Rechnerstrukturen, Teil 2

Vorlesung 4 SWS WS 17/18

2.3 Register-Transfer-Strukturen

Prof. Dr. Jian-Jia Chen
Fakultät für Informatik – Technische Universität Dortmund
jian-jia.chen@cs.uni-dortmund.de
http://ls12-www.cs.tu-dortmund.de

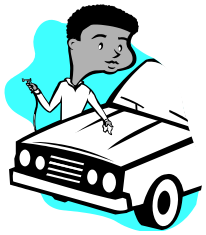
Kontext

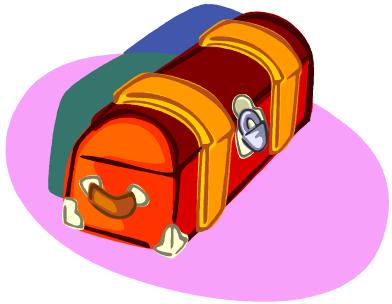
Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information
[Gesellschaft für Informatik]

Externe Architektur → interne Architektur

Bislang:

Sicht des Programmierers auf den Prozessor:
Befehlsschnittstelle,
externe Architektur

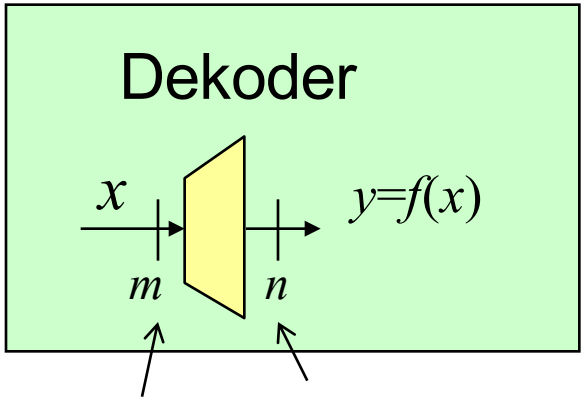




Jetzt:

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

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

z.B.

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

Beispiel 2: Prioritätsencoder (m>n)

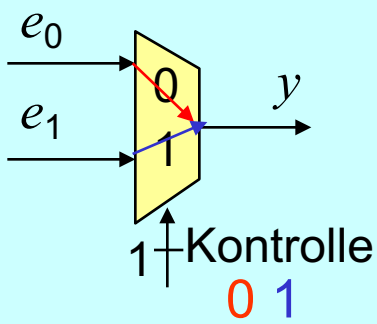
$$nat(f(x)) = \begin{cases} i, \text{ wenn } i \text{ der größte Index ist, für den } x_i = \text{'1' ist} \\ undefiniert \text{ für } x = \text{"0...00"} \end{cases}$$

z.B.: $y = \text{"101"}, nat(y) = 5$ für $x = \text{"00}\underset{5}{1}01\underset{0}{1}1\text{"}$

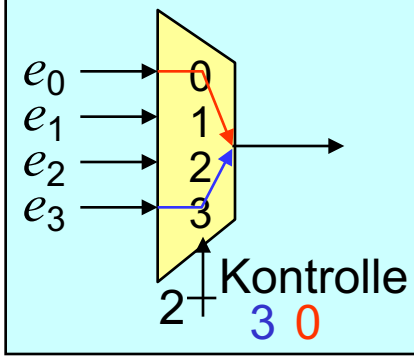
$nat(a)=\Sigma a_i 2^i$: (Abbildung Bitvektor →natürliche Zahl)

Komponenten von RT-Strukturen: Multiplexer

z.B. 2 auf 1 Multiplexer



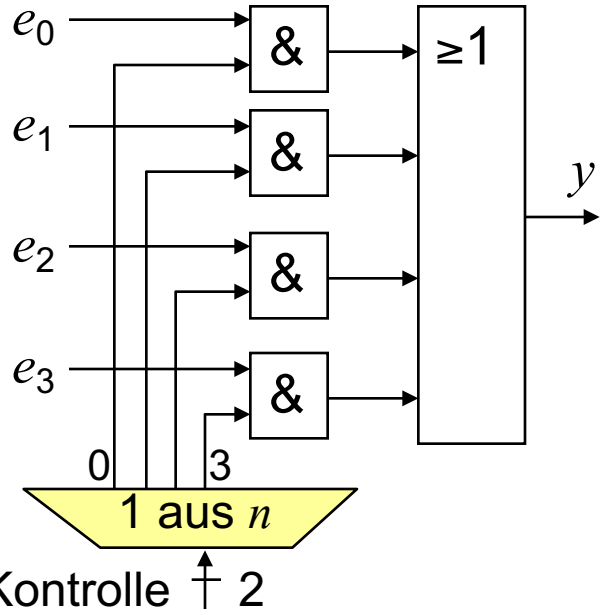
z.B. 4 auf 1 Multiplexer



$y = e_i$ für nat(Kontrolle)=i

e_i, y: einzelne Bits oder Bitvektoren

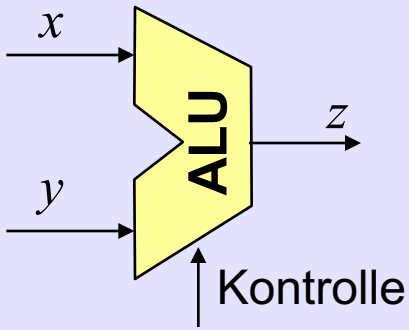
Mögliche Realisierung (für einzelnes Bit)



Komponenten von RT-Strukturen: ALUs

Addierer, arithmetisch/
logische Einheiten (ALUs)

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



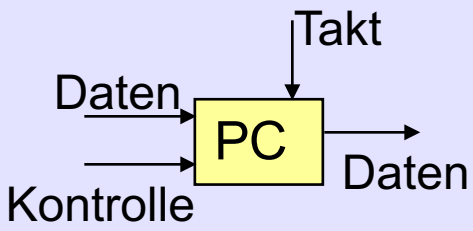
Beispiel

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

x, y, z, Kontrolle: Bitvektoren

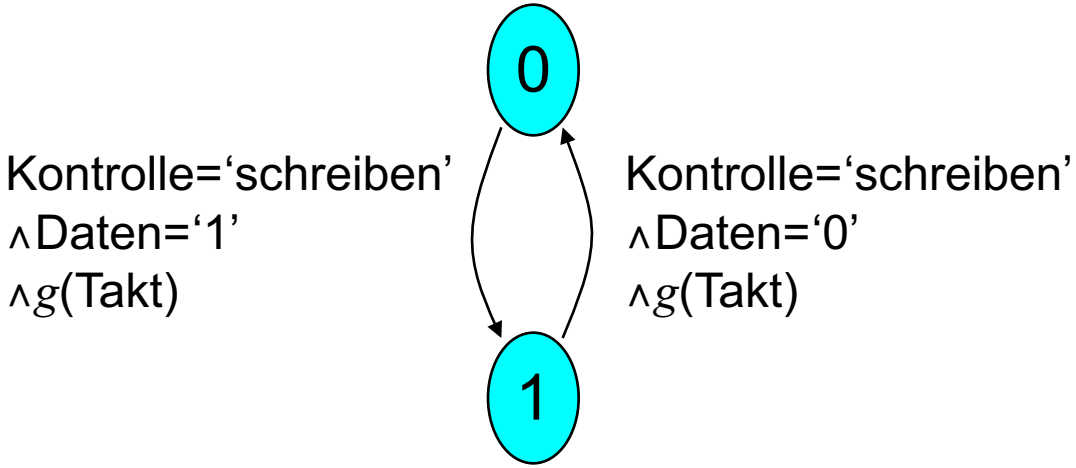
Komponenten von RT-Strukturen: Register

Register



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

Für jedes einzelne Bit:

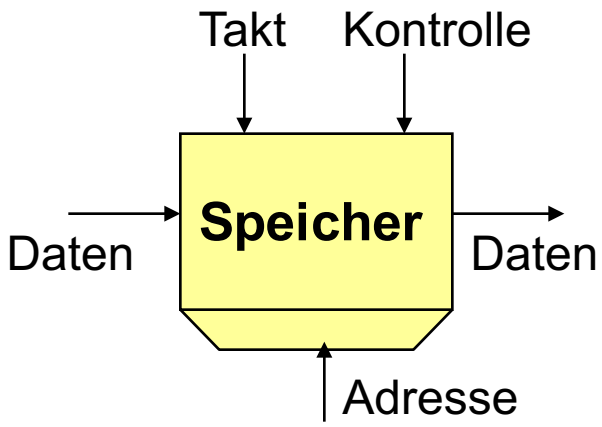


g: modelliert unterschiedliche Taktabhängigkeit

- 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

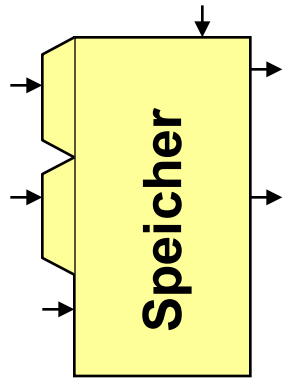


Liest ständig die am Adresseingang ausgewählte Speicherzelle aus und stellt ihren Wert am Datenausgang mit gewisser Verzögerung zur Verfügung.
Übernimmt mit dem Takt die Daten in die ausgewählte Speicherzelle,
sofern der Kontrolleingang auf „Schreiben" gesetzt ist.

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)

Zusammenfassung

Jetzt Sicht auf die interne Architektur:

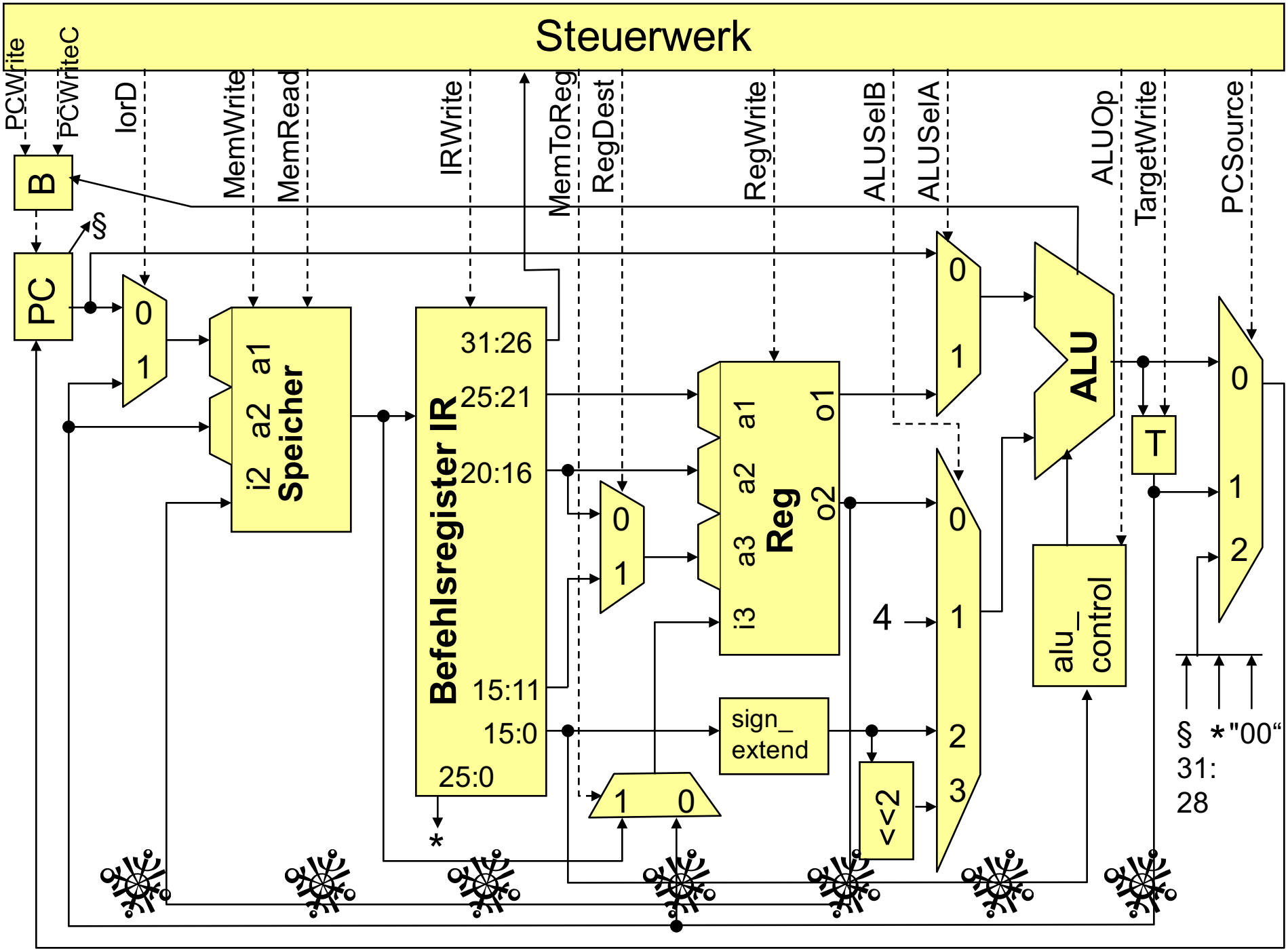
- Komponenten von RT-Strukturen
  - Dekoder
  - Multiplexer
  - ALUs
  - Register
  - Speicher

Eine Mikroarchitektur für die MIPS-Maschine

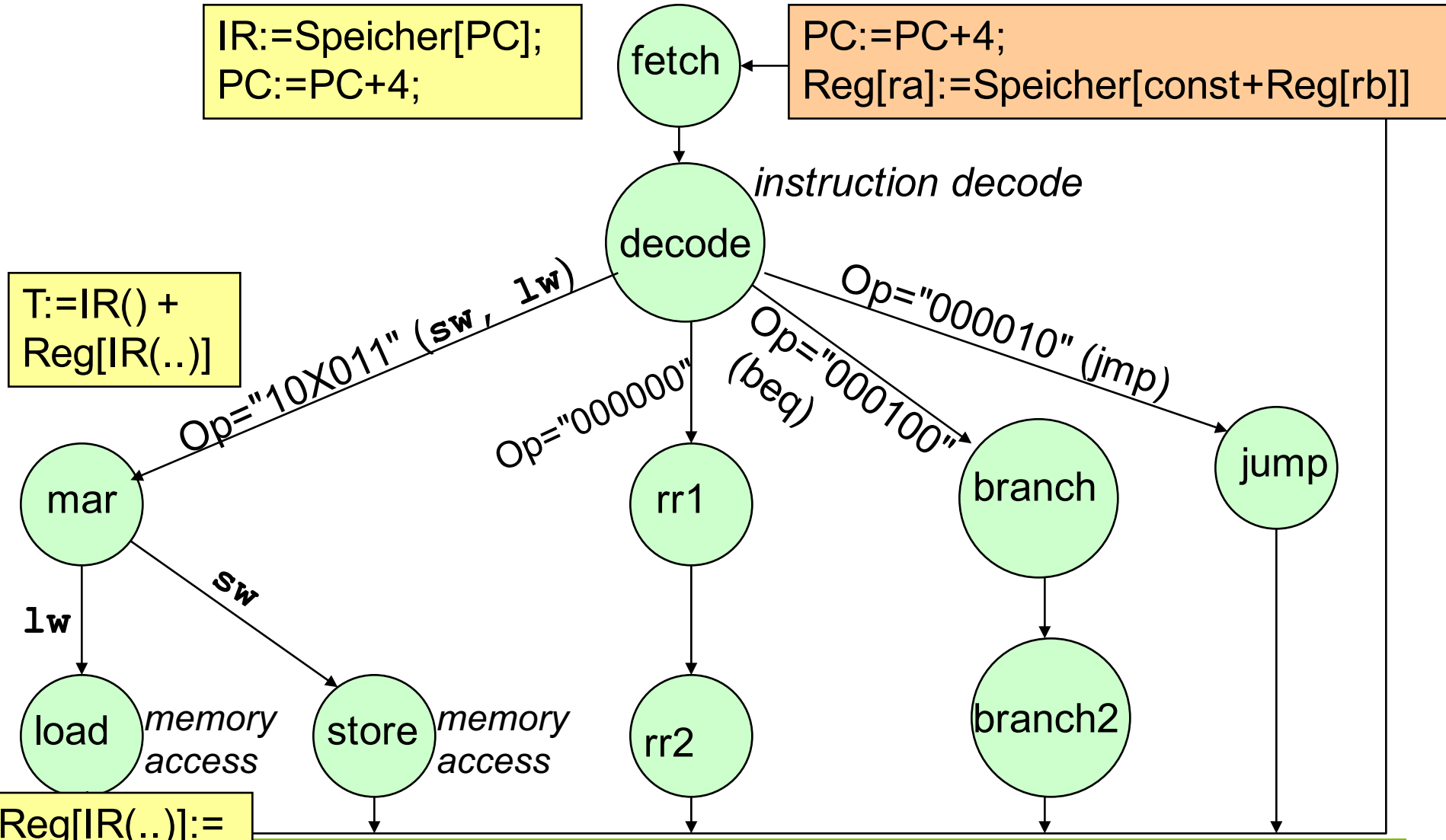
- Automatische Erzeugung von Mikroarchitekturen aus ISA heraus (µArchitektursynthese): wurde untersucht, ist aber
  a) im Rahmen einer Erstsemestervorlesung zu kompliziert
  b) für unsere einfachen Befehle auch nicht nötig.
- Mit Erfahrung (Betrachtung wesentlicher Komponenten/ Verbindungen):
  ☞ manueller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

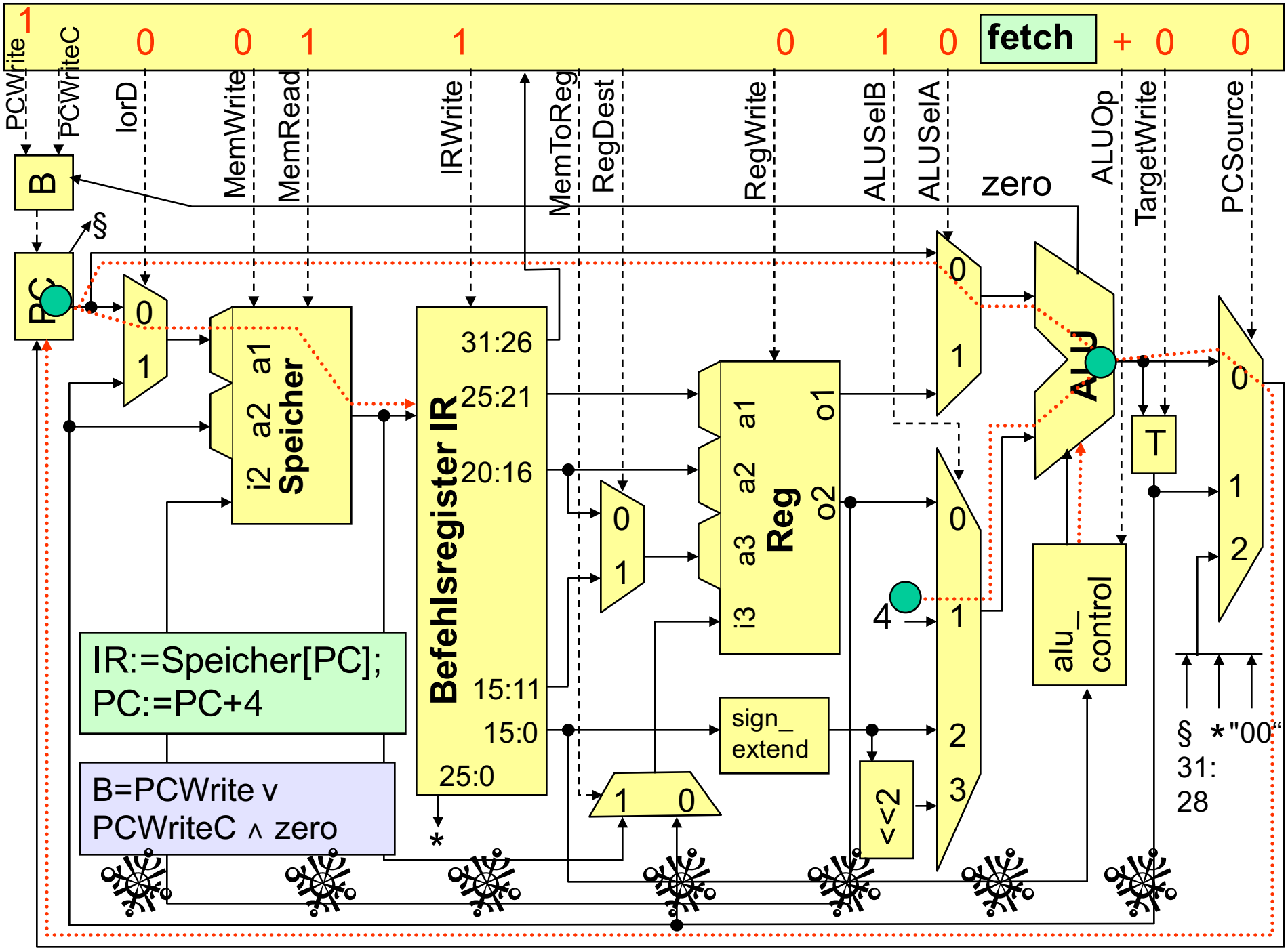
- Verifikation: ∀Befehle: nachweisen, dass die µArchitektur den Befehl ausführen kann.

☞ 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
*
i3 a3 a1 a2
Reg
o1 o2
4
sign_
extend
<<2
ALU
alu_
control
T
2
3
§ 31: 28
*"00"

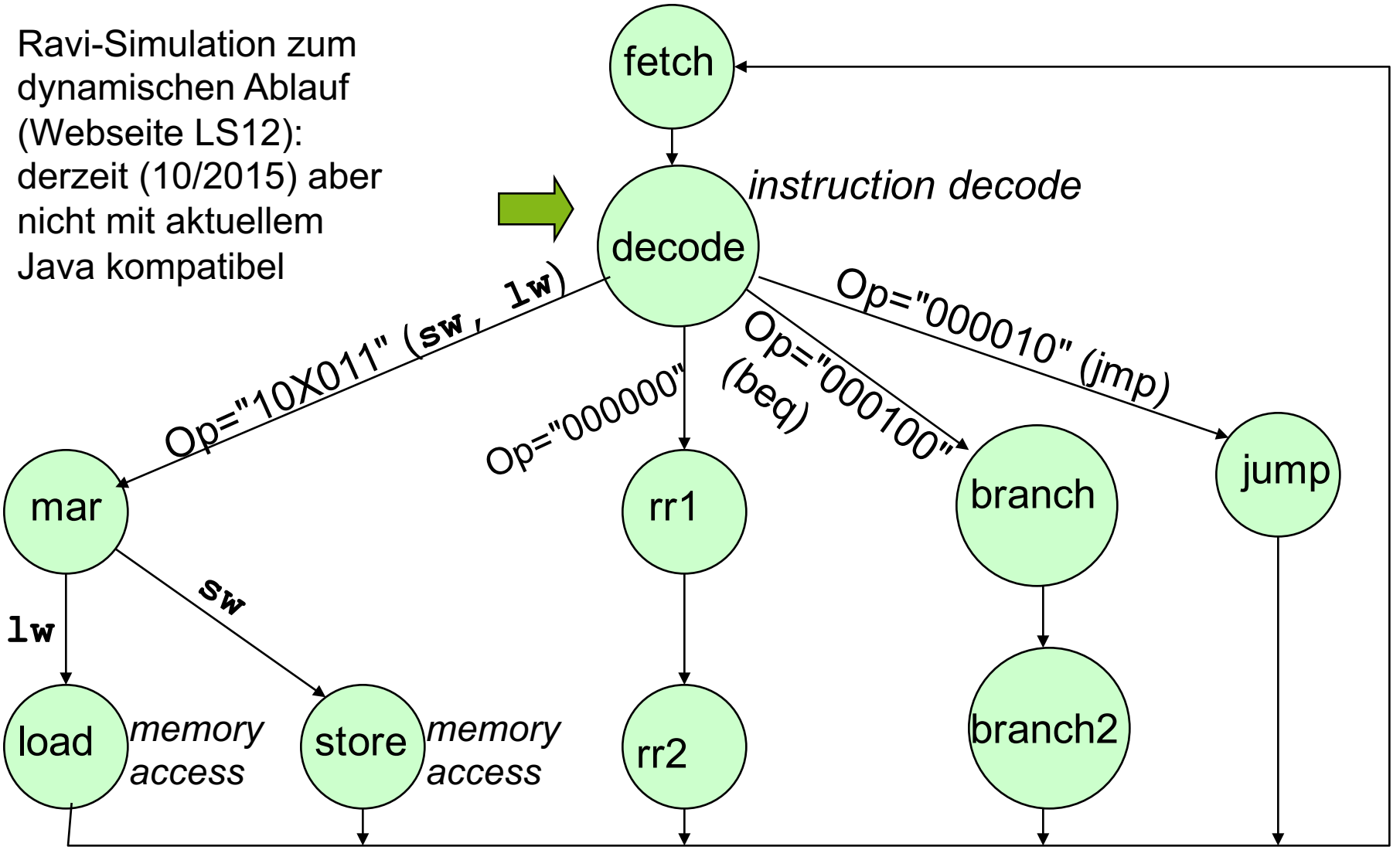
Überprüfung der Ausführbarkeit ☞ Zustandsgraph der Ausführung einiger MIPS-Befehle

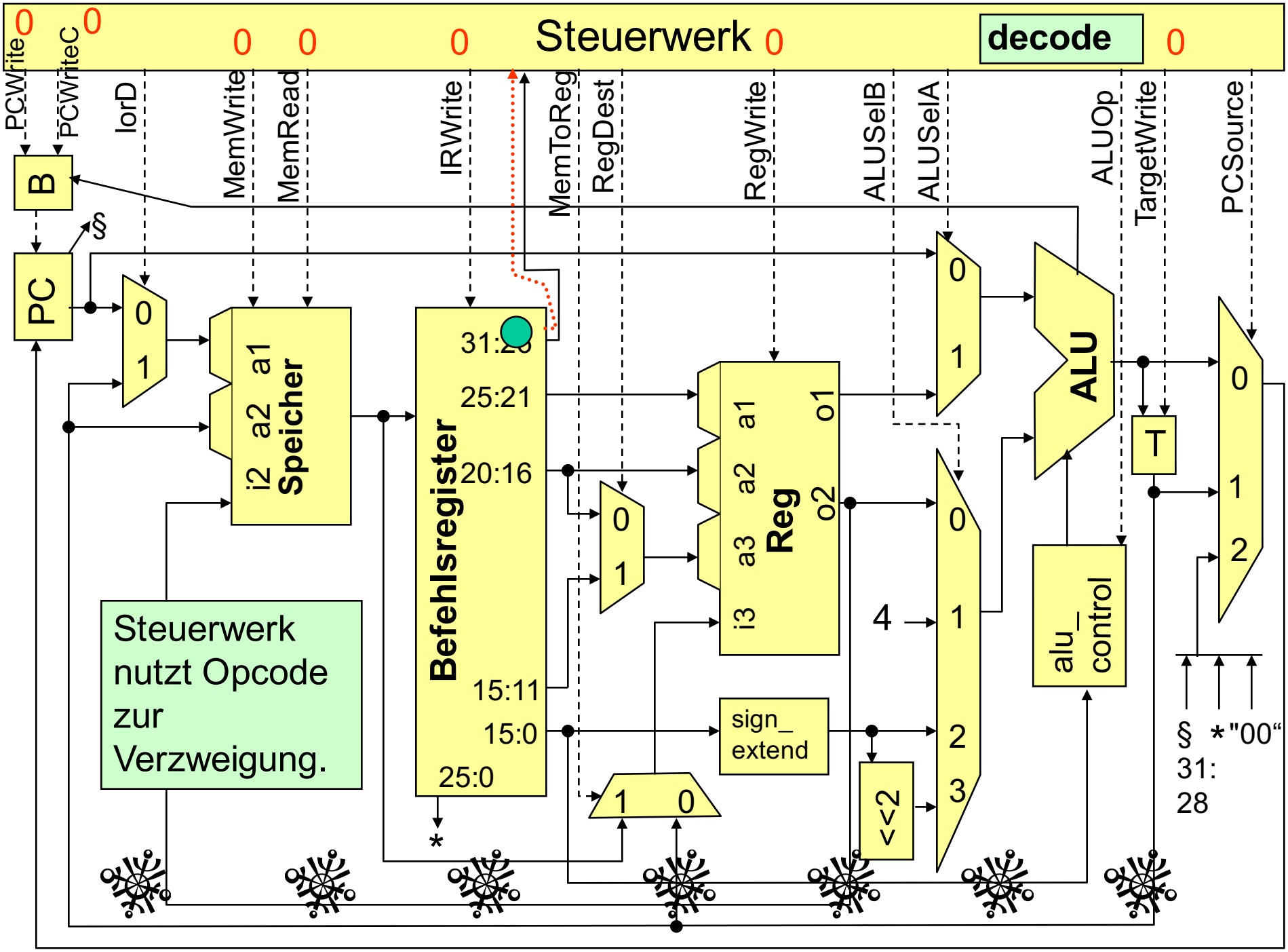



1
PCWrite
PCWriteC
0
0
1
1
0
1
0
fetch
+
0
0
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
zero
ALUOp
TargetWrite
PCSource
B
§
PC
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister IR
*
0
1
i3 a3 a1 a2
Reg
o1
o2
0
1
4
sign_
extend
1 0
<<2
0
1
2
3
ALU
alu_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

