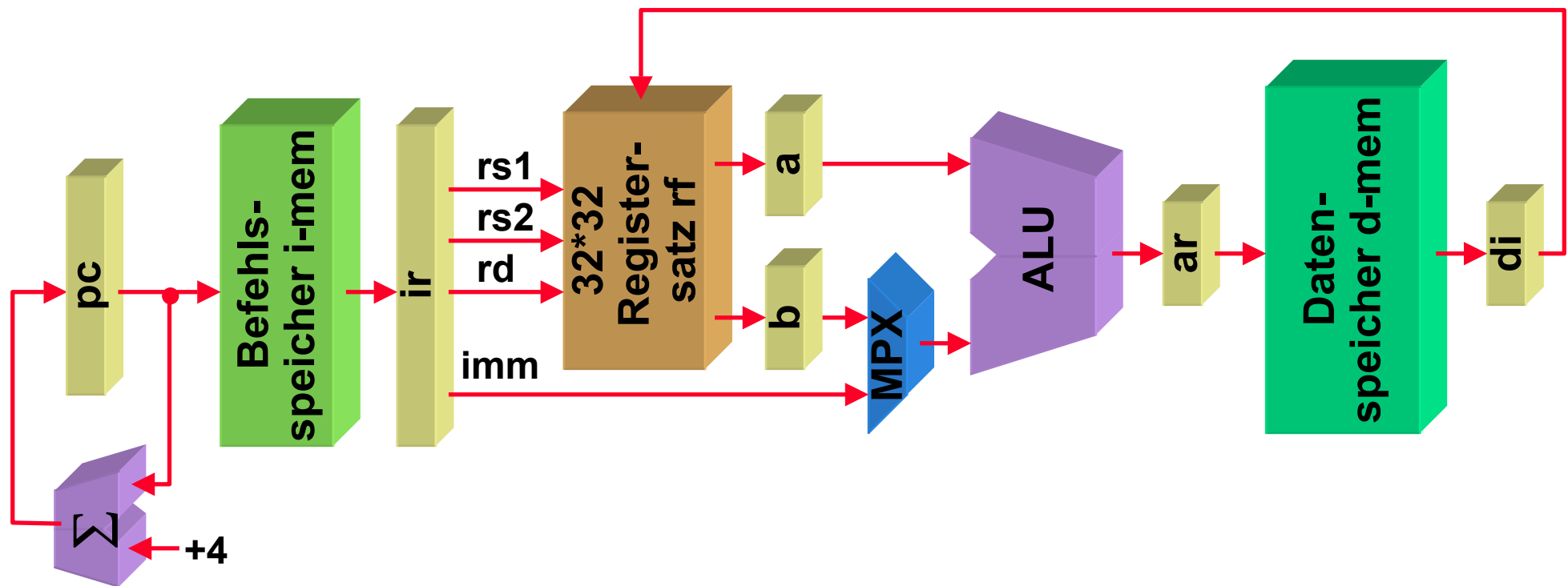
4.1 Befehlsablauf und Datenpfad der DLX

- Um die Anzahl der Entwurfsalternativen einzuschränken, werden **weitere Annahmen** über die Hardware getroffen
 - ☞ derartige Festlegungen bedeuten die Wahl einer **Zielarchitektur**, d.h. eine Festlegung auf eine Klasse von möglichen Realisierungen
- **Annahme**: der Registersatz erlaubt gleichzeitig zwei Leseoperation mit unabhängigen Adressen
- **Annahme**: in einem Takt können folgende Operationen parallel (aber nicht seriell) ausgeführt werden:
 - Register → ALU → Register
 - Register → Registersatz/Speicher
 - Registersatz/Speicher → Register
 - Register → Register
 - ☞ diese Annahmen dienen zur Balancierung des Zeitbedarfs der Befehlsphasen

4.1 Befehlsablauf und Datenpfad der DLX

LW

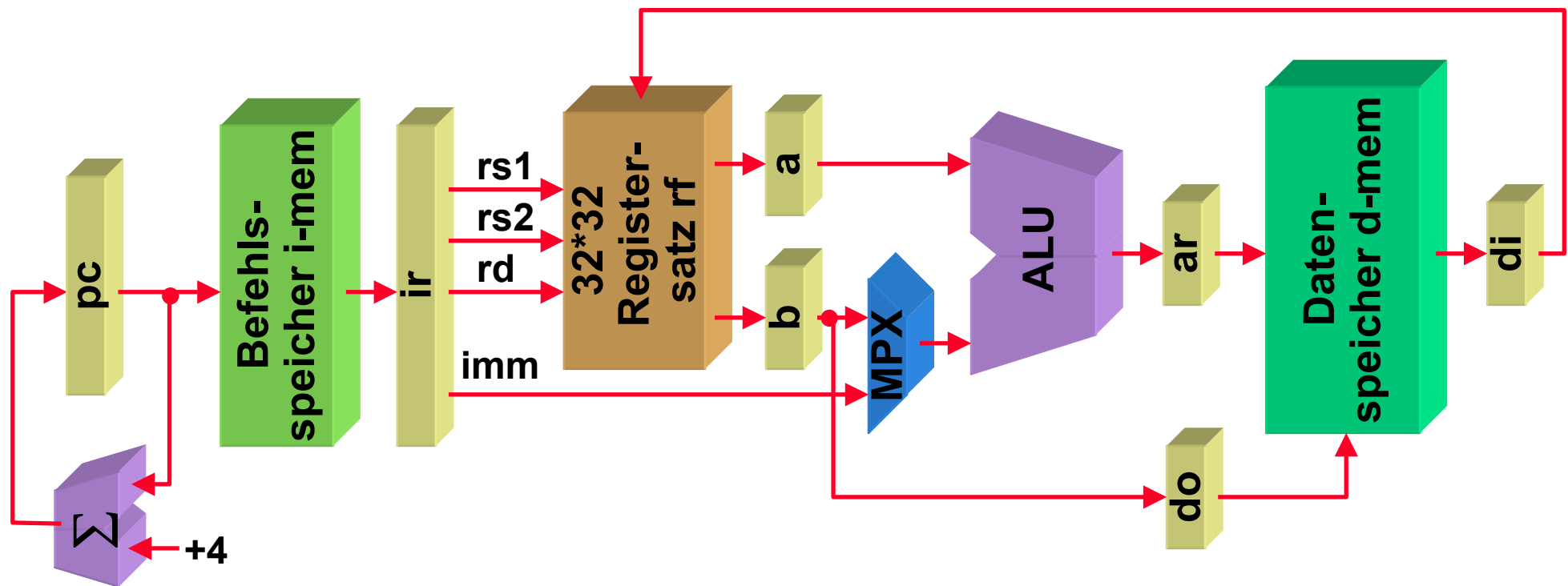
IF: $ir \leq i\text{-mem}[pc]$, $pc \leq pc + 4$	ID: $a \leq rf[ir[6:10]]$, $b \leq rf[ir[11:15]]$	EX: $ar \leq a + ir[16:31]$	MEM: $di \leq d\text{-mem}[ar]$	WB: $rf[ir[11:15]] \leq di$
---	--	--------------------------------	------------------------------------	--------------------------------



4.1 Befehlsablauf und Datenpfad der DLX

SW

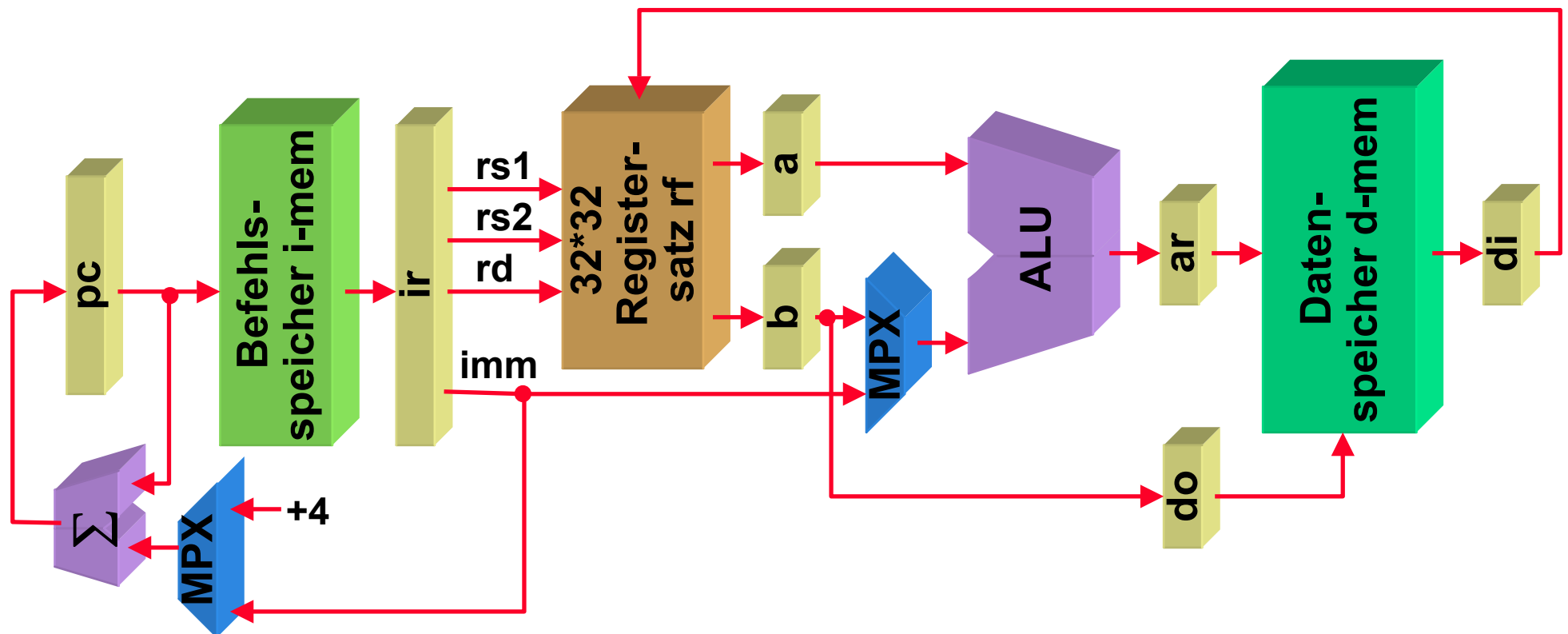
IF: $ir \leq i\text{-mem}[pc]$, $pc \leq pc + 4$	ID: $a \leq rf[ir[6:10]]$, $b \leq rf[ir[11:15]]$	EX: $ar \leq a + ir[16:31]$, $do \leq b$	MEM: $d\text{-mem}[ar] \leq do$	
---	--	---	------------------------------------	--



4.1 Befehlsablauf und Datenpfad der DLX

J

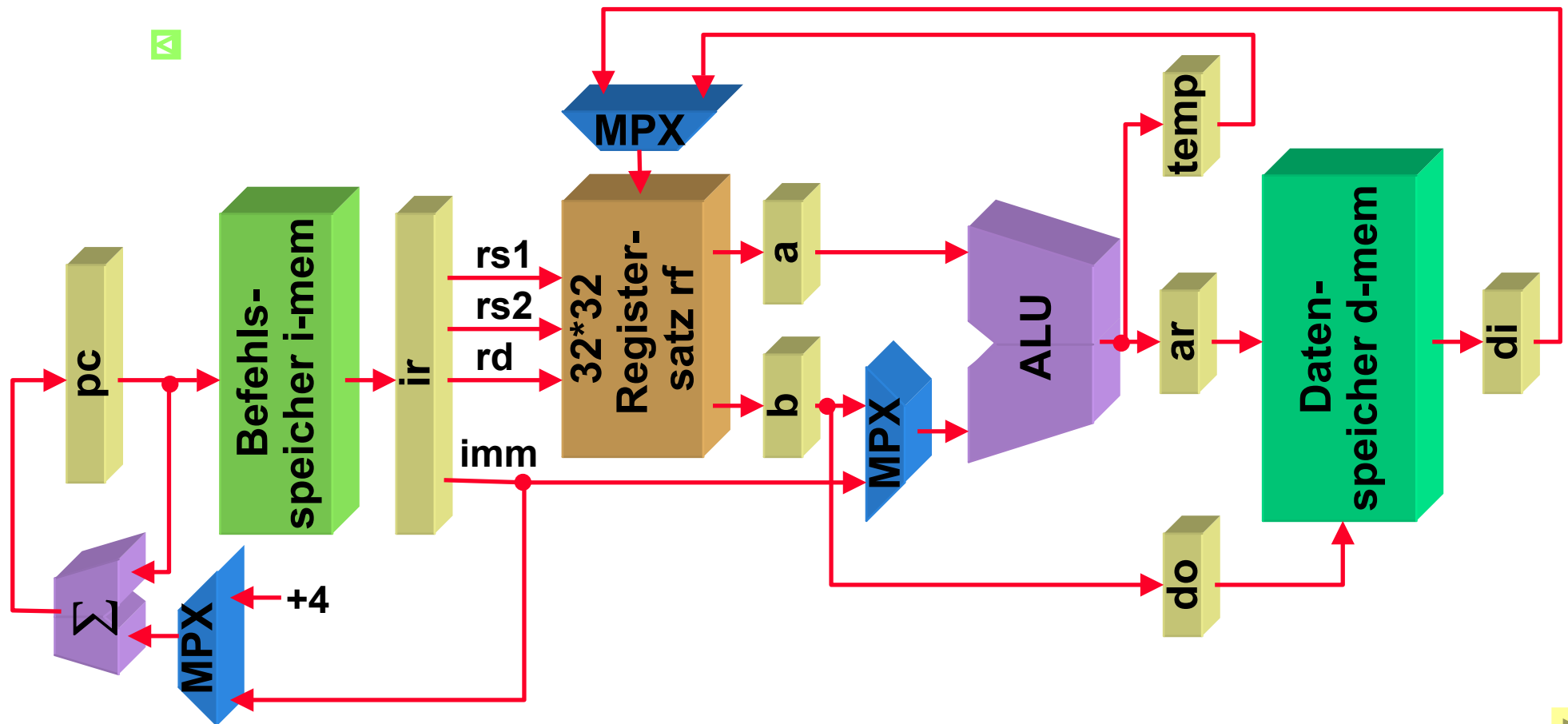
IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: pc<= pc+ir[6:31]		
-----------------------------------	---	----------------------------	--	--



4.1 Befehlsablauf und Datenpfad der DLX

ADD

IF: ir≤i-mem[pc], pc≤pc+4	ID: a≤rf[ir[6:10]], b≤rf[ir[11:15]]	EX: temp≤a+b	WB: rf[ir[16:20]]≤ temp	
---------------------------------	---	-----------------	-------------------------------	--



4.1 Befehlsablauf und Datenpfad der DLX

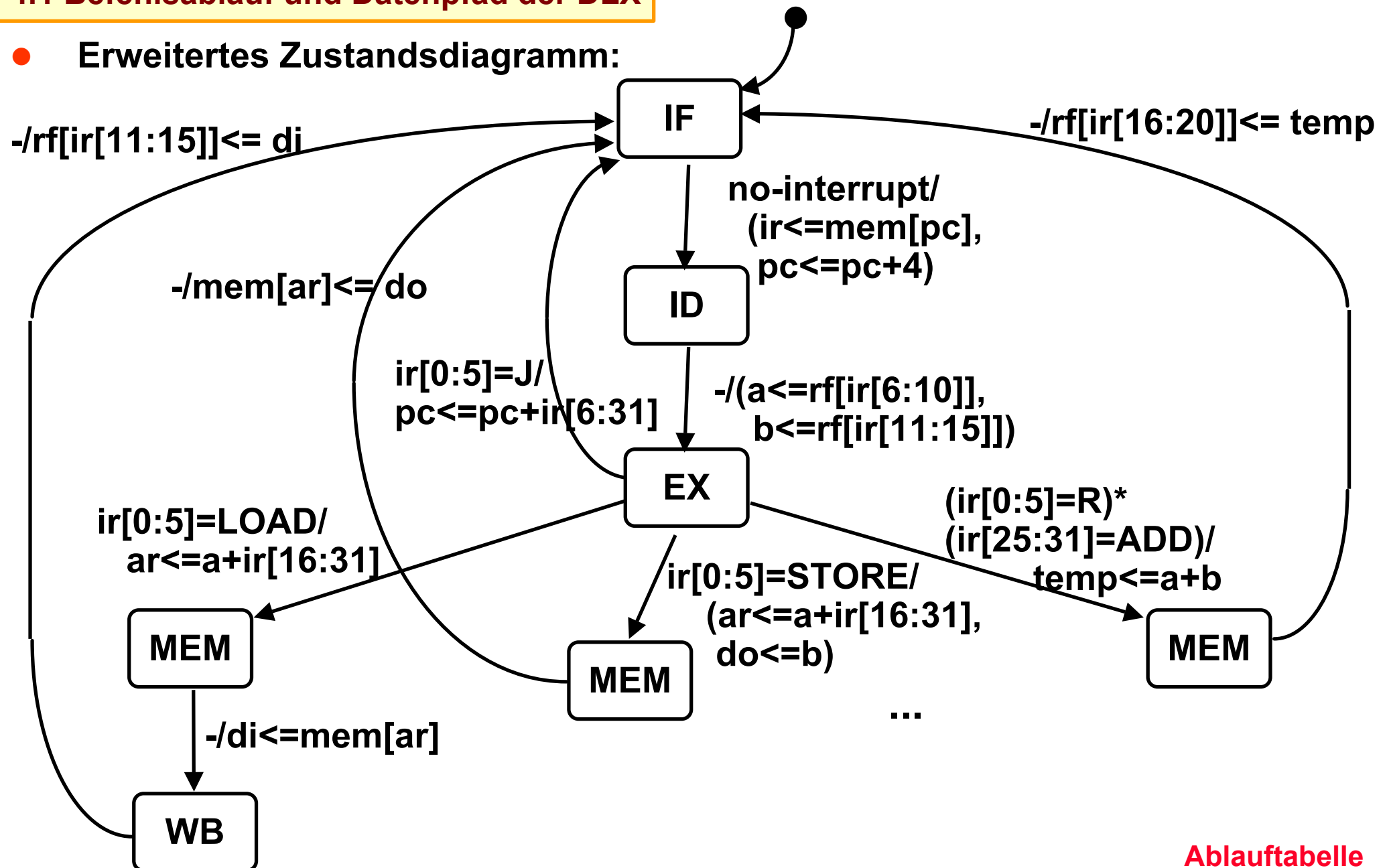
- Ablauftabelle:

LW	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: ar<=a+ir[16:31]	MEM: di<= d-mem[ar]	WB: rf[ir[11:15]]<= di
SW	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: ar<=a+ir[16:31], do<=b	MEM: d-mem[ar]<= do	
ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b	WB: rf[ir[16:20]]<= temp	
J	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: pc<= pc+ir[6:31]		
BEQZ	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: if a=0 then pc<=pc+ ir[16:31]		



4.1 Befehlsablauf und Datenpfad der DLX

- Erweitertes Zustandsdiagramm:



4.2 Mehrfachnutzung und Auslastungsgrad

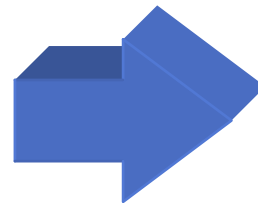
- Bei der Diskussion der Ausführung der DLX-Befehle mussten zwei Dinge berücksichtigt werden:
 - das (in der Tabelle) spezifizierte Verhalten
 - die Struktur, die die Hardware-Hilfsmittel festlegt
- das Verhalten legt den zeitlichen Ablauf und damit die **Geschwindigkeit** eines Vorgangs fest
- die Struktur bestimmt über die benutzten Verarbeitungseinheiten die **Kosten**

4.2 Mehrfachnutzung und Auslastungsgrad

- Dies ist ein Beispiel für die allgemeine Problematik der Beziehung zwischen einem zeitlichen **Vorgang** einerseits und **Verarbeitungseinheiten**, also strukturellen Komponenten andererseits:

Vorgang

Prozesse
Datenoperationen
Datentransport
Datenspeichern/-lesen



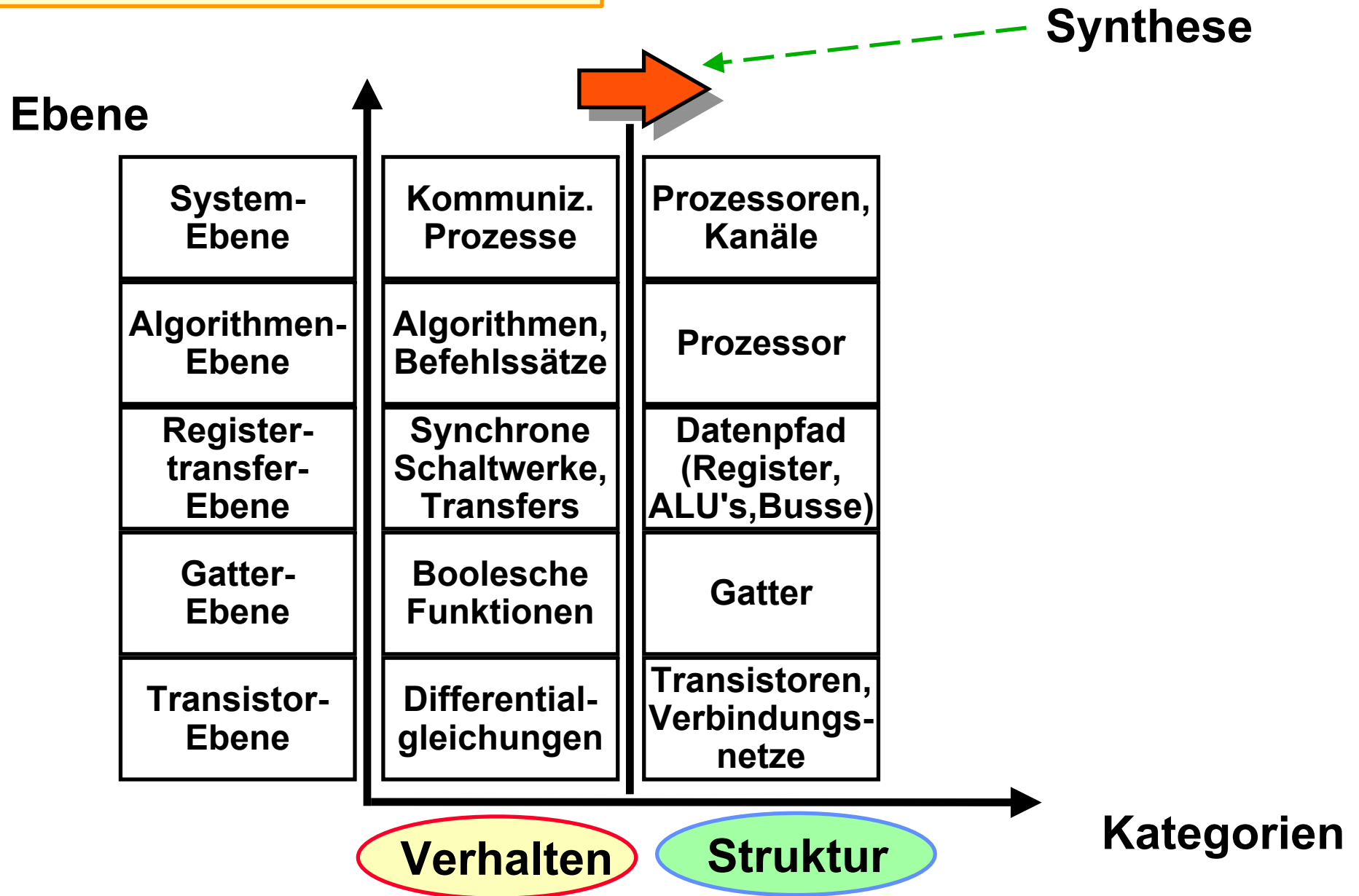
Verarbeitungseinheiten

Prozessoren
Funktionsblöcke
Busse
Register

4.2 Mehrfachnutzung und Auslastungsgrad

- Beispiele für Beziehungen zwischen Vorgängen (**Verhalten**) und Verarbeitungseinheiten (**Struktur**):
 - Operationen in Anweisungen wie "+" und Funktionsblöcke wie ALU's
 - Prozesse (Tasks) und Prozessoren
 - Datentransporte und physikalische Verbindungen (Busse)
- Beispiele für Zuordnungsprobleme:
 - **Konstruktion einer Struktur** aus einer Beschreibung des Verhaltens (Synthese)
 - Realisierung von Operationen mit den Hilfsmitteln einer **bereits existierenden Struktur**

4.2 Mehrfachnutzung und Auslastungsgrad



4.2 Mehrfachnutzung und Auslastungsgrad

- Ein wichtiger Gesichtspunkt ist die **Mehrfachnutzung** kostenverursachender Verarbeitungseinheiten (**resource sharing**)
- der **Auslastungsgrad** eines Hilfsmittels ist definiert als Quotient der Zeit, in der das Hilfsmittel benutzt wird, zur Gesamtzeit
 - ➡ der Auslastungsgrad ist bei serieller Befehlsverarbeitung oft nur 30%, bei Pipelining → 100%
- Voraussetzung der Mehrfachnutzung z.B. der ALU ist der Aufbau eines Busses vor dem B-Eingang der ALU:

ADD

IF:
ir≤i-mem[pc],
pc≤pc+4

ID:
a≤rf[ir[6:10]],
b≤rf[ir[11:15]]

EX:
temp≤a+b

WB:
rf[ir[16:20]]≤
temp

LW

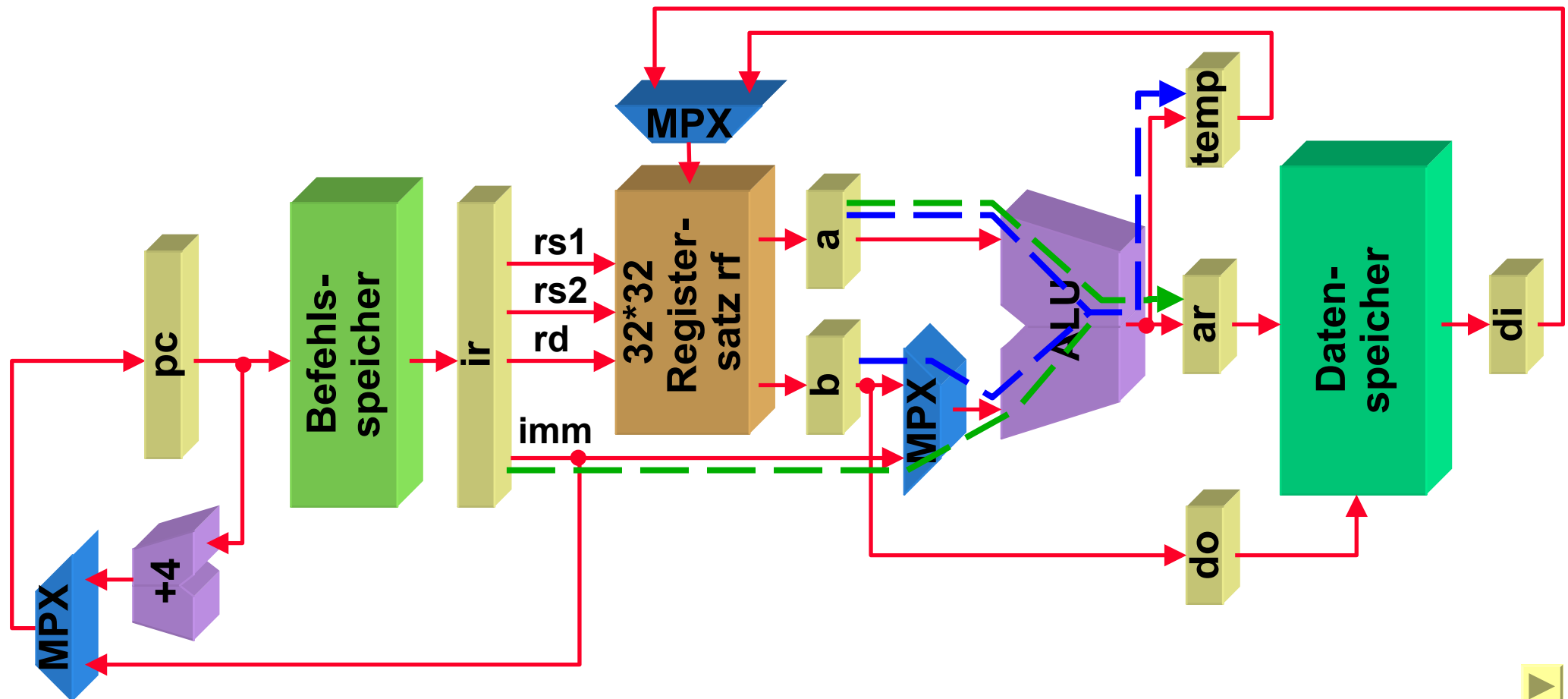
IF:
ir≤i-mem[pc],
pc≤pc+4

ID:
a≤rf[ir[6:10]],
b≤rf[ir[11:15]]

EX:
ar≤a+ir[16:31]

MEM:
di≤
d-mem[ar]

WB:
rf[ir[11:15]]≤
di

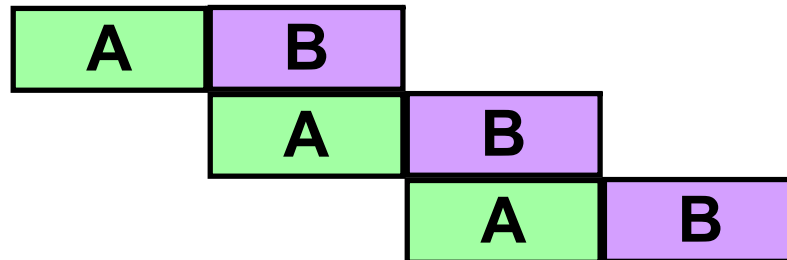
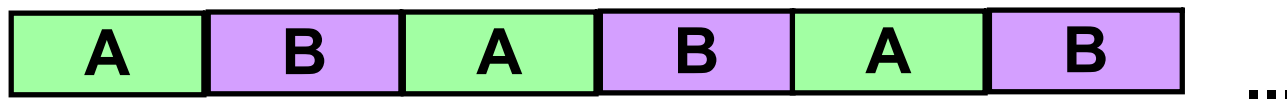


4.2 Mehrfachnutzung und Auslastungsgrad

- **Die Operationen (z.B. Registertransfers) und die Struktur (z.B. der Datenpfad) hängen eng zusammen**
 - **der Datenpfad definiert die möglichen Operationen in Bezug auf**
 - **Quellen und Ziele**
 - **mögliche Parallelität mehrerer Operationen**
 - **ein Hilfsmittel des Datenpfads (Speicher, Funktionsblock, Bus) kann zu einem Zeitpunkt jeweils nur für eine Operation benutzt werden**
 - **ein Hilfsmittel des Datenpfads kann andererseits zu unterschiedlichen Zeitpunkten auch für unterschiedliche Operationen benutzt werden**

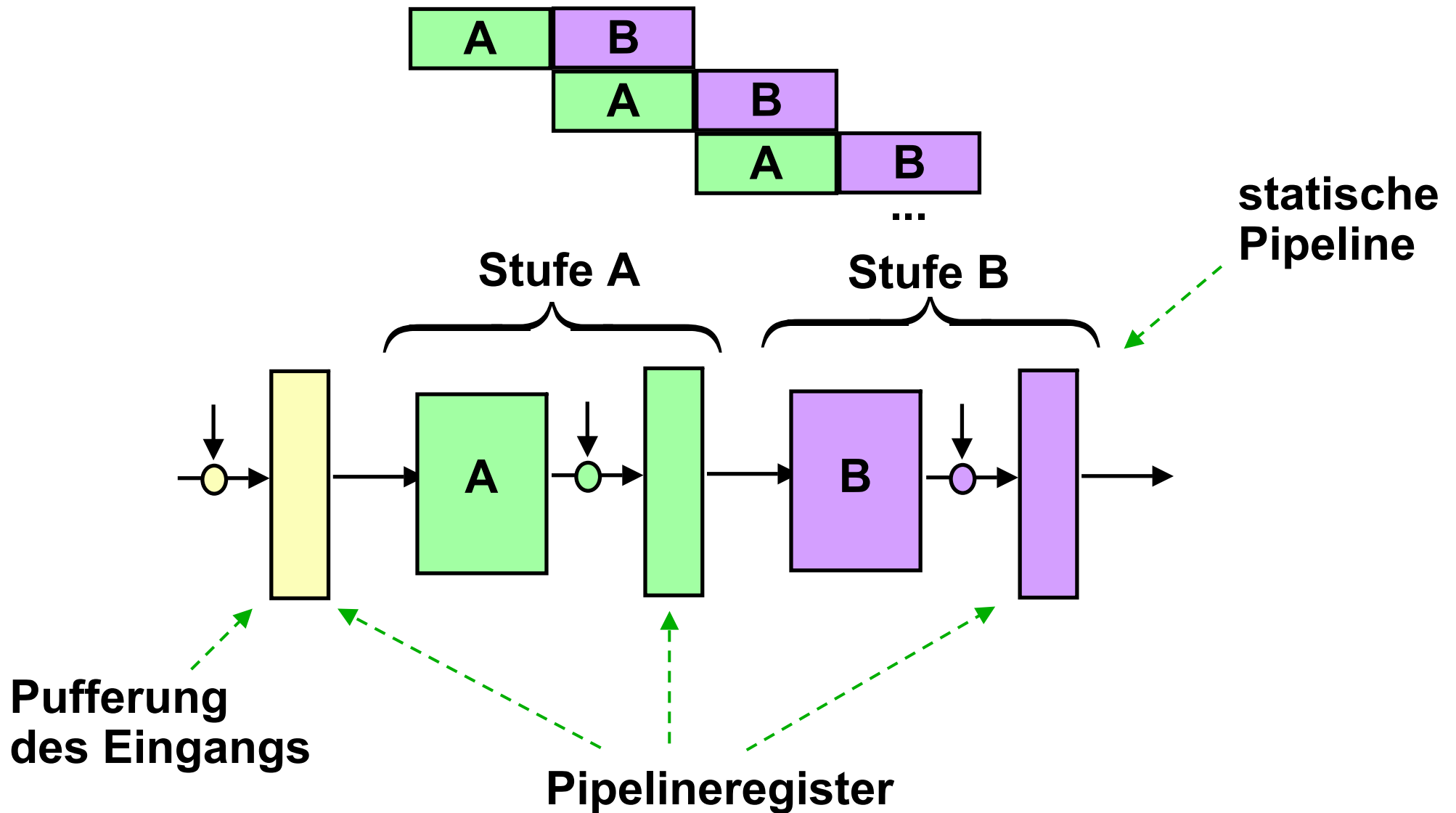
4.3 Pipelining-Grundlagen

- **Pipelining** (Fließbandprinzip) ist ein wichtiges Organisationsprinzip, um den Auslastungsgrad zu erhöhen
- weite Verbreitung in Prozessoren, digitaler Signalverarbeitung, ...
- mehrere Verarbeitungsvorgänge (Schleifendurchläufe, Prozesse, ...) sind gleichzeitig **parallel** aktiv




