technische universität
dortmund

fakultät für informatik
informatik 12

2.3 Register-Transfer-Strukturen

Peter Marwedel

Informatik 12

Otto-Hahn-Str. 16

Tel. 755 6111

E-mail: peter.marwedel@tu-dortmund.de

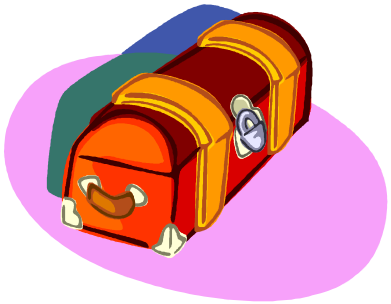
Sprechstunde: Mo 13:00-14:00

2010/12/20
(xmas)

Externe Architektur → interne Architektur

Bislang:

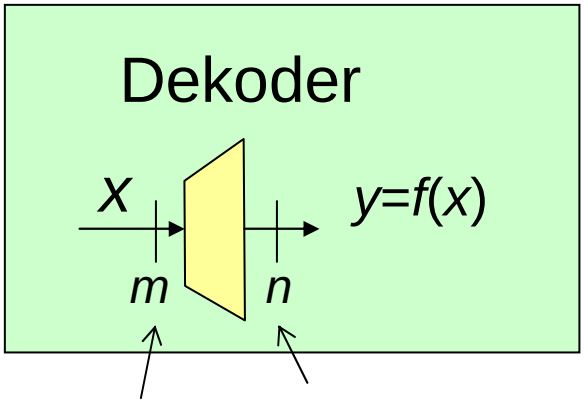
Sicht des Programmierers auf den Prozessor:
Befehlsschnittstelle,
externe Architektur



Jetzt:

interne Realisierung:
interne Architektur,
Mikroarchitektur,
Realisierung mit Register-Transfer-Strukturen
(der Inhalt welchen Registers wird in welches andere Register transferiert?)

Komponenten von RT-Strukturen: Dekoder



Beispiel 1: 1 aus *n*-Dekoder

$$f(x) = \text{"00...}\overset{y_i}{1}\text{...00" wenn } nat(x) = i$$

z.B.

$$f(x) = \begin{cases} \text{"0...0001"} & \text{wenn } x = \text{"0...00"} \\ \text{"0...0010"} & \text{wenn } x = \text{"0...01"} \\ \text{"0...0100"} & \text{wenn } x = \text{"0...10"} \\ \text{"0...1000"} & \text{wenn } x = \text{"0...11"} \\ \text{...} & \text{...} \quad \text{...} \quad \text{...} \quad \text{...} \end{cases}$$

Beispiel 2: Prioritätsencoder (*m*>*n*)

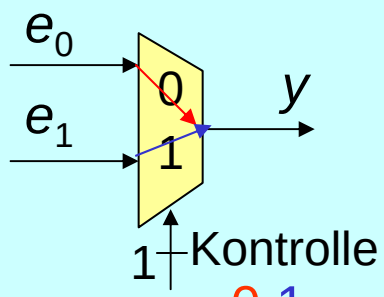
$$nat(f(x)) = \begin{cases} i \text{ wenn der größte Index ist, für den } x_i = \text{'1' ist} \\ \textit{undefiniert} \text{ für } x = \text{"0...00"} \end{cases}$$

z.B.: $y = \text{"101" für } x = \text{"00}\underline{1}\text{01011"}$

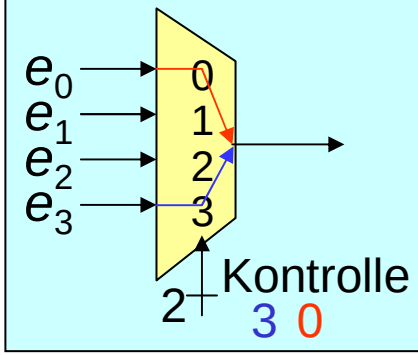
nat(a)=$\Sigma a_i 2^i$: (Abbildung Bitvektor $\rightarrow$natürliche Zahl)

Komponenten von RT-Strukturen: Multiplexer

z.B. 2 auf 1 Multiplexer



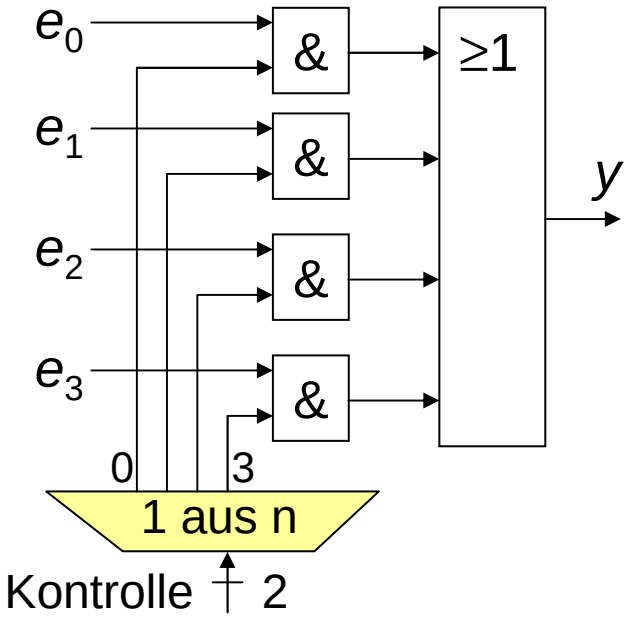
z.B. 4 auf 1 Multiplexer



$y = e_i$ für nat(Kontrolle)=i

e_i, y: einzelne Bits oder Bitvektoren

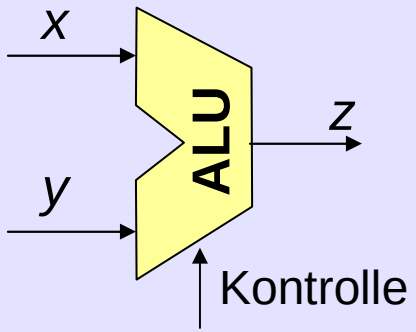
Mögliche Realisierung (für einzelnes Bit)



Komponenten von RT-Strukturen: ALUs

Addierer, arithmetisch/
logische Einheiten (ALUs)

$z=f(x,y,\text{Kontrolle})$



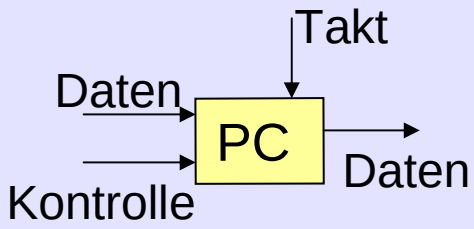
Beispiel

$$z = \begin{cases} x + y & \text{wenn} \quad \text{nat(Kontrolle)} = 0 \\ x - y & \text{wenn} \quad \text{nat(Kontrolle)} = 1 \\ x \wedge y & \text{wenn} \quad \text{nat(Kontrolle)} = 2 \\ x \vee y & \text{wenn} \quad \text{nat(Kontrolle)} = 3 \end{cases}$$

*x.y.z,*Kontrolle*:*
Bitvektoren

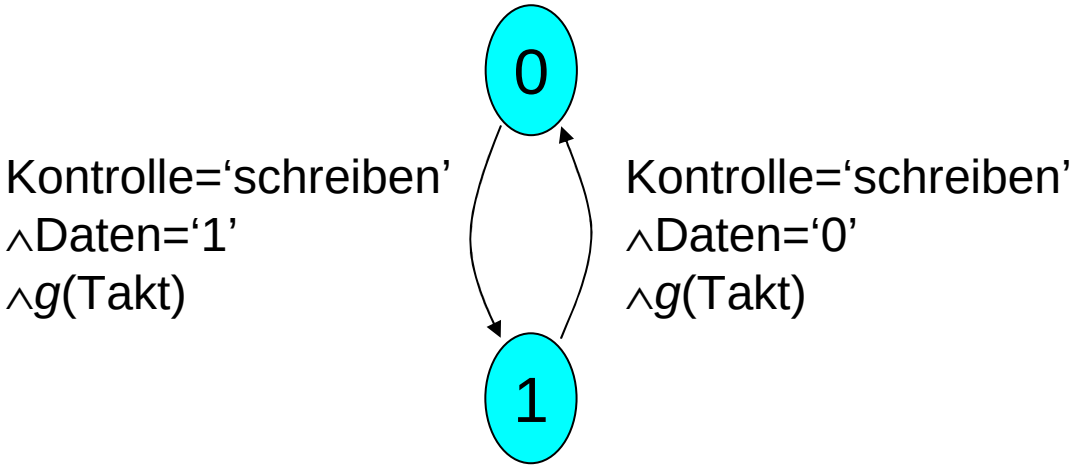
Komponenten von RT-Strukturen: Register

Register



Übernimmt mit dem Takt die Daten, sofern der Kontrolleingang auf "schreiben" gesetzt ist.
Daten*:* einzelne Bits oder Bitvektoren

Für jedes einzelne Bit:

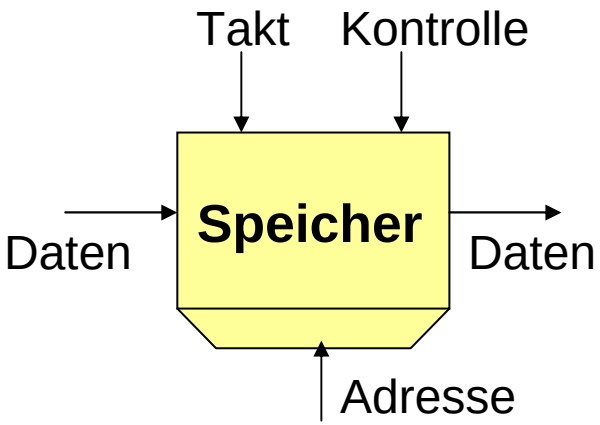


g: modelliert unterschiedliche Taktabhängigkeit

- Flankenabhängigkeit: Wechsel nur bei pos. (“positiv flankengetriggert”) oder neg. Flanke)
- oder Pegelabhängigkeit: Wechsel sofern Takt=‘1’ oder Takt =‘0’ ist

Komponenten von RT-Strukturen: Speicher

Speicher

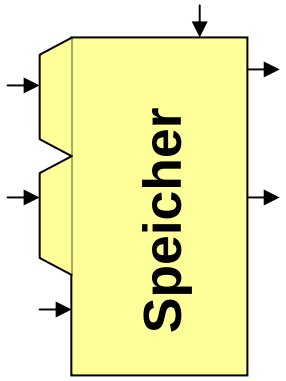


Liest ständig die am Adresseingang ausgewählte Speicherzelle aus und stellt ihren Wert am Datenausgang mit gewisser Verzögerung zur Verfügung.
Übernimmt mit dem Takt die Daten in die ausgewählte Speicherzelle,
sofern der Kontrolleingang auf „Schreiben" gesetzt ist.

Wir stellen den 1-aus-n-Adressdekoder immer explizit dar, um den Adresseingang zu identifizieren.

Multiportspeicher

Multiport-Speicher



Besitzt mehrere Adresseingänge, die zu einem zugeordneten **Port** gehören.
Jedes Leseport stellt ständig die gelesene Speicherzelle am Ausgang bereit;
jedes Schreibport übernimmt die Eingangdaten in die ausgewählte Speicherzelle, sofern der Kontrolleingang auf "schreiben" gestellt ist.
Konflikt, falls mehrere Schreibports dieselbe Zelle auswählen (soll vermieden werden; falls es doch vorkommt: z.B. UND-Verknüpfung der Eingangsdaten)

Speichertechnologien

Einige Unterscheidungen:

Flüchtiger (*volatile*) und **nicht-flüchtiger** (*non-volatile*) **Speicher**:
Nicht-flüchtiger Speicher: Information bleibt nach dem Ausschalten des Stromes erhalten

Statischer und **dynamischer Speicher**:

- Statischer Speicher (SRAM): i.d.Regel 6 Transistoren/Bit.
- Dynamischer Speicher (DRAM): 1 Transistor, 1 Kondensator/Bit, langsamer als SRAM,
  erfordert periodisches Auffrischen der Information,
  ist billiger als SRAM

* www.mb-tech.at/images/pc333.jpg

Siehe auch Teil 1 der Vorlesung sowie (weiterführend):
U. Schwiegelshohn, Technische Informatik, WS 08/09, http://www.irf.tu-dortmund.de/IT/public/TechnischeInformatik/Skript&Folien/ti-skript-0809.pdf, Abschnitt 4.1

Zusammenfassung

Jetzt Sicht auf die interne Architektur:

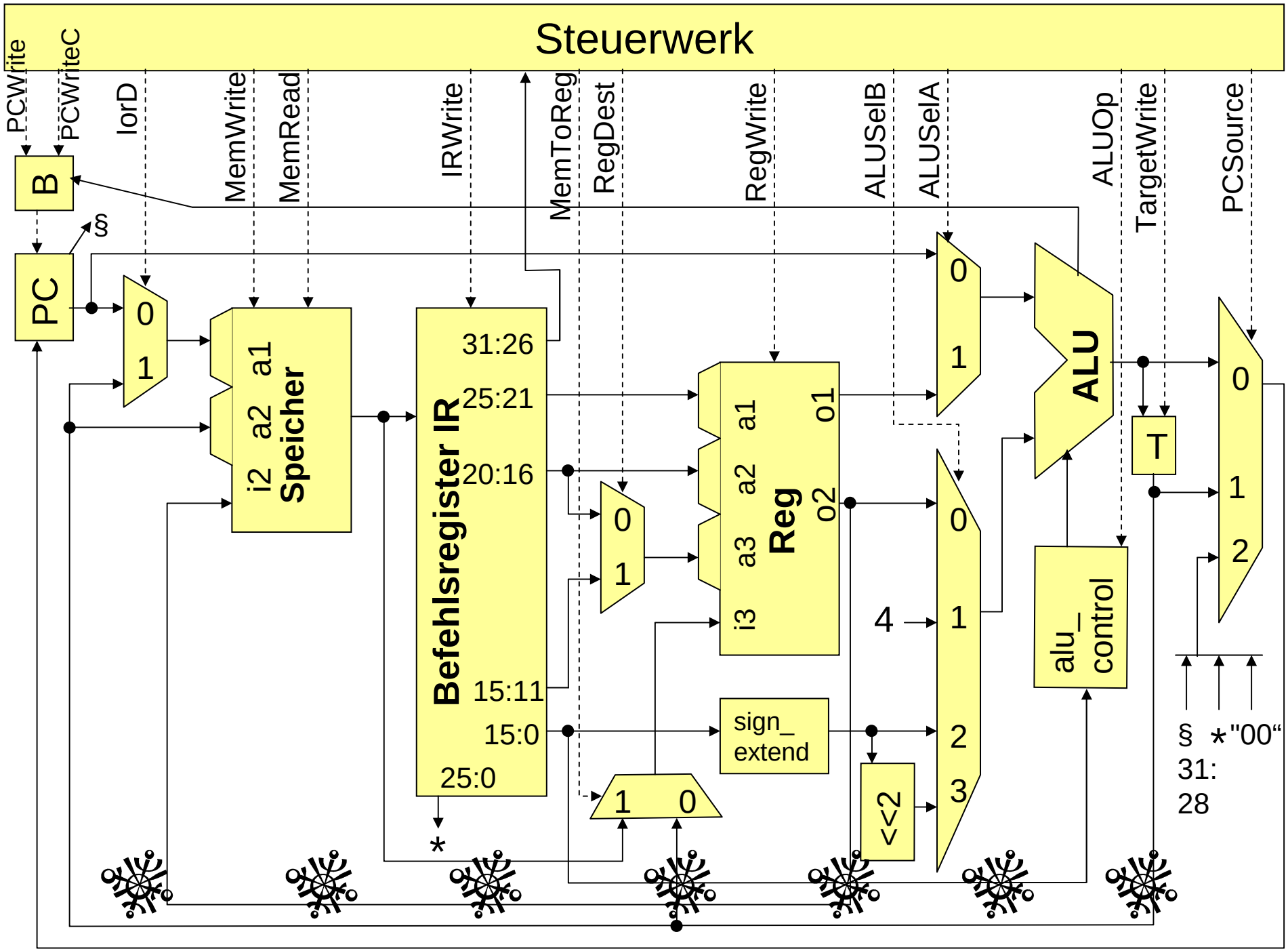
- Komponenten von RT-Strukturen
  - Speicher
  - Register
  - Multiplexer
  - ALUs
  - Dekoder
  - …

Eine Mikroarchitektur für die MIPS-Maschine

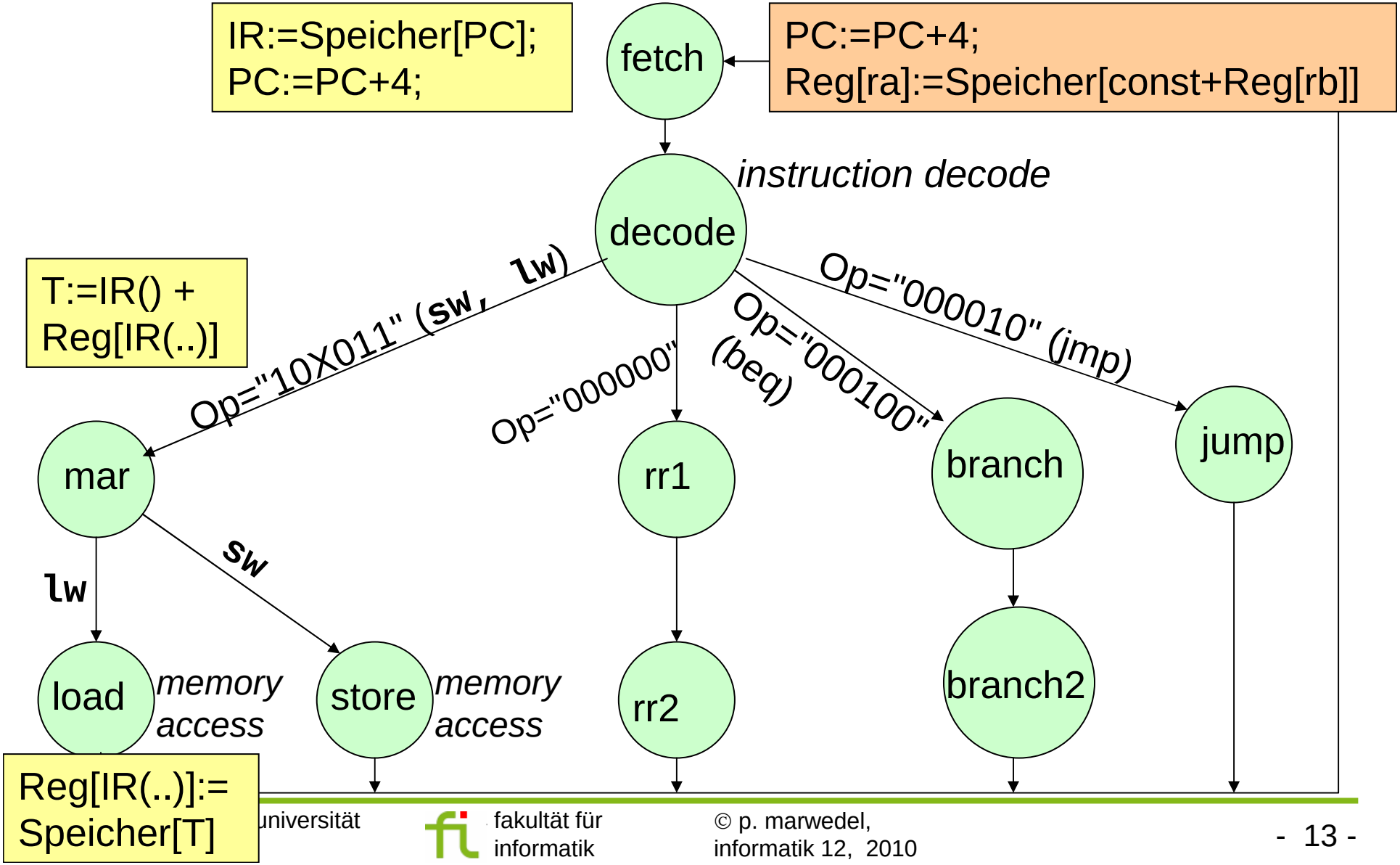
- Automatische Erzeugung von Mikroarchitekturen aus ISA heraus (µArchitektursynthese): wurde untersucht, ist aber
  a) im Rahmen einer Erstsemestervorlesung zu kompliziert
  b) für unsere einfachen Befehle auch nicht nötig.
- Mit Erfahrung (Betrachtung wesentlicher Komponenten/ Verbindungen):
  ☞ manueller µArchitekturentwurf.
  Benötigt Minimum an Komponenten.
  Enthält ein Rechenwerk und ein Steuerwerk.

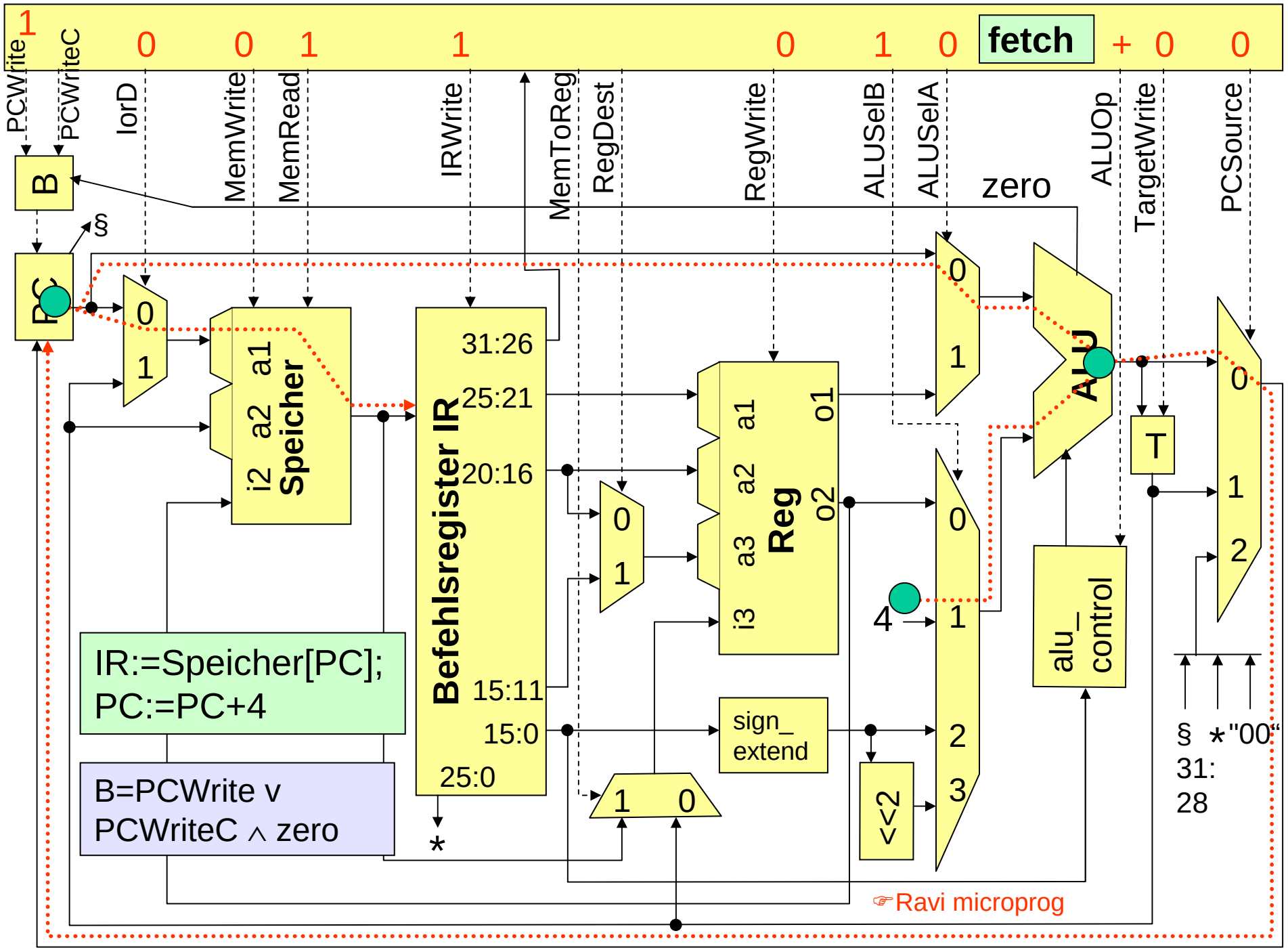
- Verifikation: ∀Befehle: nachweisen, dass die µArchitektur den Befehl ausführen kann.

☞ Vorgehen für die MIPS-Maschine

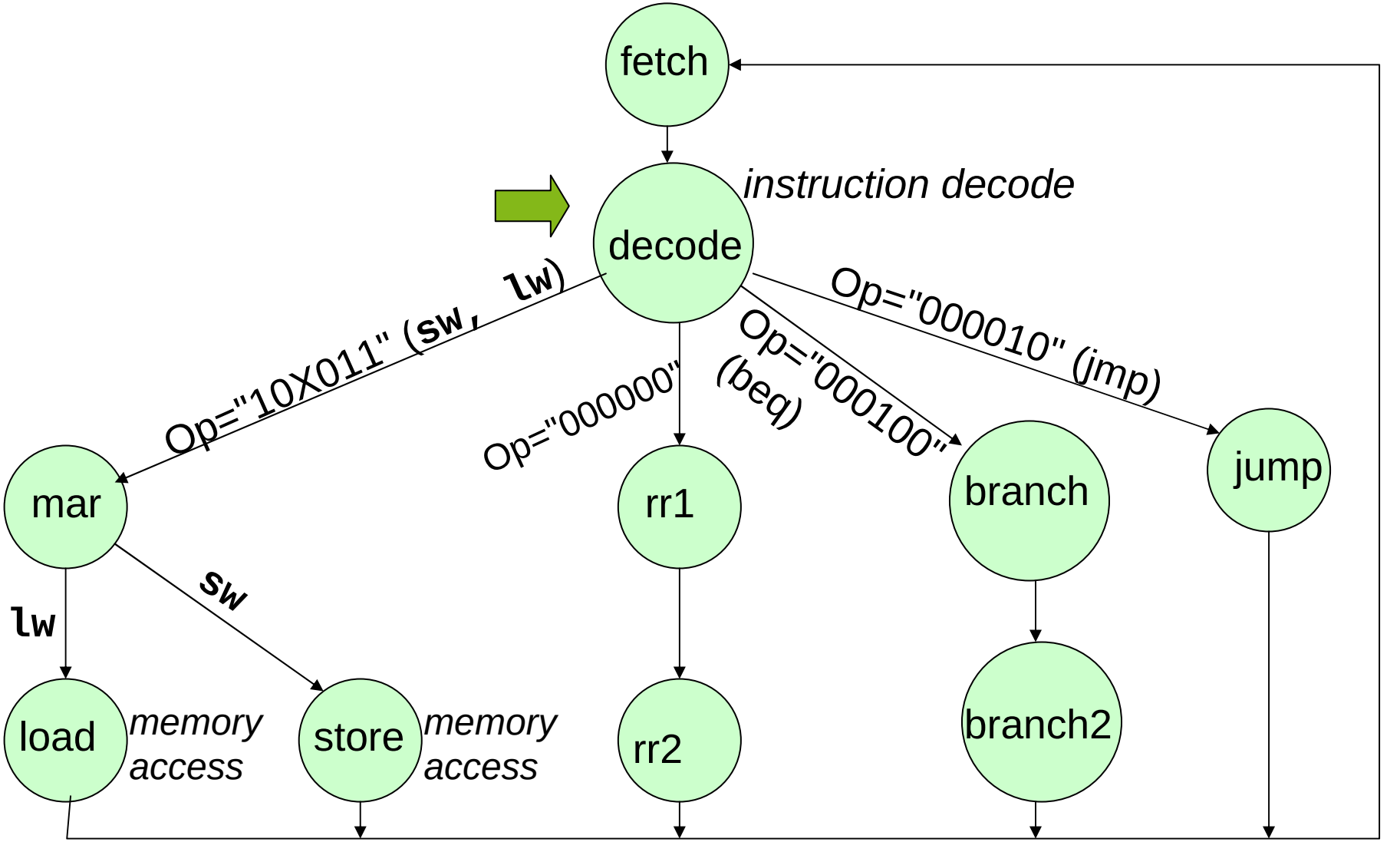

Steuerwerk
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
Befehlsregister IR
31:26
25:21
20:16
15:11
15:0
25:0
*
0
1
i3 a3 a1 a2
Reg
o1 o2
0
1
4
2
3
sign_
extend
<<2
1 0
ALU
alu_
control
T
0
1
2
§
31:
28
*"00“

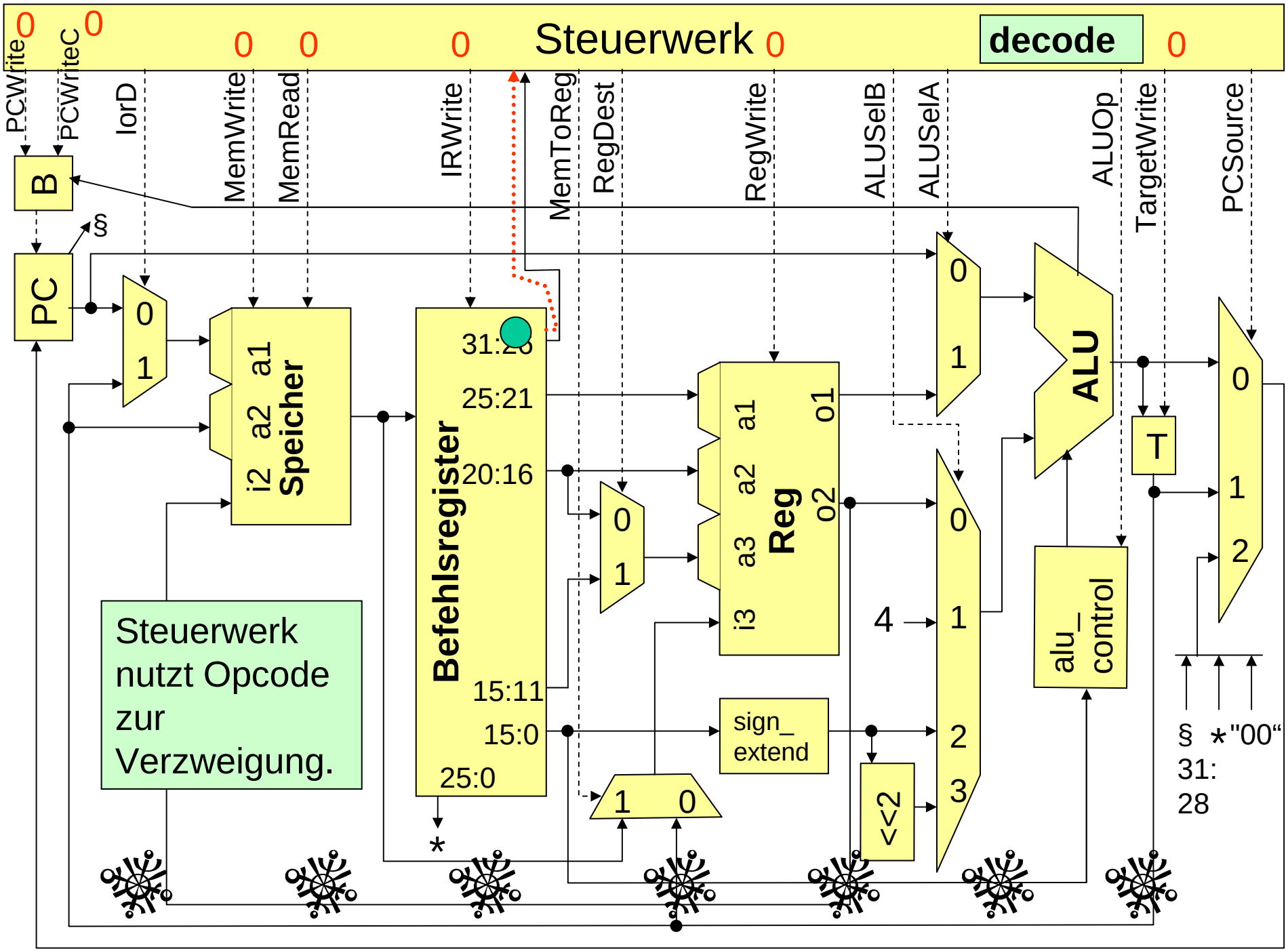
Überprüfung der Ausführbarkeit ☞ Zustandsgraph der Ausführung einiger MIPS-Befehle



