ulm university universität ulm

Grundlagen der Rechnerarchitektur

[CS3100.010]

Wintersemester 2014/15

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

ulm university universität ulm

Kapitel 6

Grundlagen der Rechnerarchitektur

Inhalte der Vorlesung

1. Einführung
2. Kombinatorische Logik
3. Sequentielle Logik
4. Technologische Grundlagen
5. Rechnerarithmetik
6. **Grundlagen der Rechnerarchitektur**
7. Speicher-Hardware
8. Ein-/Ausgabe

Inhalte des Kapitels (1)

6. Grundlagen der Rechnerarchitektur

- Grundbegriffe der Rechnerarchitektur
  - Definitionen von „Rechnerarchitektur“
  - Von Neumann-Modell
- Programmiermodelle / die Befehlsschnittstelle
  - Adressierungsarten (Referenzstufen 0, 1, 2, *n*)
  - 3-, 2-, 1½-, 1-, 0-Adressmaschinen
  - CISC & RISC
  - Digitale Signalprozessoren (DSPs)
  - Multimedia-Befehlssätze
  - *Very Long Instruction Word*-Maschinen (VLIW)
  - Netzwerk-Prozessoren (NPUs)
- ...

Inhalte des Kapitels (2)

6. Grundlagen der Rechnerarchitektur

- …
- Aufbau einer MIPS-Einzelzyklusmaschine
  - Befehlsholphase *(Instruction Fetch)*
  - Dekodierphase *(Instruction Decode)*
  - Ausführungsphase *(Execute)*
  - Speicherzugriff *(Memory Access)*
  - Register zurückschreiben *(Write Back)*
- …

Inhalte des Kapitels (3)

6. Grundlagen der Rechnerarchitektur

- …
- Fließbandverarbeitung / *Pipelining*
  - Fließband-Architektur
  - *Pipeline-Hazards*
    - Datenabhängigkeiten (RAW, WAR, WAW)
    - Bypässe / *Forwarding*
    - *Pipeline Stalls*
    - Kontrollabhängigkeiten, *branch delay penalty*
    - *Delayed Branches*
- Dynamisches *Scheduling*
  - *In-order* und *Out-of-order Execution*
  - *Scoreboarding*

Definitionen von „Rechnerarchitektur“ (1)

Definition nach Stone:

*The study of computer architecture is the study of the **organization and interconnection of components** of computer systems. Computer architects construct computers from **basic building blocks** such as memories, arithmetic units and buses.*

*From these building blocks the computer architect can construct **anyone of a number of different types of computers**, ranging from the smallest hand-held pocket calculator to the largest ultra-fast super computer. The functional behaviour of the components of one computer are similar to that of any other computer, whether it be ultra-small or ultra-fast.*

Definitionen von „Rechnerarchitektur“ (2)

Definition nach Stone: *(Forts.)*
By this we mean that a memory performs the storage function, an adder does addition, and an input/output interface passes data from a processor to the outside world, regardless of the nature of the computer in which they are embedded.

The major differences between computers lie in the way the modules are connected together, and the way the computer system is controlled by the programs. ***In short, computer architecture is the discipline devoted to the design of highly specific and individual computers from a collection of common building blocks.***

Definitionen von „Rechnerarchitektur“ (3)

Definition nach Amdahl, Blaauw, Brooks:

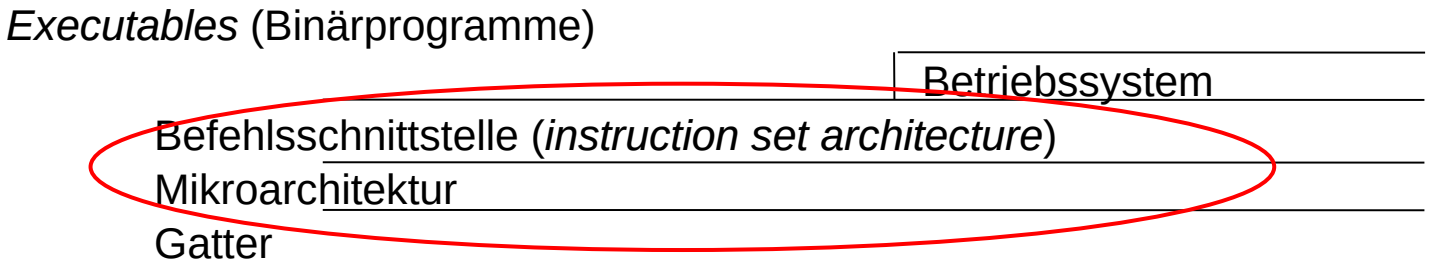
*The term architecture is used here to describe the attributes of a system as seen **by the programmer**, i.e., the conceptual structure and functional behaviour, as distinct from the organization and data flow and control, the logical and physical implementation.*

Gegenüberstellung der Definitionen

Programmierschnittstelle	Interner Aufbau
Externe Rechnerarchitektur	Interne Rechnerarchitektur
Architektur	Mikroarchitektur
Rechnerarchitektur	Rechnerorganisation

Die externe Rechnerarchitektur definiert

- Programmier- oder Befehlssatzschnittstelle
- engl. *Instruction Set Architecture (ISA)*
- eine (reale) Rechenmaschine bzw.
- ein *Application Program Interface (API)*



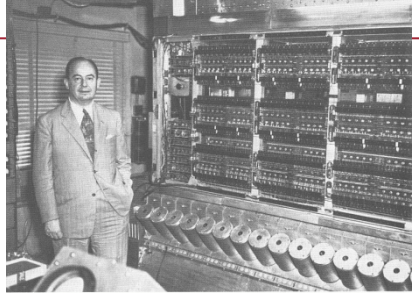
Wieso ist Verständnis von Rechnerarchitektur wichtig?

Zentral: Möglichkeiten und Grenzen des „Handwerkszeugs“ eines Informatikers einschätzen können!

Grundverständnis wird u.a. benötigt bei

- der Geräteauswahl,
- der Fehlersuche,
- der Leistungsoptimierung / Benchmarkentwürfen,
- Zuverlässigkeitsanalysen,
- dem Neuentwurf von Systemen,
- der Codeoptimierung im Compilerbau,
- Sicherheitsfragen,
- ...

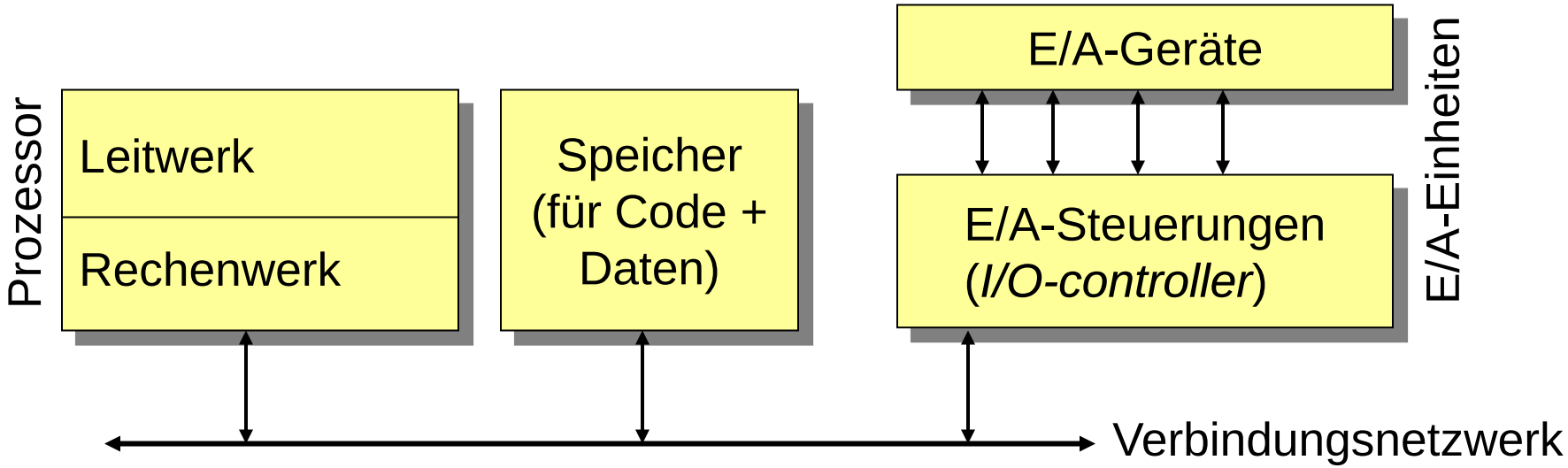
☞ **Keine groben Wissenslücken in zentralen Bereichen der IT!**

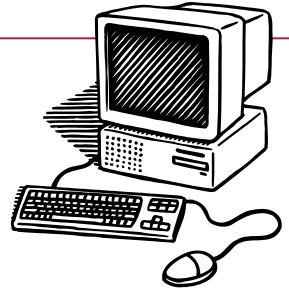


Externe Rechnerarchitektur – das Von Neumann-Modell (1)

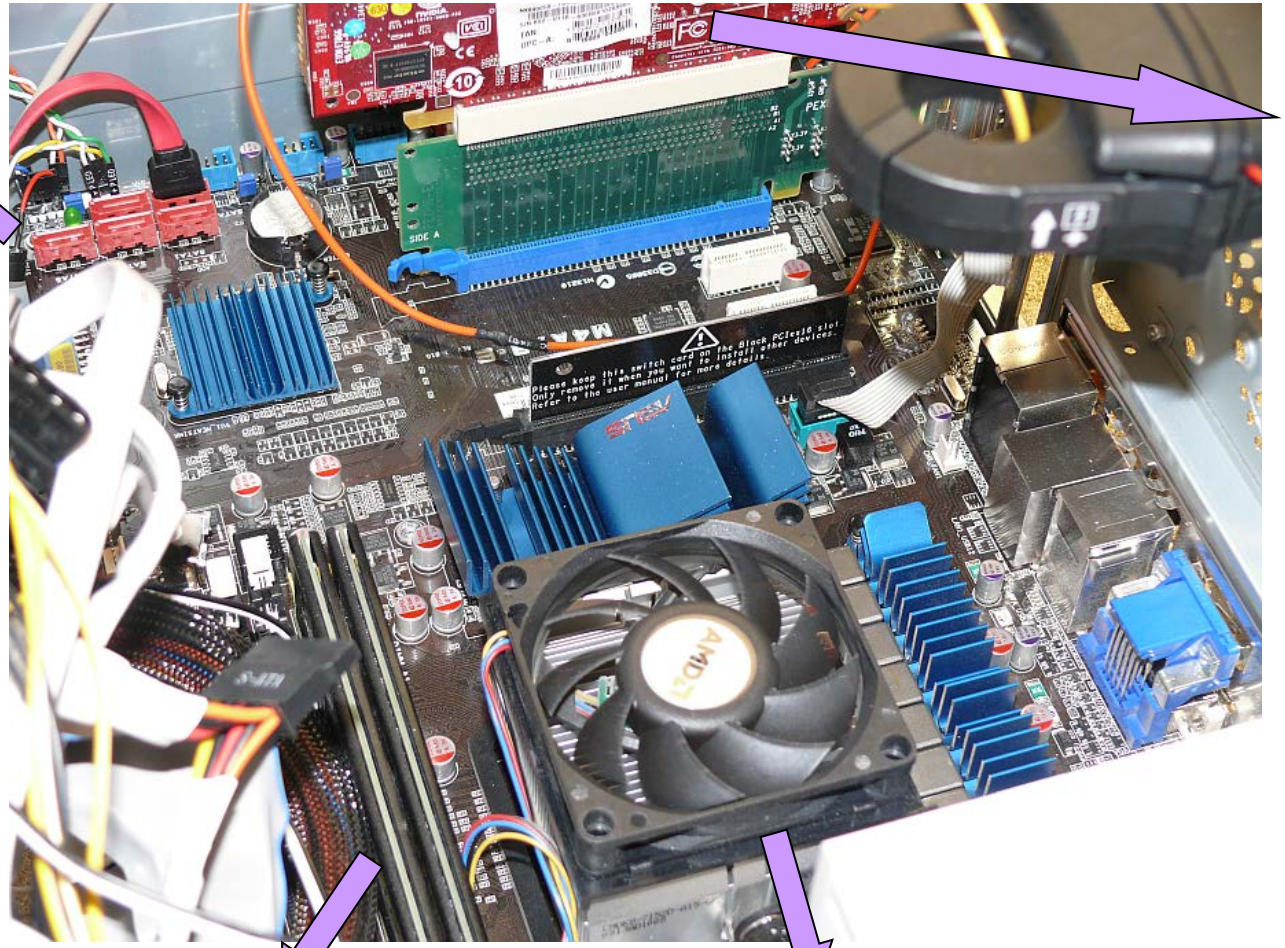
Fast alle der heute üblichen Rechner gehen auf den Von Neumann-Rechner mit folgenden Eigenschaften zurück:

1. Die Rechenanlage besteht aus den Funktionseinheiten **Speicher**, **Leitwerk** (bzw. Steuerwerk, engl. *controller*), dem **Rechenwerk** (engl. *data path*) und **Ein-/Ausgabe-Einheiten**.





Wo sind diese Komponenten auf PC-*Boards*?



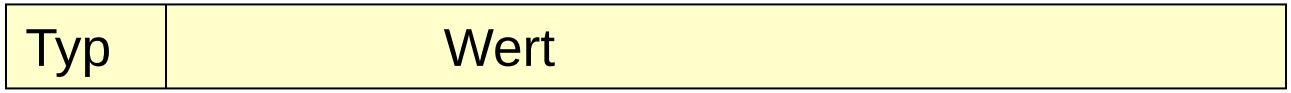
Externe Rechnerarchitektur – das Von Neumann-Modell (2)

2. Die Struktur der Anlage ist unabhängig vom bearbeiteten Problem. Die Anlage ist **speicherprogrammierbar**.
3. **Anweisungen und Operanden** (einschl. Zwischenergebnissen) werden **in demselben physikalischen Speicher** gespeichert.
4. Der Speicher wird in **Zellen gleicher Größe** geteilt. Die Zellnummern heißen **Adressen**.
5. Das Programm besteht aus einer Folge von elementaren **Befehle**n, die **in der Reihenfolge der Speicherung bearbeitet** werden.
6. Abweichungen von der Reihenfolge sind mit (bedingten oder unbedingten) **Sprungbefehlen** möglich.

Externe Rechnerarchitektur – das Von Neumann-Modell (3)

7. Es werden **Folgen von Binärzeichen** (nachfolgend Bitvektoren genannt) verwendet, um alle Größen darzustellen.
8. Die Bitvektoren erlauben **keine explizite Angabe des repräsentierten Typs**. Aus dem Kontext heraus muss stets klar sein, wie die Bitvektoren zu interpretieren sind.