4.3 Pipelining-Grundlagen

- **Pipelinstufen**

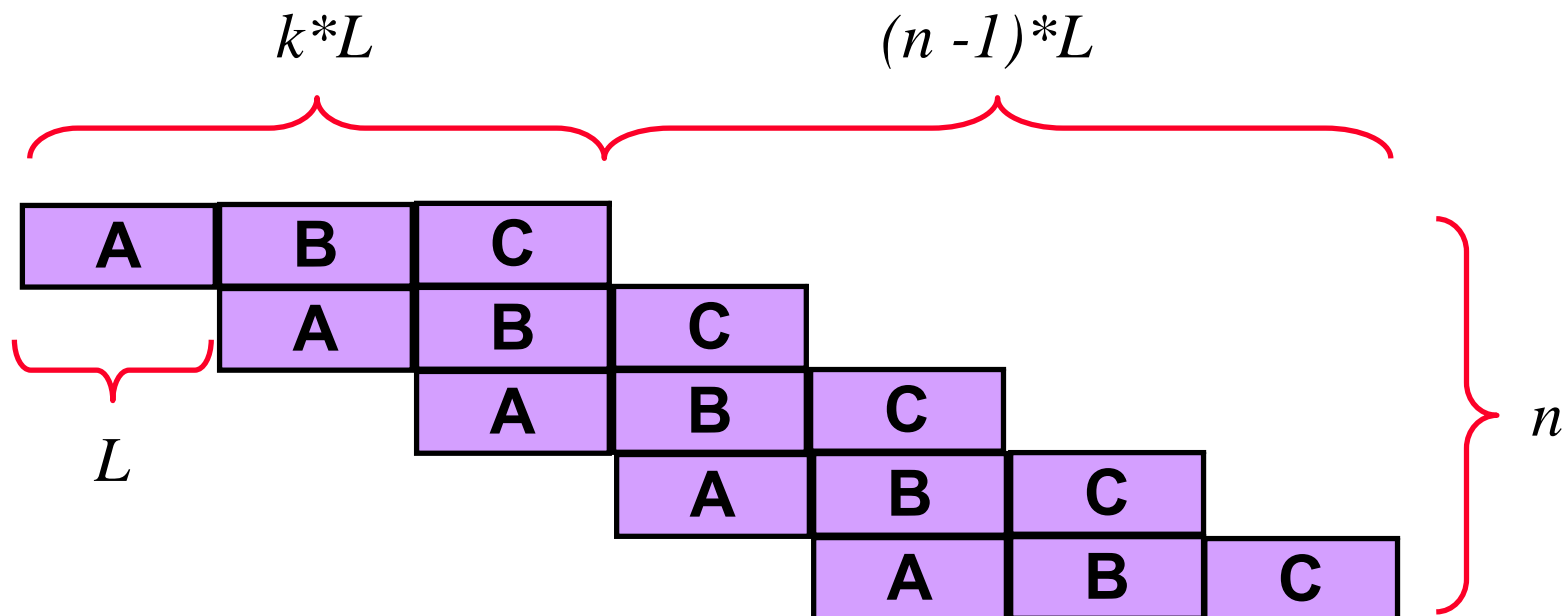


4.3 Pipelining-Grundlagen

- Eine statische Pipeline zerlegt einen kontinuierlichen Datenfluß in eine Reihe von Schritten
-  Pipelining **erhöht den Auslastungsgrad** der Stufen

4.3 Pipelining-Grundlagen

- **Latenzzeit** L : Zeit zwischen dem Beginn zweier Verarbeitungsvorgänge
- **Gesamtverarbeitungszeit in einer Pipeline:**
Annahme: k Stufen, n Tasks, L Latenzzeit,
Gesamtverarbeitungszeit: $(k+n-1)*L$



4.3 Pipelining-Grundlagen

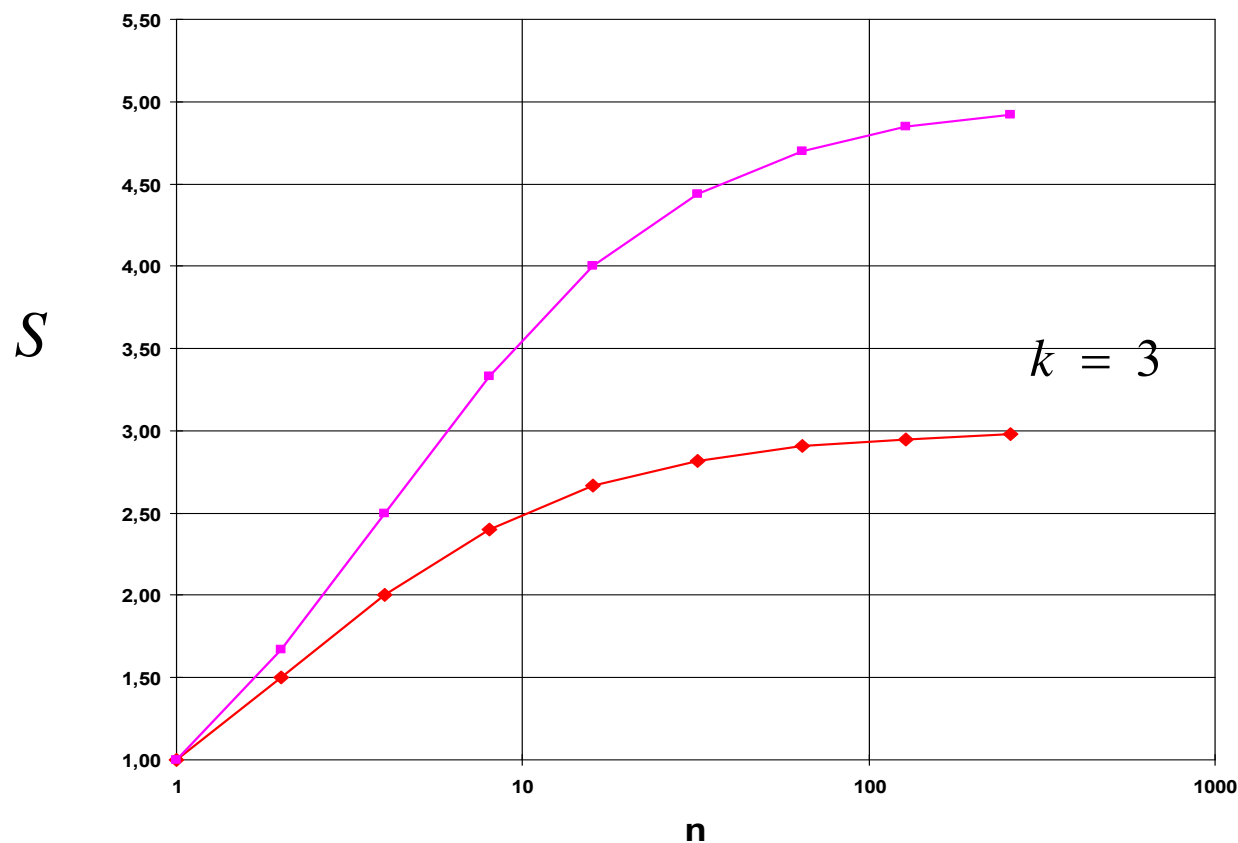
- **Durchsatz:** Anzahl der ausgeführten Operationen (z.B. Maschinenbefehle) pro Zeit
- bei einer eingeschwungenen Pipeline ist der Durchsatz $1/L$

4.3 Pipelining-Grundlagen

● Beschleunigung durch Pipelining

$$S = \frac{n * k * L}{(k + n - 1) * L} = \frac{n * k}{k + n - 1} \xrightarrow{n \rightarrow \infty} k$$

$k = 5$



4.3 Pipelining-Grundlagen

- **Beispiel: RISC-Prozessor, fünfstufige Pipeline, 20% der Befehle Verzweigungsbefehle, d.h. im Mittel wird nach jedem vierten Befehl der Kontrollfluß geändert.**
Falls die Pipeline-Verarbeitung bei jedem dieser Befehle neu begonnen werden muß, ergibt sich für die erreichte Beschleunigung statt des Faktors 5 nur

$$S = \frac{n * k}{k + n - 1} = \frac{4 * 5}{5 + 4 - 1} = \frac{20}{8} = 2,5$$

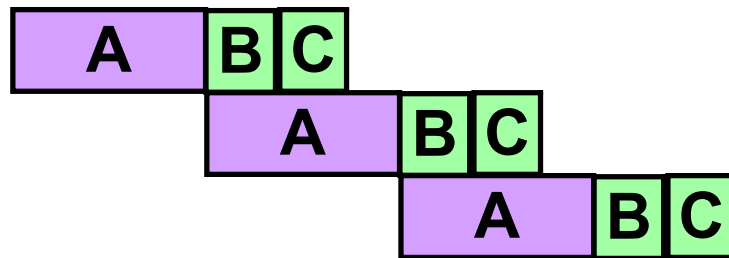
4.3 Pipelining-Grundlagen

- Um Pipelining wirkungsvoll anwenden zu können, müssen eine Reihe von **Voraussetzungen** erfüllt sein:
 1. die **Anzahl** der nacheinander zu verarbeitenden Schritte muß **hoch** sein, da das Füllen der Pipeline aufwendig ist
 2. die Pipeline muß **füllbar** sein
 - Beispiel: Speicherzugriffszeit : Prozessortaktzeit z.B. 10...2 : 1. Um in jedem Takt eine Instruktion neu beginnen zu können, sind Daten- und Instruktions-Caches (Harvard-Architektur) mit Zugriffszeit = Prozessortaktzeit notwendig. Das heißt, daß es unsinnig ist, einen Pipeline-Prozessor ohne Caches zu bauen.

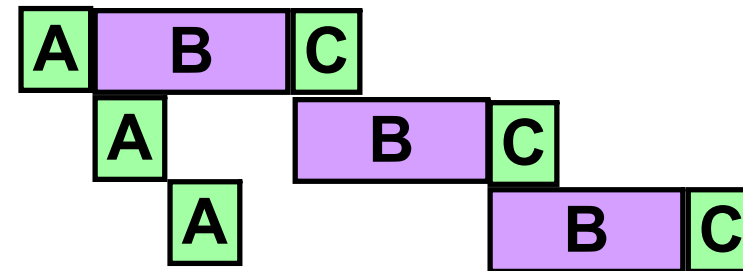
4.3 Pipelining-Grundlagen

3. die Pipeline-Stufen sollten **balancierte Verarbeitungszeiten** haben

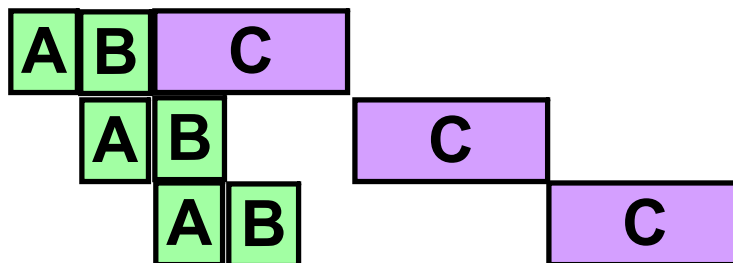
☞ die Gesamtverarbeitungszeit wird von der langsamsten Stufe bestimmt



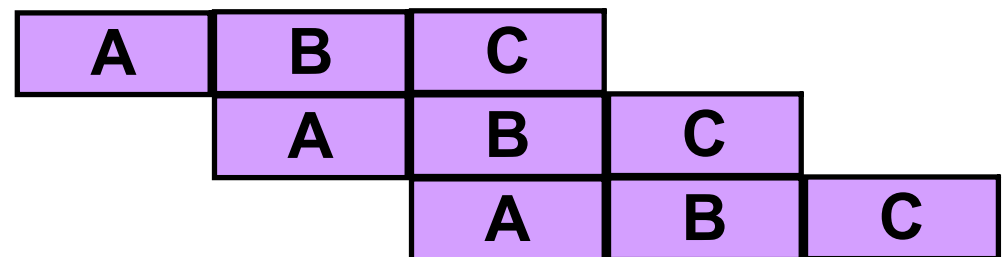
...



...



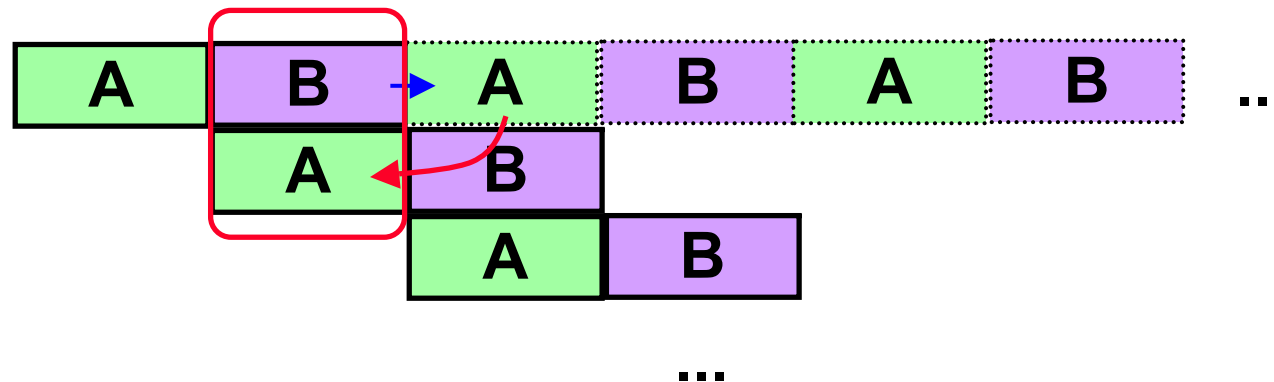
...



...

4.3 Pipelining-Grundlagen

- Pipelining bedeutet immer die **Parallelisierung** von Vorgängen
- dabei ist auf innere Abhängigkeiten der Teilvorgänge zu achten
 - Beispiel: B produziert ein **Ergebnis**, das von A des nächsten Durchlaufs benötigt wird
- Frage: unter welchen Bedingungen können Teilvorgänge parallelisiert werden?



4.3 Pipelining-Grundlagen

- Diese Frage ist besonders einfach für **lineare Sequenzen von Transfers** beantwortbar
 - **Transfers** werden wie üblich mit der Notation **(Ziel <= Ausdruck)** notiert
 - **synchron parallele Transfers** werden durch **Komma** getrennt umschlossen
 - die **sequentielle Komposition** wird mit **;** beschrieben
 - Beispiel:

```
(x1<= a * b);  
(x2<= c + d);  
(x4<= x1 + x2, x5<= a * d);  
(x6<= x4 * x5);
```

4.3 Pipelining-Grundlagen

- Beispiel: ist es erlaubt, den Transfer $x5 \leftarrow a * d$ zwei Schritte vorzuziehen und mit dem Transfer $x1 \leftarrow a * b$ zu parallelisieren?

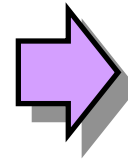
```
(x1 ← a * b);  
(x2 ← c + d);  
(x4 ← x1 + x2, x5 ← a * d);  
(x6 ← x4 * x5);
```

```
(x1 ← a * b, x5 ← a * d);  
(x2 ← c + d);  
(x4 ← x1 + x2);  
(x6 ← x4 * x5);
```

4.3 Pipelining-Grundlagen

- **Bernstein'sche Regeln (1966)**

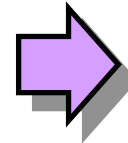
D1 ≤ S1;
(D2 ≤ S2, D3 ≤ S3);



(D1 ≤ S1, D2 ≤ S2);
D3 ≤ S3;

falls $D1 \cap D2 = \emptyset$, $D1 \cap S2 = \emptyset$, $D2 \cap S3 = \emptyset$

(D1 ≤ S1, D2 ≤ S2);
D3 ≤ S3;



D1 ≤ S1;
(D3 ≤ S3, D2 ≤ S2);

falls $D1 \cap S2 = \emptyset$, $D2 \cap D3 = \emptyset$, $D2 \cap S3 = \emptyset$

☞ die Bernstein'schen Regeln beziehen sich auf das Verschieben von Transfers, **ohne diese zu ändern** (z.B. durch Substitution von Ausdrücken für Variable, usw.)

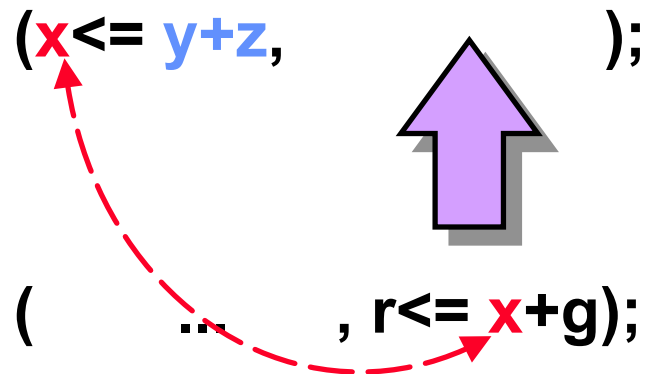
4.3 Pipelining-Grundlagen

- Beim Pipelining sind die Bernstein'schen Bedingungen **oft nicht** erfüllt
- es gibt aber **2 Techniken**, um die Parallelisierung beim Pipelining dennoch zu ermöglichen:
 - **Forwarding** und
 - **Einführen von Pipelineregistern**

4.3 Pipelining-Grundlagen

1. Forwarding:

(**x** ≤ y+z,);
(..., r ≤ **x**+g);



Verschieben verboten durch
Bernstein'sche Regeln !

(x ≤ y+z, r ≤ y+z+g);

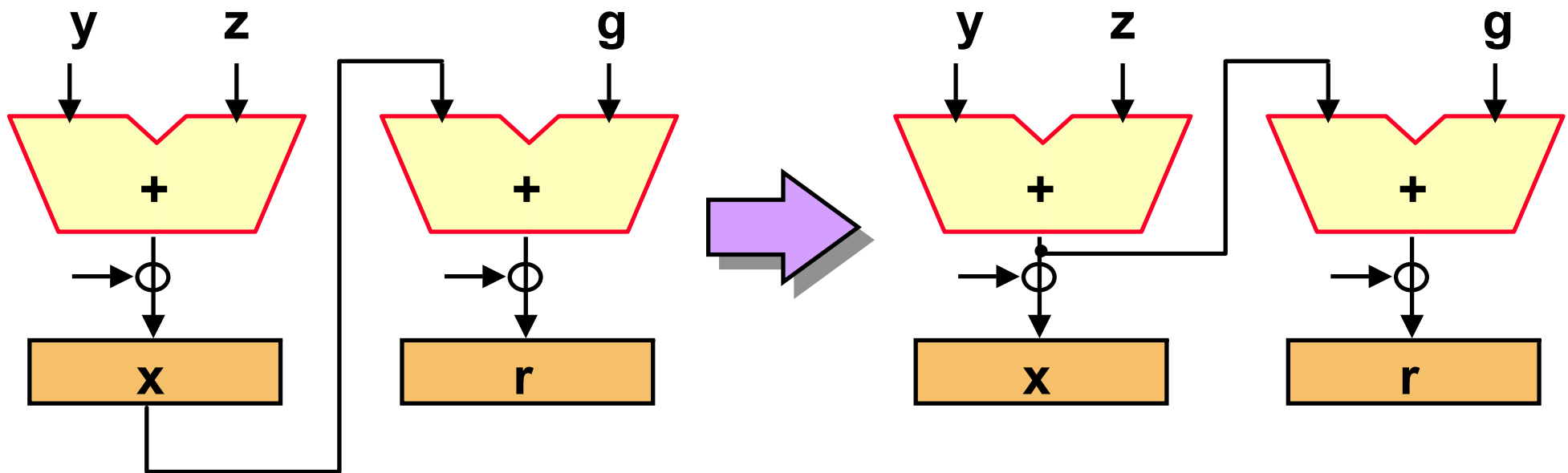
(...);

Forwarding des Resultats x



4.3 Pipelining-Grundlagen

- Von der Hardware-Realisierung aus gesehen, bedeutet Forwarding eine Verzweigung des Resultats für x:

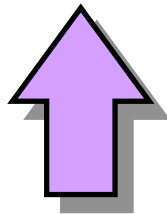


➡ hierdurch kann die notwendige Taktdauer erhöht werden !

4.3 Pipelining-Grundlagen

2. Einführen von Pipelineregistern:

① $(x \leq y+z, \dots);$
 $(\dots, k \leq x+g);$



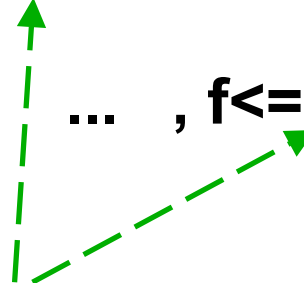
Verschieben verboten durch
Bernstein'sche Regeln !

$(x \leq r+s, \dots, f \leq x+s);$

Resultat: $f' = y+z+s$

② $(x \leq y+z, \dots);$
 $(\dots, xp \leq x, k \leq x+g);$

$(x \leq r+s, \dots, f \leq xp+s);$

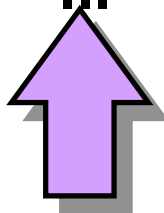


Resultat: ebenfalls
 $f' = y+z+s$

Einführen eines Pipelineregisters xp

4.3 Pipelining-Grundlagen

②. (x<= y+z, ...);
 (... , **xp**<= x, k<= x+g);



Verschieben erlaubt

(x<= r+s, ... , f<= **xp**+s);

Resultat: $f' = y + z + s$

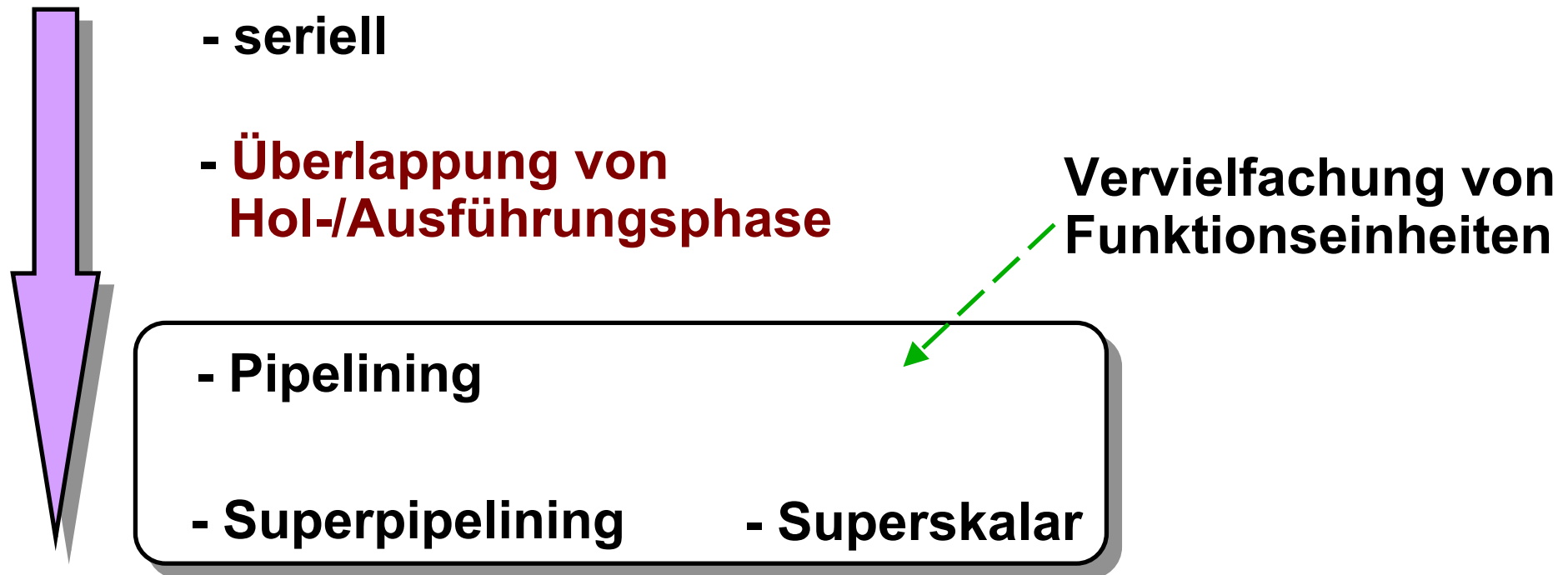
③. (x<= y+z, ...);
 (x<= r+s, **xp**<= x, k<= x+g);

(... , f<= **xp**+s);

Resultat: $f' = y + z + s$

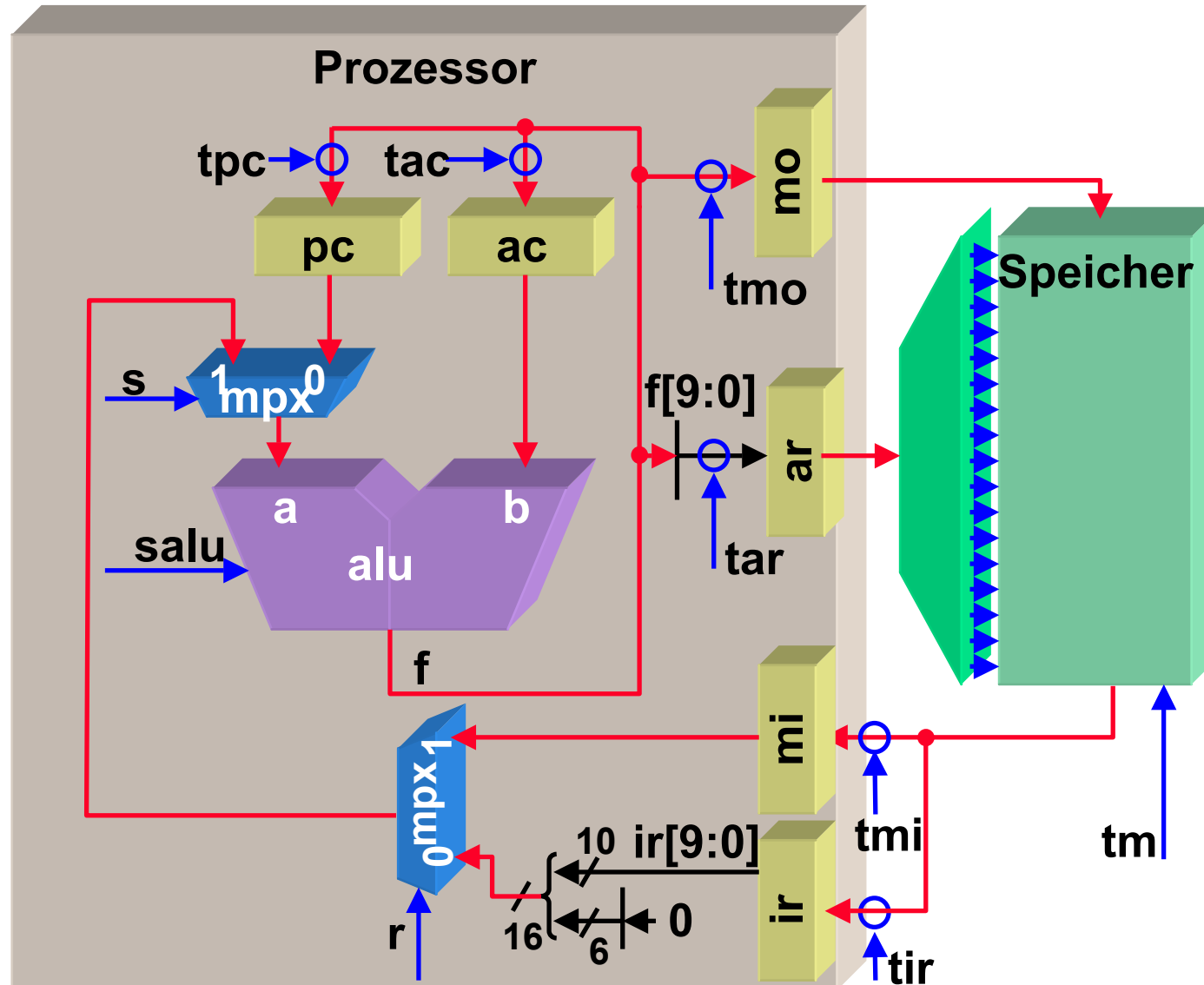
4.4 Überlappung von Befehlshol- und -ausführungsphase

- **Genealogie von Techniken zur Beschleunigung der Instruktionausführung**



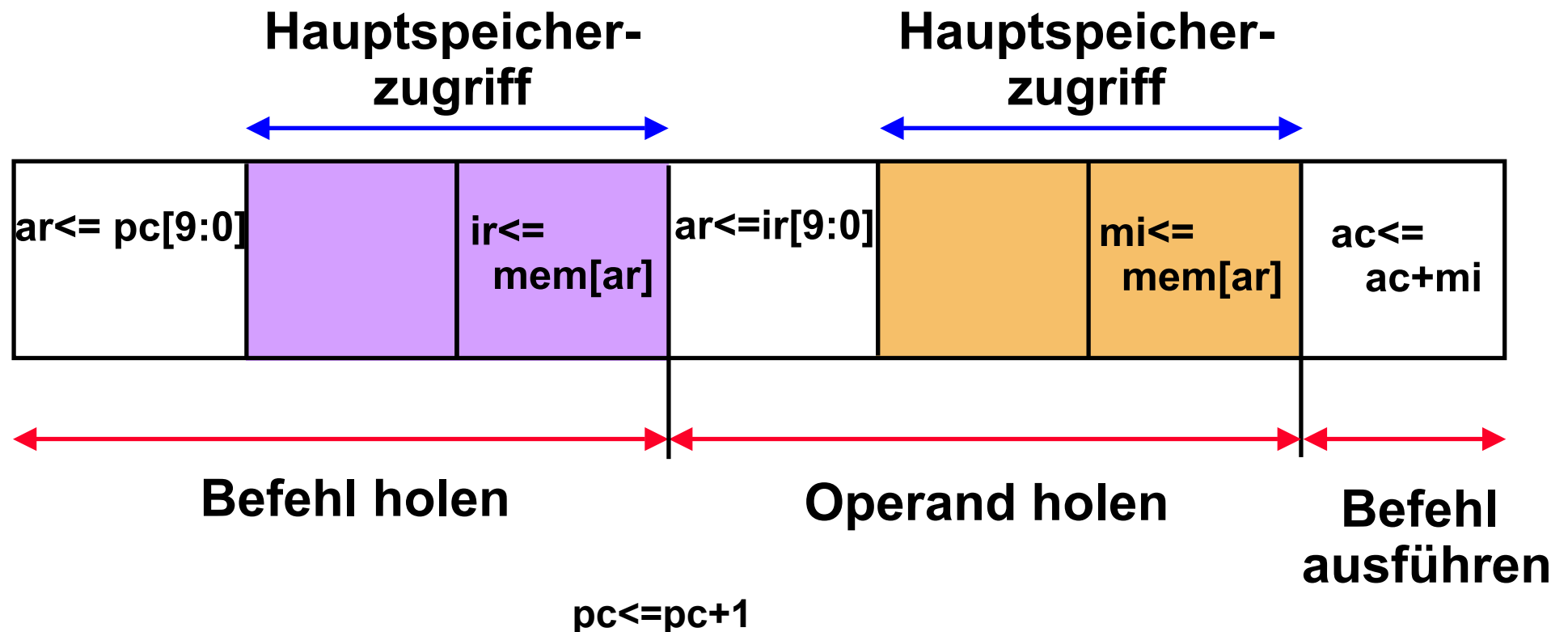
4.4 Überlappung von Befehlshol- und -ausführungsphase

● Beispiel: einfache Akkumulatormaschine



4.4 Überlappung von Befehlshol- und -ausführungsphase

- Annahmen:
 - Hauptspeicherzugriffszeit = 2 * Prozessortaktzeit
 - Befehlsauslegung für größtmögliche Geschwindigkeit
 - Beispiel: Addiere-zu-Akkumulator-Befehl

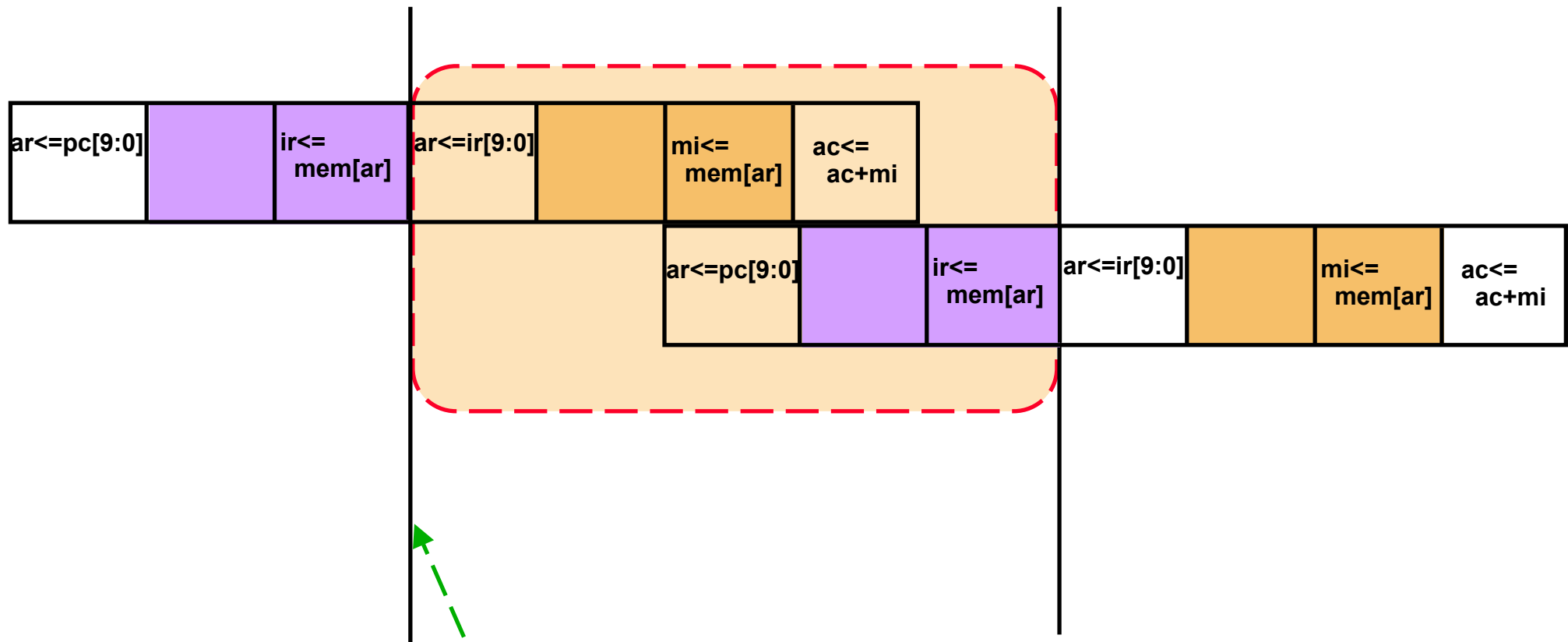


4.4 Überlappung von Befehlshol- und -ausführungsphase

- 7 Takte notwendig
- Auslastung bei serieller Ausführung:
 - Hauptspeicher 57%
 - CPU 57%