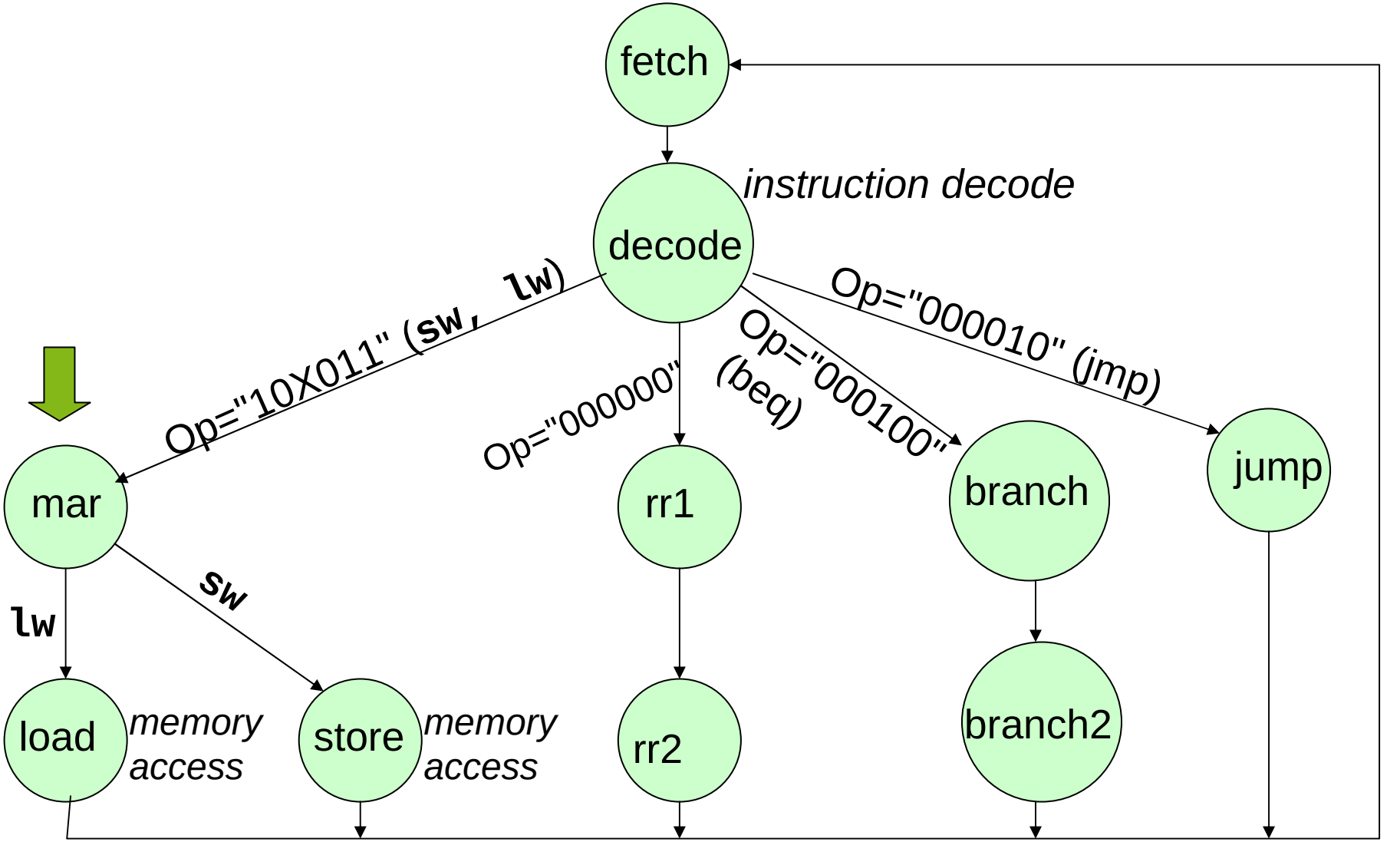

1 0 0 1 1 0 1 0 fetch + 0 0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
zero
ALUOp
TargetWrite
PCSource
B
§
PC
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister IR
*
0
1
i3 a3 a2 a1
Reg
o2 o1
4
sign_
extend
<<2
0
1
2
3
ALU
T
alu_
control
0
1
2
§
31:
28
*"00"
IR:=Speicher[PC];
PC:=PC+4
B=PCWrite v
PCWriteC ∧ zero
☞Ravi microprog

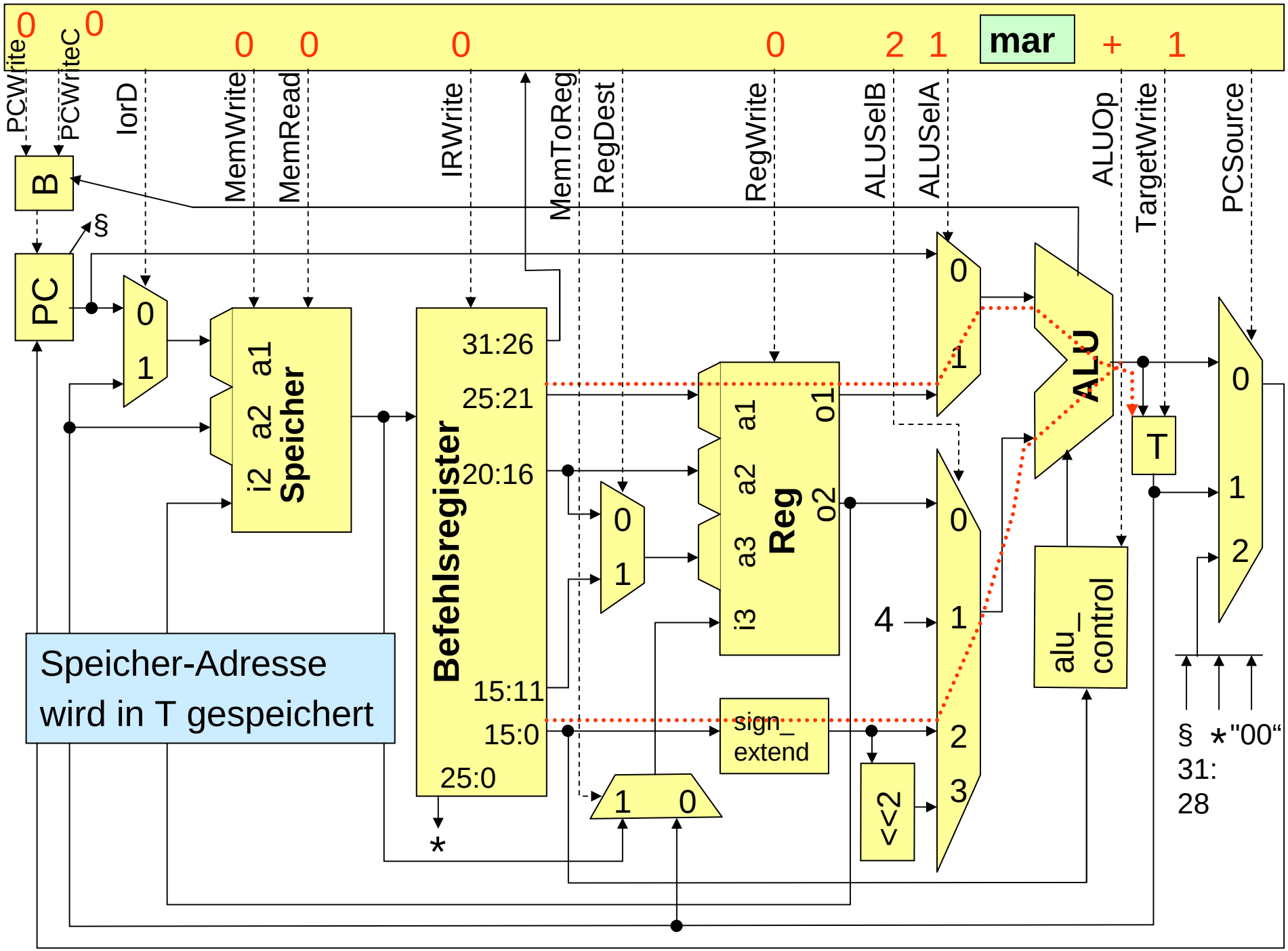
Zustandsgraph
der Ausführung einiger MIPS-Befehle



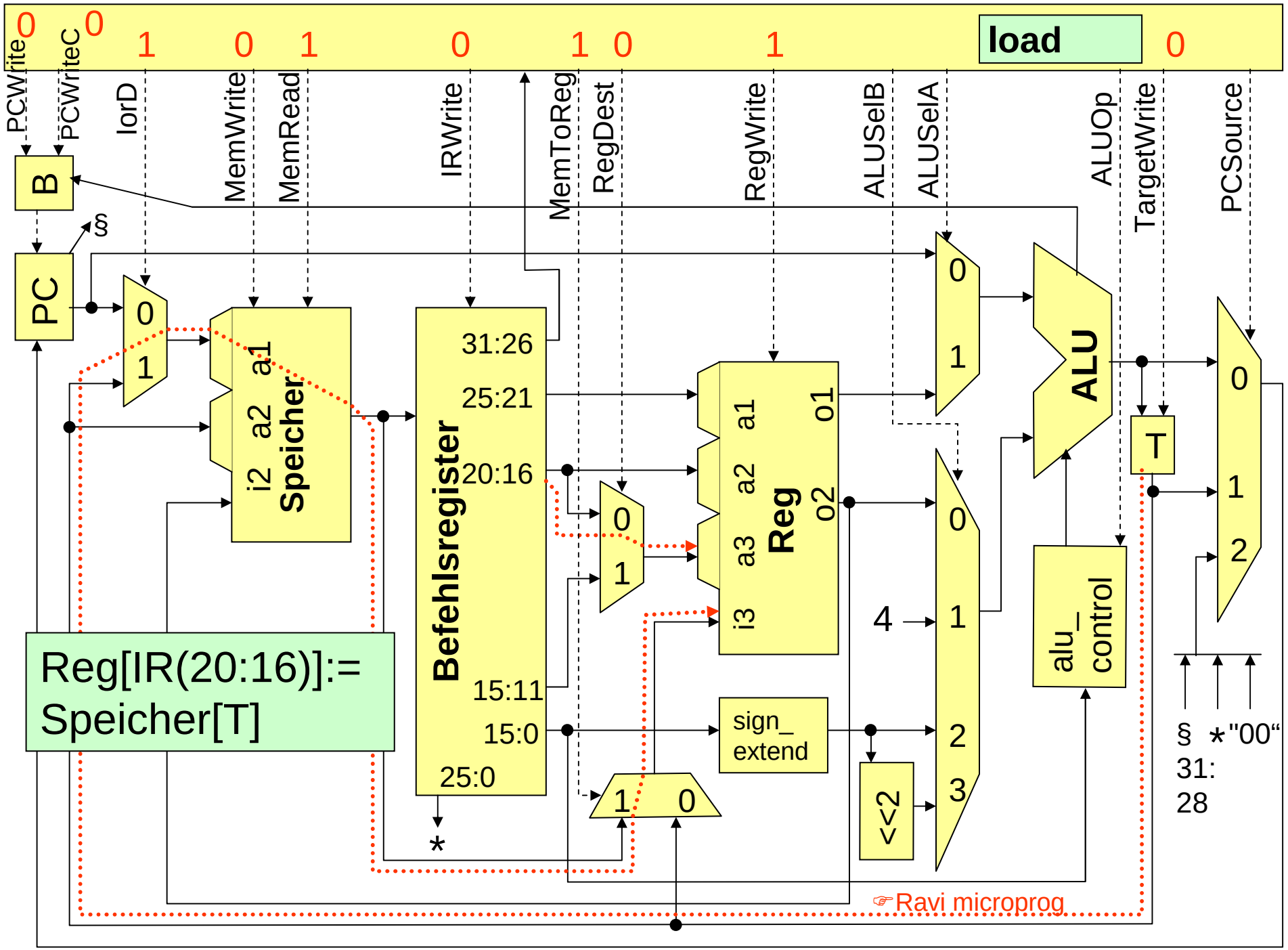

Steuerwerk
decode
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
Speicher
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
Reg
ALU
T
alu_control
sign_extend
<<2
4
§ 31:28
*"00"
Steuerwerk nutzt Opcode zur Verzweigung.

Zustandsgraph
der Ausführung einiger MIPS-Befehle

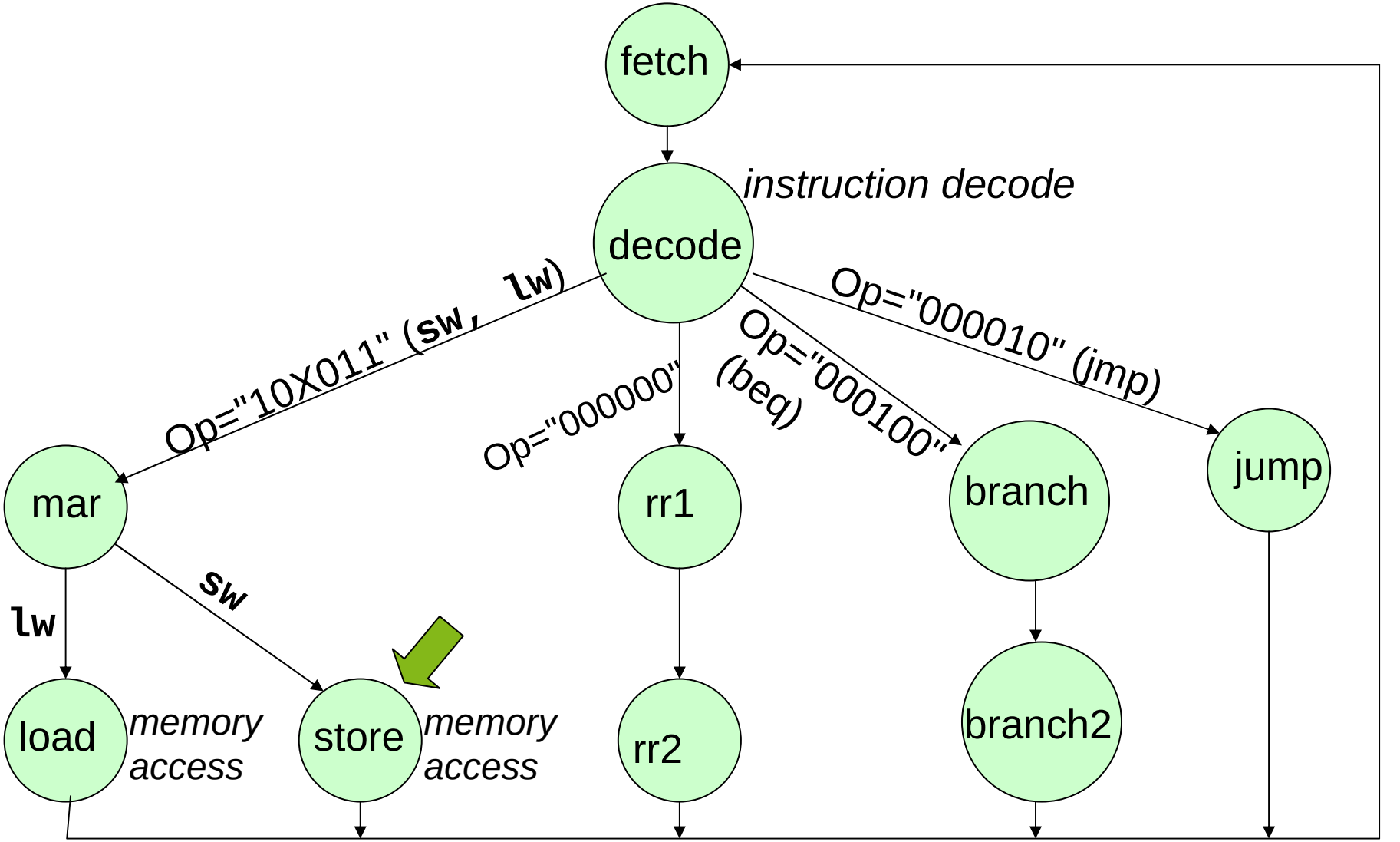


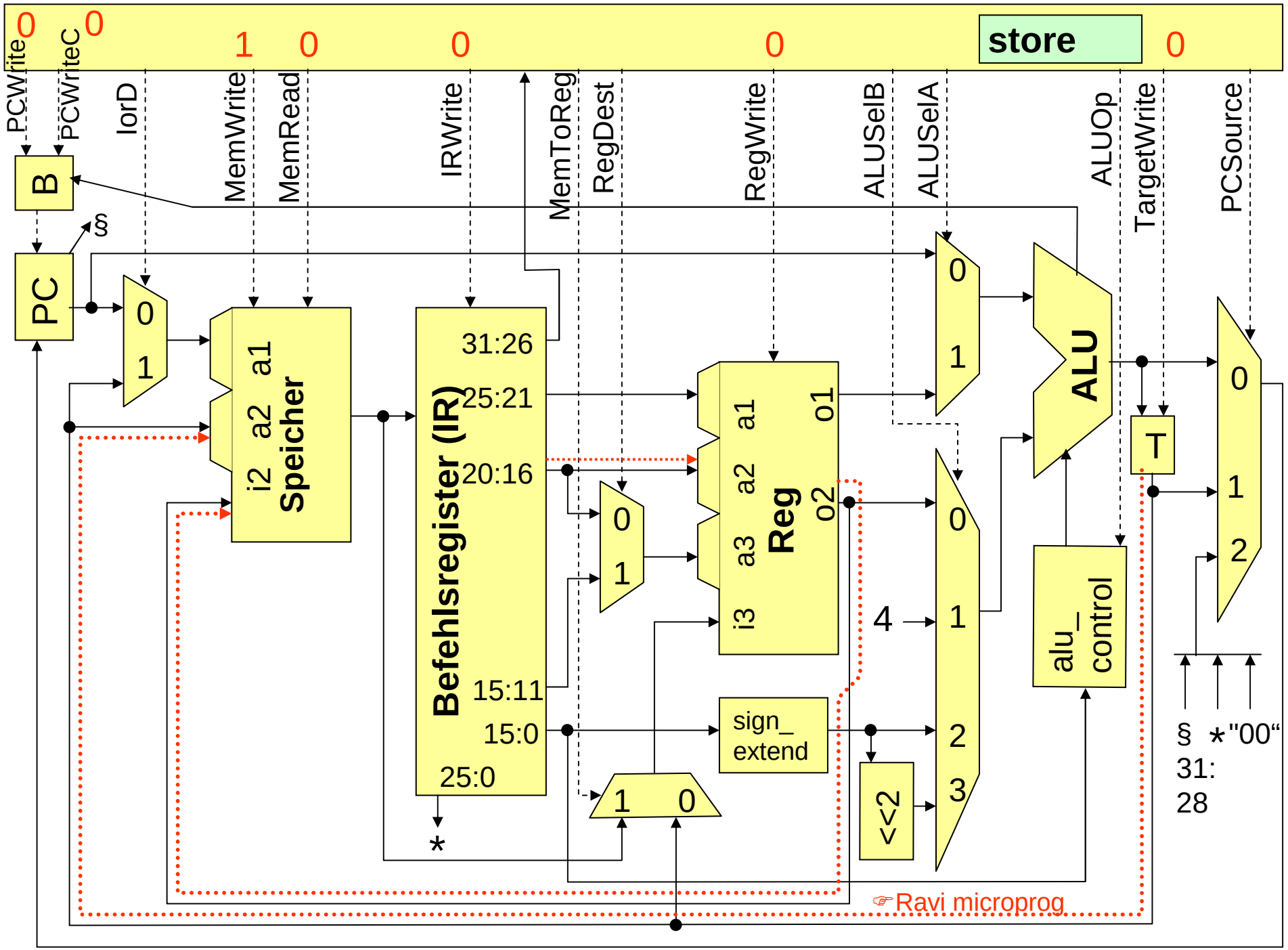

0 0 0 0 0 0 2 1 mar + 1
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
*
i3 a3 a2 a1
Reg
o1 o2
4
sign_ extend
<<2
ALU
alu_ control
T
1 0
§ 31: 28
*"00“
Speicher-Adresse wird in T gespeichert

Zustandsgraph der Ausführung einiger MIPS-Befehle

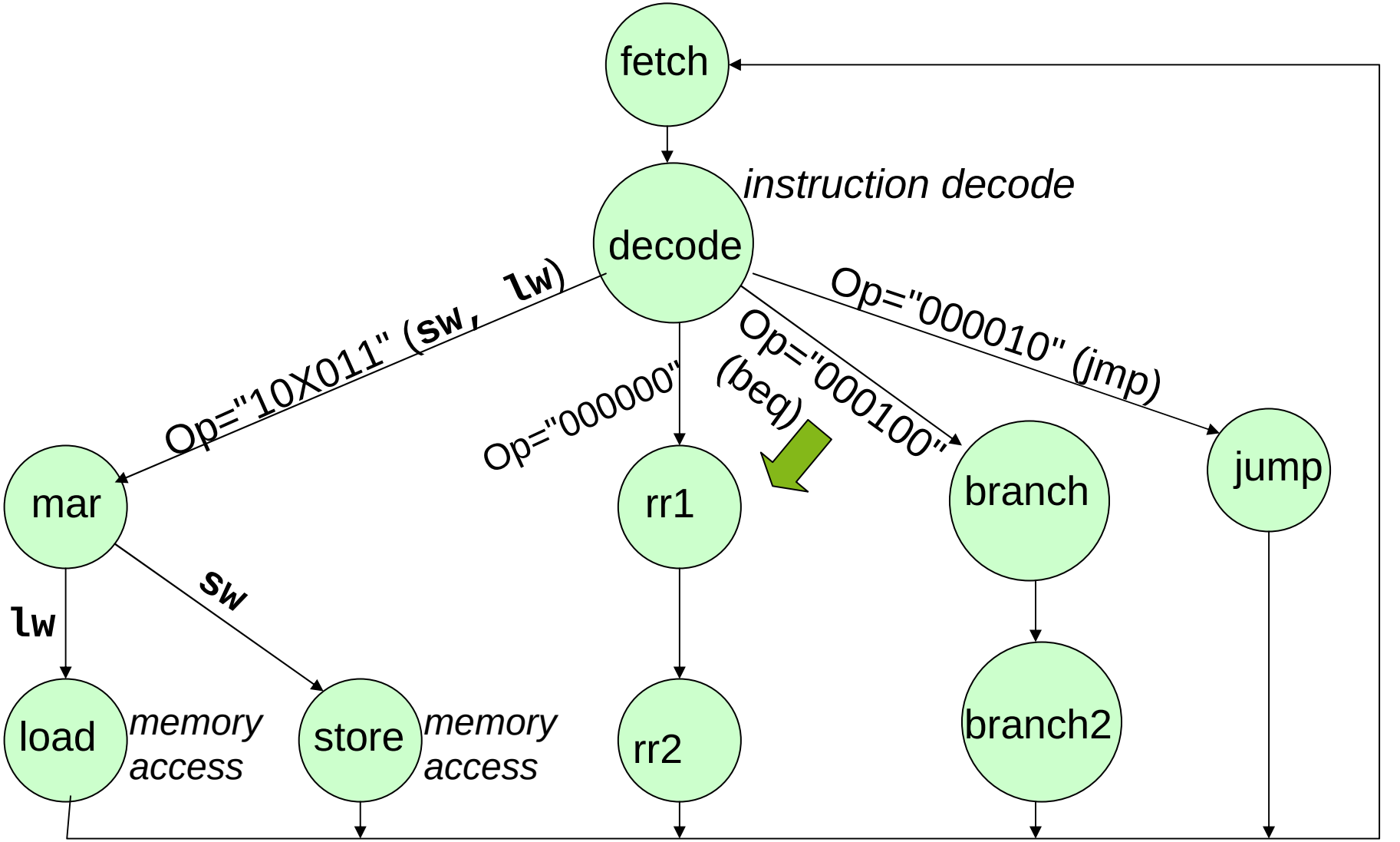

0 0 1 0 1 0 1 0 1
load
0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
§
PC
0
1
i2 a2 a1
Speicher
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
*
0
1
i3 a3 a2 a1
Reg
o2 o1
0
1
4
0
1
2
3
ALU
T
0
1
2
alu_
control
sign_
extend
<<2
1 0
§ 31: 28
*"00“
Reg[IR(20:16)]:=
Speicher[T]
☞Ravi microprog

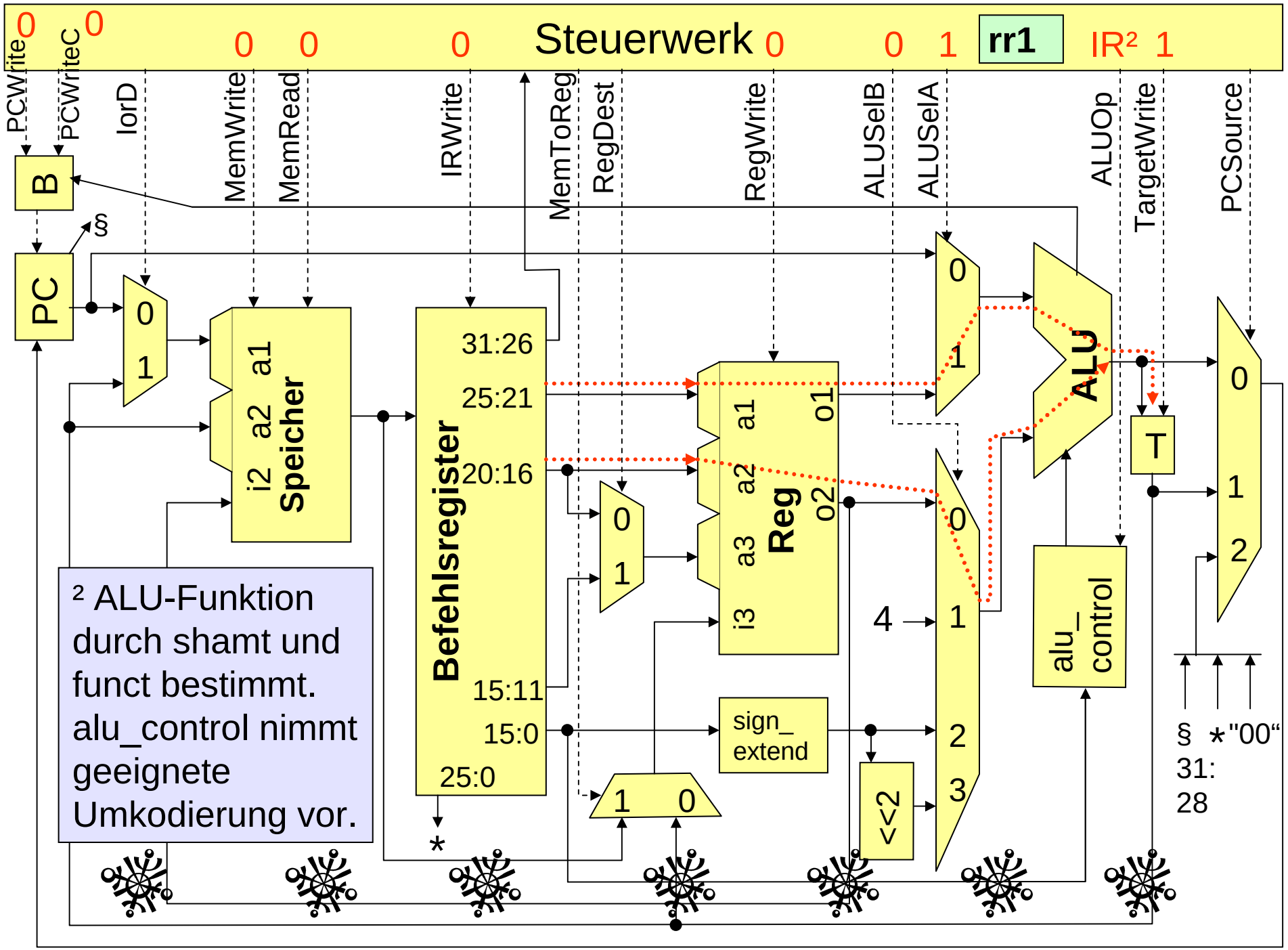
Zustandsgraph der Ausführung einiger MIPS-Befehle



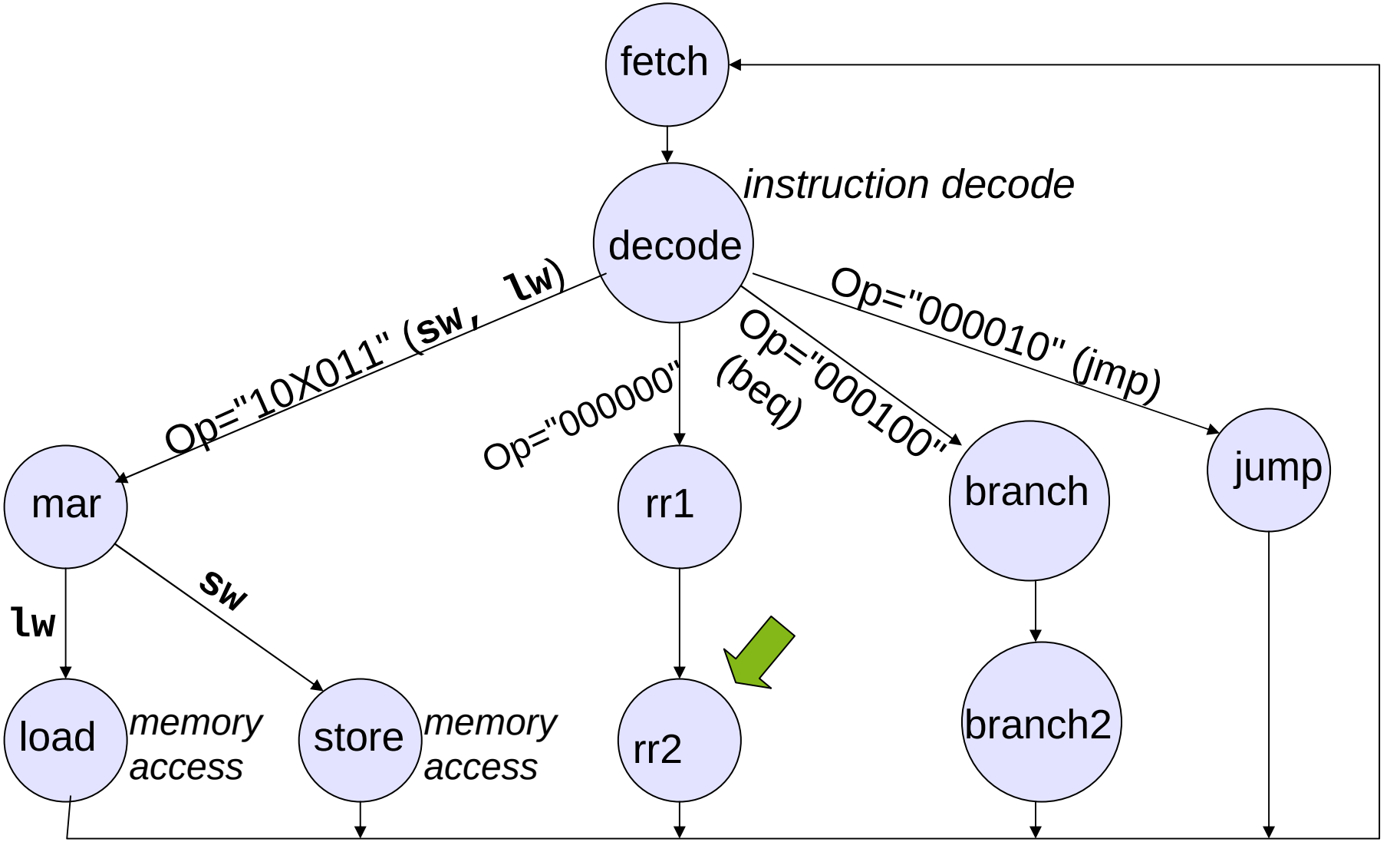

0
0
1
0
0
0
store
0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
§
PC
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister (IR)
*
0
1
i3 a3 a2 a1
Reg
o1 o2
4
sign_
extend
1 0
<<2
0
1
2
3
ALU
T
alu_
control
0
1
2
§
31:
28
*"00“
☞Ravi microprog

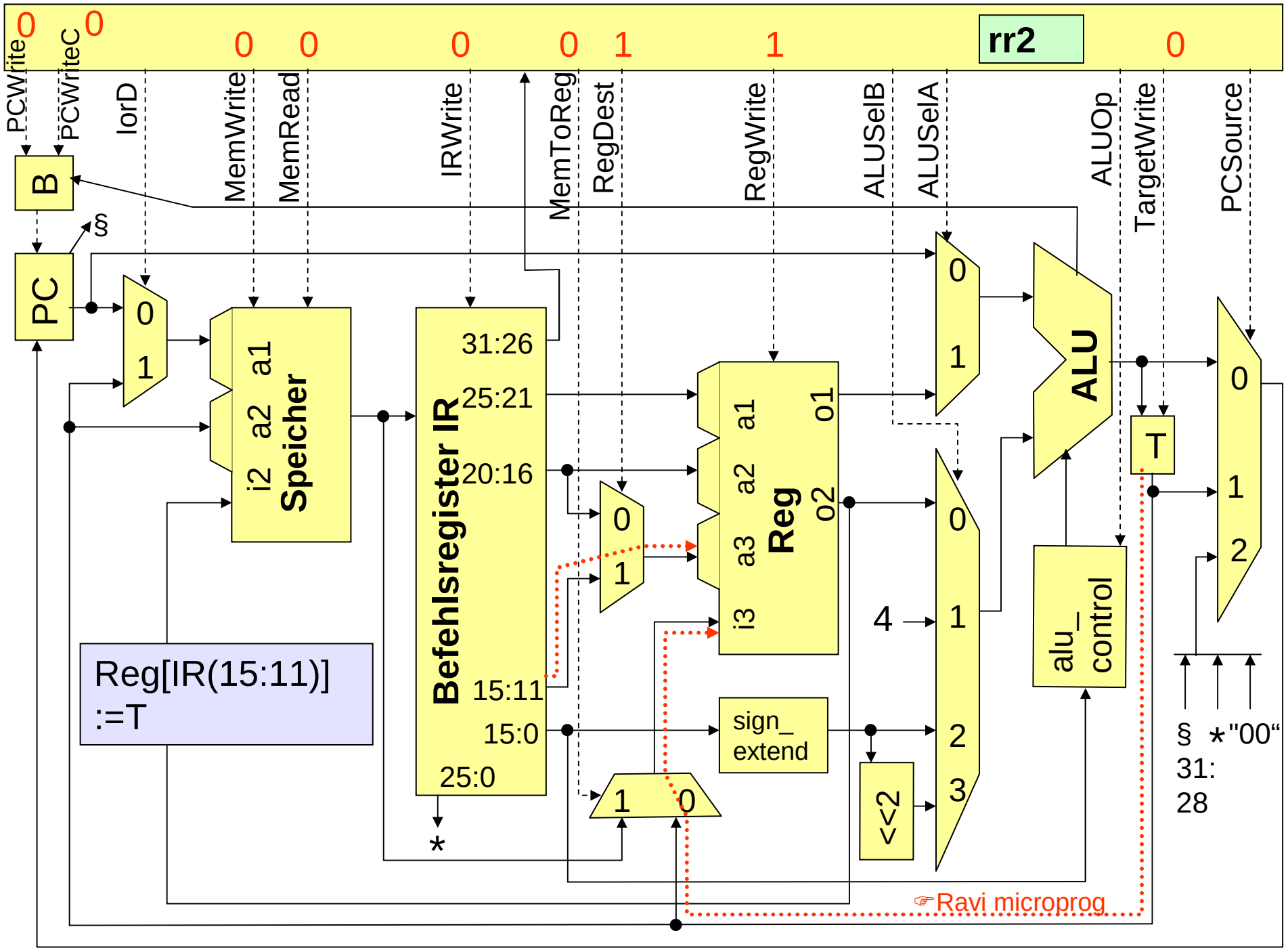
Zustandsgraph
der Ausführung einiger MIPS-Befehle



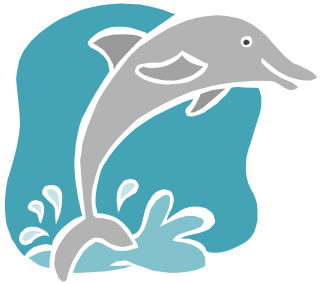

Steuerwerk
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
rr1
IR²
B
PC
§
Speicher
i2 a2 a1
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
Reg
i3 a3 a2 a1
o2 o1
ALU
T
alu_ control
sign_ extend
<<2
4
§ 31: 28
*"00"
*
² ALU-Funktion durch shamt und funct bestimmt. alu_control nimmt geeignete Umkodierung vor.