Alternative:



Roter Faden

6. Grundlagen der Rechnerarchitektur

- Grundbegriffe der Rechnerarchitektur
  - Definitionen von „Rechnerarchitektur“
  - Von Neumann-Modell
- Programmiermodelle / die Befehlsschnittstelle
- Aufbau einer MIPS-Einzelzyklusmaschine
- Fließbandverarbeitung / *Pipelining*
- Dynamisches *Scheduling*

Externe Rechnerarchitektur – Befehlsschnittstelle

Befehlsgruppen

- Transferbefehle (**`lw`**, **`sw`**, **`mfhi`**)
- E/A-Befehle (**`in`**, **`out`**)
- Arithmetische Befehle (**`add`**, **`sub`**, **`mul`**, **`div`**)
- Logische Befehle (**`and`**, **`or`**, **`not`**)
- Vergleichsbefehle (**`sgt`**, …, oder Seiteneffekt arithmetischer Befehle)
- Bitfeld- und Flag-Befehle
- Schiebebefehle
- Sprungbefehle
- Kontrollbefehle (*disable interrupt*)
- Ununterbrechbare Befehle (*test-and-set*)

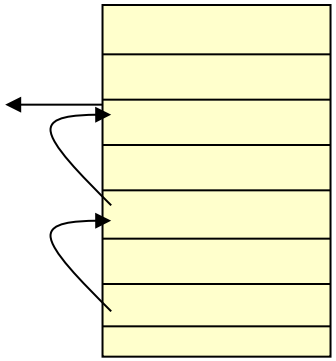
Befehlsschnittstelle

Adressierungsarten

Klassifikation der Adressierung nach der Anzahl der Zugriffe auf den Speicher

– 0-stufige Adressierung
  z.B. Operanden in Registern, Direktoperanden
– 1-stufige Speicheradressierung (Referenzstufe 1)
  z.B. klassische **`lw`**, **`sw`**-Befehle
– 2-stufige Adressierung (Referenzstufe 2)

– *n*-stufige Adressierung (Referenzstufe *n*)



0-stufige Adressierung

– **Registeradressierung**
  ausschließlich Operanden aus & Ziele in Registern
  Beispiele:

`mflo $15`	**`Reg[15]:=Lo`**	MIPS
`clr,3`	**`D[3]:=0`**	680x0
`ldpsw,3`	**`D[3]:=PSW`**	680x0

– **Unmittelbare Adressierung, Direktoperanden, *immediate addressing***
  Operanden sind Teil des Befehlsworts
  Beispiele:

`lui $15,3`	**`Reg[15]:=3 << 16`**	MIPS
`ld D3,#100`	**`D[3]:=100`**	680x0

1-stufige Adressierung, Referenzstufe 1 (1)

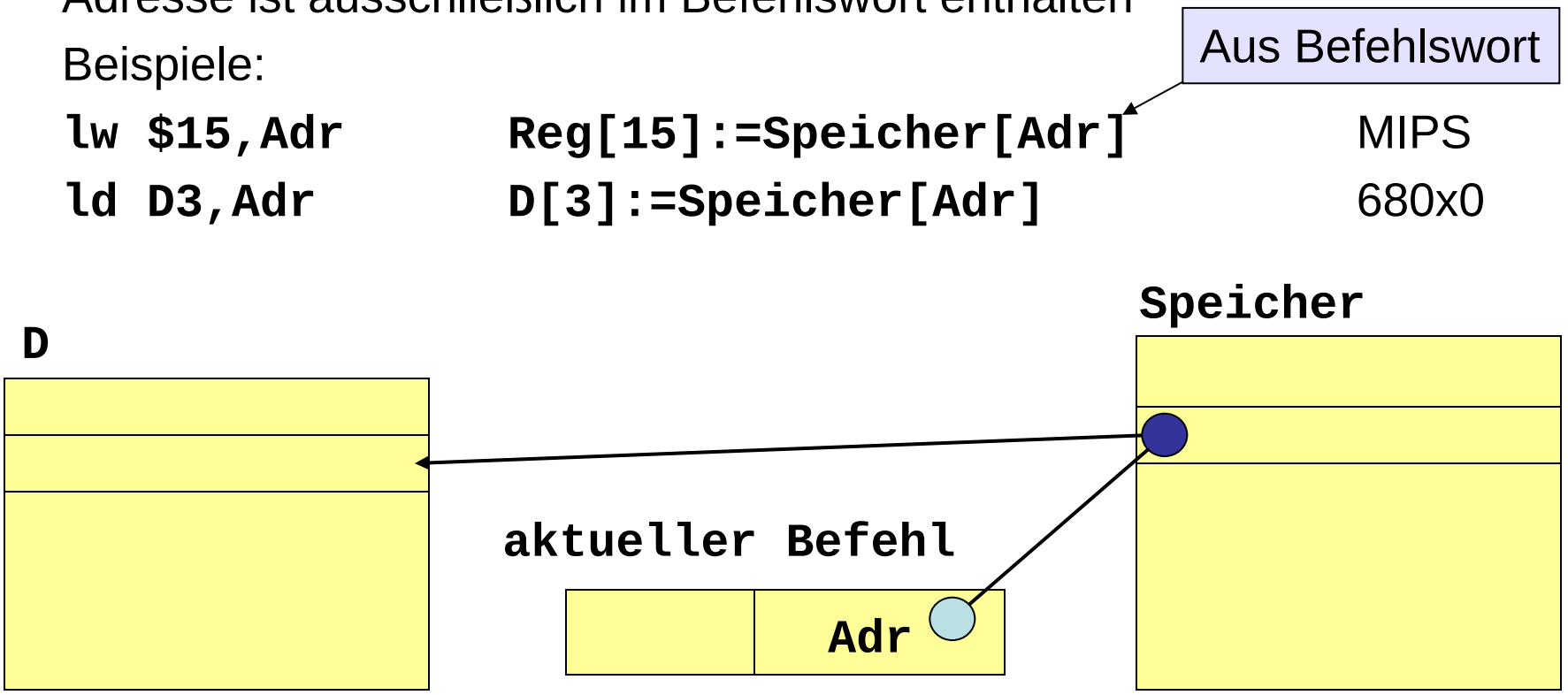
Genau 1 Zugriff auf den Speicher

– **Direkte Adressierung, absolute Adressierung**

Adresse ist ausschließlich im Befehlswort enthalten

Beispiele:

`lw $15,Adr`	`Reg[15]:=Speicher[Adr]`	MIPS
`ld D3,Adr`	`D[3]:=Speicher[Adr]`	680x0



1-stufige Adressierung, Referenzstufe 1 (2)

– **Register-indirekte Adressierung**

Adresse ist ausschließlich im Register enthalten

Beispiele:

`lw $15,($2)`	**`Reg[15]:=Speicher[Reg[2]]`**	MIPS
`ld D3,(A4)`	**`D[3]:=Speicher[A[4]]`**	680x0

Varianten: *pre/post-increment/decrement* zur Realisierung von Stapeloperationen

Beispiel:

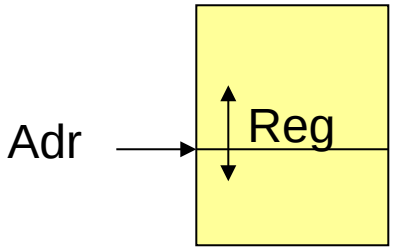
`ld D3,(A4)+` **`D[3]:=Speicher[A[4]];`**
`A[4]:=A[4]+4` (beim Laden von 32 Bit)

1-stufige Adressierung, Referenzstufe 1 (3)

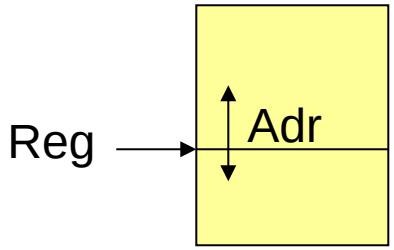
– **Relative Adressierung, indizierte Adressierung, Basis-Adressierung**
  Adresse ergibt sich aus der Addition eines Registerinhalts und einer Konstanten im Befehl
  Beispiele:

```
lw $15,Adr($2)   Reg[15]:=Speicher[Adr+Reg[2]]   MIPS
ld D3,Adr(A4)    D[3]:=Speicher[Adr+A[4]]         680x0
```

  Varianten (nicht immer einheitlich bezeichnet):
  – **Indizierte Adressierung**
    `Adr` umfasst vollen Adressbereich, Register evtl. nicht



  – **Basisadressierung, Register-relative Adressierung**
    Register umfasst vollen Adressbereich, **`Adr`** evtl. nicht



1-stufige Adressierung, Referenzstufe 1 (4)

– **Register-relative Adressierung mit Index**

Addition zweier Register, z.B. zeigt eines auf relevanten Speicherbereich, ein zweites enthält einen *Array*-Index

Beispiele:

```
ld D3,Adr(A3,D4)        D[3]:=Speicher[Adr+A[3]+D[4]]
```

680x0

– **Programmzähler-relative Adressierung**

Addition des Programmzählers zu **Adr**

Beispiele:

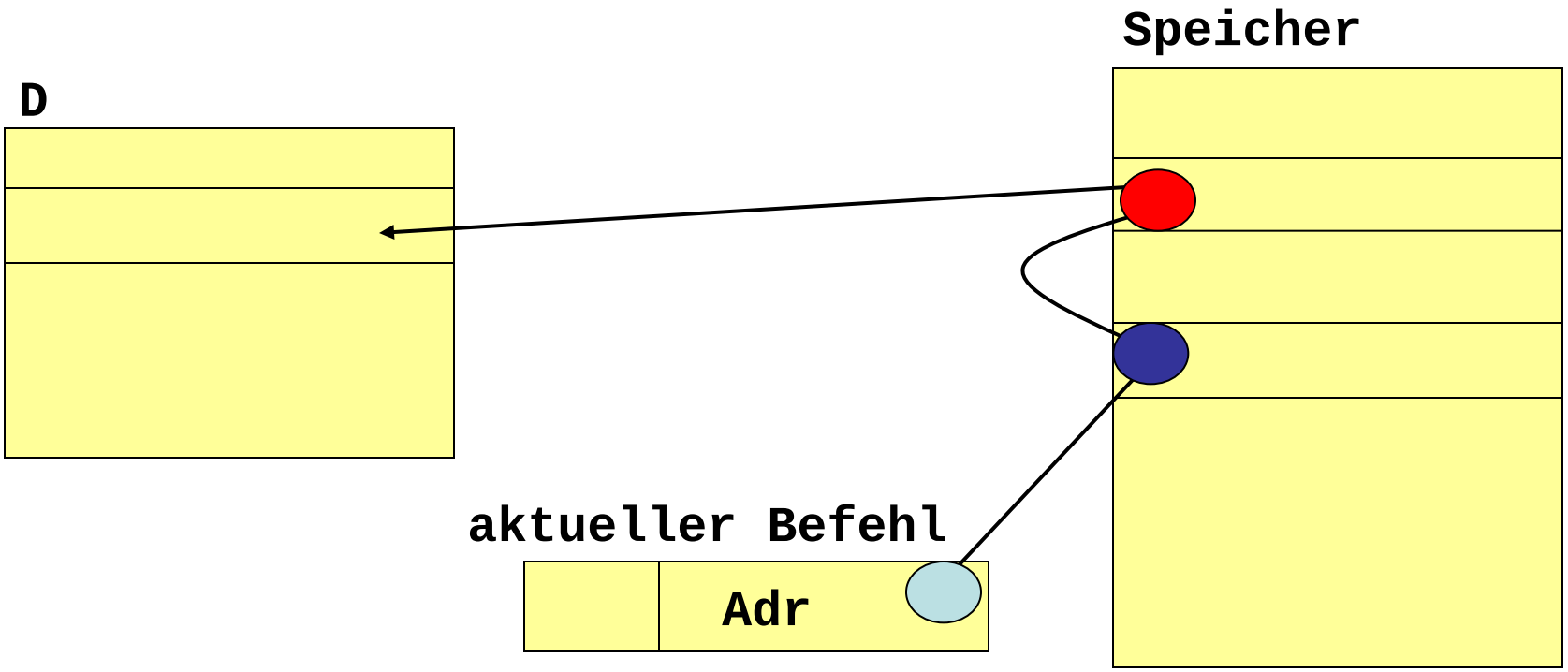
```
L:bne $4,$0,Z           PC:=PC+(if Reg[4]!=0 then Z-L
                                                else 4)
L:bra Z                 PC:=PC+(Z-L)
```

680x0

2-stufige Adressierung, Referenzstufe 2 (1)

– **Indirekte (absolute) Adressierung**

`D[Reg]:=Speicher[Speicher[Adr]]`



2-stufige Adressierung, Referenzstufe 2 (2)

- **Indirekte Register-indirekte Adressierung**
  `D[Reg]:=Speicher[Speicher[D[IReg]]]`

- **Indirekte indizierte Adressierung, Vorindizierung**
  `D[Reg]:=Speicher[Speicher[Adr+D[Ireg]]]`

- **Indirekte indizierte Adressierung, Nachindizierung**
  `D[Reg]:=Speicher[Speicher[Adr]+D[IReg]]`

- **Indirekte Programmzähler-relative Adressierung**
  `D[Reg]:=Speicher[Speicher[Adr+PC]]`

n-stufige Adressierung

- Referenzstufen > 2 werden nur in Ausnahmefällen realisiert
- Die fortgesetzte Interpretation des gelesenen Speicherworts als Adresse des nächsten Speicherworts nennt man Dereferenzieren

- Fortgesetztes Dereferenzieren ist z.B. zur Realisierung der logischen Programmiersprache PROLOG mittels der *Warren Abstract Machine (WAM)* wichtig
- Bei der WAM wird die Anzahl der Referenzstufen durch Kennzeichen-Bits in den gelesenen Speicherworten bestimmt

Befehlsschnittstelle

n-Adressmaschinen

Klassifikation von Befehlssätzen bzw. Befehlen nach der Anzahl der Adressen bei 2-stelligen Arithmetik-Befehlen

- 3-Adressmaschinen
  - Operanden und Ziel einer Operation werden explizit angegeben
- 2-Adressmaschinen
  - Überschreiben eines Operanden mit dem Ergebnis
- 1½-Adressmaschinen
  - wie 2-Adressmaschinen, nur unter Verwendung von Registern
- 1-Adressmaschinen
  - Nutzung von nur 1 Register
- 0-Adressmaschinen
  - Kellermaschinen

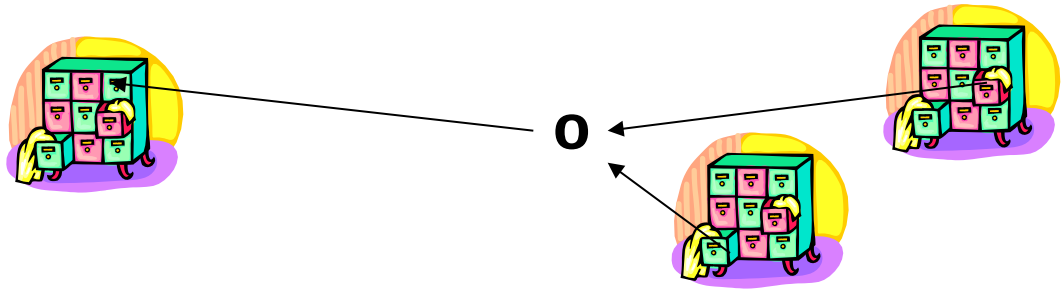
3-Adressmaschinen (1)

3-Adressbefehle

IBM-Bezeichnung „SSS“; Befehle bewirken Transfer:

`Speicher[s1] := Speicher[s2] o Speicher[s3]`

`s1`, **`s2`**, **`s3`**: Speicheradressen, **`o`**: 2-stellige Maschinenoperation



Mögliches Befehlsformat

Opcode	s1	s2	s3

4x 32 = 128 Bit Befehlswortbreite

3-Adressmaschinen (2)

– Anwendung: Zerlegung der Anweisung

D = (A + B) * C;

Mit **a**=Adresse von A, **b**=Adresse von B, usw.:

```
add t1,a,b      Speicher[t1]:=Speicher[a]+Speicher[b]
mult d,t1,c     Speicher[d]:=Speicher[t1]*Speicher[c]
```

– Programmgröße: 2*128 = 256 Bit
Speicherzugriffe: 2*3 = 6

2-Adressmaschinen (1)

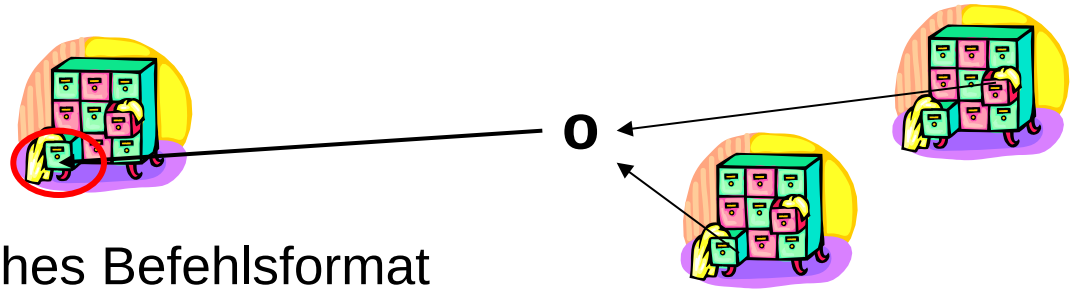
Verkürzung des Befehlsworts durch Überschreiben eines Operanden mit dem Ergebnis.