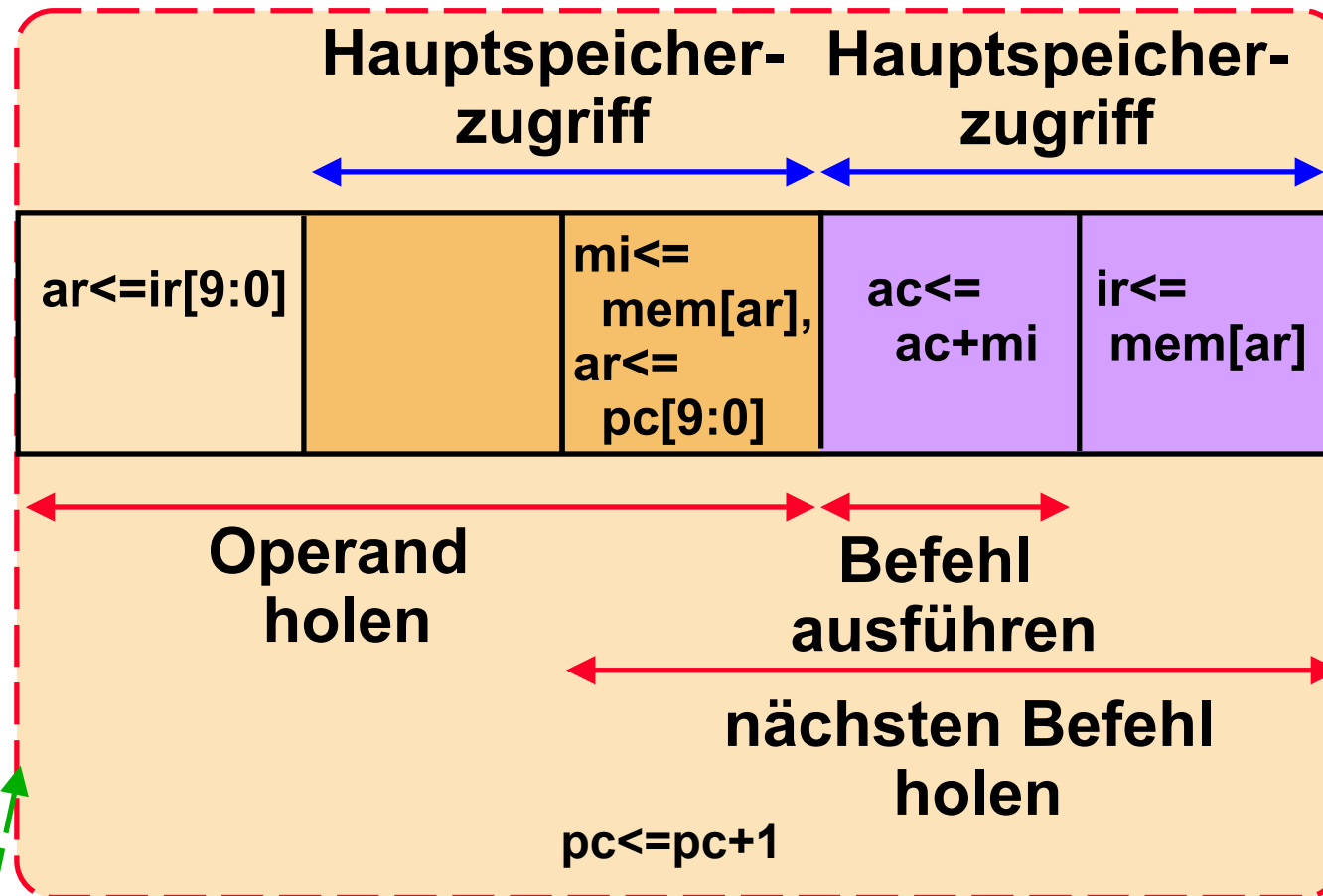
4.4 Überlappung von Befehlshol- und -ausführungsphase

- Überlappung der Ausführungsphase mit dem Holen des nächsten Befehls:



Zeitpunkt, an dem der auszuführende Befehl gerade aus dem Speicher geholt wurde und in ir steht

4.4 Überlappung von Befehlshol- und -ausführungsphase



Zeitpunkt, an dem der auszuführende Befehl gerade aus dem Speicher geholt wurde und in ir steht

4.4 Überlappung von Befehlshol- und -ausführungsphase

- **Wirkungsvolles Verfahren, um die Leistung einfacher Prozessoren zu steigern**
 - hier nur noch 5 Takte notwendig
 - keine zusätzlichen Kosten
 - Auslastung bei überlappter Ausführung:
 - Hauptspeicher 80%
 - CPU 80%
- **Anwendung z.B. in einfachen Mikrocontrollern (z.B. PIC16C5X)**

4.5 Pipelining der DLX

- Im folgenden werden die Schritte zum Entwurf eines DLX-Prozessors mit fünfstufiger Pipeline gezeigt
- Ausgangspunkt ist die in Abschnitt 4.1 entworfene Tabelle für den seriellen Ablauf der Befehlsverarbeitung

4.5 Pipelining der DLX

- Ablauftabelle mit belegten Ressourcen:

LW	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: ar<=a+ir[16:31]	MEM: di<= d-mem[ar]	WB: rf[ir[11:15]]<= di
SW	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: ar<=a+ir[16:31], do<=b	MEM: d-mem[ar]<= do	
ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b	WB: rf[ir[16:20]]<= temp	
J	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: pc<= pc+ir[6:31]		
BEQZ	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: if a=0 then pc<=pc+ ir[16:31]		

Befehlsspeicher, Registersatz
ALU

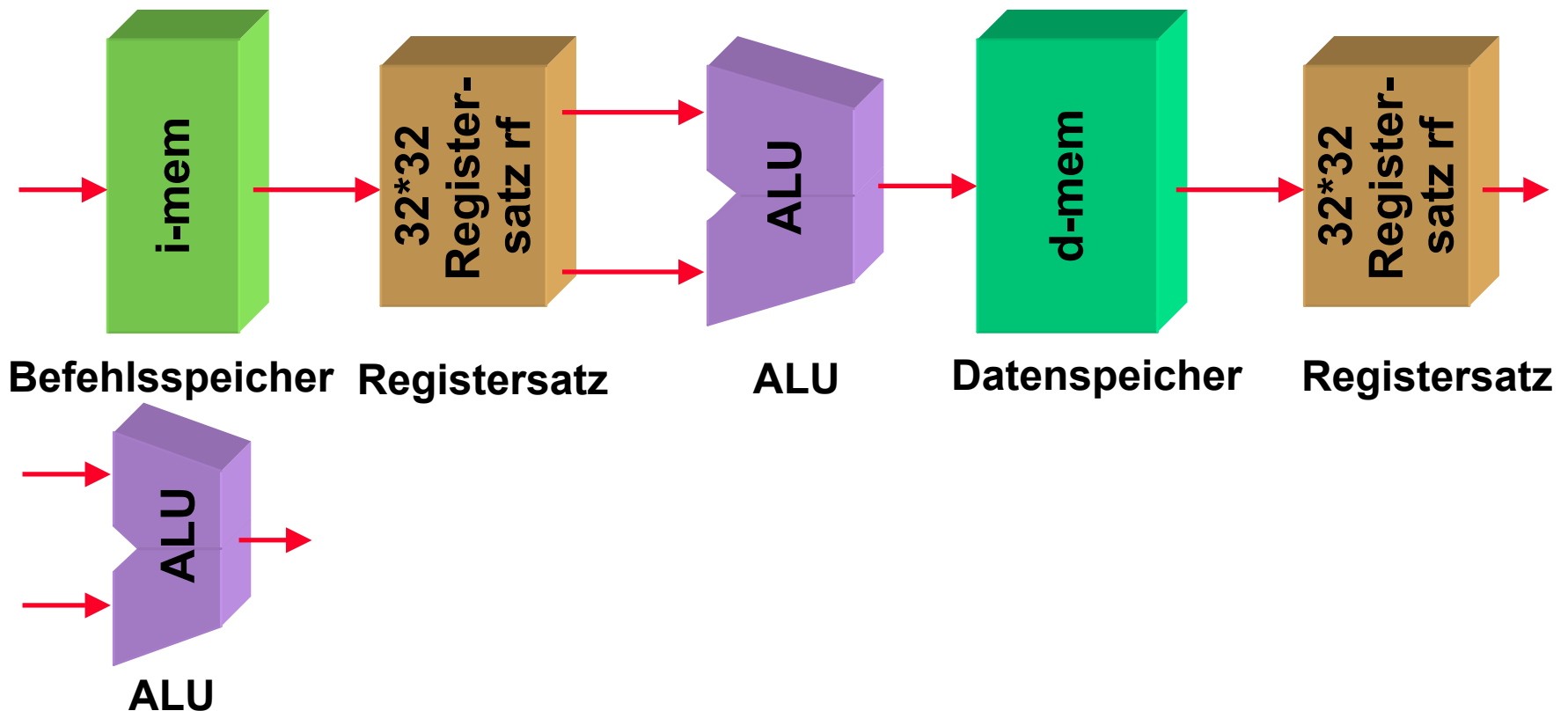
ALU

Datenspeicher, Registersatz
Registersatz

4.5 Pipelining der DLX

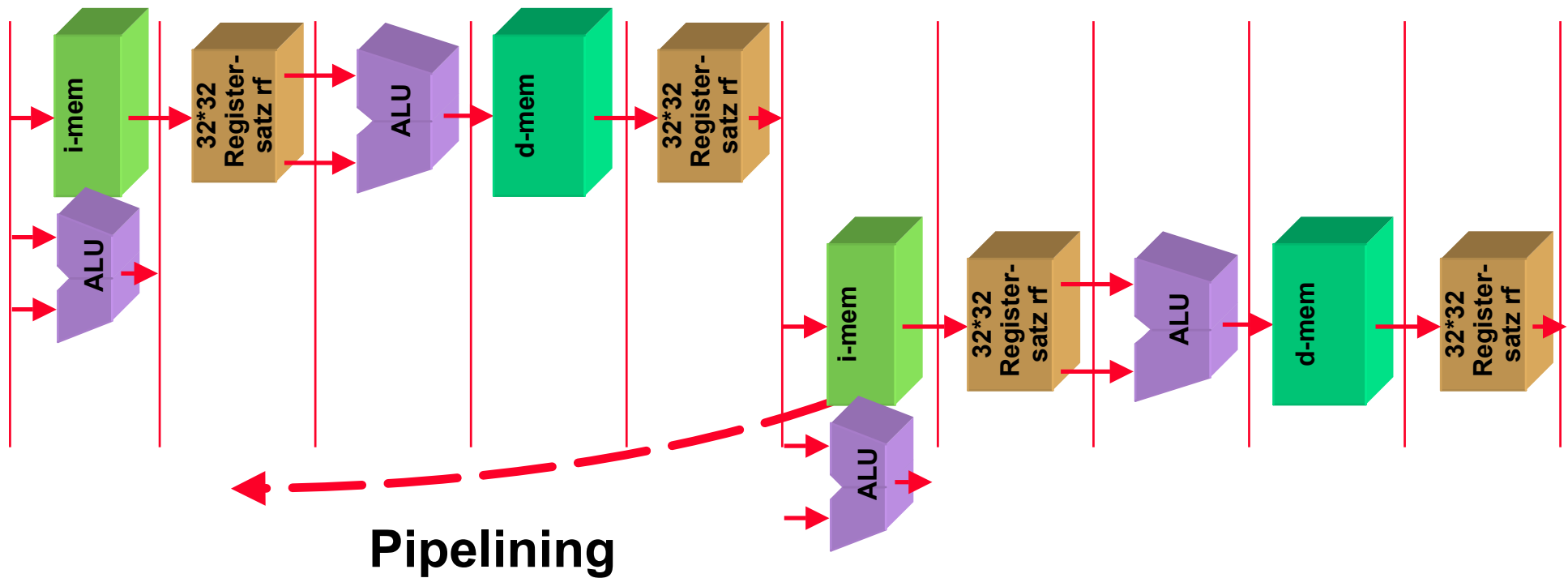
- Belegte Ressourcen für LW-Befehl:

LW	IF: $ir \leq i\text{-mem}[pc]$, $pc \leq pc + 4$	ID: $a \leq rf[ir[6:10]]$, $b \leq rf[ir[11:15]]$	EX: $ar \leq a + ir[16:31]$	MEM: $di \leq$ $d\text{-mem}[ar]$	WB: $rf[ir[11:15]] \leq$ di



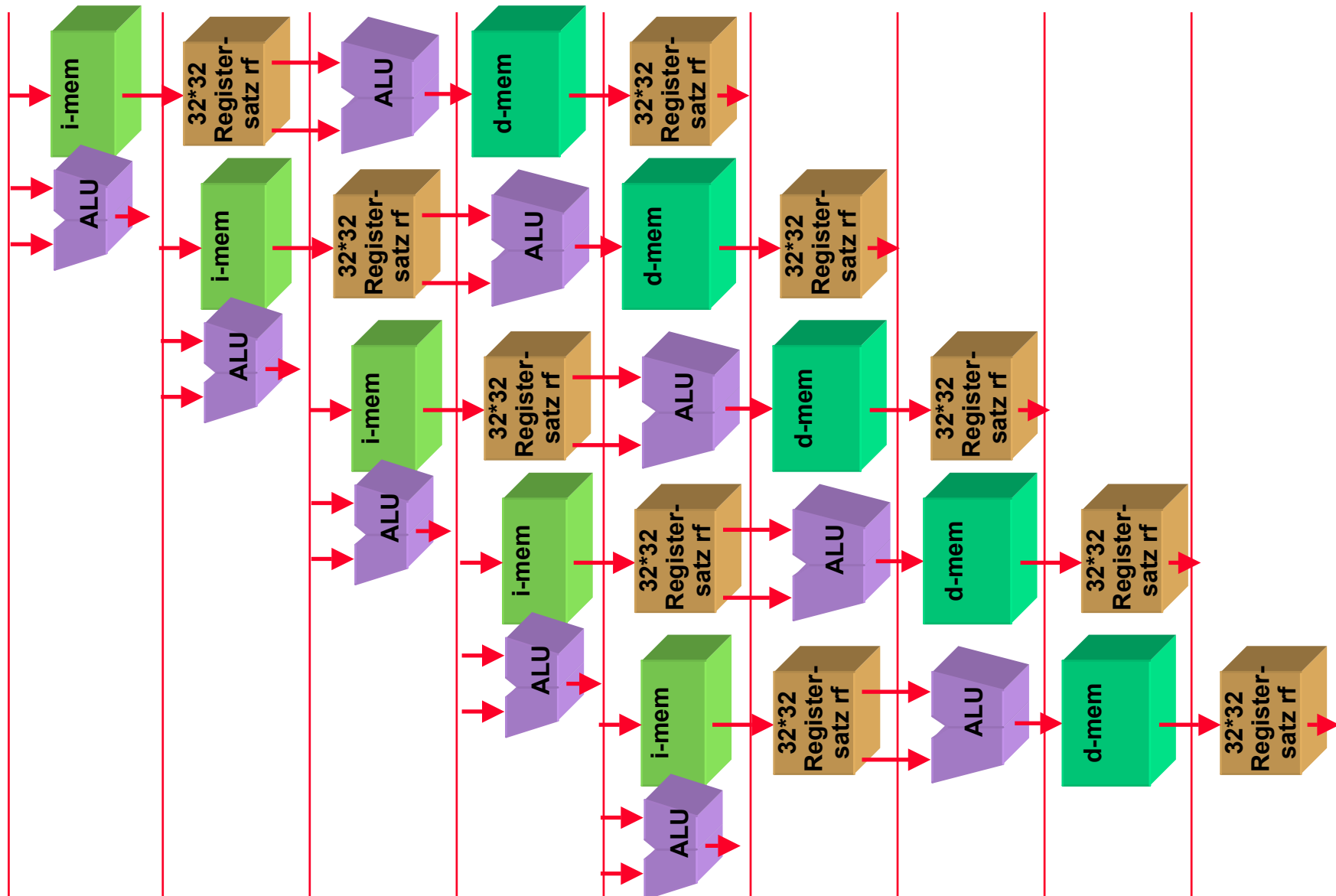
4.5 Pipelining der DLX

- **Serielle Ausführung:**



4.5 Pipelining der DLX

● Fünfstufiges Pipelining, nur LW-Befehle:



4.5 Pipelining der DLX

- Beim Pipelining können drei Konfliktarten auftreten:

- **Ressourcenkonflikte**

- ein Hilfsmittel wie z.B. ein Funktionsblock oder Speicher soll für unterschiedliche Operationen benutzt werden

- **Datenkonflikte**

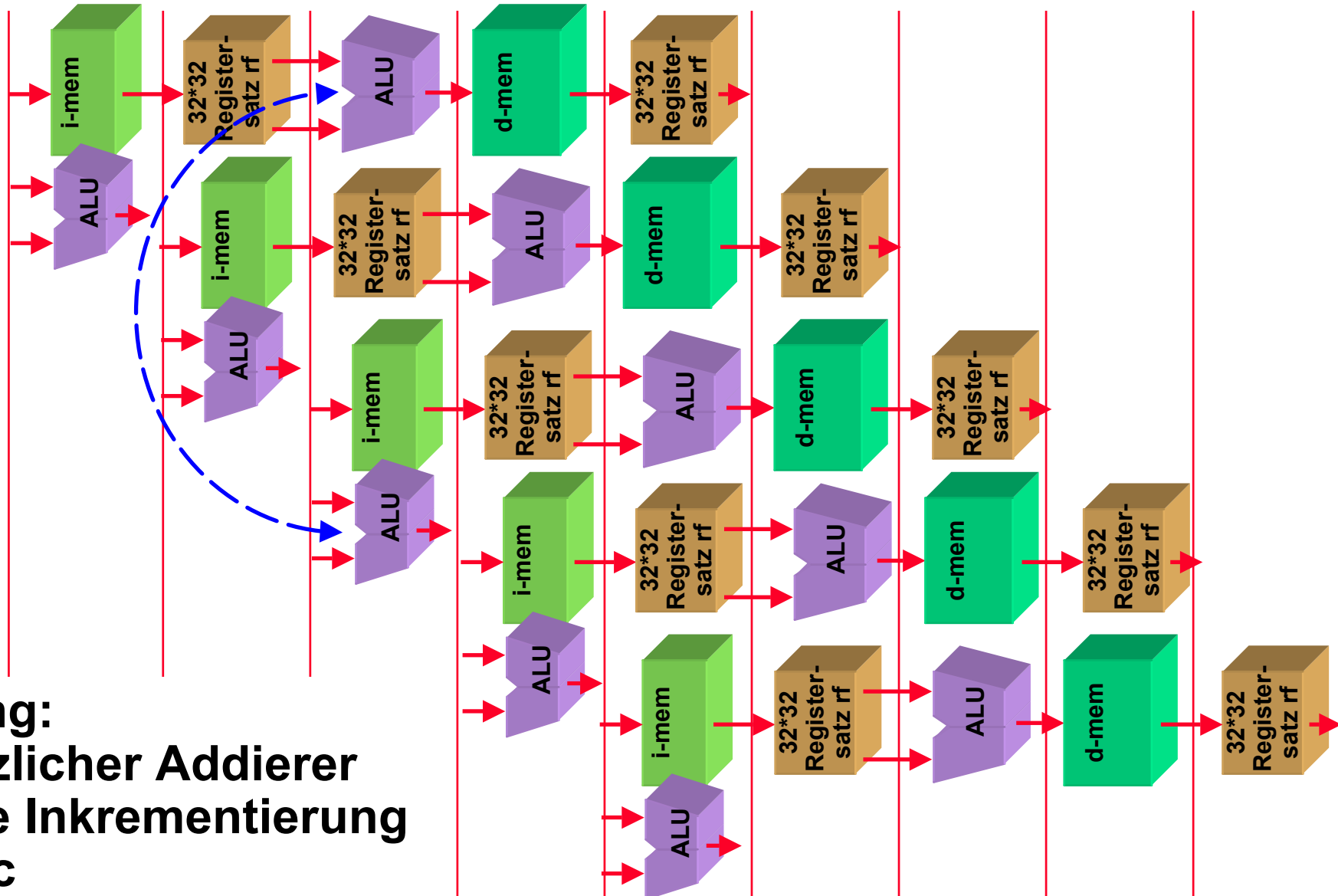
- auf Befehlsebene: Daten, die in einem Befehl benutzt werden sollen, stehen nicht zur Verfügung
- auf Transferebene: Registerinhalte, die in einem Schritt benutzt werden, stehen nicht zur Verfügung

- **Steuerkonflikte**

- die Pipeline muß wegen Verzweigungen geleert und neu gefüllt werden

4.5 Pipelining der DLX

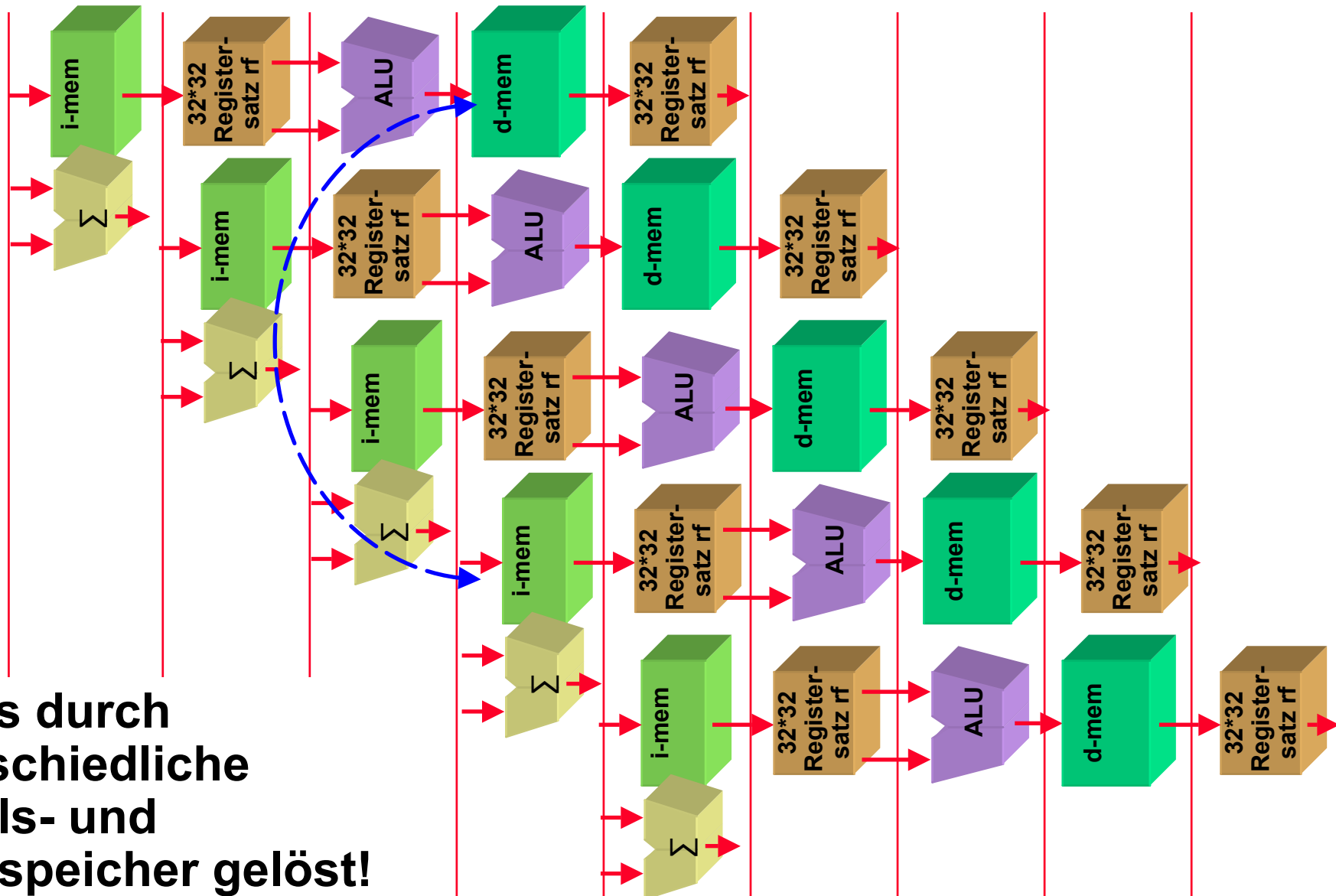
- 1. Ressourcen**konflikt**, nur LW-Befehle: ALU



Lösung:
zusätzlicher Addierer
für die Inkrementierung
des pc

4.5 Pipelining der DLX

● 2. Ressourcenkonflikt, nur LW-Befehle: Speicher

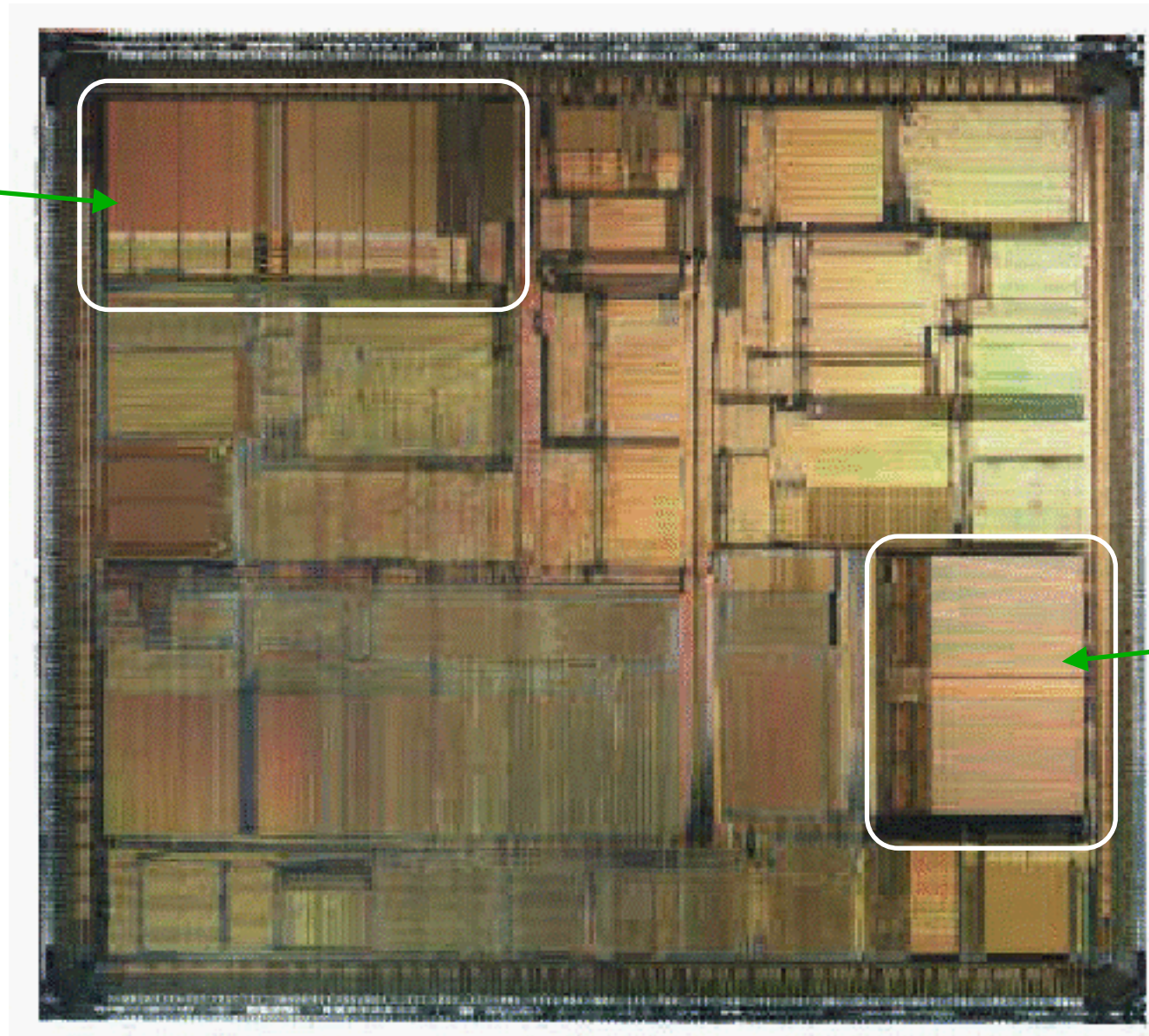


bereits durch
unterschiedliche
Befehls- und
Datenspeicher gelöst!

4.5 Pipelining der DLX

Ultra SPARC-II

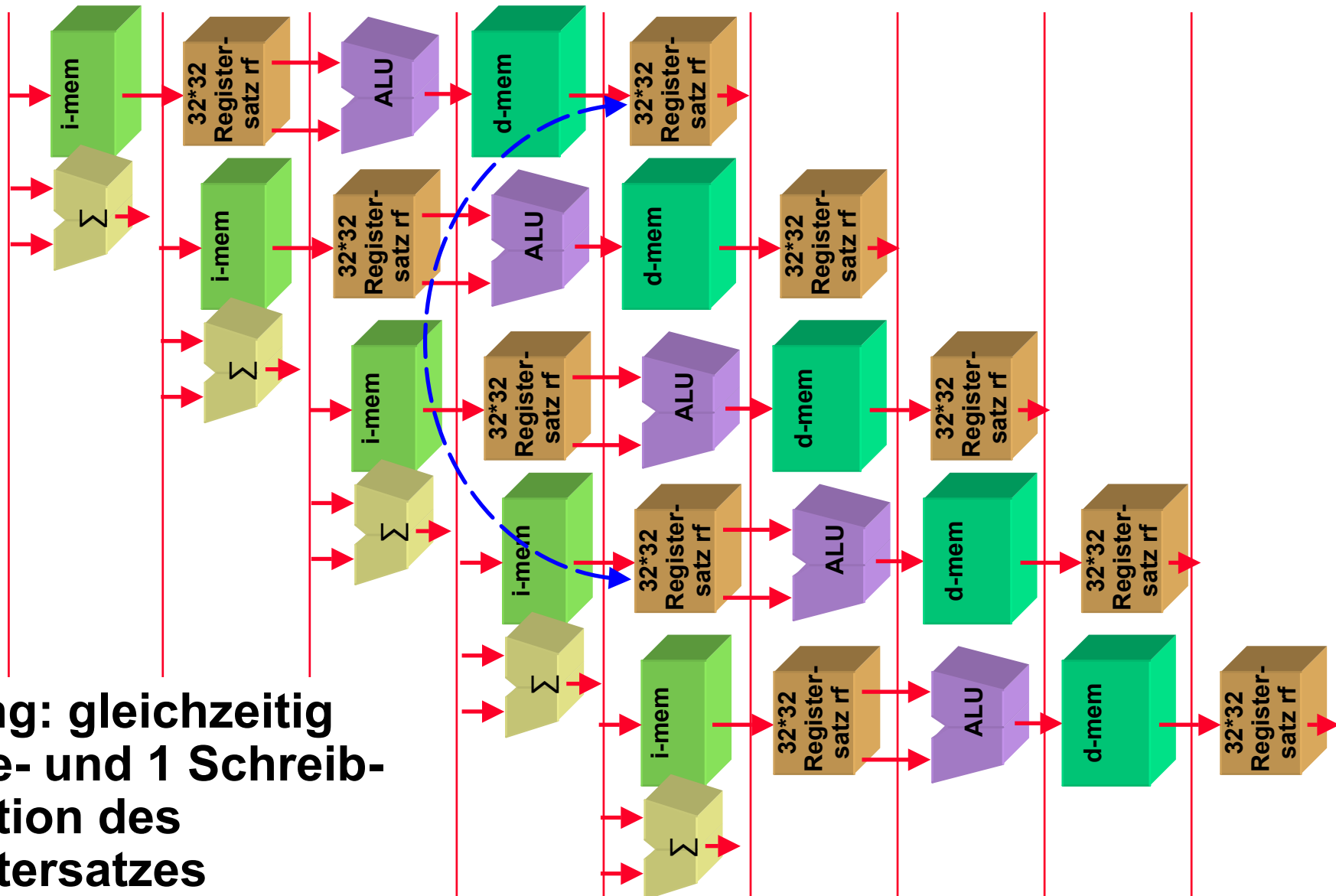
Befehls-
Cache
i-mem



Daten-
Cache
d-mem

4.5 Pipelining der DLX

- 3. Ressourcen**konflikt**, nur LW-Befehle: Registersatz



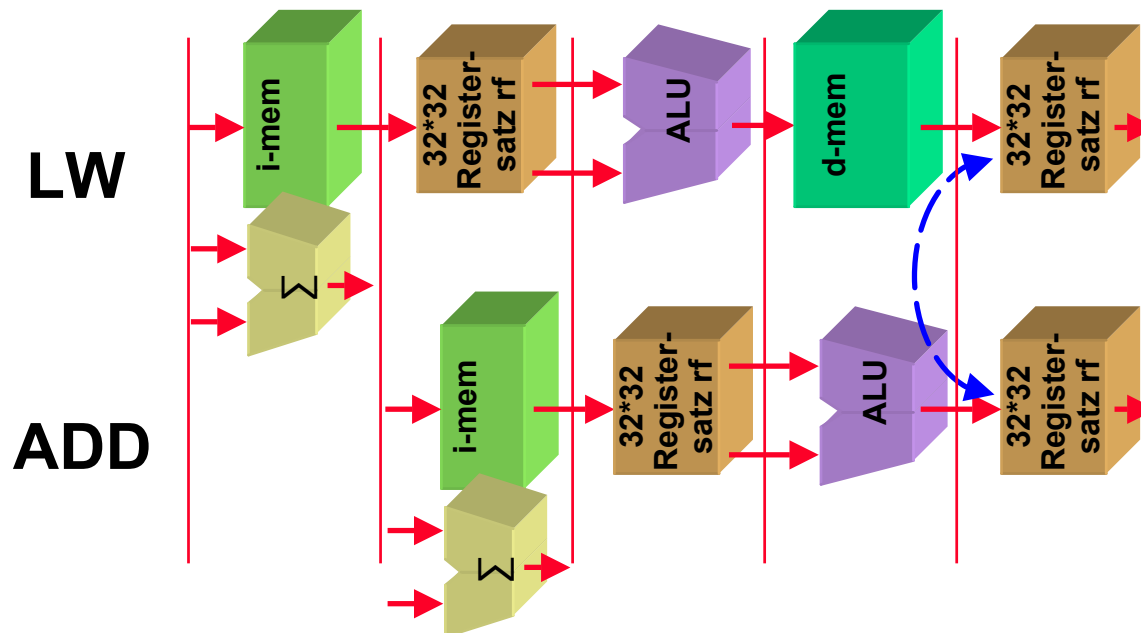
**Lösung: gleichzeitig
2 Lese- und 1 Schreib-
operation des
Registersatzes**

4.5 Pipelining der DLX

- 4. Ressourcenkonflikt LW/ALU-Befehle: Schreiben

Registersatz

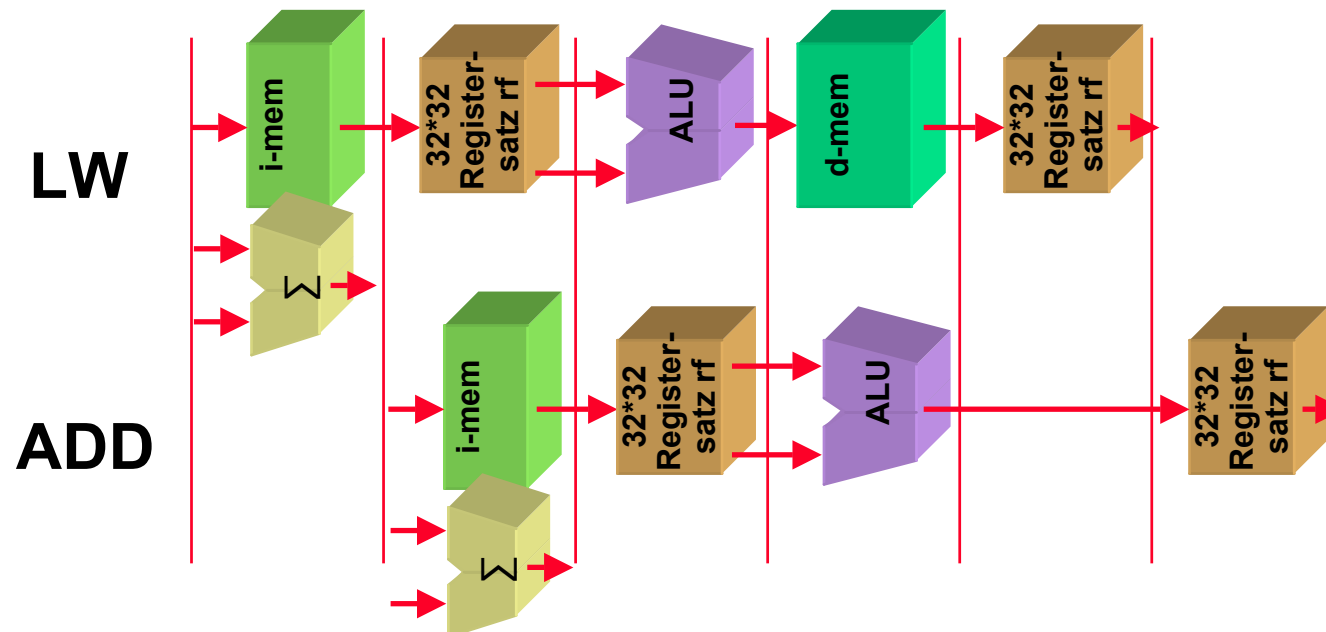
LW	IF: $ir \leq i\text{-mem}[pc]$, $pc \leq pc+4$	ID: $a \leq rf[ir[6:10]]$, $b \leq rf[ir[11:15]]$	EX: $ar \leq a+ir[16:31]$	MEM: $di \leq d\text{-mem}[ar]$	WB: $rf[ir[11:15]] \leq di$
	ADD	IF: $ir \leq i\text{-mem}[pc]$, $pc \leq pc+4$	ID: $a \leq rf[ir[6:10]]$, $b \leq rf[ir[11:15]]$	EX: $temp \leq a+b$	WB: $rf[ir[16:20]] \leq temp$