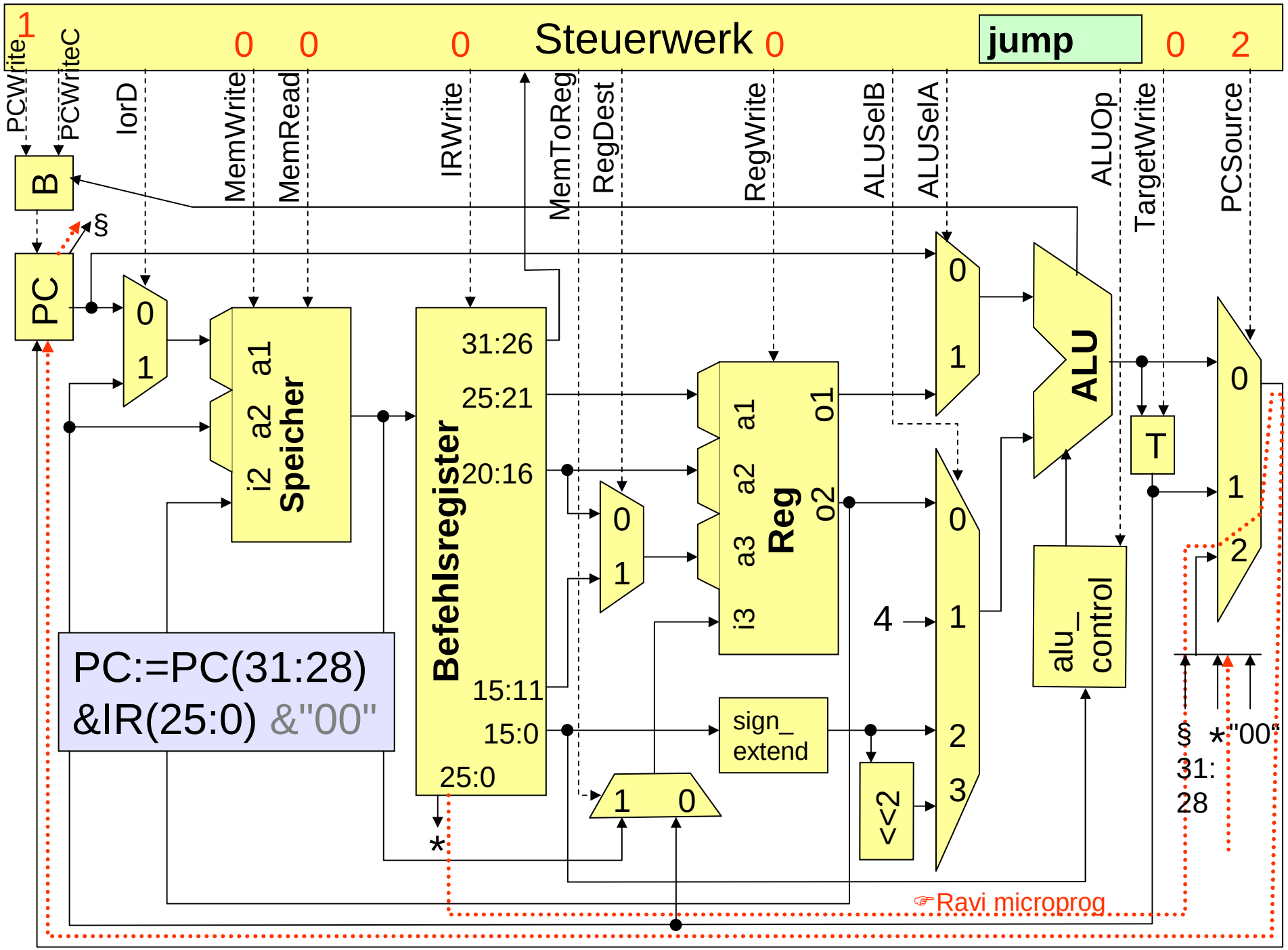
Zustandsgraph der Ausführung einiger MIPS-Befehle



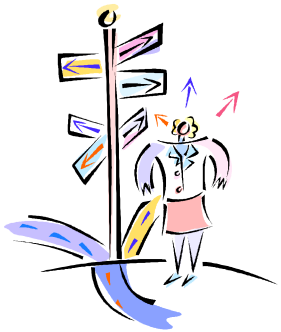

0 0 0 0 0 0 1 1 rr2 0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister IR
*
i3 a3 a2 a1
Reg
o2 o1
4
sign_
extend
<<2
ALU
T
alu_
control
§ 31: 28
*"00“
Reg[IR(15:11)]
:=T
☞Ravi microprog

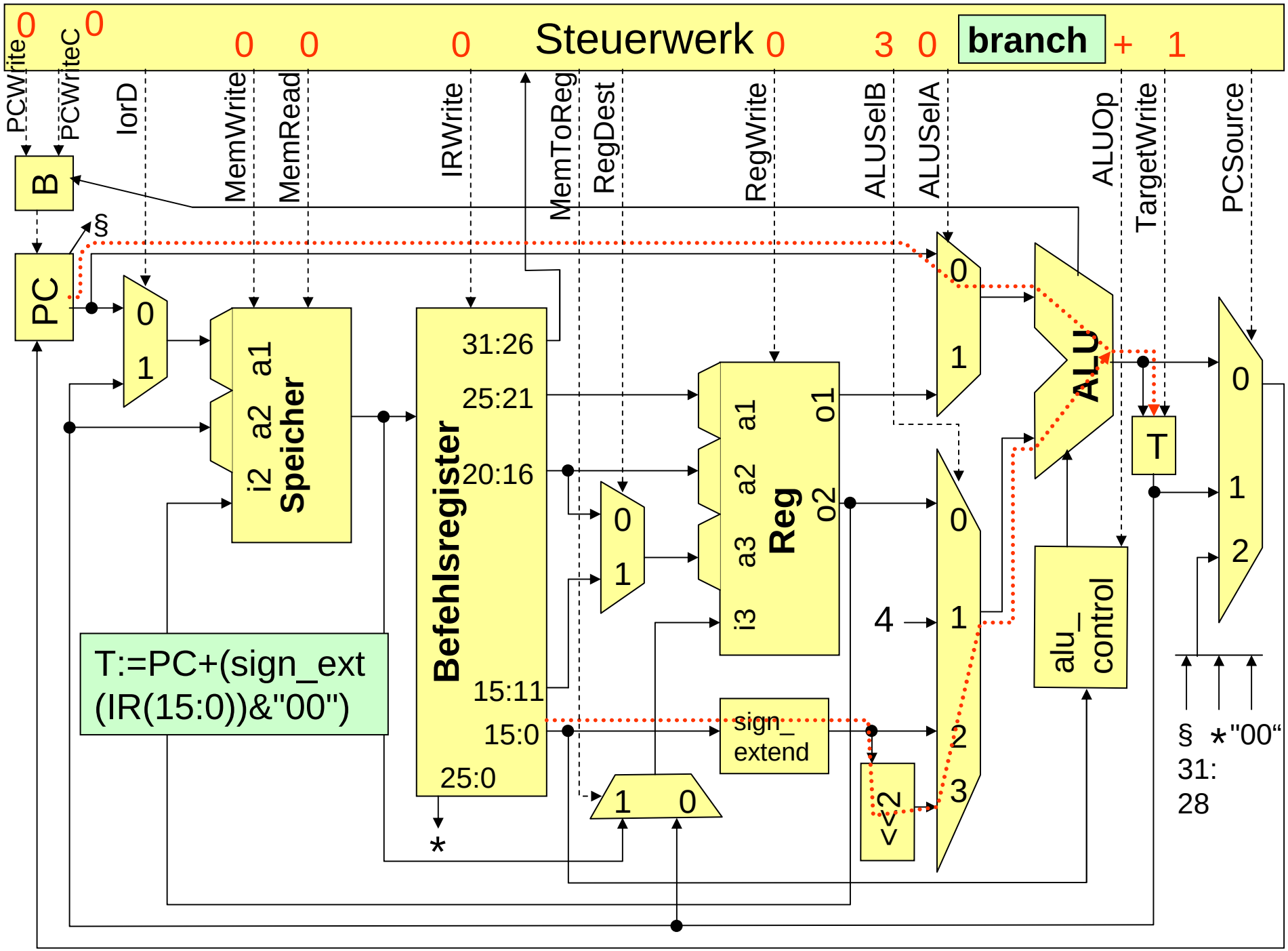
Zustandsgraph der Ausführung einiger MIPS-Befehle



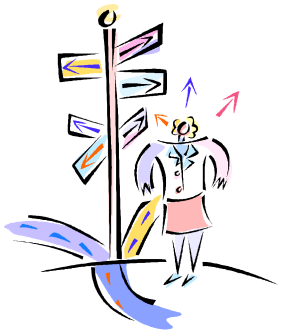

1
PCWrite
PCWriteC
0
IorD
0
MemWrite
MemRead
0
IRWrite
Steuerwerk
MemToReg
0
RegDest
RegWrite
jump
ALUSelB
0
ALUSelA
2
ALUOp
TargetWrite
PCSource
B
§
PC
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister
*
0
1
i3 a3 a2 a1
Reg
o2 o1
0
1
4
0
1
2
3
ALU
T
0
1
2
alu_
control
sign_
extend
<<2
1 0
PC:=PC(31:28)
&IR(25:0) &"00"
§ 31:28
*
"00"
Ravi microprog

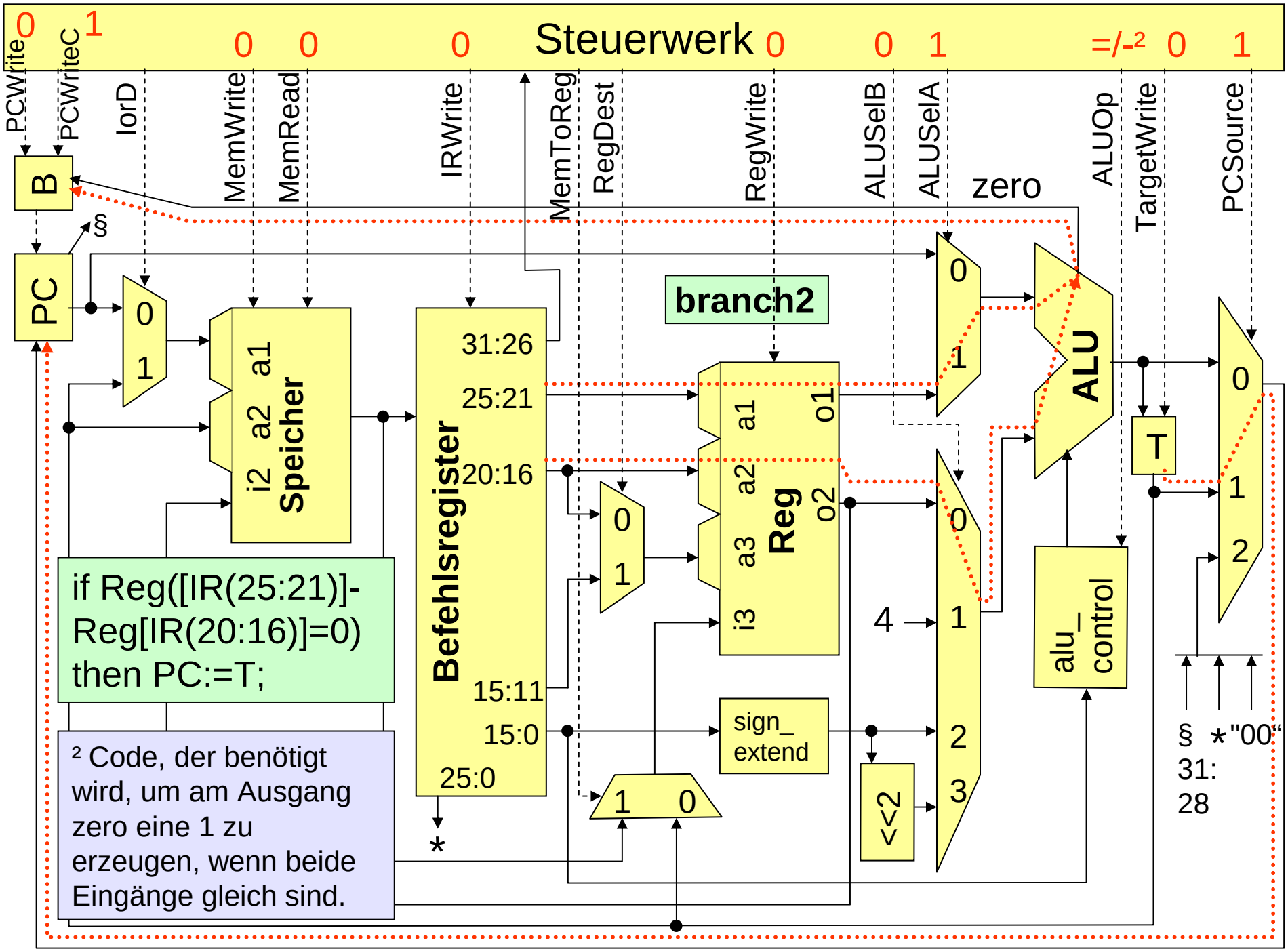
Zustandsgraph
der Ausführung einiger MIPS-Befehle



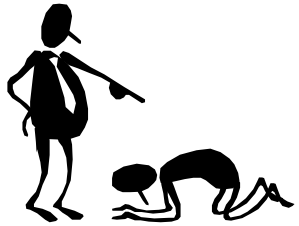

Steuerwerk
branch
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
Speicher
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
Reg
ALU
T
alu_control
sign_extend
<<2
T:=PC+(sign_ext (IR(15:0))&"00")
§ 31: 28
*"00"

Zustandsgraph der Ausführung einiger MIPS-Befehle



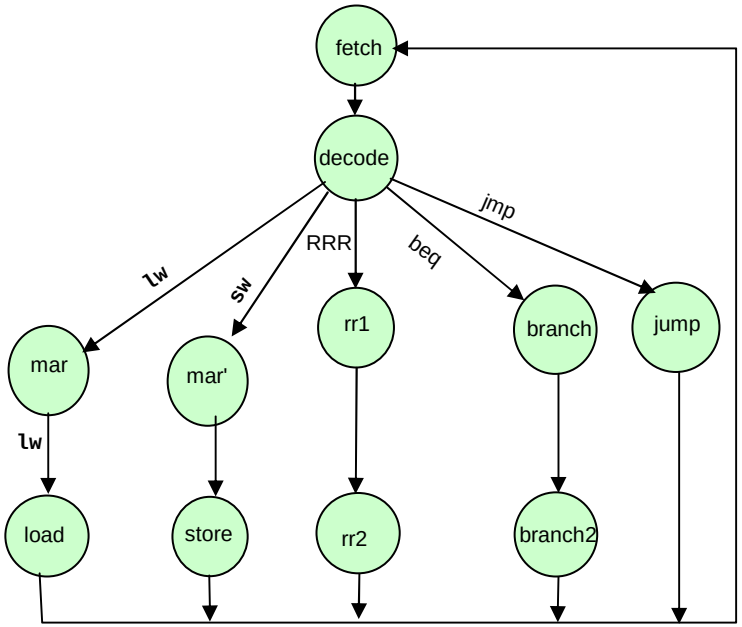

Steuerwerk
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
zero
ALUOp
TargetWrite
PCSource
B
PC
§
Speicher
i2 a2 a1
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
branch2
Reg
i3 a3 a2 a1 o1 o2
ALU
T
alu_ control
sign_ extend
<<2
4
*
§ 31: 28
*"00"
if Reg([IR(25:21)]-Reg[IR(20:16)]=0) then PC:=T;
² Code, der benötigt wird, um am Ausgang zero eine 1 zu erzeugen, wenn beide Eingänge gleich sind.

Das Steuerwerk

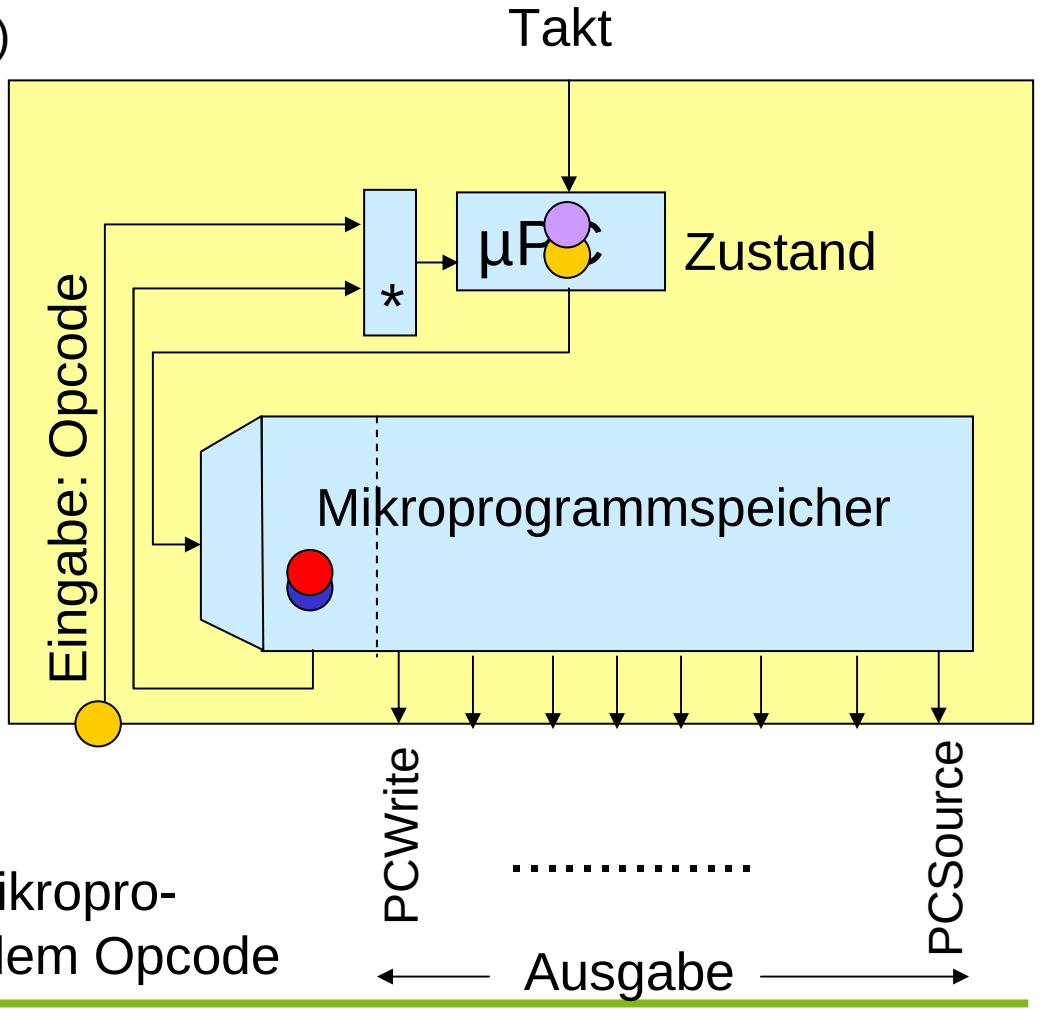


Verhalten, vereinfacht

(mar‘ vermeidet Verzweigung bei mar)

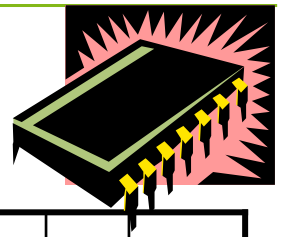


Struktur



* Folgezustand bestimmt durch Mikropro-grammspeicher, bei decode aus dem Opcode

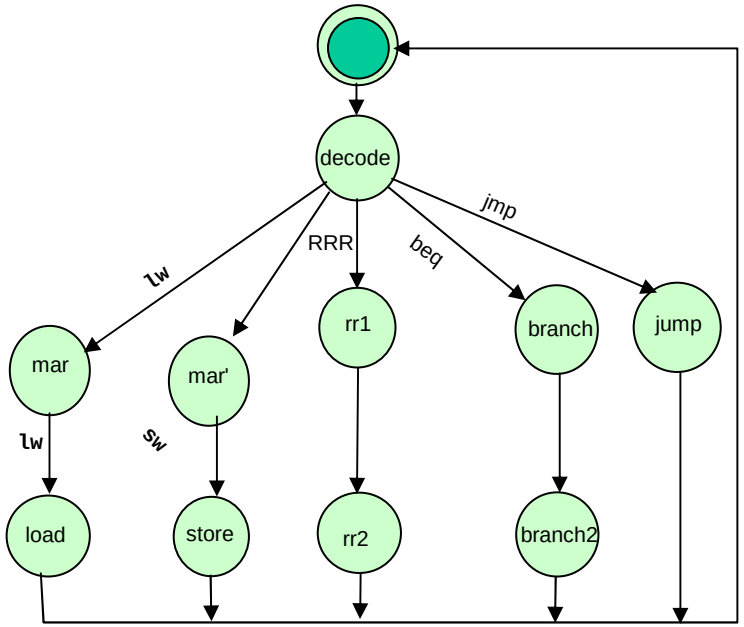
Inhalt des Mikroprogrammspeichers



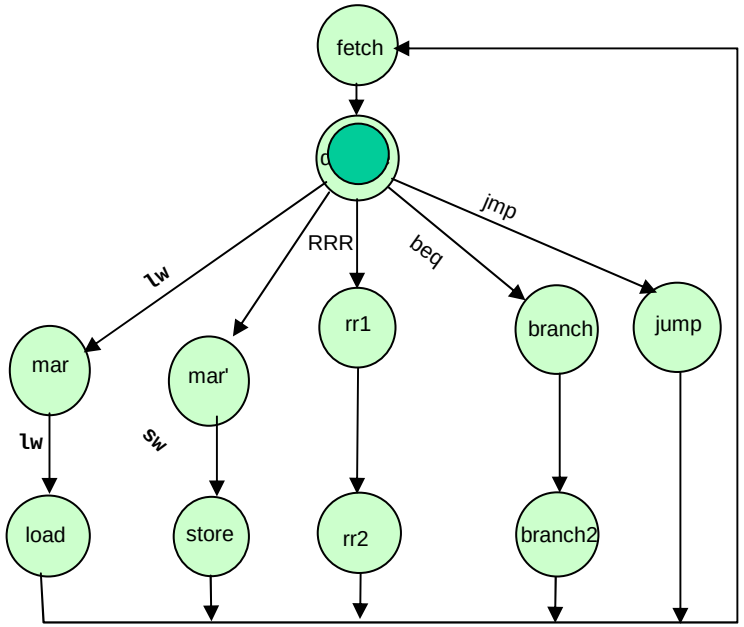
Zustand	Folge-Zustand bzw. – Zustände	PCWrite	PCWriteC	IorD	MemWrite	MemRead	IRWrite	Mem2Reg	RegDest	RegWrite	ALUSelB	ALUSelA	ALUOp	TargetWri	PCSource
fetch	decode	1	0	0	0	1	1	X	X	0	01	0	+	0	00
decode	f(Opcode)	0	0	X	0	0	0	X	X	0	XX	X	X	0	XX
mar, mar‘	load, store	0	0	X	0	0	0	X	X	0	10	1	+	1	XX
load	fetch	0	0	1	0	1	0	1	0	1	XX	X	X	0	XX
store	fetch	0	0	X	1	0	0	X	X	0	XX	X	X	0	XX
rr1	rr2	0	0	X	0	0	0	X	X	0	00	1	IR	1	XX
rr2	fetch	0	0	X	0	0	0	0	1	1	XX	X	X	0	XX
branch	branch2	0	0	X	0	0	0	X	X	0	11	0	+	1	XX
branch2	fetch	0	1	X	0	0	0	X	X	0	00	1	=/-	0	01
jump	fetch	1	0	X	0	0	0	X	X	0	XX	X	X	0	10

+ Art der Bestimmung des Folgezustands

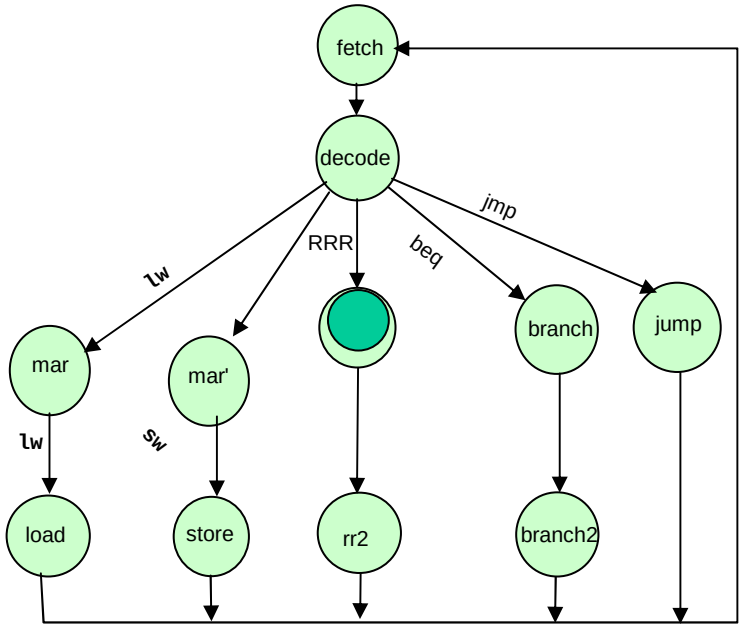
Mikroprogrammierung



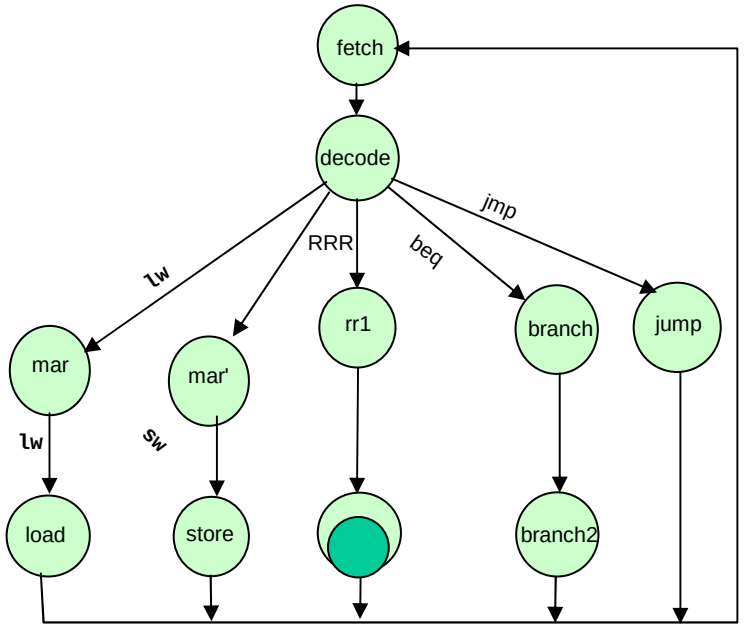
Mikroprogrammierung



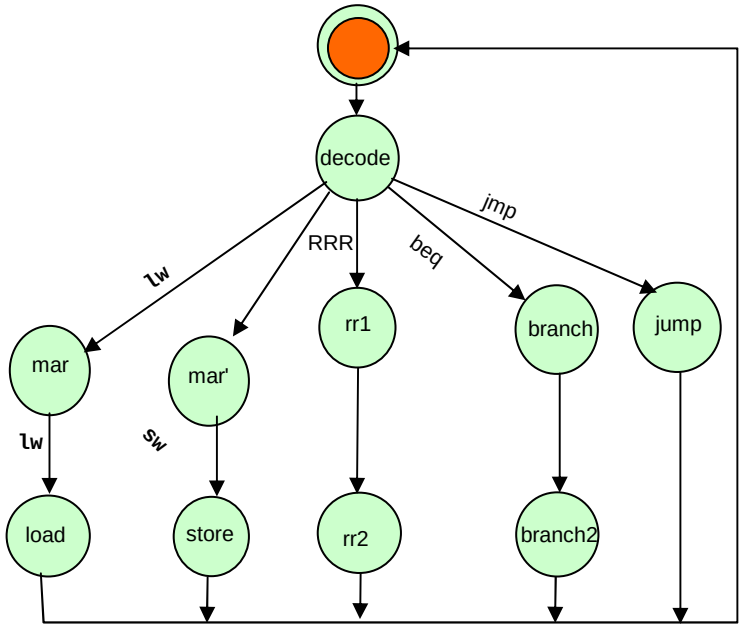
Mikroprogrammierung



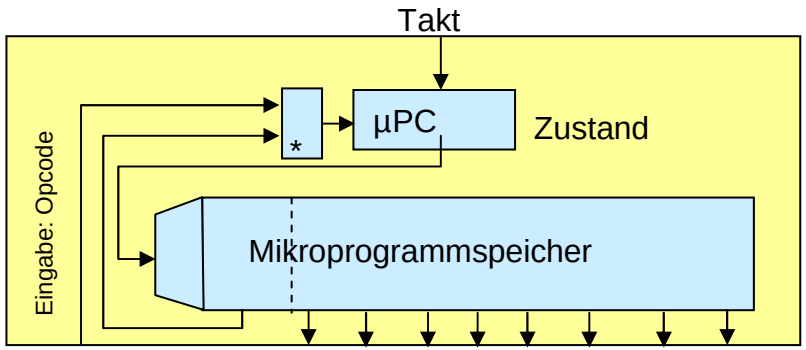
Mikroprogrammierung



Mikroprogrammierung



Struktur



Zusammenfassung

Mikroprogrammierung gestattet die strukturierte Realisierung von Rechensystemen aus RT-Struktur-Komponenten

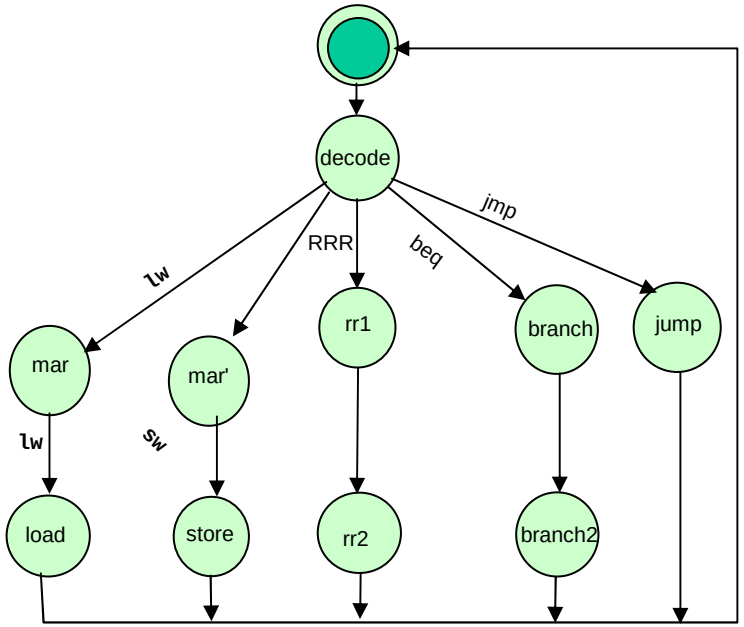
Vorteile:

- einfache, strukturierte Realisierung auch großer, komplexer Befehlssätze
- leichte Änderbarkeit