2-Adressbefehle

IBM-Bezeichnung „SS“; Befehle bewirken Transfer der Form:

`Speicher[s1] := Speicher[s1] o Speicher[s2]`

`s1`, **`s2`**: Speicheradressen, **`o`**: 2-stellige Maschinenoperation



Mögliches Befehlsformat

Opcode	s1	s2

3x 32 = 96 Bit Befehlswortbreite

2-Adressmaschinen (2)

– Anwendung: Zerlegung der Anweisung

D = (A + B) * C;

Mit **a**=Adresse von A, **b**=Adresse von B, usw.:

```
move t1,a       Speicher[t1]:=Speicher[a]
add t1,b        Speicher[t1]:=Speicher[t1]+Speicher[b]
mult t1,c       Speicher[t1]:=Speicher[t1]*Speicher[c]
move d,t1       Speicher[d]:=Speicher[t1]
```

– Programmgröße: 4*96 = 384 Bit
Speicherzugriffe: 2*3 + 2*2 = 10

[Frage: Wie lässt sich obige Anweisung effizienter zerlegen?]

1½-Adressmaschinen (1)

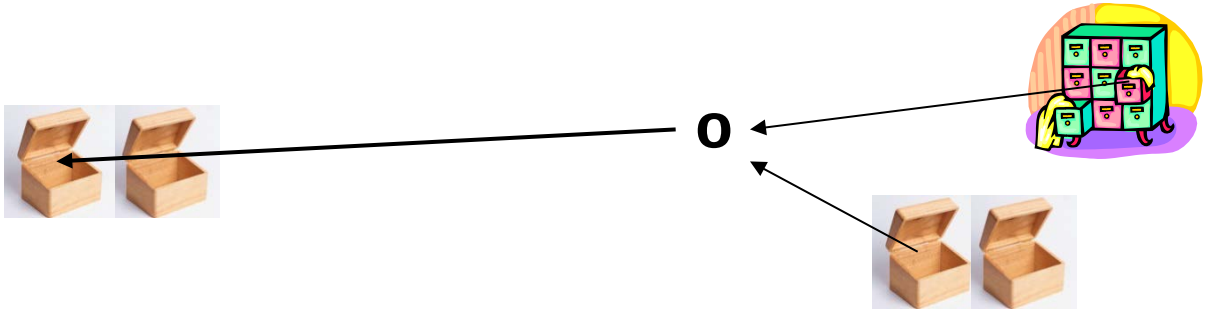
Weitere Verkürzung des Befehlsworts mit Registerspeichern.

1½-Adressbefehle

IBM-Bezeichnung „RS“; Befehle bewirken Transfer der Form:

```
Reg[r] := Reg[r] o Speicher[s]
```

s: Speicheradresse, **r**: Registernummer, **o**: 2-stellige Maschinenoperation



Mögliches Befehlsformat

Opcode RegNr.	s

2x 32 = 64 Bit Befehlswortbreite

Enthält zusätzlich RR-Befehle der Wirkung
Reg[r1]:= Reg[r2]

1½-Adressmaschinen (2)

– Anwendung: Zerlegung der Anweisung

D = (A + B) * C;

Mit **a**=Adresse von A, **b**=Adresse von B, usw.:

```
lw $8,a          Reg[8]:=Speicher[a]
add $8,b         Reg[8]:=Reg[8]+Speicher[b]
mult $8,c        Reg[8]:=Reg[8]*Speicher[c]
sw $8,d          Speicher[d]:=Reg[8]
```

– Programmgröße: 4*64 = 256 Bit
Speicherzugriffe: 4

1-Adressmaschinen (1)

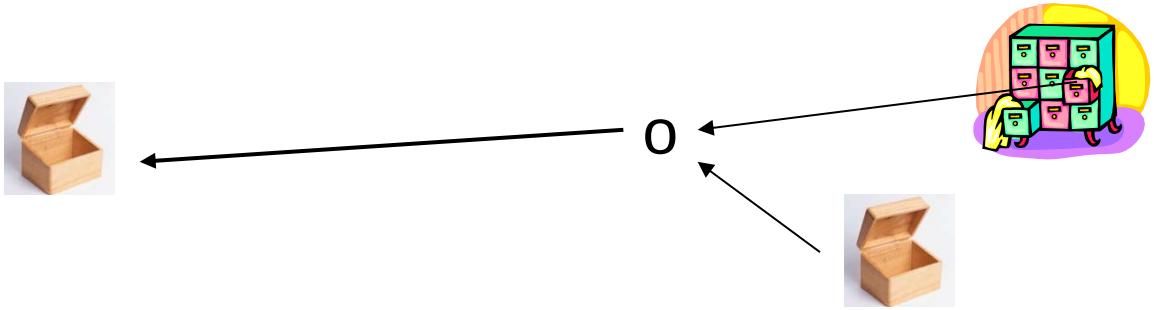
Sonderfall der Nutzung von nur 1 Register („Akkumulator“)

1-Adressbefehle

Befehle bewirken Transfer der Form:

```
accu := accu o Speicher[s]
```

s: Speicheradresse, **o**: 2-stellige Maschinenoperation



Mögliches Befehlsformat

Opcode	s

2x 32 = 64 Bit Befehlswortbreite

1-Adressmaschinen (2)

– Anwendung: Zerlegung der Anweisung

D = (A + B) * C;

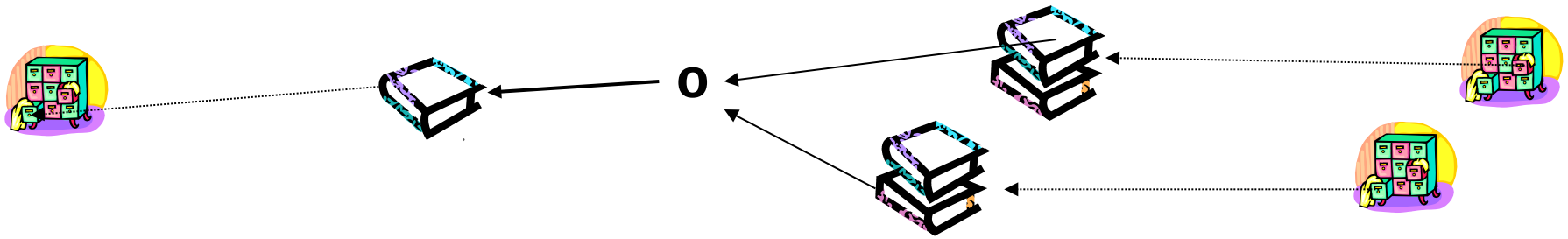
Mit **a**=Adresse von A, **b**=Adresse von B, usw.:

```
lw a          accu:=Speicher[a]
add b         accu:=accu+Speicher[b]
mult c        accu:=accu*Speicher[c]
sw d          Speicher[d]:=accu
```

– Programmgröße: 4*64 = 256 Bit
  Speicherzugriffe: 4

0-Adressmaschinen (1)

Arithmetische Operationen werden auf einem Kellerspeicher *(Stack)* ohne Angabe der Operanden ausgeführt. Die 2 obersten *Stack*-Elemente werden verknüpft und durch das Ergebnis der Operation ersetzt.



0-Adressbefehle

3 Sorten von Befehlen:

`push a`	**`Speicher[a]`** wird oberstes *Stack*-Element
`add`	Addiere die beiden obersten *Stack*-Elemente, entferne beide vom *Stack*, schreibe die Summe oben auf den Stack (analog für **`-`**, **`*`**, **`/`**, …)
`pop a`	**`Speicher[a]:=`** oberstes *Stack*-Element, entferne dieses Element vom *Stack*

0-Adressmaschinen (2)

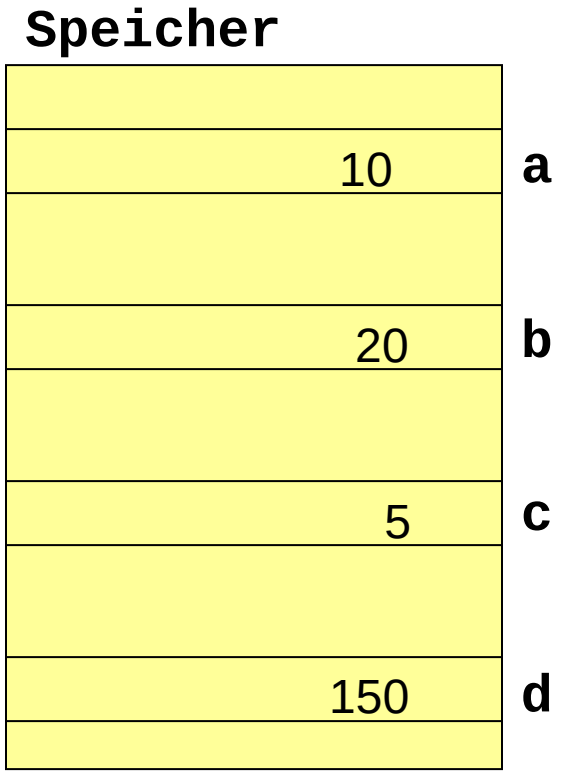
– Anwendung: Zerlegung der Anweisung

D = (A + B) * C;

Mit **a**=Adresse von A, **b**=Adresse von B, usw.:

```
push a
push b
add
push c
mult
pop d
```

Stack



– Mögliche Befehlsformate:
Arithmetikbefehle: 1 Byte
push, **pop**: 1 Byte + 4 Byte (Adresse)

Programmgröße: 22 Bytes/176 Bits
Speicherzugriffe: 4

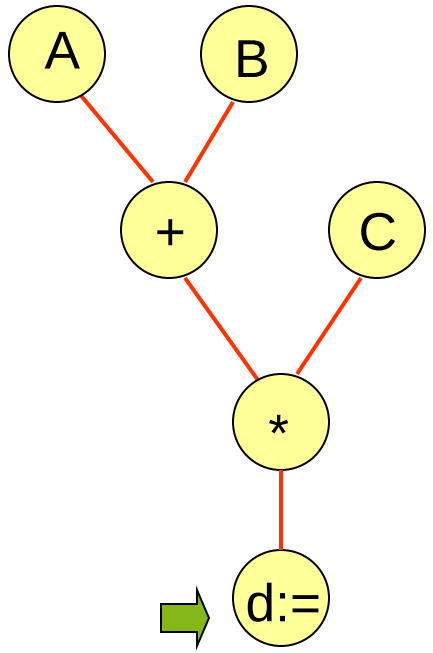
0-Adressmaschinen (3)

Wie kommt man vom Ausdruck
D = (A + B) * C zur Befehlssequenz?

1. Verwendung der ***postfix***-Notation:
   AB+C*

2. ***Postorder-Durchlauf*** durch den Ausdrucksbaum

```
GenerateCode( p )
{
  gehe zum linken Sohn;
  gehe zum rechten Sohn;
  erzeuge Befehl für aktuellen Knoten;
}
```




```
push a
push b
add
push c
mult
pop d
```

Befehlsschnittstelle

Programmiermodelle, *Instruction Set Architectures (ISAs)*

Klassifikation von Befehlssätzen nach der Gestaltung/Ausprägung der vorhandenen Maschinenbefehle

- CISC, RISC
- Digitale Signalprozessoren (DSPs)
- Multimedia-Befehlssätze
- *Very Long Instruction Word*-Maschinen (VLIW)
- Netzwerk-Prozessoren (NPUs)

Complex Instruction Set Computers (CISC)

Ursprünge

Entstanden in Zeiten schlechter Compiler und großer Geschwindigkeitsunterschiede Speicher / Prozessor

– Befehle sollten möglichst nahe an den Hochsprachen sein: keine semantische Lücke zwischen Quellcode und Maschinenprogramm
– Mit jedem geholten Befehl sollte der Prozessor viel tun

☞ sehr komplexe Befehle

CISC-Beispiel Motorola MC80x0 (1)

Beispiel: Motorola 68000 (erster Prozessor der 680x0 Serie)

Format des Kopierbefehls **`MOVE`**:

Opcode	Größe	Ziel		Quelle	
"00"	"01"=Byte, "11"=Wort, "10"=Doppelwort (32 Bit)	Register	Modus	Register	Modus
bis zu 4 Erweiterungsworte zu je 16 Bit					

Viele komplexe Adressierungsarten schon in den ersten Prozessoren der Serie.

In Form der ColdFire-Prozessoren weiterhin eingesetzt.

CISC-Beispiel Motorola MC80x0 (2)

Modus	Register feld	Erweit.	Notation	Adressierung
"000"	n	0	Dn	Register-Adressierung
"001"	n	0	An	Adressregister-Adressierung
"010"	n	0	(An)	Adressregister indir.
"011"	n	0	(An)+	Adressreg. indirekt.mit *postincrement*
"100"	n	0	-(An)	Adressreg. indirekt.mit *predecrement*
"101"	n	1	d(An)	Relative Adressierung mit 16 Bit Distanz
"110"	n	1	d(An,Xm)	Register-relative Adressierung mit Index
"111"	"000"	1	d	direkte Adressierung (16 Bit)
"111"	"001"	2	d	direkte Adressierung (32 Bit)
"111"	"010"	1	d(*)	Programmzähler-relativ
"111"	"011"	1	d(*,Xn)	Programmzähler-relativ mit Index
"111"	"100"	1-2	#zahl	unmittelbare Adressierung

Complex Instruction Set Computers (CISC)

Eigenschaften

- Relativ kompakte Codierung von Programmen
- Für jeden Befehl wurden mehrere interne Zyklen benötigt
  - Die Anzahl der Zyklen pro Befehl war groß
- (Mikro-) Programm zur Interpretation der Befehle nötig
- Compiler konnten viele Befehle gar nicht nutzen

Reduced Instruction Set Computers (RISC) (1)

Definition:
Unter dem CPI-Wert (engl. *cycles per instruction*) einer Menge von Maschinenbefehlen versteht man die mittlere Anzahl interner Taktzyklen pro Maschinenbefehl.

RISC-Architekturen: Wenige, einfache Befehle wegen folgender Ziele

- Hohe Ausführungsgeschwindigkeit
  - durch kleine Anzahl interner Zyklen pro Befehl
  - durch Fließbandverarbeitung (siehe später)

Programmlaufzeit =

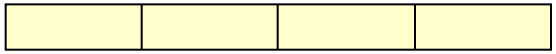
Anzahl auszuführender Befehle * CPI-Wert * Dauer eines Taktzyklus

CISC-Maschinen: schwierig, unter CPI = 2 zu kommen

RISC-Maschinen: CPI möglichst nicht über 1

Reduced Instruction Set Computers (RISC) (2)

Eigenschaften



- feste Befehlswortlänge
- LOAD/STORE-Architektur

ld $2,..; ld $3,..; add $3,$3,$2; sw $3

- einfache Adressierungsarten

z.B. keine indirekte Adr.