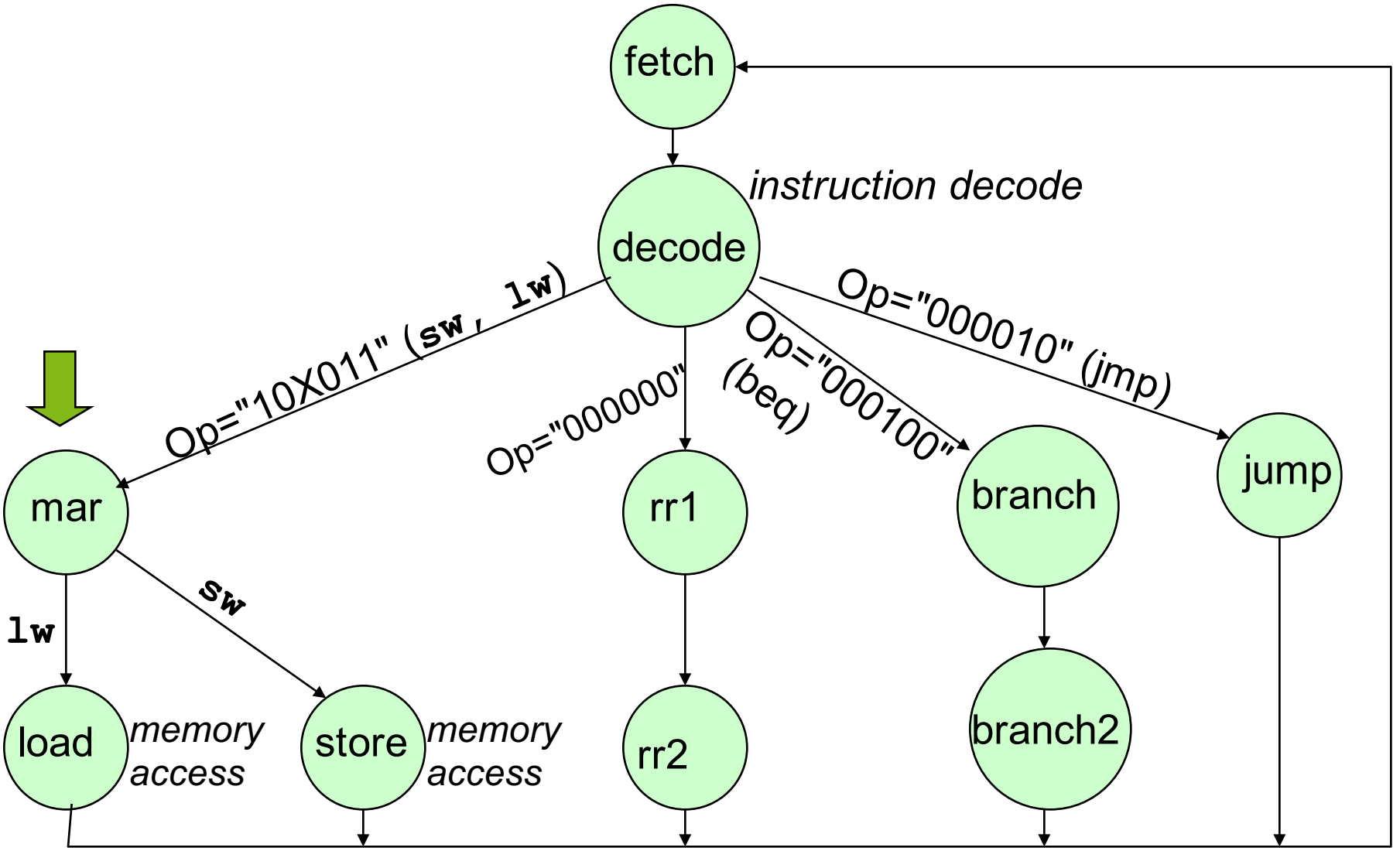
Zustandsgraph der Ausführung einiger MIPS-Befehle

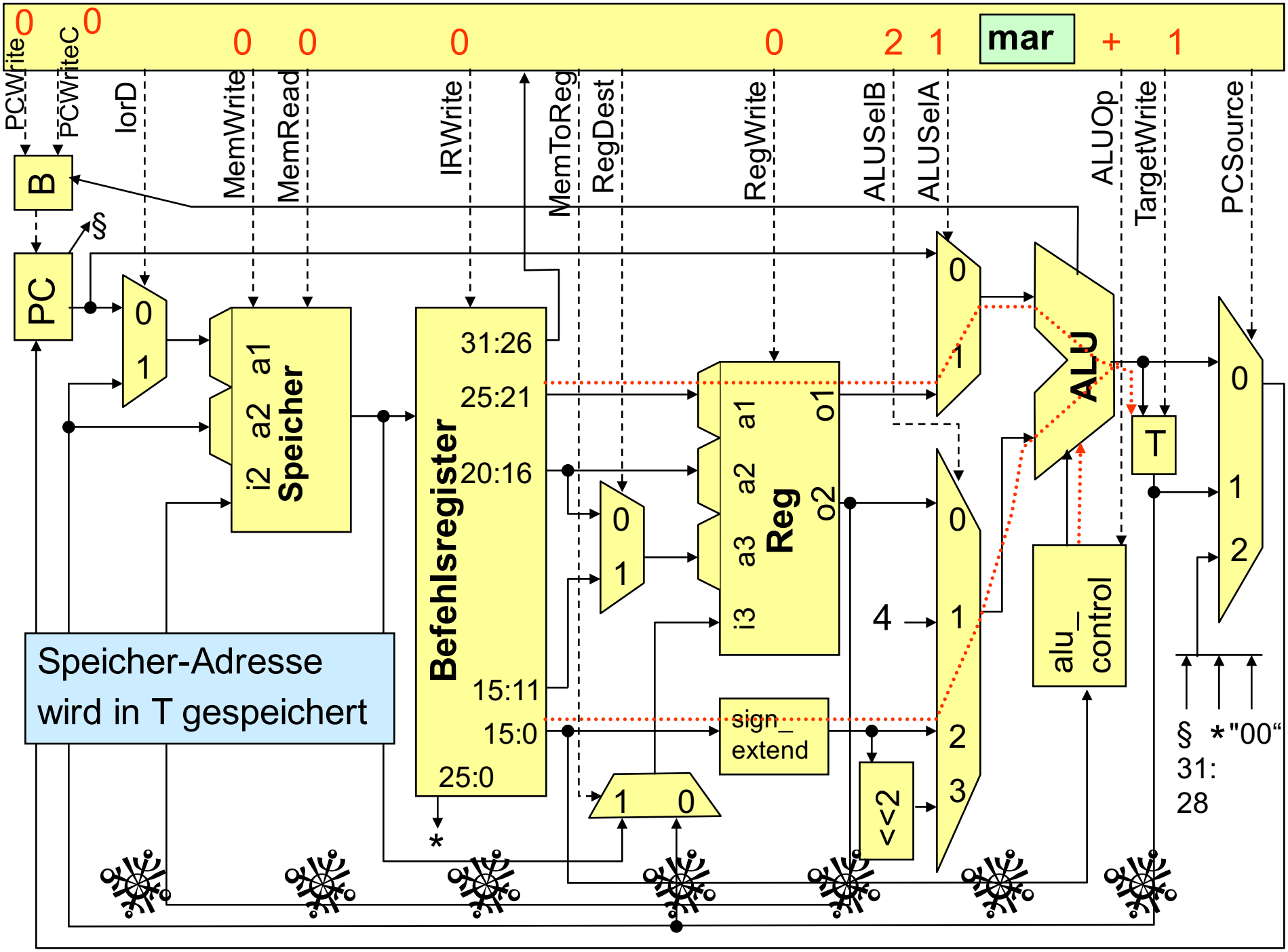
Ravi-Simulation zum dynamischen Ablauf (Webseite LS12): derzeit (10/2015) aber nicht mit aktuellem Java kompatibel



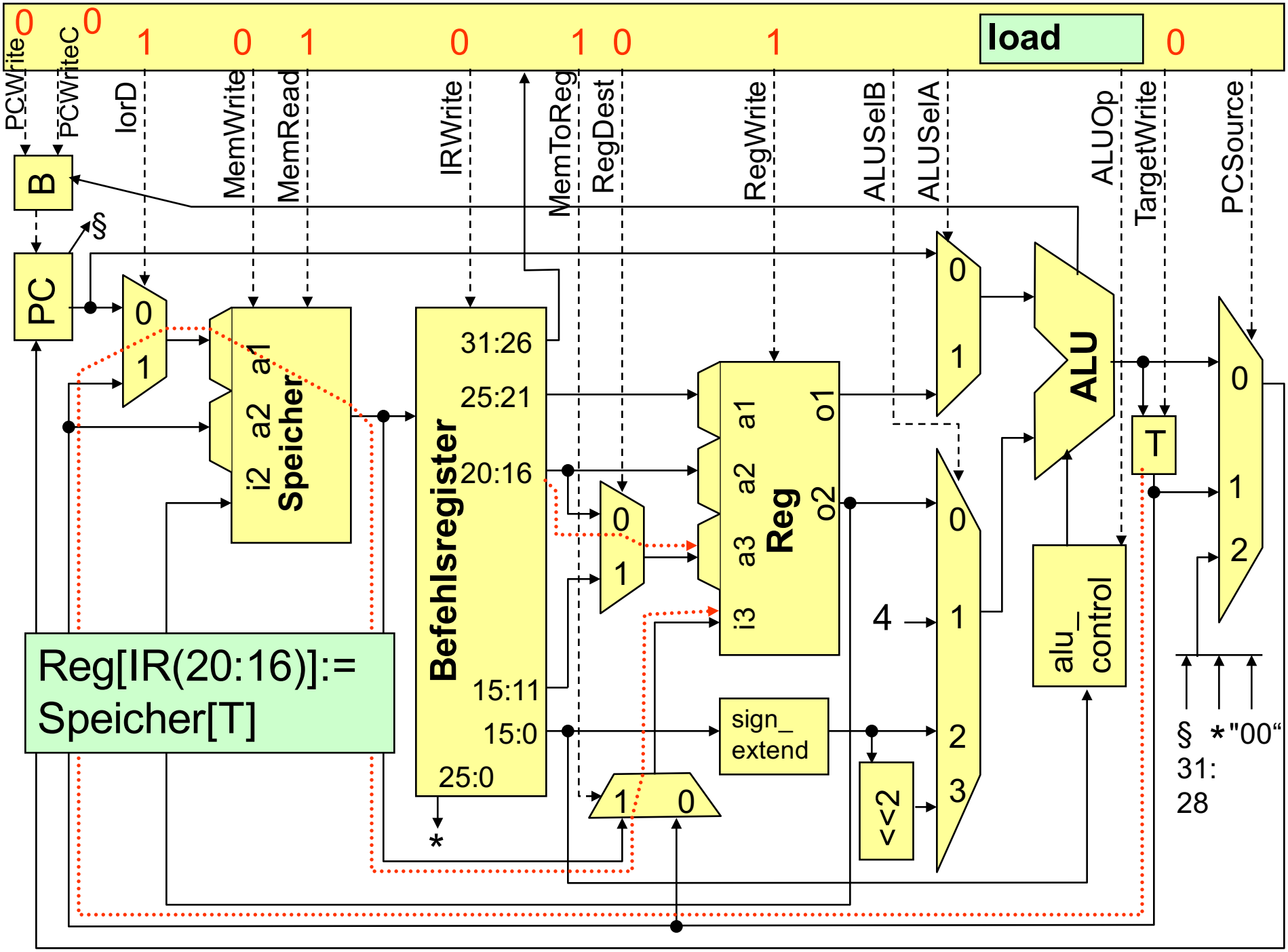

Steuerwerk
decode
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
Speicher
i2 a2 a1
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
Reg
i3 a3 a2 a1 o2 o1
ALU
T
alu_ control
sign_ extend
<<2
4
§ 31: 28
*"00"
Steuerwerk nutzt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

Zustandsgraph der Ausführung einiger MIPS-Befehle

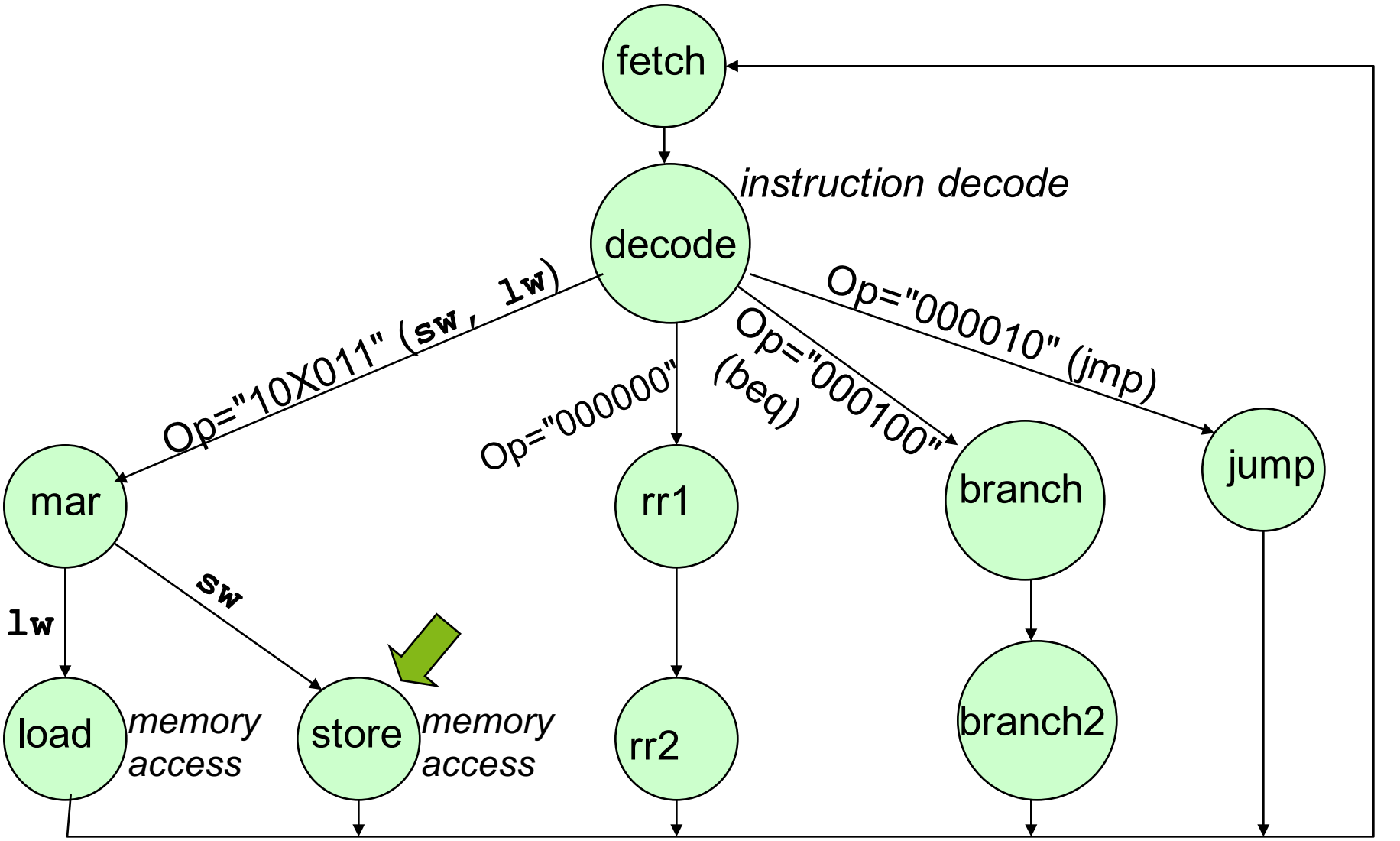


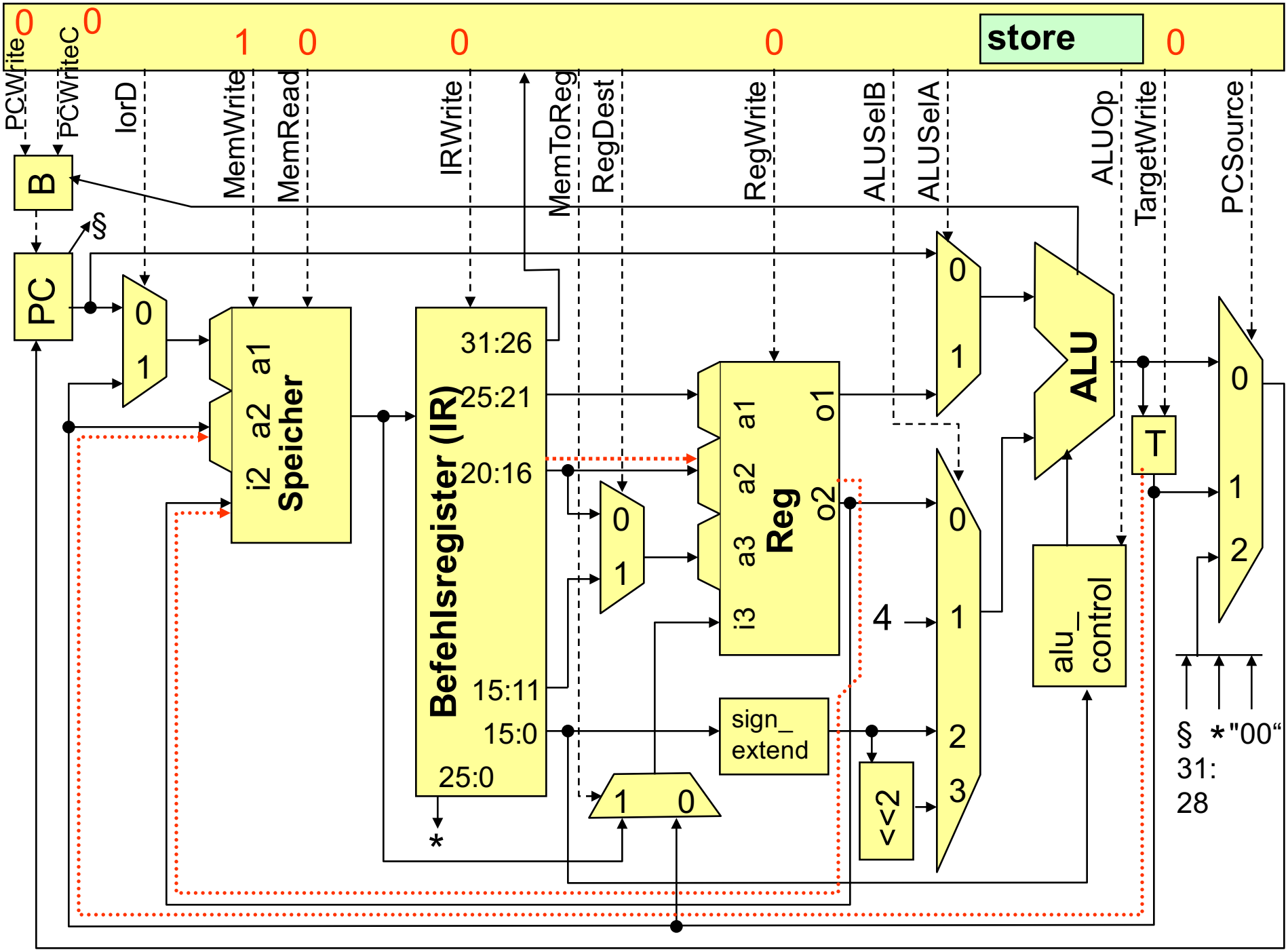

0
PCWrite
0
PCWriteC
0
IorD
0
MemWrite
0
MemRead
0
IRWrite
0
MemToReg
0
RegDest
0
RegWrite
2
ALUSelB
1
ALUSelA
mar
ALUOp
+
1
TargetWrite
PCSource
B
§
PC
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister
*
0
1
i3 a3 a1 a2
Reg
o1 o2
0
1
4
0
1
2
3
sign_
extend
<<2
1 0
0
1
ALU
alu_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

Zustandsgraph der Ausführung einiger MIPS-Befehle

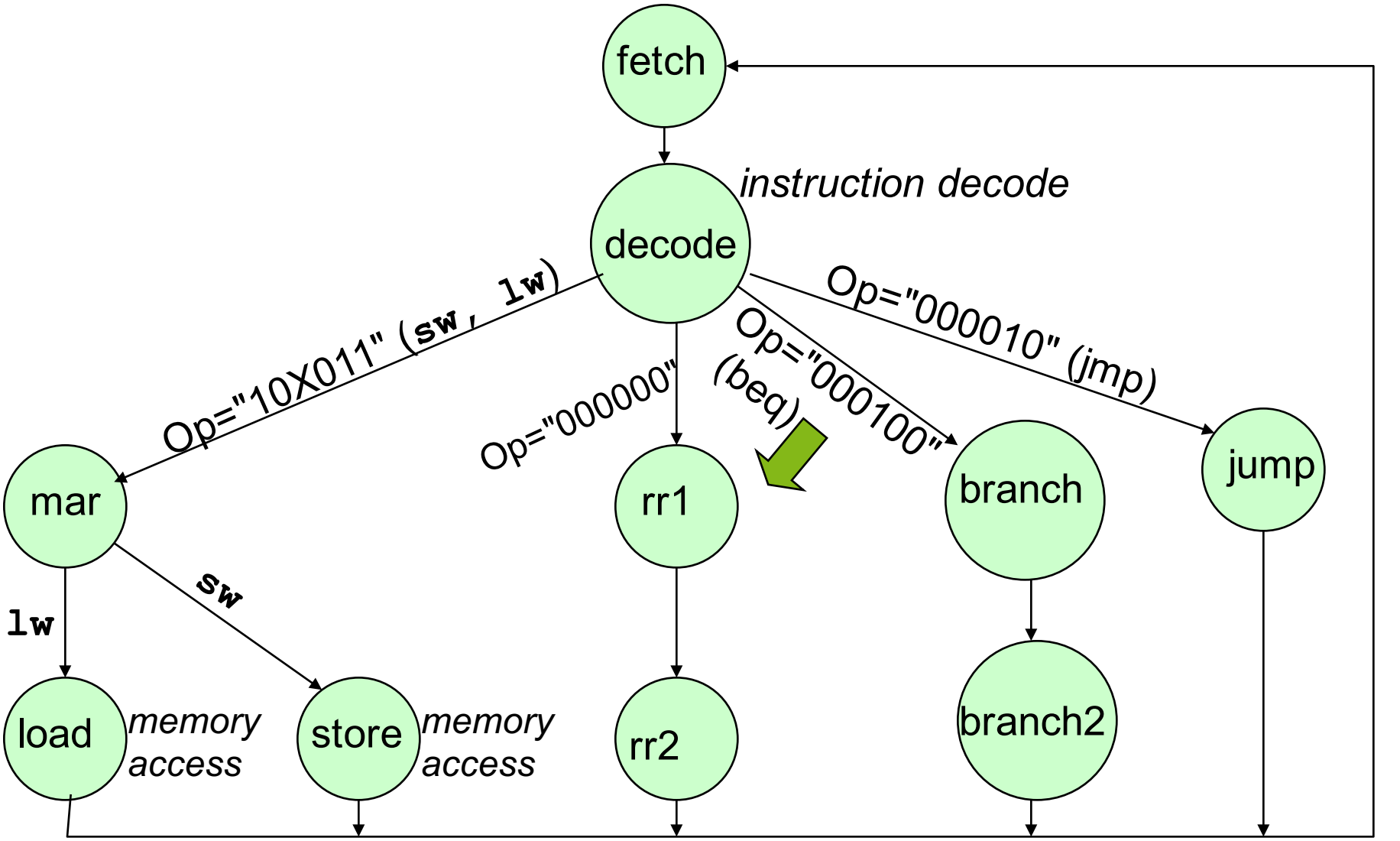

0
0
1
0
1
0
1
0
1
load
0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister
*
0
1
i3 a3 a2 a1
Reg
o1
o2
0
1
4
sign_
extend
<<2
1 0
0
1
2
3
ALU
alu_
control
T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

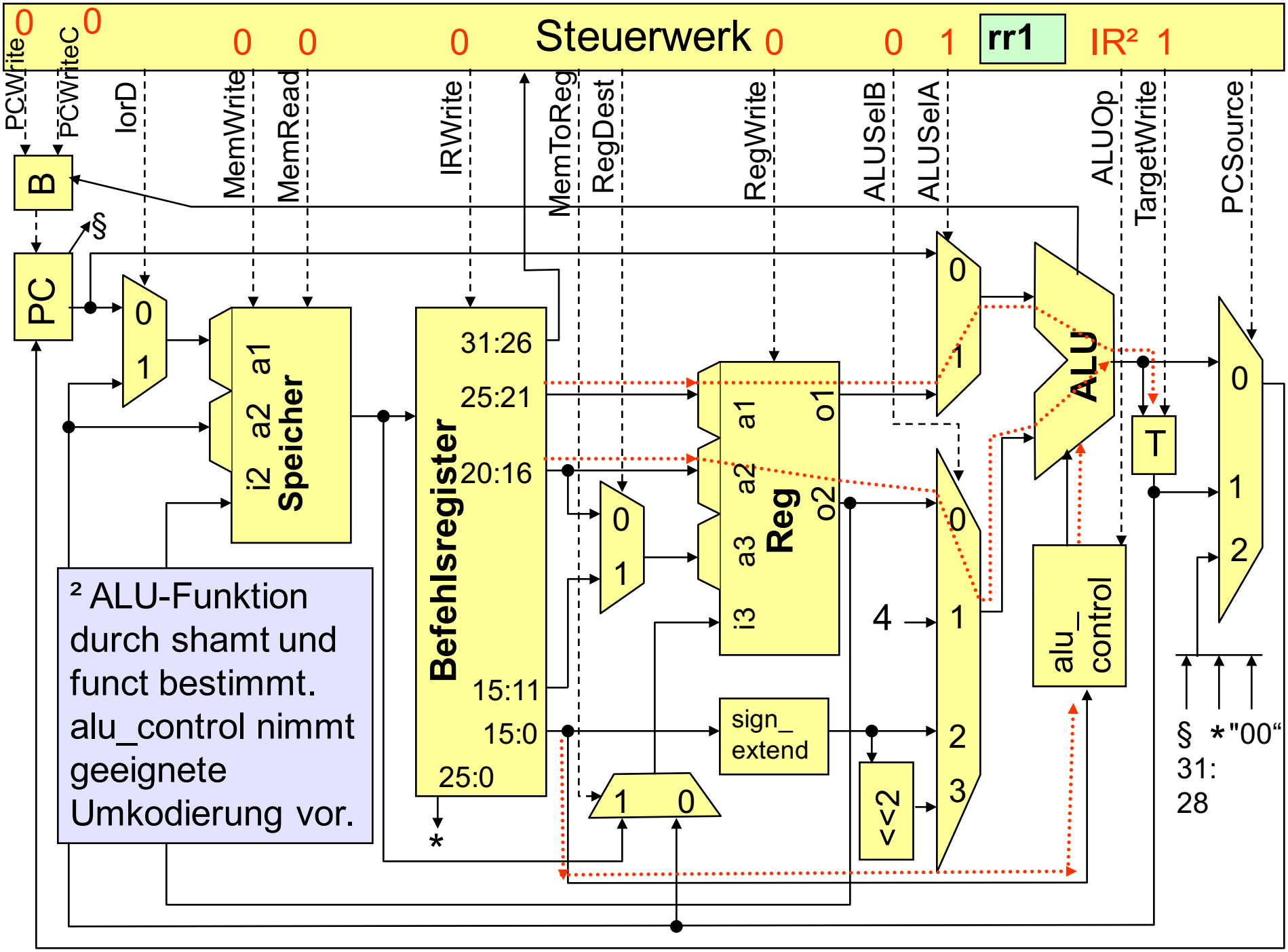
Zustandsgraph der Ausführung einiger MIPS-Befehle