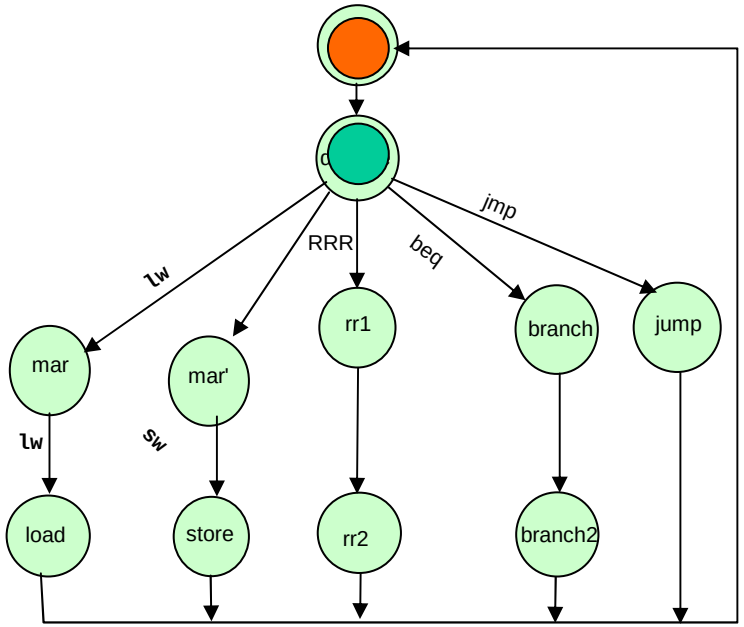
Nachteile:

- Overhead (*fetch* und *decode* enthalten keine Operationen des auszuführenden Programms)
- Große CPI-Werte
- ☞ Versuch, Mikroprogramme zu vermeiden

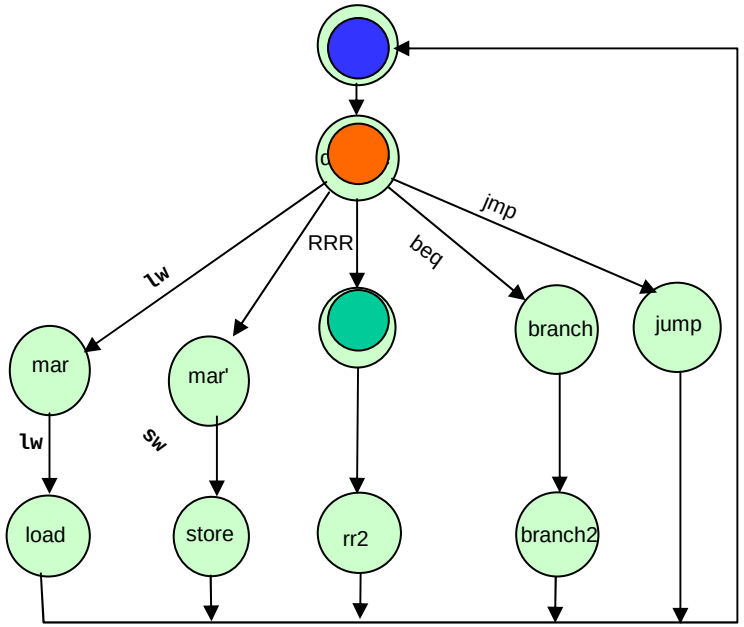
Mikroprogrammierung → Fließbandverarbeitung



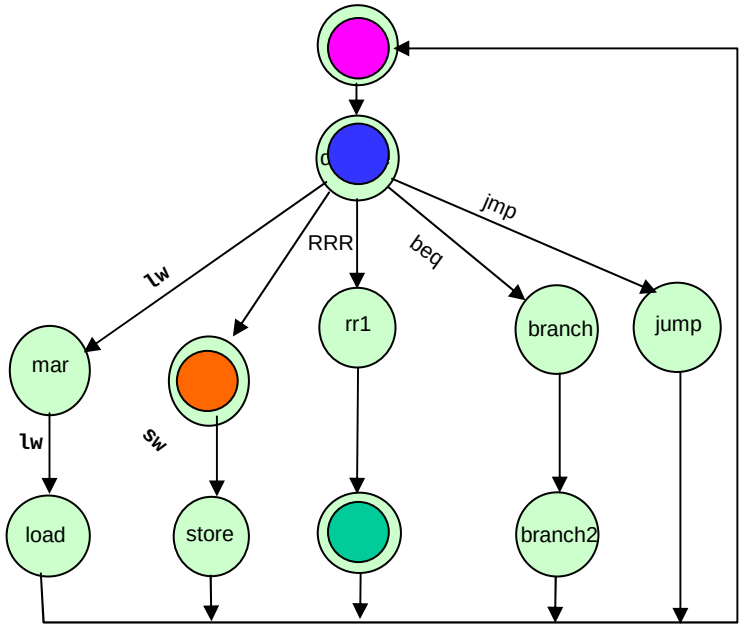
Mikroprogrammierung → Fließbandverarbeitung



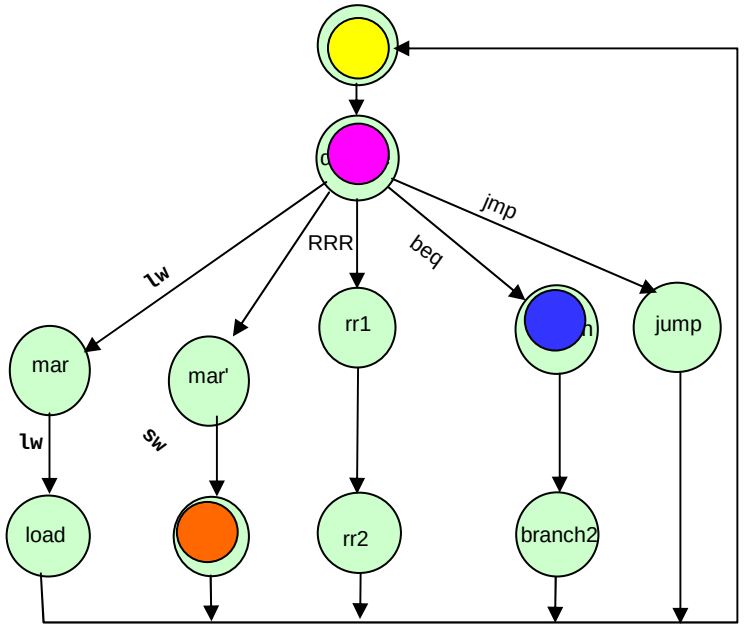
Mikroprogrammierung → Fließbandverarbeitung

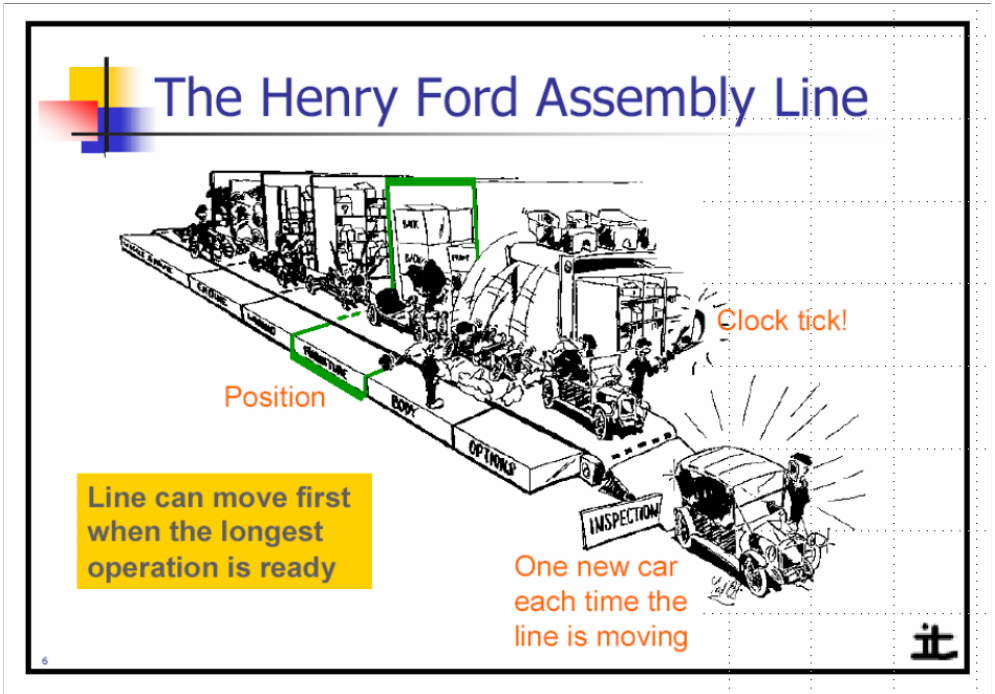


Mikroprogrammierung → Fließbandverarbeitung



Mikroprogrammierung → Fließbandverarbeitung

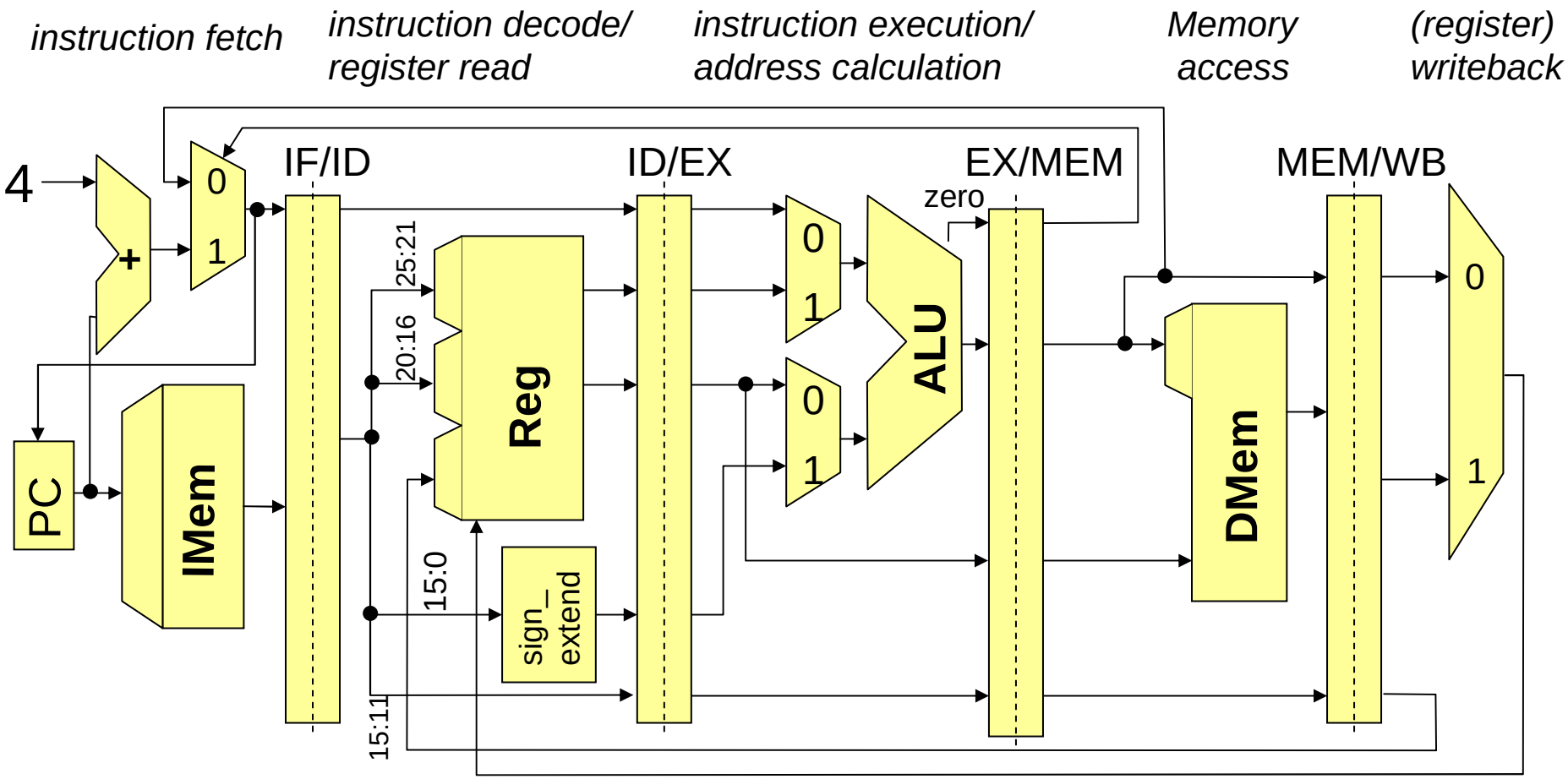




www.it.lth.se/courses/dsi/material/Lectures/Lecture6.pdf

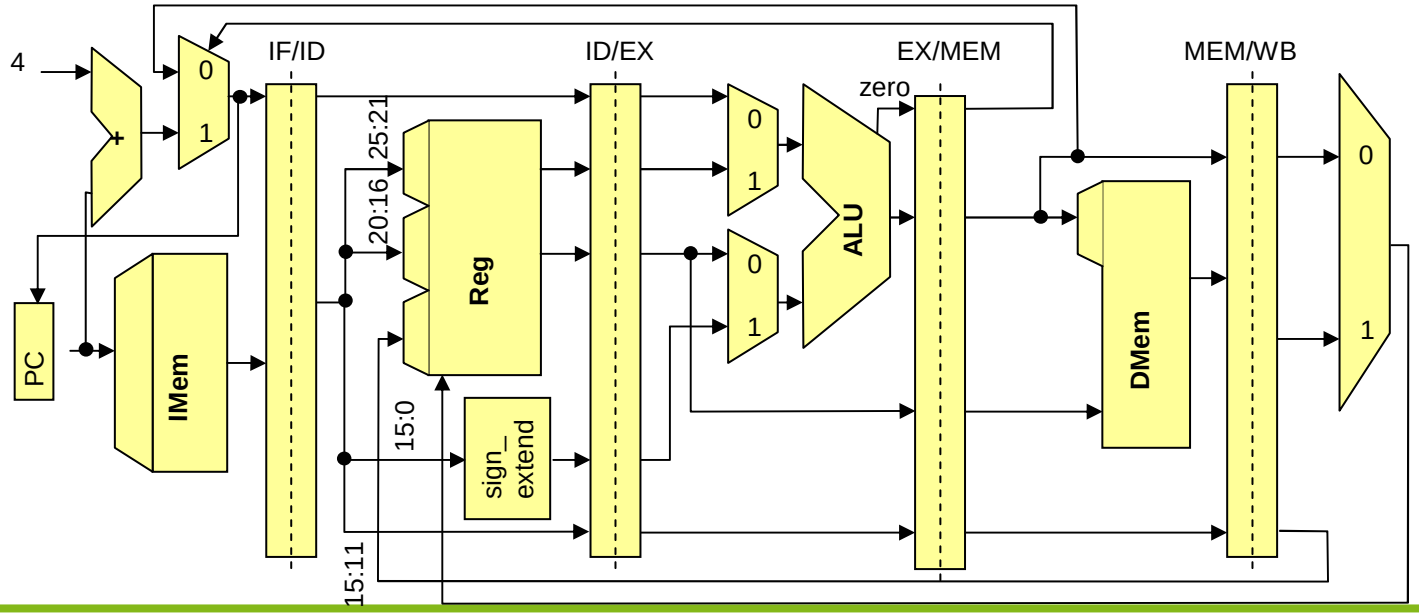
2.3.2 Fließbandverarbeitung

Fließband-Architektur (engl. *pipeline architecture*): Bearbeitung mehrerer Befehle gleichzeitig, analog zu Fertigungsfließbändern. Beispiel MIPS:



Änderungen gegenüber der Struktur ohne Fließband

- Separater Addierer für Programm-Folgeadressen.
- Konzeptuelle Aufteilung des Speichers in Daten- und Befehlsspeicher.
- Aufteilung des Rechenwerks in Fließbandstufen, Trennung durch Pufferregister, **`T`** und **`Befehlsregister`** werden Pufferregistern.

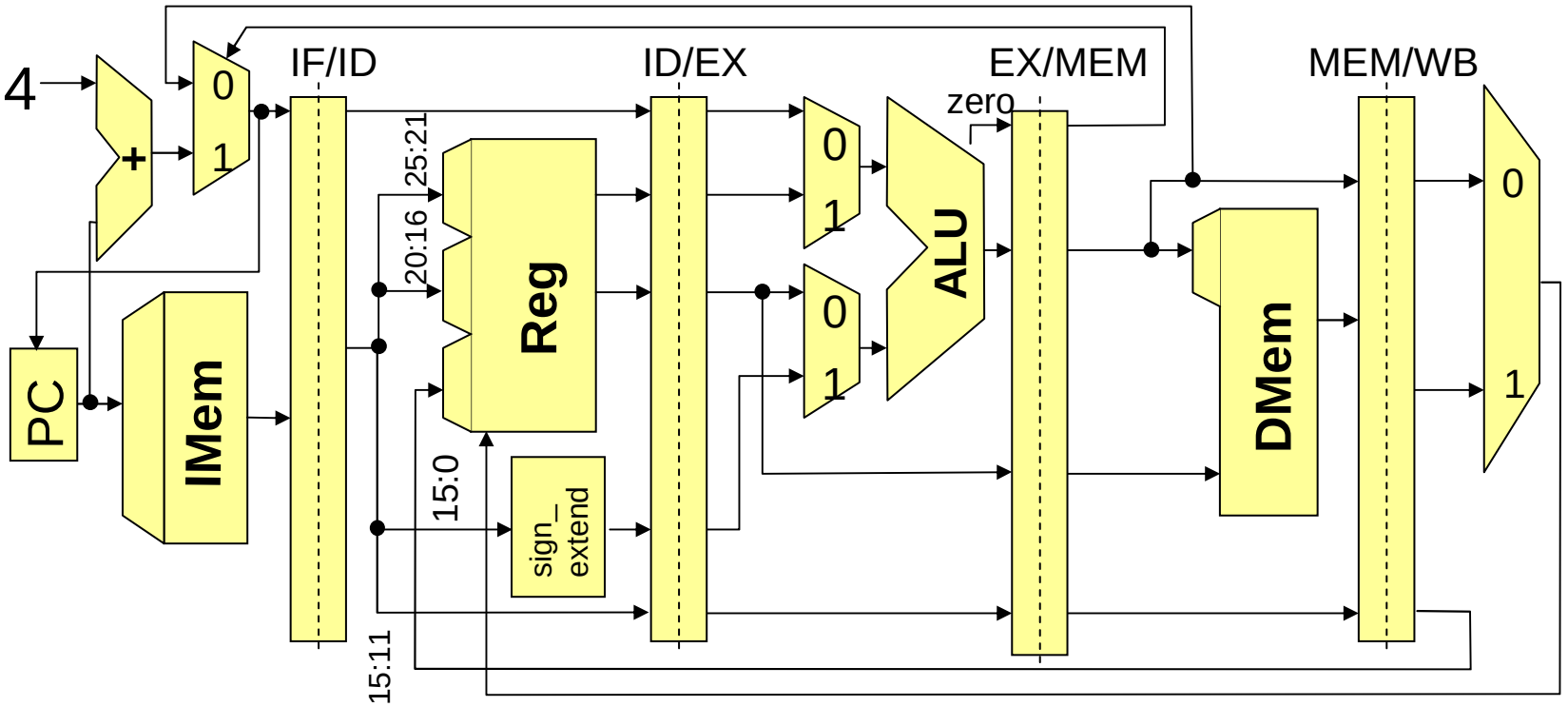


Steuer-
werk
nicht
dar-
gestellt

Aufgaben der einzelnen Phasen bzw. Stufen

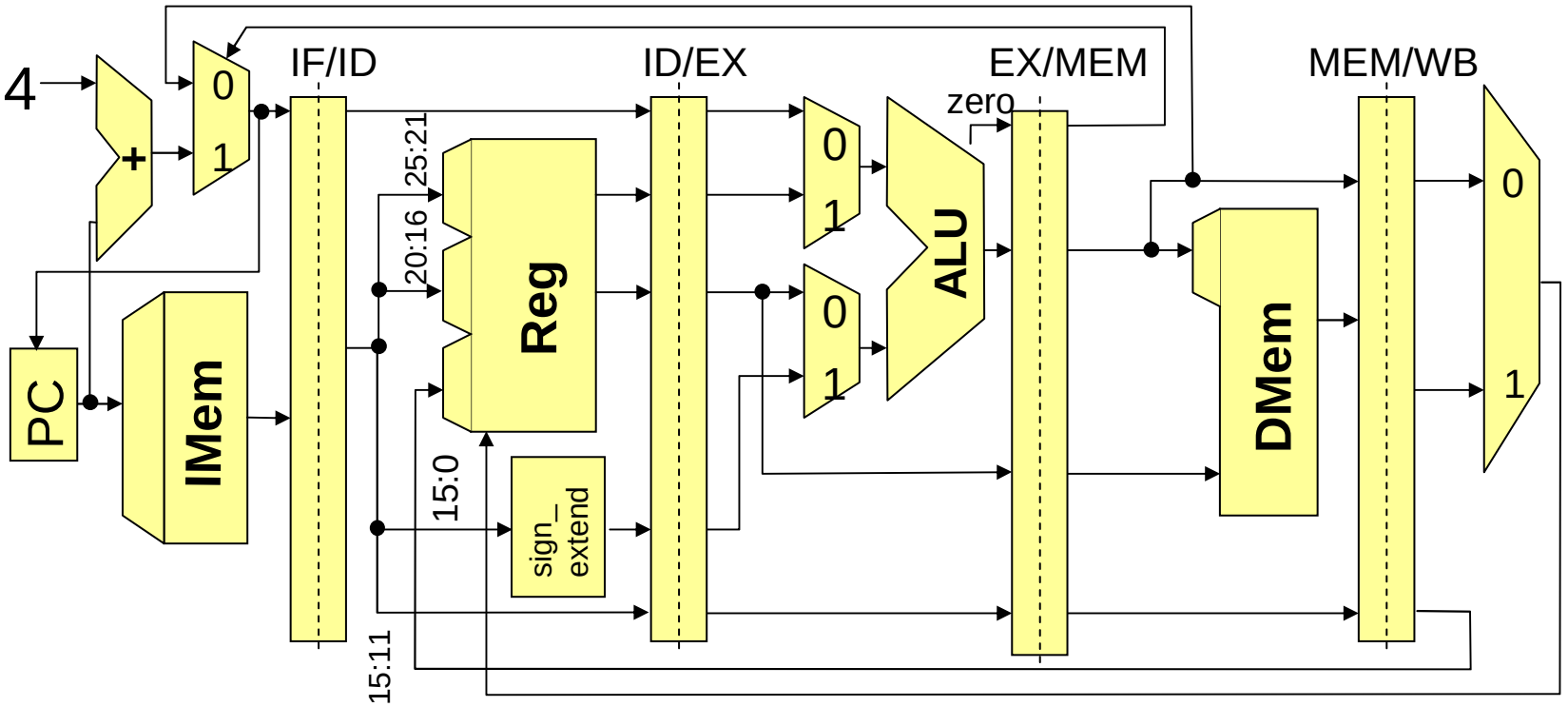
- **Befehlsholphase**
  Lesen des aktuellen Befehls; separater Speicher, zur Vermeidung von Konflikten mit Datenzugriffen (☞ Cache).
- **Dekodier- und Register-Lese-Phase**
  Lesen der Register möglich wegen fester Plätze für Nr.
- **Ausführungs- und Adressberechungsphase**
  Berechnung arithmetischer Funktion bzw. Adresse für Speicherzugriff.
- **Speicherzugriffsphase**
  Wird nur bei Lade- und Speicherbefehlen benötigt.
- **Abspeicherungsphase**
  Speichern in Register, bei Speicherbefehlen nicht benötigt.

Idealer Fließbanddurchlauf



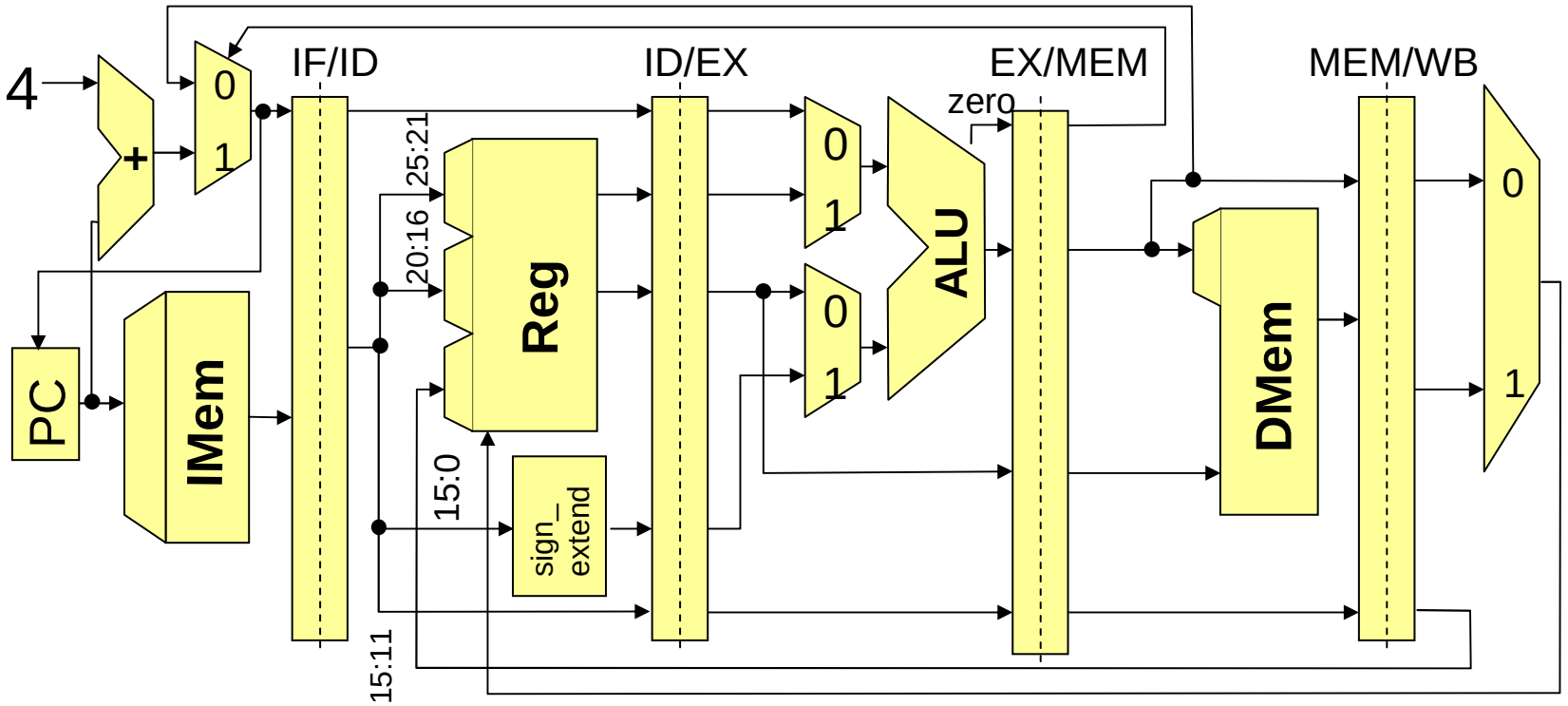
Zyklus 1				
Befehl 1				

Idealer Fließbanddurchlauf



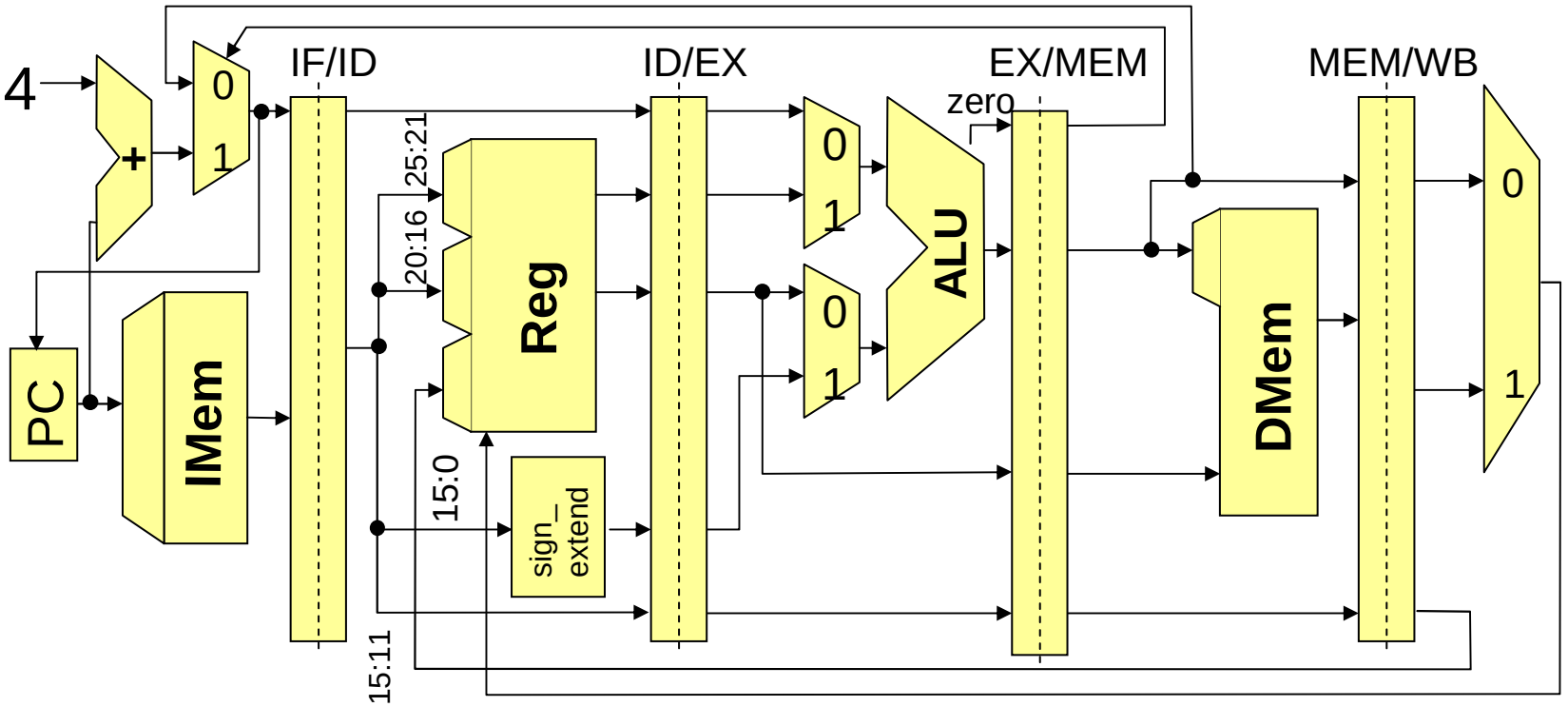
Zyklus 2				
Befehl 2	**Befehl 1**			

Idealer Fließbanddurchlauf



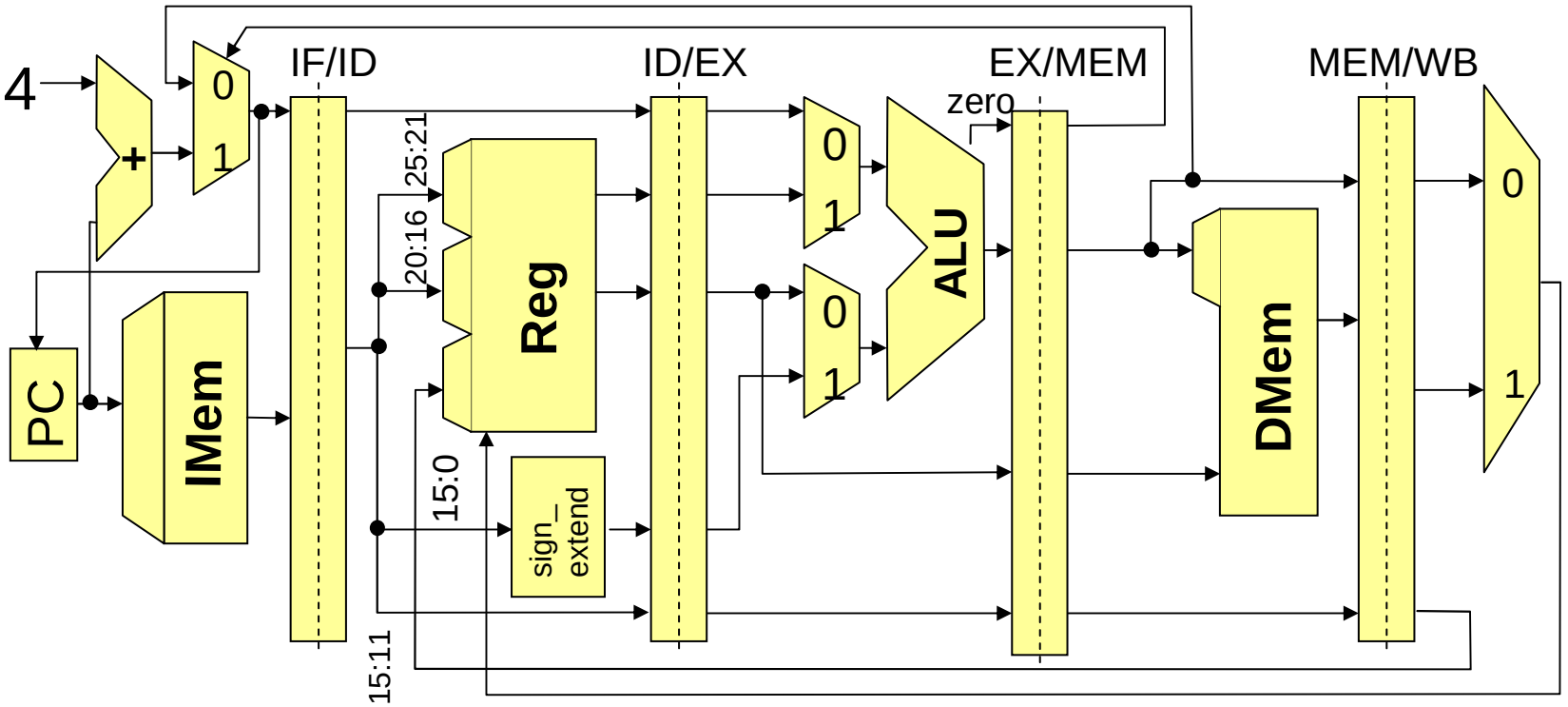
Zyklus 3				
Befehl 3	**Befehl 2**	**Befehl 1**		