- „semantische Lücke“ zwischen Hochsprachen & Assemblerbefehlen durch Compiler überbrückt
- statt aufwändiger Hardware zur Beseitigung von Besonderheiten (z.B. 256 MB-Grenze bei MIPS, 16-Bit Konstanten) wird diese Aufgabe der Software übertragen
- rein in Hardware realisierbar („mit Gattern und Flip-Flops“, ☞ Kapitel 3), keine Mikroprogrammierung

☞ **Nahezu jeder heutige Rechner ist eine RISC-Maschine.**

Digitale Signalprozessoren

DSP = *Digital Signal Processing*

Spezialanwendungen für Rechner, in der Regel in eingebetteten Systemen *(embedded systems)*.
IT ist in eine Umgebung eingebettet, z.B.

– im Telekommunikationsbereich (Mobiltelefon)
– im Automobilbereich (Spurhalteassistent)
– im Consumerbereich (Audio/Video-Komprimierung)

Wichtige Teilaufgabe: Digitale Signalverarbeitung

w und *x* sind Signale (mathematisch: Abbildungen von der Zeit auf Signalwerte)

Digitale Signalprozessoren

Eigenschaften

- Optimiert für Digitale Signalverarbeitung
  (z.B. Filter, Fourier-Transformation, …)
- Heterogene Registersätze, eingeteilt für Spezialzwecke
- Teilweise parallele Befehlsabarbeitung
- Spezielle Adressrechenwerke / Adressierungsmodi
- *Multiply-Accumulate*-Befehl *(a = a + b * c)*
- *Zero-Overhead* Loops
- Sättigungsarithmetik
- Effizienz und Realzeitverhalten extrem wichtig

DSPs: Heterogene Registersätze

Beispiel Infineon TriCore 1.3:

– Separate Adress- & Datenregister

Address Registers	*Data Registers*
A15	D15
A14	D14
A13	D13
A12	D12
A11	D11
A10	D10
A9	D9
A8	D8
A7	D7
A6	D6
A5	D5
A4	D4
A3	D3
A2	D2
A1	D1
A0	D0

DSPs: Heterogene Registersätze

Beispiel Infineon TriCore 1.3:

- Separate Adress- & Datenregister
- Register mit besonderer Bedeutung
- 64-bit Datenregister *(extended Regs)*
- Oberer & unterer Kontext *(UC & LC)*: *UC* bei Funktionsaufruf automatisch gesichert, *LC* nicht

	Address Registers	*Data Registers*	
UC	*A15 (Implicit AREG)*	*D15 (Implicit DREG)*	E14
	A14	D14	
	A13	D13	E12
	A12	D12	
	A11 (Return Addr)	D11	E10
	A10 (Stack Ptr)	D10	
	A9 (Global AREG)	D9	E8
	A8 (Global AREG)	D8	
LC	A7	D7	E6
	A6	D6	
	A5	D5	E4
	A4	D4	
	A3	D3	E2
	A2	D2	
	A1 (Global AREG)	D1	E0
	A0 (Global AREG)	D0	

DSPs: Teilweise Parallelität

Beispiel Infineon TriCore 1.3:

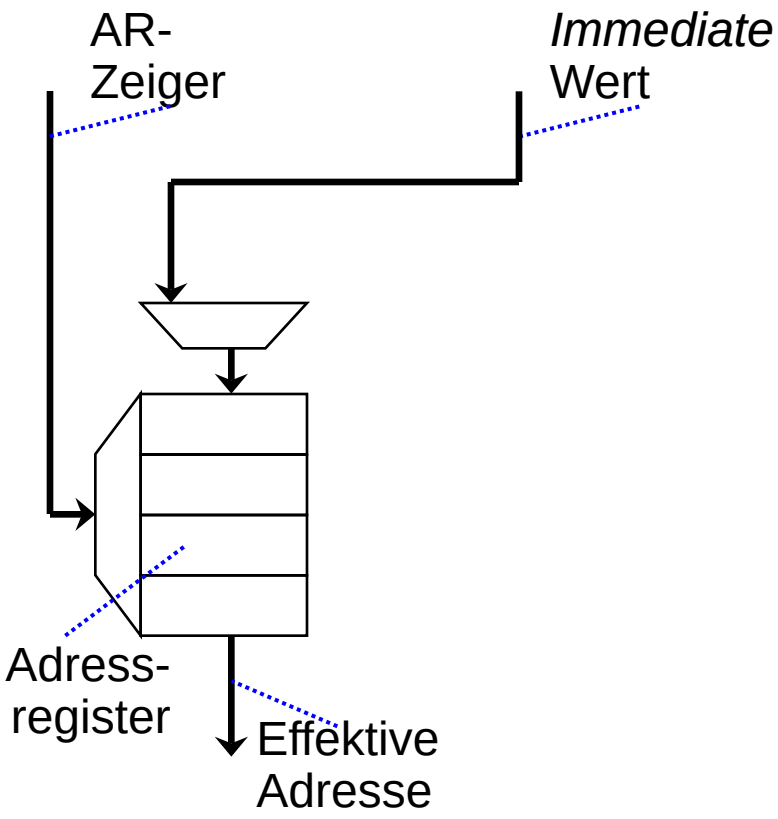
- *Integer Pipeline:* Arithmetische Befehle, bedingte Sprünge
- *Load/Store Pipeline:* Speicherzugriffe, Adressarithmetik, unbedingte Sprünge, Funktionsaufrufe
- *Loop-Pipeline:* Schleifen-Befehle

- Teilweise Parallelität
  - *Pipelines* arbeiten im Idealfall unabhängig / parallel
  - Wenn nicht Idealfall:
    Stall in L/S-*Pipeline* → *Stall* in I-*Pipeline* und umgekehrt

DSPs: *Address Generation Units (AGUs)*

Allgemeine Architektur von Adressrechenwerken:

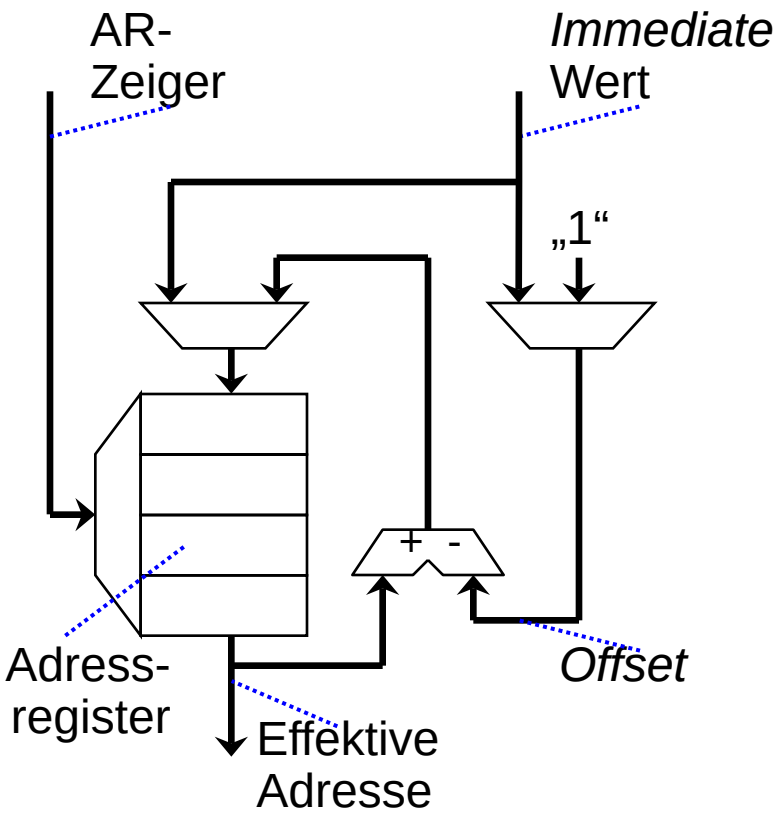
- Adressregister enthalten *effektive Adressen* zur Speicher-Adressierung
- Befehlswort codiert, welches AR zu nutzen ist *(AR-Zeiger)*
- ARs können explizit mit im Befehlswort codierten Konstanten geladen werden *(Immediates)*



DSPs: *Address Generation Units (AGUs)*

Allgemeine Architektur von Adressrechenwerken:

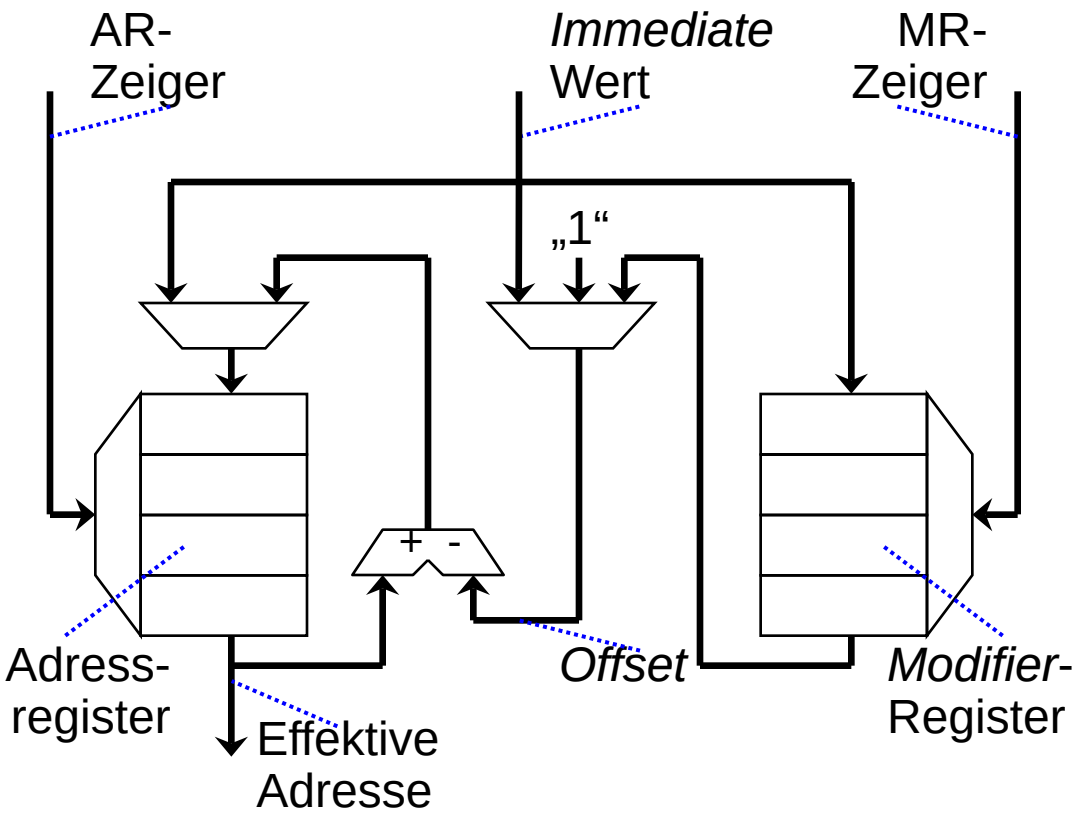
- ARs können über einfache ALU erhöht / erniedrigt werden
- Erhöhung / Erniedrigung um *Offset* als *Immediate*-Wert
- Inkrement / Dekrement um Konstante „1“ als *Offset*



DSPs: *Address Generation Units (AGUs)*

Allgemeine Architektur von Adressrechenwerken:

- Inkrement / Dekrement um Inhalt von *Modifier*-Register *(MR)*
- Befehlswort codiert, welches MR zu nutzen ist *(MR-Zeiger)*
- MRs können explizit mit Immediate-Werten geladen werden

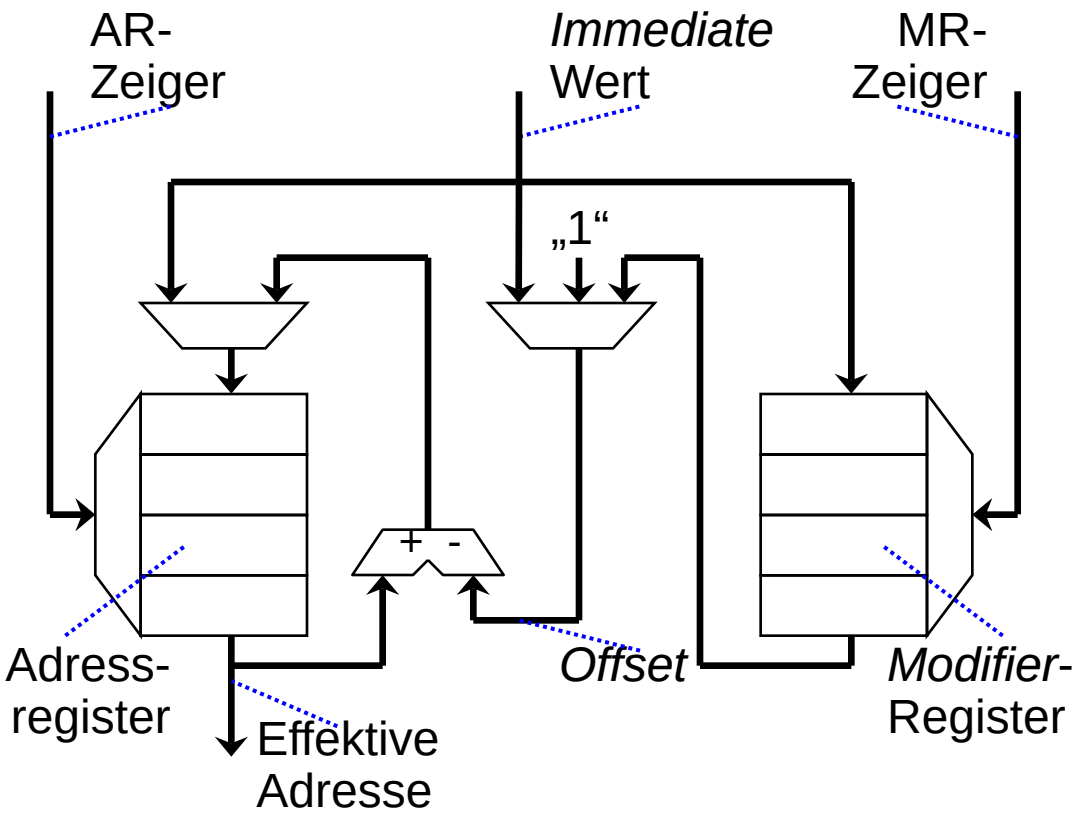


DSPs: *Address Generation Units (AGUs)*

Allgemeine Architektur von Adressrechenwerken:

- AR laden: AR = <*const*>
- MR laden: MR = <*const*>
- AR ändern: AR ± <*const*>
- *Auto-Increment:* AR ± „1“
- *Auto-Modify:* AR ± MR

- *„Auto”-Befehle:* Parallel zu Datenpfad, keine extra Laufzeit, hocheffizient!
- *Alle anderen:* Brauchen Extra-Instruktion für Datenpfad, weniger effizient.