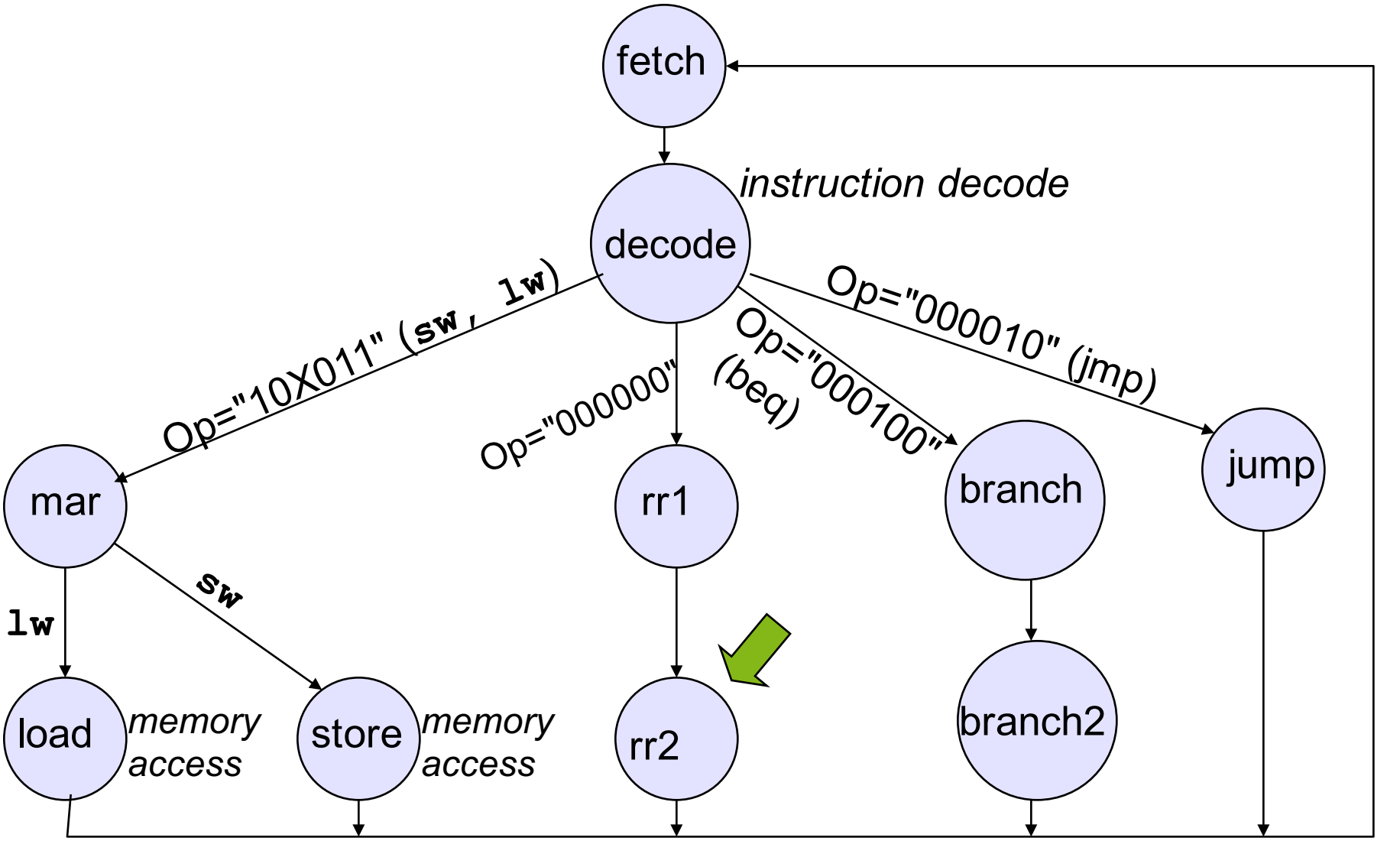

0
0
1
0
0
0
store
0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
31:26
25:21
20:16
15:11
15:0
25:0
Befehlsregister (IR)
*
0
1
i3 a3 a2 a1
Reg
o2 o1
4
sign_
extend
1 0
<<2
0
1
2
3
ALU
alu_
control
T
0
1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

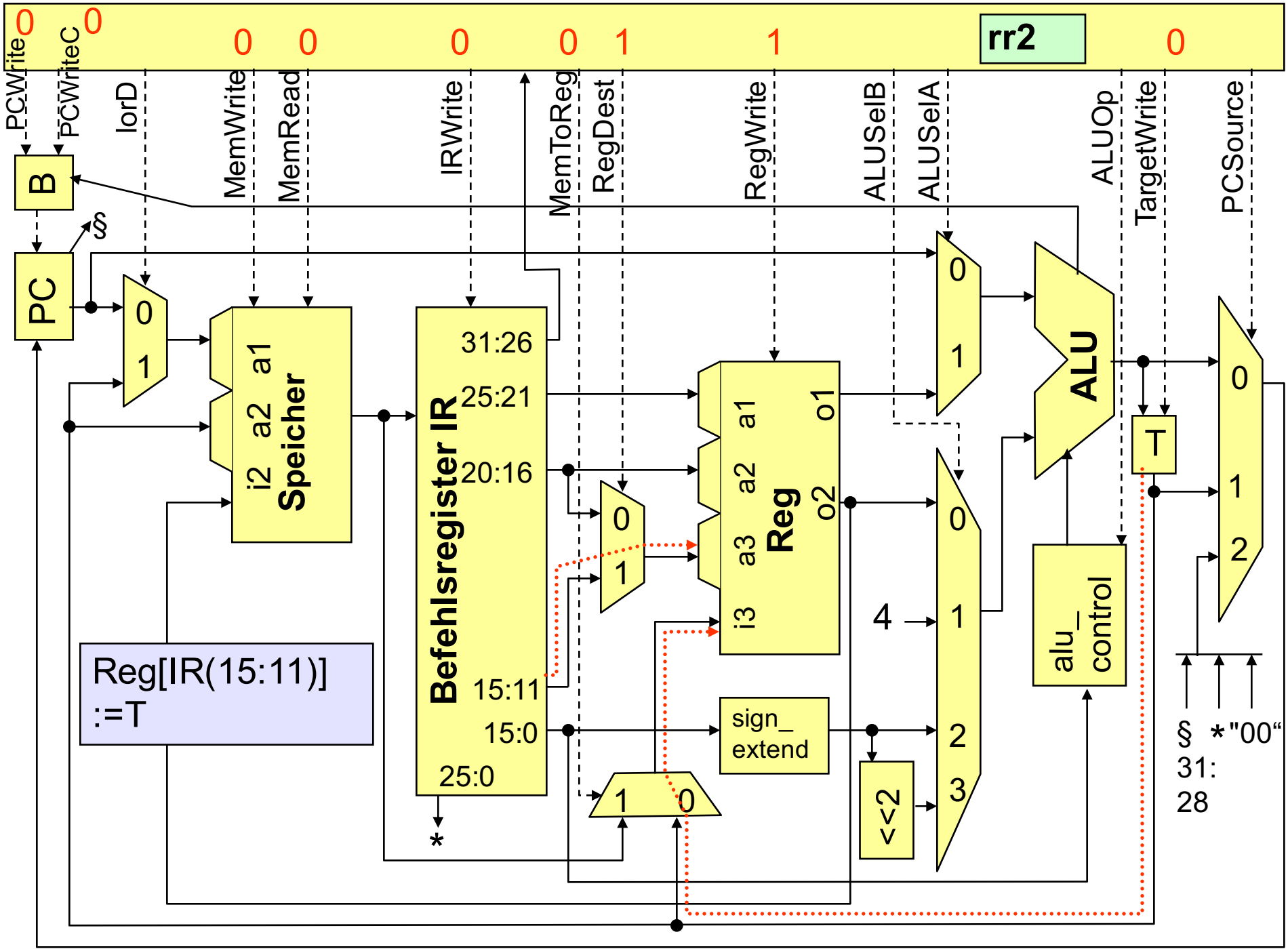
Zustandsgraph der Ausführung einiger MIPS-Befehle



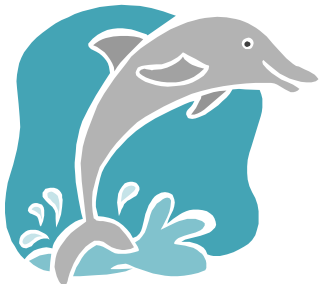

0 PCWrite
0 PCWriteC
0 IorD
0 MemWrite
0 MemRead
0 IRWrite
Steuerwerk
0 MemToReg
0 RegDest
1 RegWrite
rr1 ALUSelB
IR² ALUSelA
1 ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
*
0
1
i3 a3 a1 a2
Reg
o1 o2
0
1
4
2
3
sign_
extend
1 0
<<2
ALU
alu_
control
T
0
1
2
§ 31:28
*"00"
² ALU-Funktion durch shamt und funct bestimmt. alu_control nimmt geeignete Umkodierung vor.

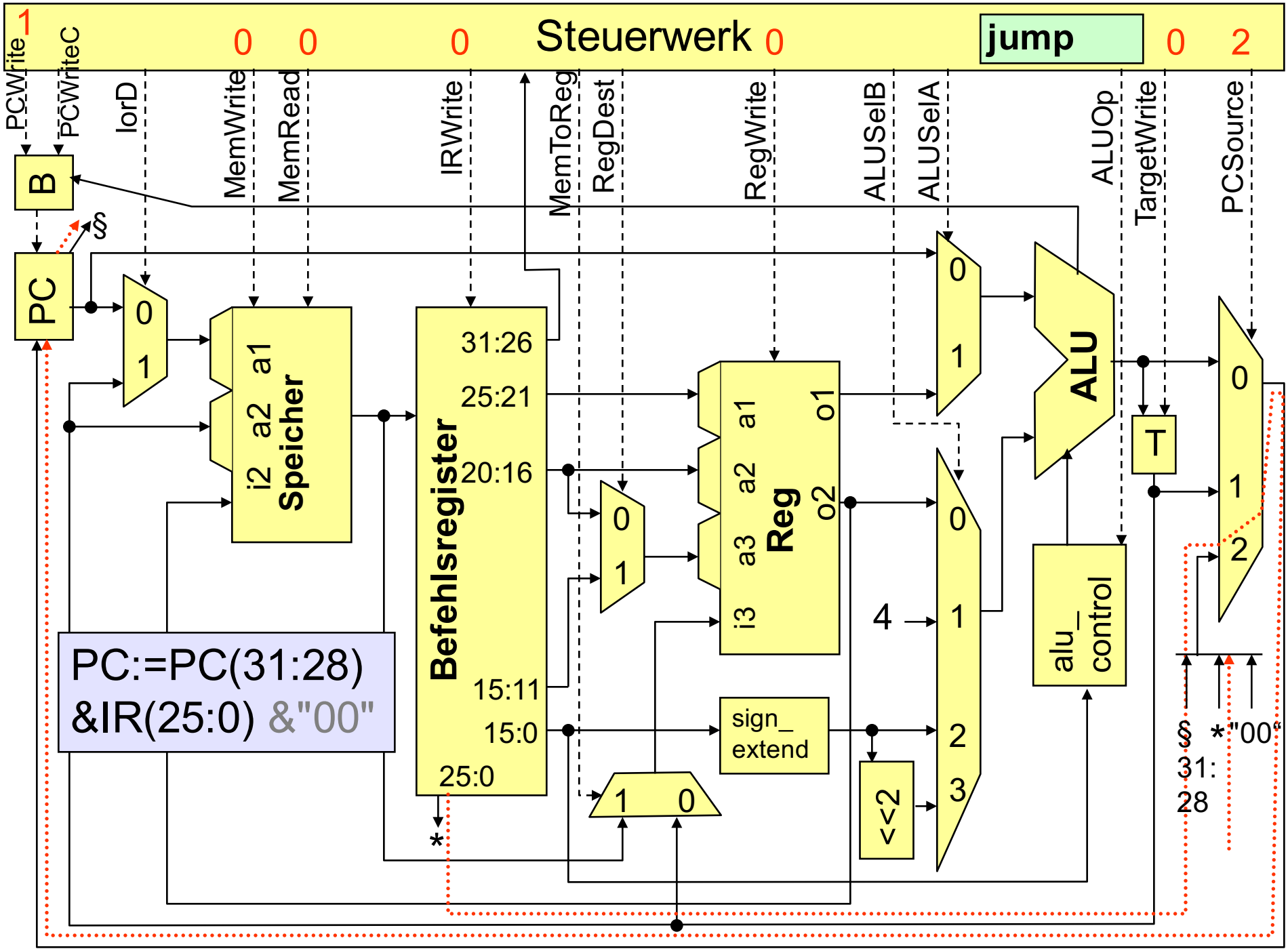
Zustandsgraph der Ausführung einiger MIPS-Befehle



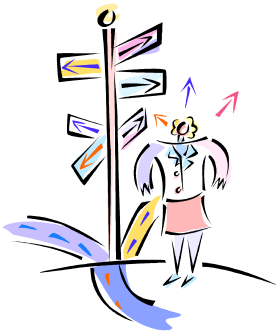

0 0 0 0 0 0 1 1 rr2 0
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
0
1
i2 a2 a1
Speicher
Befehlsregister IR
31:26
25:21
20:16
15:11
15:0
25:0
*
Reg[IR(15:11)]
:=T
i3 a3 a1 o1 o2
Reg
4
sign_
extend
<<2
1 0
ALU
alu_
control
T
2
3
§ 31: 28
*"00"

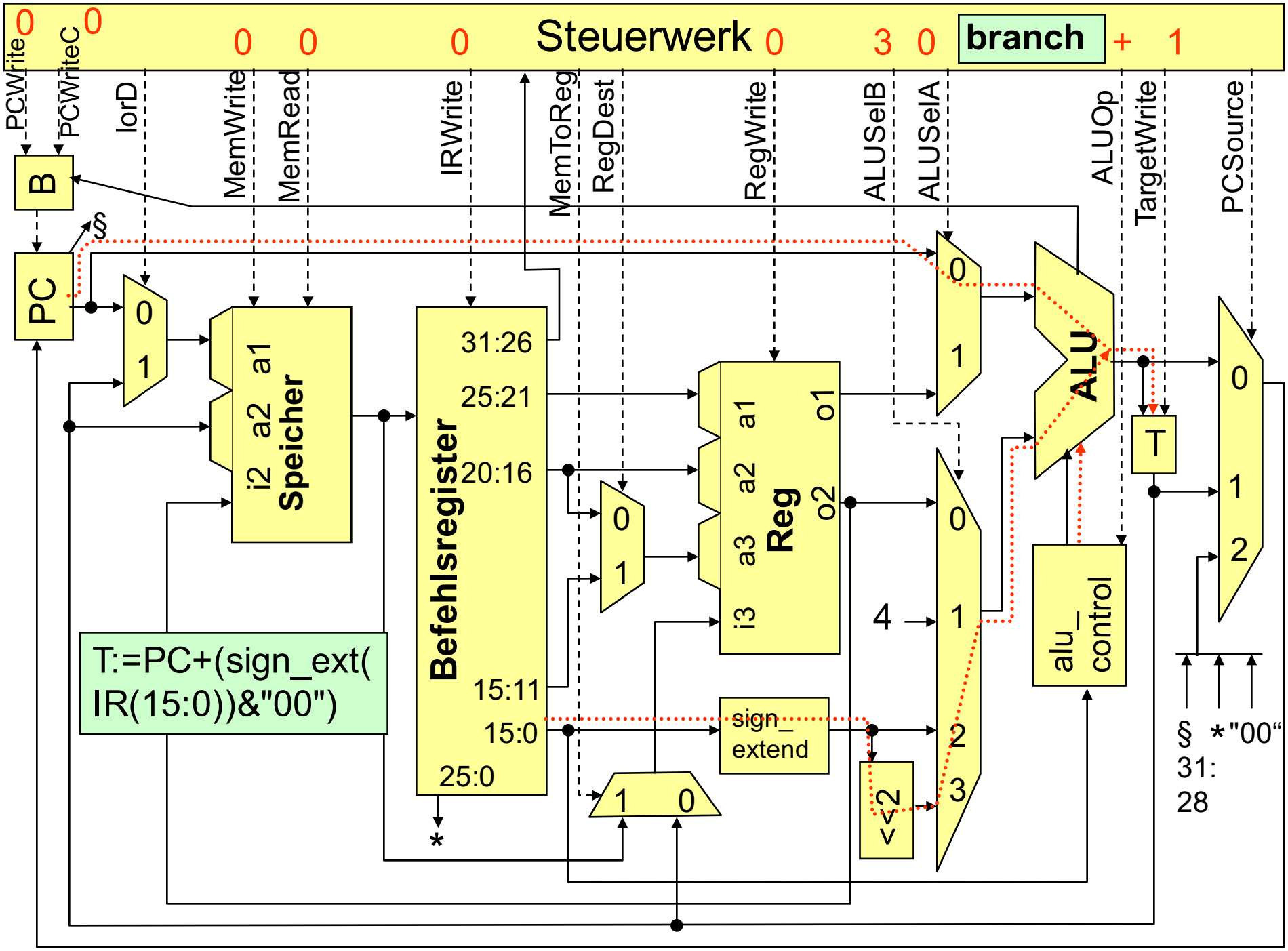
Zustandsgraph der Ausführung einiger MIPS-Befehle




1
PCWrite
PCWriteC
0
0
0
Steuerwerk
0
jump
0
2
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 a1 a2
Reg
o1 o2
0
1
4
2
3
<<2
sign_
extend
1 0
ALU
alu_
control
T
0
1
2
§
31:
28
*
"00"
PC:=PC(31:28)
&IR(25:0) &"00"

Zustandsgraph der Ausführung einiger MIPS-Befehle



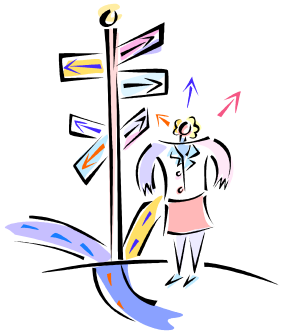

Steuerwerk
branch
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
ALUOp
TargetWrite
PCSource
B
PC
§
Speicher
i2 a2 a1
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
Reg
i3 a3 a2 a1 o2 o1
ALU
T
alu_control
sign_extend
<<2
4
*
§ 31:28
*"00"
T:=PC+(sign_ext(IR(15:0))&"00")

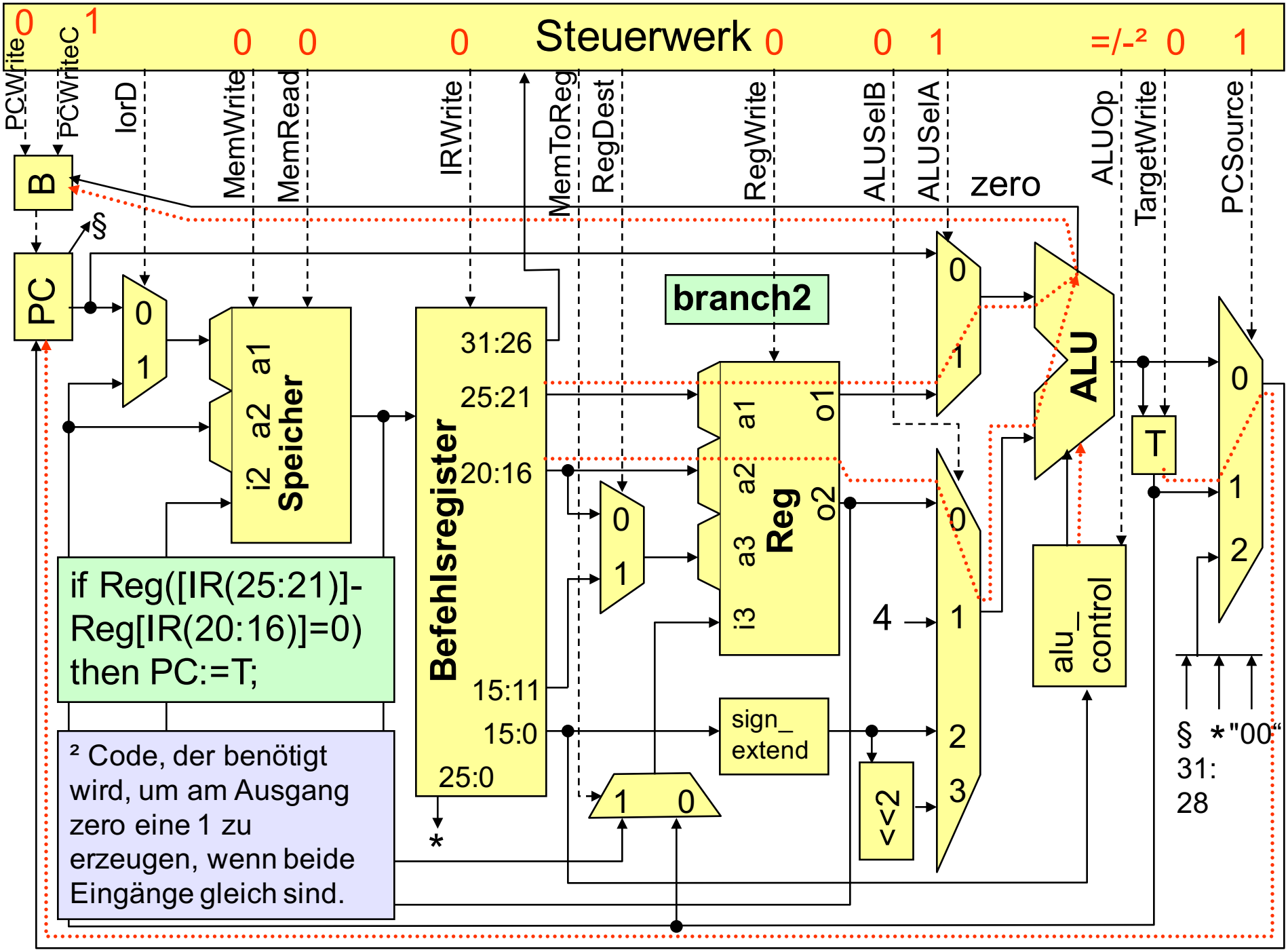
Quiz

0x400000: beq $8, $9, 0x02

Welche Befehlsadresse wird nach beq (0x400000) ausgeführt, wenn $8 und $9 gleich sind?

Zustandsgraph der Ausführung einiger MIPS-Befehle

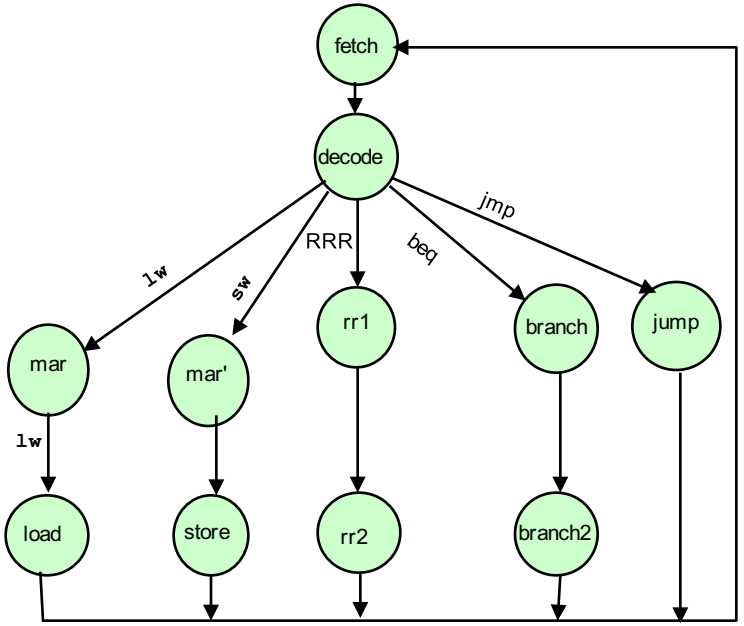



Steuerwerk
PCWrite
PCWriteC
IorD
MemWrite
MemRead
IRWrite
MemToReg
RegDest
RegWrite
ALUSelB
ALUSelA
zero
ALUOp
TargetWrite
PCSource
B
PC
§
branch2
Speicher
i2 a2 a1
Befehlsregister
31:26
25:21
20:16
15:11
15:0
25:0
Reg
i3 a3 a2 a1 o2 o1
ALU
T
alu_ control
sign_ extend
<<2
4
*
§ 31: 28
*"00"
if Reg([IR(25:21)]-Reg[IR(20:16)]=0)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

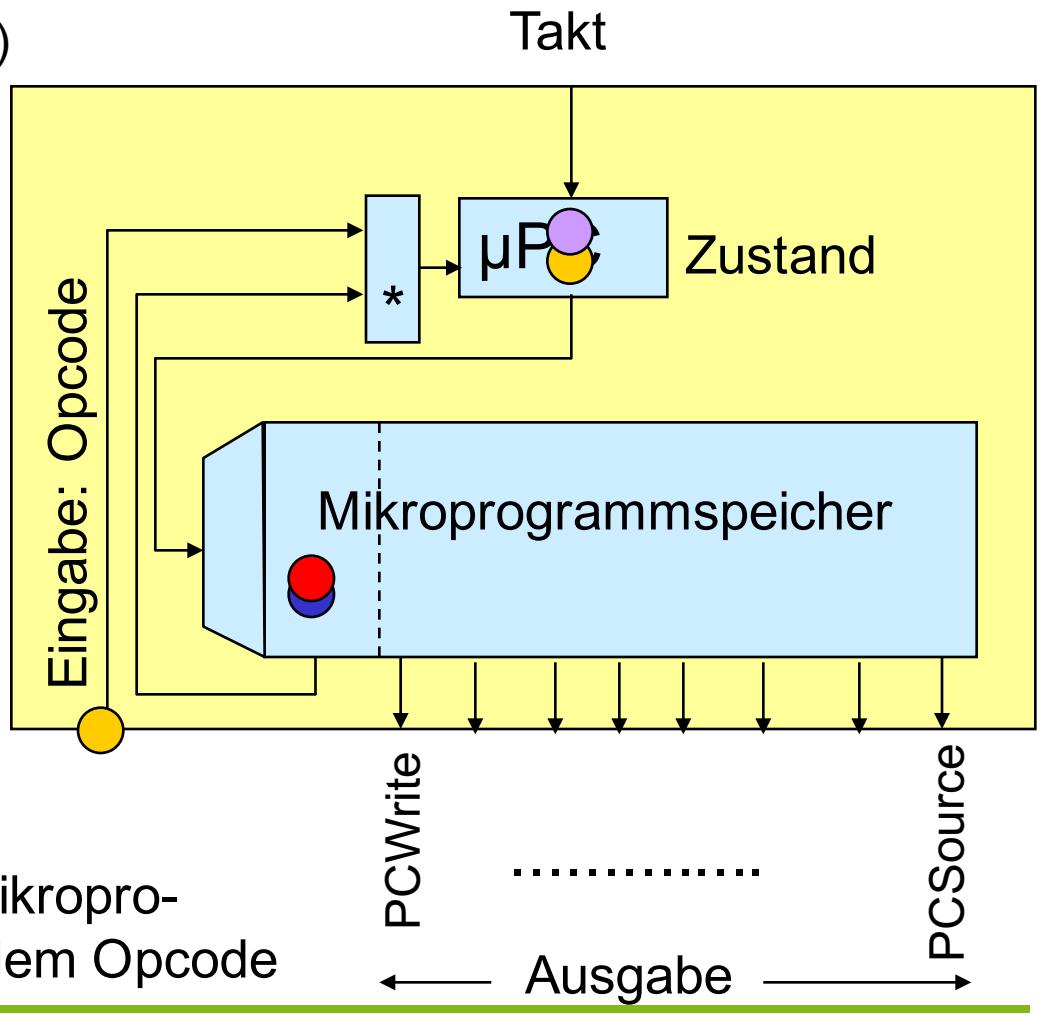
Das Steuerwerk

Verhalten, vereinfacht

(mar‘ vermeidet Verzweigung bei mar)