Idealer Fließbanddurchlauf



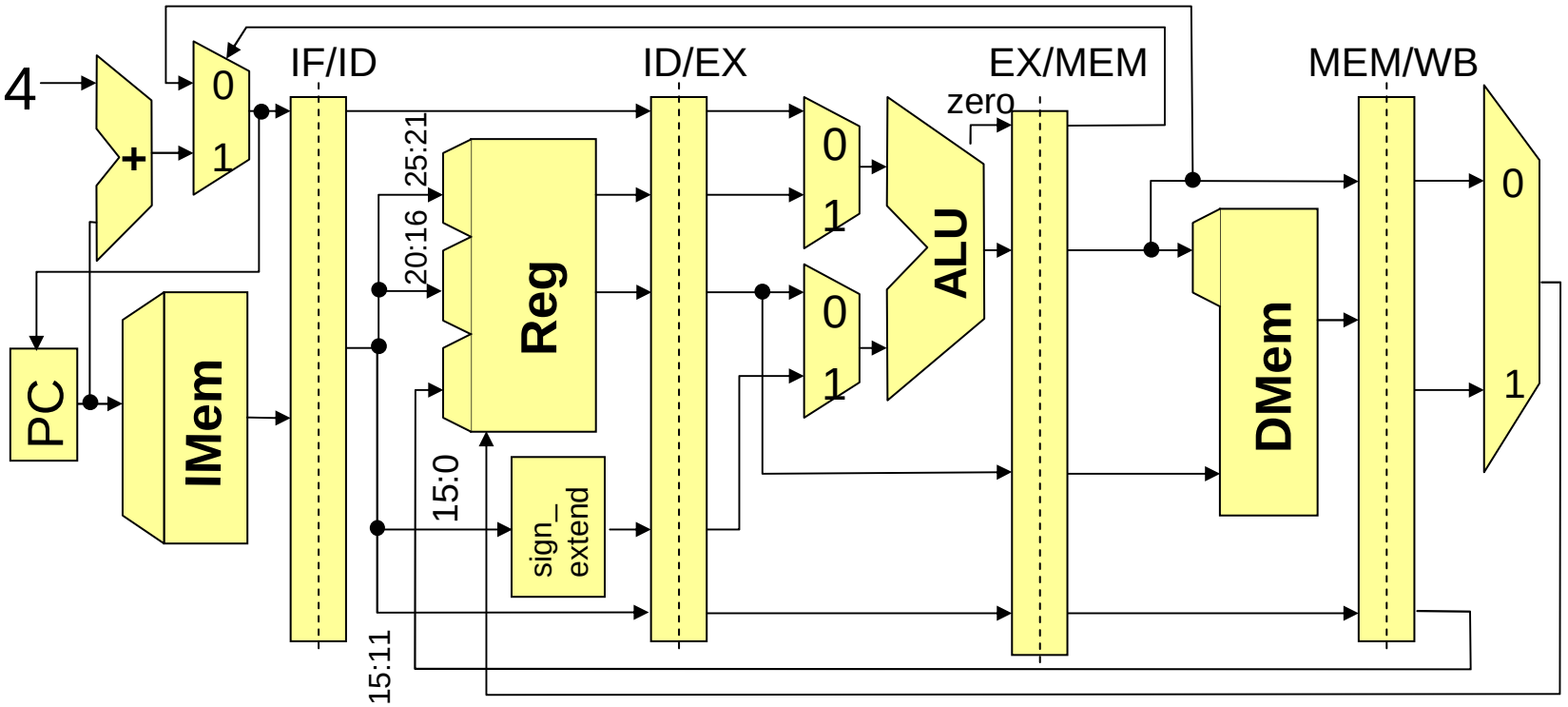
Zyklus 4				
Befehl 4	**Befehl 3**	**Befehl 2**	**Befehl 1**	

Idealer Fließbanddurchlauf

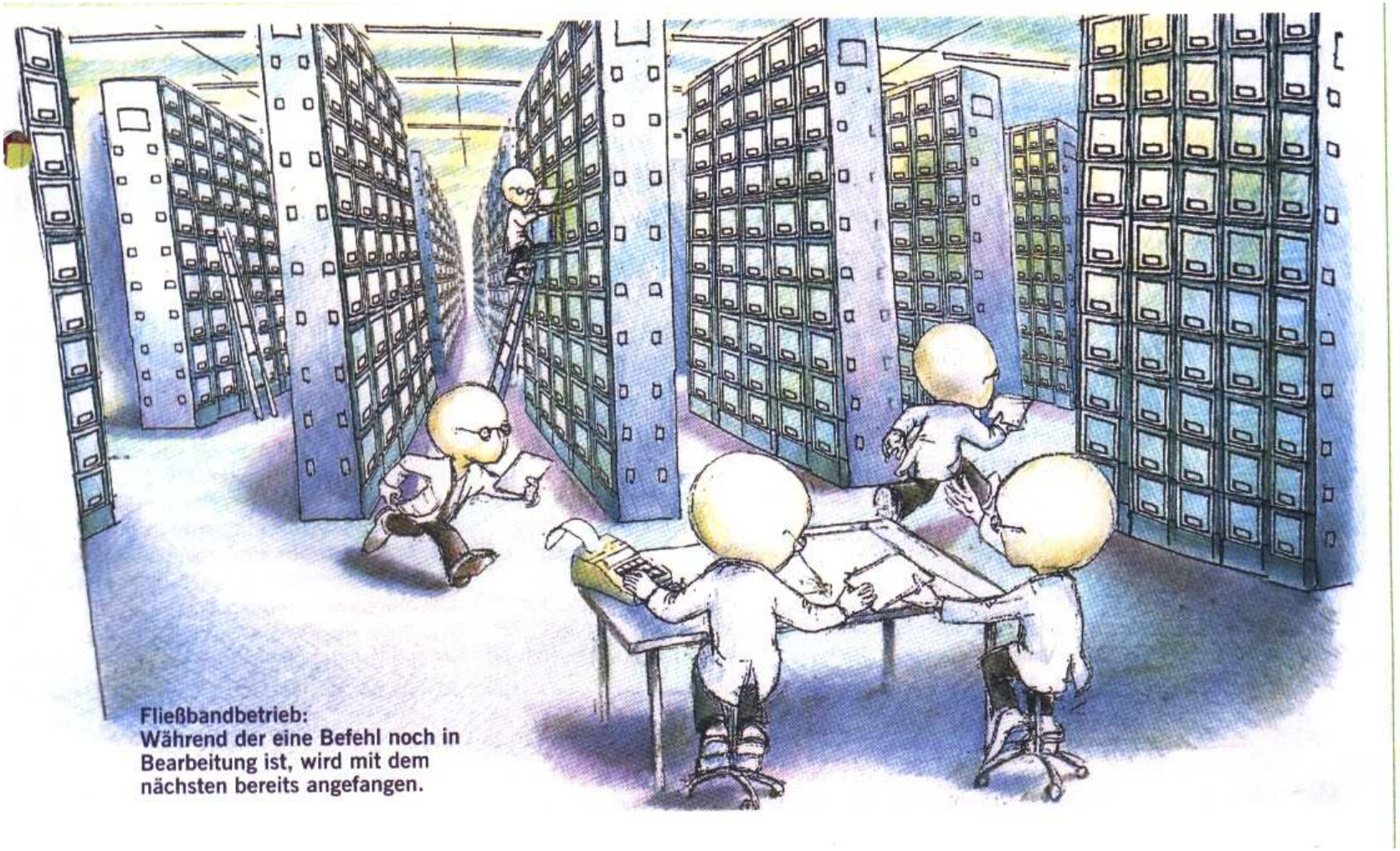


Zyklus 5				
Befehl 5	**Befehl 4**	**Befehl 3**	**Befehl 2**	**Befehl 1**

Idealer Fließbanddurchlauf



Zyklus 6				
Befehl 6	**Befehl 5**	**Befehl 4**	**Befehl 3**	**Befehl 2**


Fließbandbetrieb:
Während der eine Befehl noch in
Bearbeitung ist, wird mit dem
nächsten bereits angefangen.

© Bode, TUM

Pipeline-Hazards

Structural hazards
(deutsch: strukturelle Abhängigkeiten oder Gefährdungen).

Verschiedene Fließbandstufen müssen auf dieselbe Hardware-Komponente zugreifen, weil diese nur sehr aufwändig oder überhaupt nicht zu duplizieren ist.

Beispiele:

Speicherzugriffe, sofern für Daten und Befehle nicht über separate Pufferspeicher (*caches*) eine weitgehende Unabhängigkeit erreicht wird.

Bei Gleitkommaeinheiten lässt sich häufig nicht mit jedem Takt eine neue Operation starten (zu teuer).

☞ Eventuell Anhalten des Fließbandes (*pipeline stall)* nötig.

Datenabhängigkeiten (1)

Gegeben sei eine Folge von Maschinen-Befehlen.

Def.: Ein Befehl *j* heißt von einem vorausgehenden Befehl *i* **datenabhängig**, wenn *i* Daten bereitstellt, die *j* benötigt.

Beispiel:

```
add $12,$2,$3
sub $4,$5,$12
and $6,$12,$7
or  $8,$12,$9
xor $10,$12,$11
```

Diese 4 Befehle sind vom **add**-Befehl wegen **$12** datenabhängig

Diese Art der Abhängigkeit heißt (bei Hennessy und anderen) *read after write*- (oder RAW-) Abhängigkeit.

Datenabhängigkeiten (2)

Gegeben sei wieder eine Folge von Maschinen-Befehlen.

Def.: Ein Befehl *i* heißt von einem **nachfolgenden** Befehl *j* **antidatenabhängig**, falls *j* eine Speicherzelle beschreibt, die von *i* noch gelesen werden müsste.

Beispiel:

```
add $12,$2,$3
sub $4,$5,$12
and $6,$12,$7
or  $12,$12,$9
xor $10,$12,$11
```

Diese 2 Befehle sind vom **or**-Befehl wegen **$12** antidatenabhängig

Diese Art der Abhängigkeit heißt (bei Hennessy und anderen) *write after read* - (oder WAR-) Abhängigkeit.

Datenabhängigkeiten (3)

Gegeben sei (wieder) eine Folge von Maschinen-Befehlen.

Def.: Zwei Befehle *i* und *j* heißen voneinander **Ausgabe-abhängig**, falls *i* und *j* dieselbe Speicherzelle beschreiben.

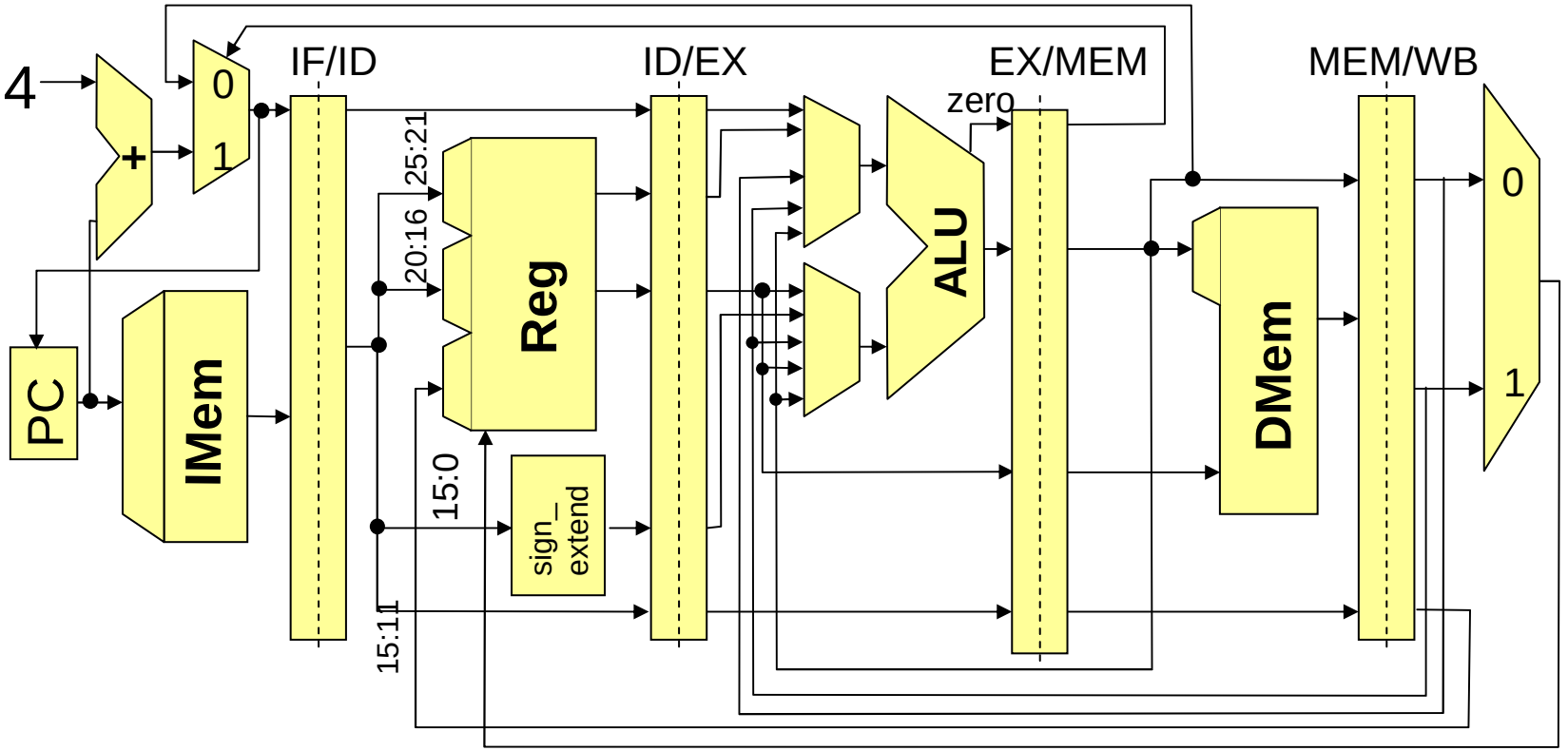
Beispiel:

```
add $12,$2,$3
sub $4,$5,$12
and $6,$12,$7
or  $12,$12,$9
xor $10,$12,$11
```

Voneinander ausgabeabhängig.

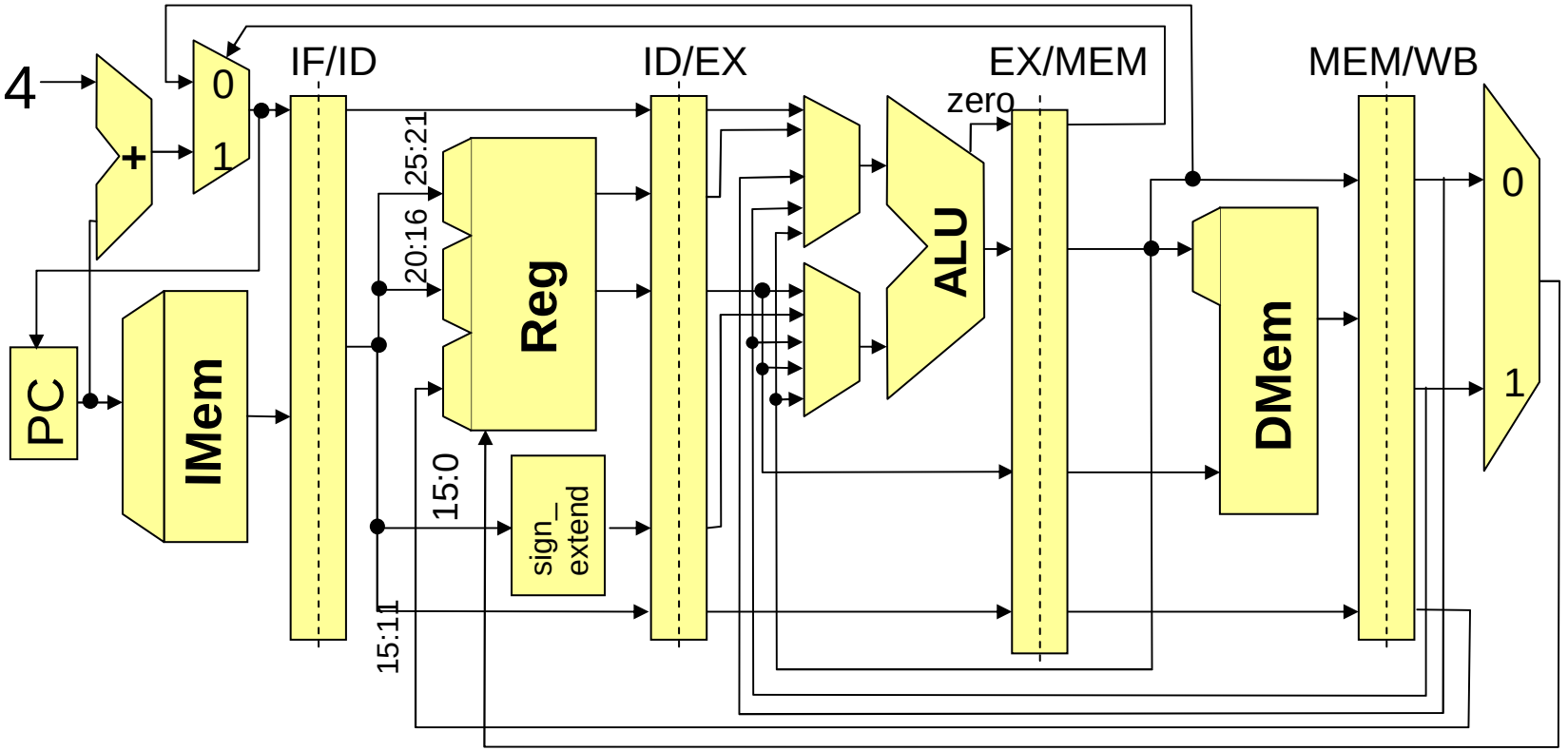
Diese Art der Abhängigkeit heißt (bei Hennessy und anderen) *write after write* - (oder WAW-) Abhängigkeit.

Bypässe, *forwarding*: Behandlung des *data hazards* bei and und sub



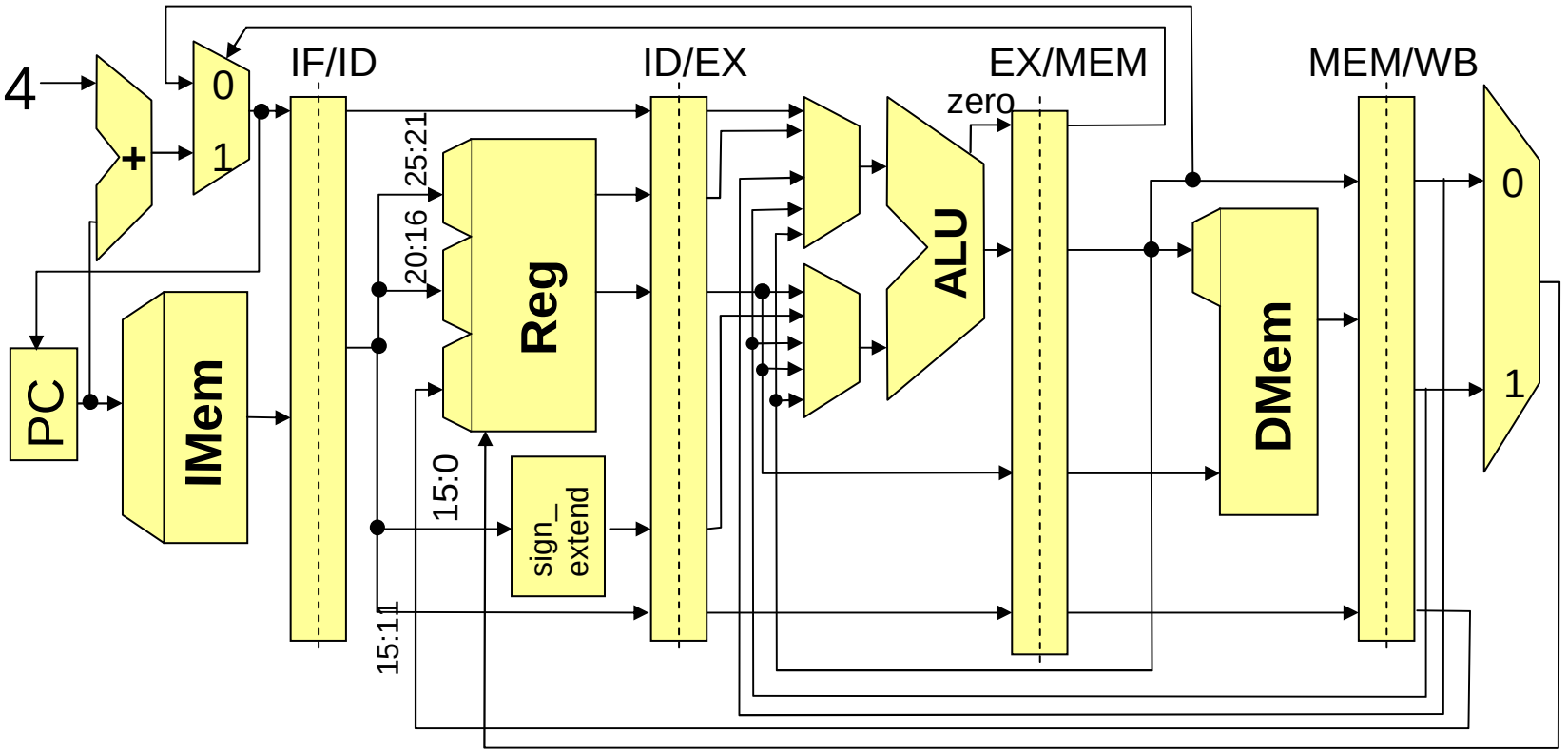
Zyklus 1				
add $1,$2,$3				

Bypässe, *forwarding*: Behandlung des *data hazards* bei and und sub



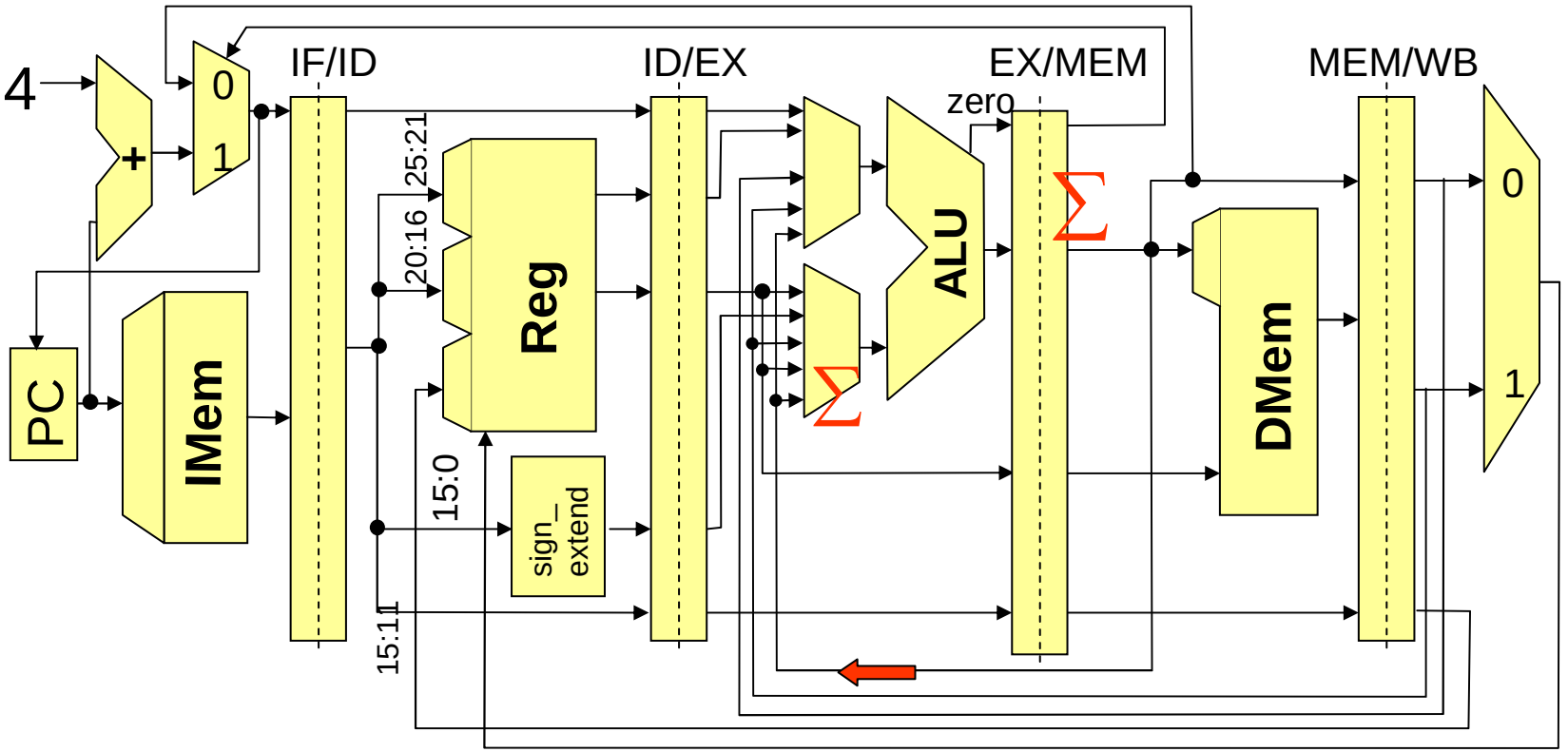
Zyklus 2				
sub $4,$5,$1	**add $1,$2,$3**			

Bypässe, *forwarding*: Behandlung des *data hazards* bei and und sub



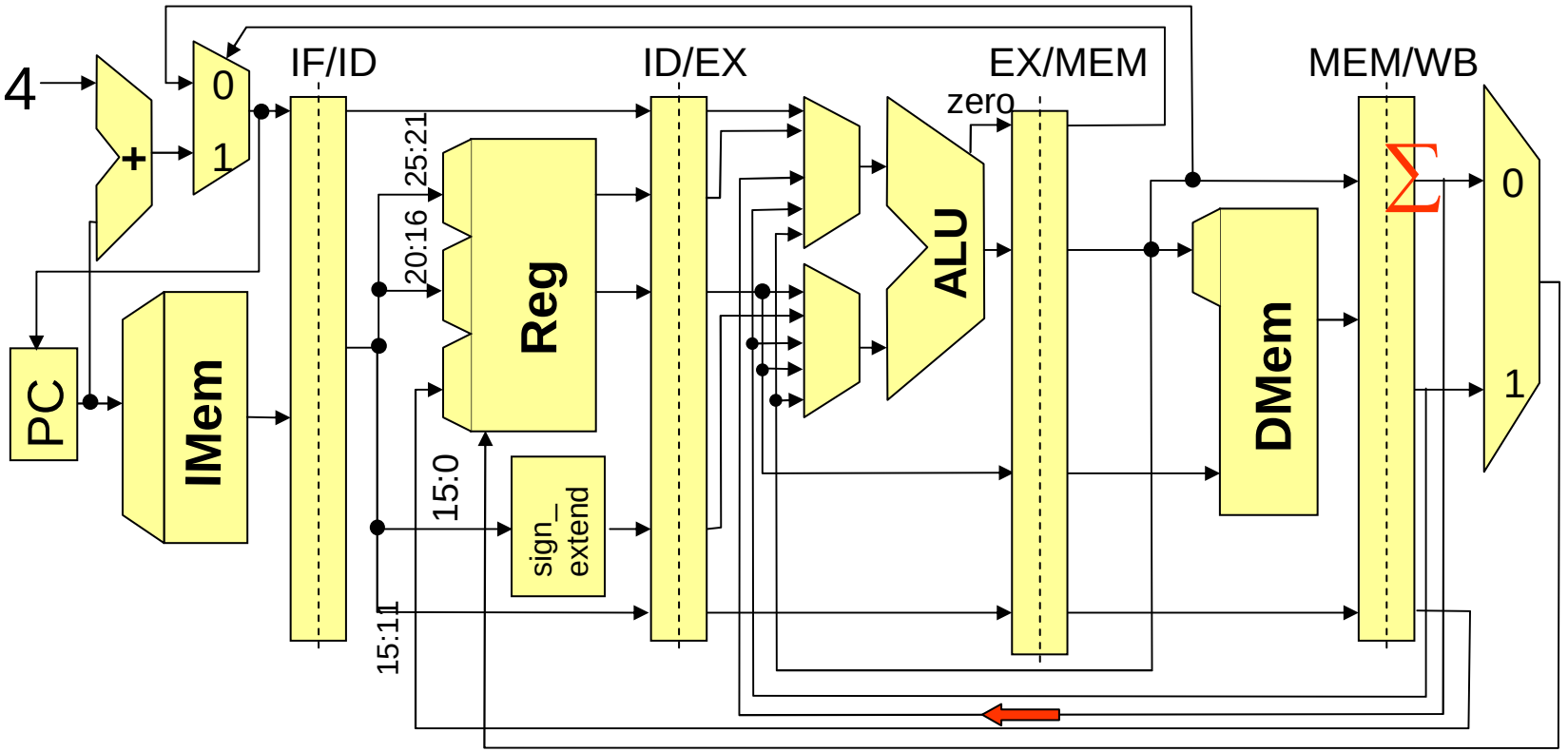
Zyklus 3				
and $6,$1,$7	**sub $4,$5,$1**	**add $1,$2,$3**		

Bypässe, *forwarding*: Behandlung des *data hazards* bei and und sub



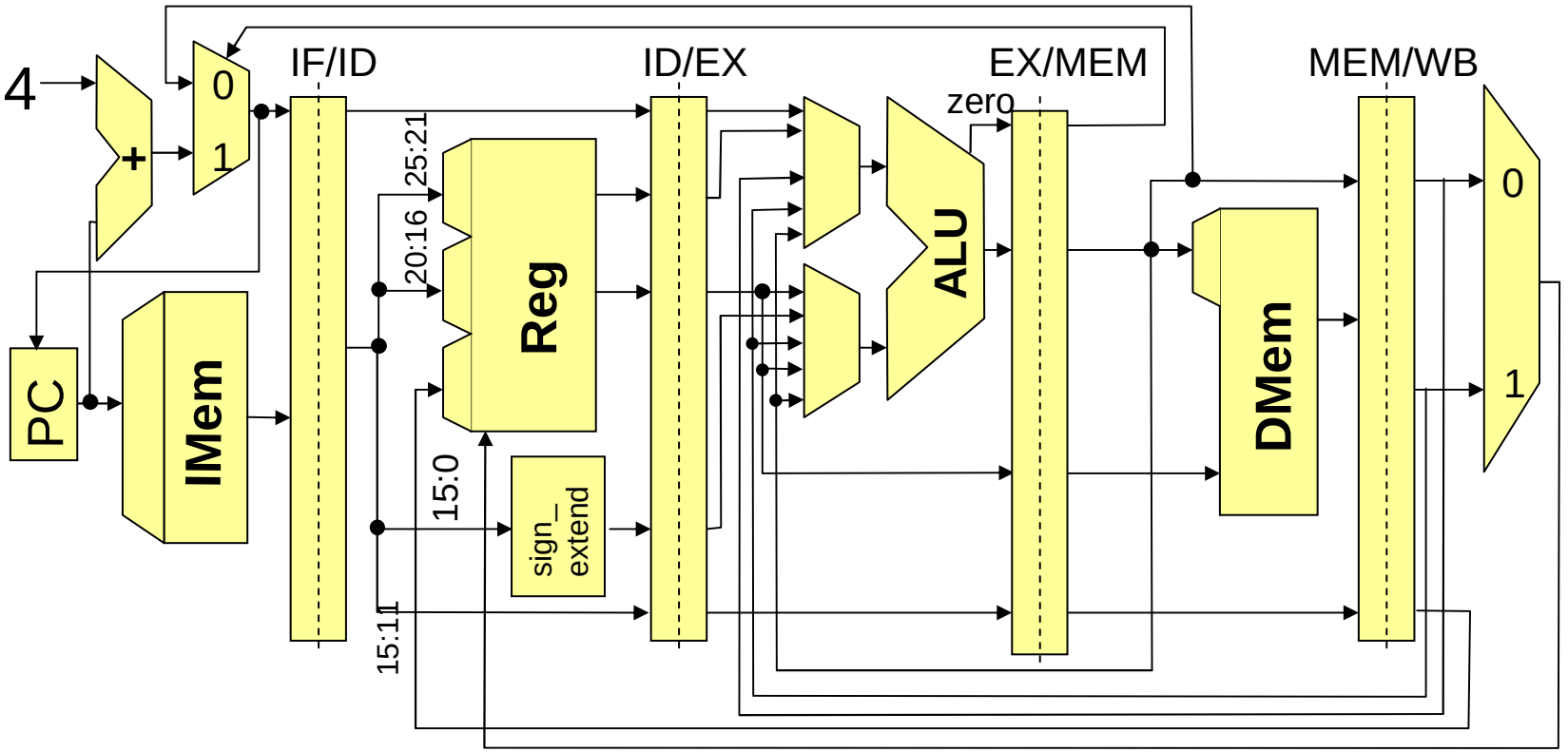
Zyklus 4				
or $8,$1,$9	**and $6,$1,$7**	**sub $4,$5,$1**	**add $1,$2,$3**	