DSPs: Konventioneller Code für Schleifen

C-Code einer Schleife:

```
int i = 10;
do {
  ...
  i--;
} while ( i );
```

Konventioneller ASM-Code:
(TriCore 1.3)

```
    mov %d8, 10;
.L0:
    ...
    add %d8, -1;
    jnz %d8, .L0;
```

Eigenschaften

- Dekrement & bedingter Sprung: Beide in *Integer-Pipeline*
  - ☞ Keine parallele Ausführung
- 2 Takte * 10 Iterationen = mind. 20 Takte Schleifen-*Overhead*
- *Bei Delay-Slots für Sprünge noch mehr!*

DSPs: Optimierter Code für Schleifen

C-Code einer Schleife:

```
int i = 10;
do {
  ...
  i--;
} while ( i );
```

Zero-Overhead Loops:
(TriCore 1.3)

```
  mov %a12, 10;
.L0:
  ...
  loop %a12, .L0;
```

Eigenschaften

- Dekrement & bedingter Sprung: Parallel in *Loop-Pipeline*
- **`loop`**-Befehl: Verbraucht Laufzeit nur in 1. & letzter Iteration
  - ☞ Nur 2 Takte Schleifen-*Overhead*

Problem der *wrap around* Arithmetik (1)

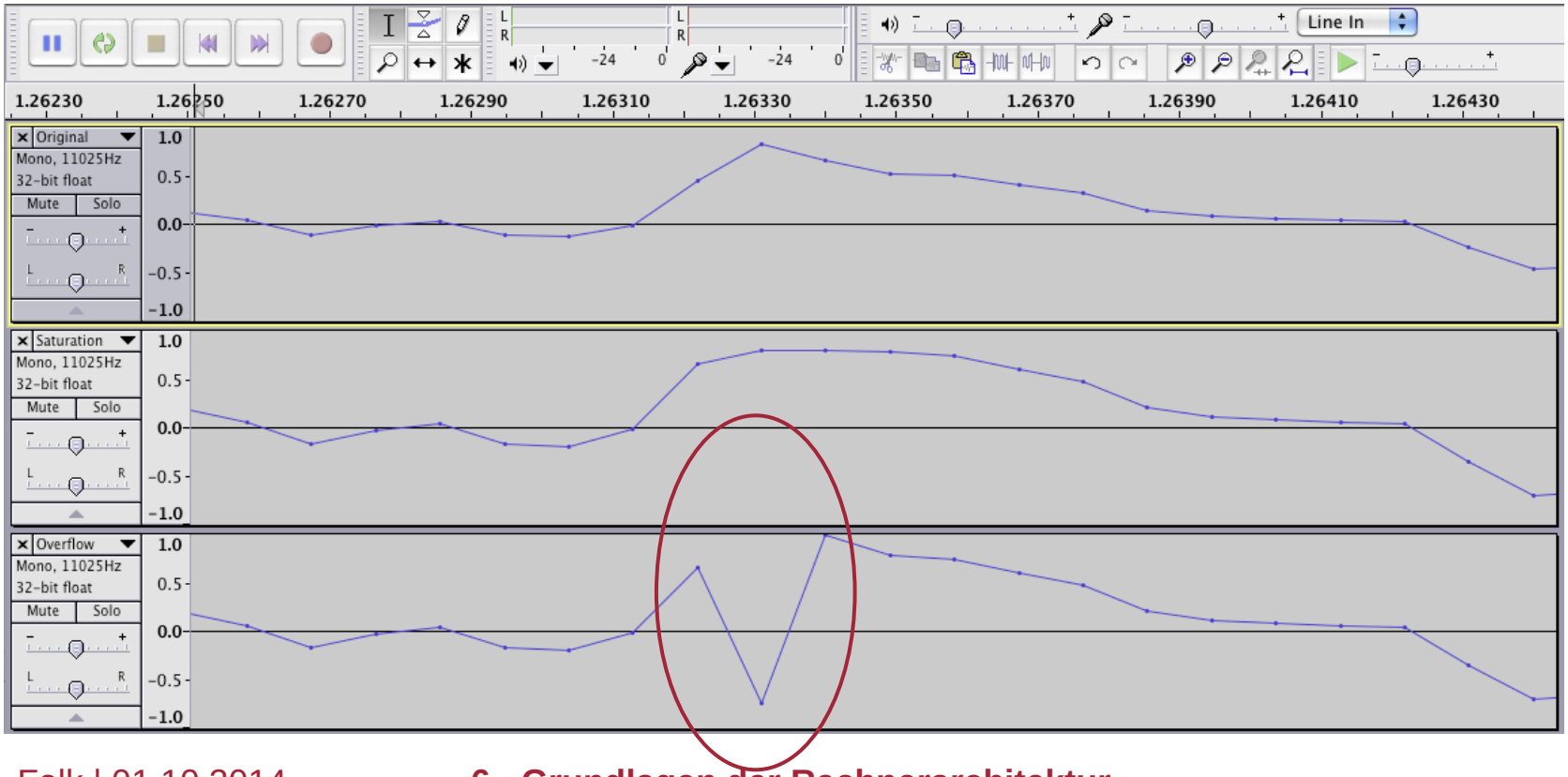
Standard-Arithmetik führt bei Über-/Unterlauf zu *wrap around*

– Problem: Ergebnisse mit *wrap around* sind…
  …nicht nur **falsch**
  …sondern extrem **unplausibel** / nicht einmal nahe der korrekten Lösung

– Der notwendigerweise entstehende Fehler ist **maximal** (signifikanteste Bitstelle 2^n geht verloren), nicht minimal! Beispiel:
  (4 bit, 2er-Kompl.): $|(7 +_{wrap} 1) – (7 +_{exact} 1)| =$
  $|(0111_{(2)} +_{wrap} 0001_{(2)}) – 8)| =$
  $|\mathbf{1}000_{(2)} – 8| =$
  $|-8 – 8| = 16$

Problem der *wrap around* Arithmetik (2)

– Große Fehler zwischen (mit Überlauf) berechnetem und tatsächlichem Ergebnis besonders dramatisch bei Signalverarbeitung (Verstärkung eines Audiosignals / Helligkeitsänderung eines Bildpunktes)



Kleinerer Fehler bei Sättigungsarithmetik

Sättigungsarithmetik *(saturated arithmetic)* für Addition oder Multiplikation liefert **bei Über-/Unterlauf den jeweils maximal/minimal darstellbaren Zahlenwert**.

Beispiele

– Betragsdarstellung (4 bit, vorzeichenlos):

$8 +_{sat} 8 = 1000_{(2)} +_{sat} 1000_{(2)} =$ $7 +_{sat} 11 \rightarrow 15 \neq 18$

$\mathbf{1}0000_{(2)} \rightarrow 1111_{(2)} = 15 \neq 16$

– Zweierkomplementdarstellung (4 bit, vorzeichenbehaftet):

$7 +_{sat} 1 = 0111_{(2)} +_{sat} 0001_{(2)} =$ $-5 -_{sat} 7 \rightarrow -8 \neq -12$

$\mathbf{1}000_{(2)} \rightarrow 0111_{(2)} = 7 \neq 8$

Insbesondere gibt es bei Sättigungsarithmetik keine Vorzeichenumkehr!

Weiteres Beispiel

–	a		0111	
	b	+	1001	
	Standard *wrap around* Arithmetik		(1)0000	
	Sättigungsarithmetik		1111	
	(a+b)/2: korrekt		1000	
	wrap around Arithmetik		0000	
	Sättigungsarithmetik mit >>		0111	„fast richtig“

- Geeignet für DSP- / Multimedia-Anwendungen
  - Durch Überläufe ausgelöste *Interrupts*
    - ☞ Echtzeitbedingungen verletzt…?
  - Genaue Werte ohnehin weniger wichtig
  - *wrap around* Arithmetik liefert schlechtere Ergebnisse

Sättigungsarithmetik: Bewertung

Vorteil

– Plausible Ergebnisse bei Bereichsüberschreitungen

Nachteile

– Aufwändiger in der Berechnung
– Assoziativität etc. sind verletzt

Sättigungsarithmetik und „Standardarithmetik“ können auf DSPs in der Regel wahlweise benutzt werden (es existieren entsprechende Befehlsvarianten)

„Sättigung“ im IEEE 754 *floating point* Standard:

– Bei Über-/Unterlauf entsteht ± „unendlich“ als Ergebnis
– Weitere Operationen ändern diesen Wert nicht mehr!

DSPs: Realzeiteigenschaften

Das Zeitverhalten des Prozessors sollte vorhersagbar sein!

Eigenschaften, die Probleme verursachen:

- Zugriff auf gemeinsame Ressourcen
  - *Caches* mit Ersetzungsstrategien mit problematischem Zeitverhalten
  - *Unified caches* für Code und Daten gleichzeitig (Konflikte zwischen Daten und Befehlen)
  - Fließbänder *(pipelines)* mit *stall cycles („bubbles“)*
  - *Multi-cores* mit unvorhersagbaren Kommunikationszeiten
- Sprungvorhersage, spekulative Ausführung
- *Interrupts*, die zu jedem Zeitpunkt möglich sind
- Speicherauffrischen *(refresh)* zu jeder Zeit
- Befehle mit datenabhängigen Ausführungszeiten

☞ **So viele dieser Eigenschaften wie möglich vermeiden**

Multimedia-Prozessoren

Eigenschaften

- Optimiert z.B. für Bild- & Tonverarbeitung
- Bekannte kommerzielle Produkte:
  Intel MMX, SSE oder SSE2; AMD 3DNow!; Sun VIS;
  PowerPC AltiVec; HP MAX
- Motivation: Multimedia-Software nutzt oft nicht die gesamte Wortlänge eines Prozessors (d.h. **`int`**), sondern nur Teile (z.B. **`short`** oder **`char`**).
- SIMD-Prinzip: *Single Instruction, Multiple Data*
- Parallele Bearbeitung mehrerer „kleiner” Daten durch 1 Befehl

SISD vs. SIMD-Ausführung

Aufgabe: Addiere zweimal je 2 `short`-Variablen

– *SISD-Prinzip (Single Instruction, Single Data):*

Lade erste 2 Summanden in Register,
`int`-Addition,
Lade zweite 2 Summanden in Register,
`int`-Addition

☞ Kosten: 2 volle Additionen

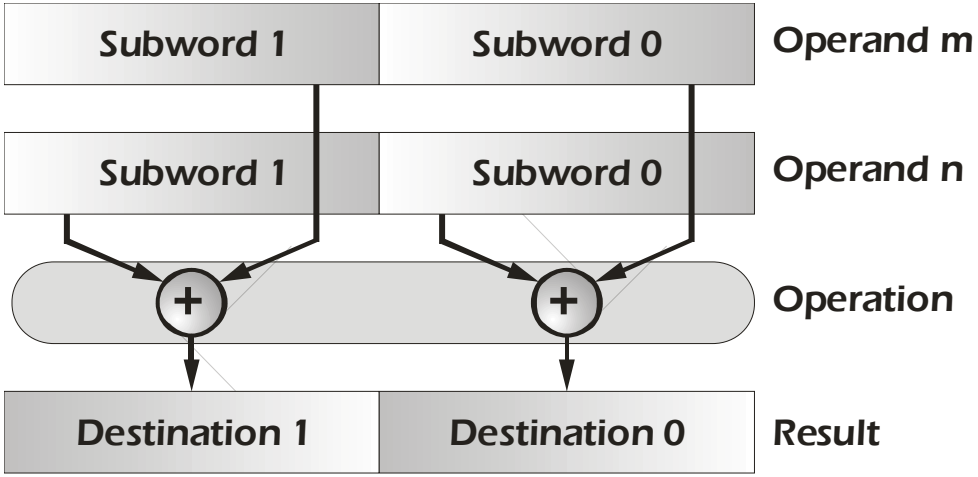
– *SIMD-Prinzip (Single Instruction, Multiple Data):*

Lade erste 2 Summanden in obere Halb-Register,
Lade zweite 2 Summanden in untere Halb-Register,
SIMD-Addition

☞ Kosten: 1 Addition

Veranschaulichung SIMD-Addition

SIMD Halbwort-Addition:



– SIMD-Instruktionen auch für Viertel-Worte gebräuchlich:
  ☞4 parallele **`char`**-Additionen bei 32-bit Prozessor

Very Long Instruction Word (VLIW): Motivation

Steigerung der Rechenleistung von Prozessoren stößt an Komplexitätsschranken

Zunehmend schwieriger, die Rechenleistung bei sequentieller Programmausführung weiter zu steigern

– Hoher Anteil an *Pipelining*-Fehlern bei Silizium-Bugs
– Superskalare Prozessoren (d.h. Prozessoren, die > 1 Befehl pro Takt starten) sind schwierig zu realisieren:

„the only ones in favor of superscalar machines are those who haven‘t built one yet“

[Bob Rau, hp Labs]

[www.trimaran.org]

Very Long Instruction Word (VLIW)

VLIW: Performance-Steigerung durch erhöhte Parallelität

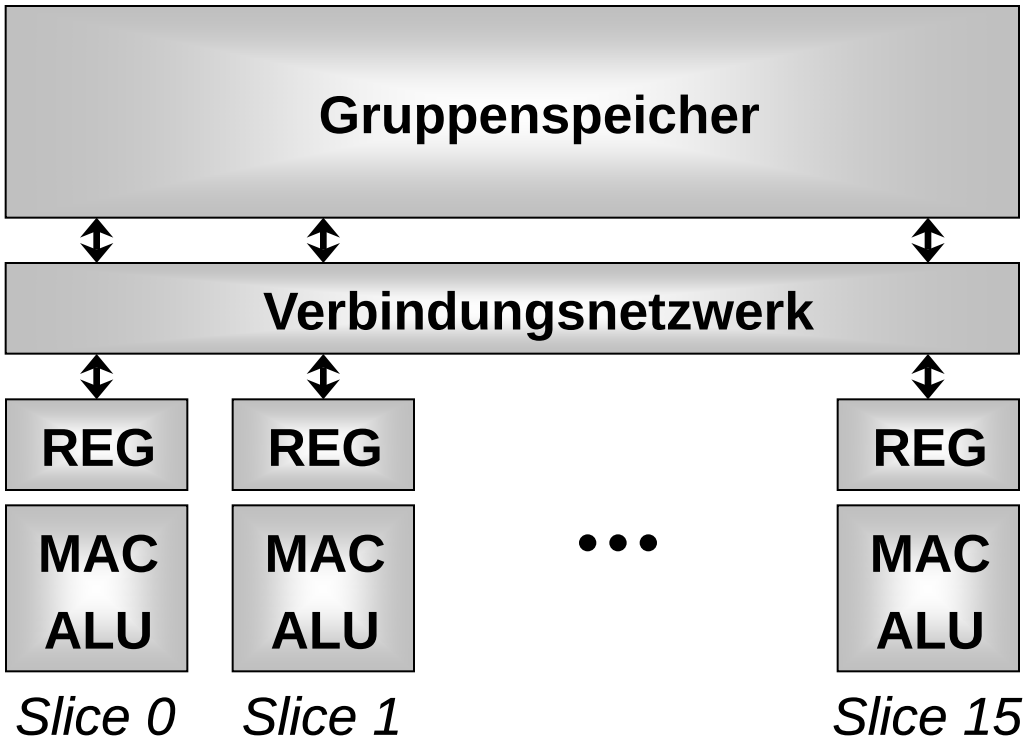
Konventionelle Prozessoren
- 1 integer-ALU
- 1 Multiplizier-Einheit
- 1 (heterogenes) Register-File

VLIW-Prozessoren
- *n* integer-ALUs
- *n* Multiplizier-Einheiten
- *n* (heterogene) Register-Files
- Verbindungsnetzwerk

Parallelität in VLIW-Befehlssätzen
- Bildung eines Befehlspaketes konstanter Länge;
- alle Befehle im Paket sind parallel auszuführen;
- Compiler bestimmt über Parallelität, indem parallel auszuführende Operationen in einem Befehlspaket gruppiert werden

☞ Sehr lange Befehlswörter (64, 128 Bit oder mehr)