Struktur

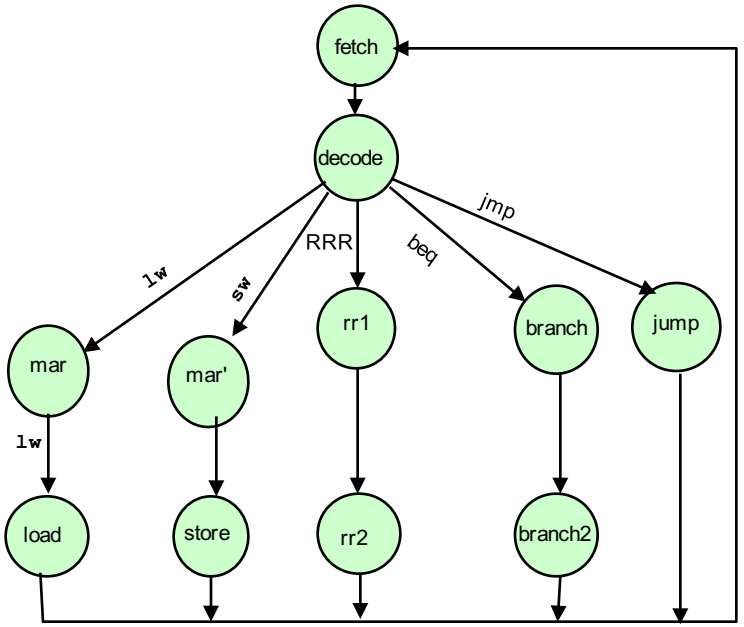


* Folgezustand bestimmt durch Mikroprogrammspeicher, bei decode aus dem Opcode

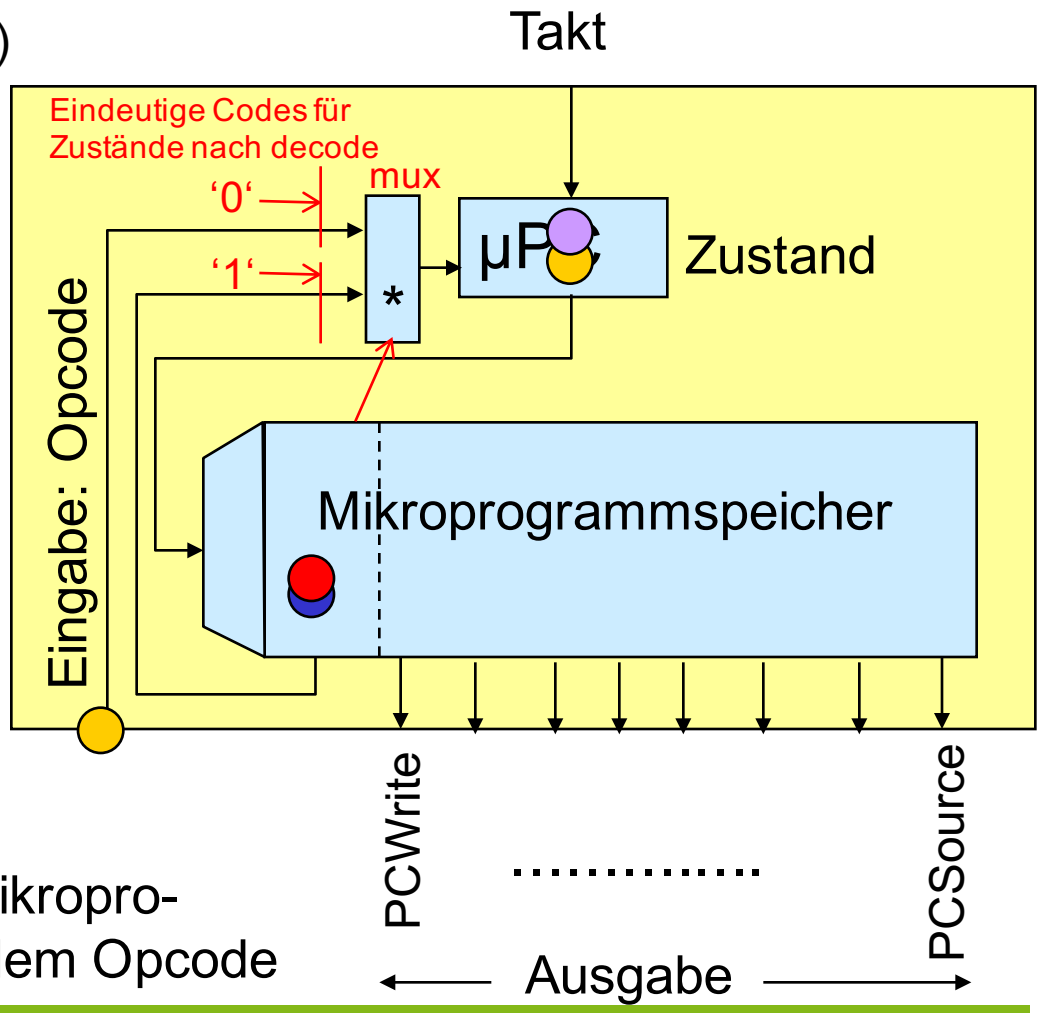
Das Steuerwerk (etwas genauer)

Verhalten, vereinfacht

(mar‘ vermeidet Verzweigung bei mar)

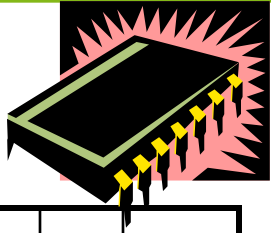


Struktur



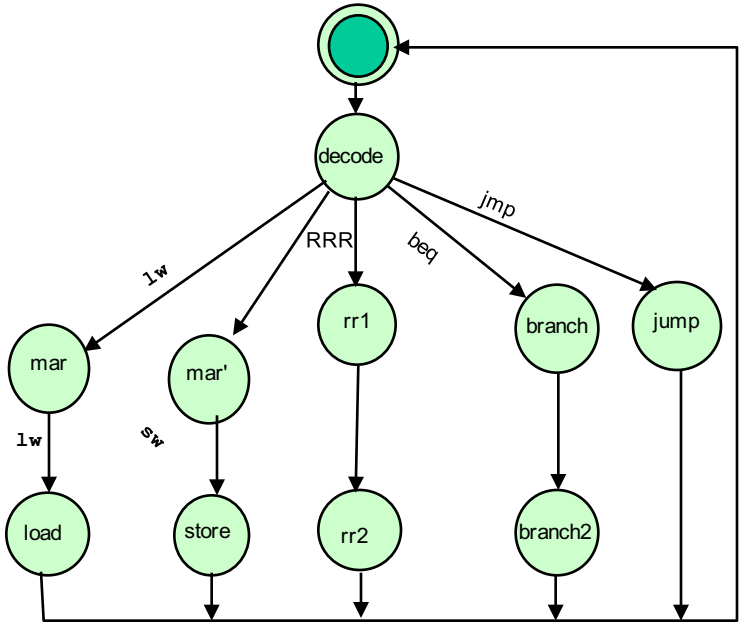
* Folgezustand bestimmt durch Mikropro-grammspeicher, bei decode aus dem Opcode

Inhalt des Mikroprogrammspeichers

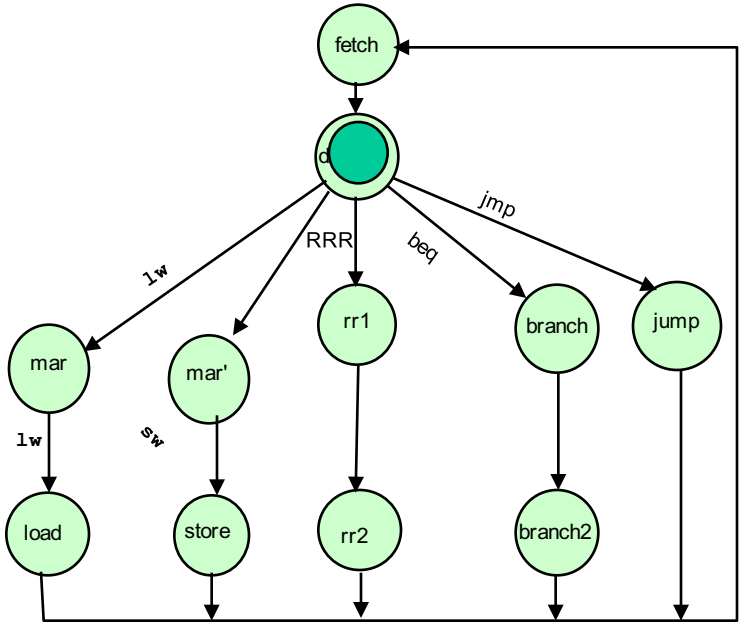


Zustand (bestimmt Adresse im Speicher)	Folge-Zustand bzw. – Zustände	+ Art der Bestimmung des Folgezustands	PCWrite	PCWriteC	IorD	MemWrite	MemRead	IRWrite	Mem2Reg	RegDest	RegWrite	ALUSelB	ALUSelA	ALUOp	TargetWri	PCSource
fetch	decode		1	0	0	0	1	1	X	X	0	01	0	+	0	00
decode	f(Opcode)		0	0	X	0	0	0	X	X	0	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

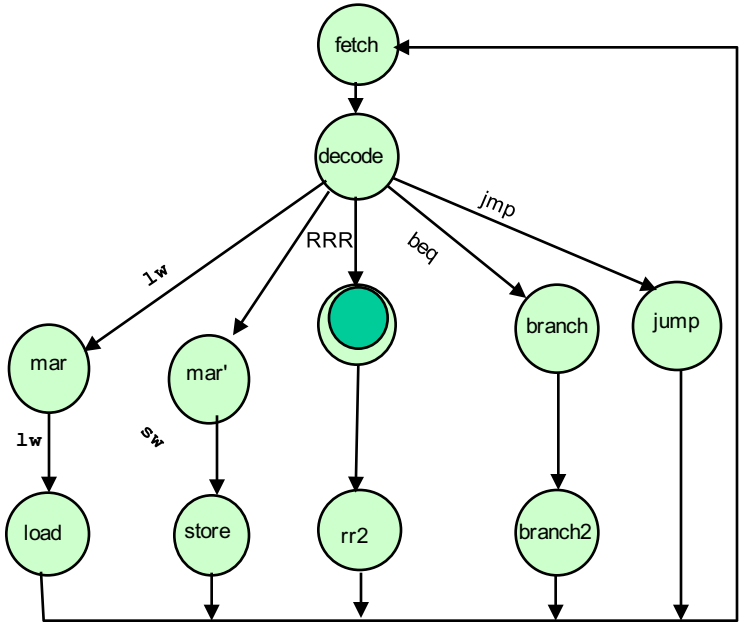
Mikroprogrammierung



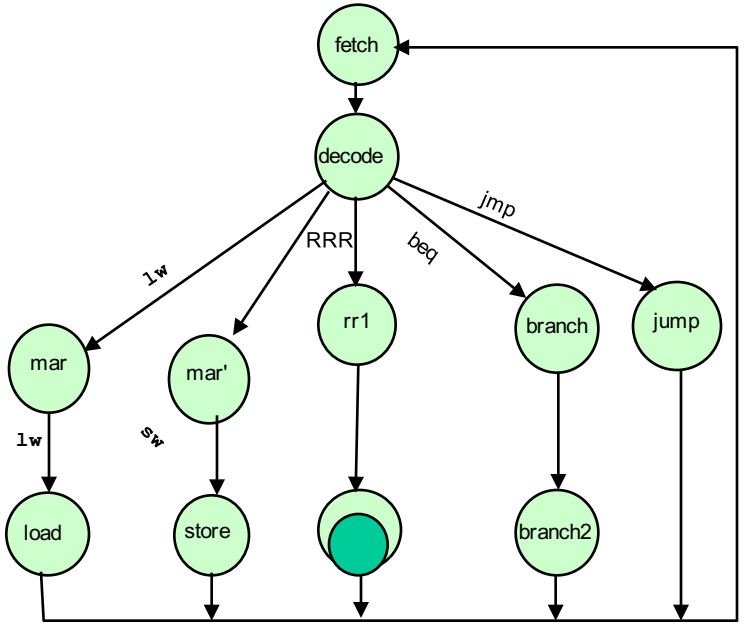
Mikroprogrammierung



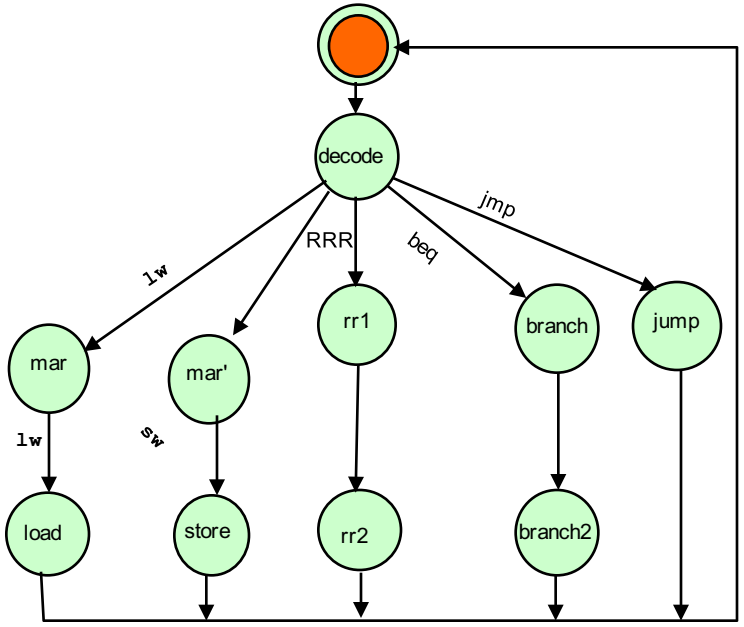
Mikroprogrammierung



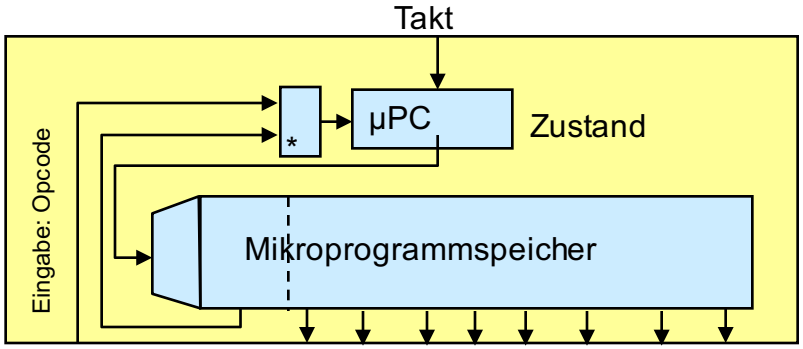
Mikroprogrammierung



Mikroprogrammierung



Struktur



Zusammenfassung

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

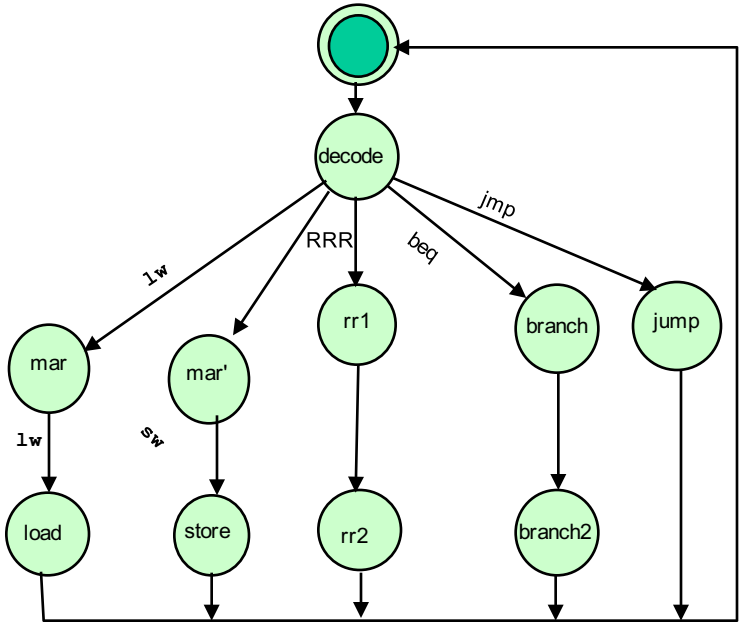
Vorteile:

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

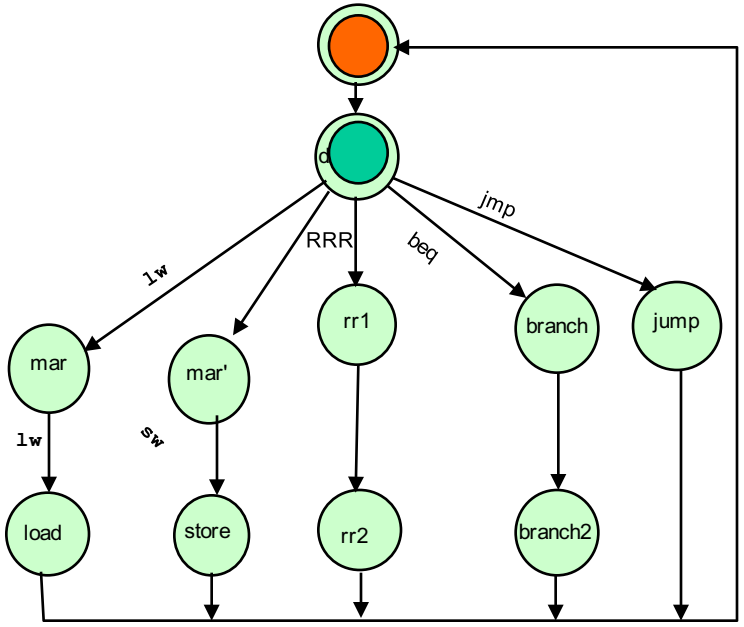
Nachteile:

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

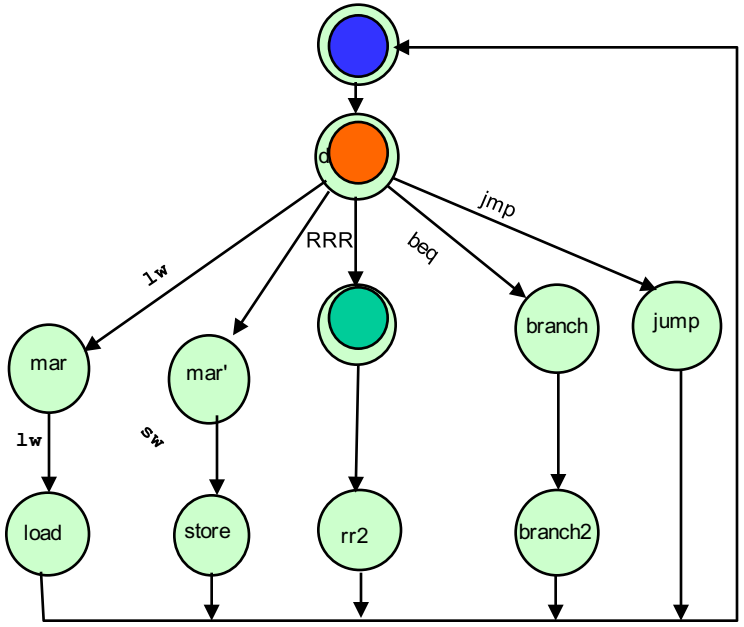
Mikroprogrammierung → Fließbandverarbeitung



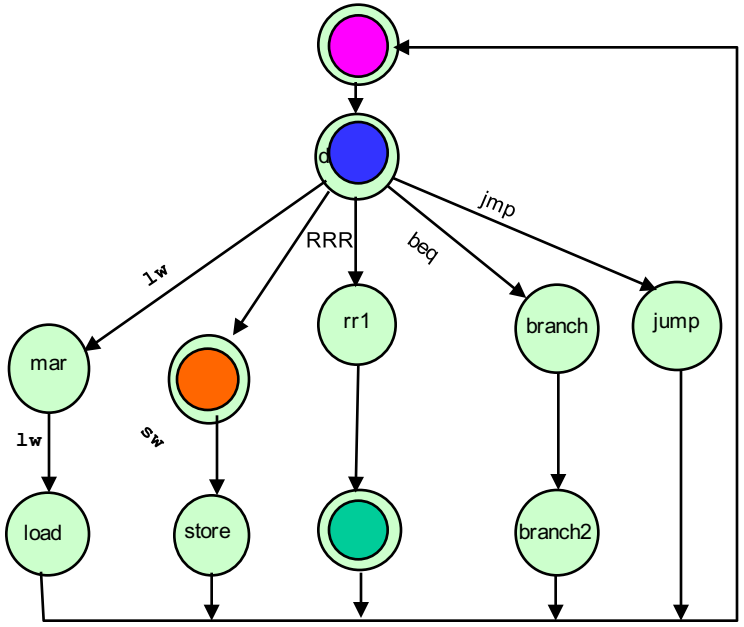
Mikroprogrammierung → Fließbandverarbeitung



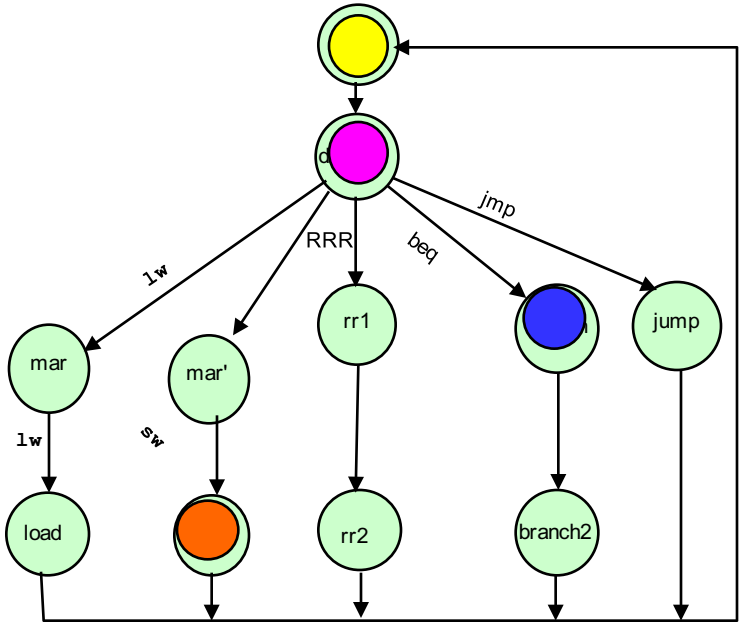
Mikroprogrammierung → Fließbandverarbeitung

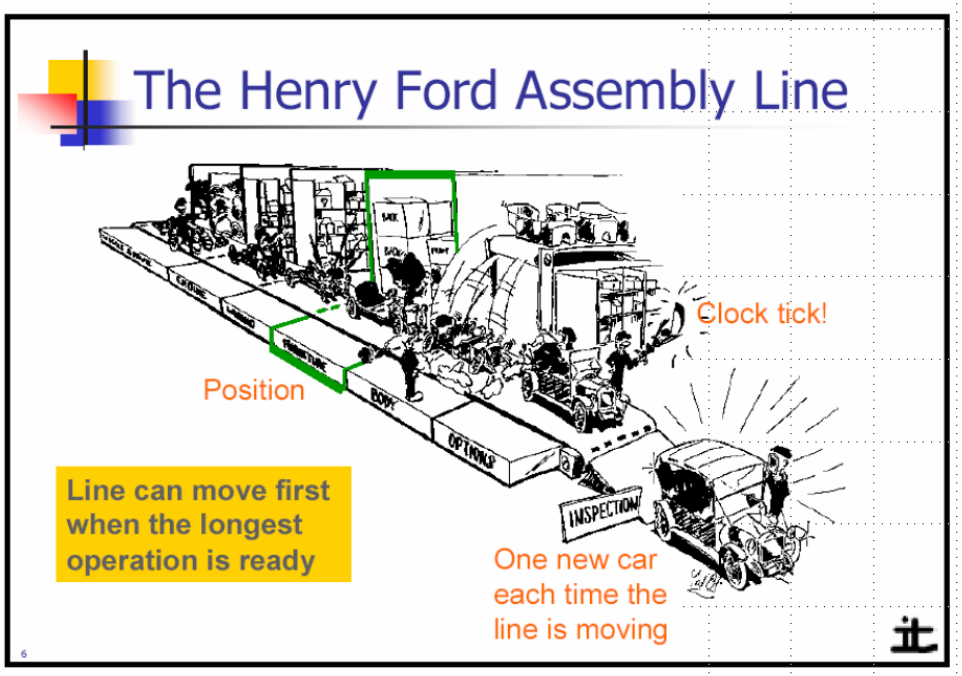


Mikroprogrammierung → Fließbandverarbeitung



Mikroprogrammierung → Fließbandverarbeitung

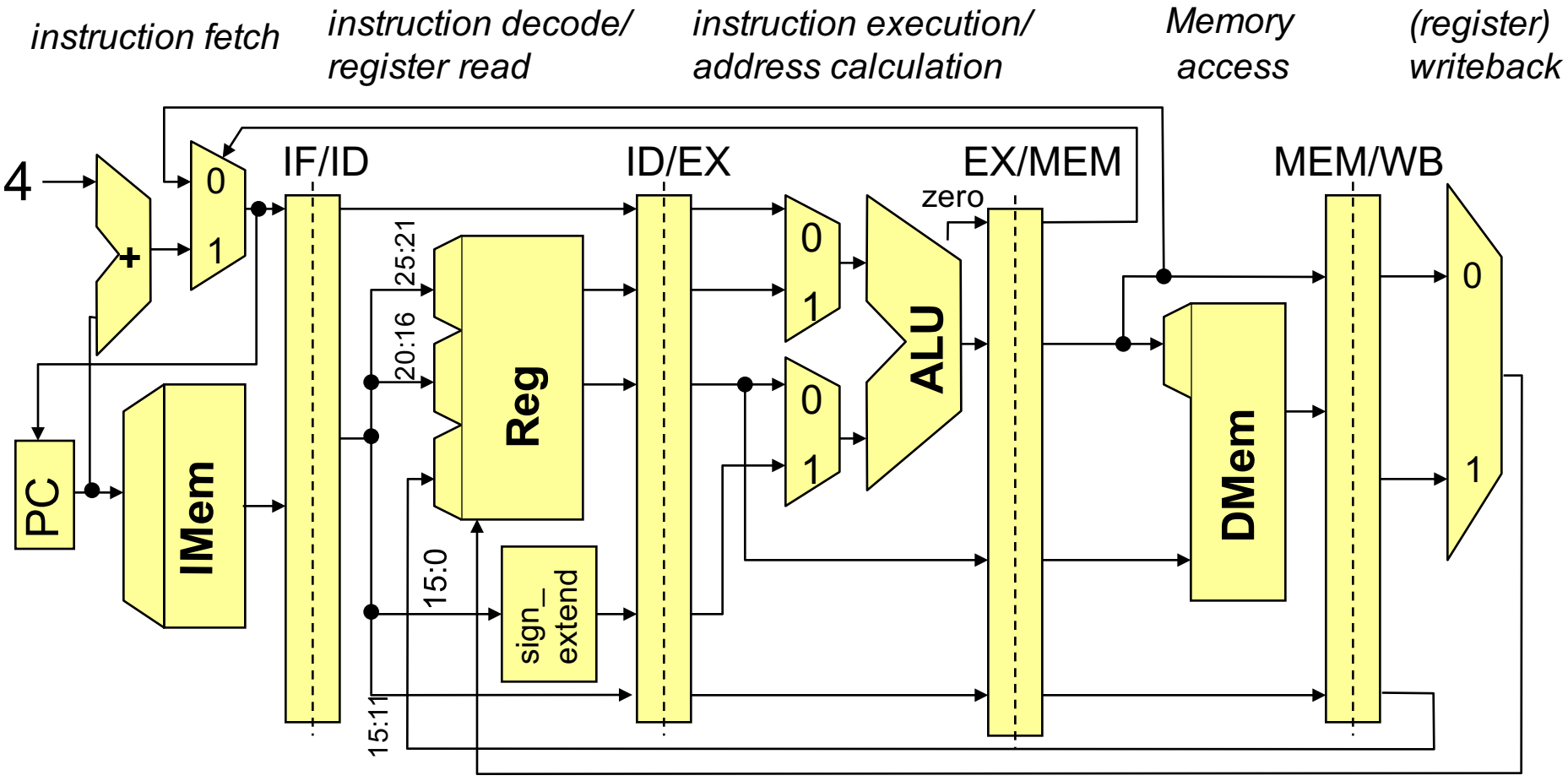




www.it.lth.se/courses/dsi/material/Lectures/Lecture6.pdf

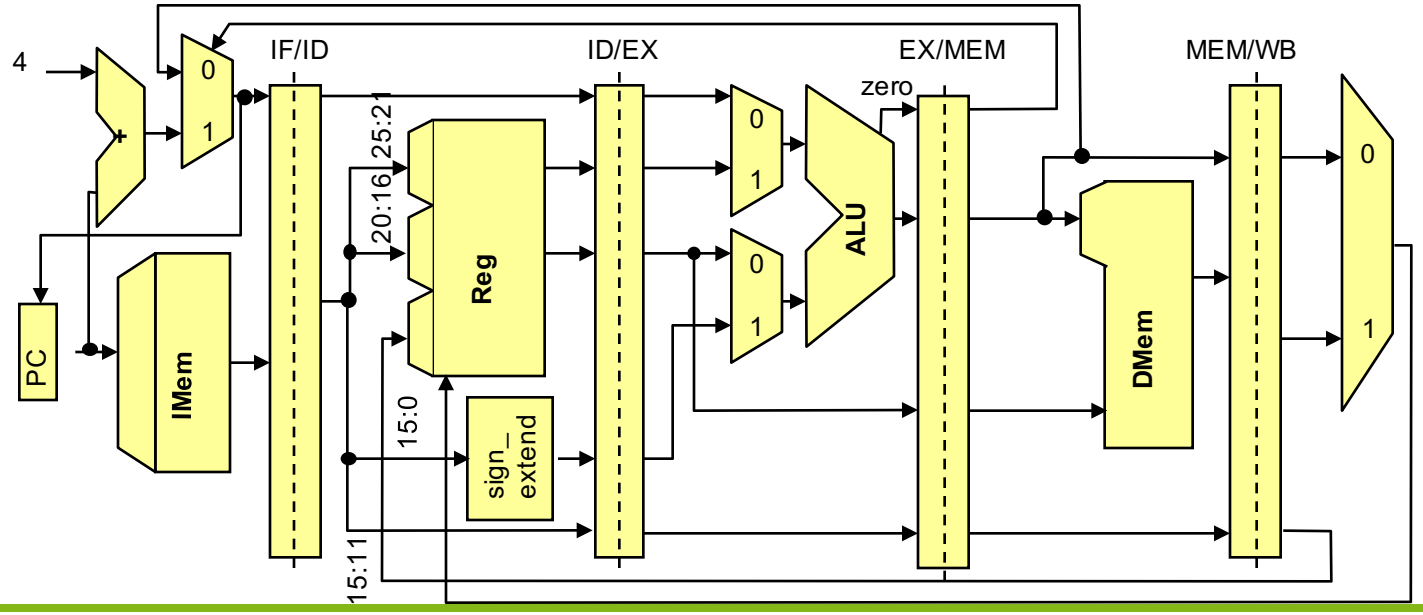
2.3.2 Fließbandverarbeitung

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.