4.5 Pipelining der DLX

- Lösung: Leerschritt bei ALU-Befehlen

LW	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: ar<=a+ir[16:31]	MEM: di<=d-mem[ar]	WB: rf[ir[11:15]]<= di
	ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b	WB: rf[ir[16:20]]<= temp



4.5 Pipelining der DLX

- **Zusammenfassung der auftretenden Ressourcenkonflikte:**
 - **ALU-Operation und PC-Inkrementierung gleichzeitig**
 - **paralleler Speicherzugriff zu Daten und Instruktionen**
 - **Lesen/Schreiben Registersatz**
 - **Schreiben/Schreiben Registersatz**
- **Lösung durch Vervielfachung der Hilfsmittel und Einfügen von Leerschritten**

4.5 Pipelining der DLX

- **Konfliktarten:**

- **Ressourcenkonflikte**

- ein Hilfsmittel wie z.B. Funktionsblock oder Speicher soll für unterschiedliche Operationen benutzt werden

- **Datenkonflikte**

- auf Transferebene: Registerinhalte, die in einem Schritt benutzt werden, stehen nicht zur Verfügung
- auf Befehlsebene: Daten, die in einem Befehl benutzt werden sollen, stehen nicht zur Verfügung

- **Steuerkonflikte**

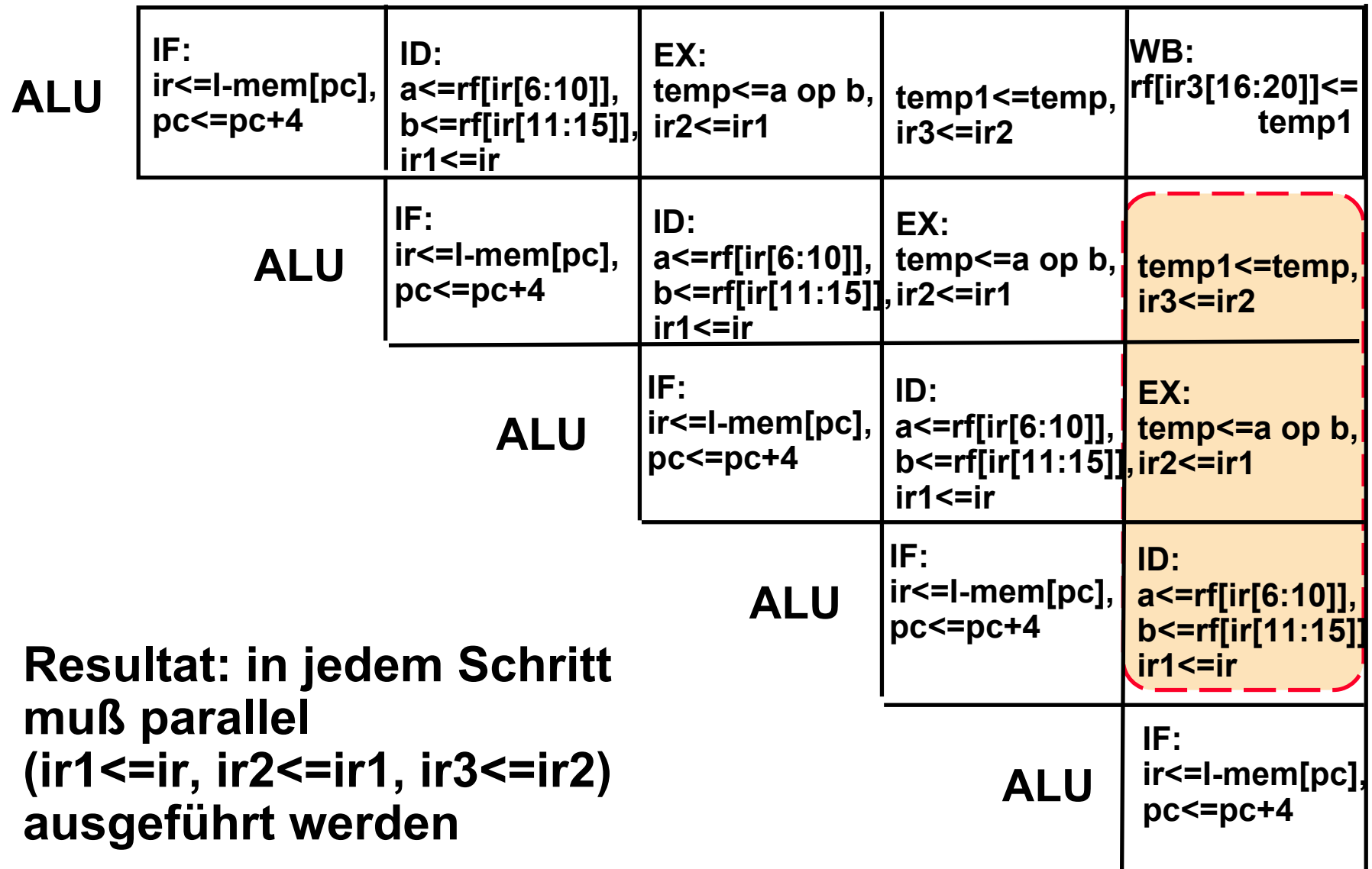
- die Pipeline muß wegen Verzweigungen geleert und neu gefüllt werden

4.5 Pipelining der DLX

- Da das Instruktionsregister in jedem Befehl neu geladen wird, der Inhalt des Instruktionsregisters (oder zumindest Teile davon) aber für die Steuerung jedes Befehls bis zu vier Takte erhalten bleiben muß, sind außer ir noch **drei Pipelineregister** ir1, ir2, ir3 aufzubauen:

ALU	IF: ir <=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a op b	temp1<=temp	WB: rf[ir[16:20]]<= temp1
	ALU	IF: ir <=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a op b	temp1<=temp
		ALU	IF: ir <=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a op b
			ALU	IF: ir <=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]
				ALU	IF: ir <=i-mem[pc], pc<=pc+4

4.5 Pipelining der DLX



4.5 Pipelining der DLX

- Entsprechend müssen alle anderen Befehle wie z.B. der LW-Befehl modifiziert werden:

LW	IF: ir<=I-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]], ir1<=ir	EX: ar<=a+ir1[16:31], ir2<=ir1	MEM: di<=D-mem[ar], ir3<=ir2	WB: rf[ir3[11:15]]<= di

 im folgenden wird das Laden der ir-Queue nicht mehr explizit in den Befehlen beschrieben

- **am Dienstag ...**
 - **Überlappung von Befehlshol- und Ausführungsphase**
 - **Pipeline-Komplikationen**
 - **Pipelining = Parallelisierung, daher:**
 - **Ressourcen-Konflikte: 1 Ressource/mehrere Nutzer**
 - **Datenkonflikte: Daten in Registern sind nicht zum richtigen Zeitpunkt vorhanden**
 - **Einführung von Pipelineregistern**

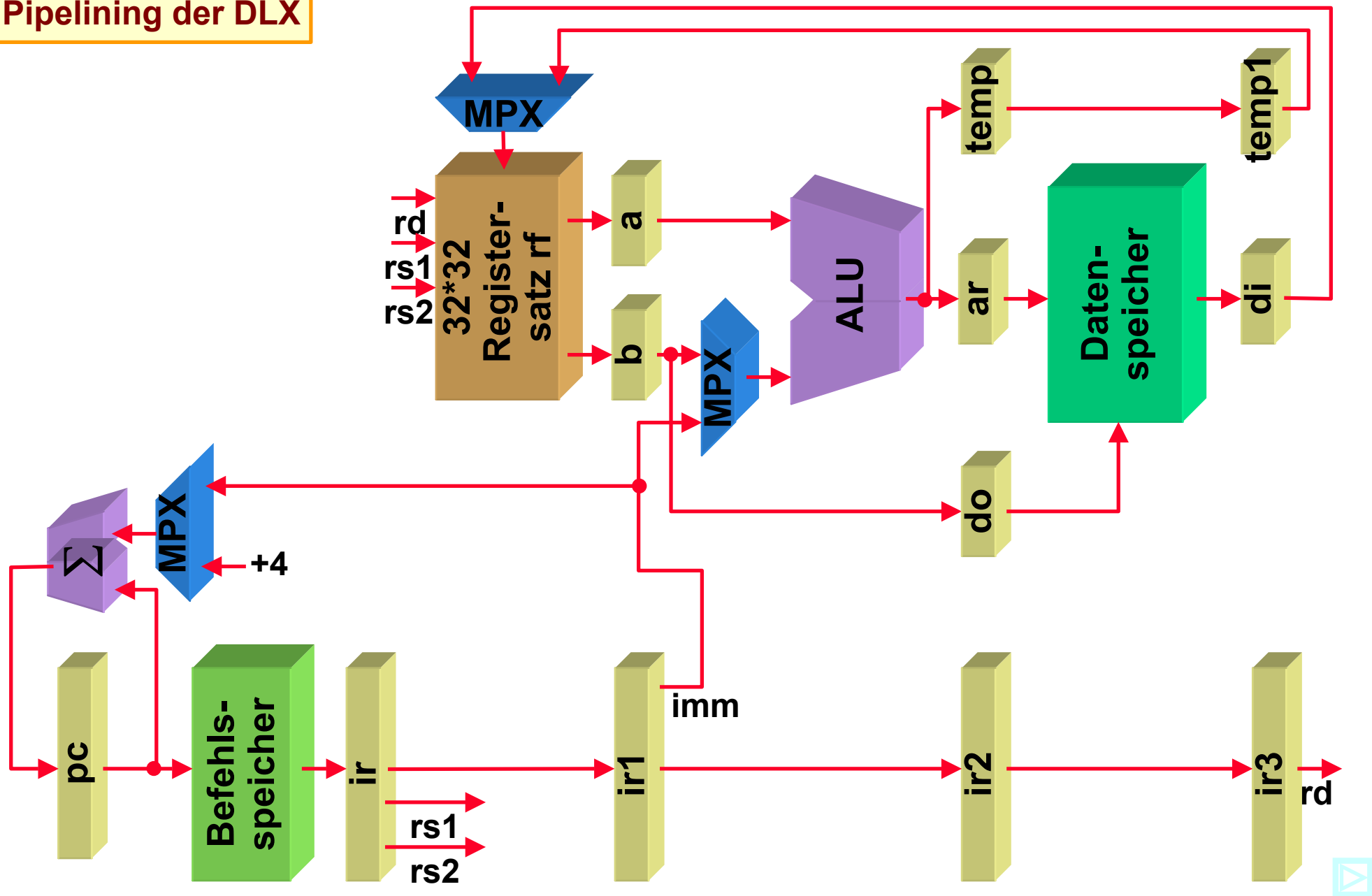
4.5 Pipelining der DLX

- Pipelineregister für **ALU-Resultate**

ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b		WB: rf[ir[16:20]]<= temp
	ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b	

ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b	temp1<=temp	WB: rf[ir[16:20]]<= temp1
	ADD	IF: ir<=i-mem[pc], pc<=pc+4	ID: a<=rf[ir[6:10]], b<=rf[ir[11:15]]	EX: temp<=a+b	temp1<=temp

4.5 Pipelining der DLX



4.5 Pipelining der DLX

- **Konfliktarten:**
 - **Ressourcenkonflikte**
 - ein Hilfsmittel wie z.B. Funktionsblock oder Speicher soll für unterschiedliche Operationen benutzt werden
 - **Datenkonflikte**
 - auf Transferebene: Registerinhalte, die in einem Schritt benutzt werden, stehen nicht zur Verfügung
 - auf Befehlsebene: Daten, die in einem Befehl benutzt werden sollen, stehen nicht zur Verfügung
 - **Steuerkonflikte**
 - die Pipeline muß wegen Verzweigungen geleert und neu gefüllt werden

4.5 Pipelining der DLX

- Prinzipiell mögliche Datenkonflikte auf Befehlsebene bei zwei aufeinanderfolgende Befehle A und B:
 - **RAW-(read-after-write-)Hazard**: B liest Wert aus Register, bevor ihn A geschrieben hat (d.h. B liest fälschlich noch den alten Wert)
 - **WAW-(write-after-write-)Hazard**: B schreibt Wert in Register, bevor ihn A geschrieben hat (d.h. es bleibt der Wert, den A geschrieben hat, fälschlich als letzter erhalten)
 - ☞ kann vermieden werden, indem nur in einer Pipelinestufe geschrieben werden darf
 - **WAR-(write-after-read-)Hazard**: B schreibt in Register, bevor es A gelesen hat (d.h. A liest fälschlich bereits den neuen Wert)

4.5 Pipelining der DLX

- Quiz: welche Arten von Hazards treten – bei einer Vertauschung der Reihenfolge – auf?

$x \leftarrow y+z;$

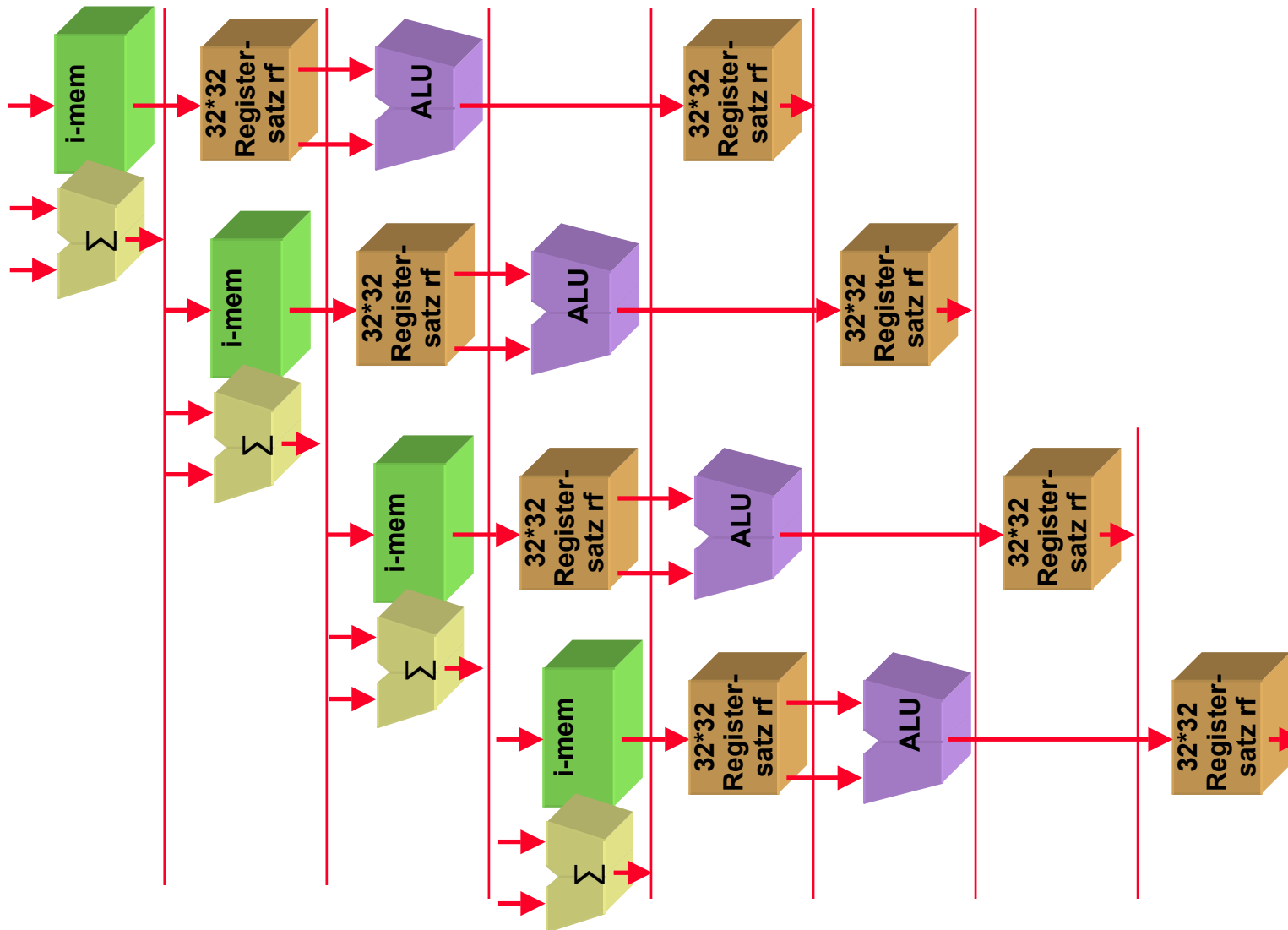
$y \leftarrow x+z;$

$y \leftarrow x+z;$

$x \leftarrow y+z;$



4.5 Pipelining der DLX



ADD **R1**,R2,R3

SUB R5,**R1**,R4

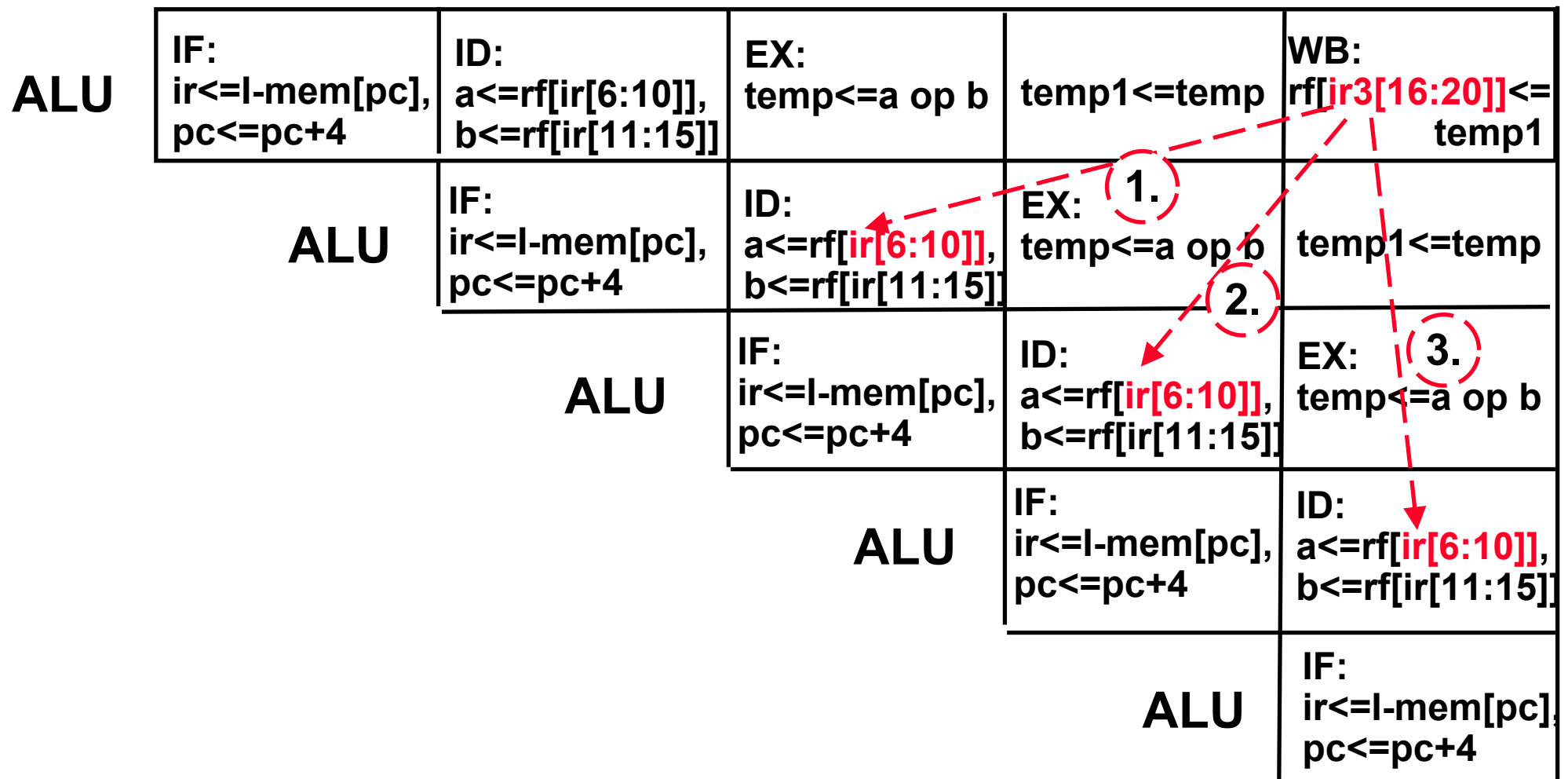
ADD R7,**R1**,R2

XOR R3,**R1**,R5

4.5 Pipelining der DLX

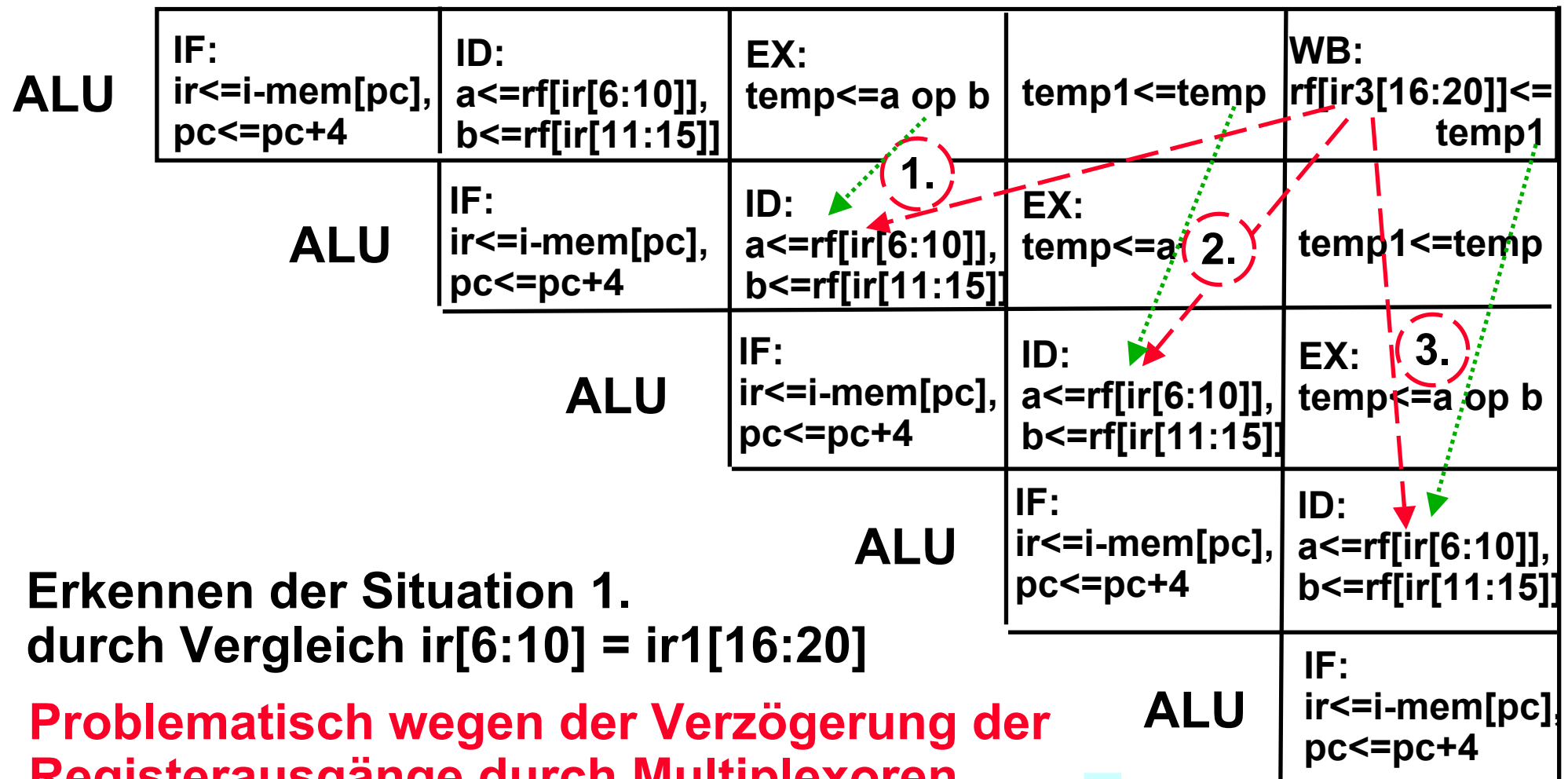
- Annahme: $ir3[16:20] = ir[6:10]$, aufeinanderfolgende **ALU-Befehle**

➤ was für ein Typ von Konflikt ?



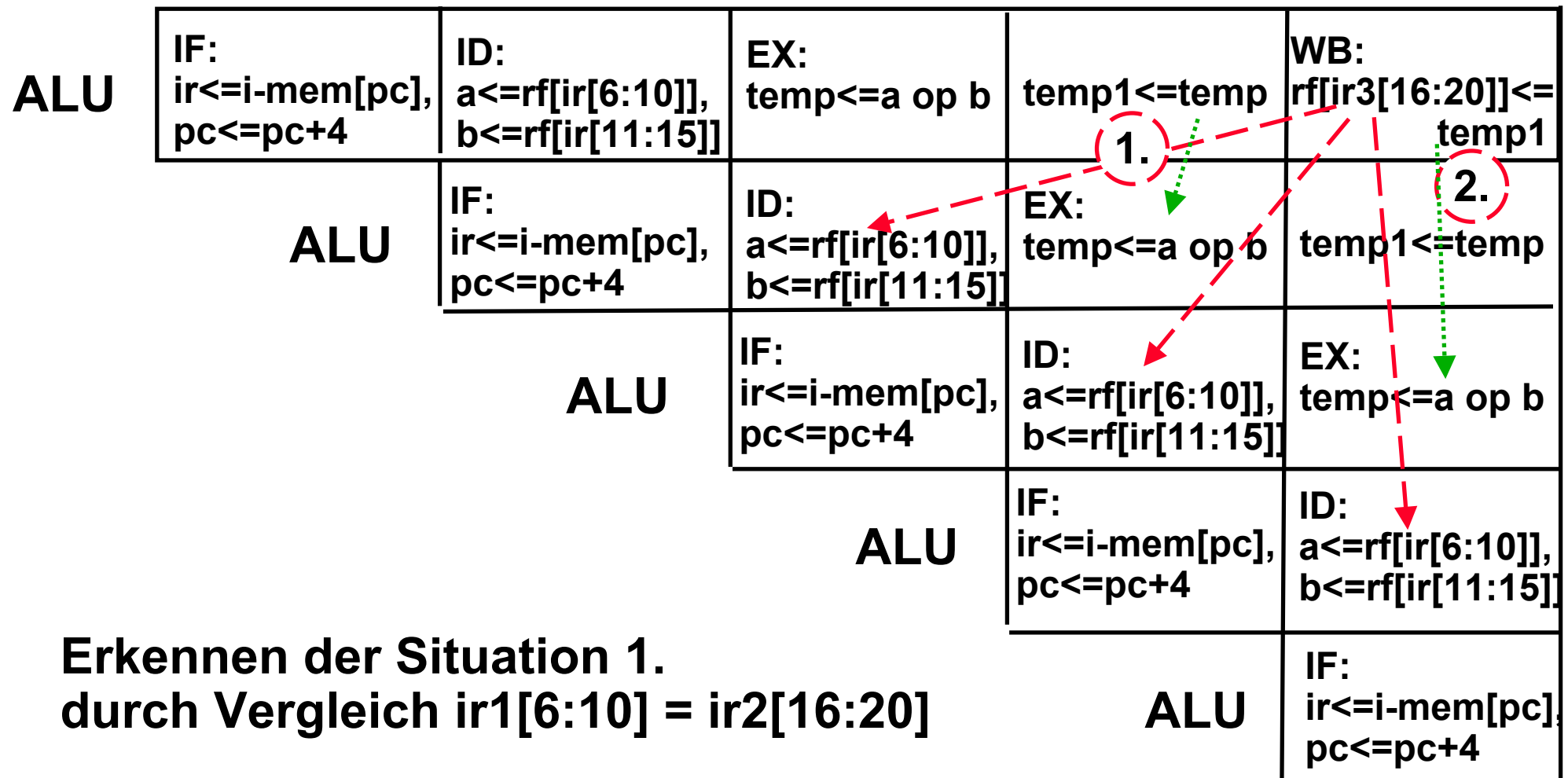
4.5 Pipelining der DLX

- 1. Lösung: **Forwarding** von a op b, temp und temp1 **vor** den a- und b-Registern:



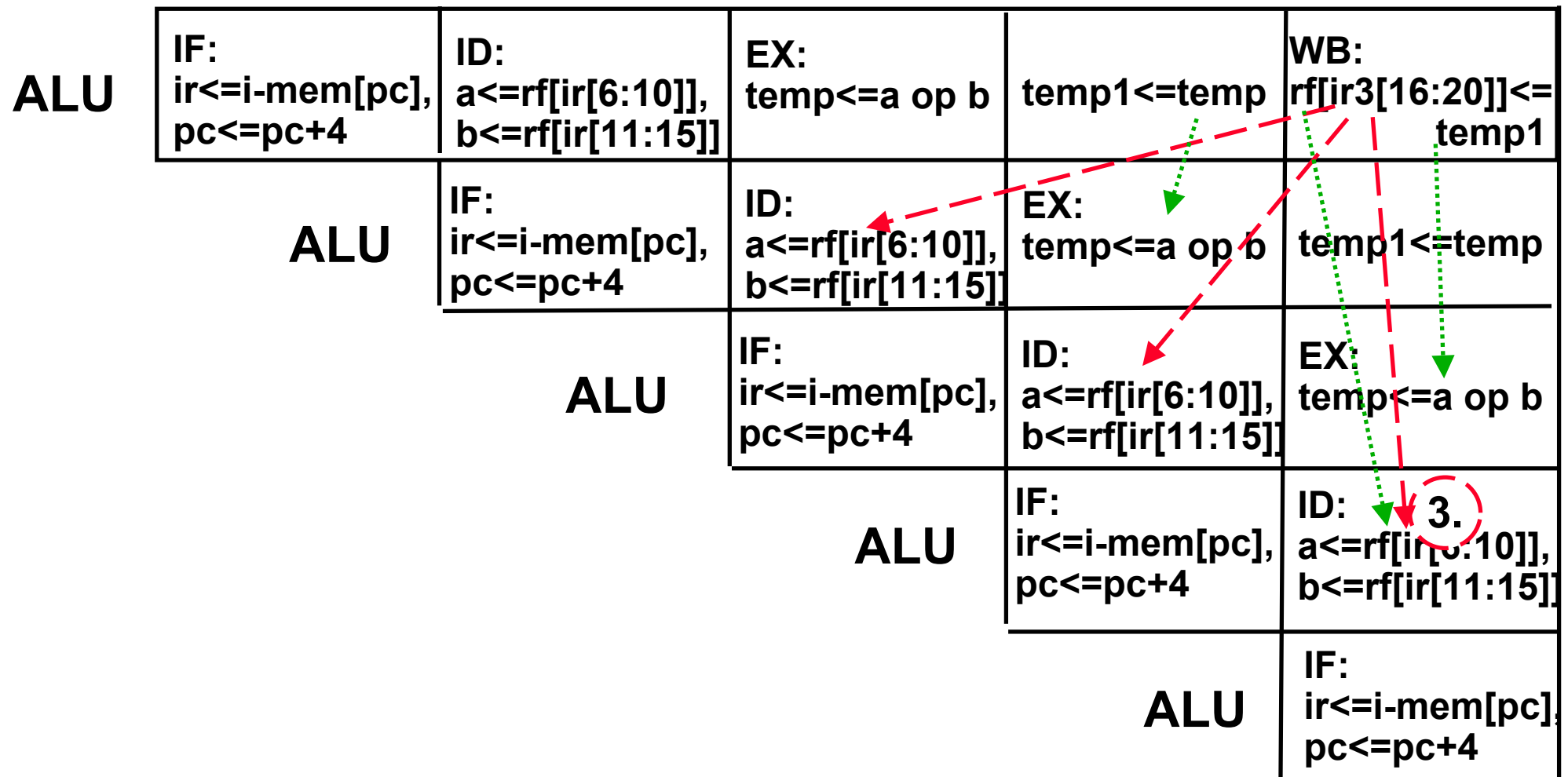
4.5 Pipelining der DLX

- 2. Lösung: **Forwarding** der temp- und temp1-Register als Operanden der ALU, "ALU-Bypass":

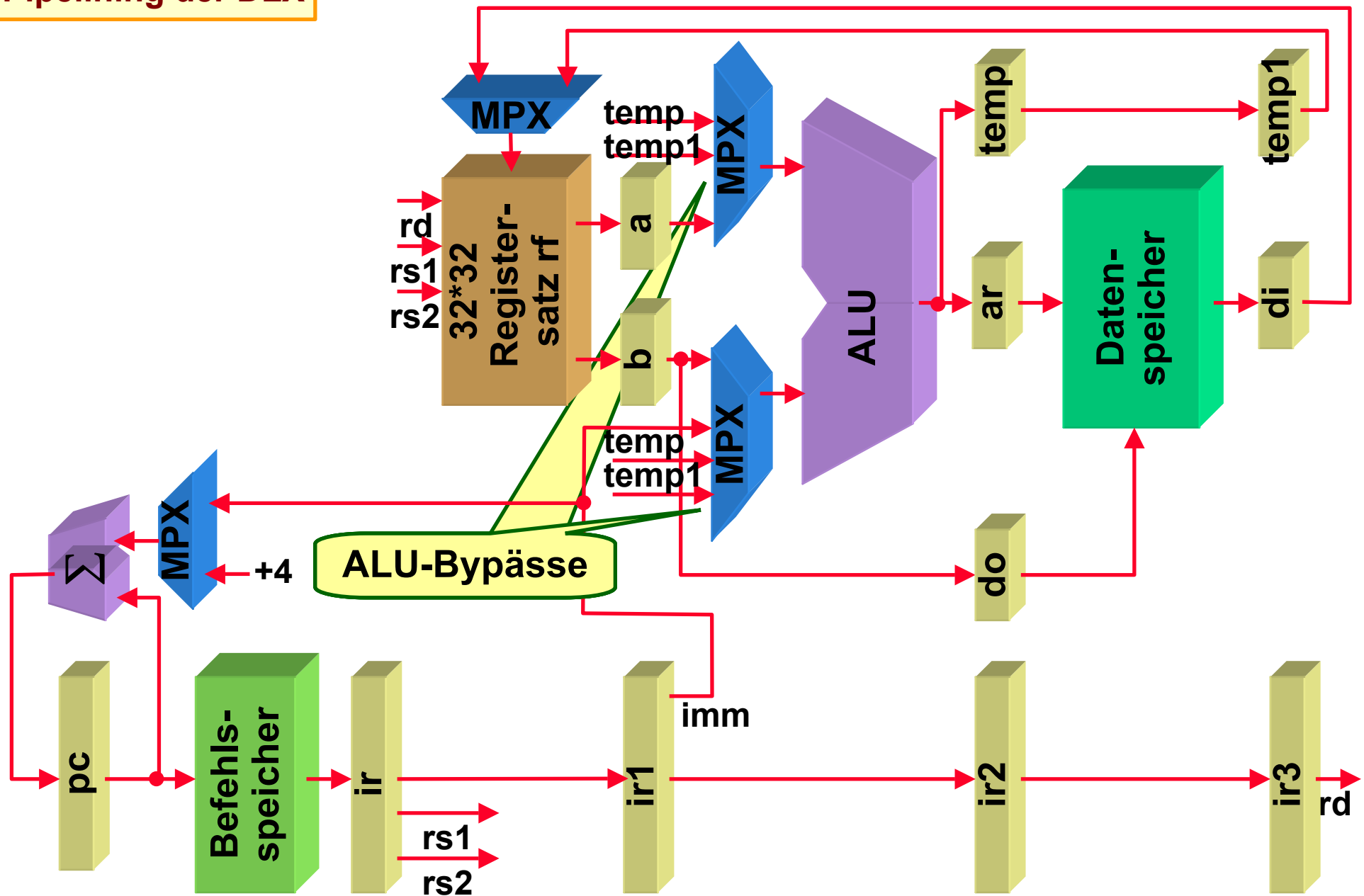


4.5 Pipelining der DLX

- Ferner gibt der Registersatz die eingelesenen Daten sofort wieder am Ausgang im selben Schritt ab:

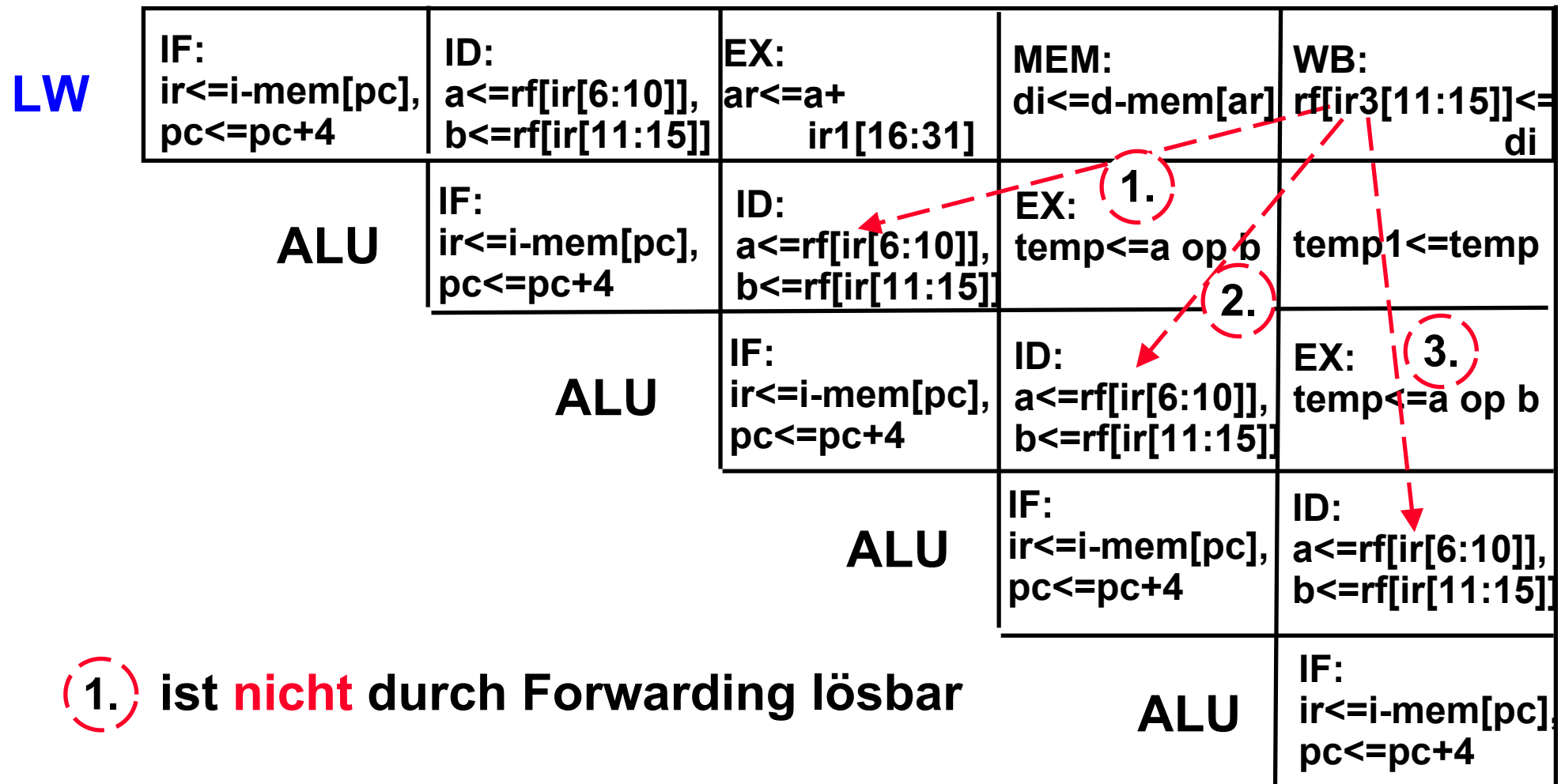


4.5 Pipelining der DLX

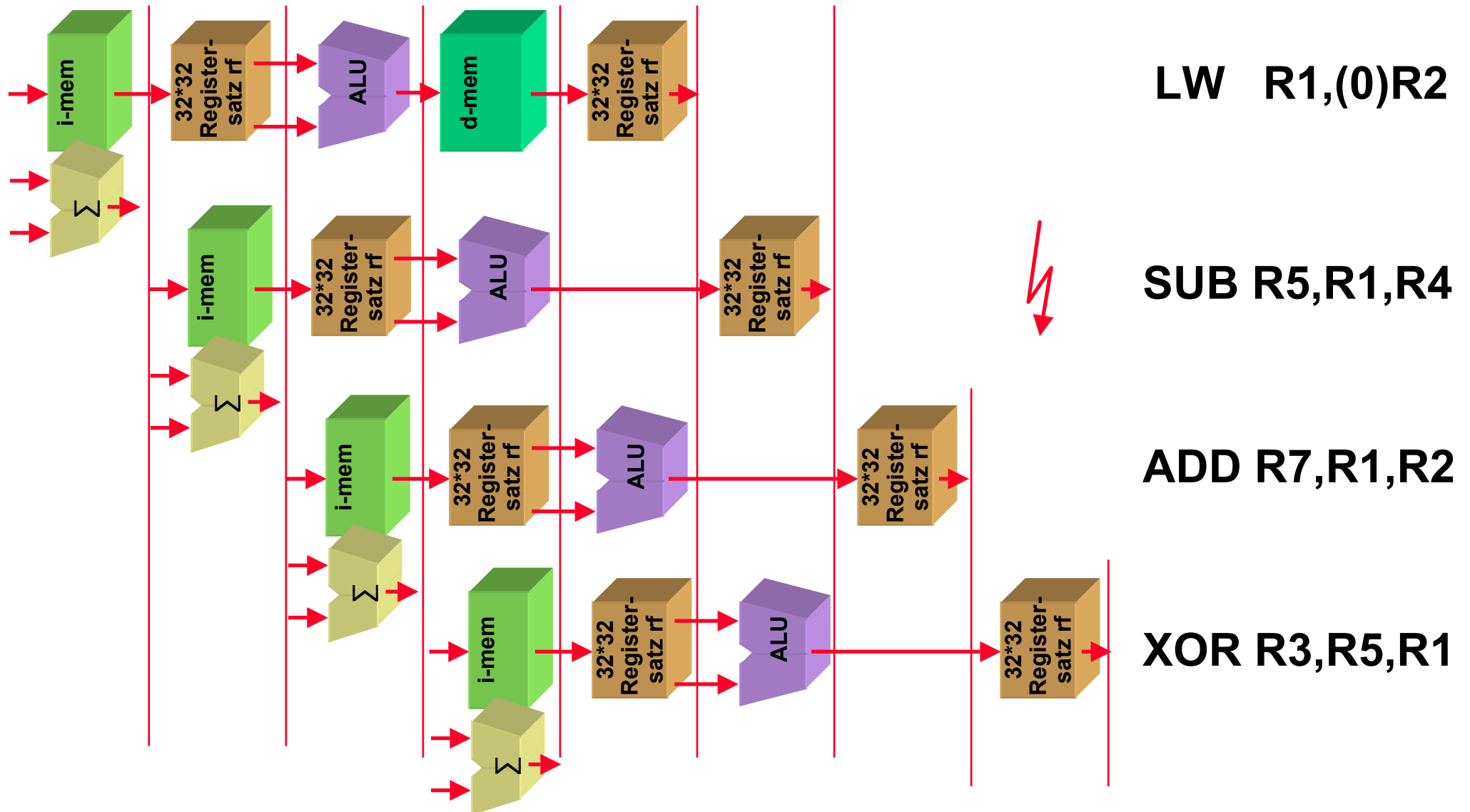


4.5 Pipelining der DLX

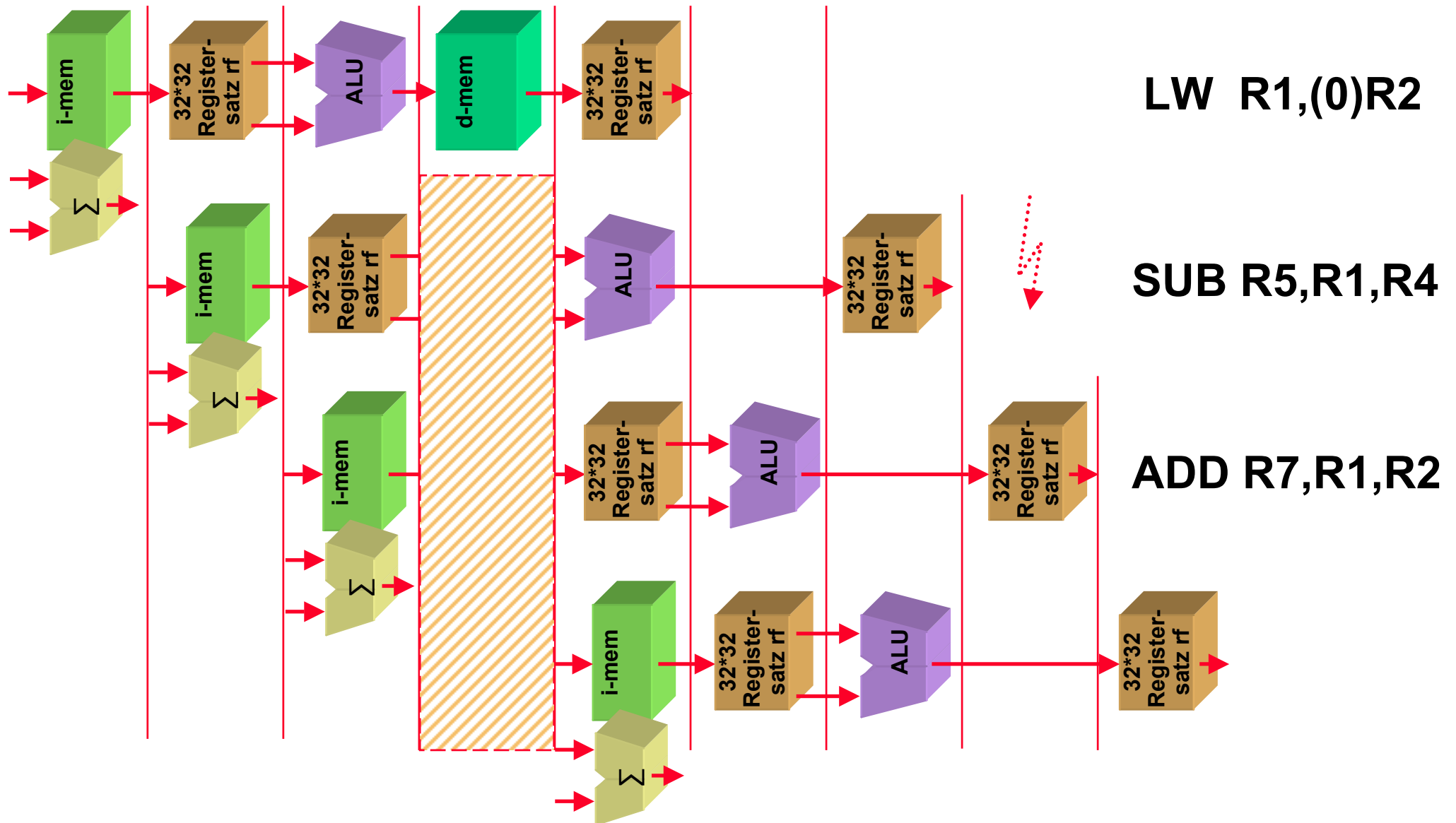
- Weitere Hazards: RAW-Hazard **nach LOAD-Befehlen**
 - Annahme: $ir3[11:15] = ir[6:10]$



4.5 Pipelining der DLX

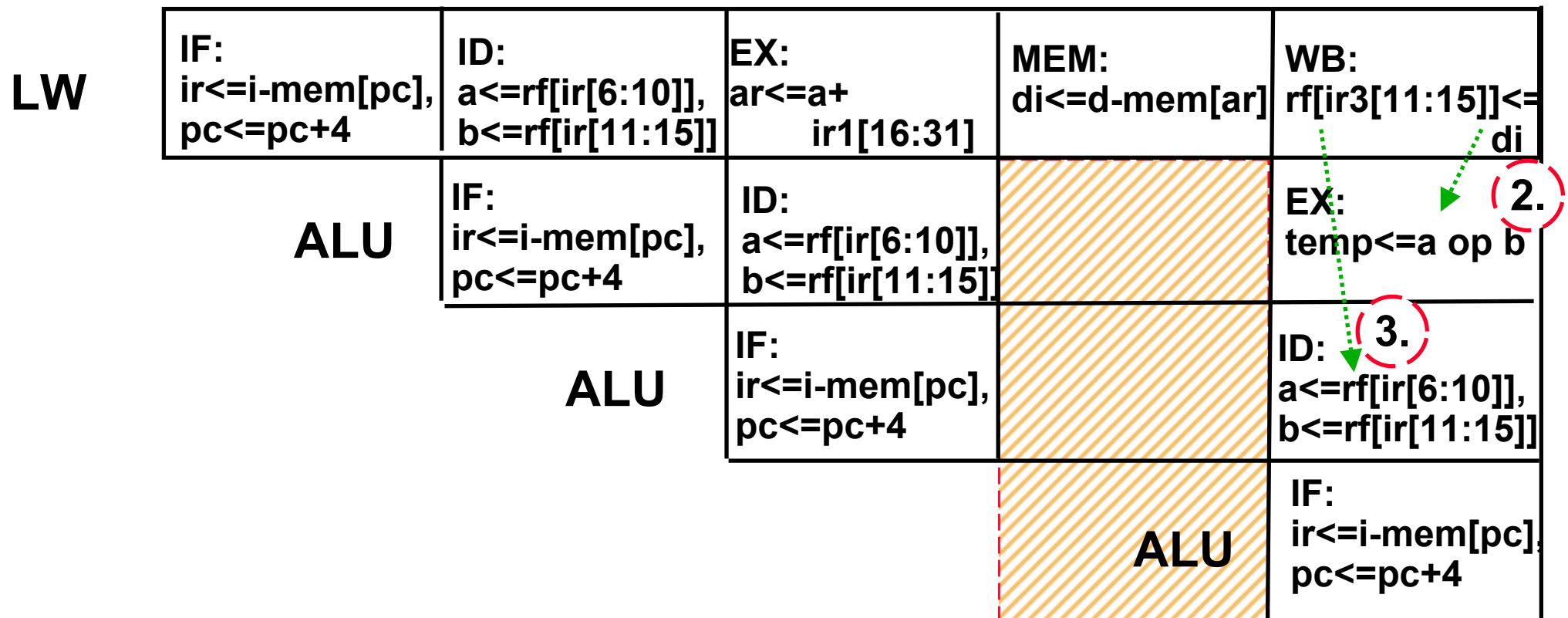


4.5 Pipelining der DLX

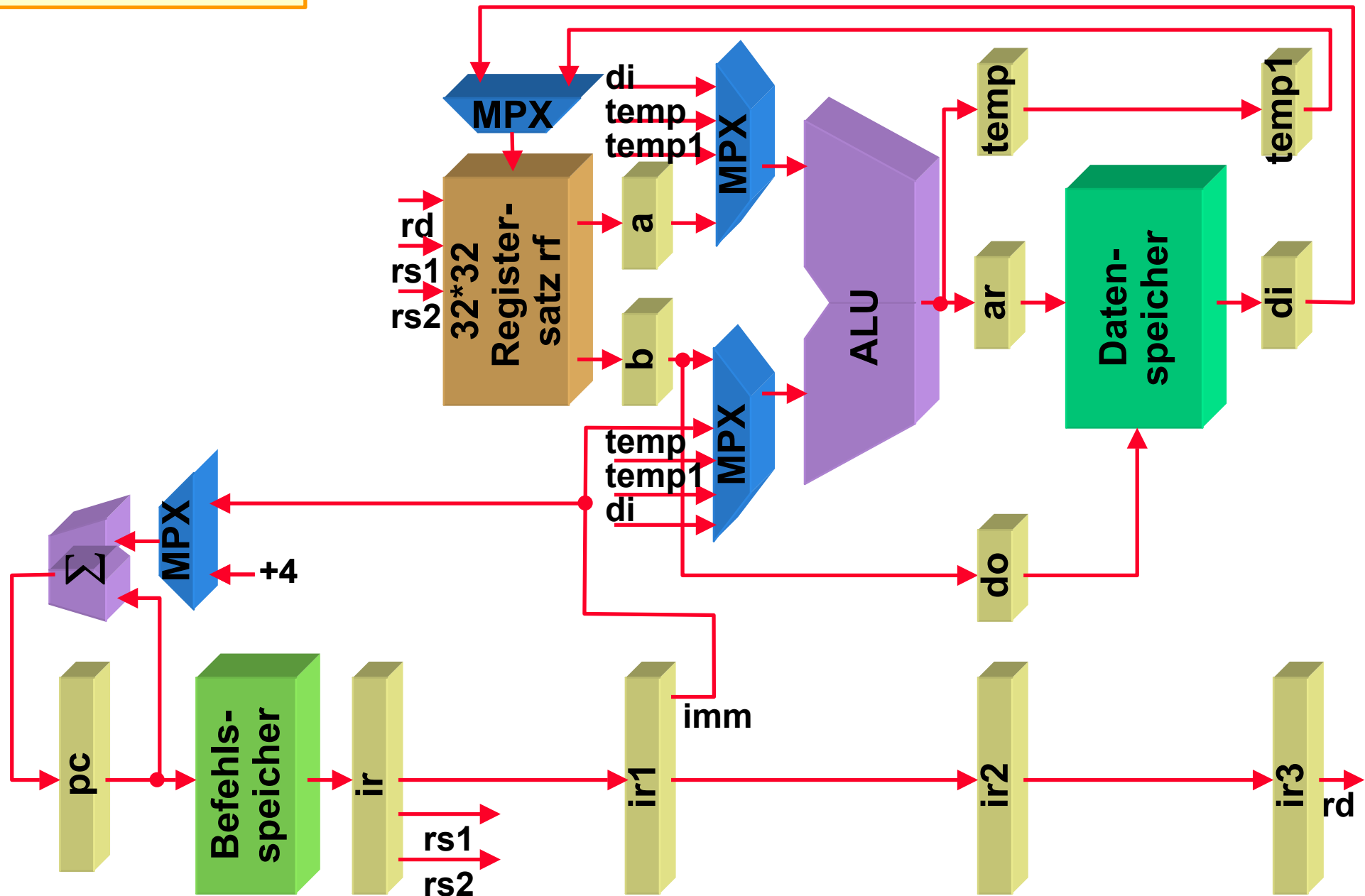
● Einfügen von **Stalls**:

4.5 Pipelining der DLX

- Forwarding von di an die ALU, Durchschleifen des Registersatzes:



4.5 Pipelining der DLX

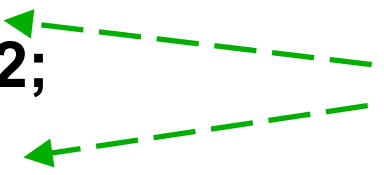


4.5 Pipelining der DLX

- Pipelinestalls können oft **durch den Compiler** vermieden werden

— Beispiel Zuweisung $x = a+b+c$;

LW: R1<= a;
LW: R2<= b;
ADD: R3<= R1+R2;
LW: R4<= c;
ADD: R3<= R3+R4;
SW: x<= R3;



2 Pipelinestalls

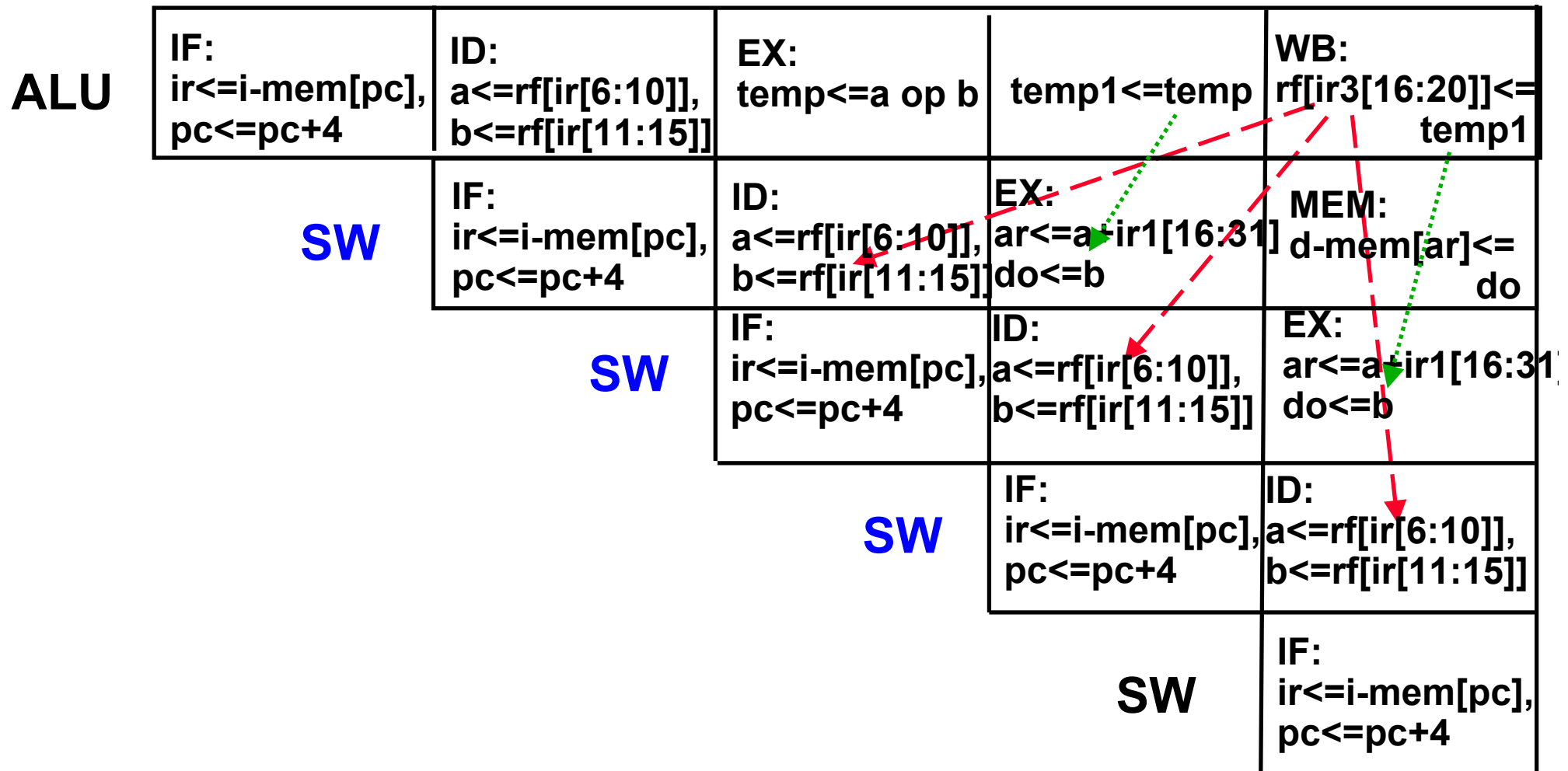
LW: R1<= a;
LW: R2<= b;
LW: R4<= c;
ADD: R3<= R1+R2;
ADD: R3<= R3+R4;
SW: x<= R3;

kein Pipelinestall !

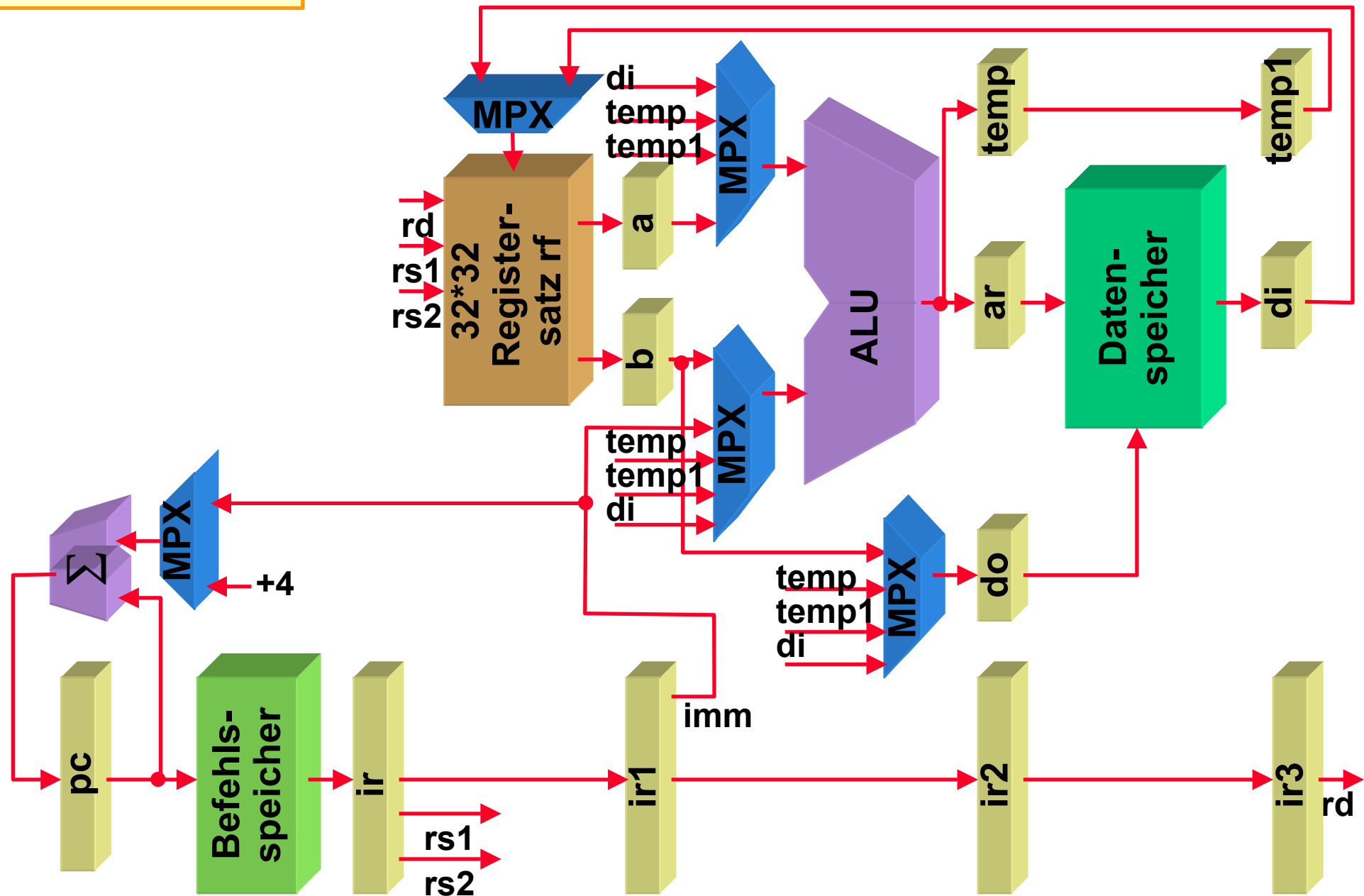
4.5 Pipelining der DLX

- Weitere Hazards: RAW-Hazard **bei STORE-Befehlen**

➤ Annahme: $ir3[11:15] = ir[11:15]$



4.5 Pipelining der DLX



4.5 Pipelining der DLX

- Beispiel ARM9E core:

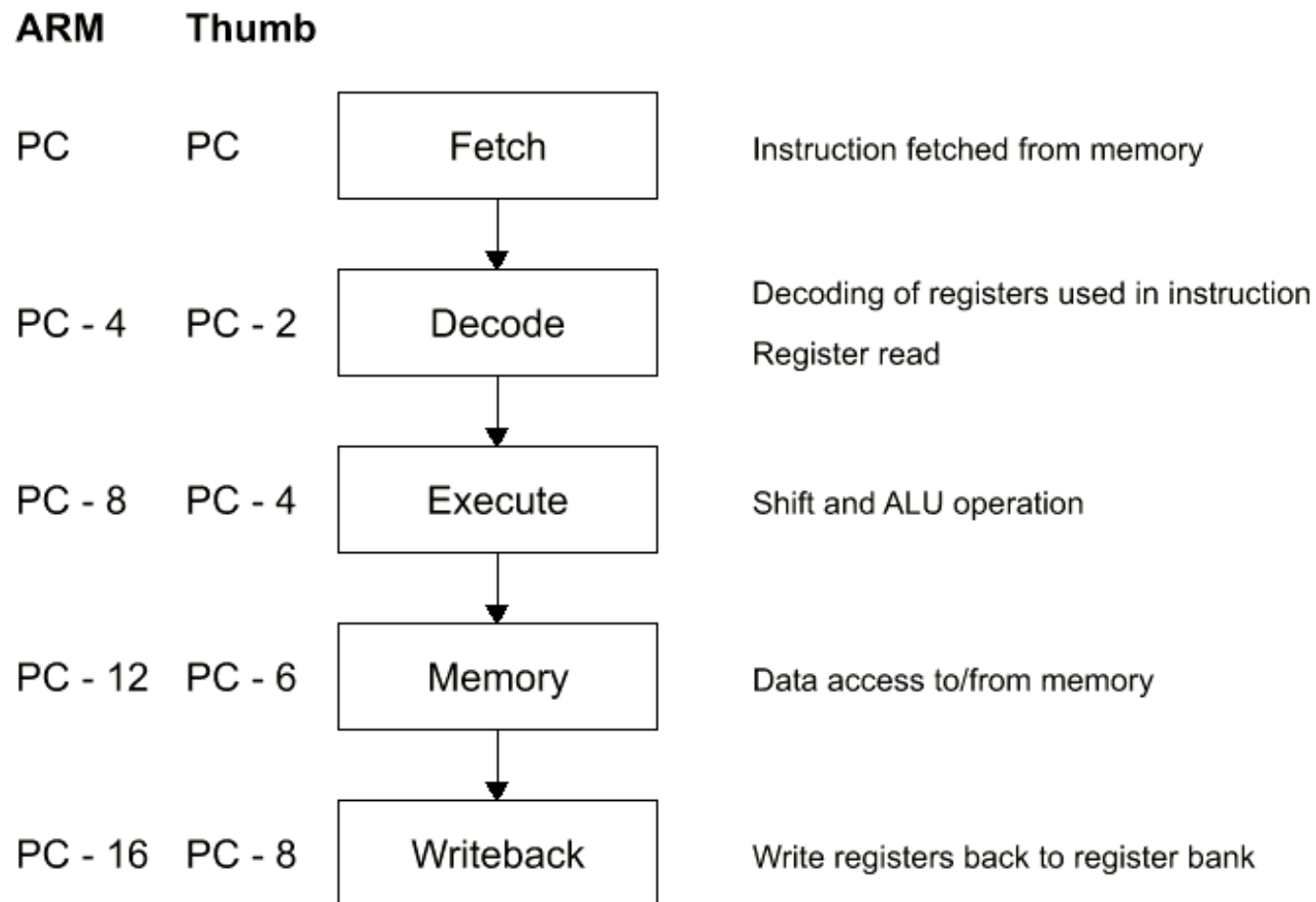


Figure 1-1 Five-stage pipeline

4.5 Pipelining der DLX

Integer core	ARM7TDMI	ARM9TDMI	ARM9E-S	ARM10E
Architecture Version	ARMV4T	ARMV4T	ARMV5TE	ARMV5TE
Pipeline/Bus Architecture	3 stage, Von Neumann	5 stage, Harvard	5 stage, Harvard	6 stage, Harvard
Typical Die Size** mm ²	0.54 (0.18u, 4LM)	1.1 (0.18u, 4/5LM)	1.3 (0.18u, 4/5LM)	7.5 (0.18u, 5LM)
Power Consumption (Ave) mW/MHz	0.25	0.36	1.0	1.5
Clock Speed** MHz (worst case)	90	180	200	225
Cycle per Instruction	1.9	1.5	1.5	1.2
Core Derivatives	720T, 740T	920T, 922T, 940T	966E-S, 946E-S	1020E, 10200

** Numbers will vary with partner process.

4.5 Pipelining der DLX

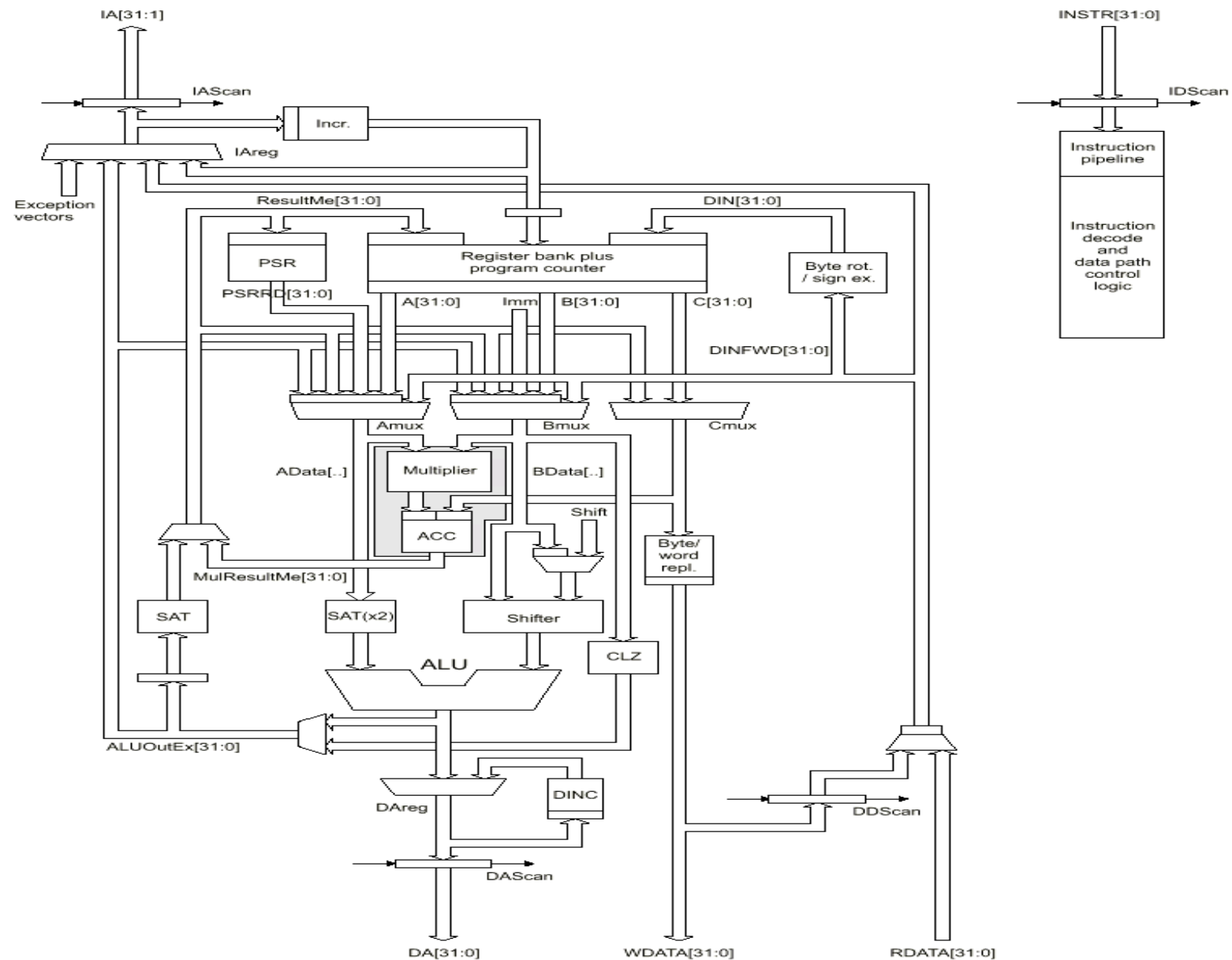


Figure 1-4 ARM9E-S core

4.5 Pipelining der DLX

- **Konfliktarten:**

- **Ressourcenkonflikte**

- ein Hilfsmittel wie z.B. Funktionsblock oder Speicher soll für unterschiedliche Operationen benutzt werden