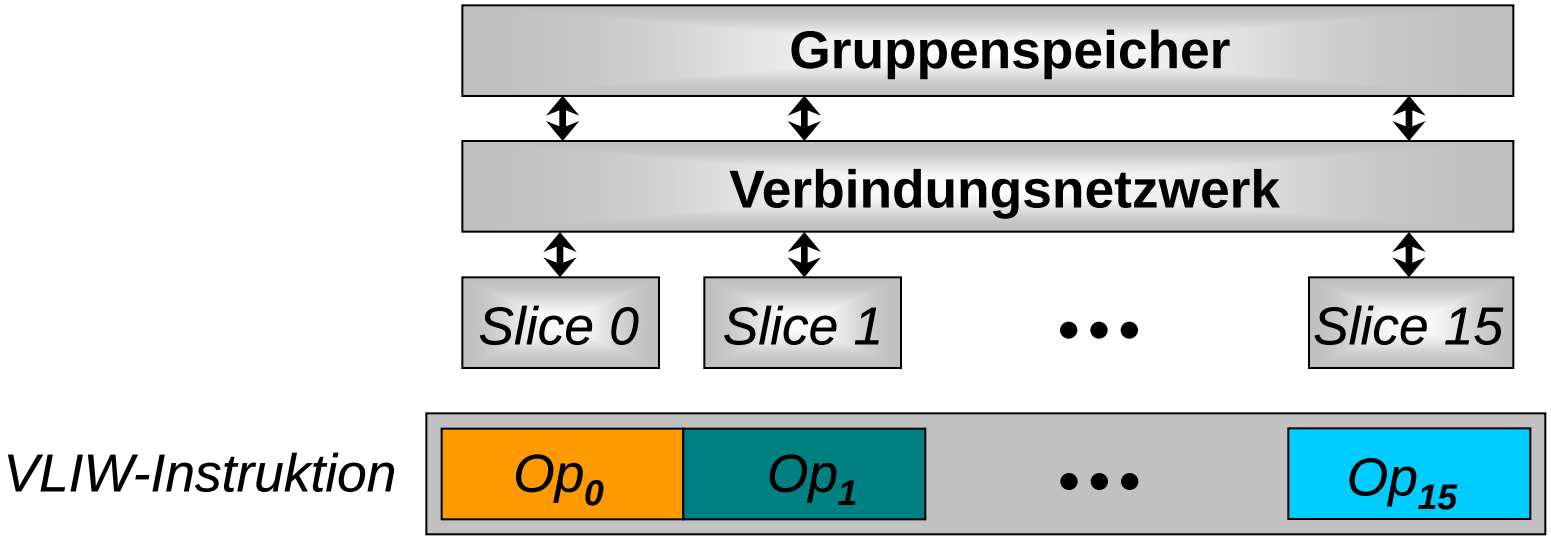
Beispiel: M3 VLIW-Prozessor



VLIW-Befehlswort

- 1 Befehlswort enthält 1 VLIW-Instruktion
- 1 VLIW-Instruktion enthält *n* VLIW-Operationen
- Jede Operation steuert genau eine *Functional Unit (FU)*
- Starre Zuordnung von Operationen im Befehlswort zu FUs:

  Operation 0 ↔ FU 0, Operation 1 ↔ FU 1, ...



VLIW: Parallelisierbarkeit (1)

Woher kommt das Potential für Parallelisierungen?

Programme führen Instruktionen prinzipiell entlang eines Kontrollflusses aus

– Operationen manipulieren Daten sequentiell
– Arbeiten also i.d.R. auf Ergebnissen *vorheriger* Operationen

Es existieren daher häufig Datenabhängigkeiten zwischen aufeinander folgenden Operationen

– Solche Operationen können nicht parallel ausgeführt werden!
– Wo also Parallelisierungspotential finden?

Auch: In der Regel nur begrenzte Zahl funktionaler Einheiten

☞ Nur bestimmte Operationstypen parallel möglich

VLIW: Parallelisierbarkeit (2)

Beispiel (auf Hochsprachenebene)

```
x = (a + b) * (a – c)
```

Realisierung (sequentiell)

```
x0 = (a + b)
x1 = (a – c)
x = x0 * x1
```

☞ Verschränkung der Berechnung teilweise möglich

Realisierung

[ALU Nr. 1]

```
x0 = (a + b)
x = x0 * x1
```

[ALU Nr. 2]

```
x1 = (a – c)
```

VLIW: Parallelisierbarkeit (3)

Problem: Kontrollfluss innerhalb von Programmen häufig nicht linear!

Beispiel:
```
if (x > y) x = c + d;
y = a + b;
```

- Zwar keine Datenabhängigkeiten, aber trotzdem nicht parallelisierbar.

Beispiel:
```
for (i = 0; i < n; i++)
  x[i] += a;
```

- Es existieren keine Datenabhängigkeiten zwischen aufeinander folgenden Ausführungen des Schleifenrumpfes
- Prinzipiell könnten Operationen parallelisiert werden

Fazit: Parallelisierung von Operationen kann nicht ohne Betrachtung des Kontrollflusses erfolgen!

VLIW: Parallelisierbarkeit von Schleifen

Problem: Wie potentielle Parallelisierbarkeit von Anweisungen in Schleifenrümpfen ausnutzen?

☞ „Abrollen“ des Schleifenrumpfs *(loop unrolling)*

Schleife:

```
for (i = 0; i < n; i++)
  x[i] += a;
```

Abgerollt (4-mal):

```
for (i = 0; i < n; i+=4) {
  x[i] += a;
  x[i+1] += a;
  x[i+2] += a;
  x[i+3] += a; }
```

☞ Instruktionen im abgerollten Schleifenrumpf können parallelisiert werden.

Im allgemeinen (falls **n** mod 4 != 0) zusätzlicher Code erforderlich

VLIW: Diskussion (1)

Vergrößerung der Codegröße

- Gründe
  - Wegen des Abrollens von Schleifen (um mehr Parallelisierungspotential zu schaffen)
  - Unbenutzte funktionale Einheiten
    unbenutzte Teile des VLIW-Befehlsworts (NOPs)
- Mögliche Gegenmaßnahmen
  - Komprimierung des Binärcodes
  - Kein festes VLIW, sondern Kodierung parallel auszuführender Befehle mit variabler Codelänge
    (☞ sog. EPIC-Befehlssätze, Itanium-64)

VLIW: Diskussion (2)

Binärkompatibilität

– Generierung der Befehlscodierung macht expliziten Gebrauch von Wissen über **interne Architektur** des Prozessors (insbes. Anzahl funktionaler Einheiten, aber auch zum *Pipelining*)

– Code ggf. **nicht** auf veränderter interner Architektur lauffähig!

– Hier: Parallelisierung durch Compiler (vs. Hardware)
Widerspricht eigentlich der Idee einer externen Architektur (Befehlssatz) als Abstraktion von Realisierung und Schnittstelle zum Programmierer (vgl. Definition Rechnerarchitektur nach Amdahl)

VLIW: Diskussion (3)

Erzeugung ausreichender Parallelität durch Compiler

– Erfolgt auf Ebene einzelner Maschineninstruktionen *(instruction-level parallelism, ILP)*

– Sprünge und Verzweigungen des Kontrollflusses verhindern oft Parallelisierung

– Daher versuchen Compiler üblicherweise, die Parallelität innerhalb sog. Basisblöcke zu erhöhen.
  Ein **Basisblock** $B = (I_1, \ldots, I_n)$ ist eine Befehlssequenz maximaler Länge,
  – so dass B nur durch die erste Instruktion I_1 betreten wird, und
  – B nur durch die letzte Instruktion I_n verlassen wird

☞ Parallele Funktionseinheiten können ggf. nicht voll ausgenutzt werden

Netzwerkprozessoren

Motivation

- Viele aktive Komponenten in heutigen LAN/WANs
- Extrem hohe Anforderungen an die Geschwindigkeit bei begrenztem Energieaufwand
- Aktive Komponente = Spezialrechner (mit Netzwerkprozessor)

Netzwerkprozessoren: Aufgaben (1)

– Klassifikation/Filterung von Netzwerkpaketen
  (z.B. Überprüfung auf Übertragungsfehler [CRC], *Firewall*, …)

– Paketweiterleitung *(IP forwarding)*
  (für Routing zwischen verschiedenen Teilnetzen)

– Adressübersetzung zwischen globalen und lokalen/privaten Adressbereichen (z.B. *IP masquerading*, virtuelle Web-Services, …)

Netzwerkprozessoren: Aufgaben (2)

- *Virtual Private Networks (VPN)*, d.h. gesicherte private Verbindungen über öffentliches, ungesichertes Netz (Kryptographie [DES] und Authentifizierung [MD5] erforderlich)
- *Intrusion Detection:* Erkennung von Angriffsszenarien auf Paketebene durch Signaturvergleich
- *Deep packet inspection:* Netzwerkkomponenten analysieren Informationen jenseits des Headers
- Daten-Umcodierung: Formatumwandlung für Multimediadaten innerhalb des Netzwerks

Netzwerkprozessoren: Verarbeitungsprinzipien

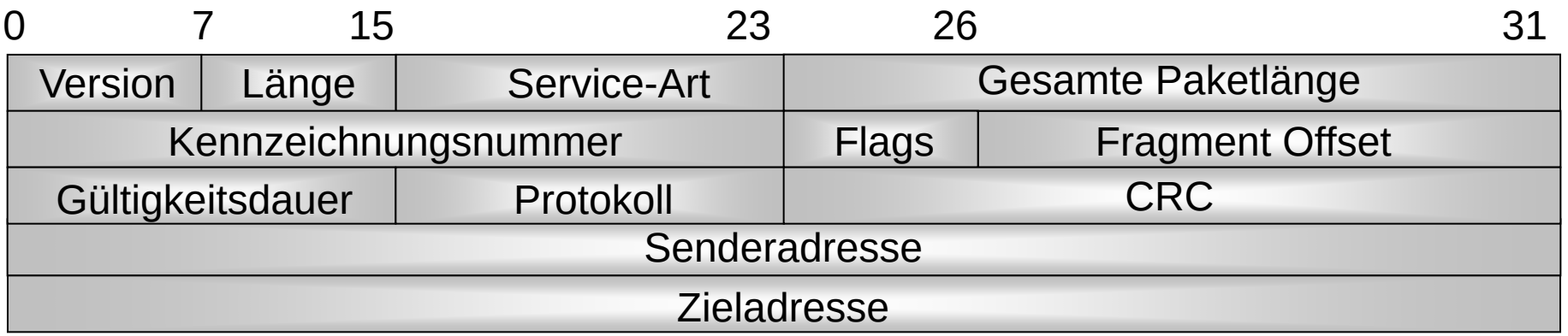
- Paketlänge (z.B. 64 Bytes) bestimmt bei maximaler Auslastung der Bandbreite die verfügbare Bearbeitungszeit
  Beispiel: Gigabit-Ethernet
  max. Bandbreite 1Gb/s, d.h. für ein Paket à 64 Bytes ☞ ca. 476 ns
- Trotzdem: Konstante Verzögerung (im Umfang der Paketbearbeitungszeit) entsteht pro aktiver Komponente entlang der Route zwischen zwei vernetzten Komponenten
- Performancesteigerung durch Parallelisierung (mehrere bzw. sehr viele Verarbeitungseinheiten in aktiven Netzwerkkomponenten) und **nicht** durch *Pipelining* (Verschränkung der Verarbeitungsschritte)

Netzwerk-Protokolle

Kommunikation zwischen entfernten Prozessoren

- Kommunikationsmedium fehleranfällig
- Nutzdaten werden in Pakete zerteilt
- Pakete werden mit Zusatz-Informationen versehen *(Header)*

Beispiel IPv4-*Header*:



Bit-Pakete

Bit-Pakete in Protokoll-*Headern*

- *Header* zerfallen in Bereiche unterschiedlicher Bedeutung
- Solche Bit-Bereiche sind nicht nach Prozessor-Wortbreiten angeordnet
- Bit-Paket:
  - Menge aufeinanderfolgender Bits
  - beliebiger Länge
  - an beliebiger Position startend
  - u.U. Wortgrenzen überschreitend

☞ Effiziente Manipulation von Daten auf Bit-Ebene notwendig!

Netzwerkprozessoren: Verarbeitungsprinzipien

- Adressierungsarten für Bit-Pakete
  - Übliche Rechner/Speicher sind byte- oder wort-adressierbar
- ALU-Operationen für Bit-Pakete
  - Übliche Architekturen unterstützen nur Registeroperanden
  - ggf. Halbwort- bzw. Byte-Operationen möglich (manchmal gepackt)
- Operanden/Operationen deutlich fein-granularer als in „Universal“-Rechnern üblich

Bit-Pakete

Network Processing Units (NPUs)

– Software zur Protokollverarbeitung:
  Hoher Code-Anteil für Verarbeitung von Bit-Paketen

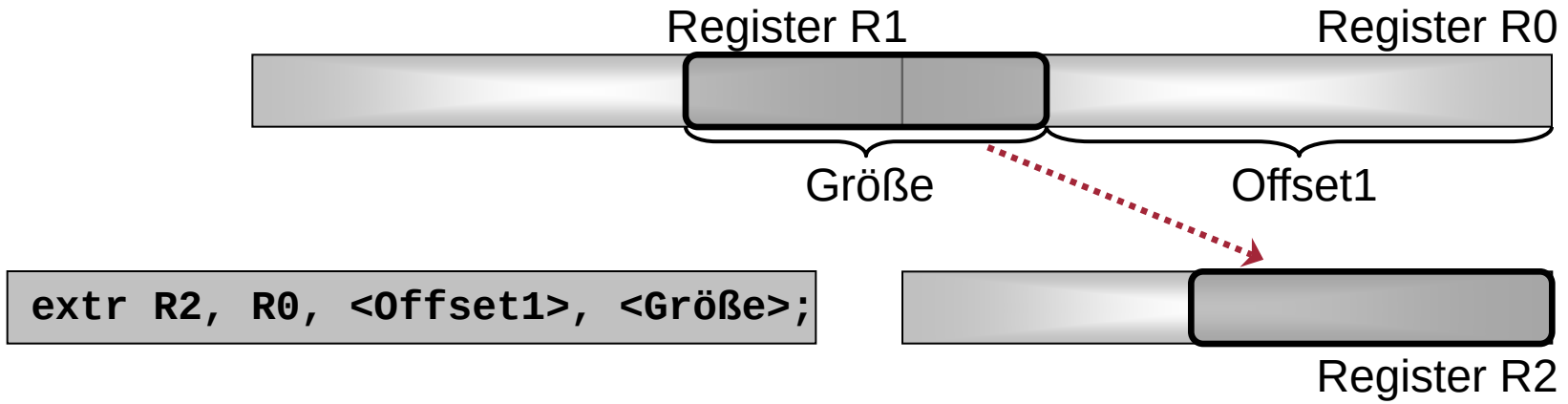
– Typischer C-Code (GSM-Kernel, TU Berlin):

```
xmc[0] = (*c >> 4) & 0x7;
xmc[1] = (*c >> 1) & 0x7;
xmc[2] = (*c++ & 0x1) << 2;
xmc[2] |= (*c >> 6) & 0x3;
xmc[3] = (*c >> 3) & 0x7;
```