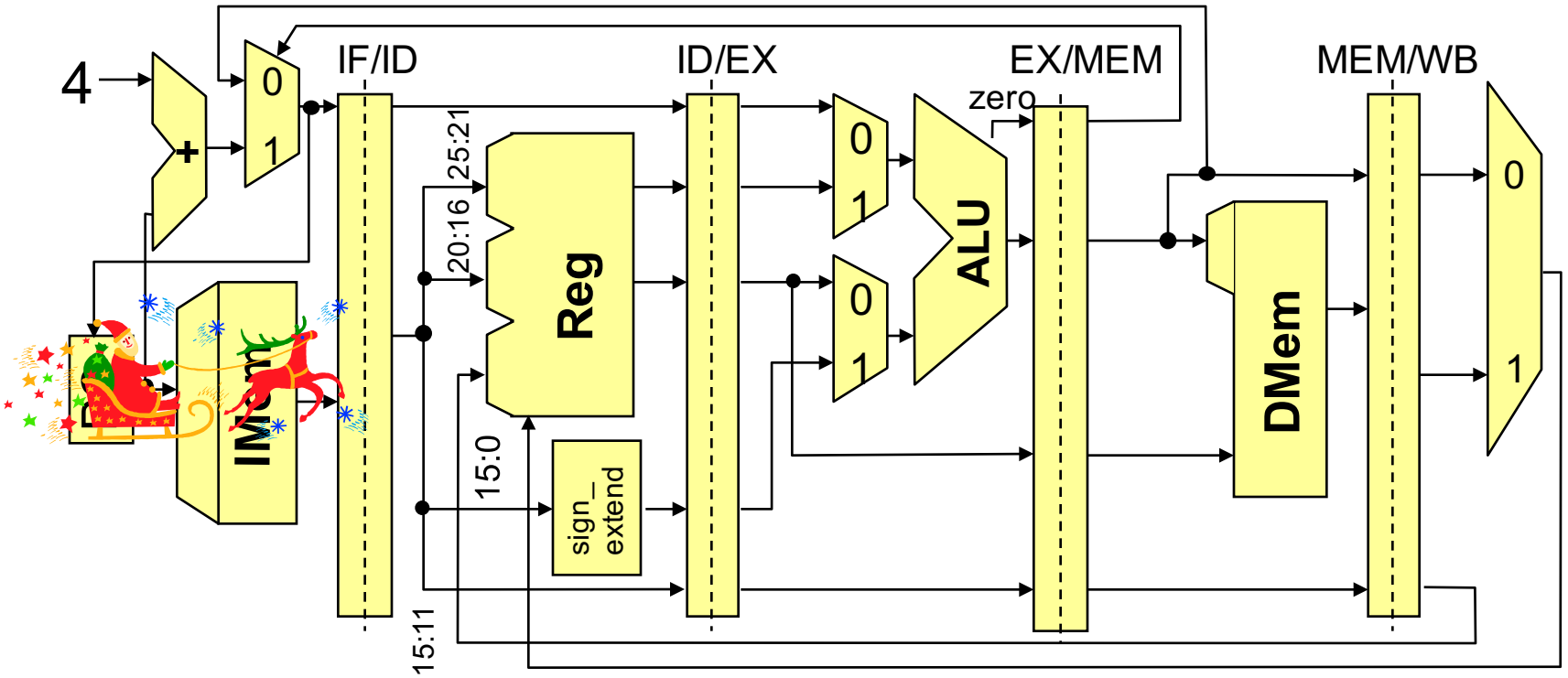


Steuer-
werk
nicht
dar-
gestellt

Aufgaben der einzelnen Phasen bzw. Stufen

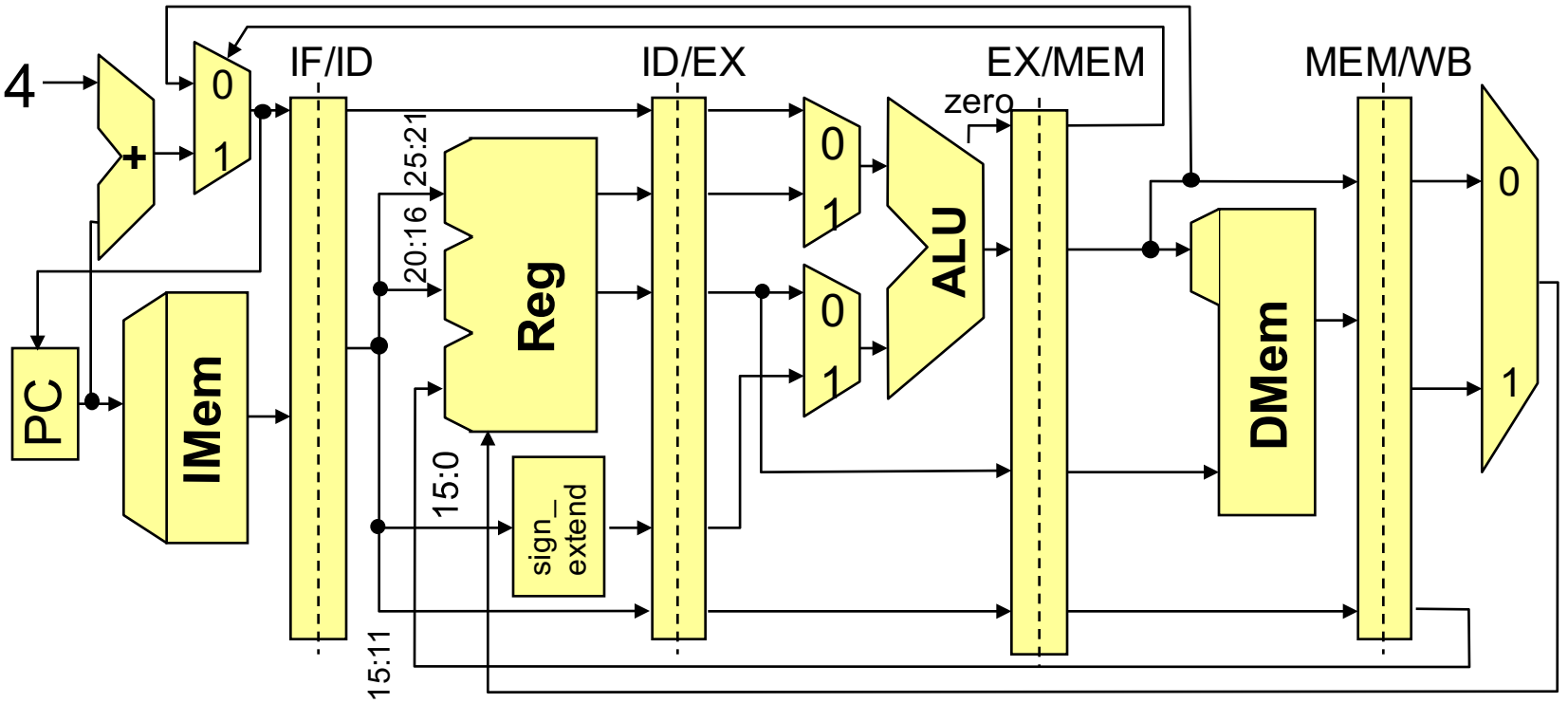
- **Befehlsholphase**
  Lesen des aktuellen Befehls; separater Speicher, zur Vermeidung von Konflikten mit Datenzugriffen (☞ Cache).
- **Dekodier- und Register-Lese-Phase**
  Lesen der Register möglich wegen fester Plätze für Nr.
- **Ausführungs- und Adressberechungsphase**
  Berechnung arithmetischer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

Idealer Fließbanddurchlauf



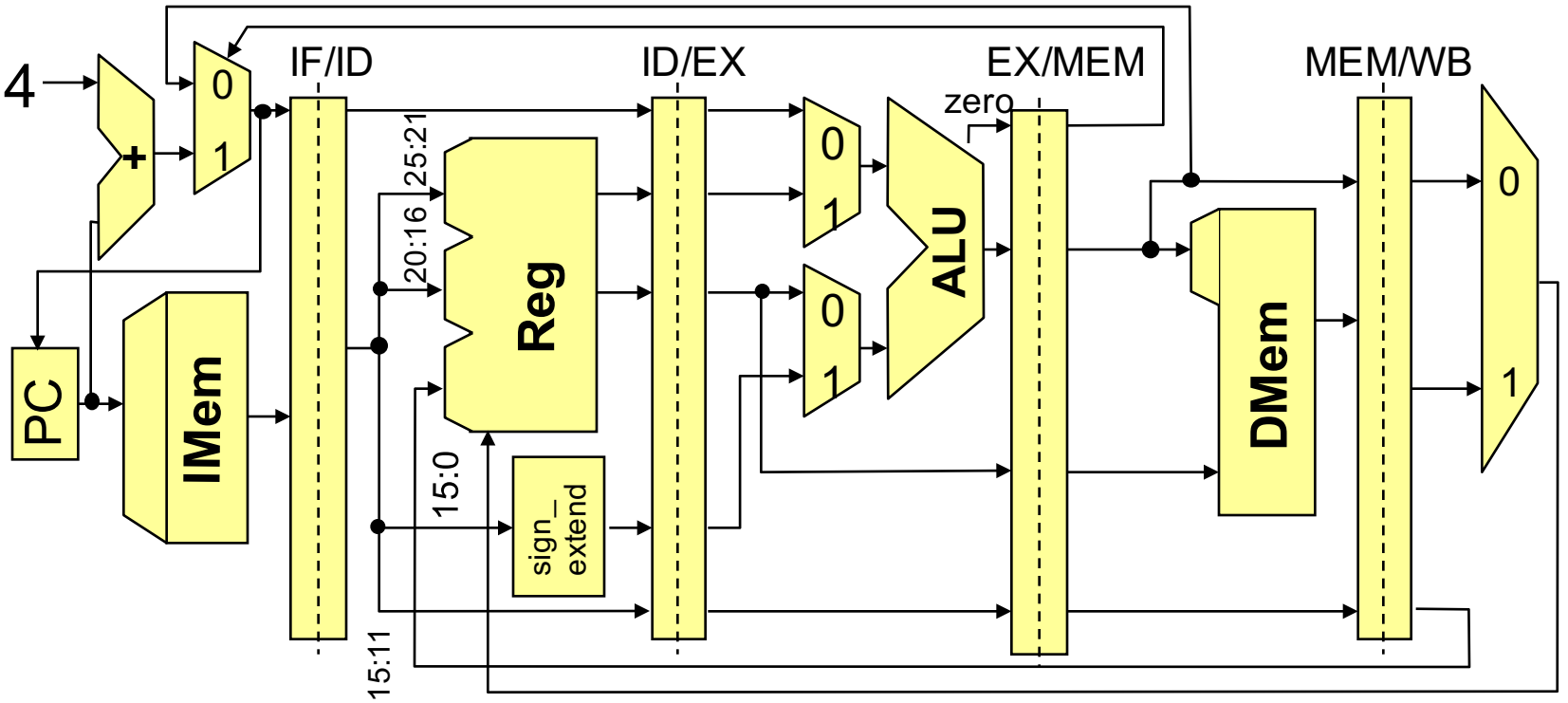
Zyklus 1				
Befehl 1				

Idealer Fließbanddurchlauf



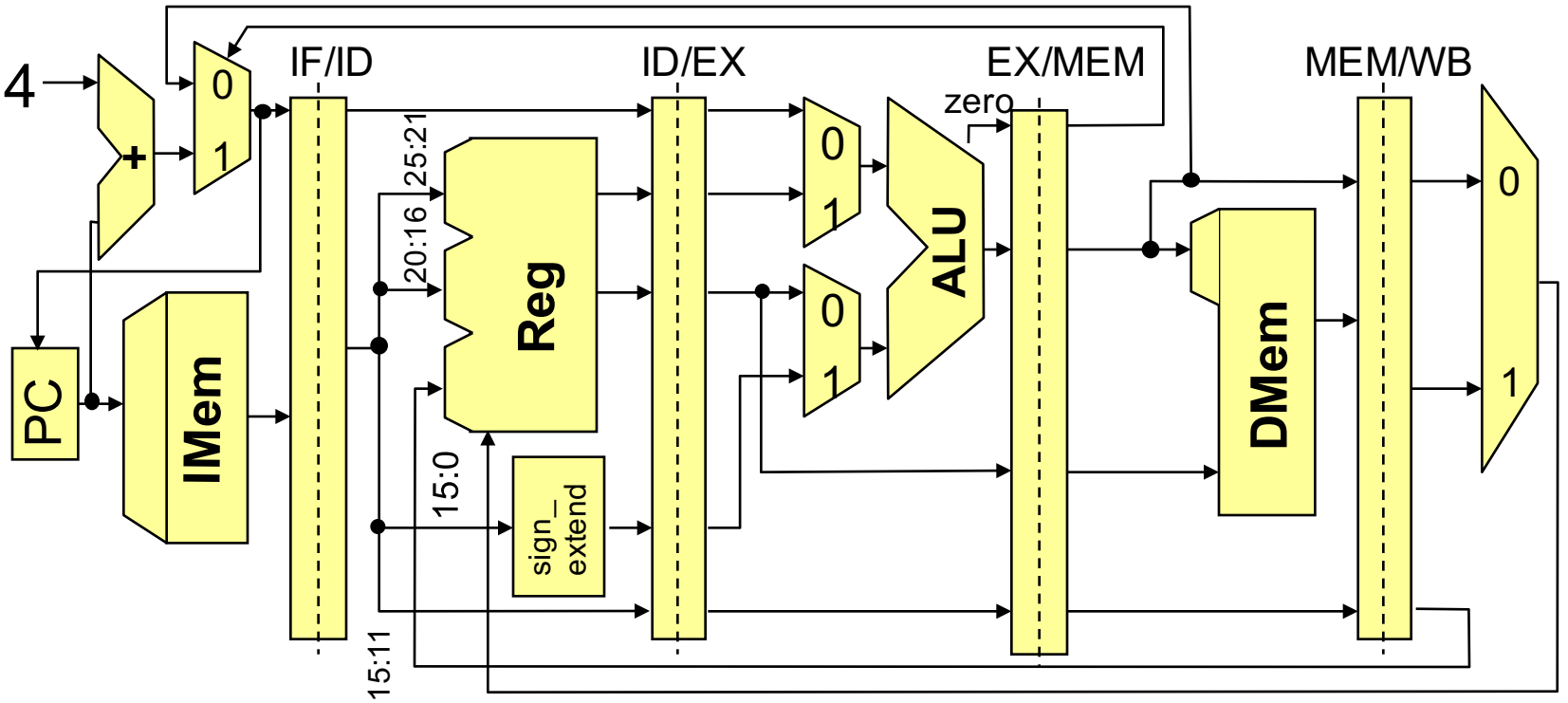
Zyklus 2				
Befehl 2	**Befehl 1**			

Idealer Fließbanddurchlauf



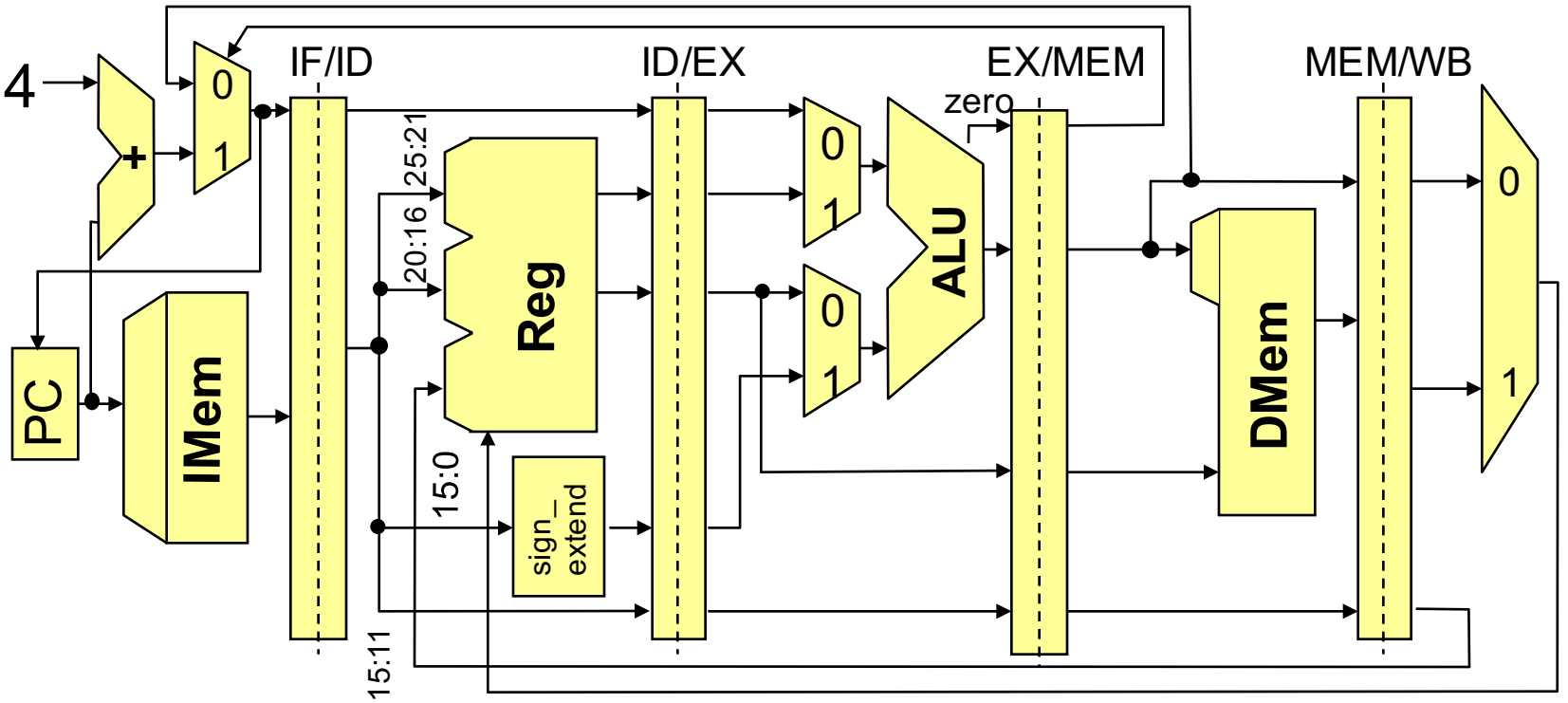
Zyklus 3				
Befehl 3	**Befehl 2**	**Befehl 1**		

Idealer Fließbanddurchlauf



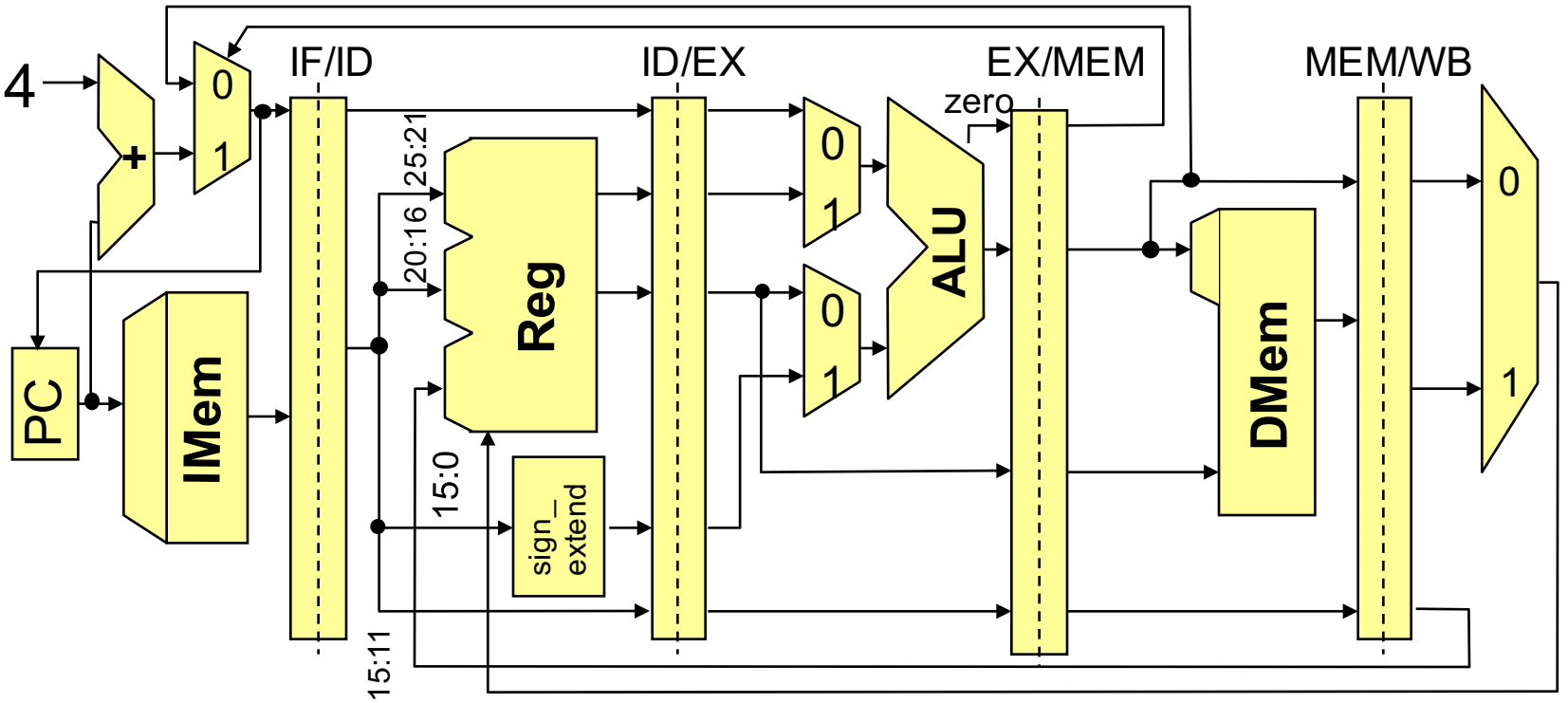
Zyklus 4				
Befehl 4	**Befehl 3**	**Befehl 2**	**Befehl 1**	

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**

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.

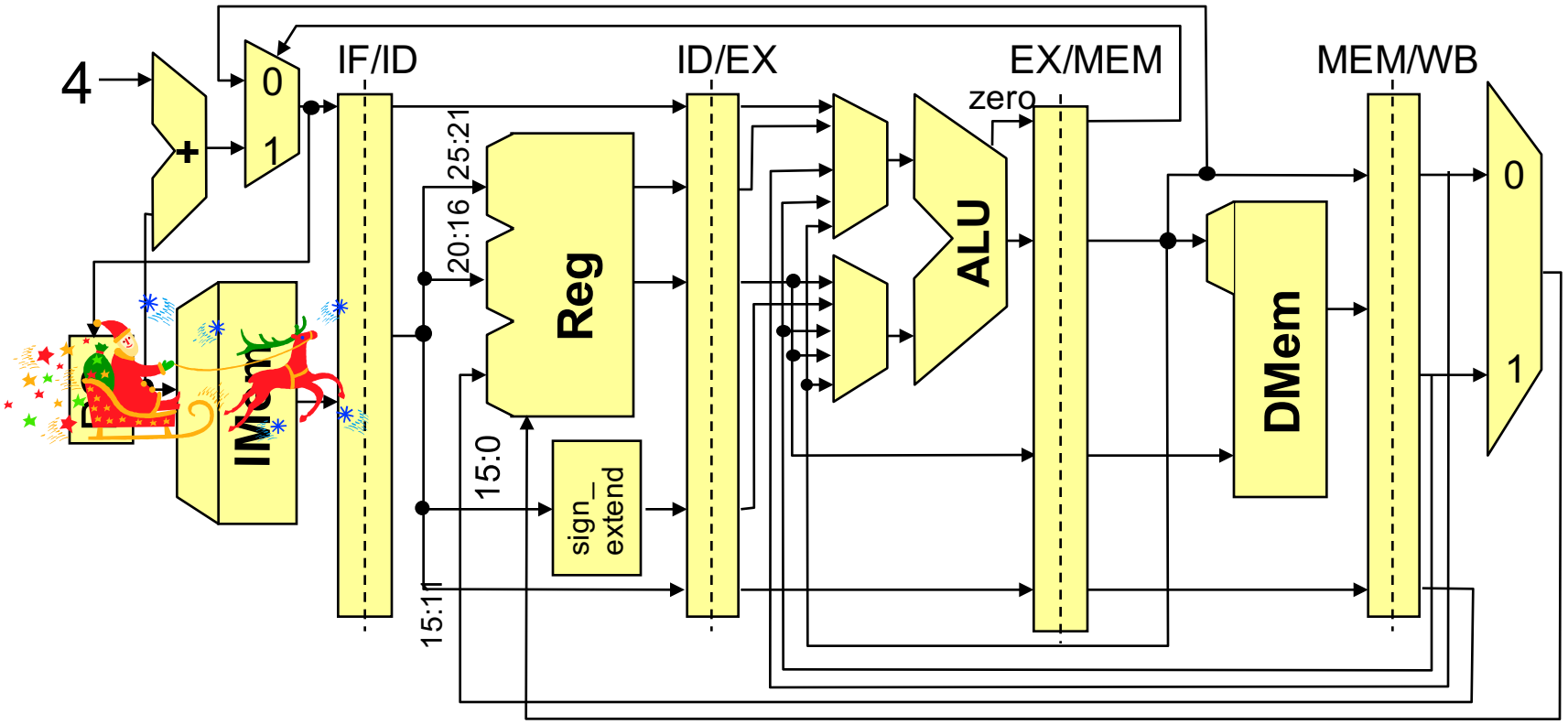
Beispiel:

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

Voneinander ausgabeabhängig.