Bypässe, *forwarding*: Behandlung des *data hazards* bei and und sub



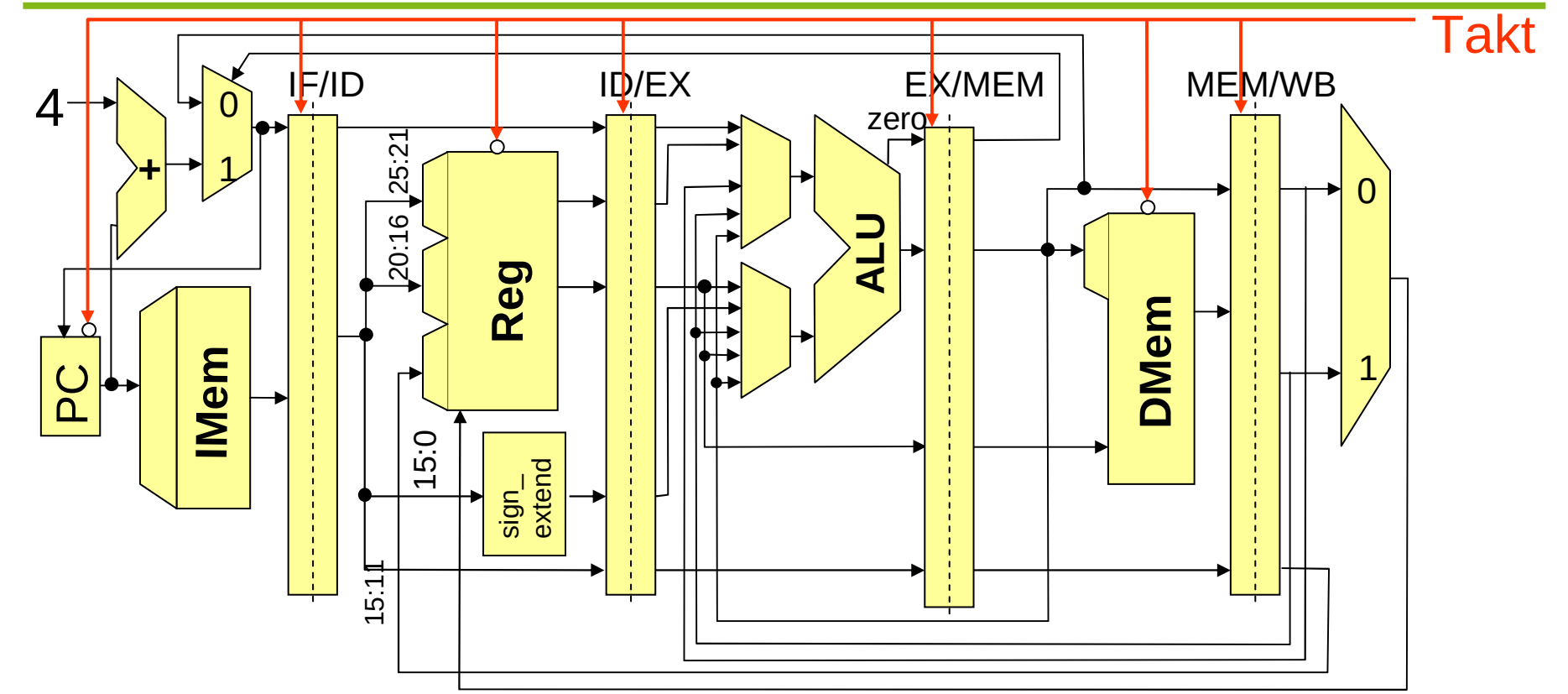
Zyklus 5				
xor $10,$1,$11	or $8,$1,$9	and $6,$1,$7	sub $4,$5,$1	add $1,$2,$3

Bypässe, *forwarding*: Behandlung des *data hazards* bei and und sub



Zyklus 6				
?	xor $10,$1,$11	or $8,$1,$9	and $6,$1,$7	sub $4,$5,$1

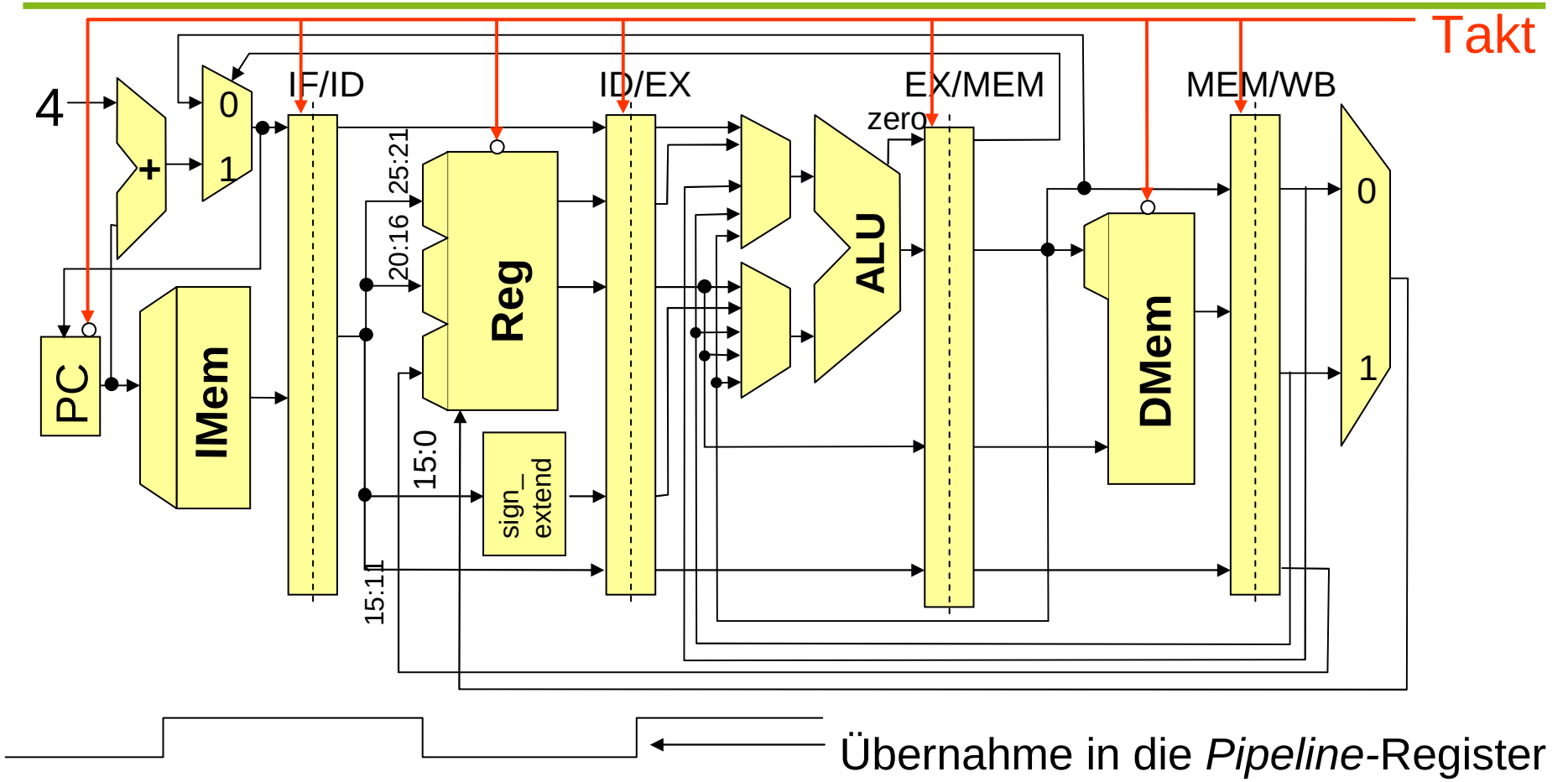
Taktung zur Behandlung des *data hazards* bei or



Übernahme in die *Pipeline*-Register

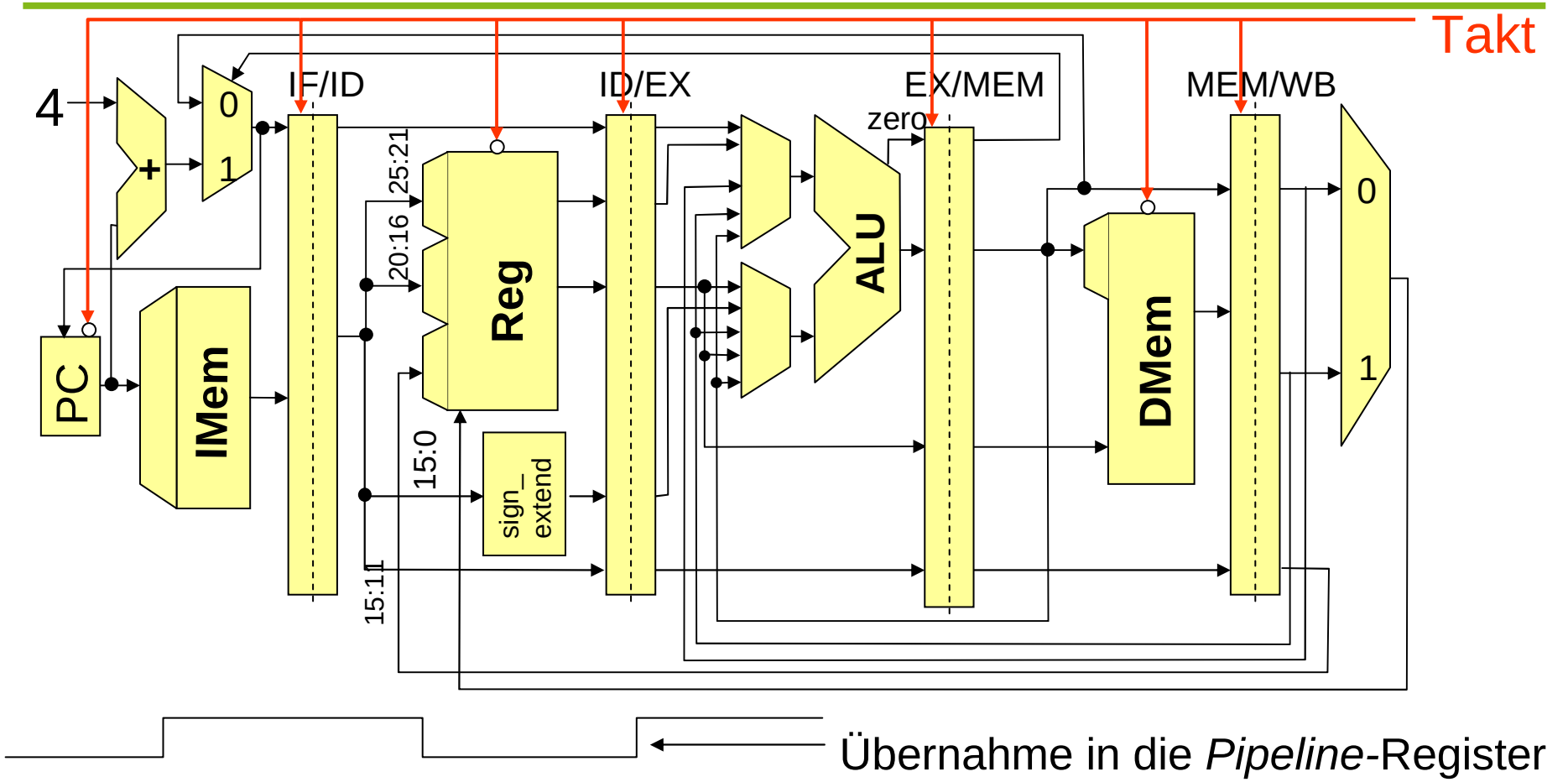
Zyklus 1				
add $1,$2,$3				

Taktung zur Behandlung des *data hazards* bei or



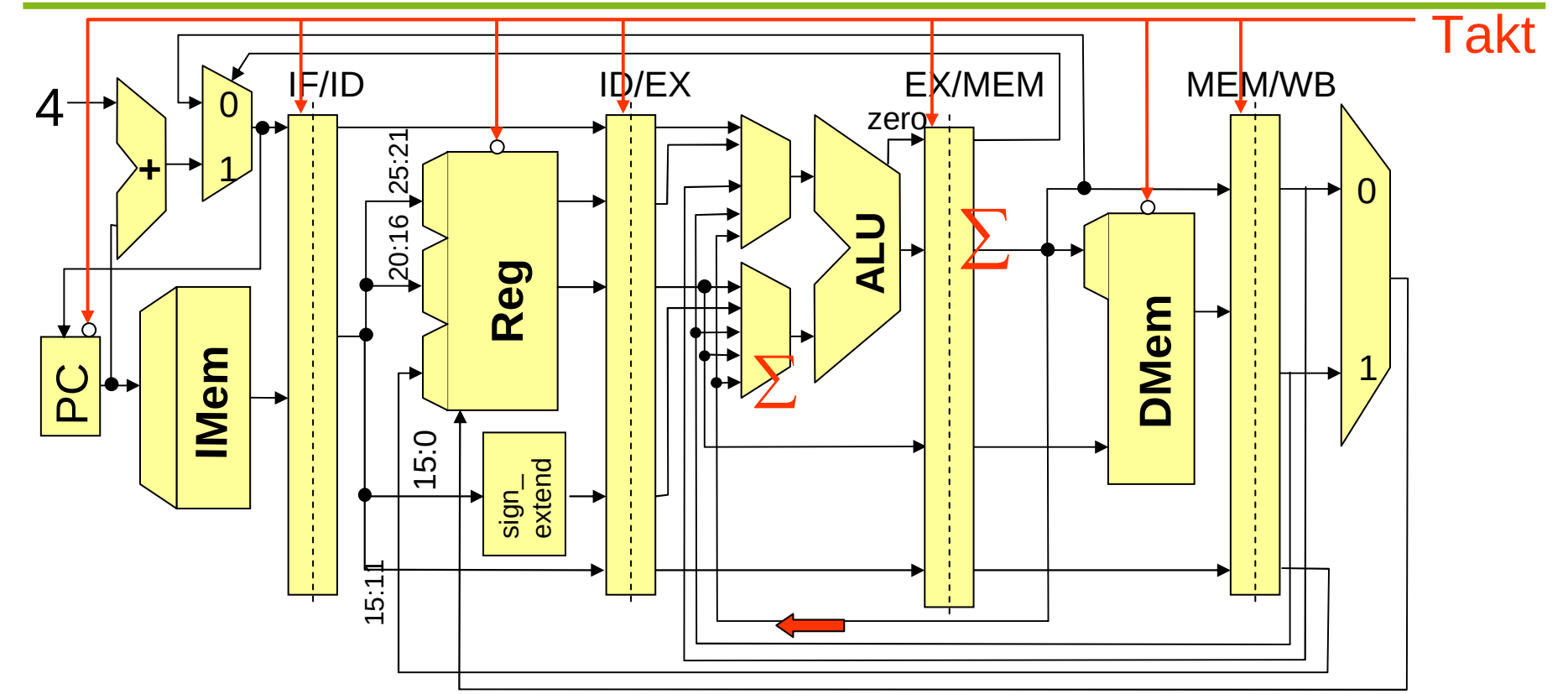
Zyklus 2				
sub $4,$5,$1	**add $1,$2,$3**			

Taktung zur Behandlung des *data hazards* bei or



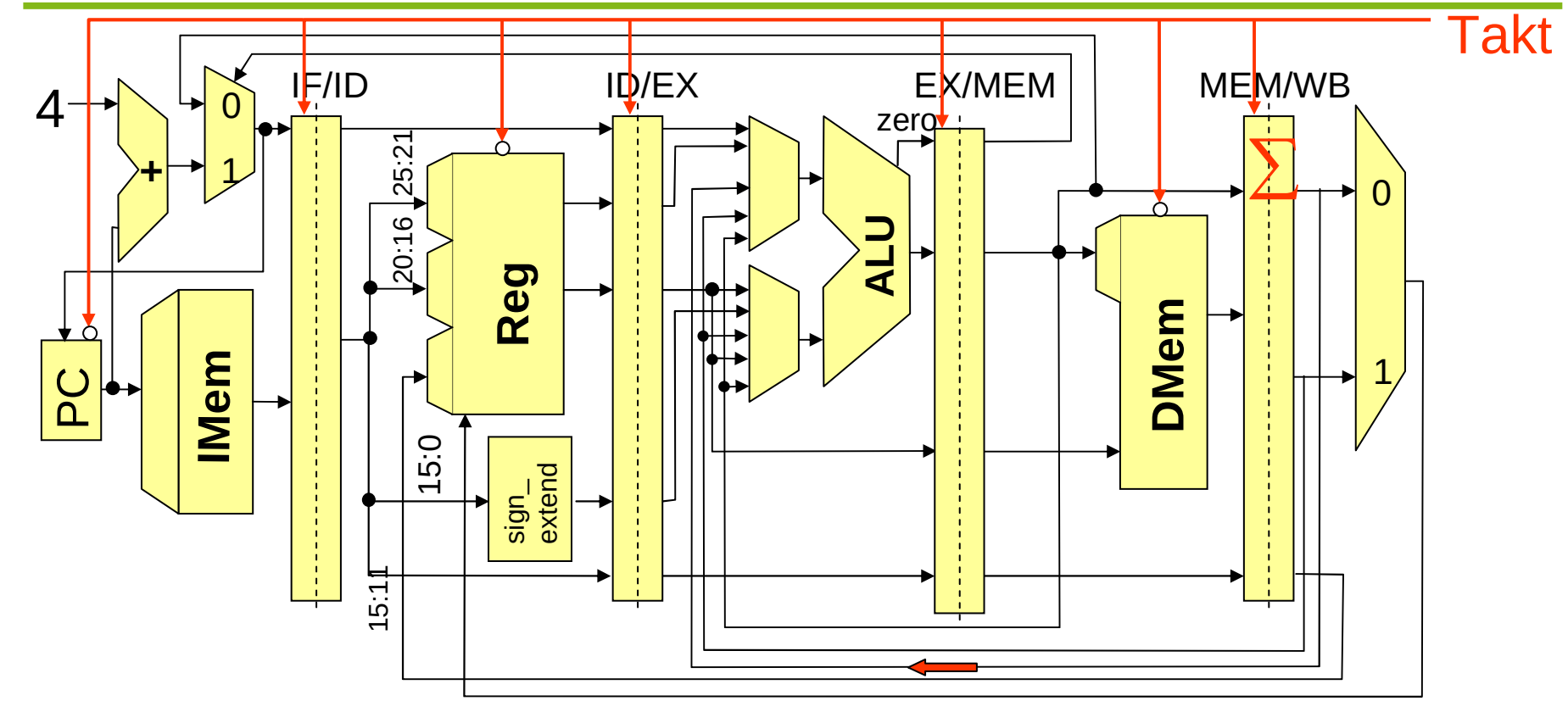
Zyklus 3				
and $6,$1,$7	**sub $4,$5,$1**	**add $1,$2,$3**		

Taktung zur Behandlung des *data hazards* bei or



Zyklus 4				
or $8,$1,$9	**and $6,$1,$7**	**sub $4,$5,$1**	**add $1,$2,$3**	

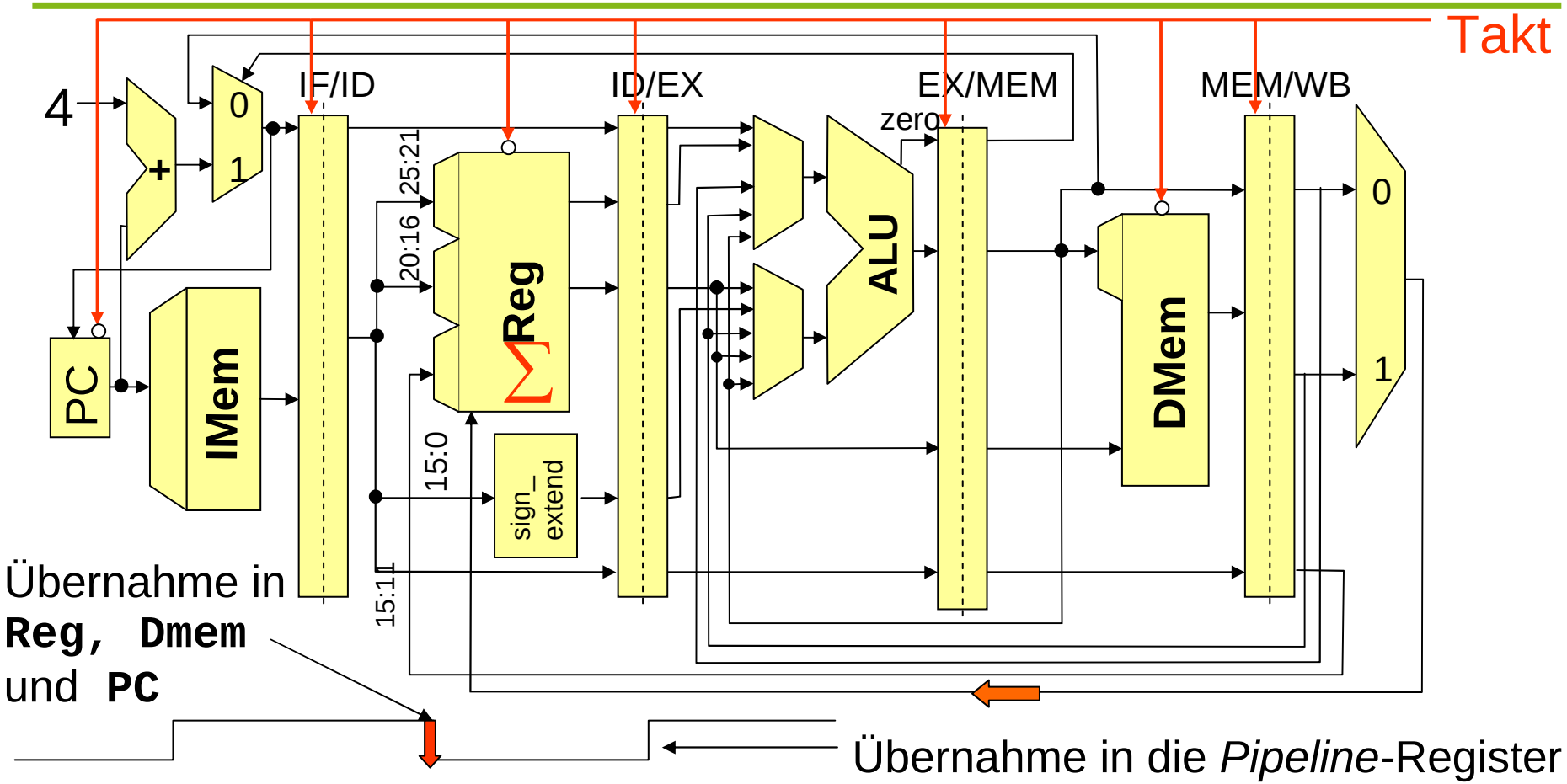
Taktung zur Behandlung des *data hazards* bei or



Übernahme in die *Pipeline*-Register

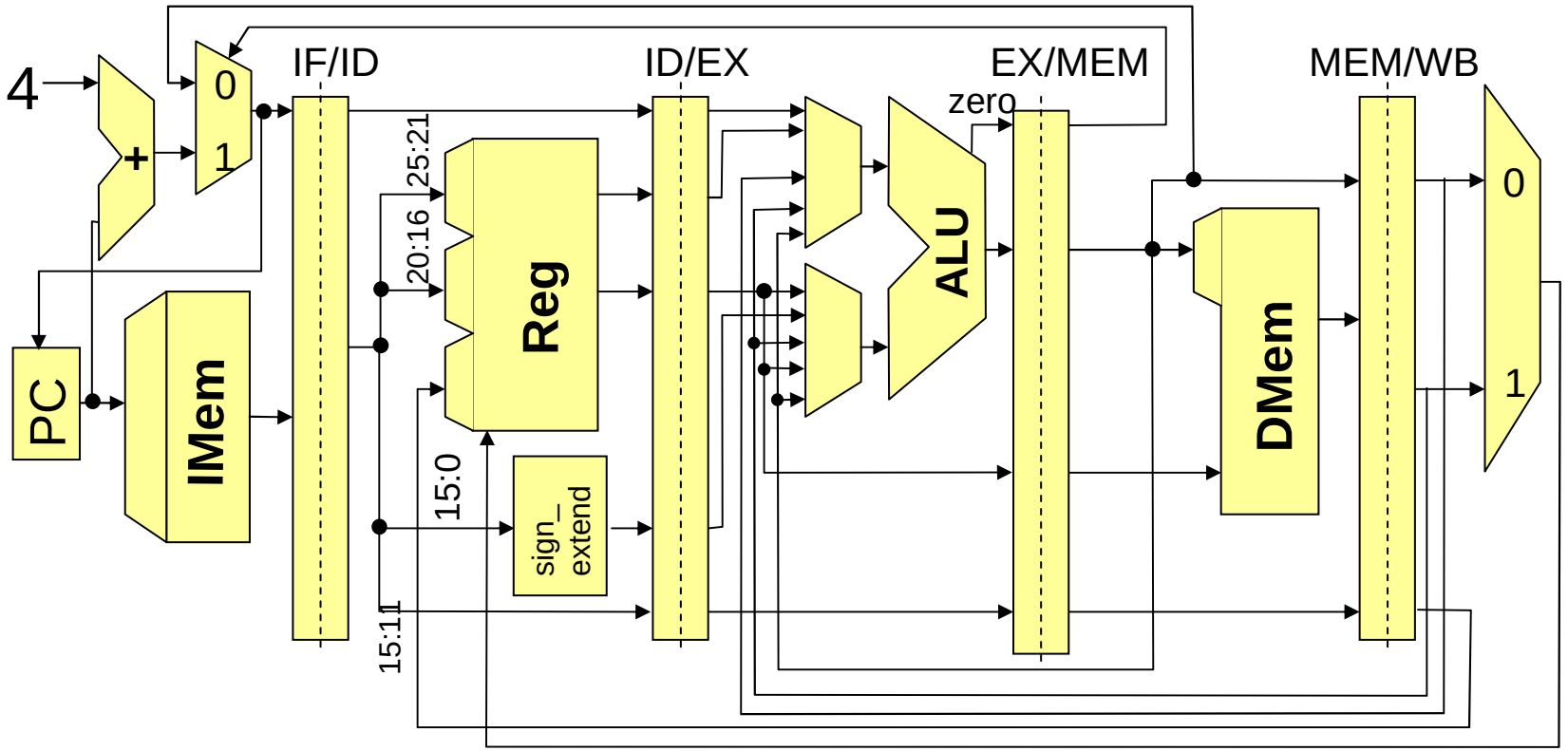
Zyklus 5				
xor $10,$1,$11	**or $8,$1,$9**	**and $6,$1,$7**	**sub $4,$5,$1**	**add $1,$2,$3**

Taktung zur Behandlung des *data hazards* bei or



Zyklus 6				
?	xor $10,$1,$11	or $8,$1,$9	and $6,$1,$7	sub $4,$5,$1

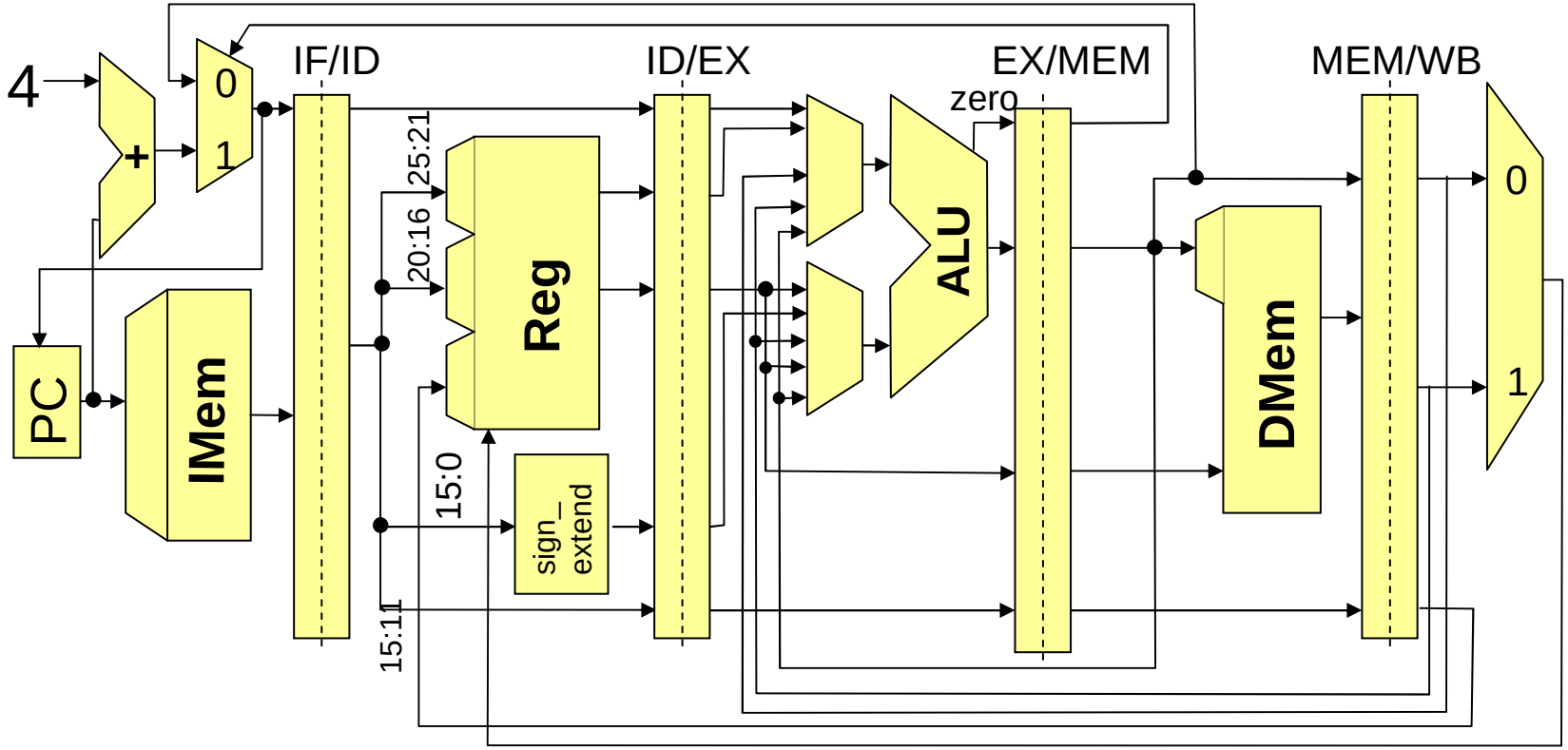
Alle *data hazards* durch Bypässe behandelbar?



Zyklus 6				
?	**xor $10,$1,$11**	**or $8,$1,$9**	**and $6,$1,$7**	**sub $4,$5,$1**

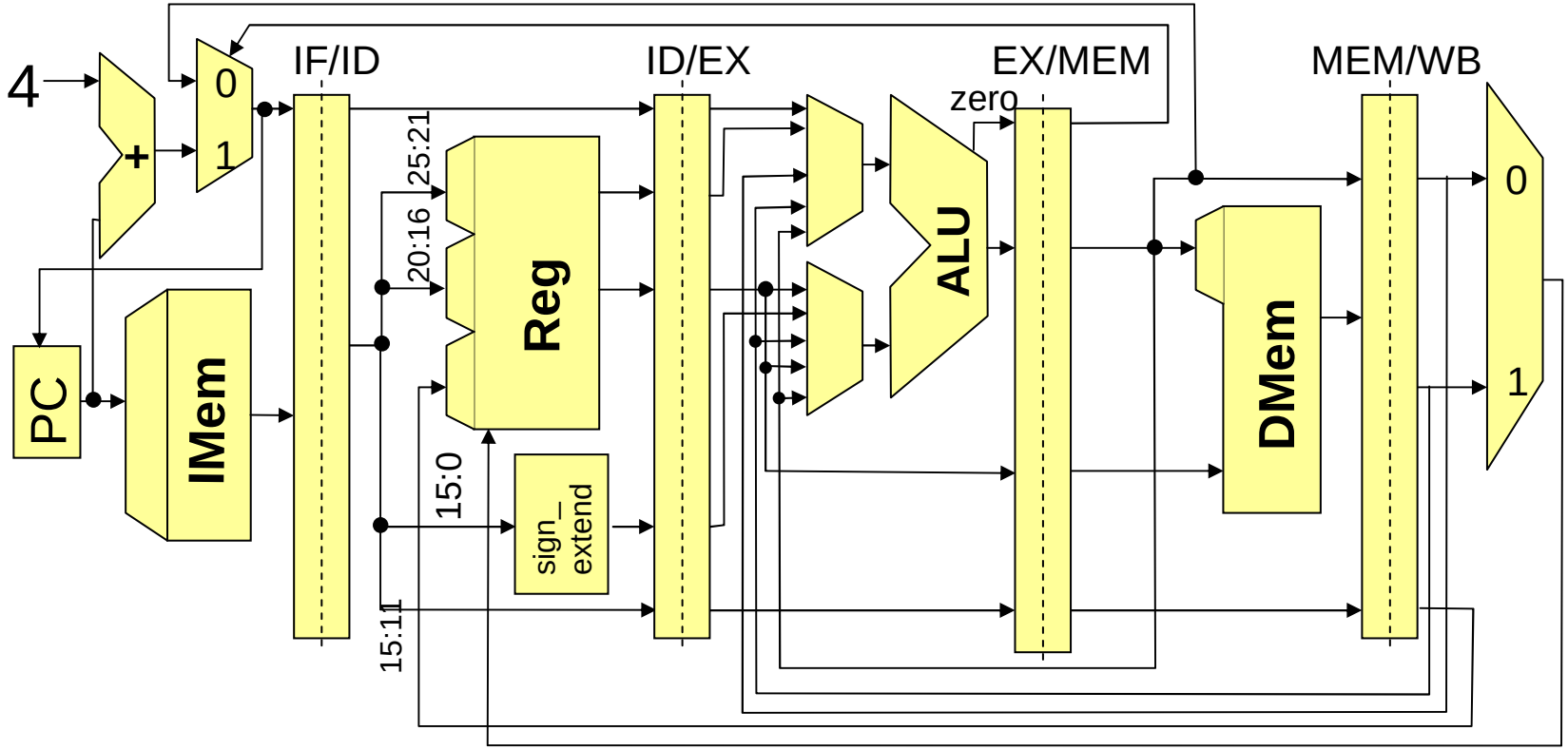
Speicherwort wird am Ende des Zyklus 4 gespeichert, steht für Differenz noch nicht bereit.