- **Datenkonflikte**

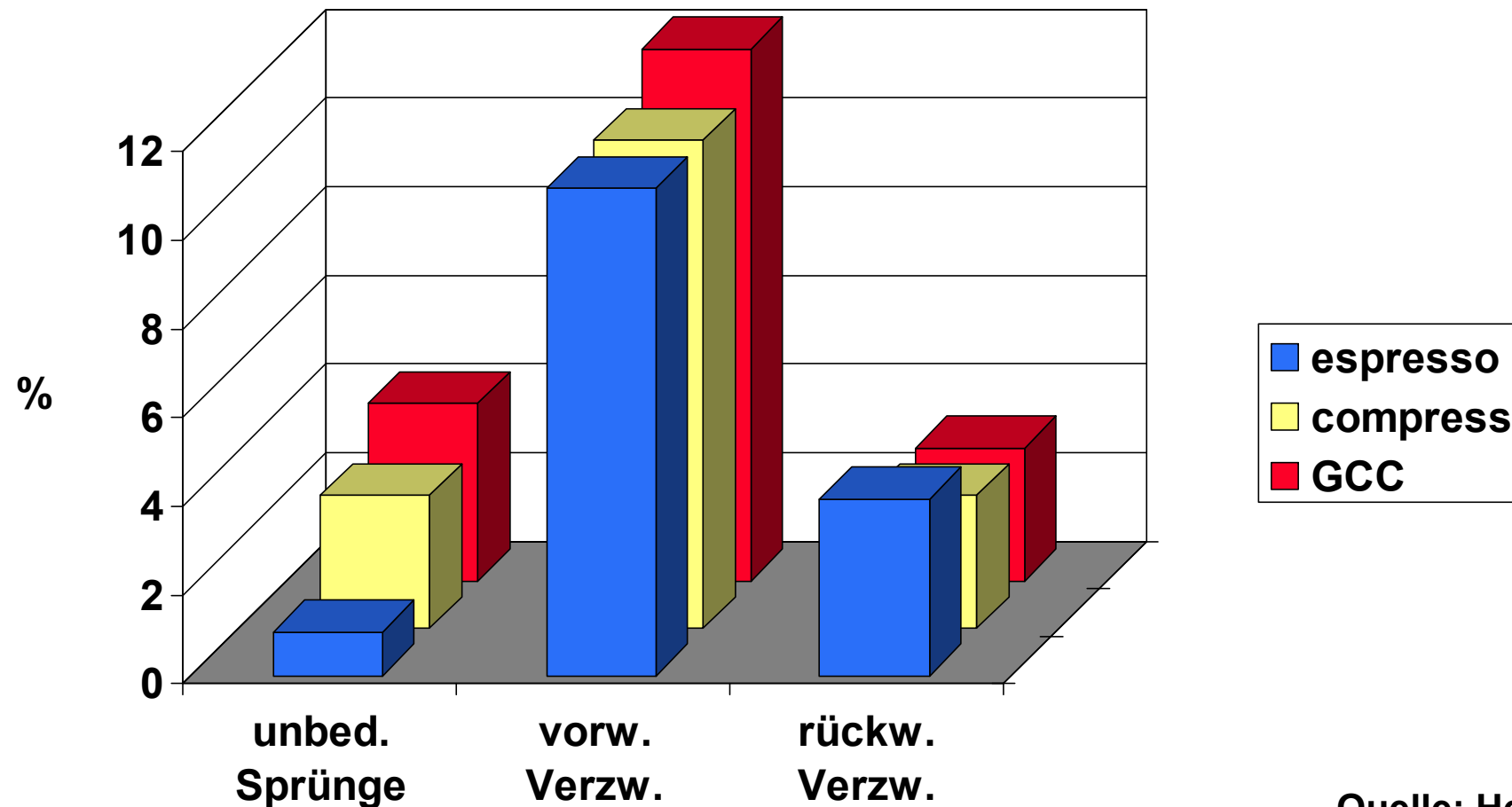
- auf Transferebene: Registerinhalte, die in einem Schritt benutzt werden, stehen nicht zur Verfügung
- auf Befehlsebene: Daten, die in einem Befehl benutzt werden sollen, stehen nicht zur Verfügung

- **Steuerkonflikte**

- die Pipeline muß wegen **Verzweigungen** geleert und neu gefüllt werden

4.5 Pipelining der DLX

- Häufigkeit unbedingter Sprünge und bedingter Verzweigungen vorwärts bzw. rückwärts



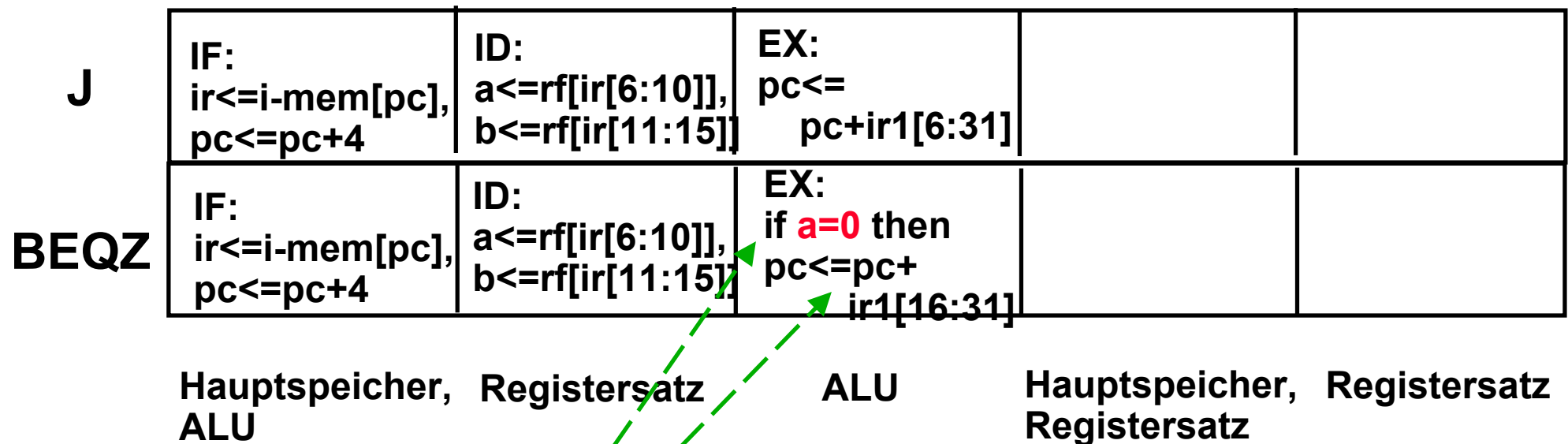
Quelle: Hen./Pat.

4.5 Pipelining der DLX

- Im Mittel werden
 - 60% aller Vorwärtsverzweigungen
 - 85% aller Rückwärtsverzweigungen (Schleifen !)
ausgeführt
 - ➡ ungefähr 13% aller Befehle
(10%: Verzweigungen, 3%: Sprünge) führen
daher zu einer Änderung des Kontrollflusses

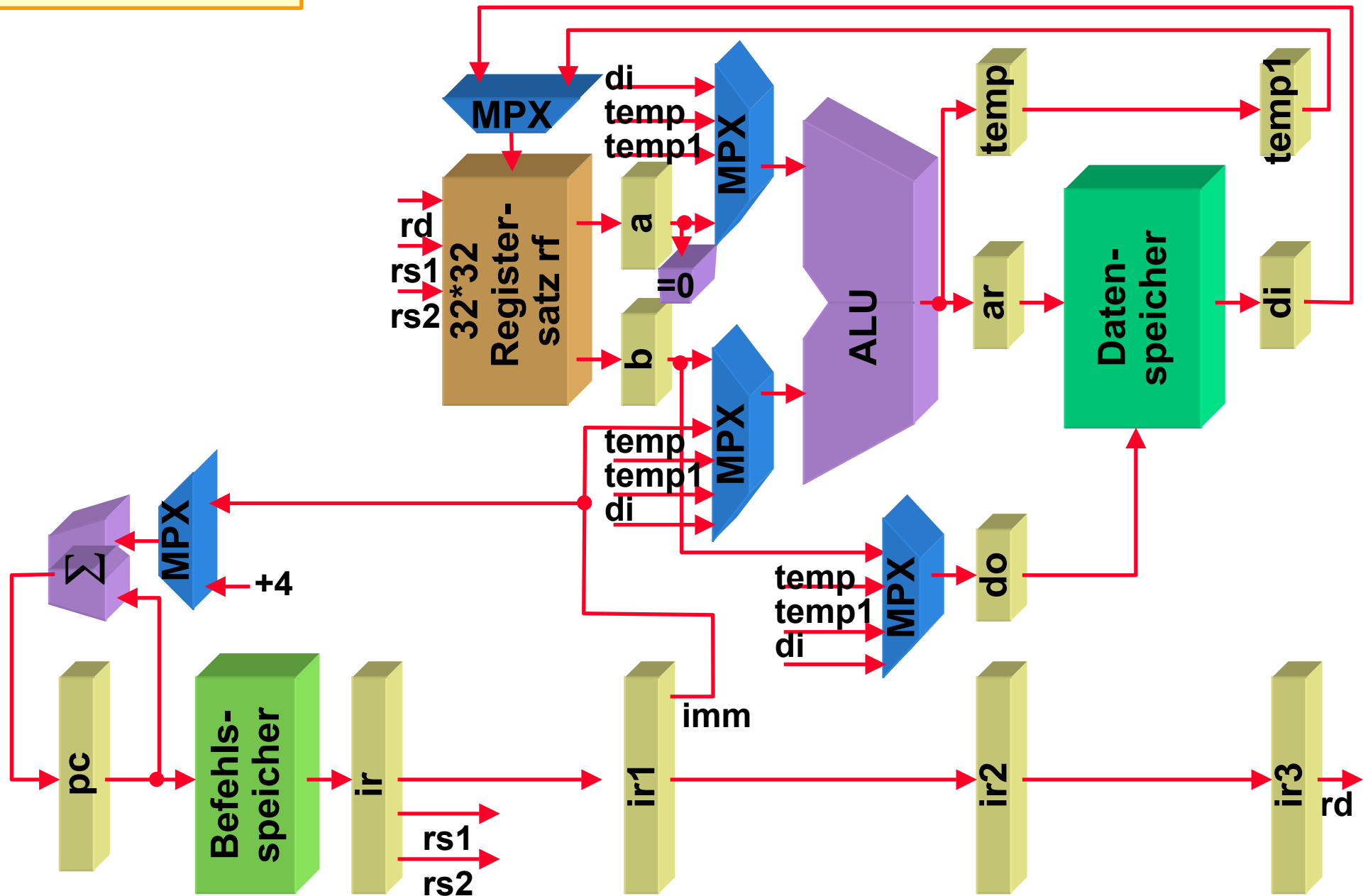
4.5 Pipelining der DLX

- Sprung- und Verzweigungsbefehle:



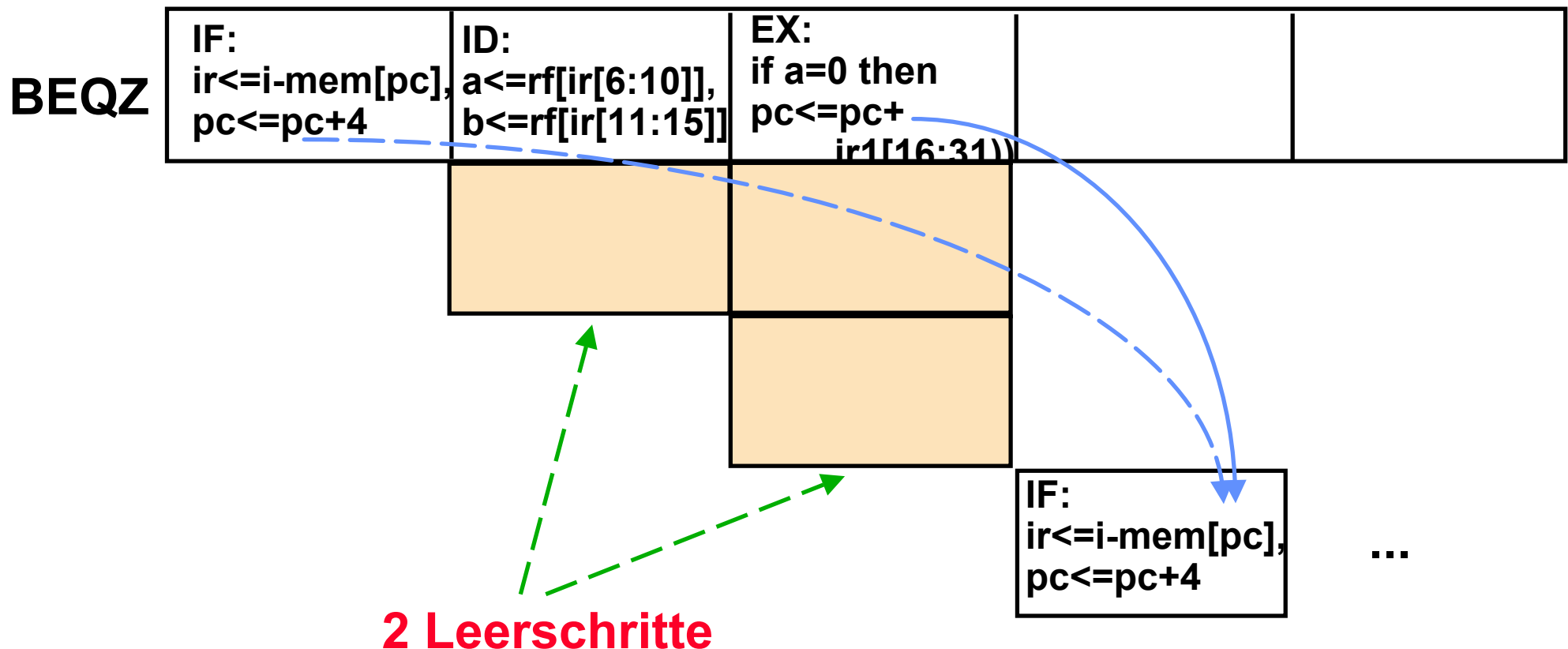
die Ausführung in einem Schritt bedeutet,
daß es neben der ALU noch einen **Vergleicher**
a = 0 gibt

4.5 Pipelining der DLX



4.5 Pipelining der DLX

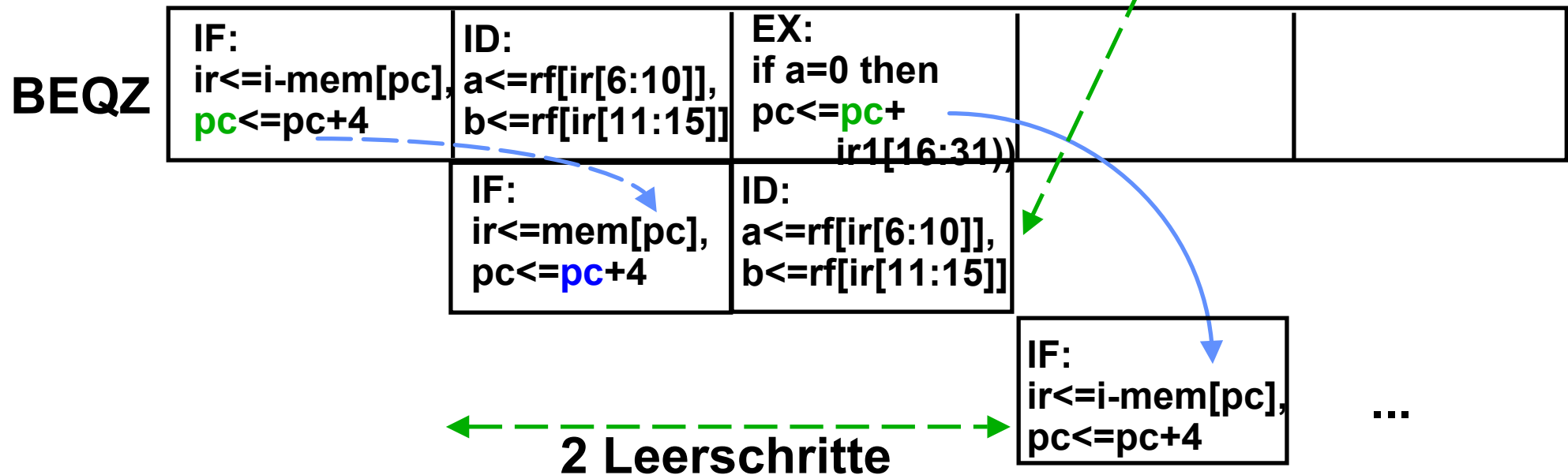
- Strategien für Verzweigungen bei der DLX:
 - 1. Lösung: immer **2 Leerschritte**



4.5 Pipelining der DLX

- 2. Lösung: nur bei Verzweigung 2 Leerschritte ("*predict-not-taken*")

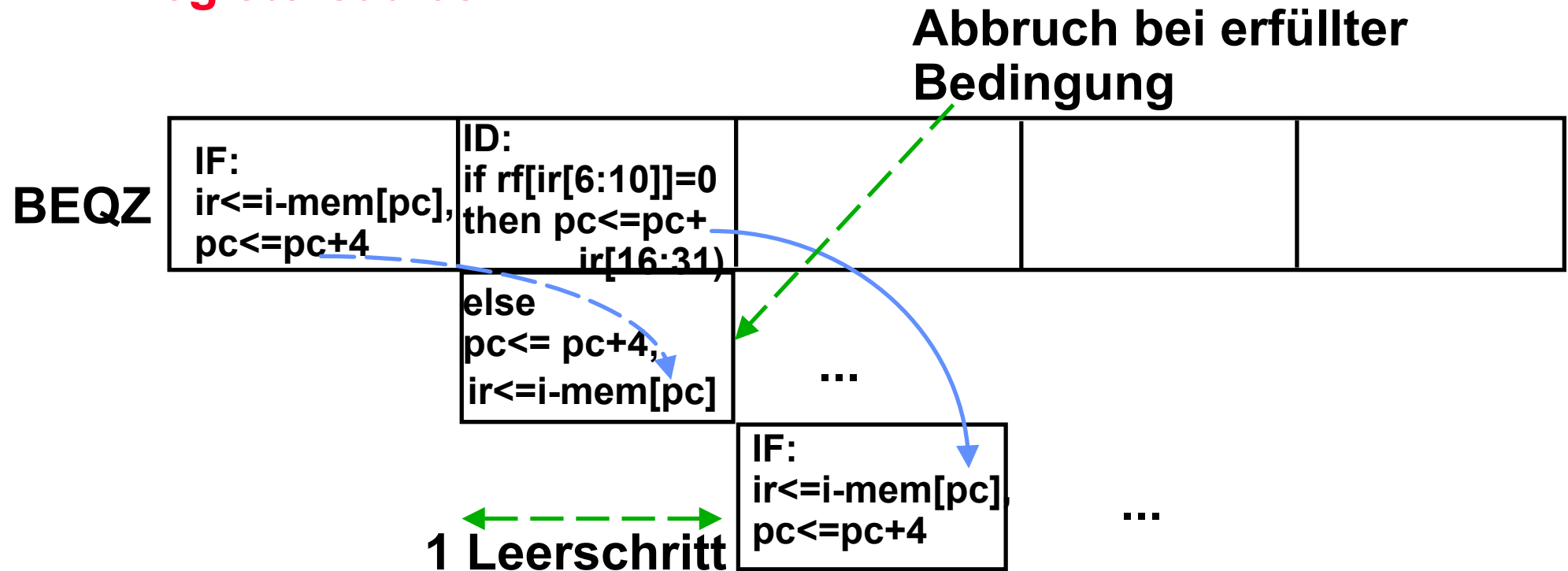
Abbruch bei erfüllter
Bedingung



- ☞ wichtig ist, daß in der ID-Phase der Prozessorzustand nicht modifiziert wird
- ☞ Pipelineregister für pc notwendig !

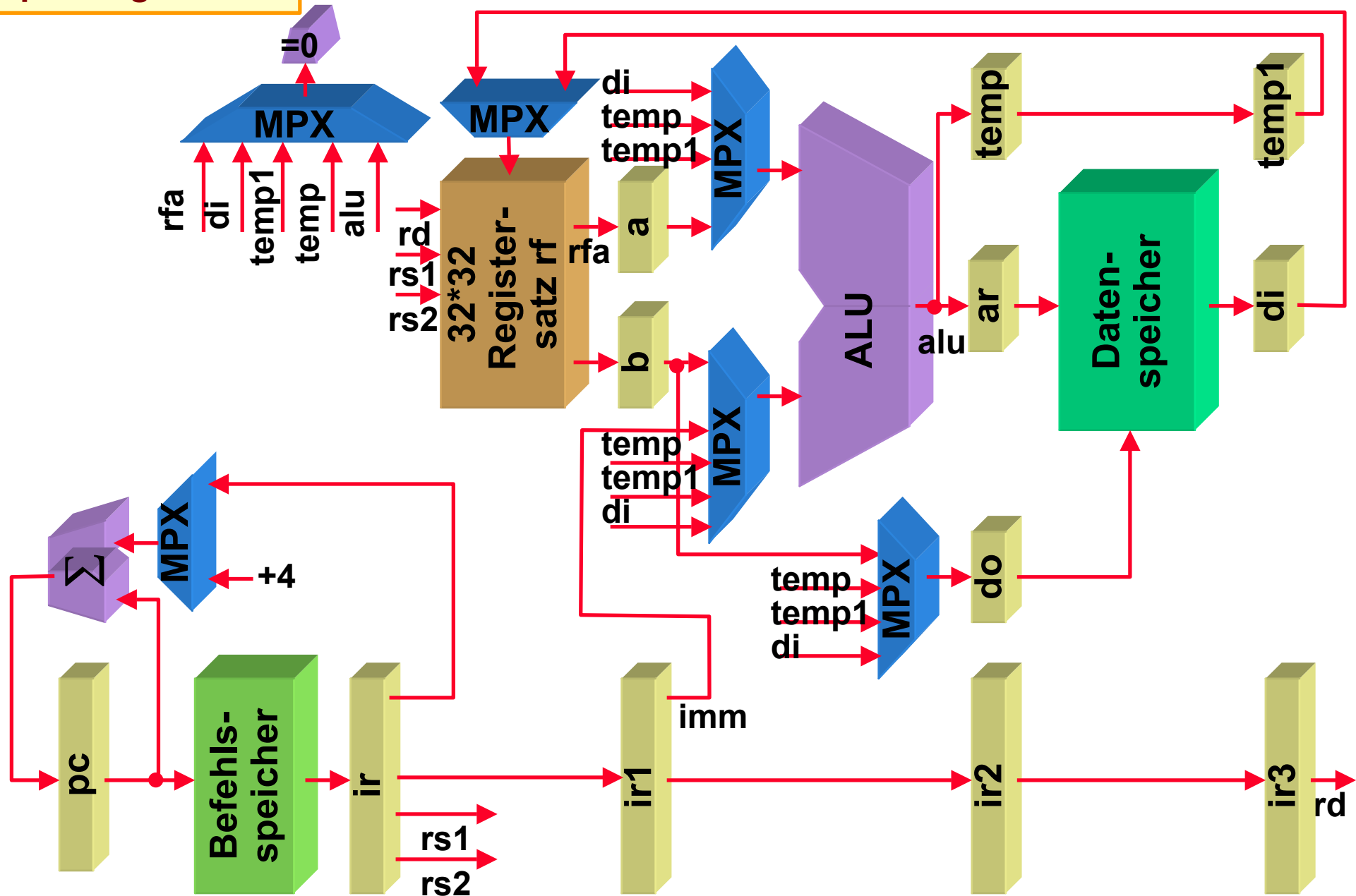
4.5 Pipelining der DLX

- 3. Lösung: 2. Lösung + Abfrage = 0 **am Ausgang des Registersatzes**

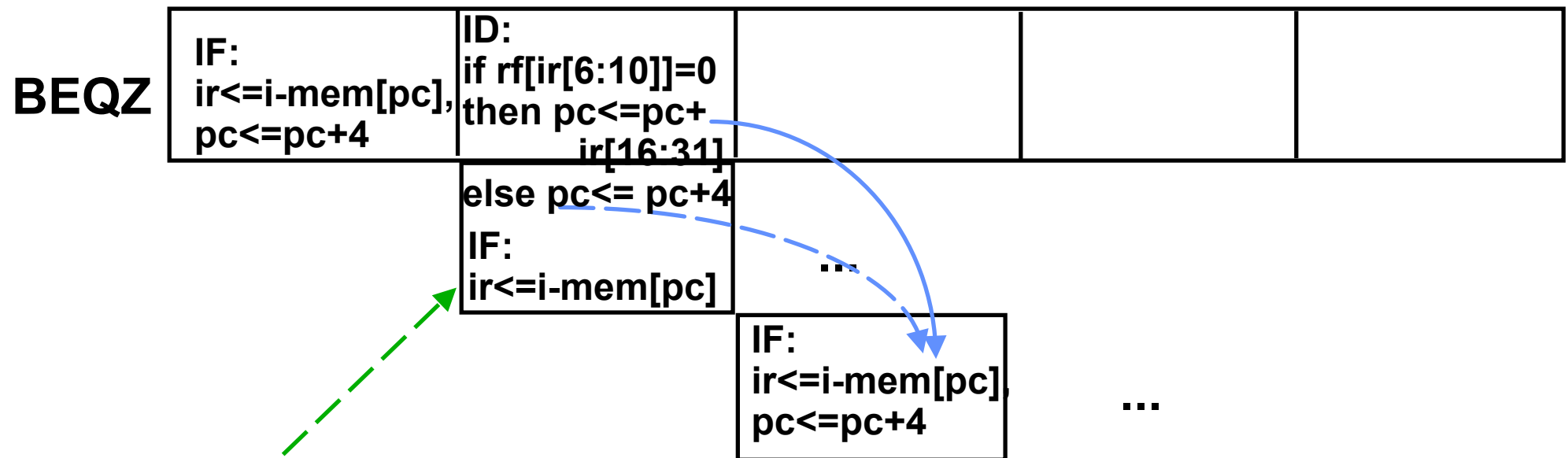


- ☞ damit ist der Verlust auf **1 Leerschritt** reduziert, falls die Verzweigung stattfindet
- ☞ ferner ist ein Forwarding für **rf[ir[6:10]]** von allen Befehlsphasen notwendig !

4.5 Pipelining der DLX



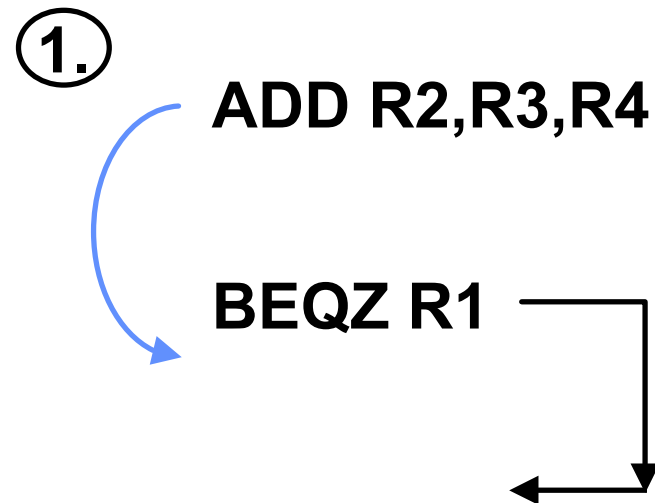
4.5 Pipelining der DLX

➤ 4. Lösung: verzögerte Verzweigung ("*delayed branch*")

nützlicher Befehl, der
immer ausgeführt wird

4.5 Pipelining der DLX

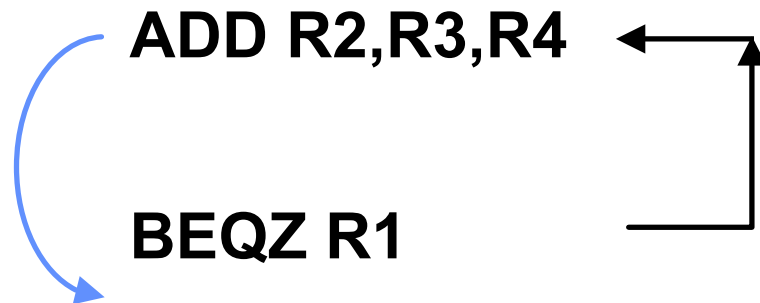
- Hardware der 3. Lösung, Einfügen einer auf die Verzweigung folgenden Anweisung **durch den Compiler**, die **immer** ausgeführt wird (notfalls NOOP)
- hierbei gibt es drei Möglichkeiten für den Compiler, Anweisungen auf die der Verzweigung folgende Adresse zu verschieben:



Voraussetzung: Verzweigung hängt nicht von der Anweisung ab.
Nützlich: immer

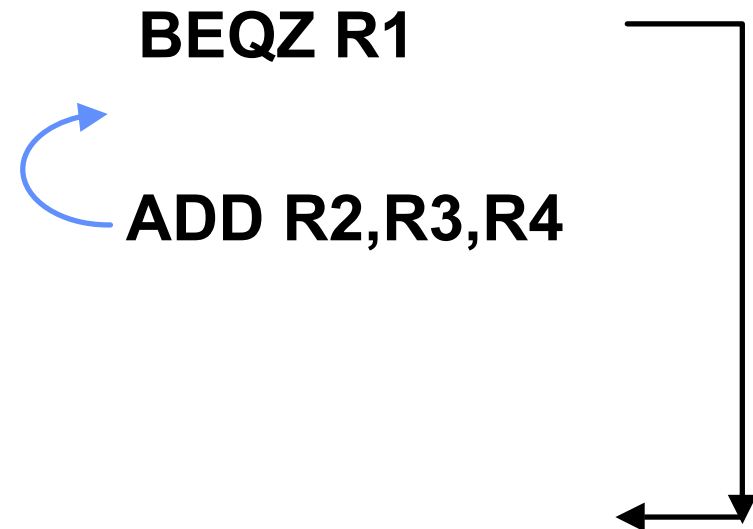
4.5 Pipelining der DLX

②.



Voraussetzung: Anweisung kann ohne Nebenwirkungen ausgeführt werden auch ohne Verzweigung.
Nützlich bei Verzweigung

③.



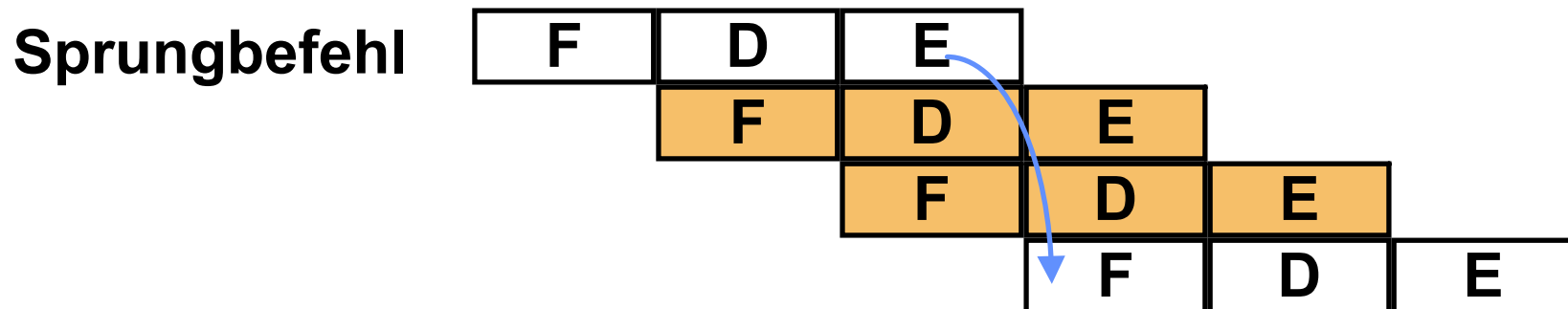
Voraussetzung: Anweisung kann ohne Nebenwirkungen ausgeführt werden auch bei Verzweigung.
Nützlich, falls keine Verzw.