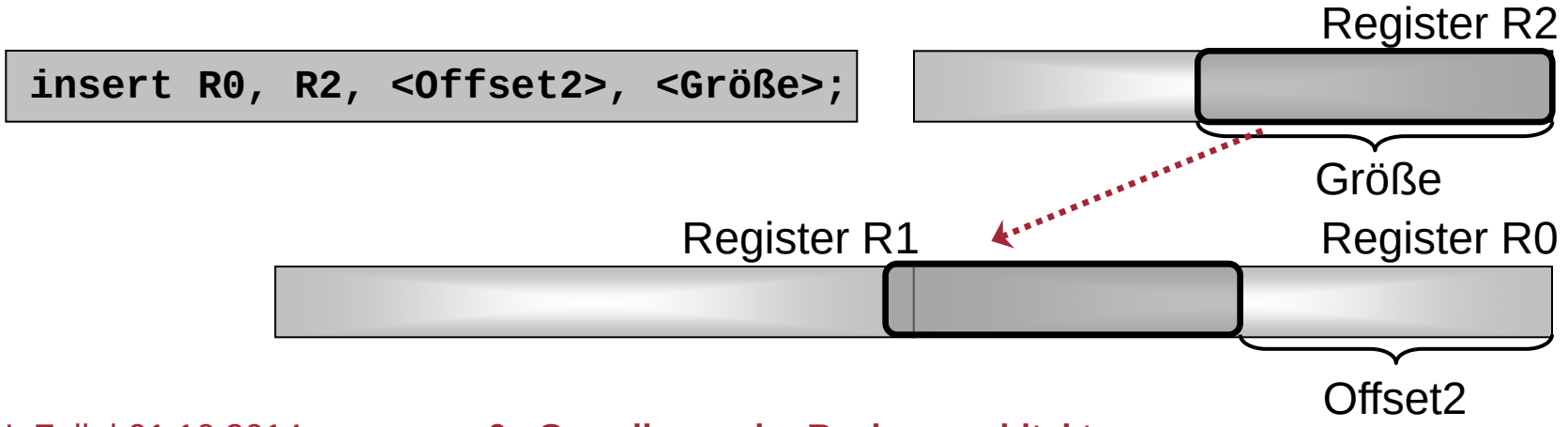
– Befehlssatz von NPUs:
  Spezial-Instruktionen zum Extrahieren, Einfügen & Bearbeiten von Bit-Paketen

Operationen auf Bit-Paketen

Extrahieren von Bit-Paketen



Einfügen von Bit-Paketen



Roter Faden

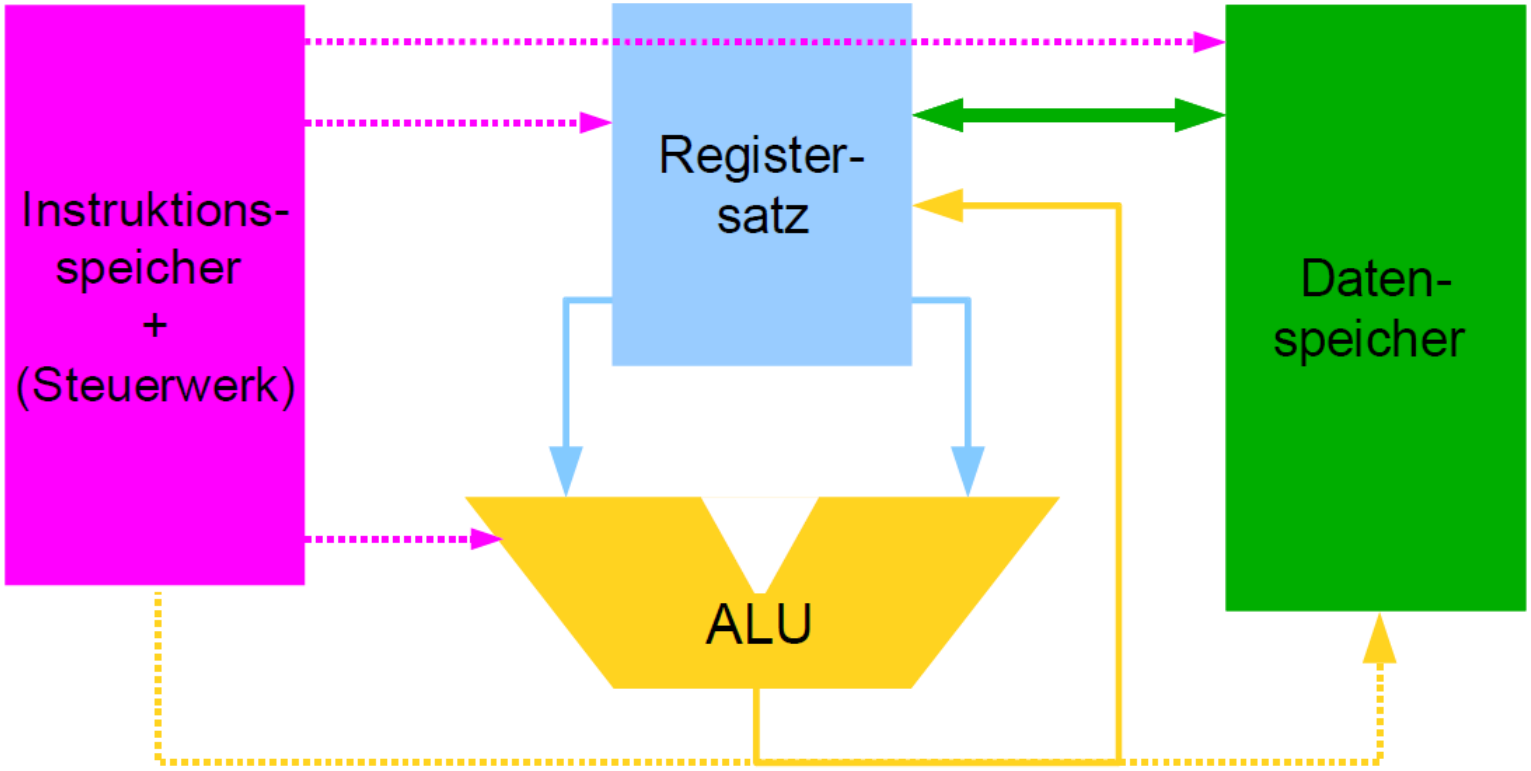
6. Grundlagen der Rechnerarchitektur

- Grundbegriffe der Rechnerarchitektur
- Programmiermodelle / die Befehlsschnittstelle
  - Adressierungsarten (Referenzstufen 0, 1, 2, *n*)
  - 3-, 2-, 1½-, 1-, 0-Adressmaschinen
  - CISC & RISC
  - Digitale Signalprozessoren (DSPs)
  - Multimedia-Befehlssätze
  - *Very Long Instruction Word*-Maschinen (VLIW)
  - Netzwerk-Prozessoren (NPUs)
- Aufbau einer MIPS-Einzelzyklusmaschine
- Fließbandverarbeitung / *Pipelining*
- Dynamisches *Scheduling*

MIPS Einzelzyklusmaschine

Vereinfachte Sicht einer MIPS

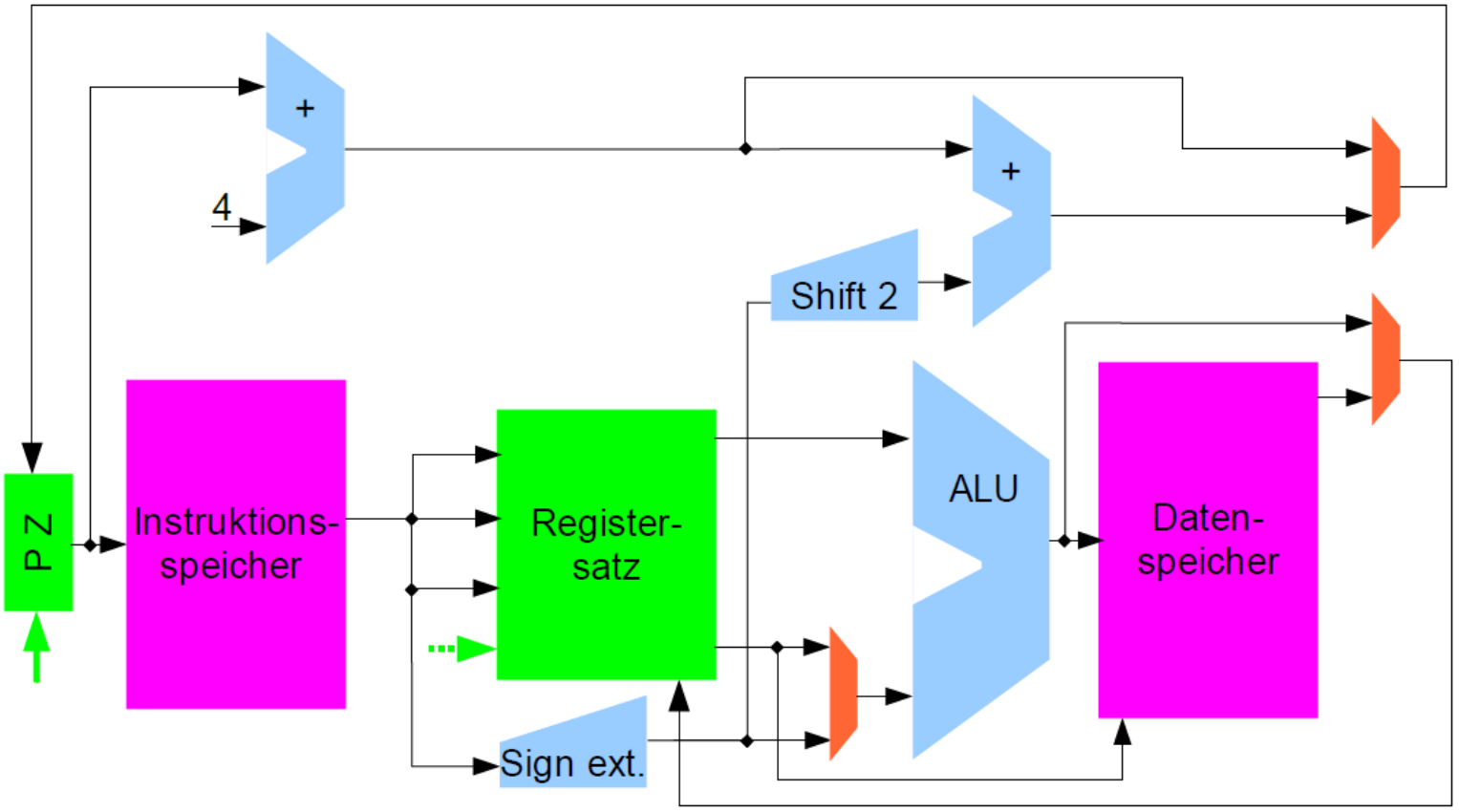
- Informationsflüsse
  - Steuersignale, Speicherdaten, Registerdaten, Adressen, Instruktionen



Einzelzyklusmaschine etwas genauer

Komponenten

– Register, ALUs, Multiplexer, Speicher



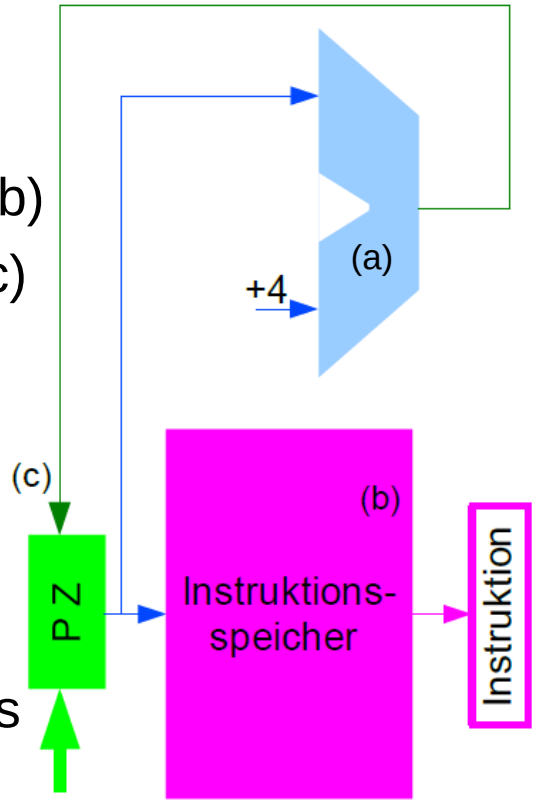
Befehlsholphase *(Instruction Fetch)*

Programmzähler (32 bit, engl. *Program Counter, PC*)
- Speichert und liefert die aktuelle auszuführende Instruktionsadresse
- an den Instruktionsspeicher (b) und das Addierwerk (a)
- übernimmt die Adresse der Folgeinstruktion (c)

Addierwerk
- Fortlaufende Addition mit 4, da 4-Byte Instruktionen (b)
- Der neue Wert für PC wird im Register gespeichert (c)
- Die hintersten 2 Bit im PC sind immer Null

Instruktionsspeicher
- Liefert die auszuführende Maschineninstruktion (b)

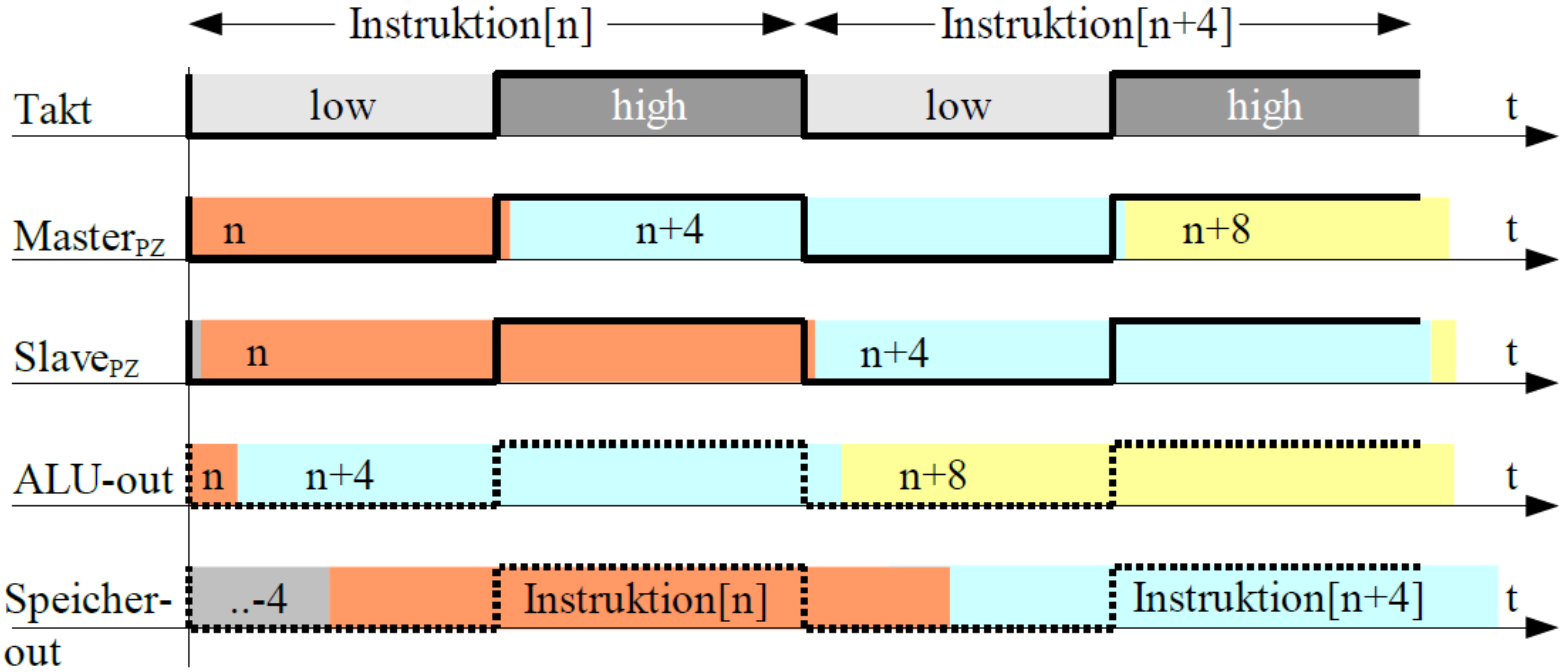
Instruktionswort (32 bit)
- Gelesener Wert erscheint am Ausgang des Speichers
- Instruktionsformat bestimmt den weiteren Ablauf