Lösung durch Anhalten des Fließbandes (*pipeline stall, hardware interlocking, bubbles*)



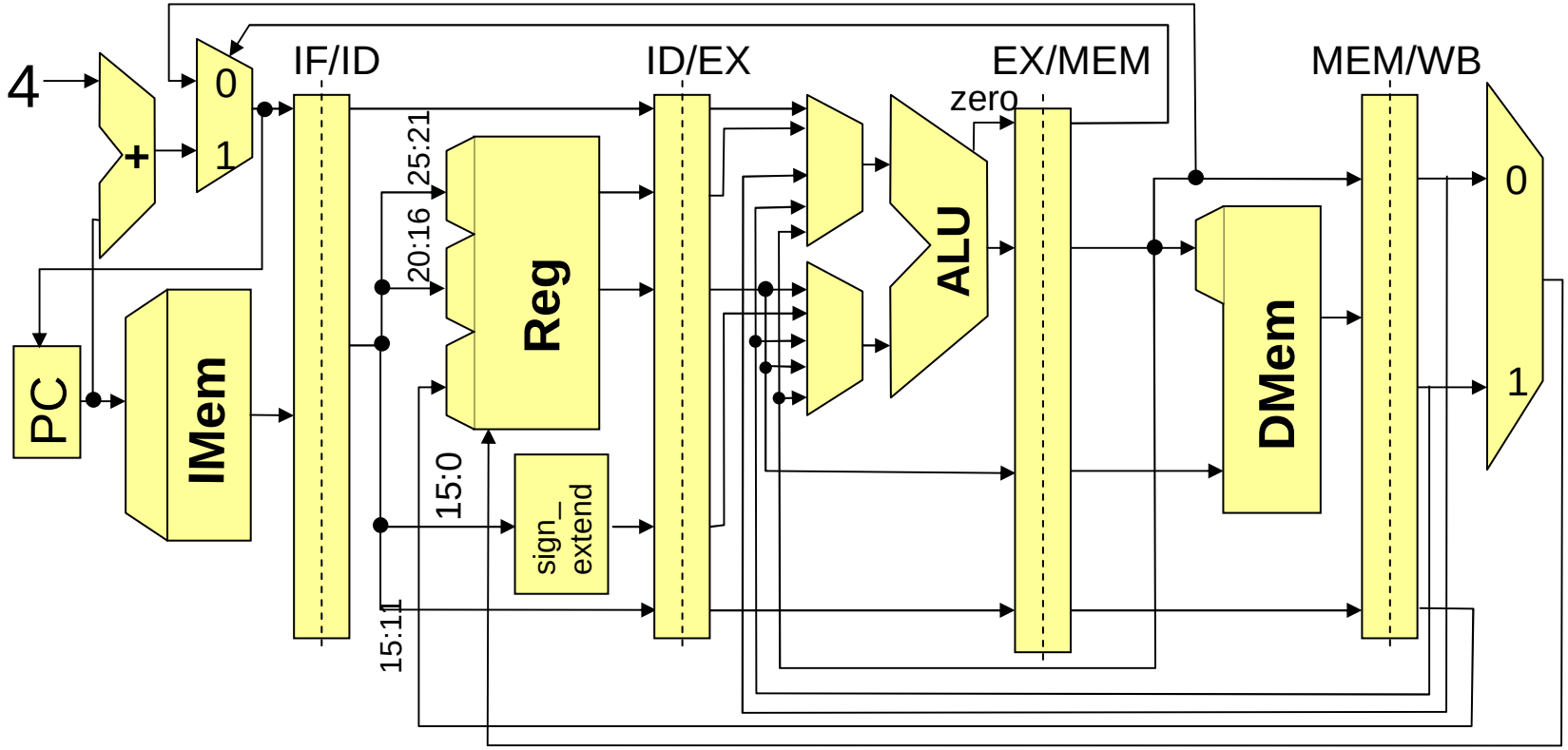
Zyklus 1				
lw $1,0($2)				

Lösung durch Anhalten des Fließbandes (*pipeline stall, hardware interlocking, bubbles*)



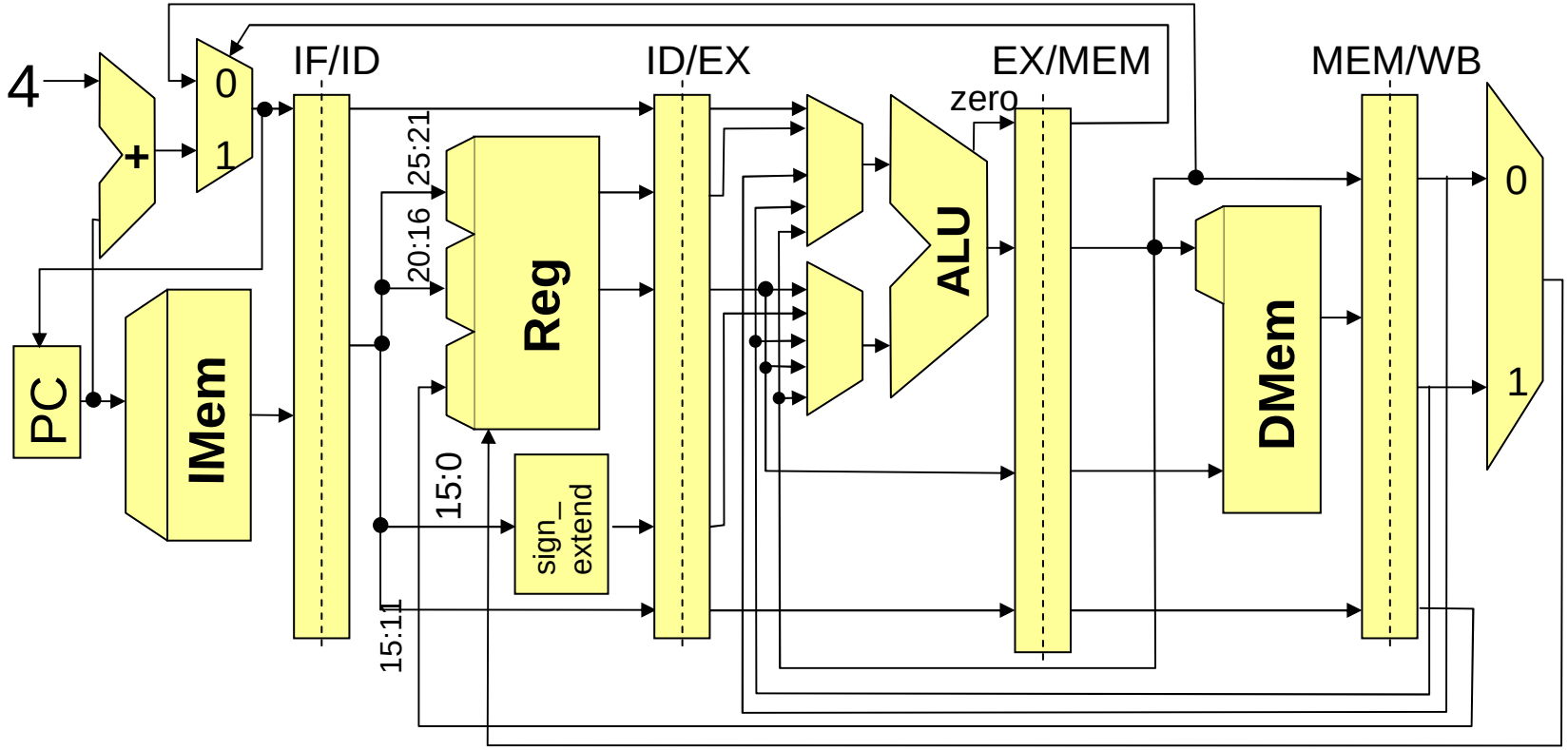
Zyklus 2				
sub $4,$5,$1	**lw $1,0($2)**			

Lösung durch Anhalten des Fließbandes (*pipeline stall, hardware interlocking, bubbles*)



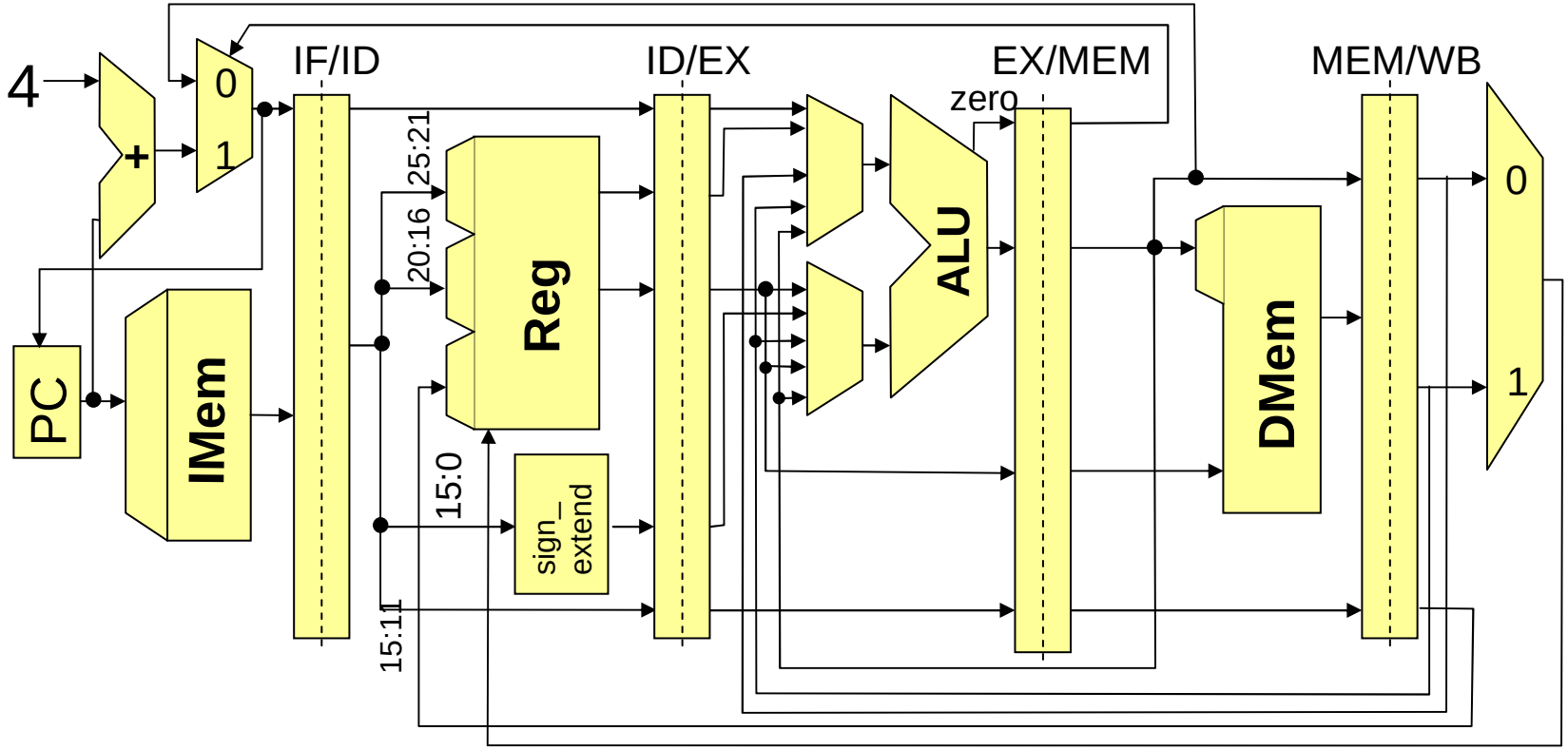
Zyklus 3				
and $6,$1,$7	**sub $4,$5,$1**	**lw $1,0($2)**		

Lösung durch Anhalten des Fließbandes (*pipeline stall, hardware interlocking, bubbles*)



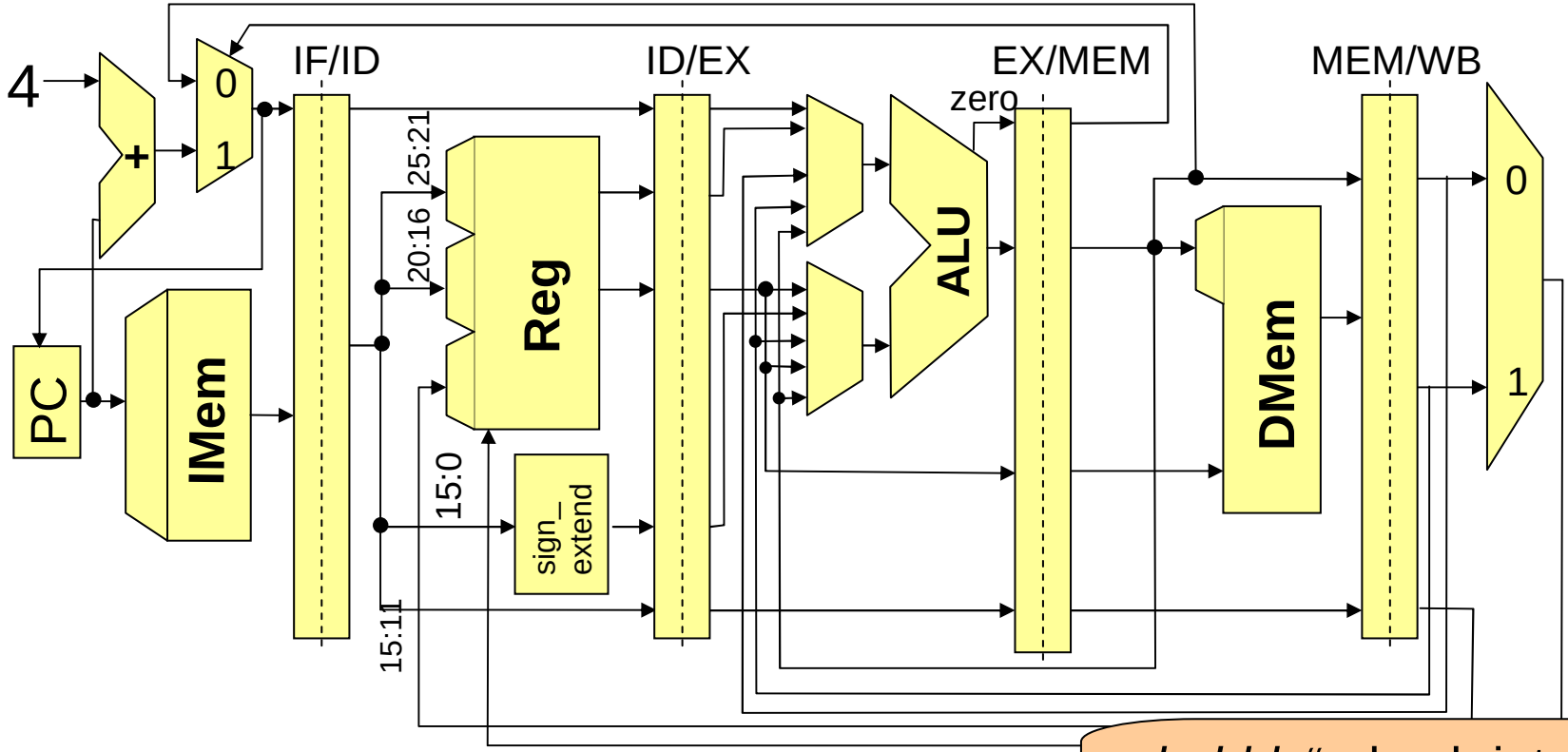
Zyklus 4				
or NOOP	and NOOP	sub NOOP	lw $1,0($2)	

Lösung durch Anhalten des Fließbandes (*pipeline stall, hardware interlocking, bubbles*)



Zyklus 5				
or $8,$1,$9	**and $6,$1,$7**	**sub $4,$5,$1**	**NOOP**	**lw $1,0($2)**

Lösung durch Anhalten des Fließbandes (*pipeline stall, hardware interlocking, bubbles*)



„*bubble*“, durch intelligente Compiler vermeiden!

Zyklus 6				
xor $10,$1,$11	or $8,$1,$9	and $6,$1,$7	sub $4,$5,$1	NOOP

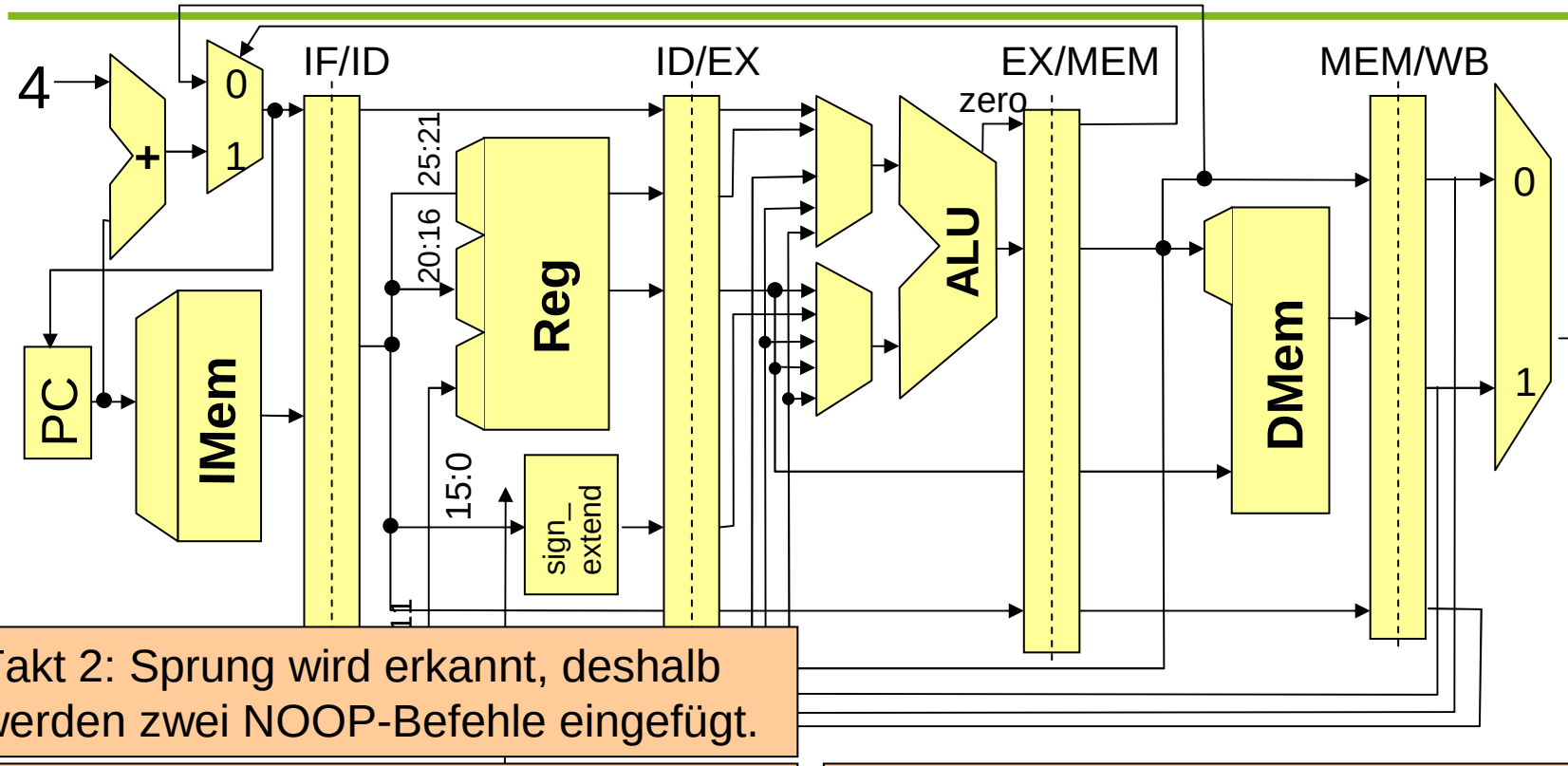
Kontrollfluss-Abhängigkeiten oder - Gefährdungen, *control hazards* (1)

Beispielprogramm:

```
   beq $12,$2,t   -- springe zur Marke t, falls Reg[12]=Reg[2]
   sub  ...
   ....
t:add ..
```

Wir versuchen zunächst, durch Einfügen von NOOPs, die intuitive Bedeutung des Programms zu realisieren...

Kontrollfluss-Abhängigkeiten oder - Gefährdungen, *control hazards* (2)



Takt 2: Sprung wird erkannt, deshalb werden zwei NOOP-Befehle eingefügt.

Takt 4: Mit fallender Flanke wird PC getaktet, mit steigender IF/ID-Register.

Takt 3: ALU müsste sowohl Vergleich wie auch das Sprungziel ausrechnen können

Zyklus 6				
...	**...**	**sub oder add**	**NOOP**	**NOOP**

Kontrollfluss-Abhängigkeiten oder - Gefährdungen, *control hazards* (3)

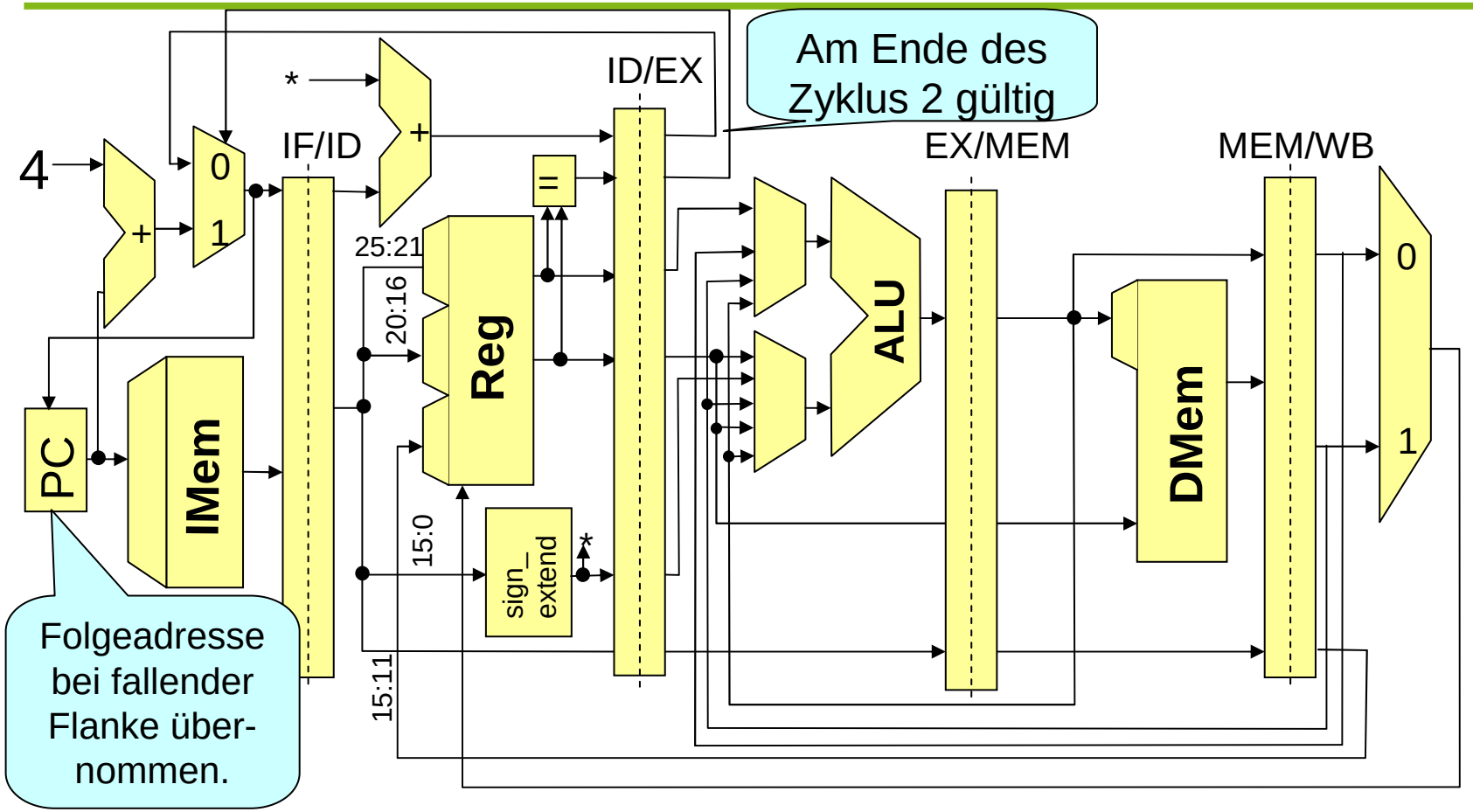
Probleme beim gezeigten Ansatz:

- Leistungsverlust durch 2 NOOPs (*branch delay penalty)*.
- ALU/Multiplexer in der gezeigten Form nicht ausreichend, um Test und Sprungzielberechnung in einem Takt auszuführen.

Lösungsansatz:

- Gleichheit der Register wird schon in der *instruction decode*-Stufe geprüft.
- Sprungziel wird in separatem Adressaddierer ebenfalls bereits in der *instruction decode*-Stufe berechnet.
- Sofern weiterhin noch Verzögerungen auftreten:
  - nächsten Befehl einfach ausführen (*delayed branch*).
  - oder weiterhin NOOP(s) einfügen (*stall*).

Reduktion der *branch delay penalty; delayed branch*



Zyklus 6				
...	...		**add**	**sub**

Delayed Branches, verzögerte Sprünge

Beim gezeigten Beispiel wird der auf den Sprungbefehl folgende Befehl immer noch ausgeführt.

```
        beq $12,$2,t
        sub  ...   #  wird immer noch ausgeführt
        ....
t:      add ..
```

„It‘s not a bug, it‘s a feature“

Einen Platz für die Aufnahme eines solchen Befehls nennt man ***delay slot,*** die Sprünge ***delayed branches.***
Manche Maschinen haben mehrere *delay slots*.
Delay slots sollten von Übersetzern mit nützlichen Befehlen gefüllt werden. Nur notfalls sollte es ein NOOP sein.
Die MIPS-Maschine hat ein *delay slot*, welches aber vom Assembler verdeckt wird.

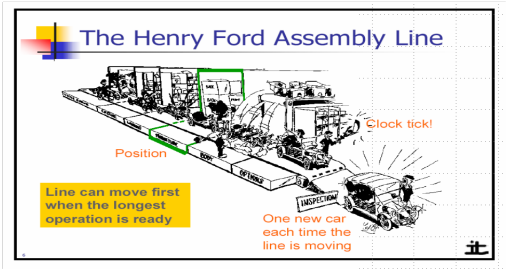
Typen von Fließband-Gefährdungen (*hazards*)

- **Strukturelle Abhängigkeiten/Gefährdungen (*structural hazards*)**
- **Datenfluß- Abhängigkeiten/Gefährdungen (*data hazards*)**
  - **aufgrund von Datenabhängigkeiten (RAW)**
    ☞ *forwarding, pipeline stalls*
  - **aufgrund von Antidatenabhängigkeiten (WAR)**
    (erst bei komplizierteren Systemen wichtig)
  - **aufgrund von Ausgabeabhängigkeiten (WAW)**
    (erst bei komplizierteren Systemen wichtig)
- **Kontrollfluß-Abhängigkeiten/Gefährdungen (*control hazards*)**

☞ *delayed branches, pipeline stalls,* spekulative Ausführung, Sprungvorhersage

Zusammenfassung

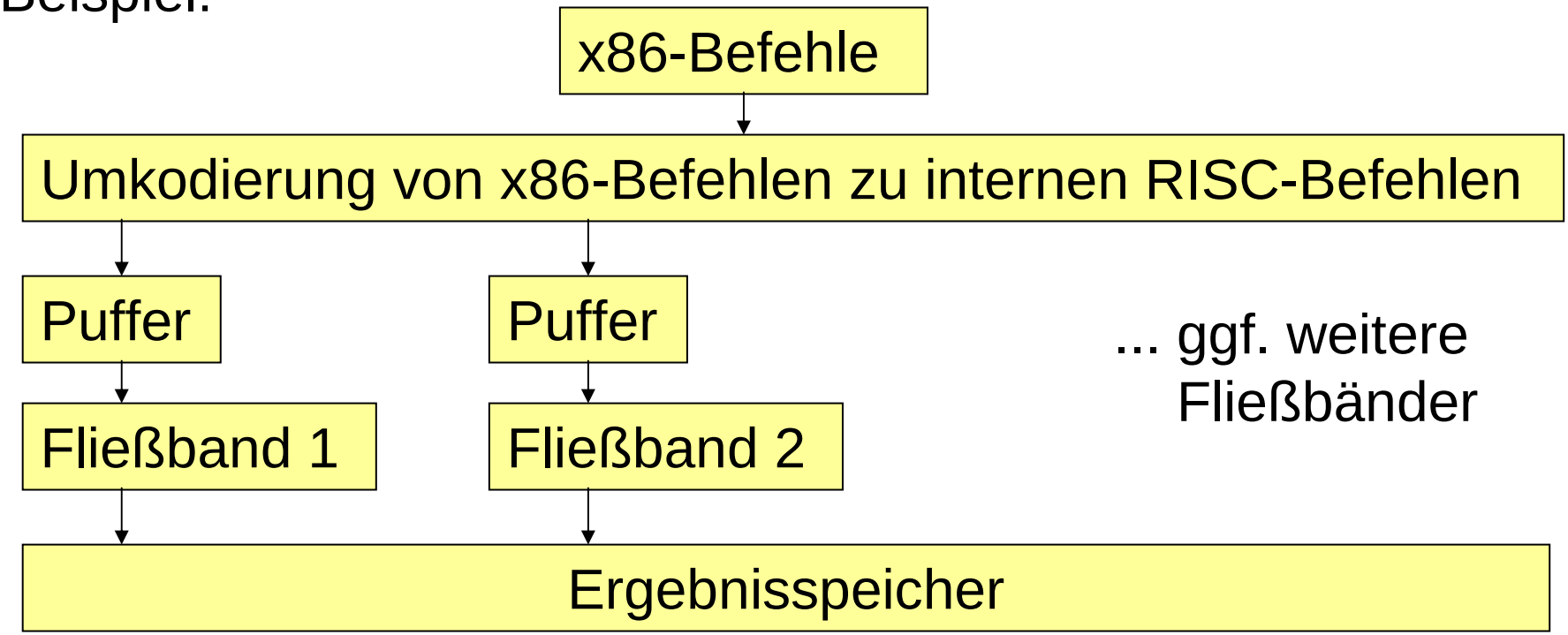
- Die Fließbandverarbeitung (engl. *pipelining*) ermöglicht es, in jedem Takt die Bearbeitung eines Befehls abzuschließen, selbst wenn die Bearbeitung eines Befehls ≥ 1Takte dauert.
- Bei mehreren Fließbändern ☞ pro Takt können mehrere Befehle beendet werden.
- 3 Typen von Gefährdungen des Fließbandbetriebs:
  - *resource hazards*
  - *data hazards (RAW, WAR, WAW)*
  - *control hazards*
- Gegenmaßnahmen:
  - *pipeline stall*
  - *forwarding/bypassing*
  - *branch prediction, delayed branches,*
  - *out-of-order execution, dynamic scheduling*



www.it.lth.se/courses/dsi/material/Lectures/Lecture6.pdf

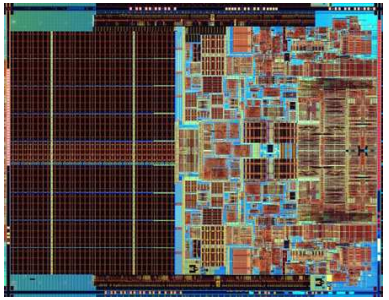
Interne Struktur von Pentium-Prozessoren