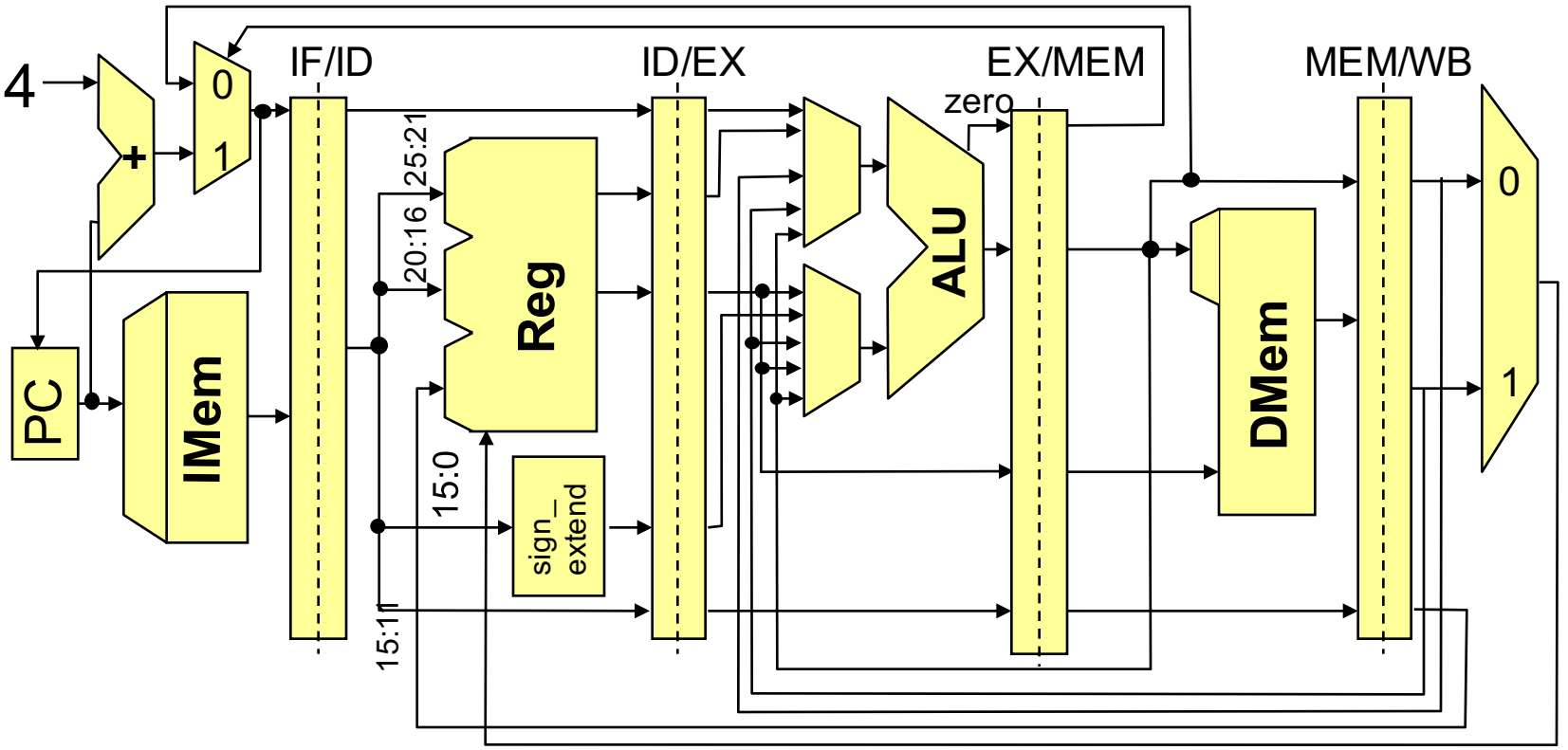
Diese Art der Abhängigkeit heißt (bei Hennessy und anderen) *write after write* - (oder WAW-) Abhängigkeit.

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



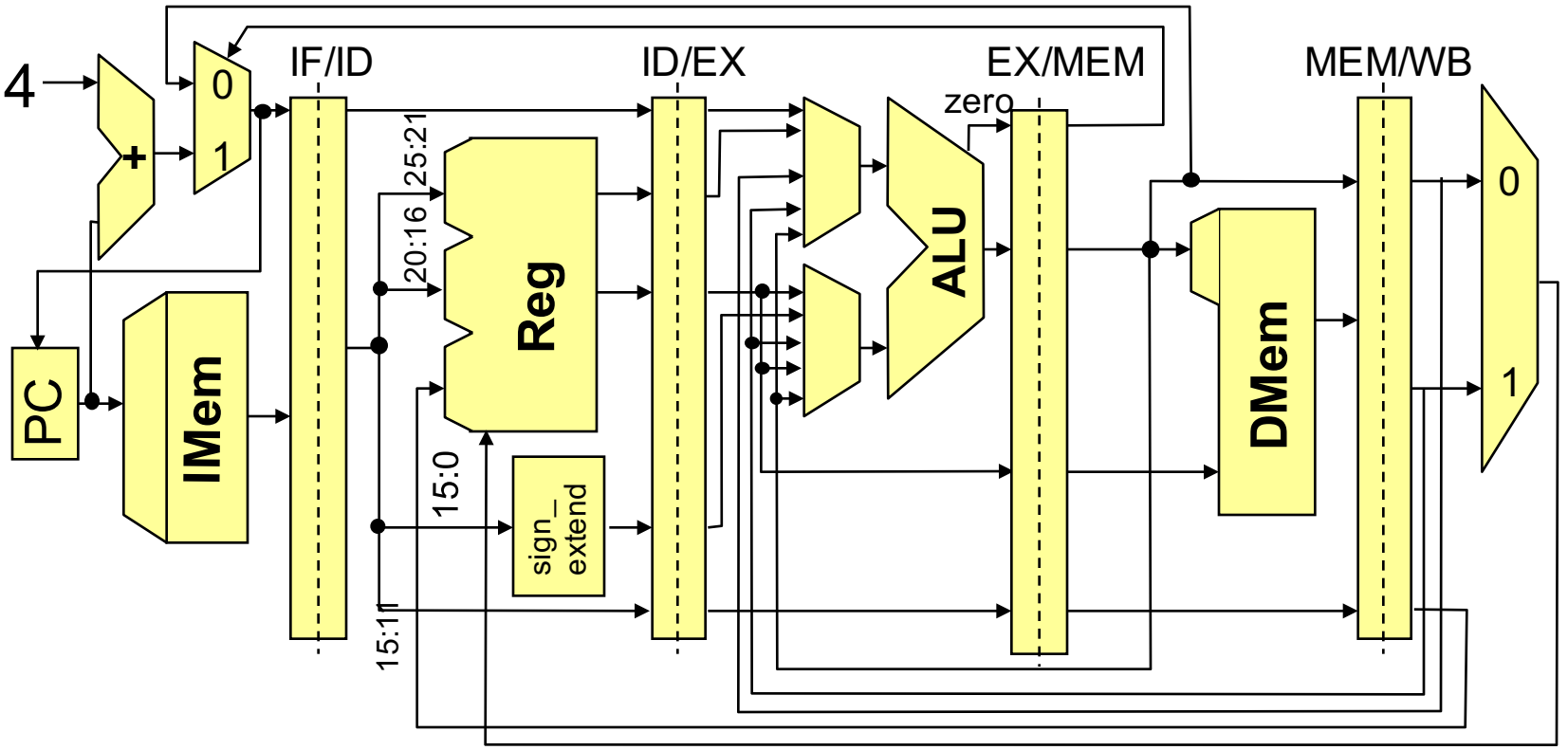
Zyklus 1				
add $1,$2,$3				

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



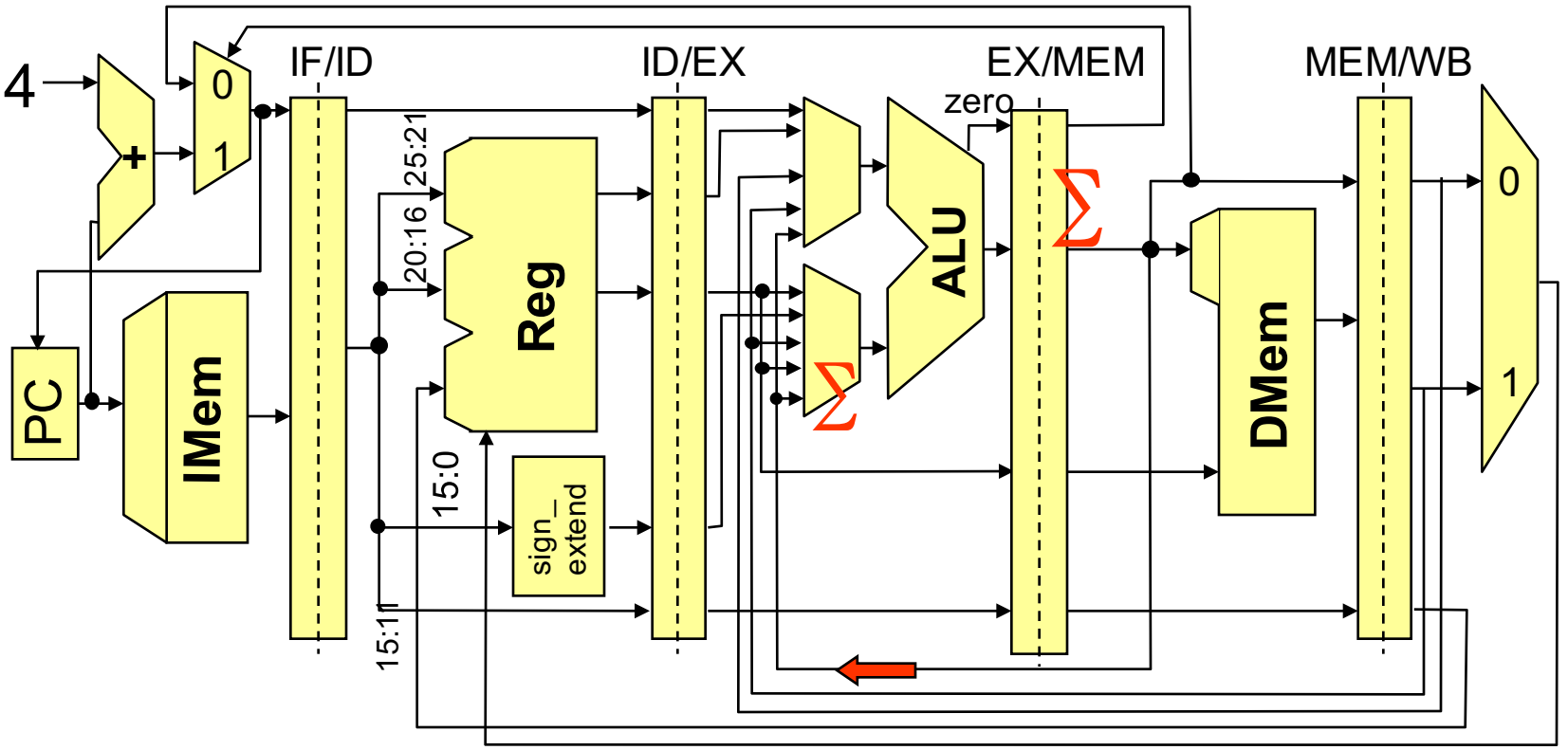
Zyklus 2				
sub $4,$5,$1	**add $1,$2,$3**			

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



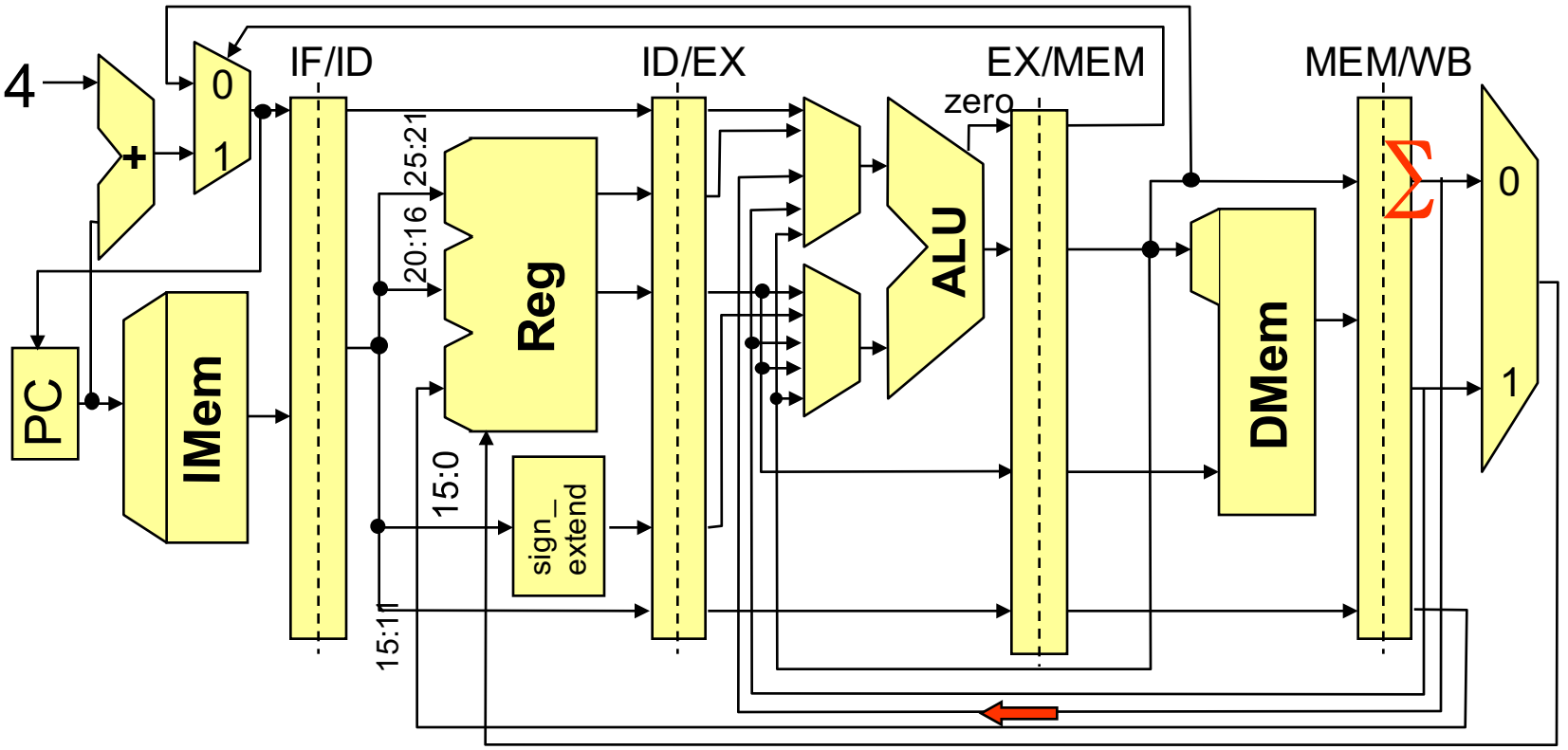
Zyklus 3				
and $6,$1,$7	**sub $4,$5,$1**	**add $1,$2,$3**		

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



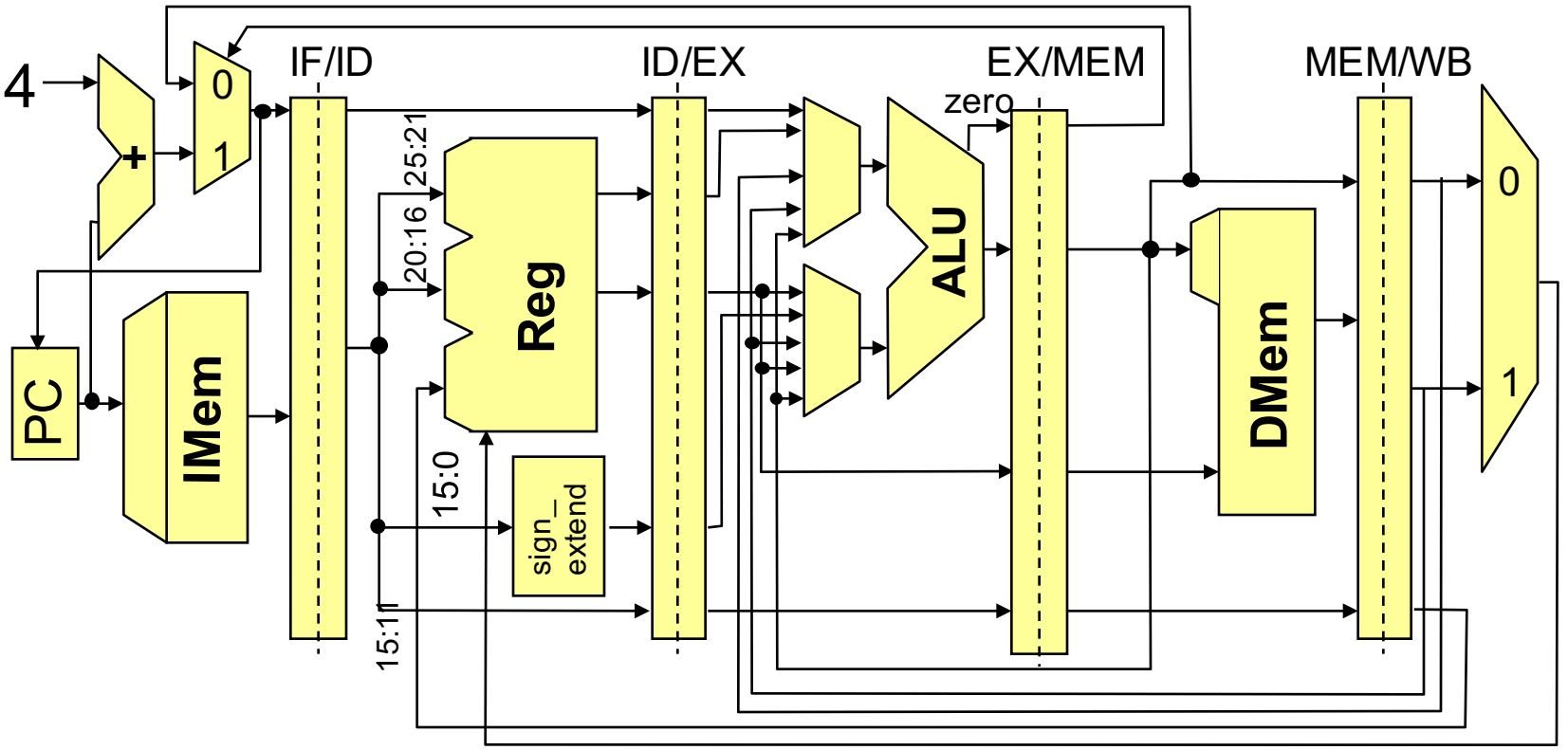
Zyklus 4				
or $8,$1,$9	**and $6,$1,$7**	**sub $4,$5,$1**	**add $1,$2,$3**	

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



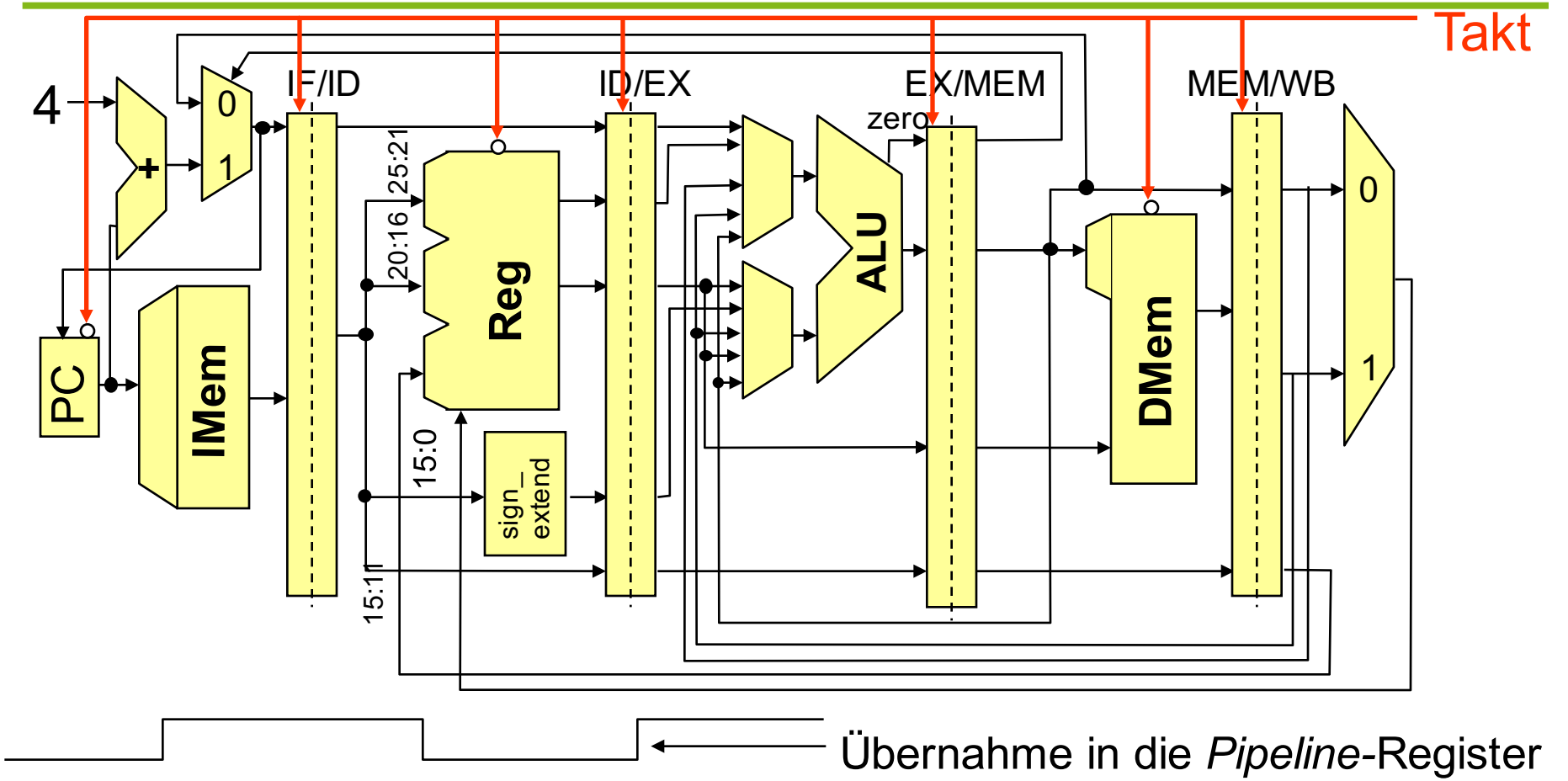
Zyklus 5				
xor $10,$1,$11	or $8,$1,$9	and $6,$1,$7	sub $4,$5,$1	add $1,$2,$3

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



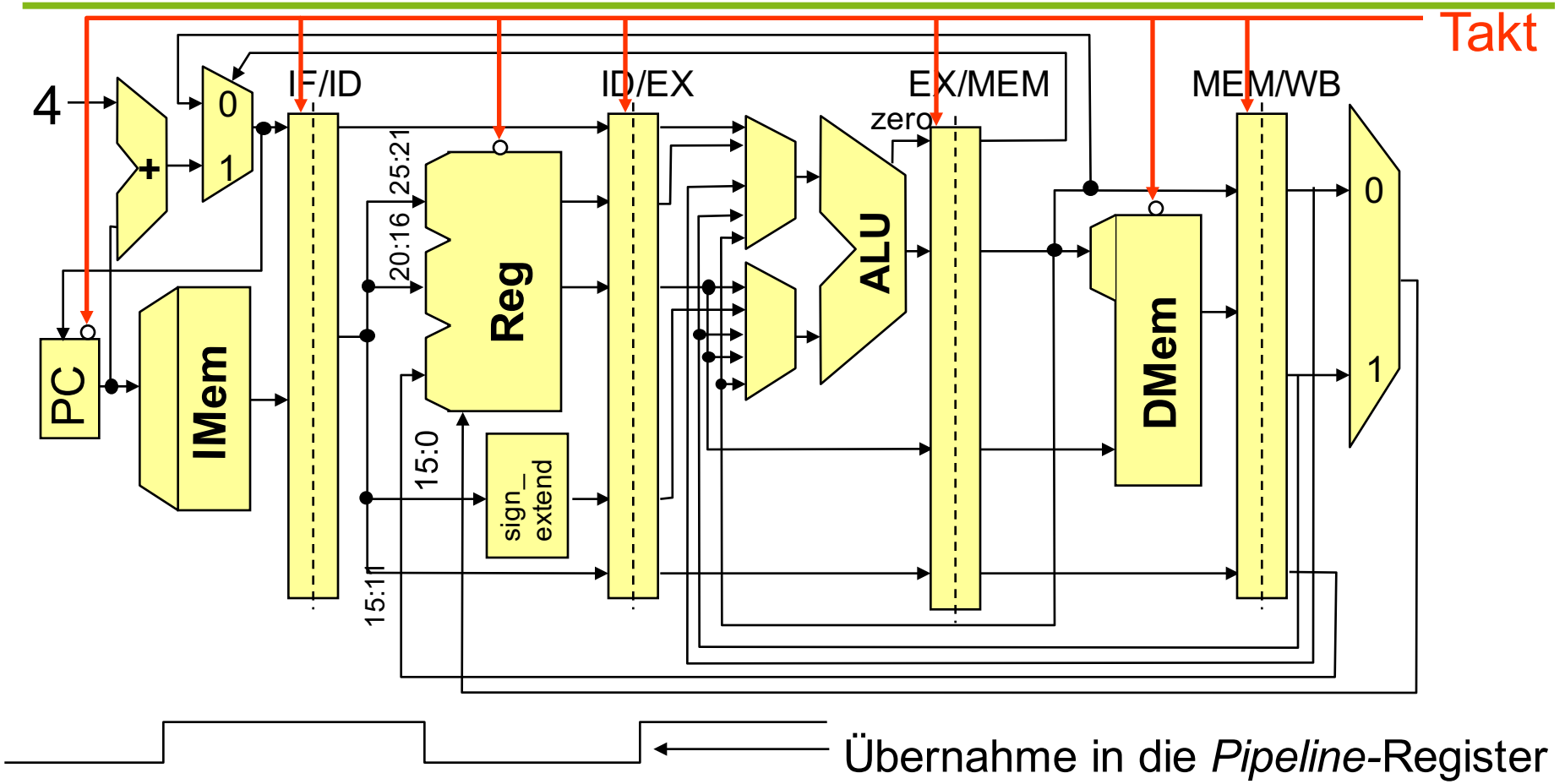
Zyklus 6				
?	xor $10,$1,$11	or $8,$1,$9	and $6,$1,$7	sub $4,$5,$1