4.5 Pipelining der DLX

- In etwa 70% der Fälle kann im Mittel eine nützliche Anweisung ausgeführt werden
 - ☞ damit nur noch 0,3 Stalls pro Verzweigung im Mittel
 - ☞ Verfahren auch anwendbar bei unbedingten Sprüngen

4.5 Pipelining der DLX

- **Beispiel SHARC ADSP2106x 32 Bit Gleitkomma-DSP (Analog Devices)**
 - dreistufige Pipeline (Fetch, Decode, Execute)
 - durch ein Bit eines Sprungbefehls wird bestimmt, ob die nächsten beiden, auf den Sprungbefehl folgenden Befehle ausgeführt werden oder nicht



4.5 Pipelining der DLX

- Effizienz der verschiedenen Verfahren
 - Leer-Schritte_{mittel} = mittlere Anzahl von Leer-Schritten verursacht durch Verzweigungen = Häufigkeit von Verzweigungen * #Leer-Schritte bei Verzweigungen
 - Beschleunigung durch Pipelining

$$B_{\text{pipe}} = \frac{\text{\#Pipelinstufen}}{1 + \text{Leer-Schritte}_{\text{mittel}}}$$

- z.B. $5/(1 + 13\%*2) = 3,97$, aber $5/(1 + 13\%*0,3) = 4,81$

Lösung 1

Lösung 4

👉 hierbei vorausgesetzt, daß CPI sonst = 1

4.5 Pipelining der DLX

- Beispiel: die Bedingung eines Verzweigungsbefehls wird erst im **10. Schritt** berechnet

☞ entsprechend hoher Verlust bei falsch vorausgesagter Verzweigung

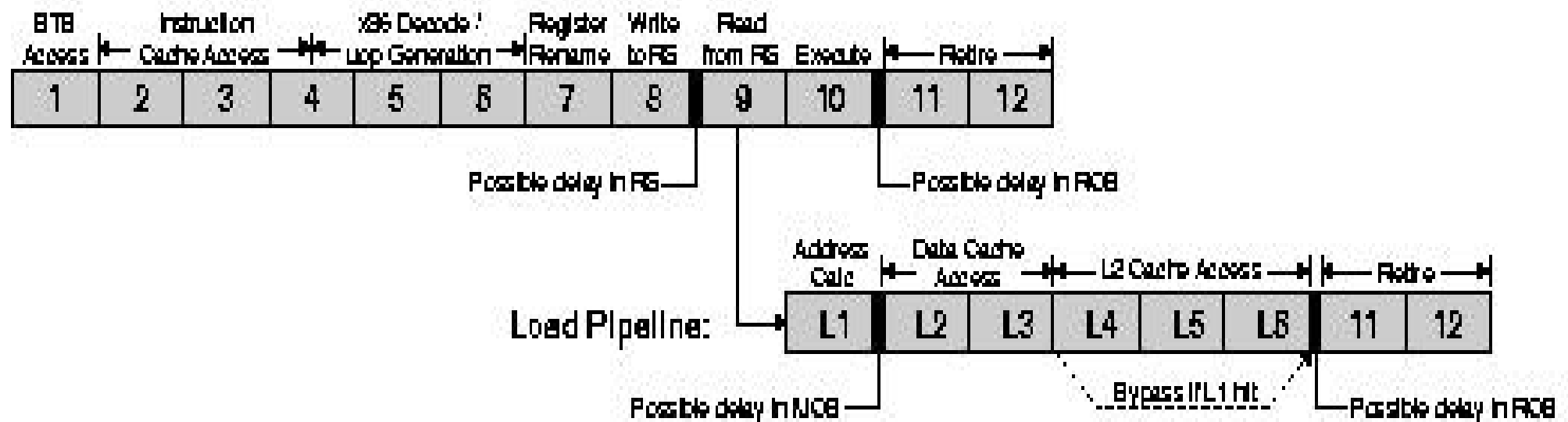
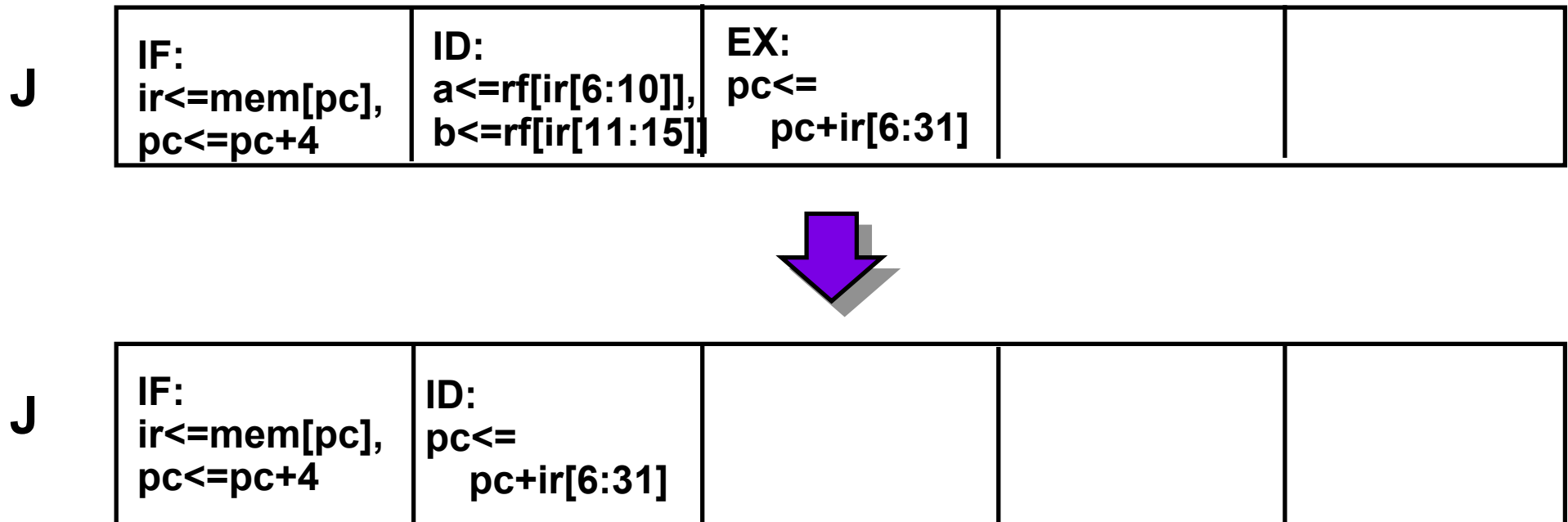


Figure 3. In the best case, instructions can flow through the P8 in 12 cycles, but the average is 18 cycles due to delays in the reservation station (RS) or the reorder buffer (ROB). Load instructions take longer and can also be delayed in the memory reorder buffer (MOB).

Quelle: Microprocessor Report
<http://www.chipanalyst.com/q/report/index.html>

4.5 Pipelining der DLX

- Unbedingte Sprungbefehle:

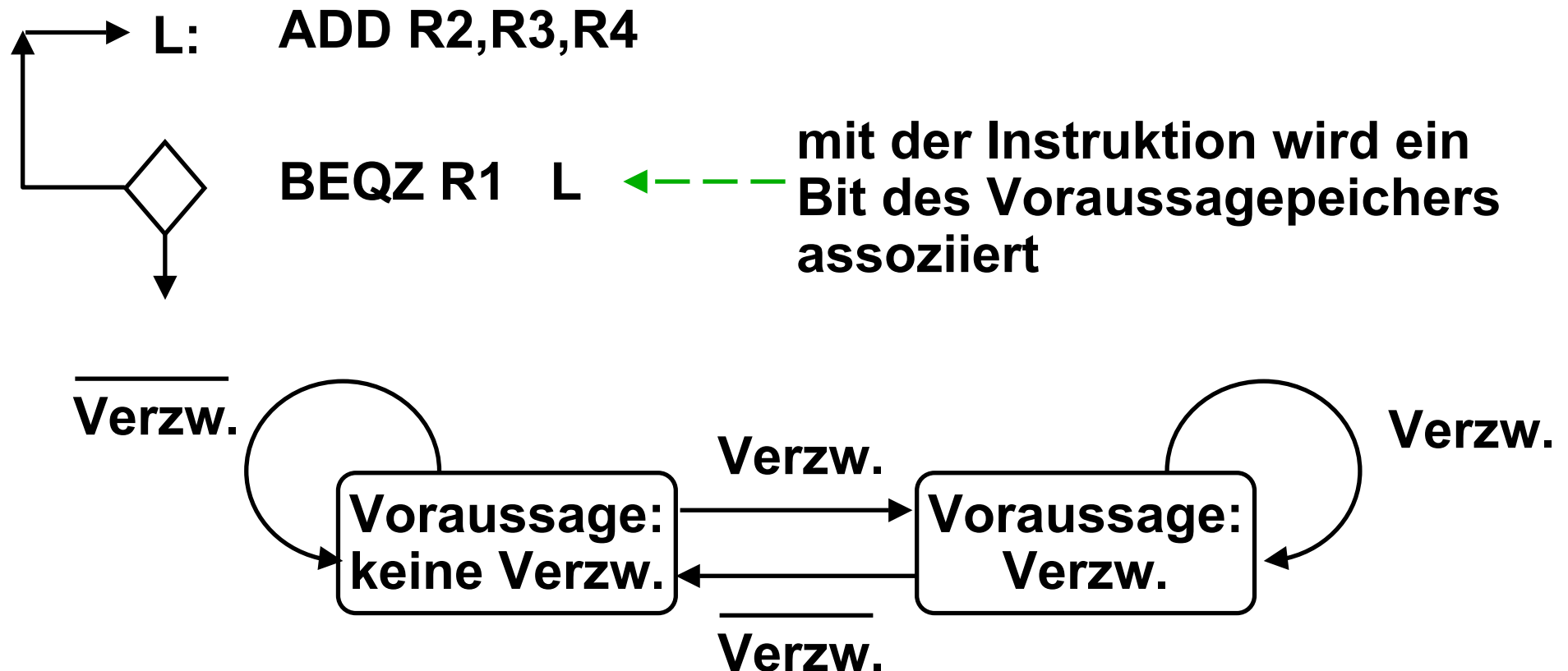


4.5 Pipelining der DLX

- **Umgang mit Steuerkonflikten**
 - **Statische** Verfahren (s.o.)
 - **Dynamische** Verfahren ("*branch prediction*"):
 - ob eine Verzweigung stattfinden wird oder nicht, wird aufgrund von Aufzeichnungen über vergangene Verzweigungen vorausgesagt
 - anhand der Voraussage wird der nächste Befehl geholt
 - in Abhängigkeit von der Verzweigungsbedingung wird dies gegebenenfalls korrigiert
 - die Aufzeichnungen werden in einem Voraussage-speicher ("*branch prediction buffer*") gehalten

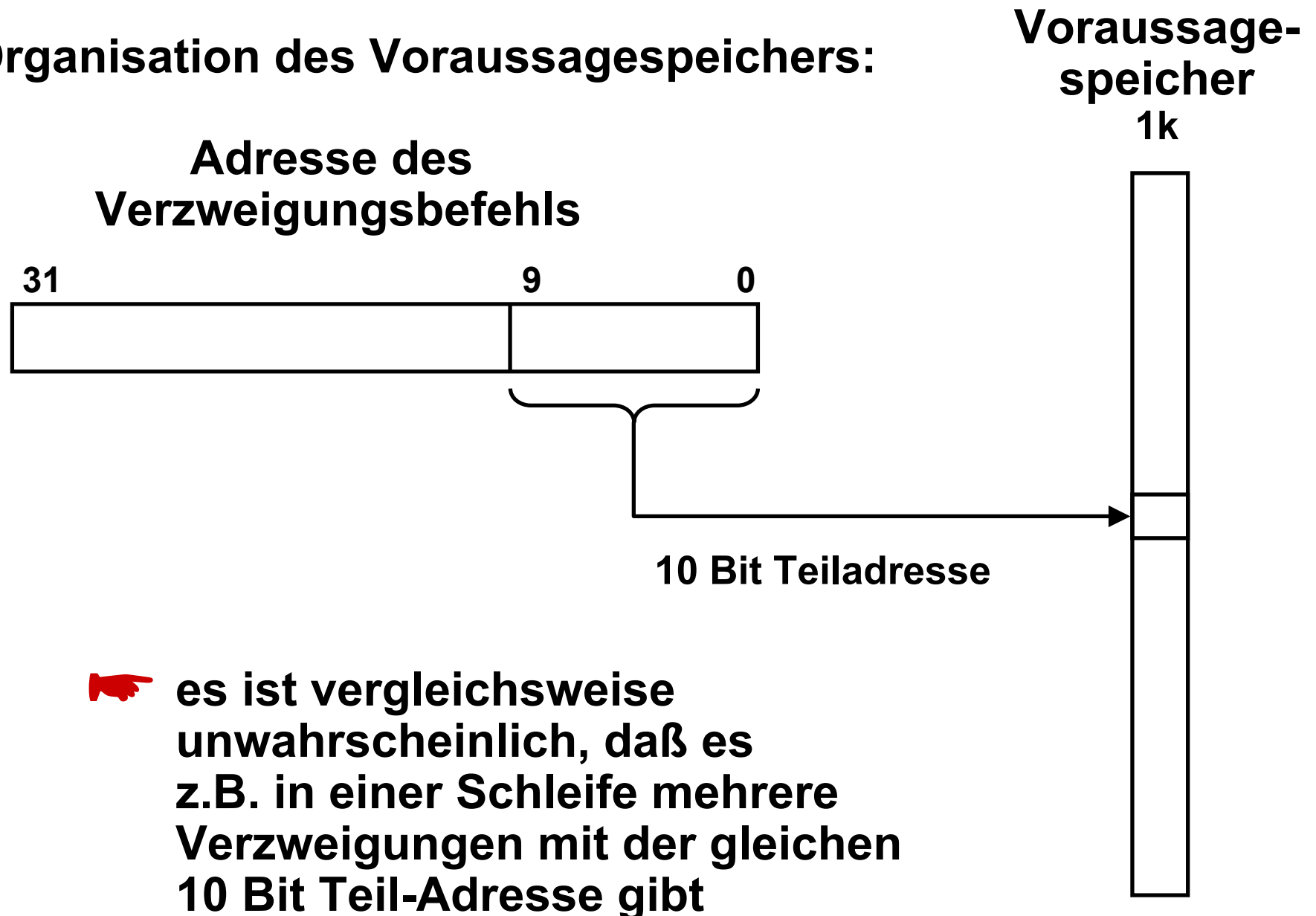
4.5 Pipelining der DLX

— Beispiel: Schleife, 1 Bit Lösung:



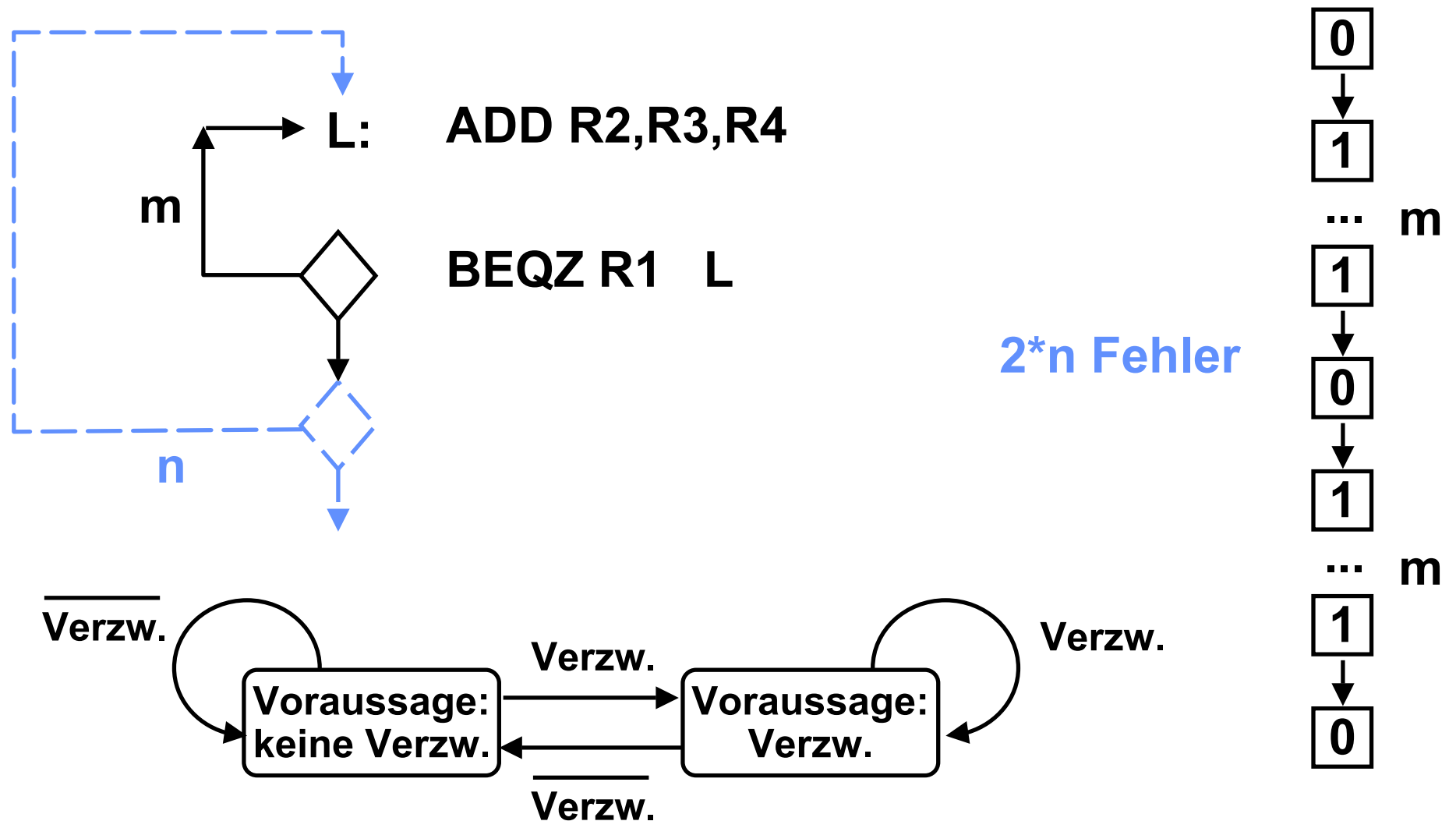
4.5 Pipelining der DLX

➤ Organisation des Voraussagespeichers:



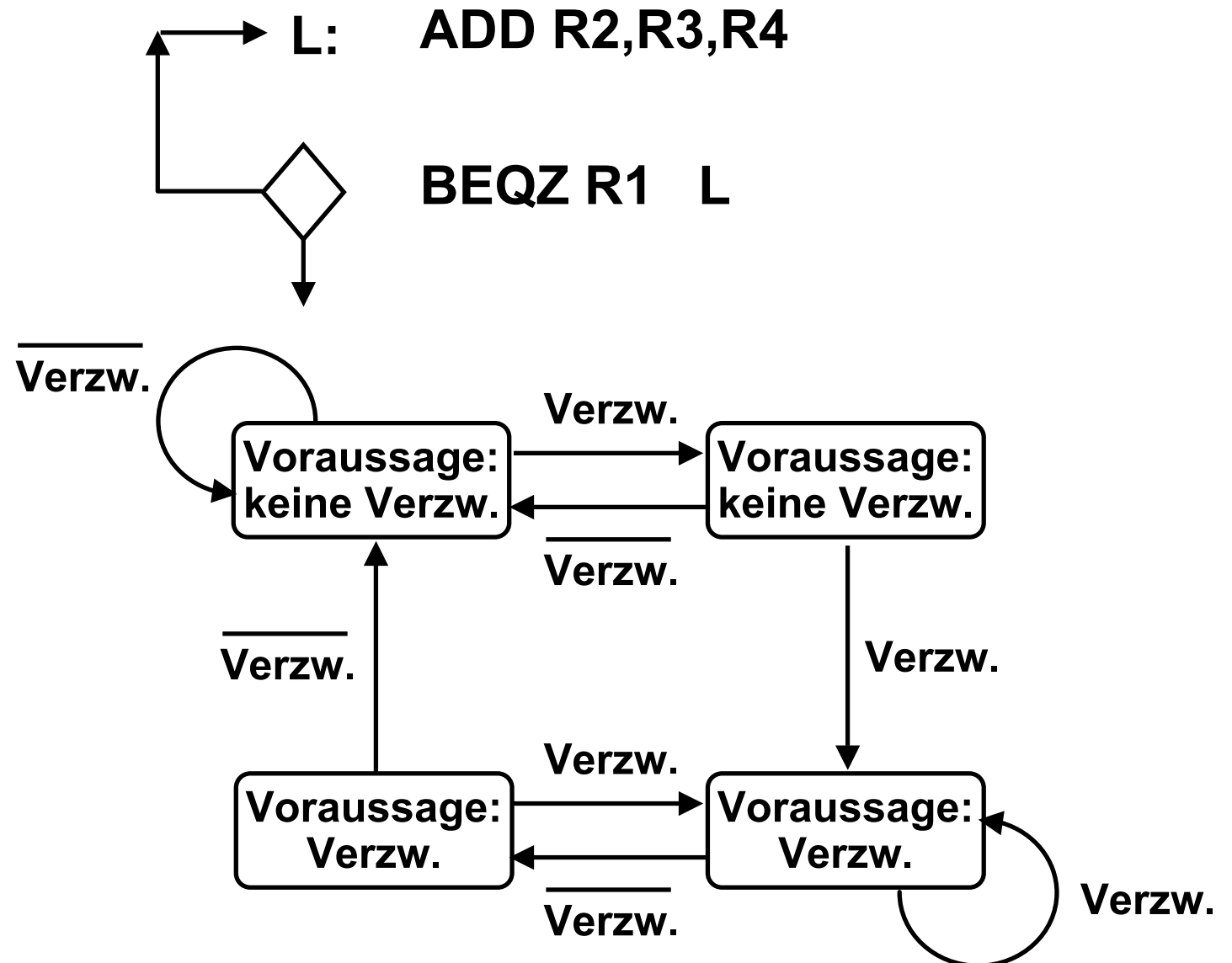
4.5 Pipelining der DLX

- Problem bei nur einem Voraussagebit : zwei falsche Voraussagen pro Schleifendurchlauf bei mehreren Iterationen

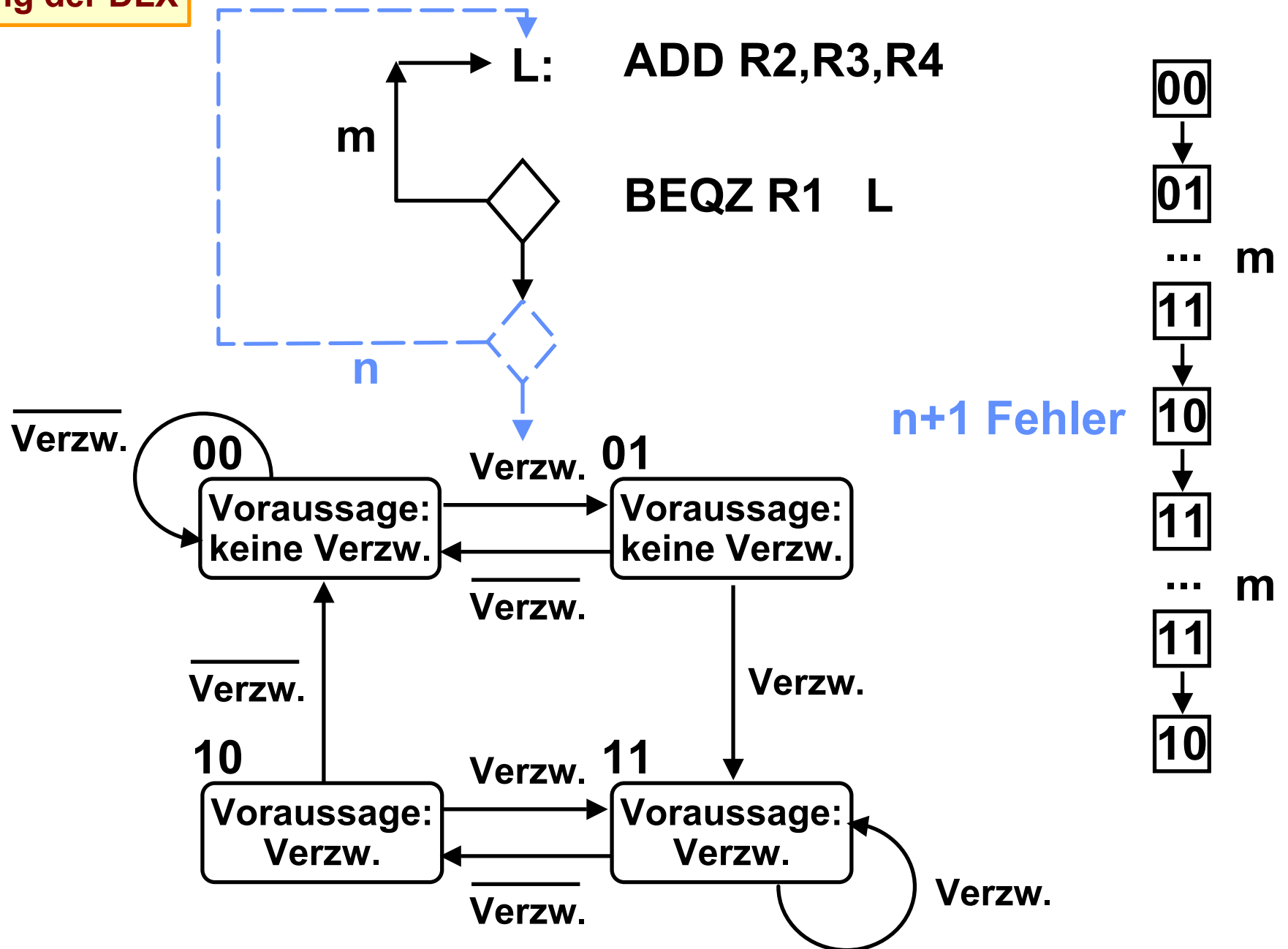


4.5 Pipelining der DLX

➤ Lösung mit 2 Bit:

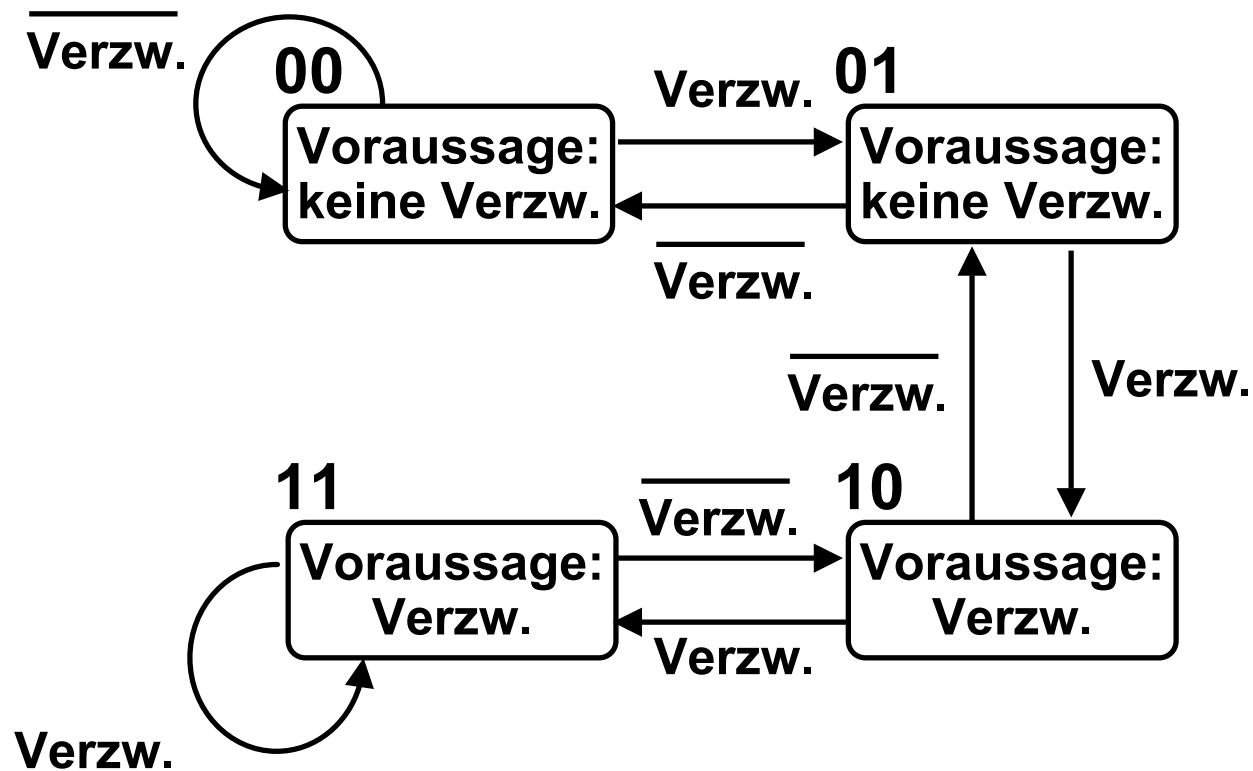


4.5 Pipelining der DLX



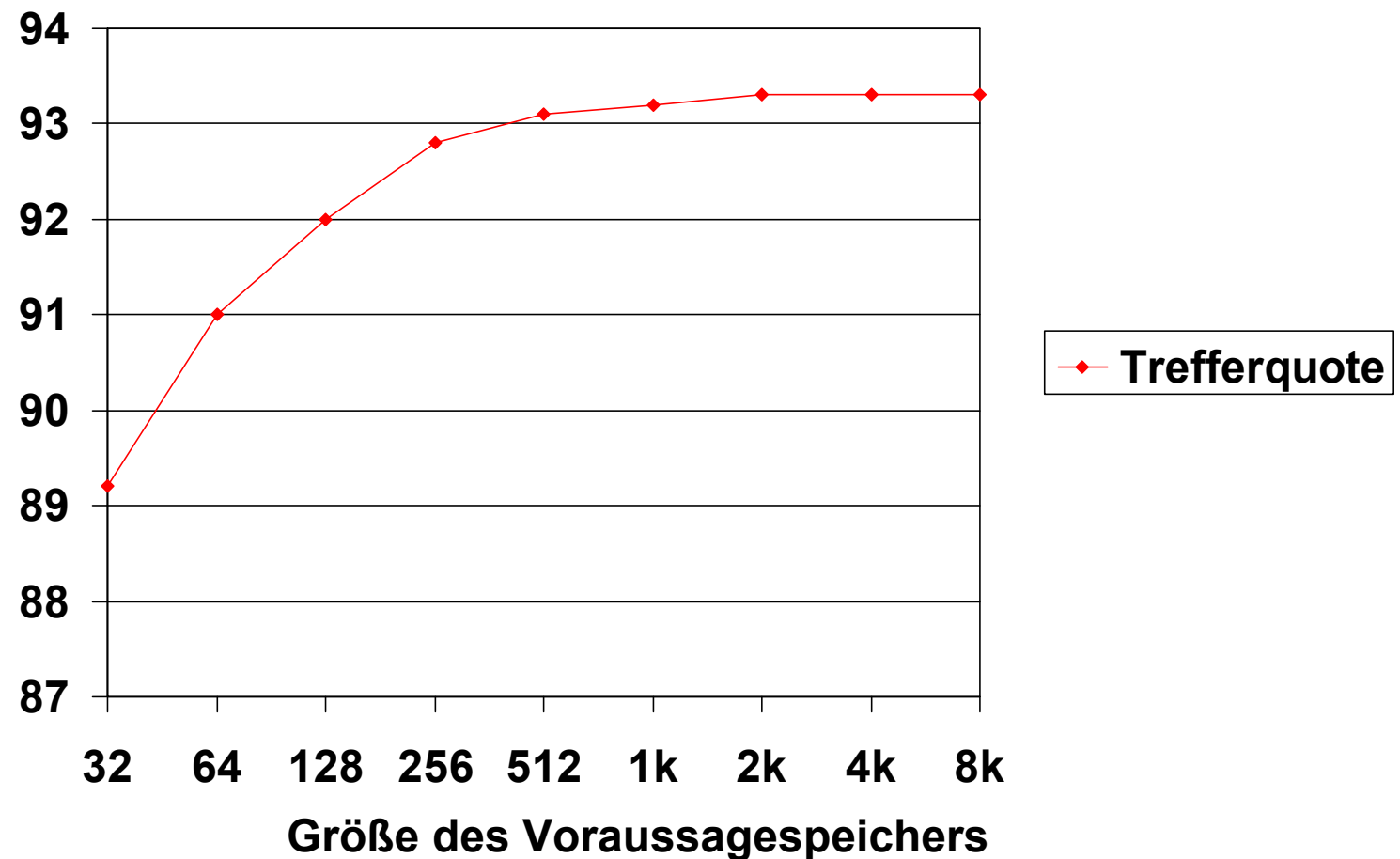
4.5 Pipelining der DLX

- Variante: Sättigungszähler



4.5 Pipelining der DLX

- Experiment (SPEC89 Benchmark, ca. 10^7 Befehle)

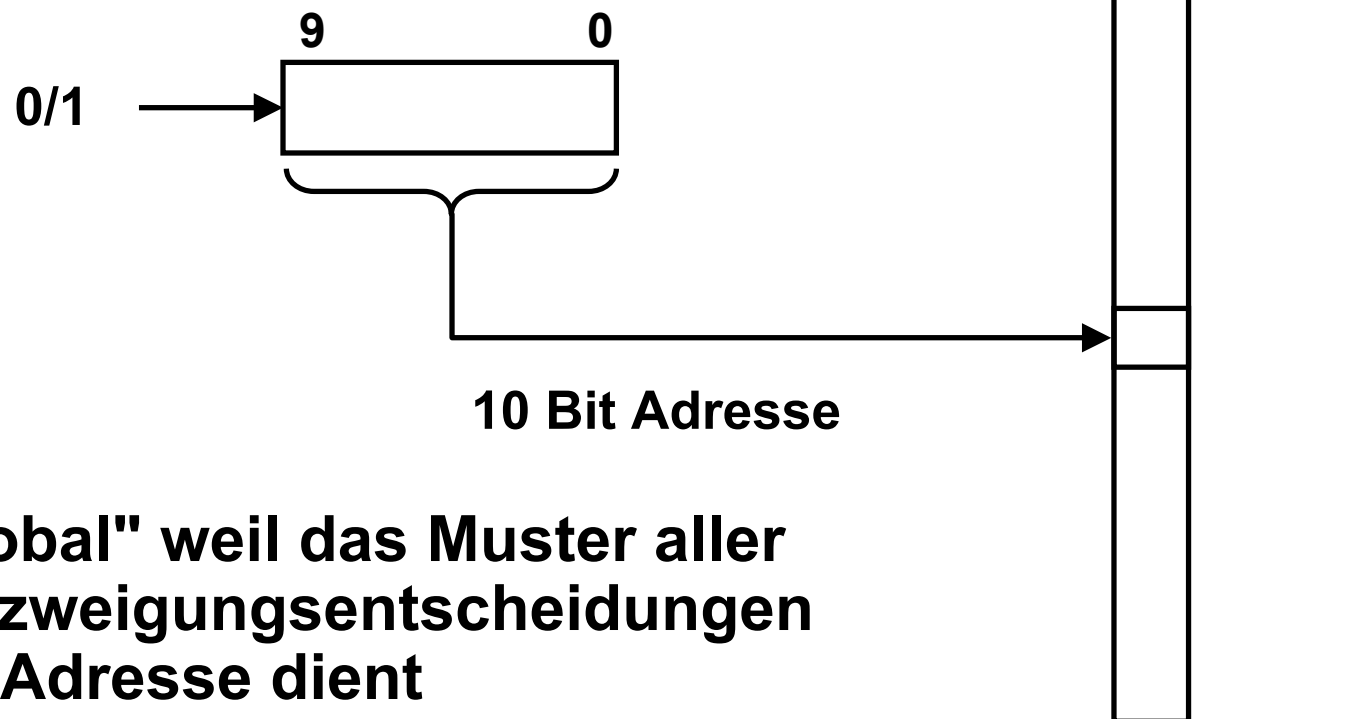


4.5 Pipelining der DLX

- Die oben vorgestellten Verfahren machen **lokale** Voraussagen der Verzweigung, da die Historie des Einzelbefehls zählt