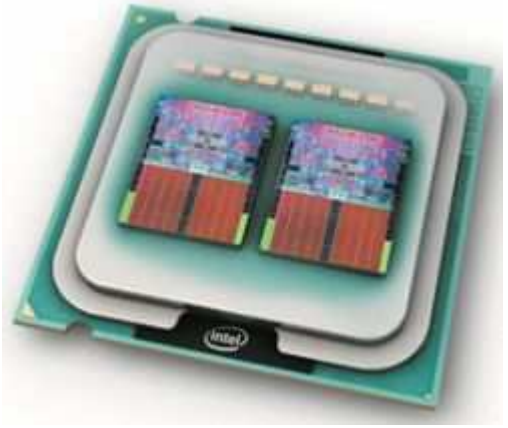
Fließbandverarbeitung bringt Performancegewinn.
Fließbandverarbeitung nur bei RISC-Befehlssätzen einigermaßen überschaubar.
Interne Umkodierung alter CISC-Befehle in RISC-Befehle.
Beispiel:



Einige Eigenschaften 2007 aktueller Prozessoren

Beispiel: Intel® Core™ 2 Extreme Quad-Core QX6000:



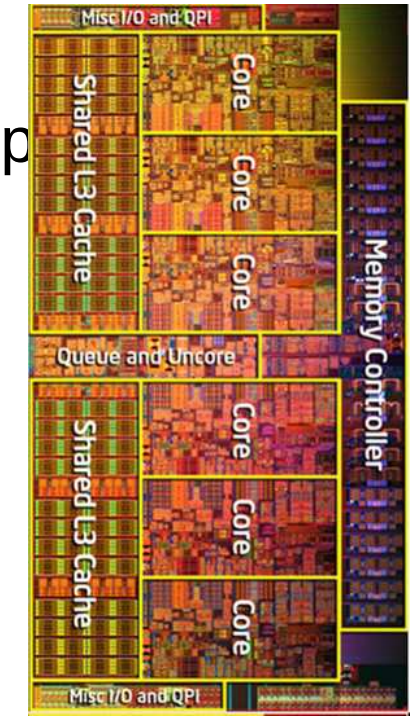


- 4 Prozessoren auf einem Chip
- **Jeder Prozessor kann pro Takt bis zu 4 Befehle beenden**
- **Befehle können sich gegenseitig überholen** (*dynamic scheduling*, *out-of-order execution*)
- Sprungvorhersage
- 64-bit und 32-bit Operationen
- Bis zu 3 GHz ext. Takt
- Verlangt thermischen Entwurf für 130 W Leistungsaufnahme
- Stromaufnahme bis zu **125 A**
- Spannungsversorgung 0,85-1,6 V je nach Anforderung
- 775 Anschlüsse, davon ~2/3 für die Spannungsversorgung

[www.intel.com] [http://media.schottenland.de/pi/IntelCore2ExtremeQx6700.jpg]

Einige Eigenschaften 2011 aktueller Prozessoren

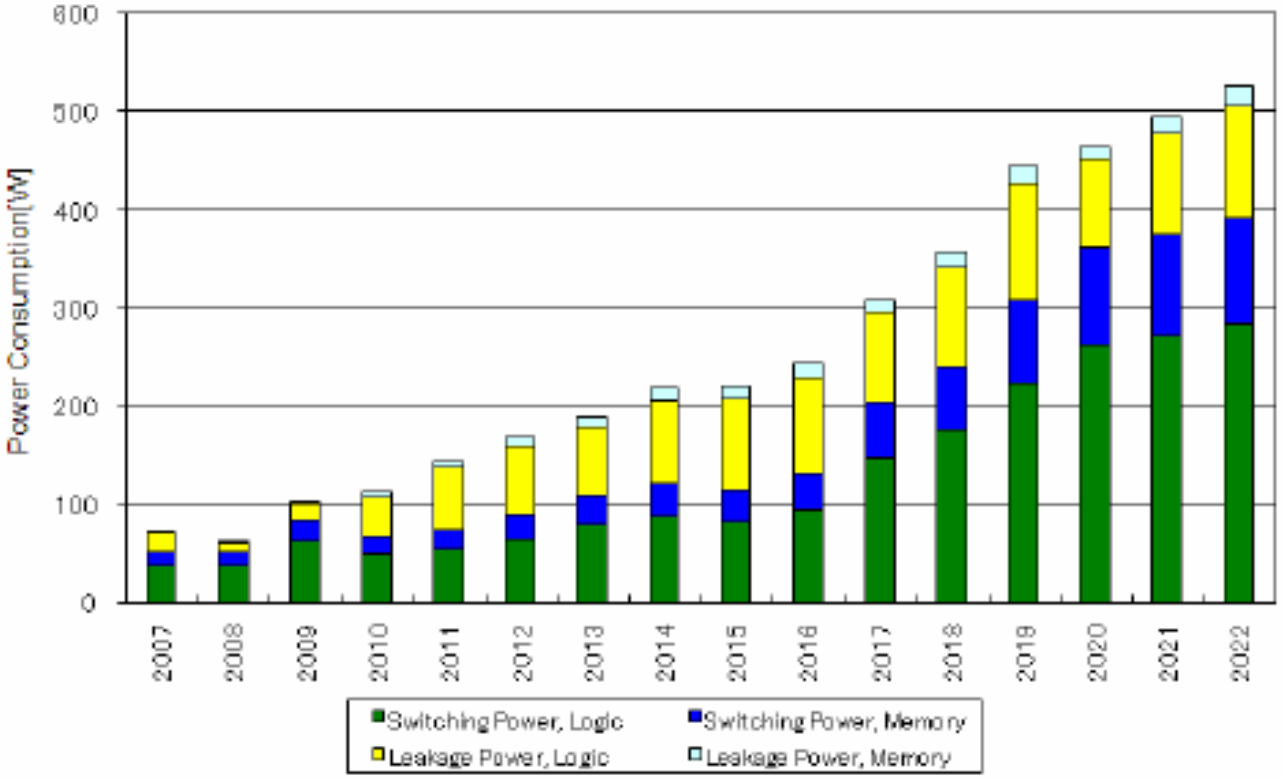
Beispiel: Intel® Core™ i7-980X Processor Extreme Edition:



- 6 Prozessoren auf einem Chip
- Hyperthreading: 2 threads/Prozessor überlapp
- Befehle können sich gegenseitig überholen
- Fertigung im 32 nm Prozess
- 1,17 x 10^9 Transistoren
- 64-bit und 32-bit Operationen
- 3,33 GHz (Turbo 3,6 GHz) ext. Takt
- Maximale Leistungsaufnahme (TDP): 130 W
- Stromaufnahme bis zu **181,1 A**
- Spannungsversorgung 0,8-1,375 V je nach Anforderung
- 1366 Anschlüsse, incl. 435 für die Spannungsversorgung

Vorhersage der Entwicklung gemäß ITRS-Roadmap

ITRS= *International Technology Roadmap for Semiconductors*:
Vereinigung der Halbleiterhersteller mit dem Zwecke, Ziele und Engpässe zu definieren (siehe http://public.**itrs**.net)
Beispiel: Leistungsaufnahme von Systemen gemäß Update 2008:

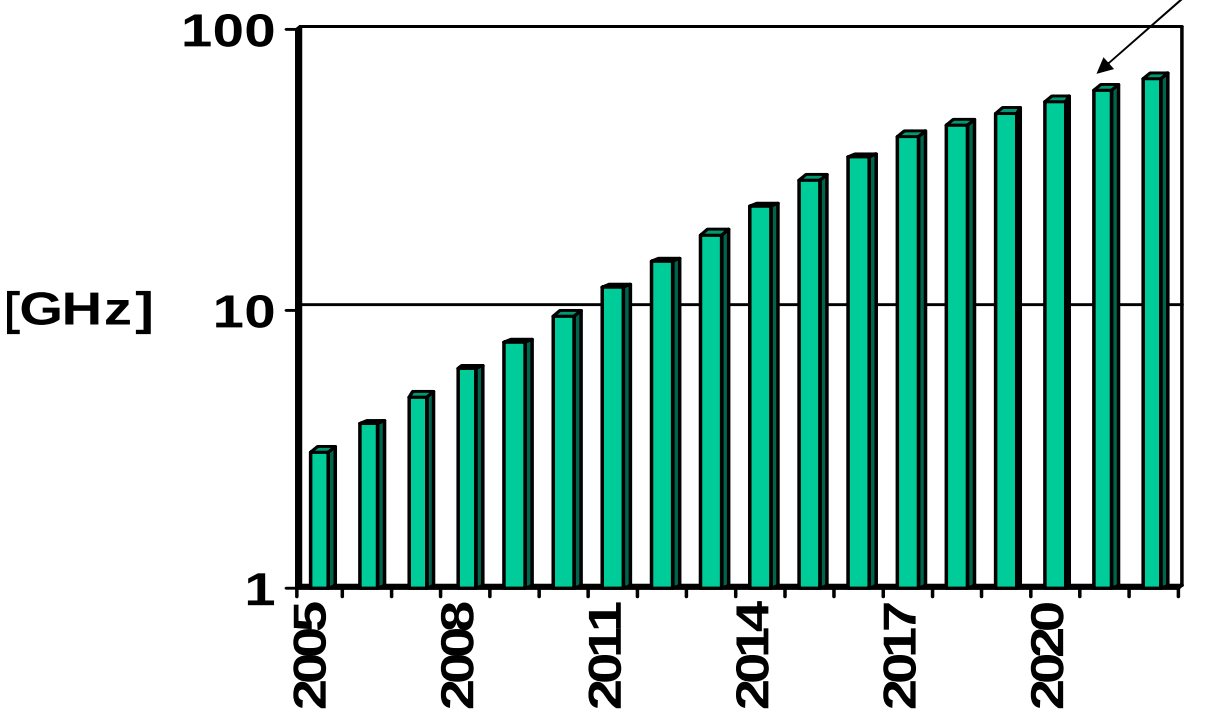


Fraglich, ob weitere Leistungs-steigerung akzeptiert wird.

[ITRS Update 2008]

Vorhersage der Taktfrequenzen

Takt *Chip-Board*

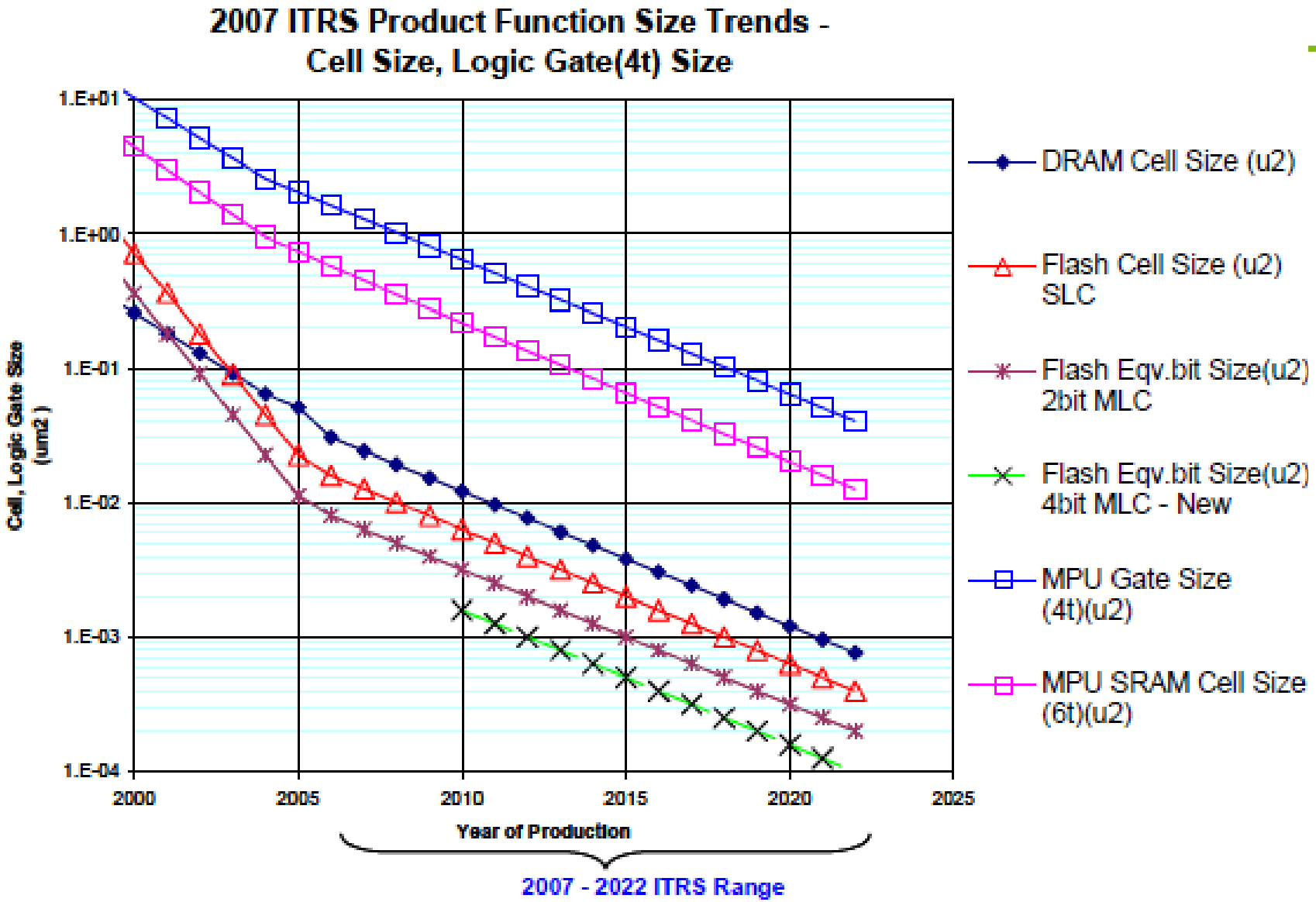


Korrektur nach unten gegenüber ITRS 2006,

notwendige Technologie ab 2018 unbekannt.

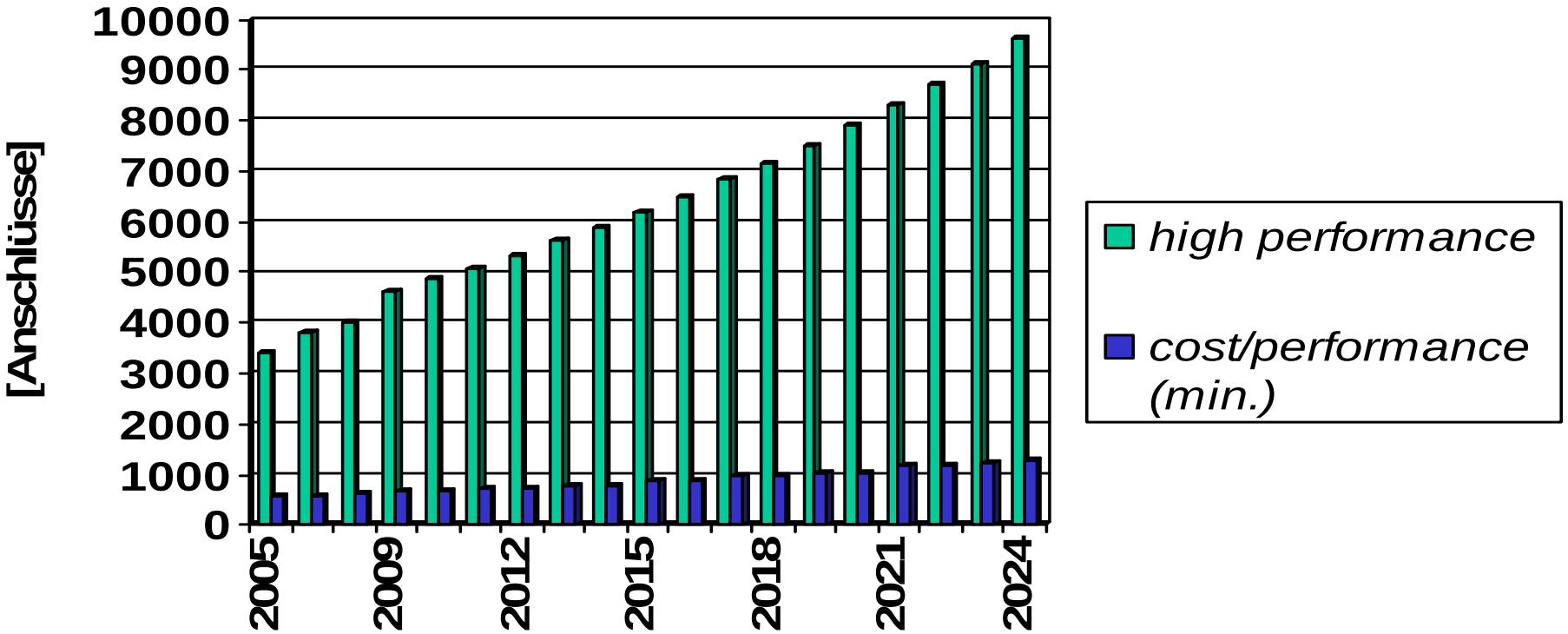
[ITRS Roadmap 2007]

Vorhersage der Zellgrößen



[ITRS Update 2008]

Vorhersage der Anzahl der Anschlüsse



[ITRS Roadmap 2009]

Vorhersage der Anzahl der Funktionen pro Chip

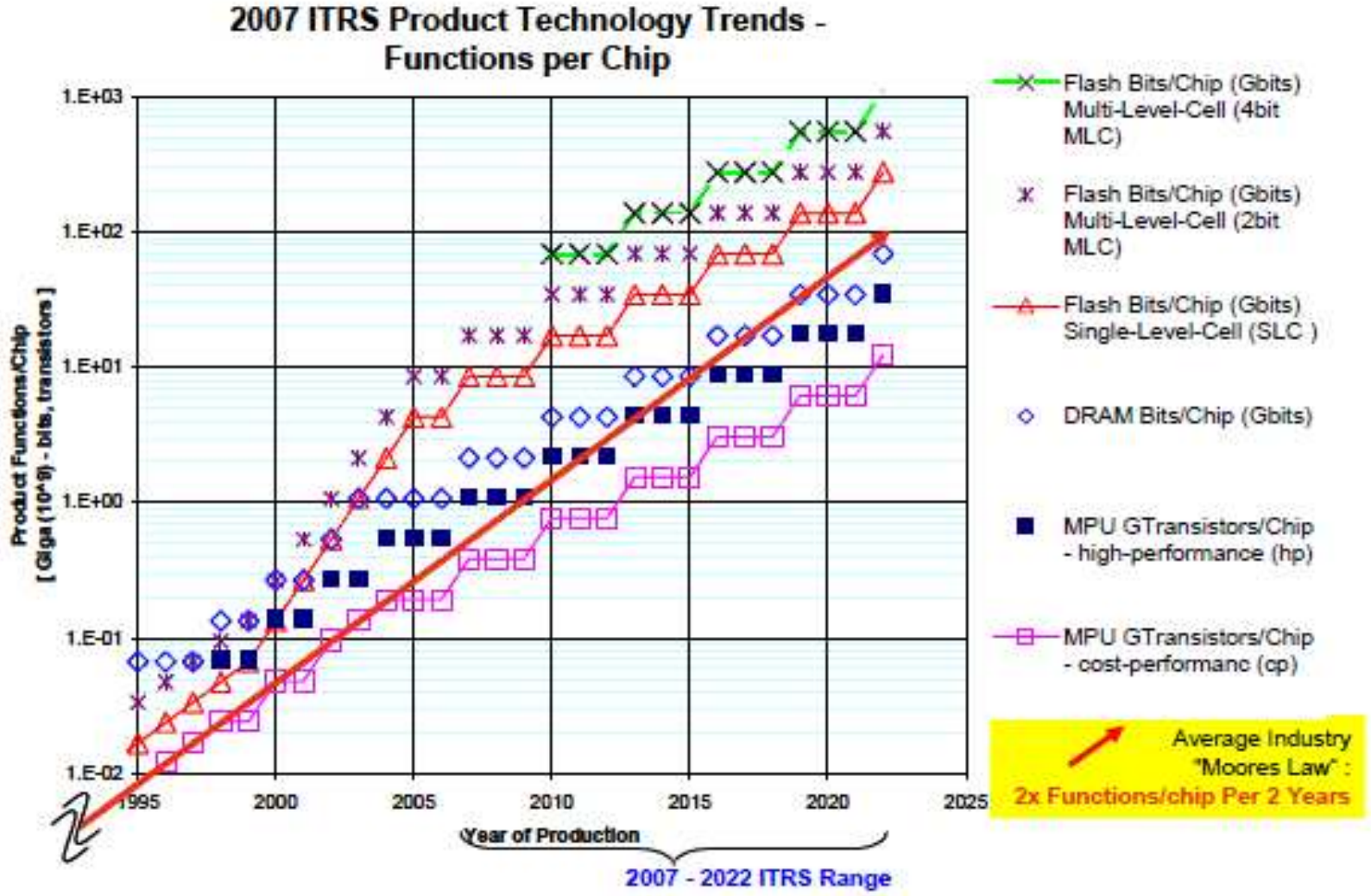


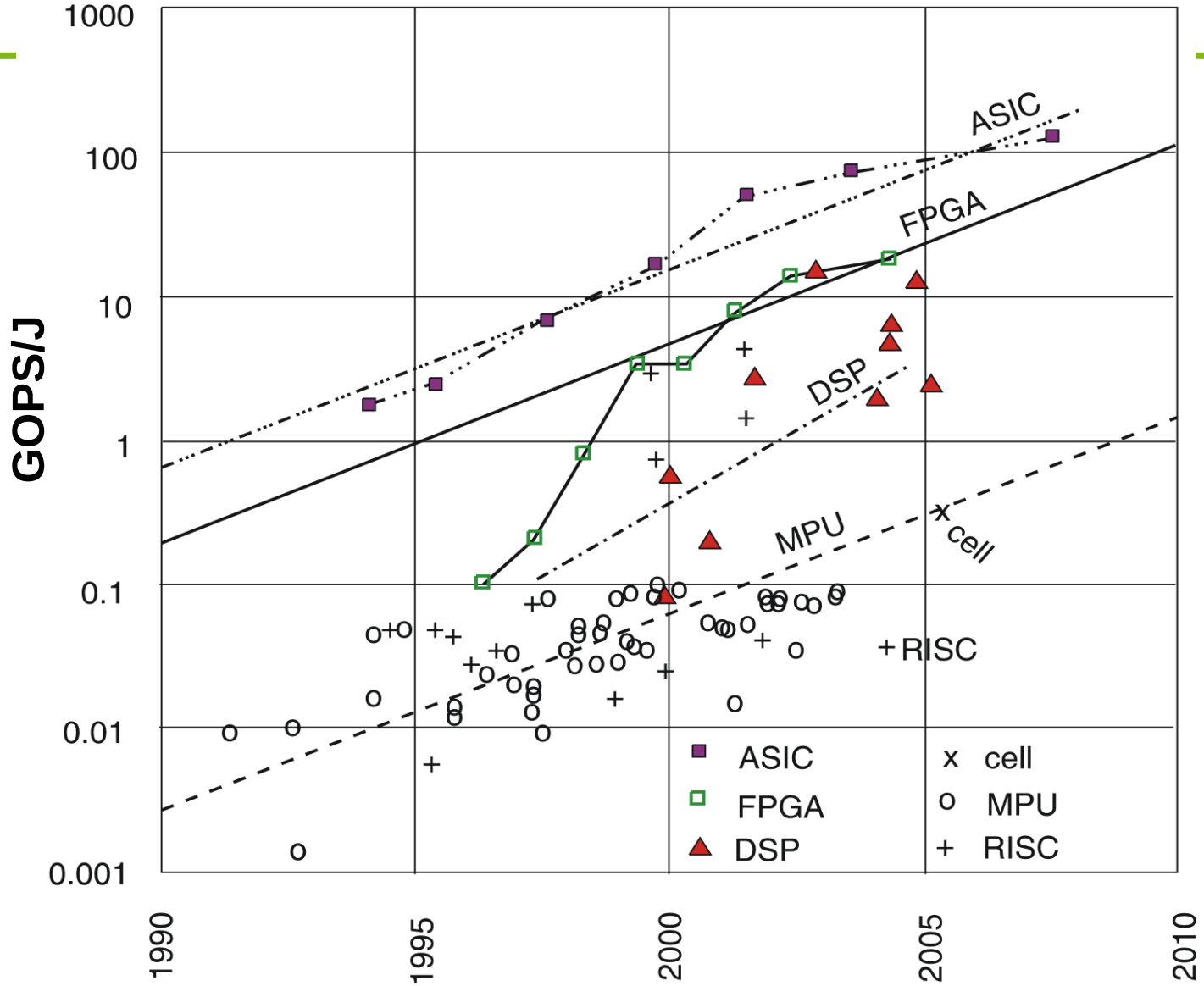
Figure ORTC2 ITRS Product Function Size Trends: MPU Logic Gate Size (4-transistor); Memory Cell Size [SRAM (6-transistor); Flash (SLC and MLC), and DRAM (transistor + capacitor)]--Updated

[ITRS Update 2008]

Alternative Architekturen

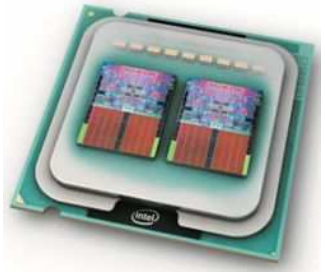
- **DSP**: Digitale Signalprozessoren; Prozessoren, die auf die Verarbeitung digitaler Signale (Sprache, Video) optimiert sind
- **VLIW**: *very long instruction word* –Prozessoren; Prozessoren, die mit breiten Paketen von Befehlen mehrere Befehle gleichzeitig starten können
- **ASIP**: *application specific instruction set processors;* Prozessoren, die für bestimmte Anwendungen (z.B. MPEG) optimiert sind
- **FPGA:** *field programmable gate array*; Schaltung, deren Verhalten man durch Programmierung verändern kann.
- **ASIC**: *application specific integrated circuit*; speziell für eine Anwendung entwickelter integrierter Schaltkreis

Trend hinsichtlich Energieeffizienz



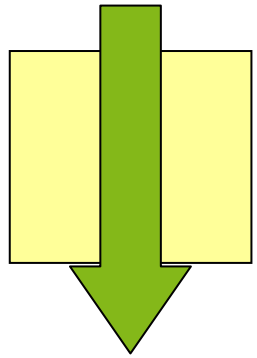
Trend zu Multiprozessor-Systemen

- Grenzen der effizienten Realisierbarkeit von Einzelprozessoren erreicht:
  - Höhere Taktraten nur schwer zu erreichen
  - **Höhere Taktraten nicht mehr energieeffizient (Kriterium Watt/Millionen Operationen)**
  - Sprungvorhersage wird immer komplizierter
  - …
- Bei Multiprozessor-Systemen werden mehrere Prozessoren zusammengeschaltet:
  - Gleiche Multiprozessorsysteme:
    ☞ homogene Multiprozessoren
  - Unterschiedliche Multiprozessorsysteme:
    ☞ heterogene Multiprozessoren



Klassifikation von Multiprozessorsystemen nach Daten- und Befehlsströmen [Flynn]

		Befehlsströme	
		1	>1
Daten-ströme	1	SISD	MISD
	>1	SIMD	MIMD

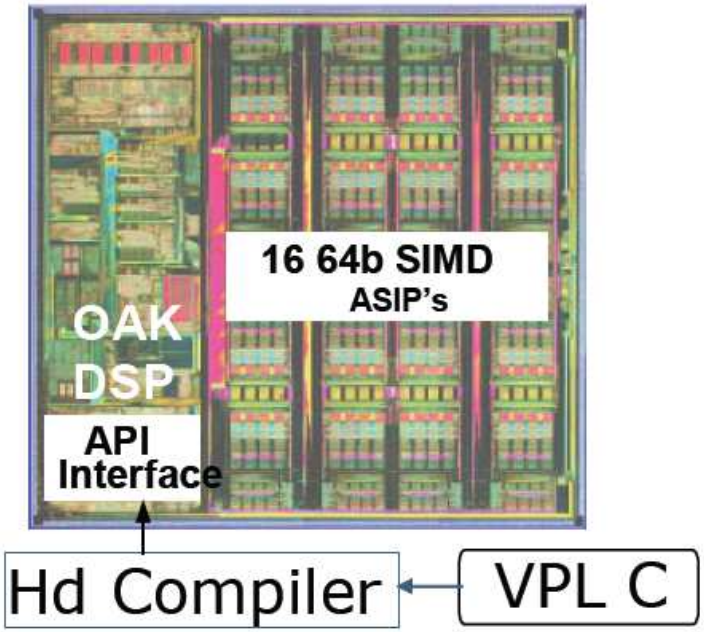


SISD	Bislang besprochene Einzelrechner
MIMD	Netze aus Einzelrechnern; sehr flexibel
SIMD	Ein Befehlsstrom für unterschiedliche Daten; identische Befehle bilden starke Einschränkung
MISD	Mehrere Befehle für ein Datum: Kann als Fließband von Befehlen auf demselben Datenstrom ausgelegt werden. Ist etwas künstlich.

Klassifikation hat nur begrenzte Aussagekraft; keine bessere vorhanden.

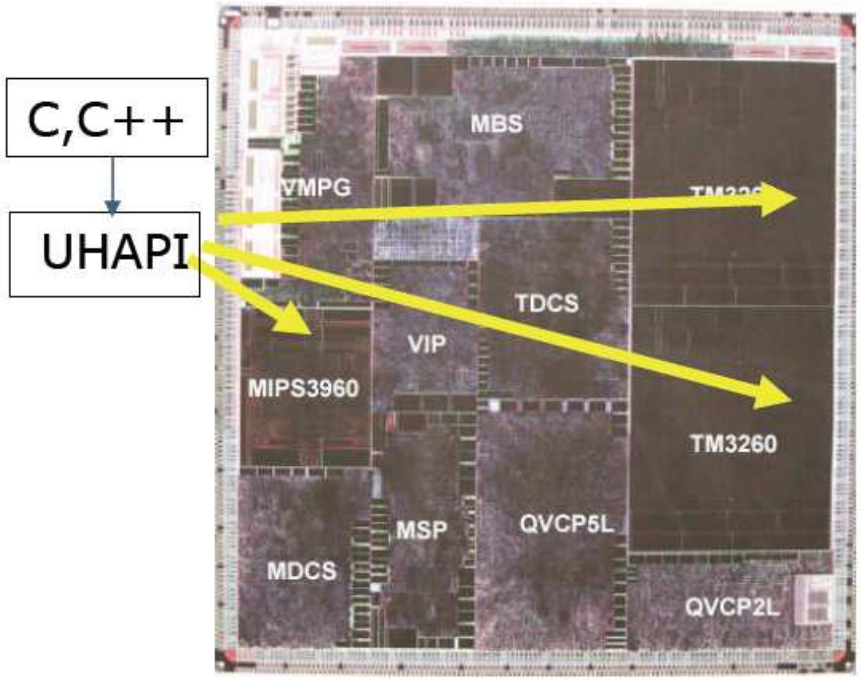
Multiprocessor Systems On A Chip (MPSoCs) - Beispiele -

VIP for car mirrors
Infineon



200MHz , 0.76 Watt
100Gops @ 8b
25Gops @ 32b

Nexperia Digital Video Platform
NXP



1 MIPS, 2 Trimedia
60 coproc, 250 RAM's
266MHz, 1.5 watt 100 Gops

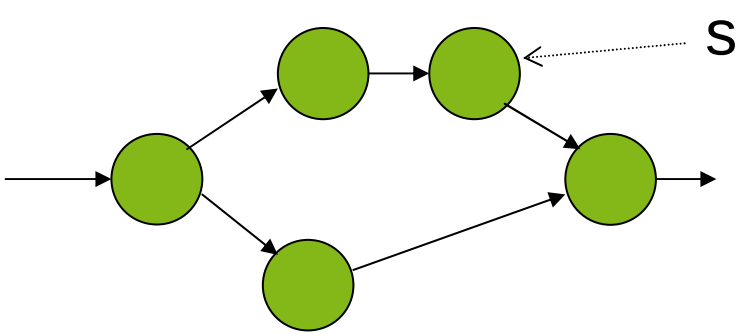
~ ½ IPE

Programmierung von Mehrprozessor-Systemen

- Übliche imperative Sprachen (C, C++, Java, …): Abstraktion der seq. Ausführung in von-Neumann Maschine.
  - ☞¬ für Programmierung von ≥ 1 Maschinen konzipiert.
  - ☞Existierende Anwendungen ¬ für Programmierung von parallelen Maschinen konzipiert.
- Kann man aus existierenden Anwendungen automatisch Parallelität extrahieren?
  - Auf der Basis riesigen Aufwandes begrenzte Erfolge im „*high performance computing*“ (HPC; Simulationen in Physik, Chemie usw.)
  - Für allgemeine Anwendung weitgehend ein Fehlschlag

Lösungsansätze

- Alternative Berechungsmodelle
  - Funktionale Sprachen
  - Datenflusssprachen
  - Signalflussgraphen
  - Task-Graphen als Anwendungsmodellierung

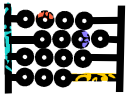


Bislang keine allgemein nutzbare Lösung,
Prozessorhersteller setzen große Summen auf MP-Technologie

Zusammenfassung

- Vorhergesagt wird ein weiteres Skalieren der Prozessoren
- Allerdings ist das Ende des Skalierens absehbar
- Unklare Situation hinsichtlich der Richtung
  - Geeignetes Berechungsmodell?
  - Geeignete Architekturen?
  - Geeignete Sprachen?

☞ Mögliches Ende des „Paradieses“, in dem immer komplexere Softwarearchitekturen durch immer leistungsfähigere Hardware möglich werden.



* http://www.markitoxs.net/blog/?p=465

Die Vertreibung aus dem Paradies / Julius Schnorr von Carolsfeld (1794-1872), 1860