Taktung zur Behandlung des *data hazards* bei `or`



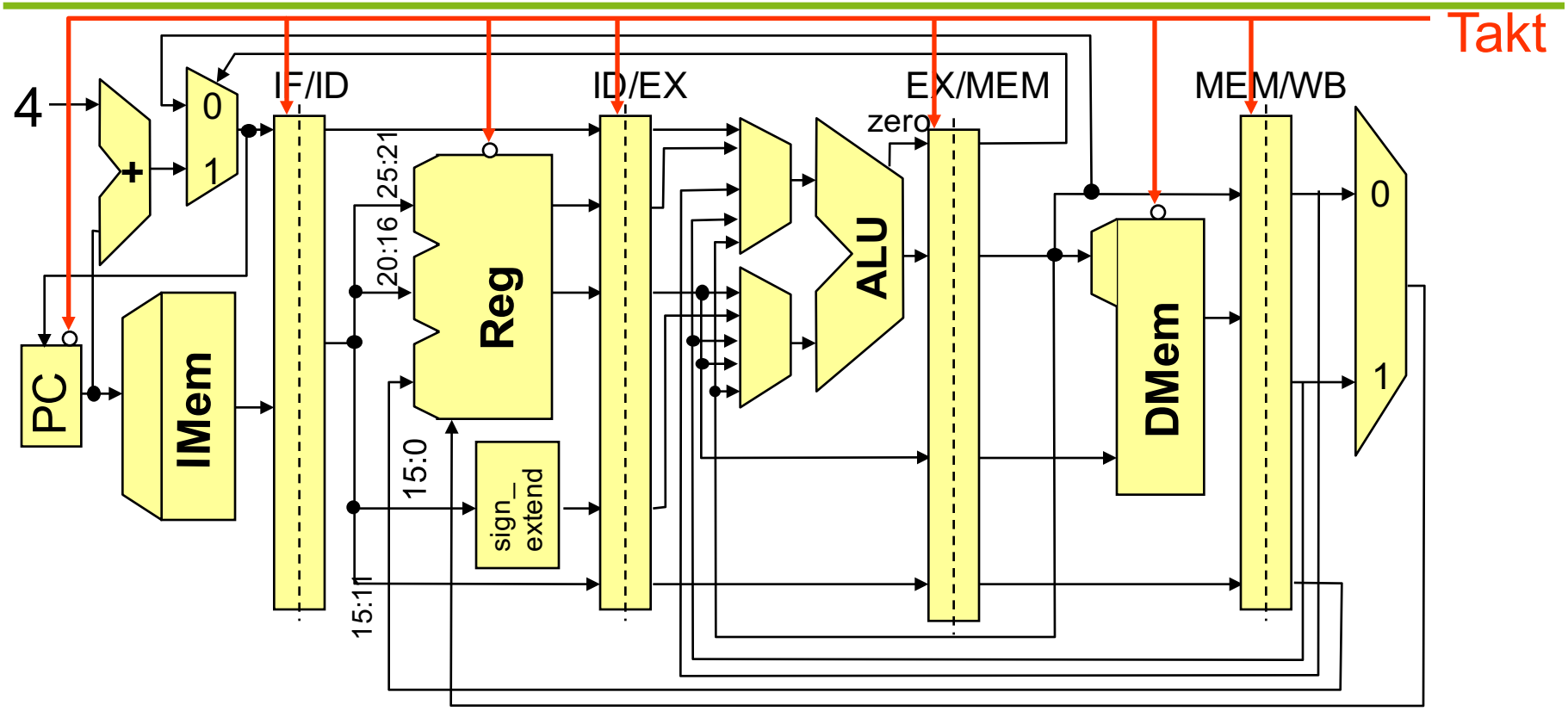
Zyklus 1				
add $1,$2,$3				

Taktung zur Behandlung des *data hazards* bei `or`



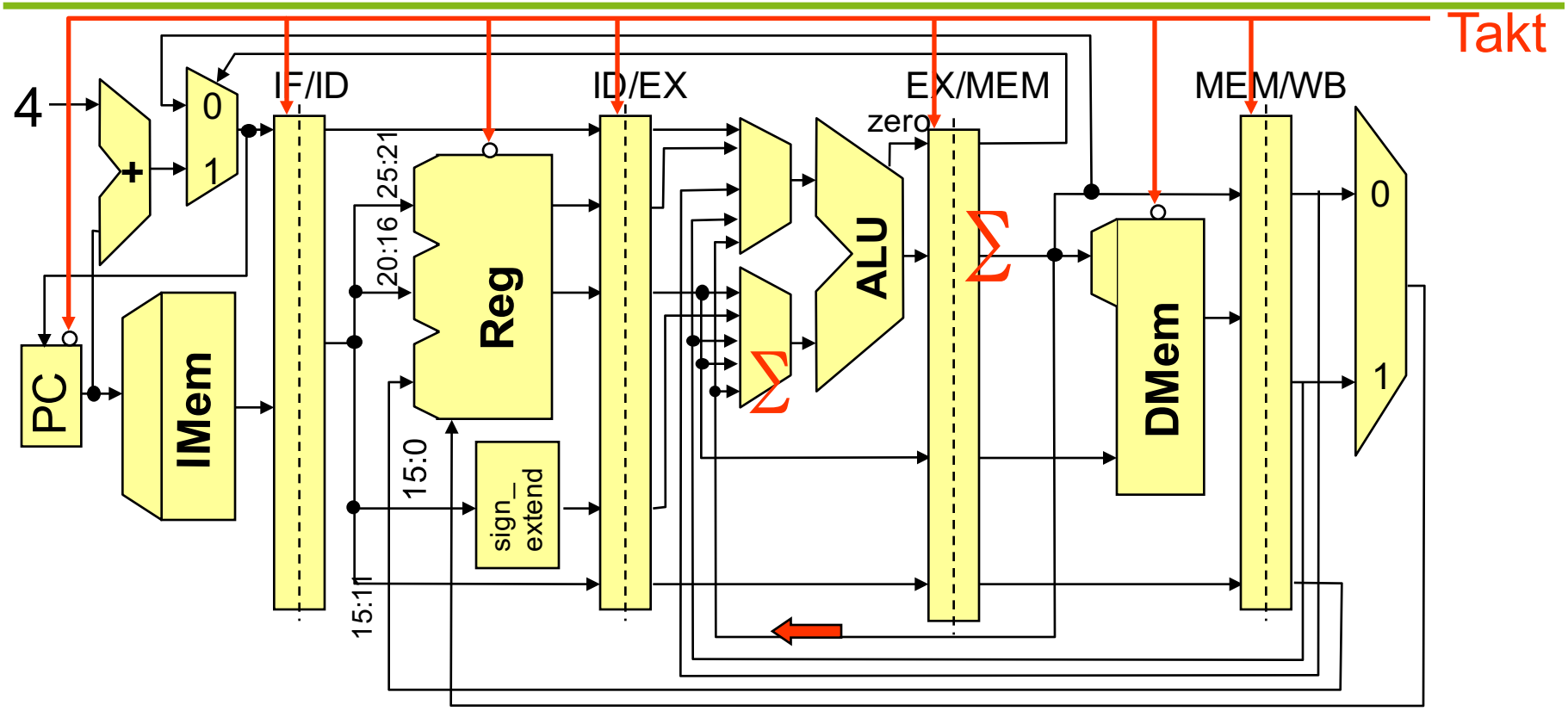
Zyklus 2				
sub $4,$5,$1	**add $1,$2,$3**			

Taktung zur Behandlung des *data hazards* bei `or`



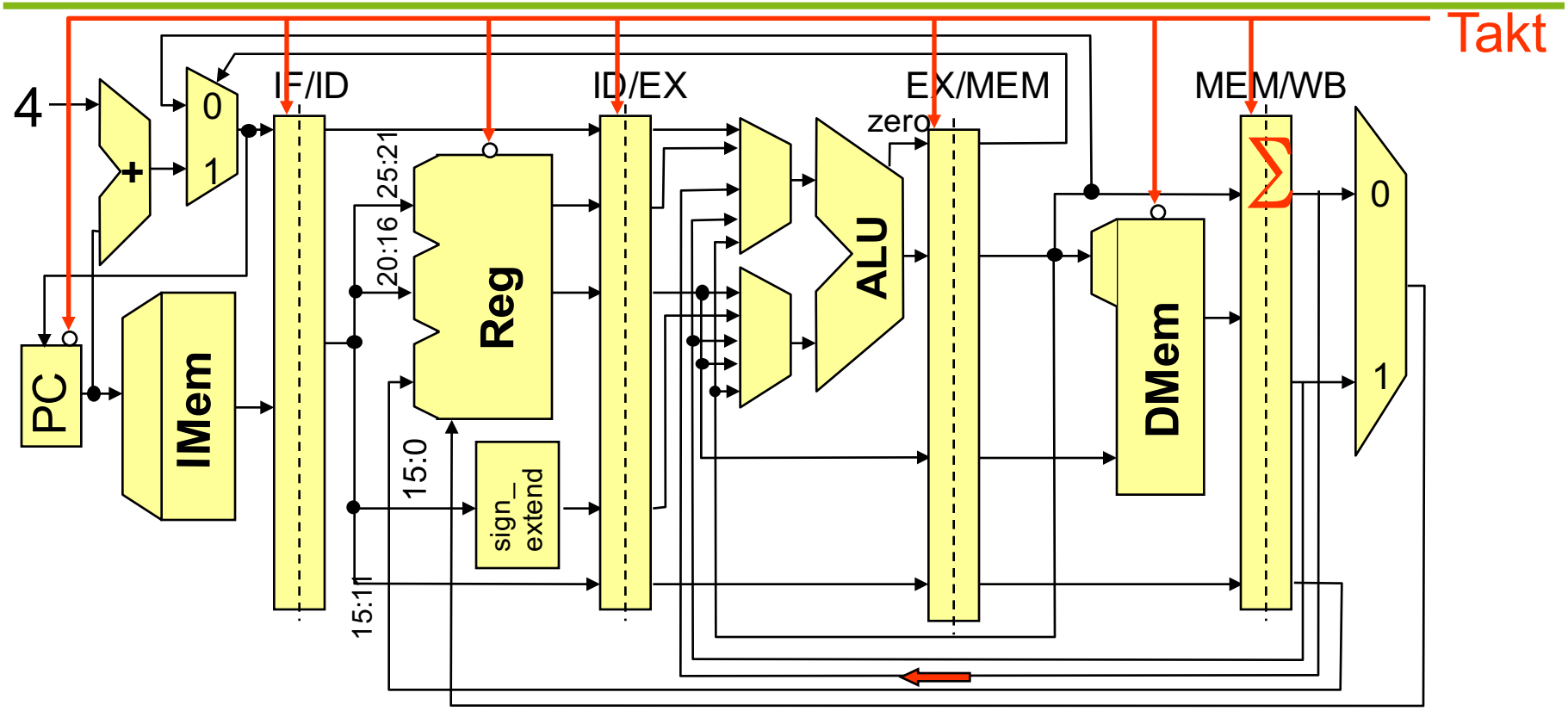
Zyklus 3				
and $6,$1,$7	**sub $4,$5,$1**	**add $1,$2,$3**		

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`

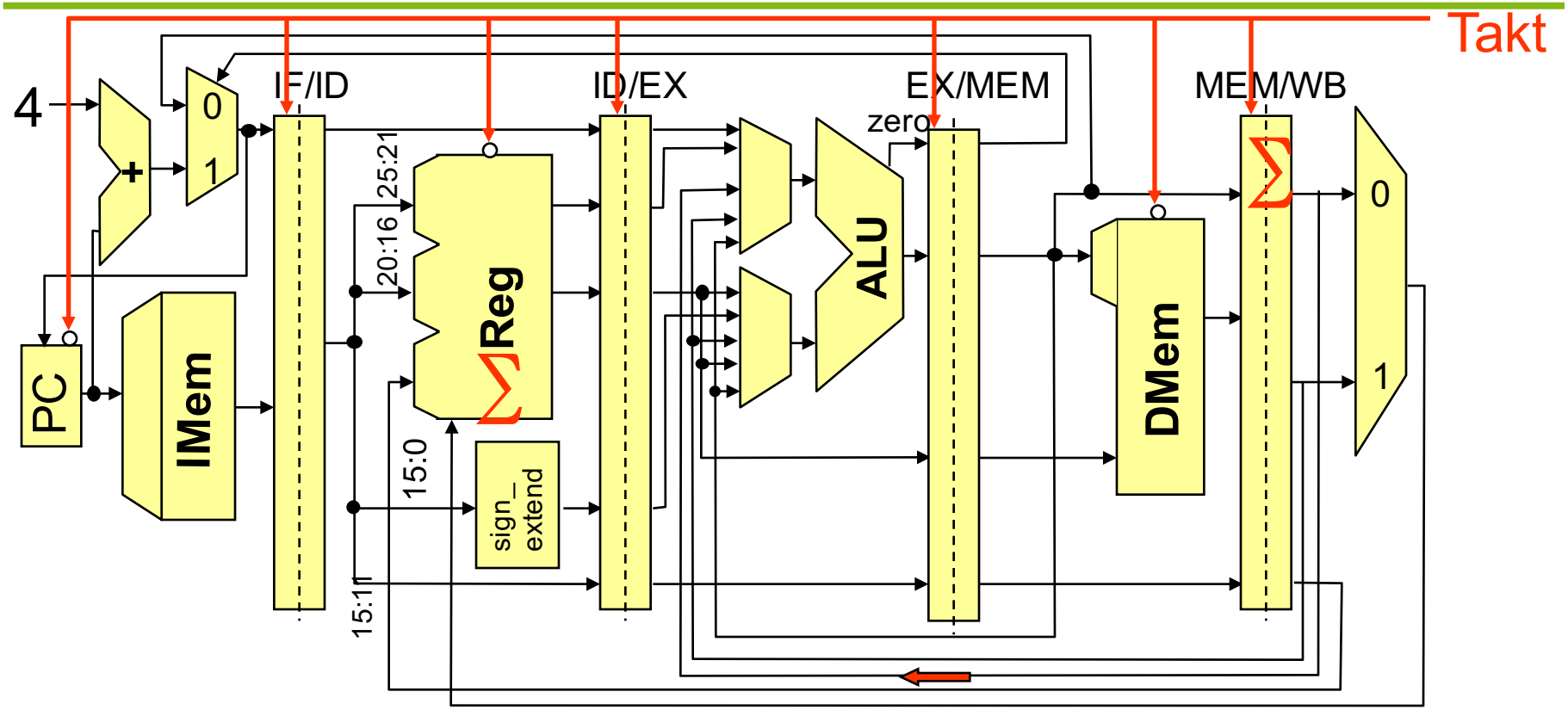


Zyklus 4

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`

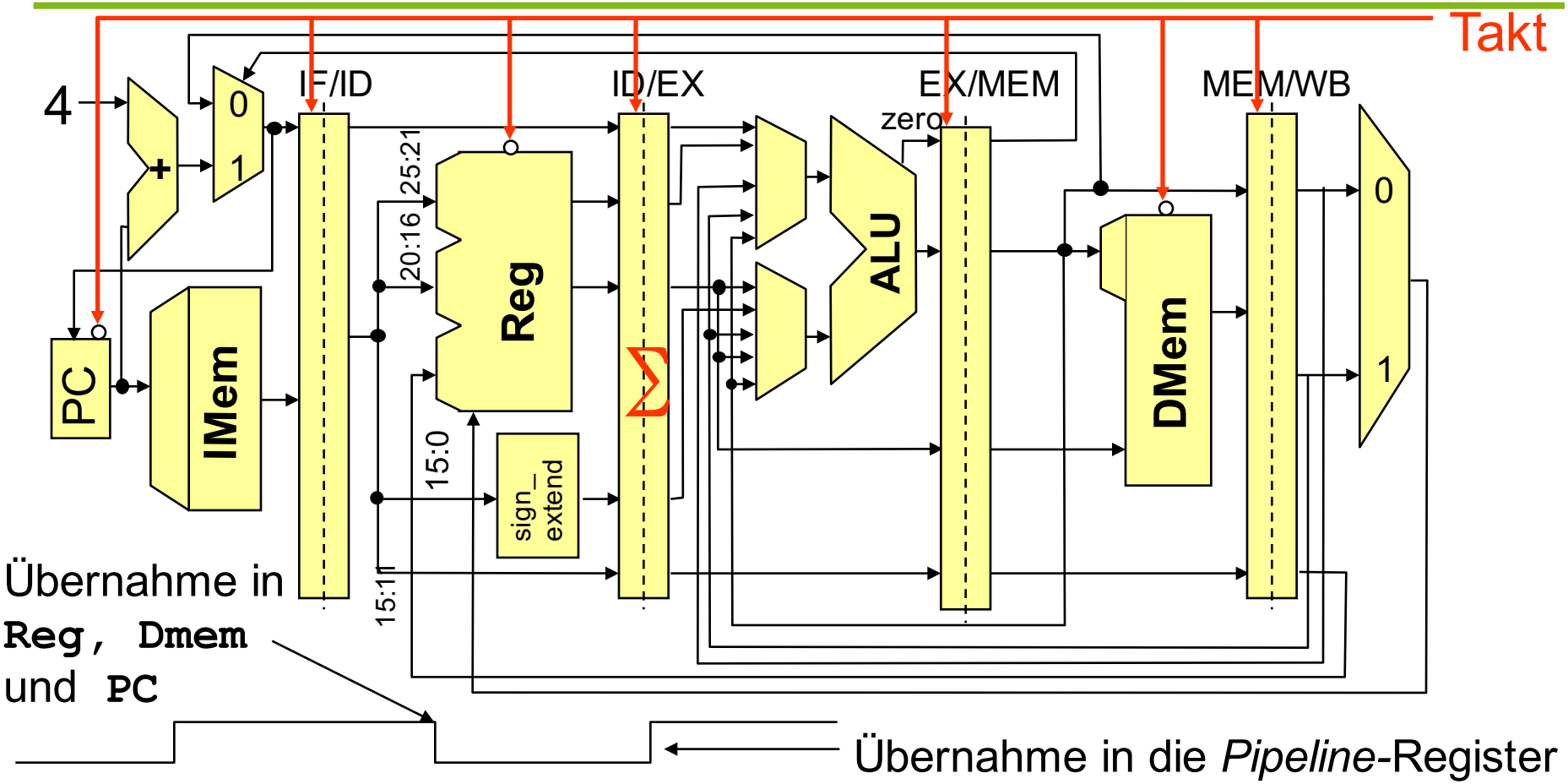


Zyklus 4

Übernahme in die *Pipeline*-Register

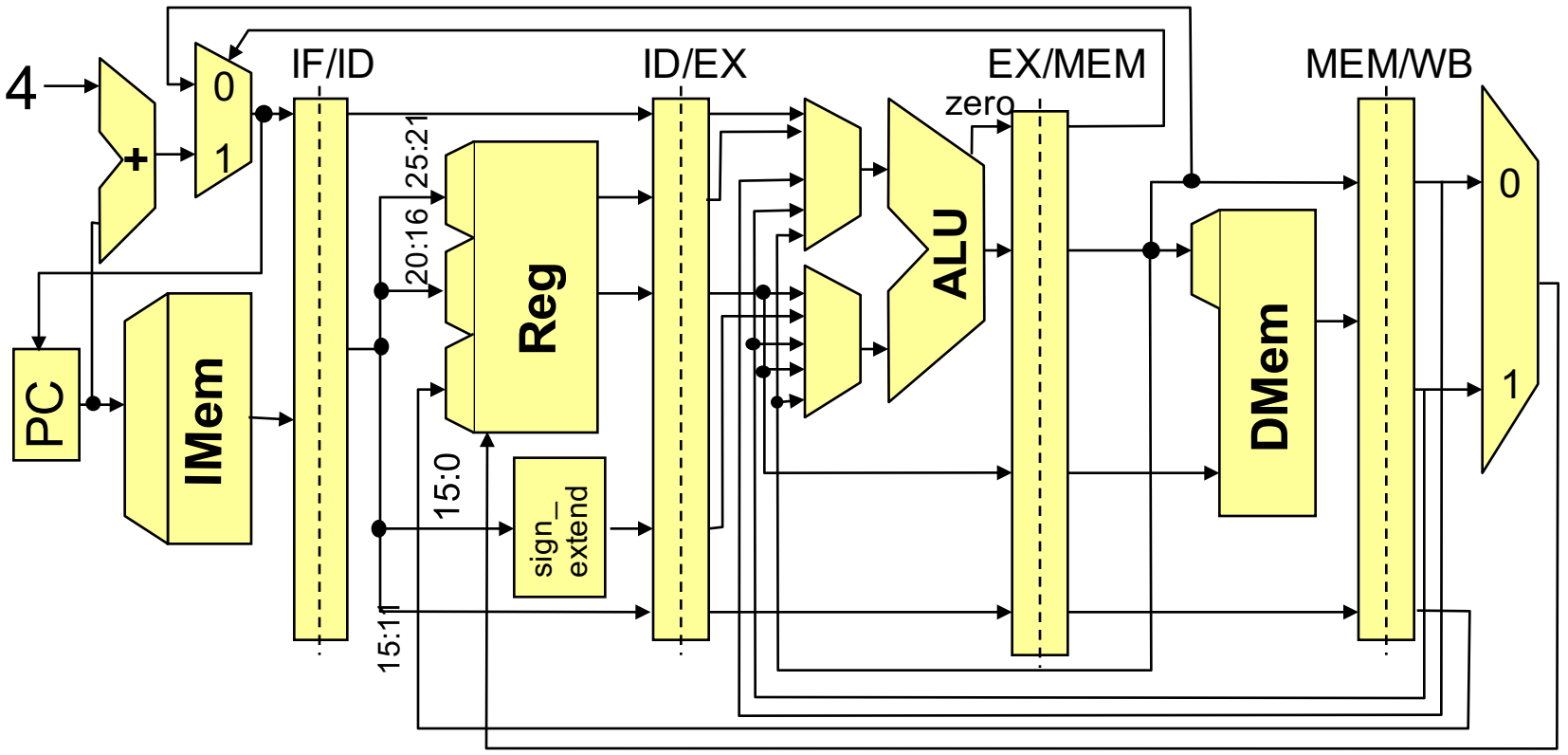
Zyklus 5				
xor $10,$1,$11	**or $8,$1,$9**	**and $6,$1,$7**	**sub $4,$5,$1**	**add $1,$2,$3**

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**

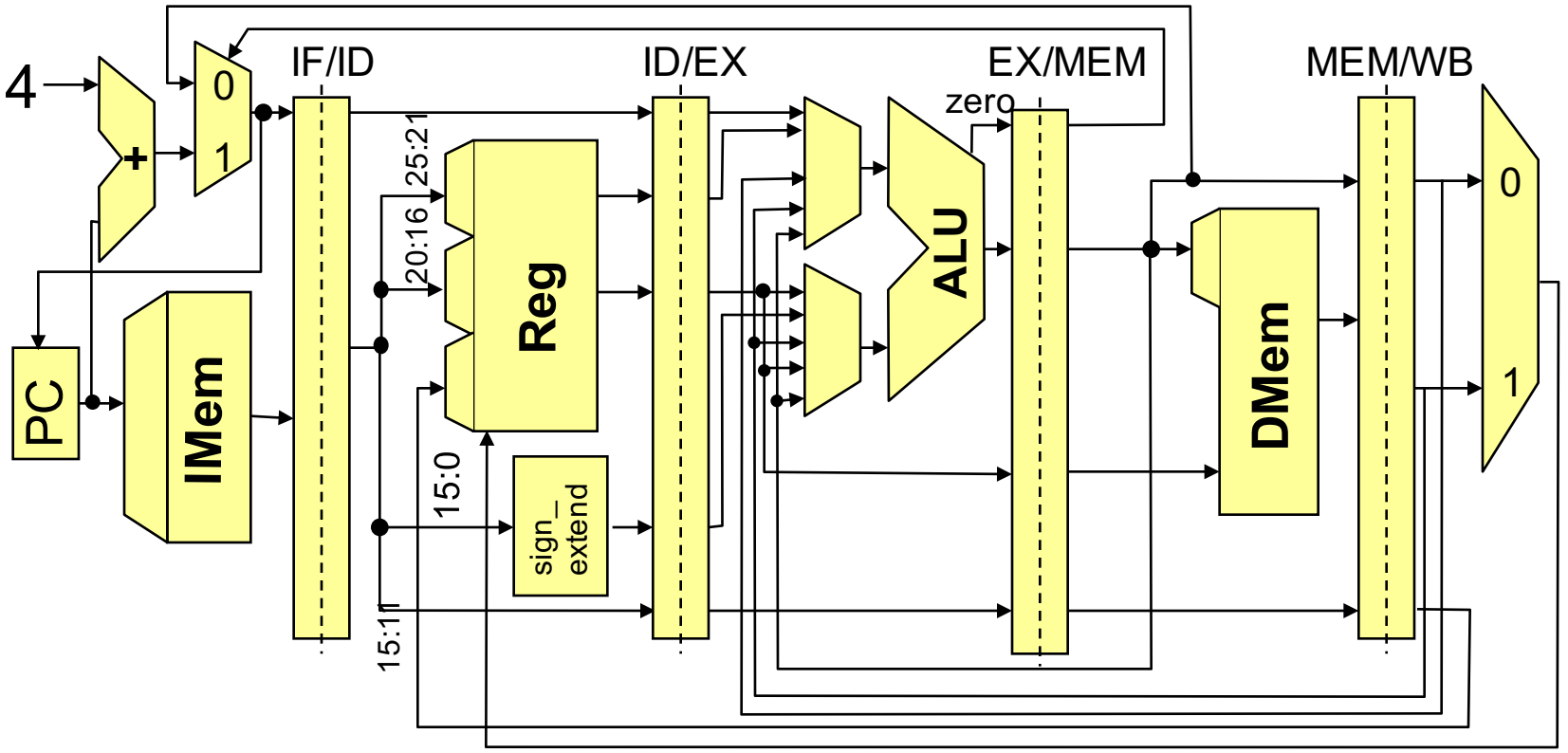
Alle *data hazards* durch Bypässe behandelbar?