4.5 Pipelining der DLX

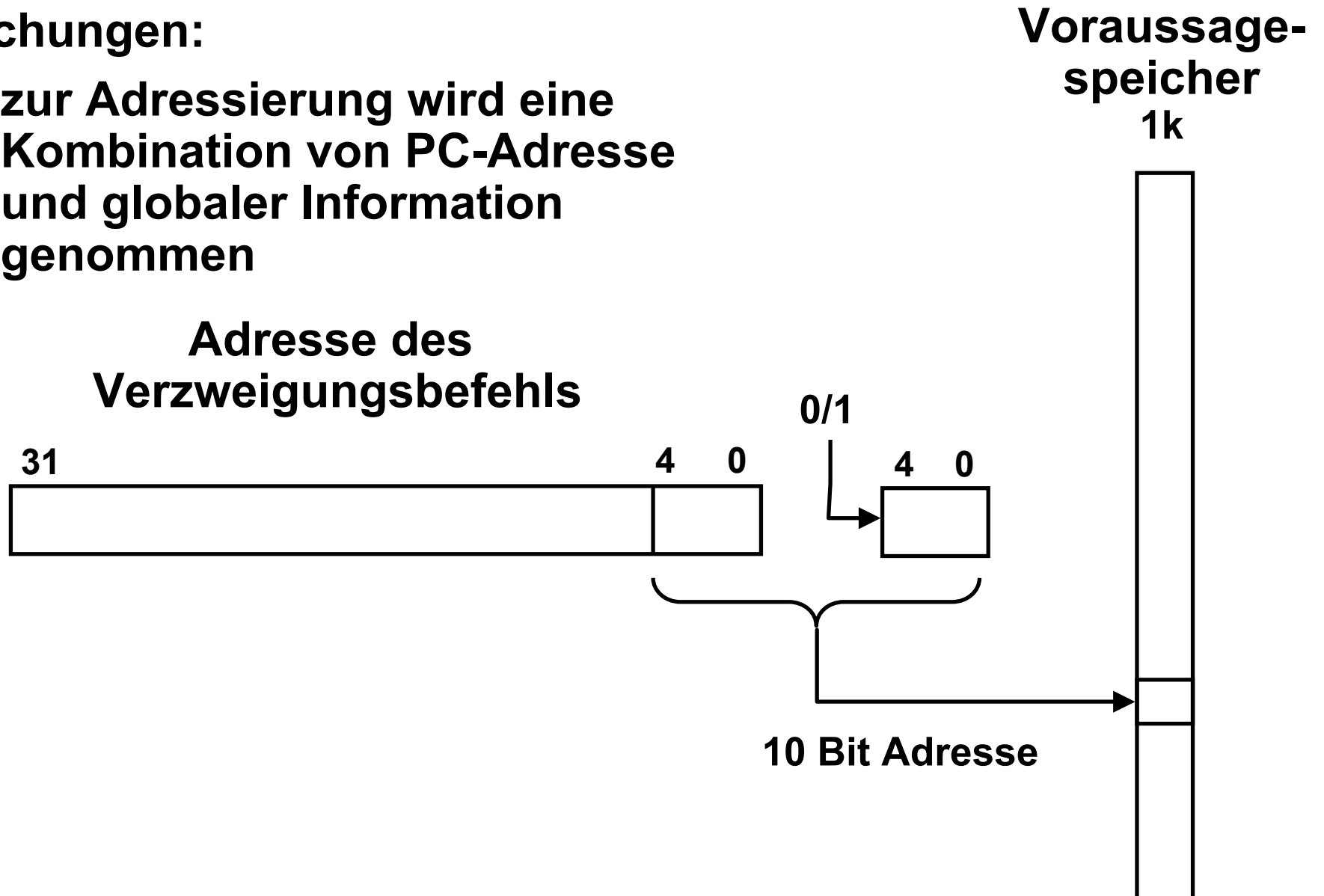
- **Globale** Voraussage der Verzweigung:
 - in einem Schieberegister werden **alle** Verzweigungsentscheidungen als Muster gespeichert
 - das Schieberegister adressiert den Voraussagespeicher



👉 "global" weil das Muster aller Verzweigungsentscheidungen als Adresse dient

4.5 Pipelining der DLX

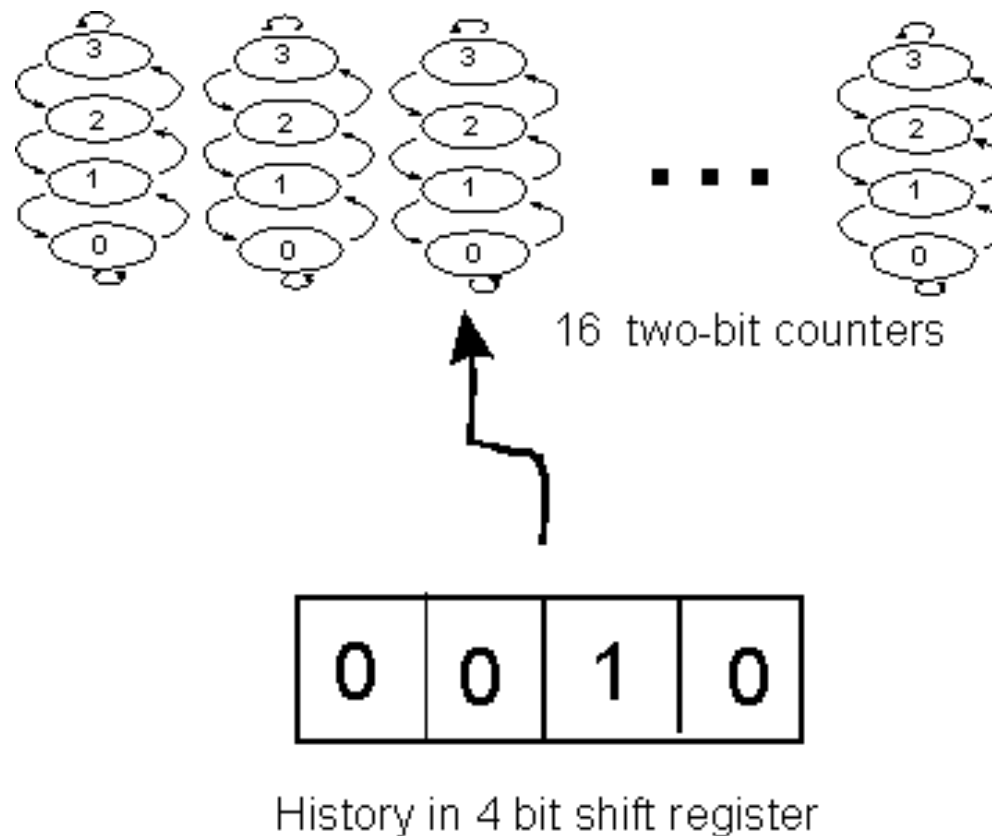
- **Mischungen:**
 - zur Adressierung wird eine Kombination von PC-Adresse und globaler Information genommen



4.5 Pipelining der DLX

— **Beispiel: Two level branch prediction in Pentium MMX, Pentium Pro, and Pentium**

(Quelle: <http://x86.ddj.com/articles/branch/branchprediction.htm>)



4.6 Unterbrechungen

- **Externe Unterbrechungen** (*interrupts*), z.B. durch ein Ein/Ausgabegerät
- **Interne Unterbrechungen** (*traps, exceptions*), z.B. durch Seitenfehler, Arithmetiküberlauf, usw.
- "Unterbrechung": eine durch die Hardware erzwungene Unterbrechung des Ablaufs eines Programms und Übergang zu einem anderen Programm, einer **Unterbrechungsroutine** (ISR, *interrupt service routine*)
- nicht alle Unterbrechungen setzen sich sofort durch: dies ist abhängig von einer der Unterbrechung zugeordneten Priorität
- die Prioritäten der Unterbrechungen sind je nach System von der Hardware vorgegeben oder können programmiert werden

4.6 Unterbrechungen

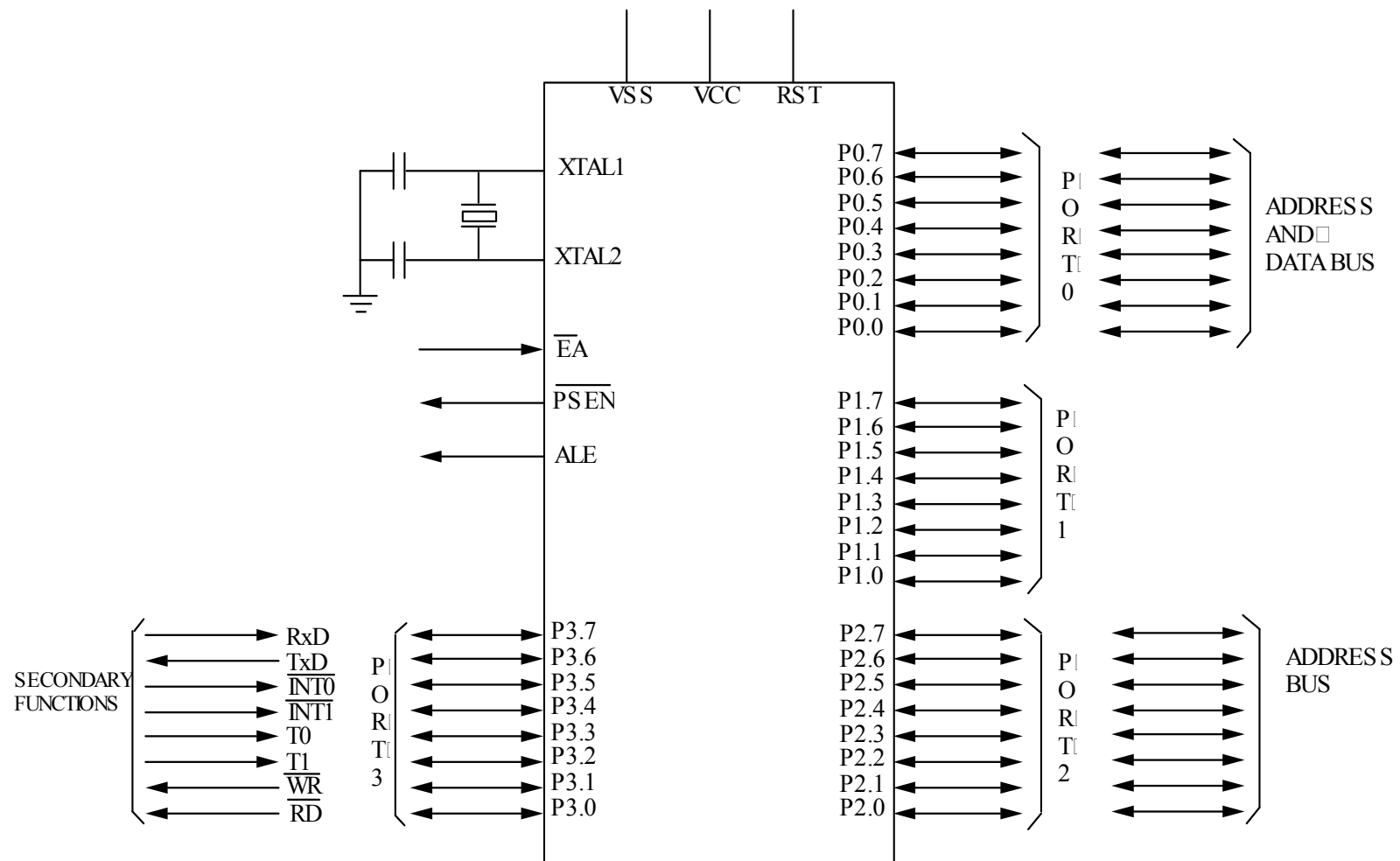
- **Externe Unterbrechungen**
 - werden z.B. ausgelöst durch die Verfügbarkeit von Daten bei einem Ein/Ausgabegerät
 - Geräte signalisieren über die Unterbrechungsleitungen von Bussen einem Unterbrechungswerk (*interrupt controller*) ihren Unterbrechungswunsch
 - externe Unterbrechungen werden nur zwischen zwei Befehlen angenommen
 - wird die Unterbrechung angenommen, wird durch die Hardware ein Sprung an eine bestimmte Adresse (Anfangsadresse der ISR) erzwungen
 - zusätzlich wird der aktuelle Befehlszählerstand z.B. auf dem Stack gerettet
 - **Unterbrechungsvektor**: jeder Unterbrechungsleitung ist eine feste Anfangsadresse zugeordnet

4.6 Unterbrechungen

- **Aufgaben der Unterbrechungsroutine (ISR):**
 - den **Status** des laufenden Programms insbesondere Registerinhalte z.B. auf dem Stack **retten**
 - möglicherweise Abfrage, von welchem Gerät ein Unterbrechungswunsch besteht (falls die ISR-Anfangsadresse nicht durch die Hardware festgelegt wird)
 - Behandlung des Unterbrechungswunsches
 - wiederherstellen des alten Programmstatus und Rückkehr zum unterbrochenen Programm

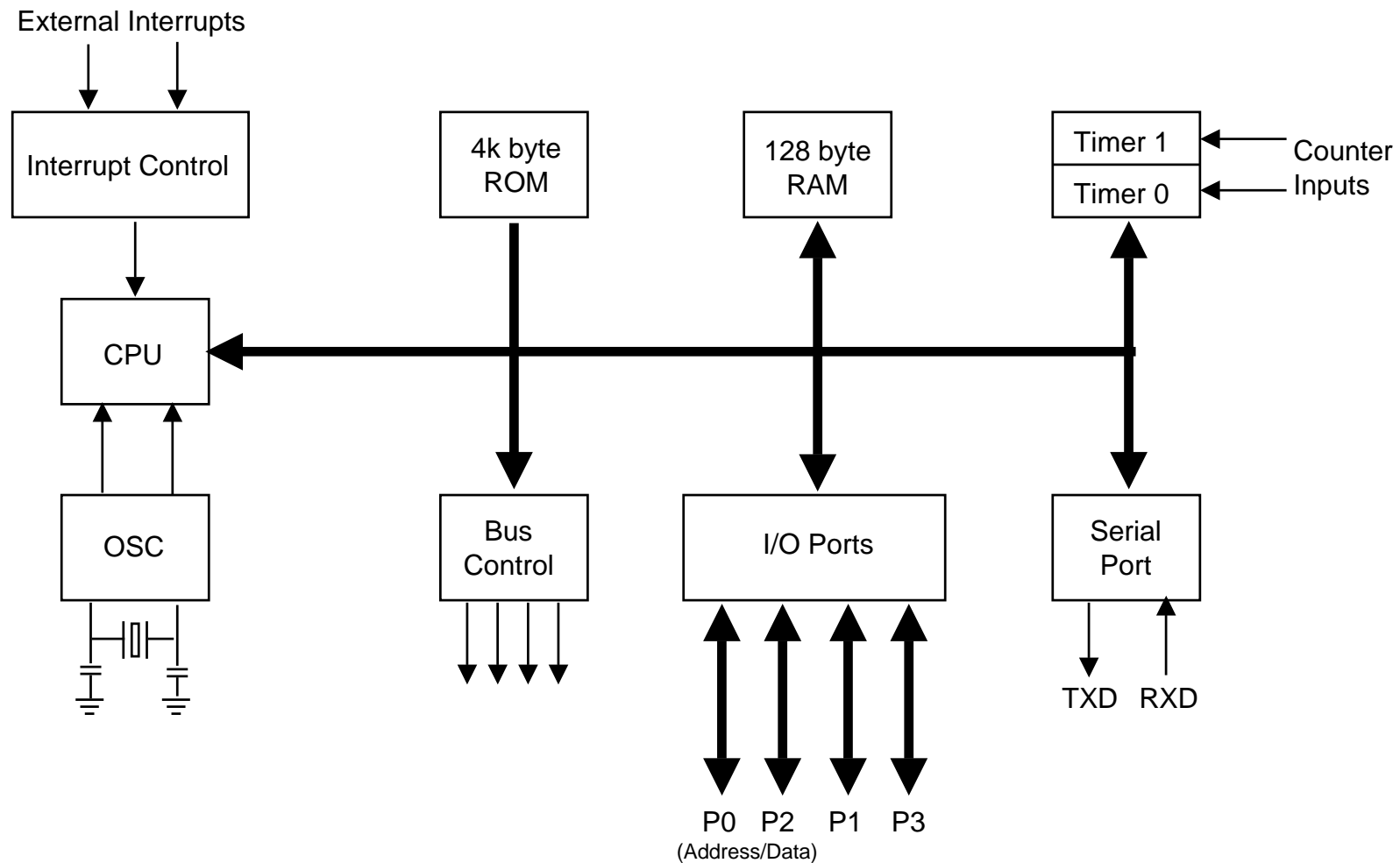
4.6 Unterbrechungen

● Beispiel: Mikrocontroller 8051



4.6 Unterbrechungen

➤ Blockdiagramm 8051:

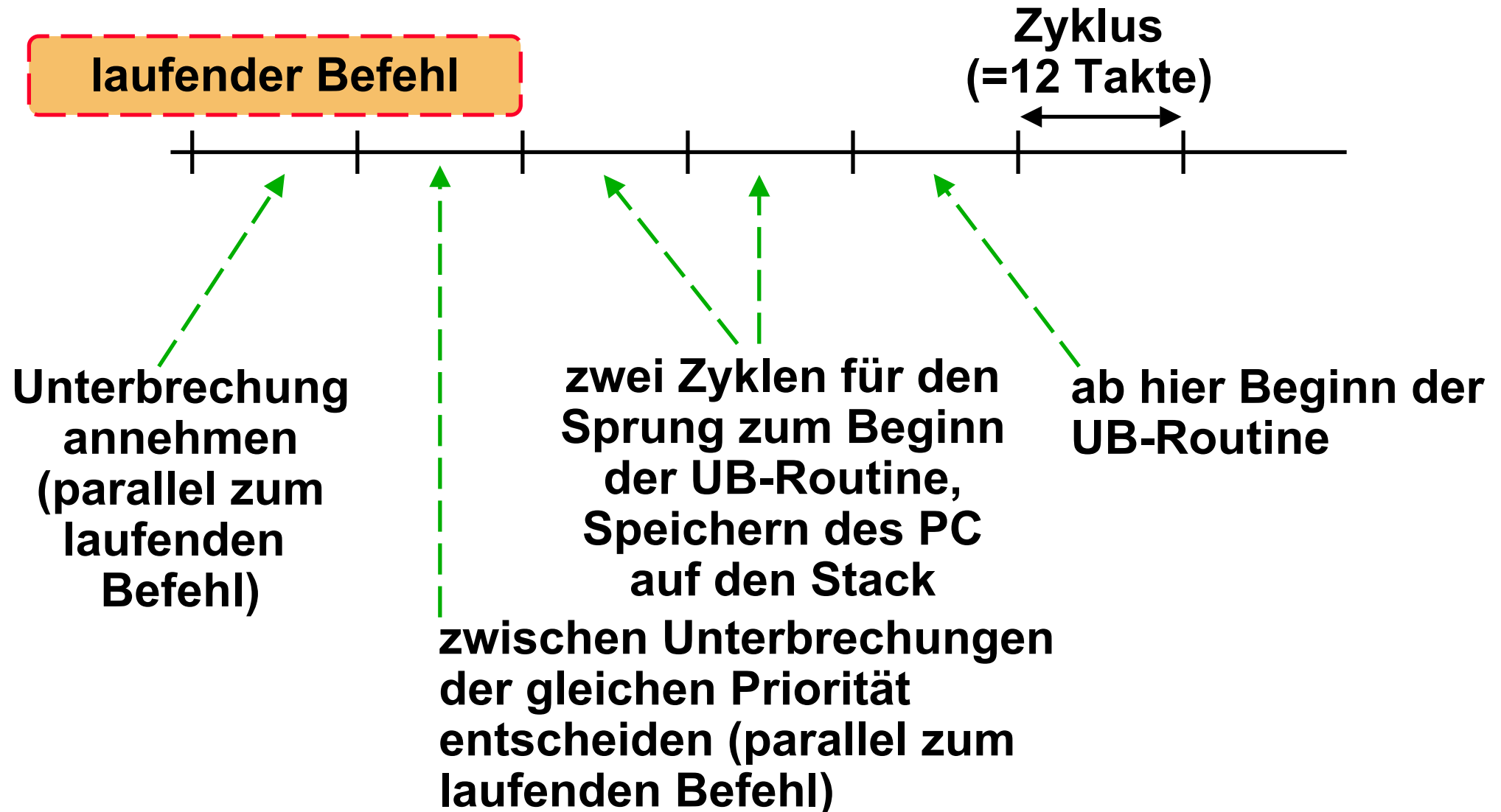


4.6 Unterbrechungen

- Die Annahme einer Unterbrechung erzwingt einen Sprung an eine festgelegte Adresse (**Unterbrechungsvektor**)
 - externe Unterbrechung 0 IE0 **0003H**
 - Zeitgeber 0 TF0 **0013H**
 - externe Unterbrechung 1 IE1 **000BH**
 - Zeitgeber 1 TF1 **001BH**
 - serielle Schnittstelle RI oder TI **0023H**

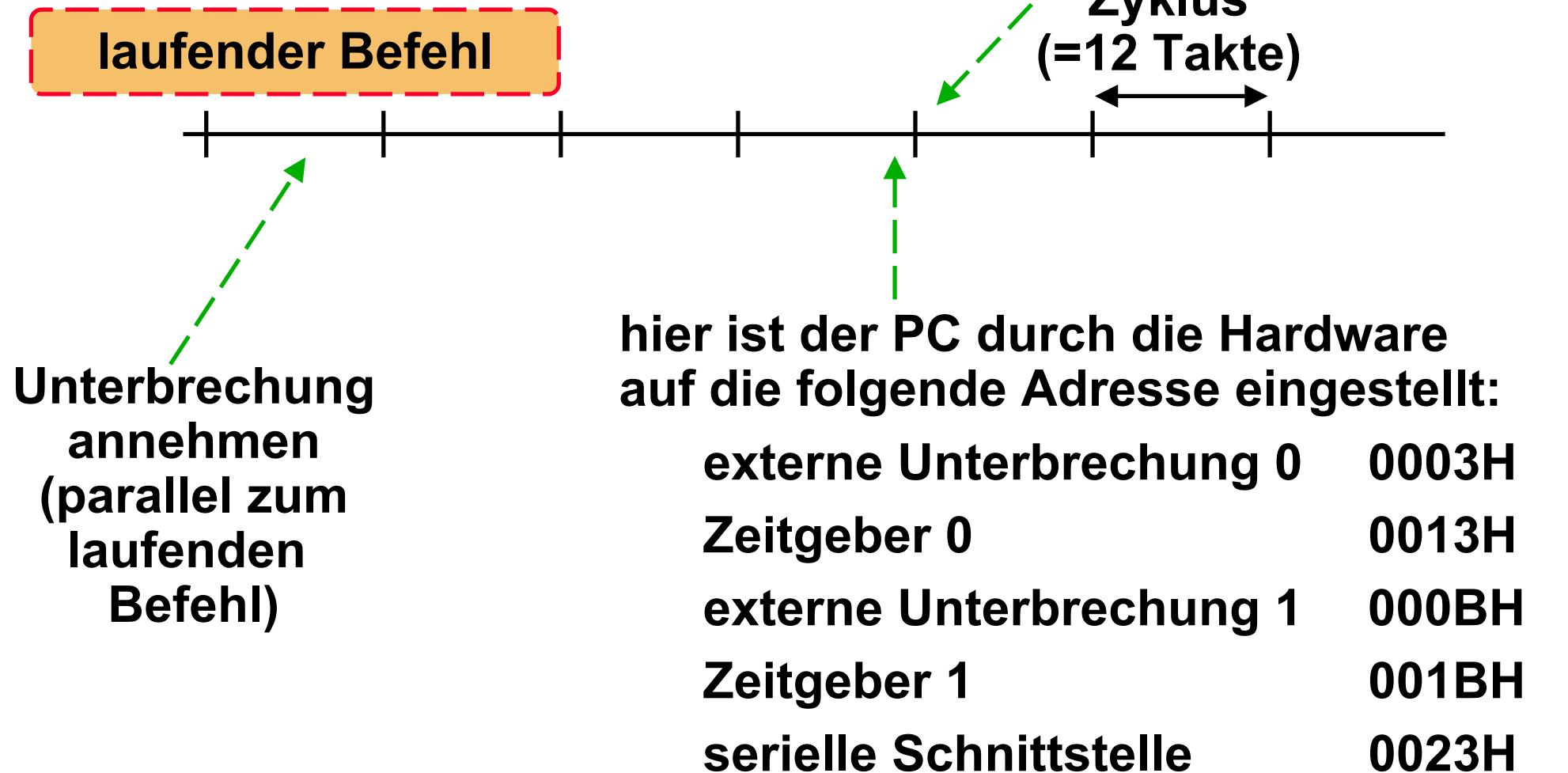
4.6 Unterbrechungen

➤ Zeitlicher Ablauf einer Unterbrechung



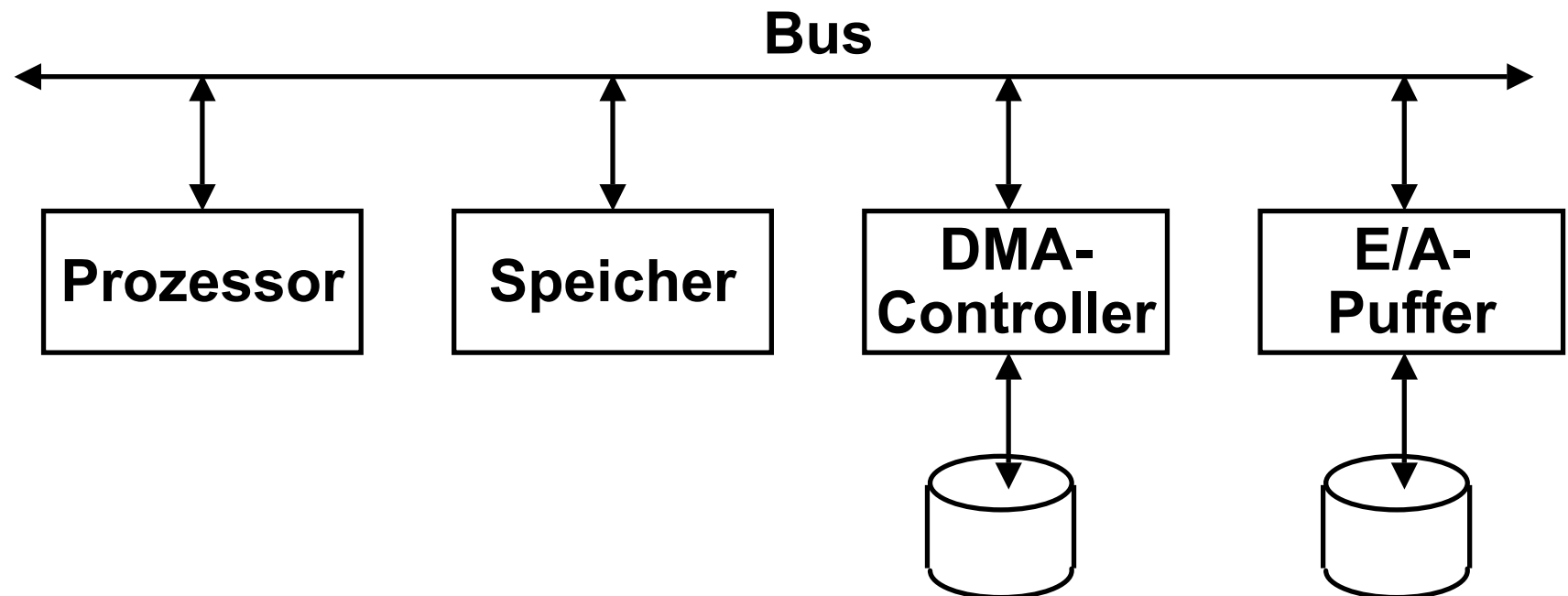
4.6 Unterbrechungen

➤ Zeitlicher Ablauf einer Unterbrechung



4.6 Unterbrechungen

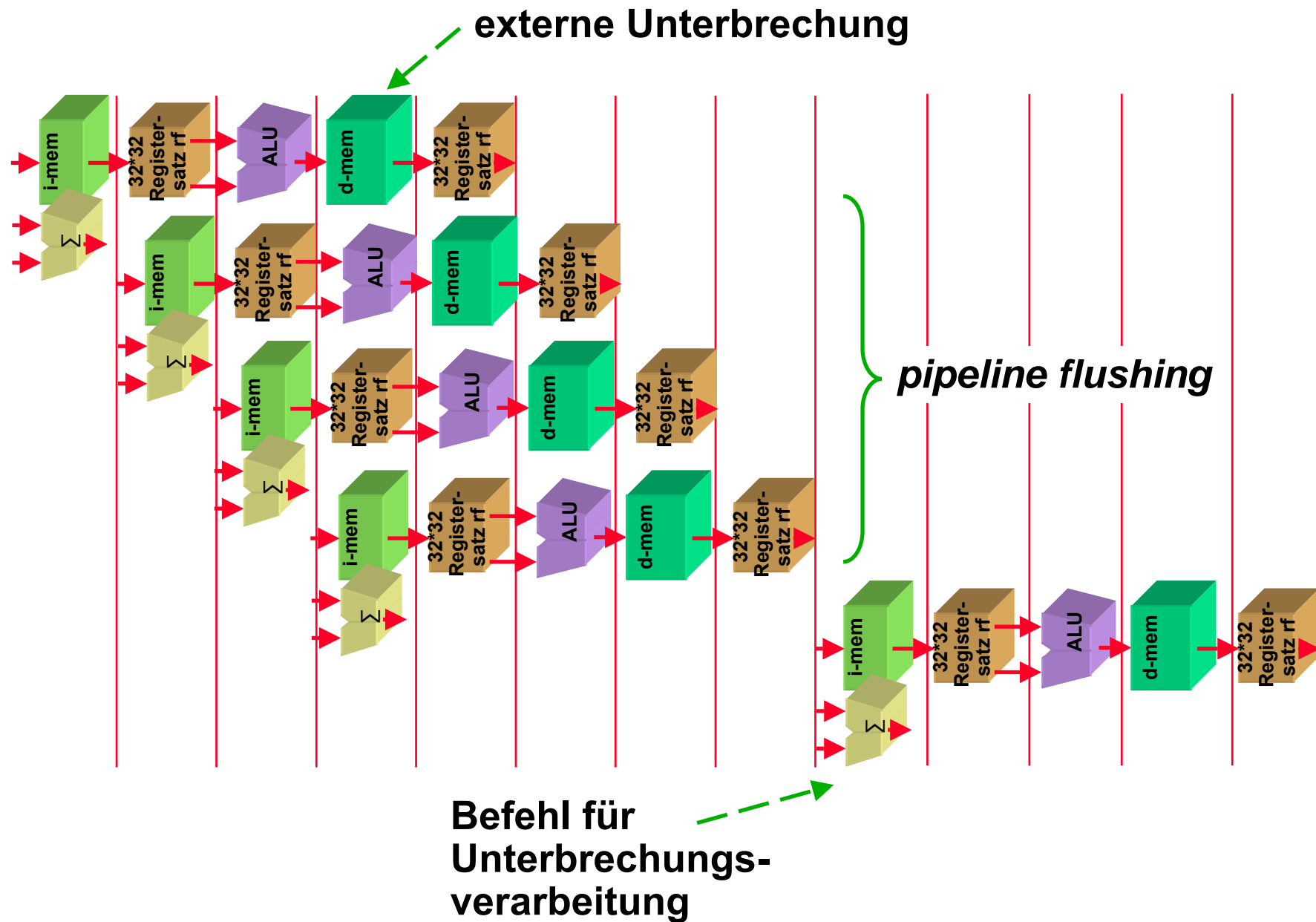
- **Viele andere Verfahren sind möglich, z.B.:**
 - ein Unterbrechungswunsch wird wie eine spezielle Busanforderung behandelt
 - bei Annahme legt das unterbrechende Gerät die ISR Anfangsadresse auf den Bus
 - die Anfangsadresse wird vom Prozessor in den PC übernommen



4.6 Unterbrechungen

- **Komplikationen:**
 - die Unterbrechungsbehandlung ist (relativ) einfach bei der seriellen Ausführung von Befehle
 - aber schwierig bei Pipelining
 - bei **externen** Unterbrechungen müssen alle noch in der Pipeline vorhandenen, also bereits gestarteten Befehle beendet werden, bevor die Unterbrechungs-routine gestartet wird (*pipeline flushing*)

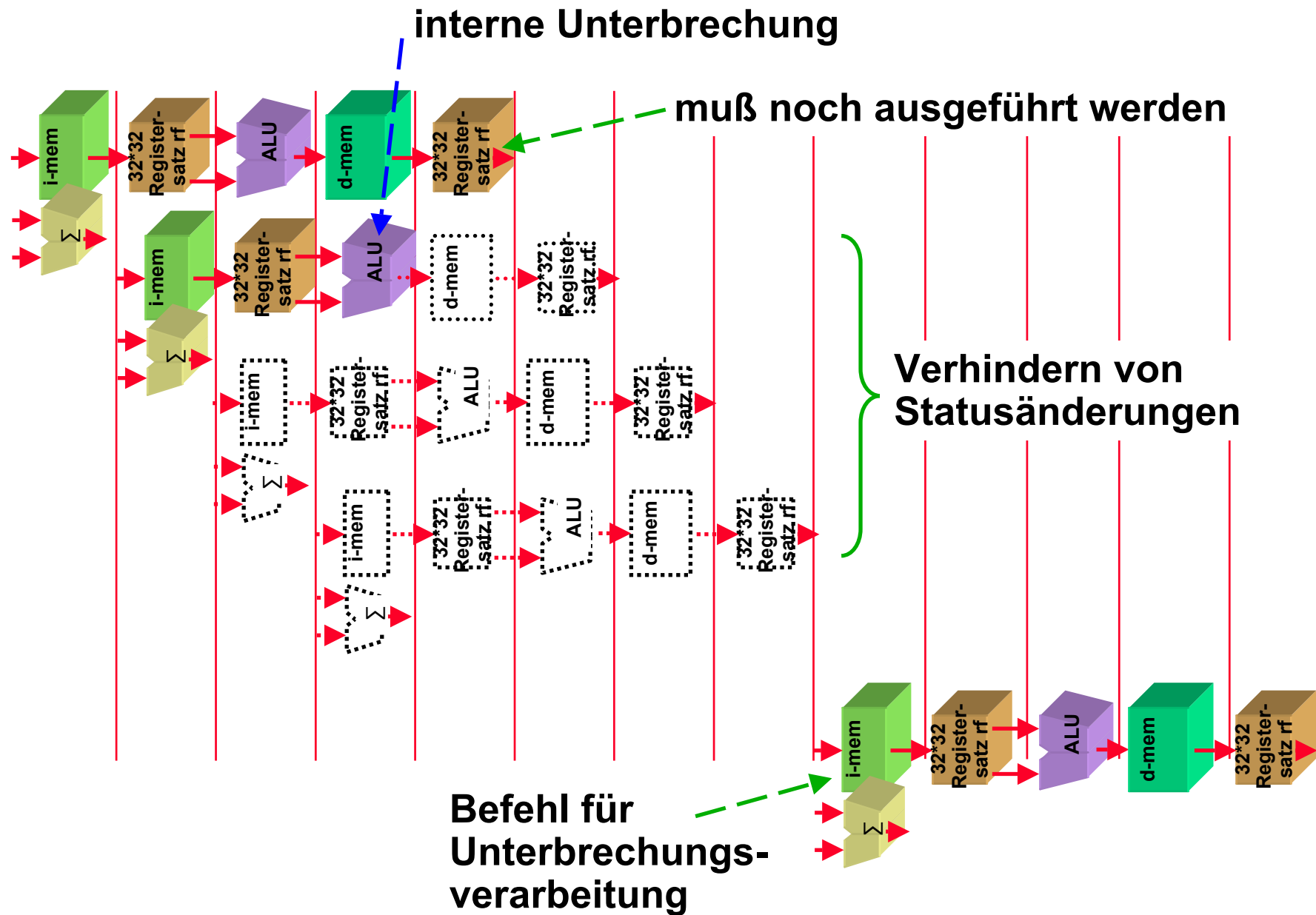
4.6 Unterbrechungen



4.6 Unterbrechungen

- Bei **internen** Unterbrechungen ist während der Befehlsverarbeitung ein Abbruch des laufenden Befehls notwendig (z.B. bei einem Arithmetiküberlauf oder Seitenfehler)
 - "Herunterfahren" aller noch in der Pipeline vorhandenen Instruktionen und Blockieren aller Statusänderungen
- Wiederstartbarkeit von Instruktionen:
 - einfach bei RISC
 - extrem schwierig bei CISC (z.B. bei Autoinkrement-Befehlen)

4.6 Unterbrechungen



4.6 Unterbrechungen

- Begriff der "**präzisen Unterbrechung**": alle Vorgängerbefehle können beendet werden und der die Unterbrechung auslösende Befehl wird nach der ISR wiedergestartet

4.6 Unterbrechungen

- Problem bei verzögerten Verzweigungen ("*delayed branches*")
 - der auf einen Verzweigungsbefehl folgende Befehl wird noch ausgeführt
 - 2 PC's notwendig, oder aber Verschieben der Unterbrechungsannahme (z.B. SHARC DSP)