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

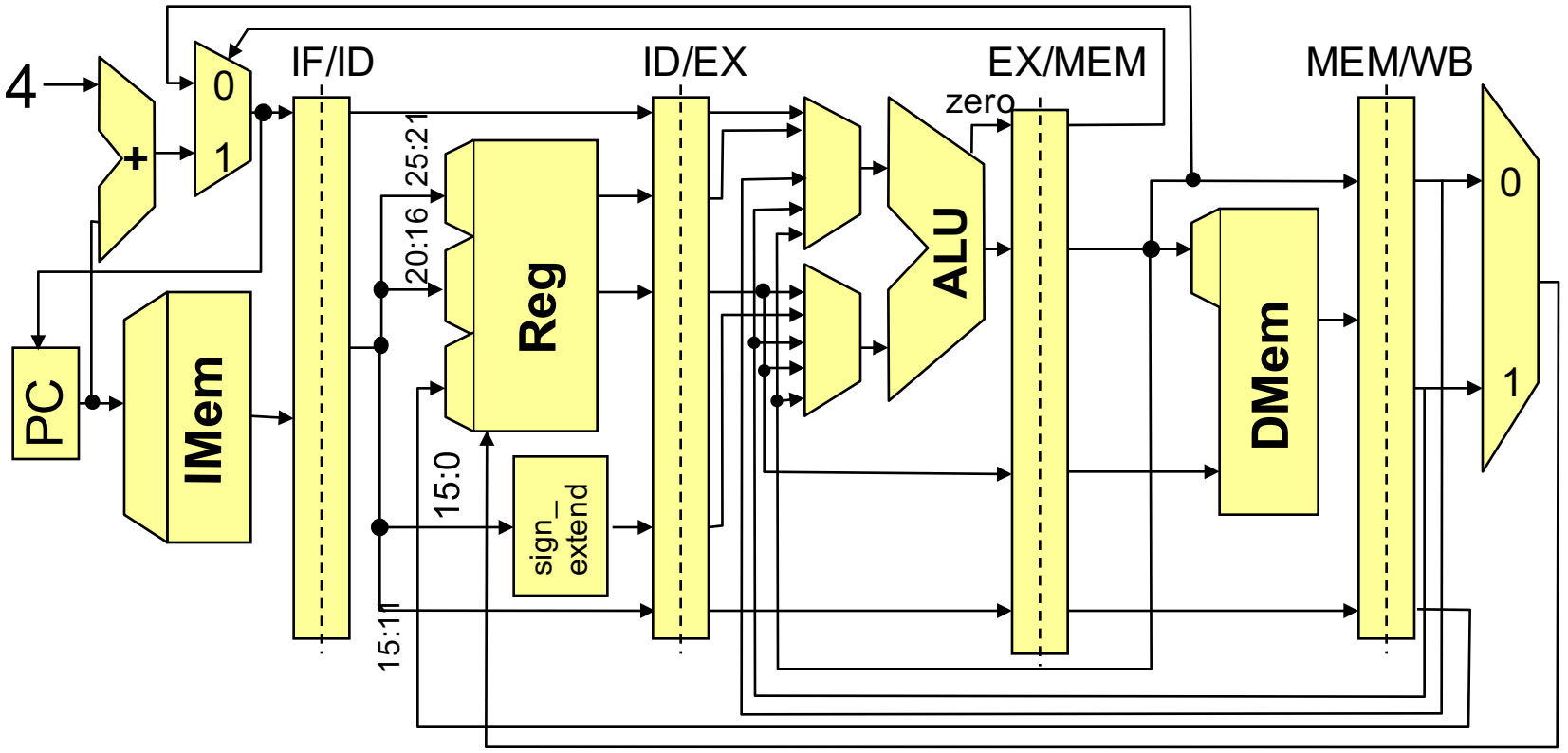
Speicherwort wird am Ende des Zyklus 4 gespeichert, steht für Differenz noch nicht bereit.

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



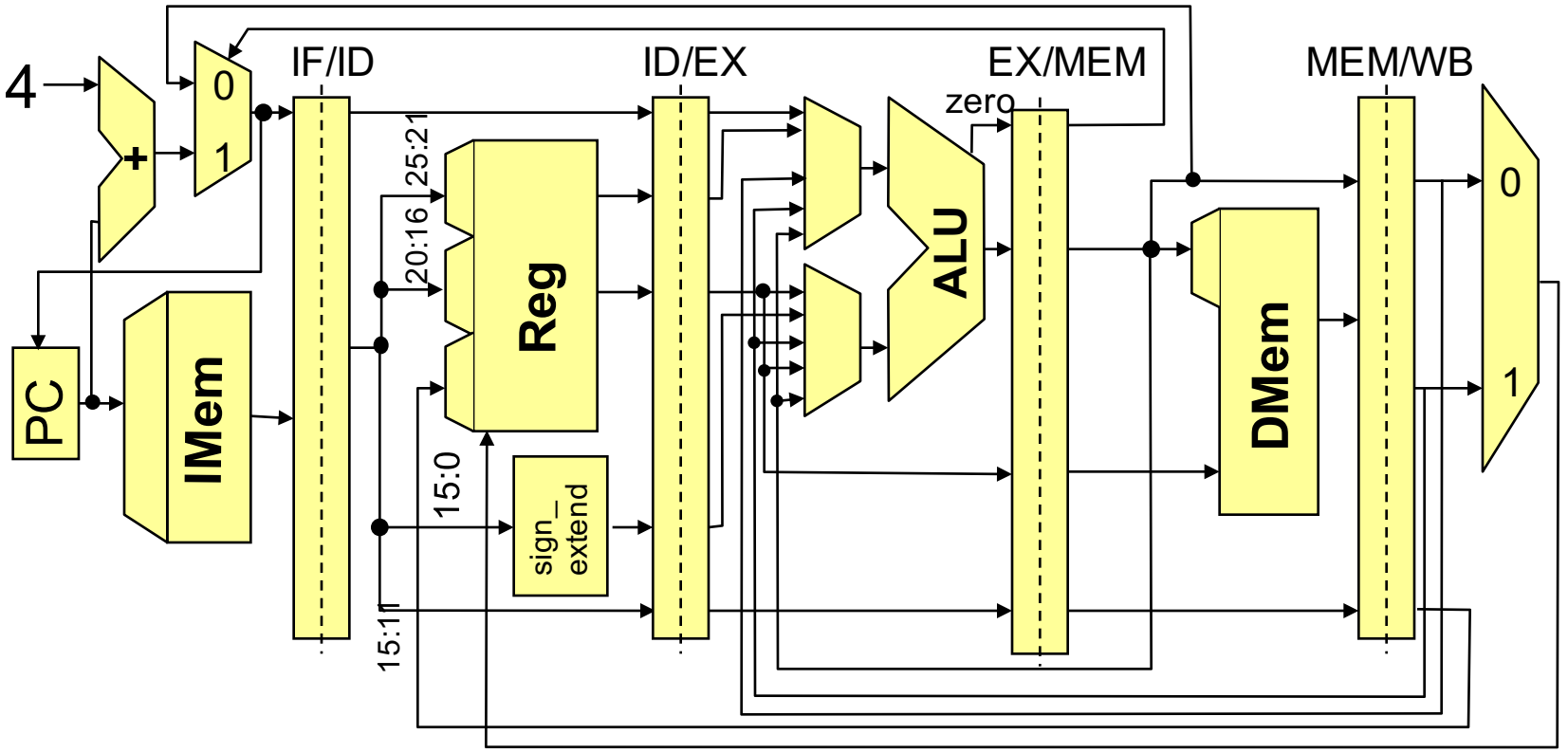
Zyklus 1				
lw $1,0($2)				

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



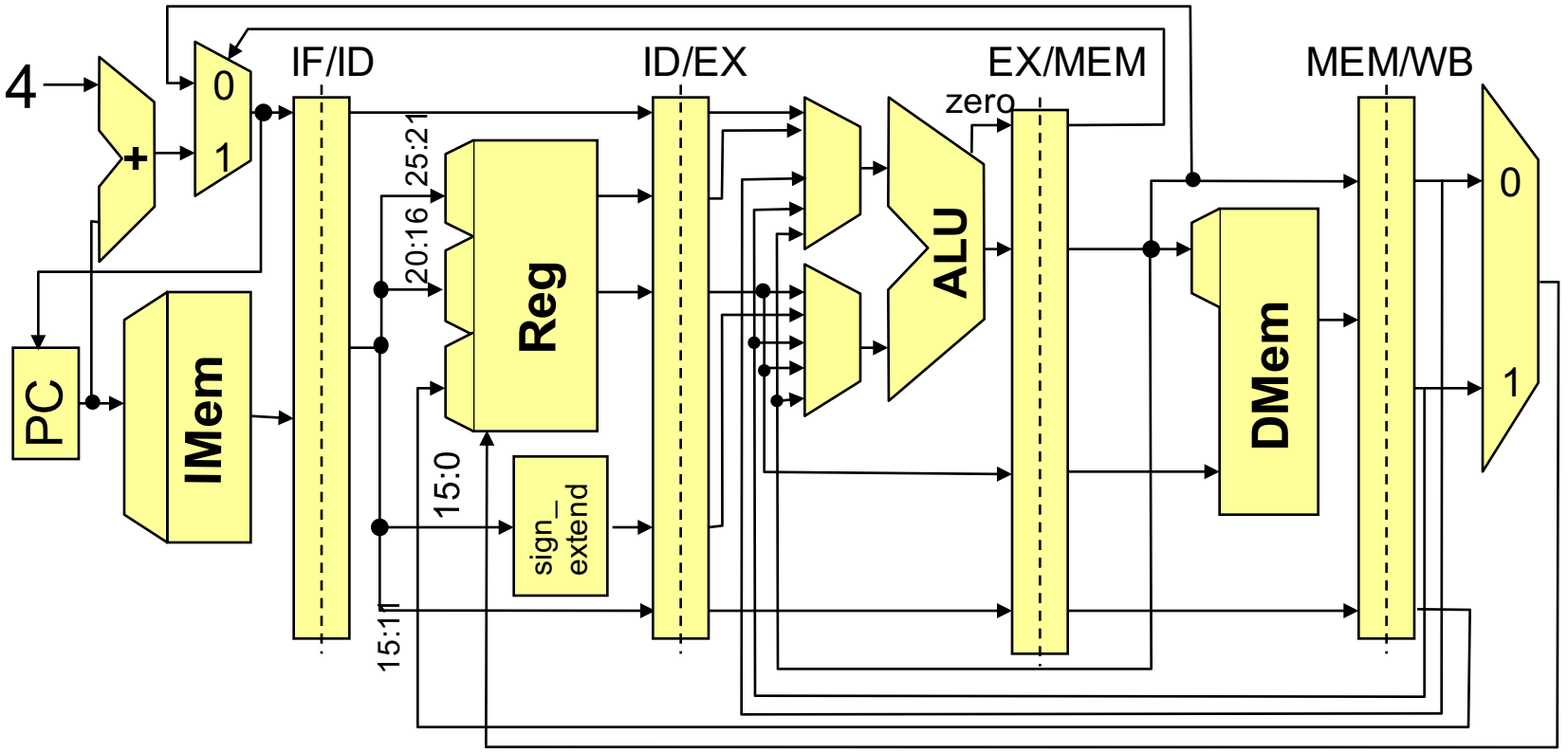
Zyklus 2				
sub $4,$5,$1	**lw $1,0($2)**			

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



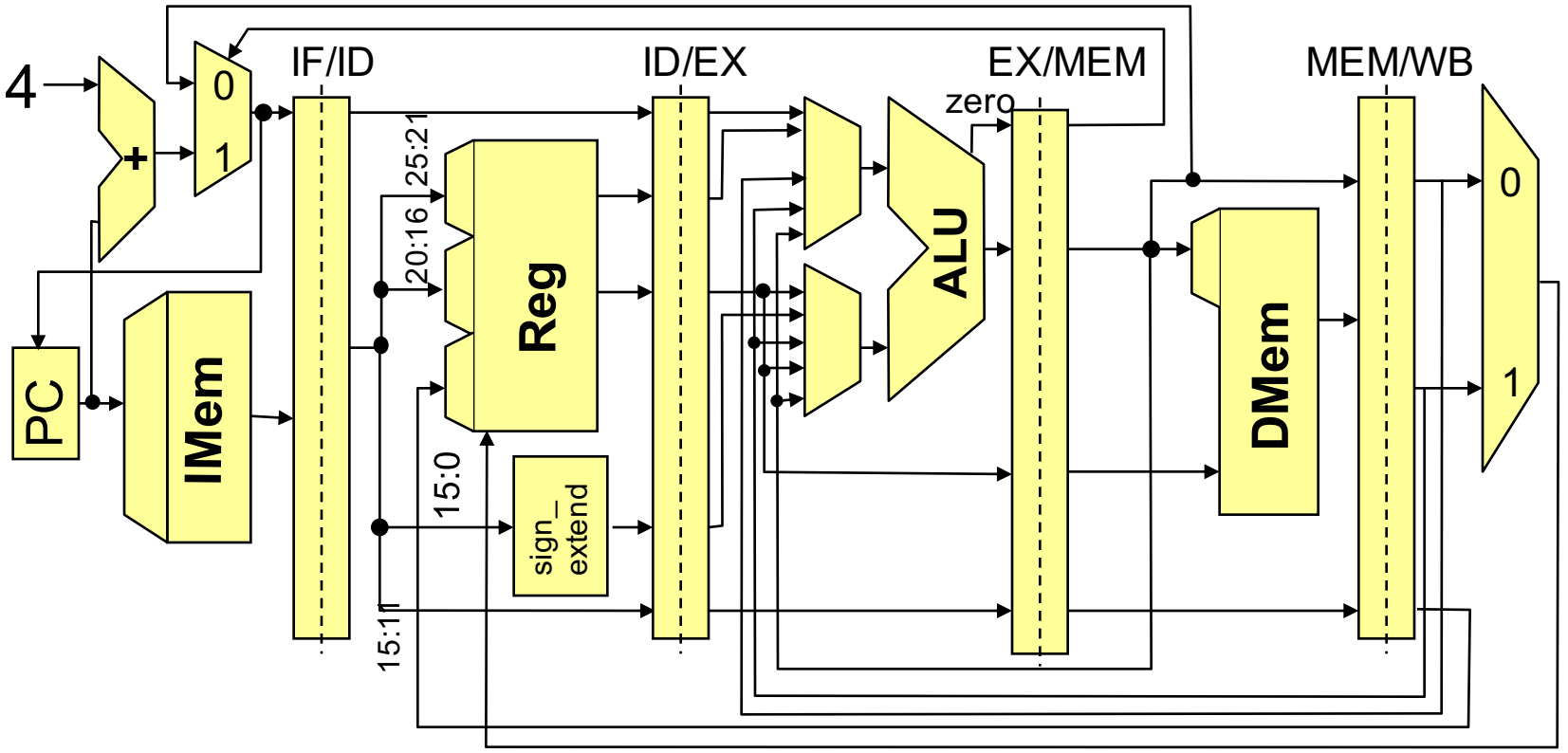
Zyklus 3				
and $6,$1,$7	**sub $4,$5,$1**	**lw $1,0($2)**		

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



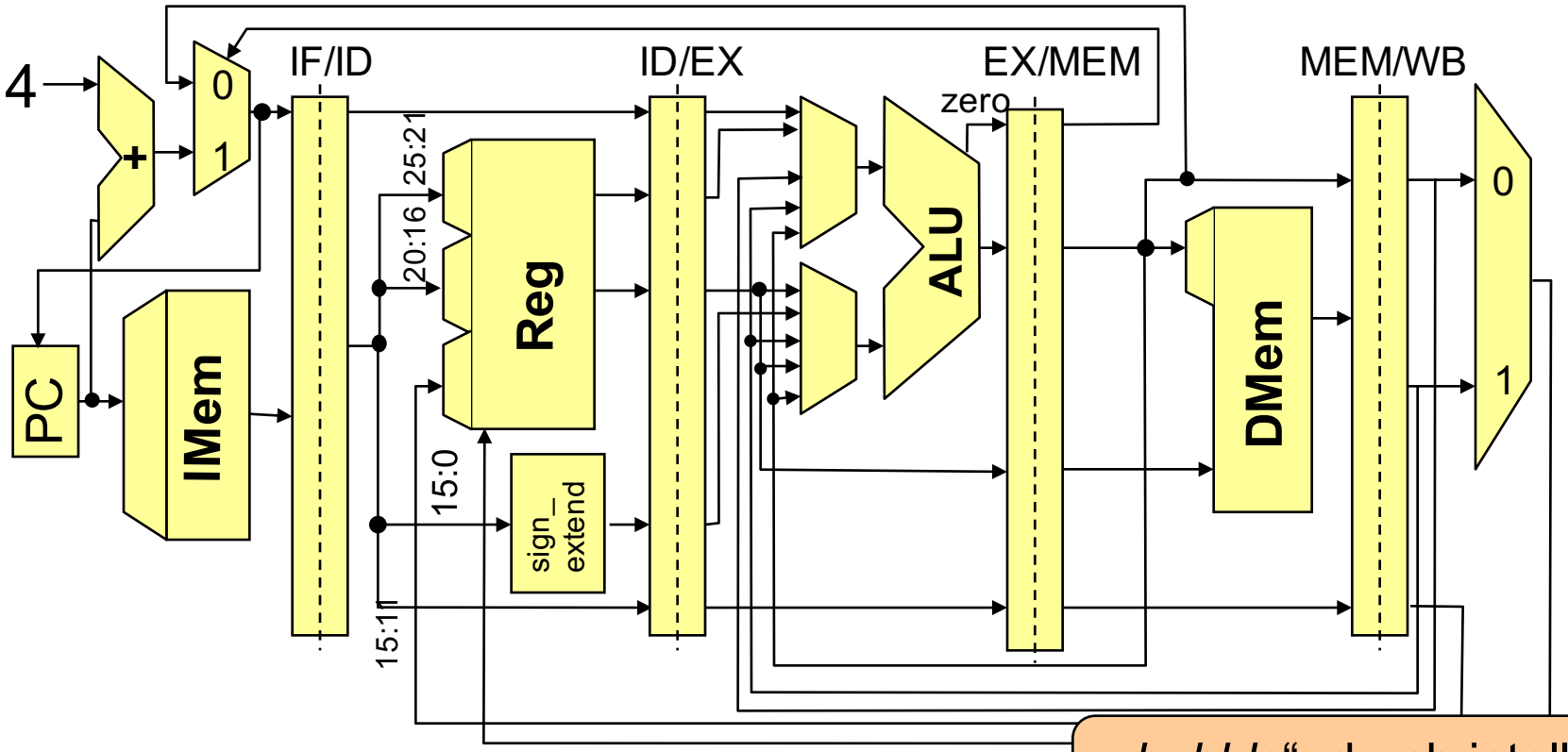
Zyklus 4				
NOOP	NOOP	NOOP	lw $1,0($2)	

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

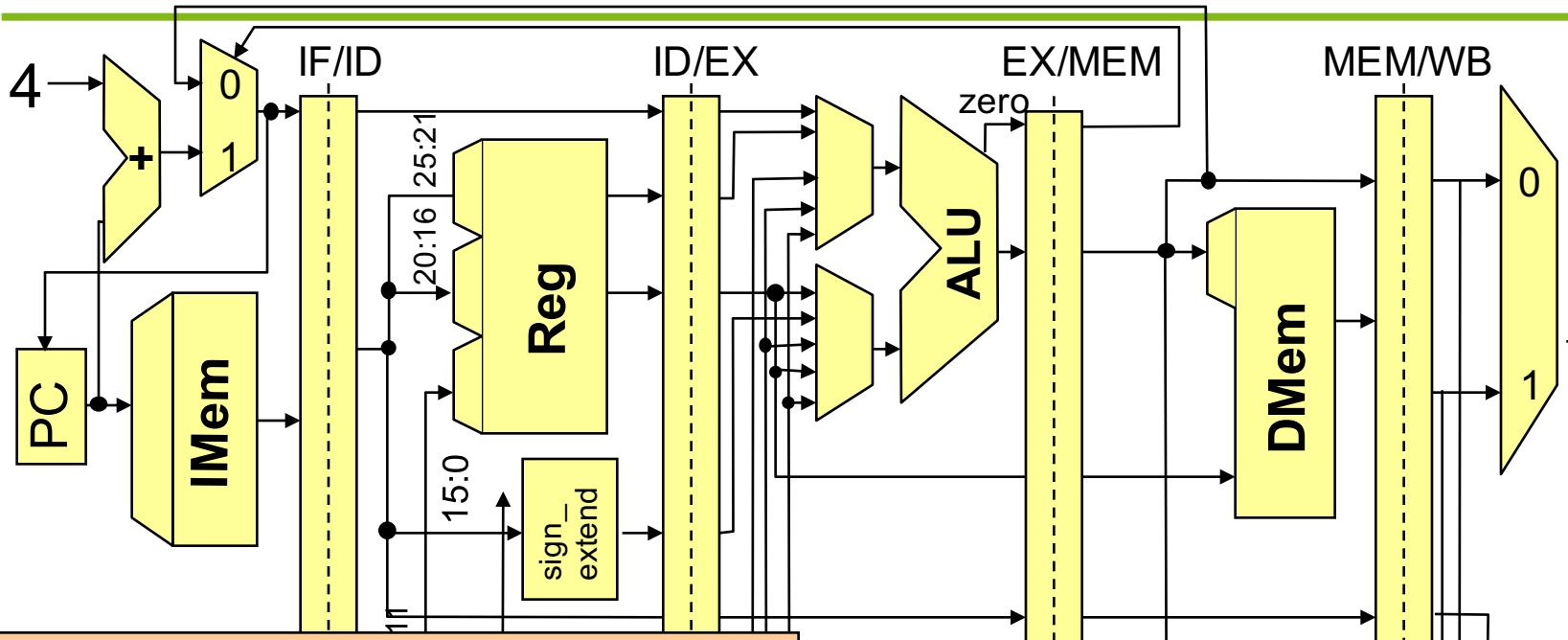
Kontrollfluss-Abhängigkeiten oder - Gefährdungen, *control hazards* (1)

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)

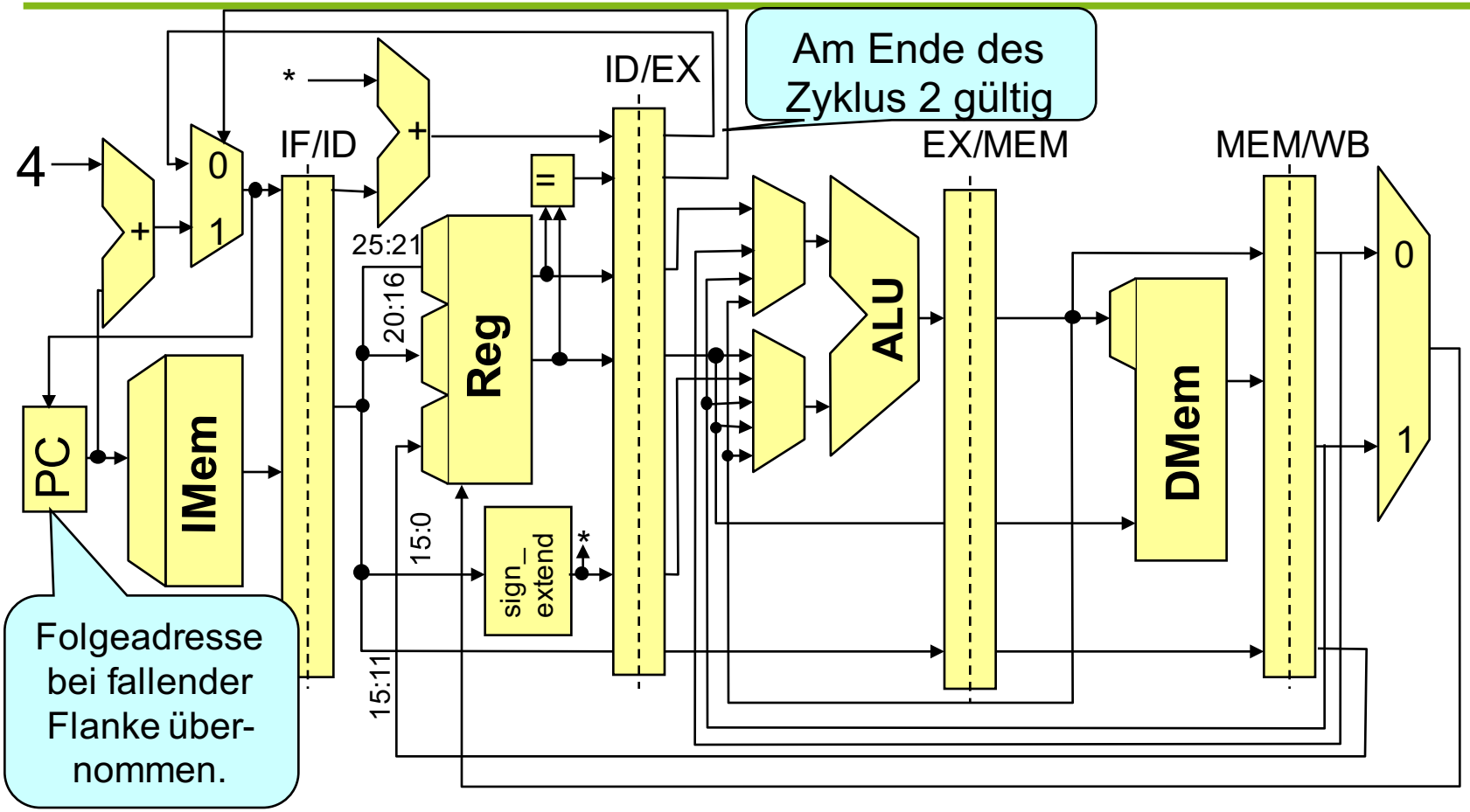
Probleme beim gezeigten Ansatz:

- 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*)

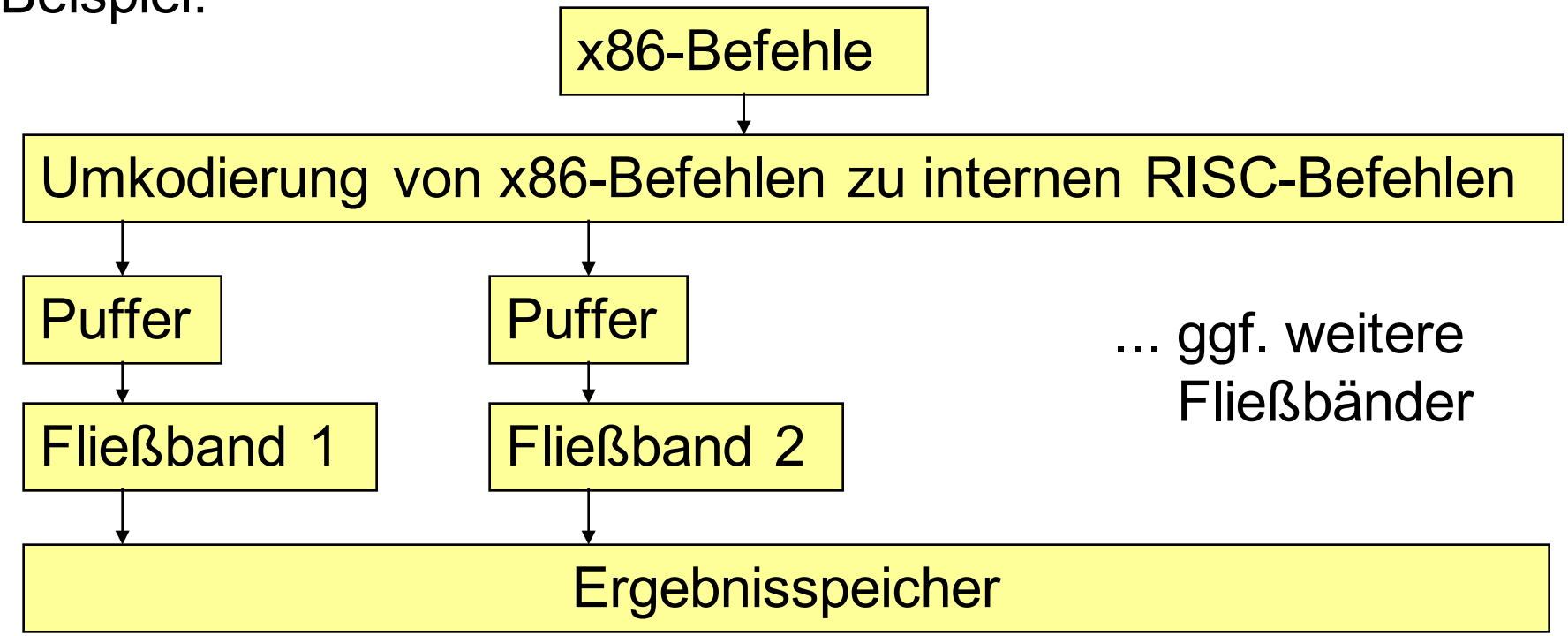
- **Strukturelle Abhängigkeiten/Gefährdungen (*structural hazards*)**
- **Datenfluß- Abhängigkeiten/Gefährdungen (*data hazards*)**
  - **aufgrund von Datenabhängigkeiten (RAW)**

    ☞ *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

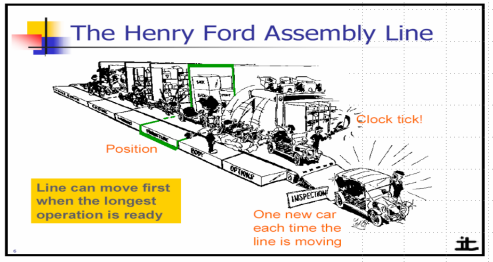
Interne Struktur von Pentium-Prozessoren

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



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*



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

- Gegenmaßnahmen:
  - *pipeline stall*
  - *forwarding/bypassing, delayed branches*
  - *branch prediction, out-of-order execution, dynamic scheduling* (☞ Rechnerarchitektur)
- Ggf. mehrere Fließbänder bei modernen Architekturen