Zeitverhalten der Befehlsholphase

Programmzähler aus 32 *Master-Slave* Flip-Flops

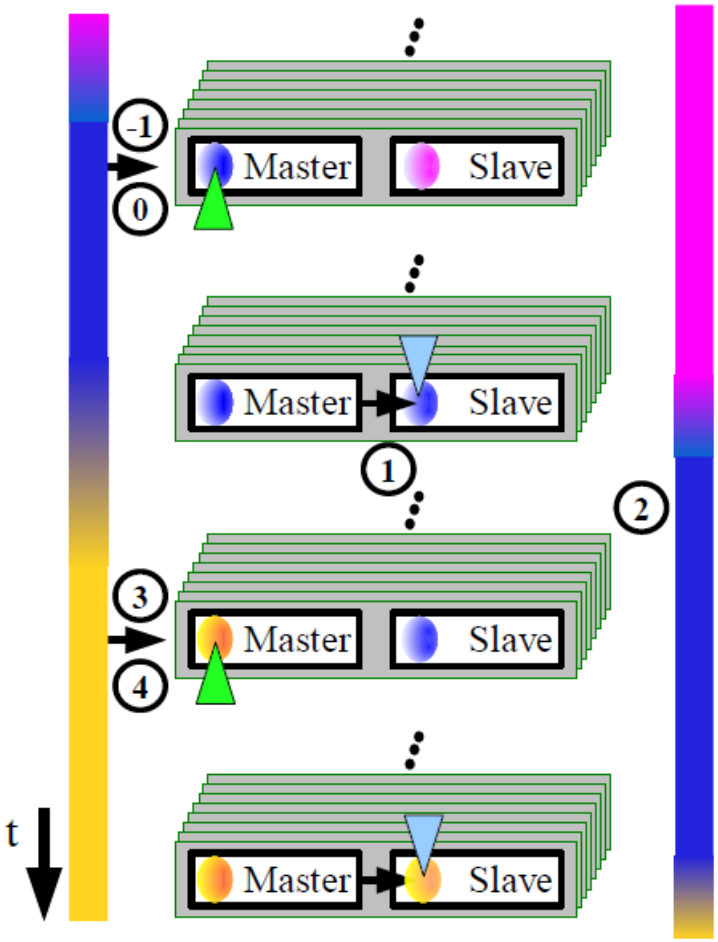
- Flankengesteuert lt. Hennessy/Patterson (Pegelsteuerung auch möglich)
- *Master* übernimmt neuen Wert bei steigender Taktflanke
- *Slave* liefert Instruktionsadresse bei fallender Taktflanke
- Instruktionszyklus beginnt mit fallender Taktflanke



Anschaulichere Darstellung

Ablauf

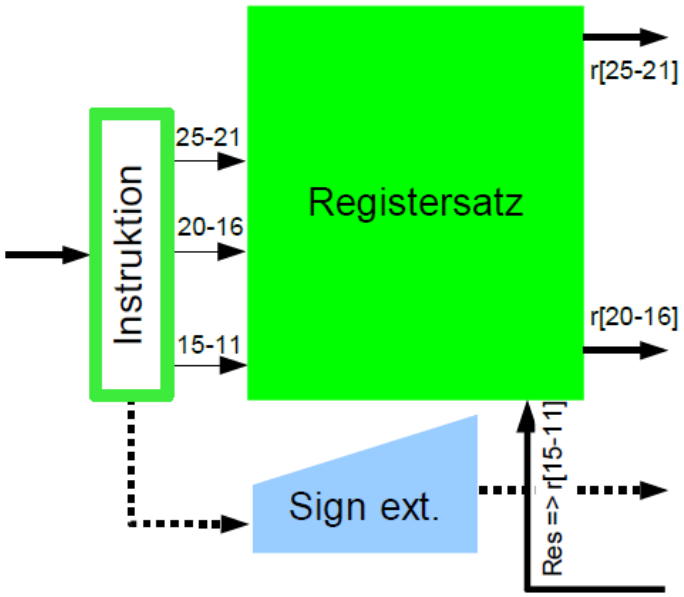
-1. Stabiler Wert am PC-Eingang
0. Steigende F.: *Master* übernimmt Wert
1. Fallende F.: *Slave* übernimmt Wert
2. Weitergabe an Speicher & Addierer
3. Addierer ist fertig, neuer PZ stabil
4. Steigende Flanke, *Master* übernimmt…



Decodierphase *(Instruction Decode)*

Z.B. R-Format Instruktion

- Zwei Register lesen, eines schreiben
- Gelesene Register weiter zur ALU
- Drei Instruktionsfelder à 5 Bit
- Resultat zurück von ALU



Quell-/Input-Register für die ALU

- ALU ist die Stufe nach dem Registersatz
- rt = IR[20-16] selektiert Register[rt] zur ALU
- rs = IR[25-21] selektiert Register[rs] zur ALU

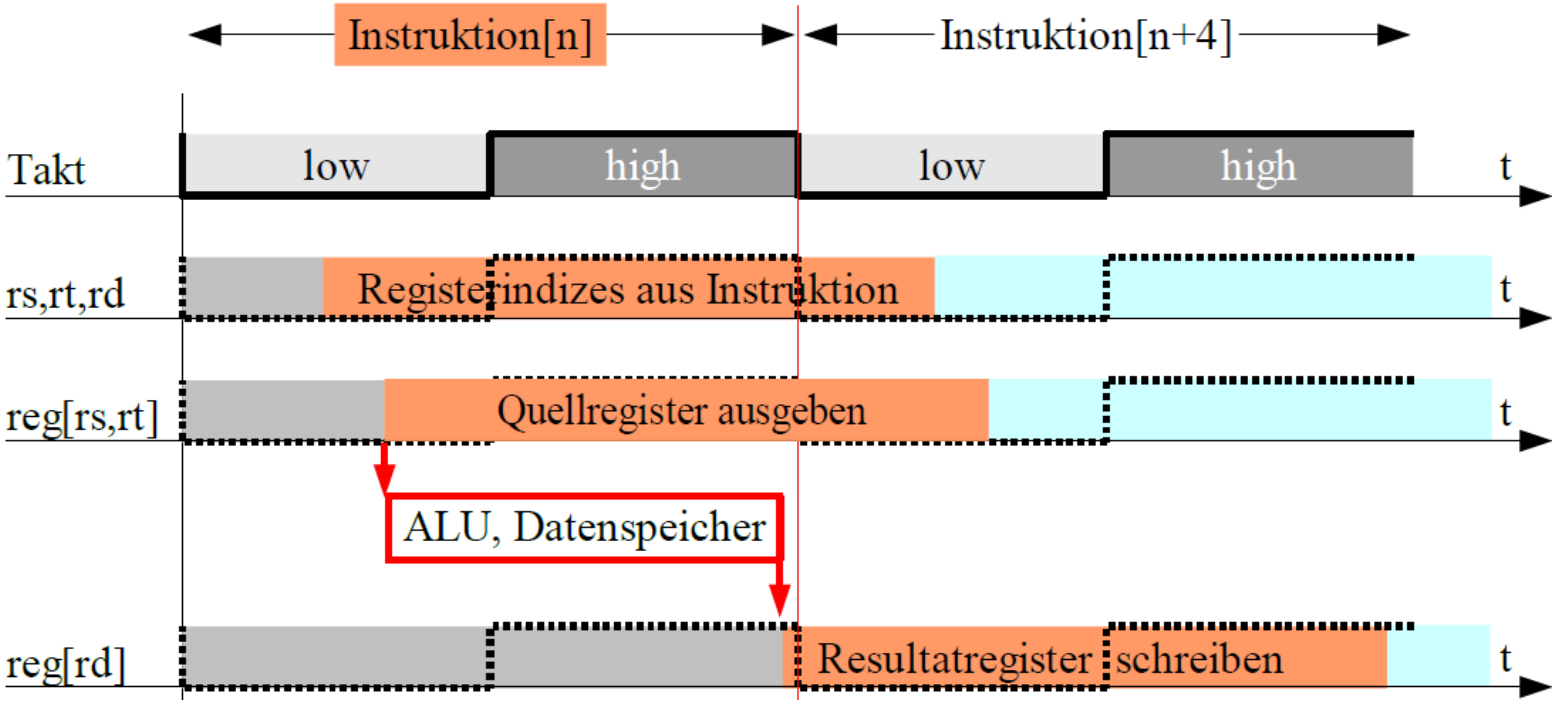
Zielregister für Resultat von ALU

- rd = IR[15-11] wählt Register[rd] für Resultat

Zeitverhalten der Befehlsdecodierung

Ansteuerung des Registersatzes

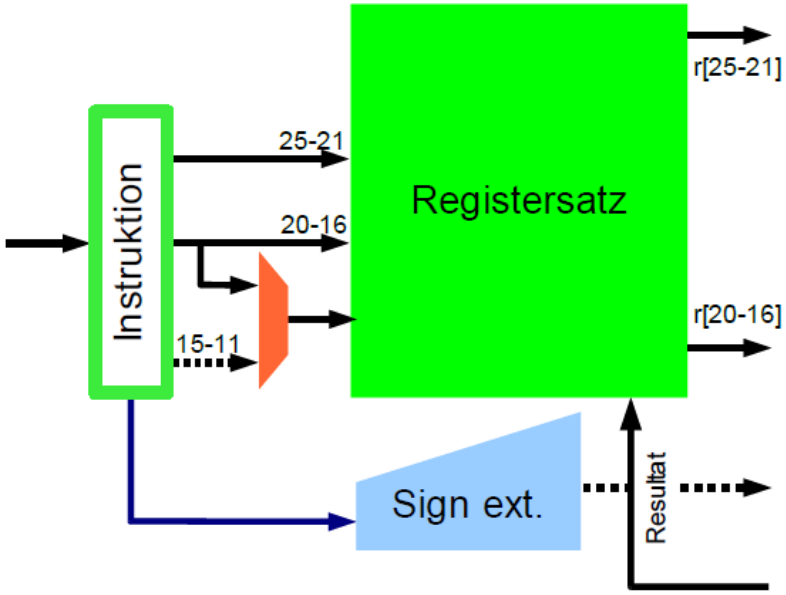
- Register immer auslesen (kein Takt) und Transport zur ALU
- Schreiben des Zielregisters Register[rd] am Ende der Taktperiode
- Zeit für Speicherzugriff und für die primäre ALU muss eingeplant werden
- Ausgabe des Instruktionsspeichers wird über die ganze Dauer gehalten



Decodierphase *(Instruction Decode)* (1)

Z.B. I-Format Instruktion

- Ein Basis-/Indexregister: rs = IR[25-21]
- Ein Ziel-/Quellregister: rt = IR[20-16]
- Direktoperand: imm = IR[15-0]

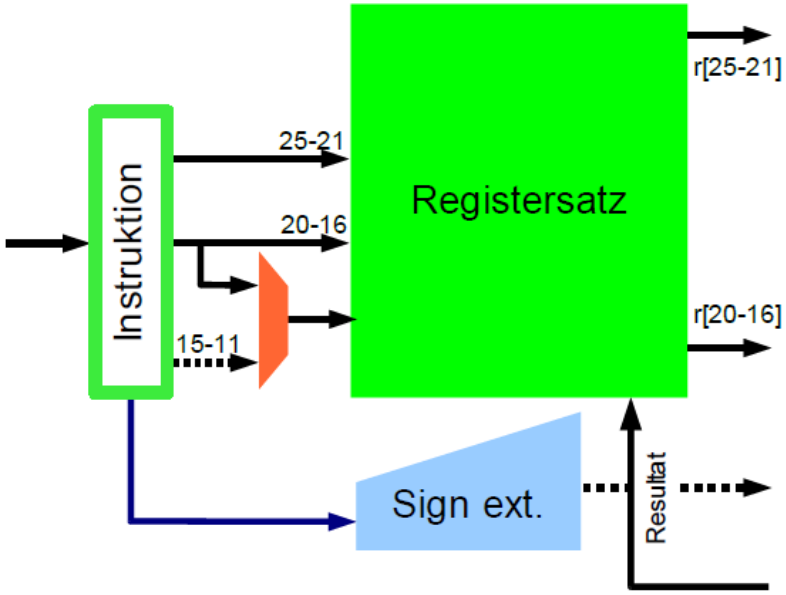


Umschalten von IR[20-16]

- Je nachdem ob R- oder I-Format vorliegt
- Quellregister bei *Store*-Instruktionen
- Zielregister bei *Load*-Instruktionen
- Zusätzlichen Multiplexer vorsehen

Decodierphase *(Instruction Decode)* (2)

Multiport-Registersatz



- Zwei gleichzeitige Lesezugriffe im selben Taktzyklus
- Kein Schreibzugriff bei *Store*-Operationen (Mem-Write)
- Zwei Lesebusse, ein Schreibbus zu den Registern

Vorzeichenerweiterung des Direktoperanden von 16 auf 32 Bit

- Erleichtert die Unterbringung kleiner Konstanten im Befehlswort
- Vom Steuerwerk aus abschaltbar für „*unsigned*“ Befehle

Binäres Format der Instruktionen

Formatbetrachtung mit Hilfe eines Testprogramms

```
        .text
Main:   add     $1,$0,$31                                   # R-Format
        # binär: 000000,00 000,11111 00001,000 00,100000
        #          opcode,    rs,   rt,   rd

        sub     $1,$31,$0
        # binär: 000000,11 111,00000 00001,000 00,100010

        and     $31,$0,$1
        # binär: 000000,00 000,00001 11111,000 00,100100

        lw      $7,0x0aff($3)                               # I-Format
        # binär: 100011,00 011,00111, 00001010 11111111
        #          opcode, index, dest,          immediate

        sw      $7,0x0aff($3)
        # binär: 101011,00 011,00111, 00001010 11111111
        #          opcode, index,  src,          immediate
```

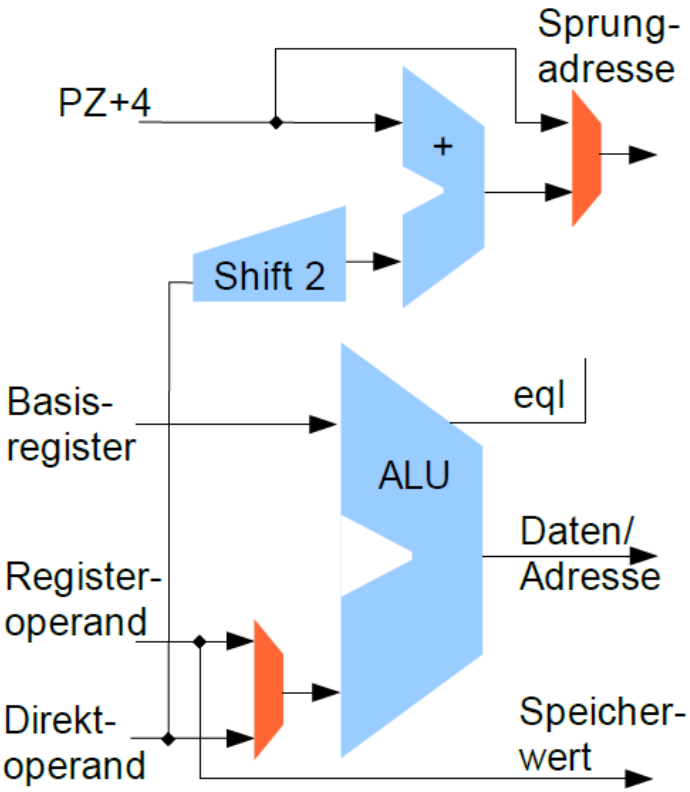
Ausführungsphase *(Execute)* (1)

ALU-Registeroperationen

- Operanden im Register oder als Direktoperand
- Üblicher Satz an ALU-Operationen
- Register **`$0`** liefert Wert 0

Adressierung von Variablen im Speicher

- Adressrechnung in der primären ALU
- Basisregister plus Direktoperand
- Registerinhalt lesen/schreiben



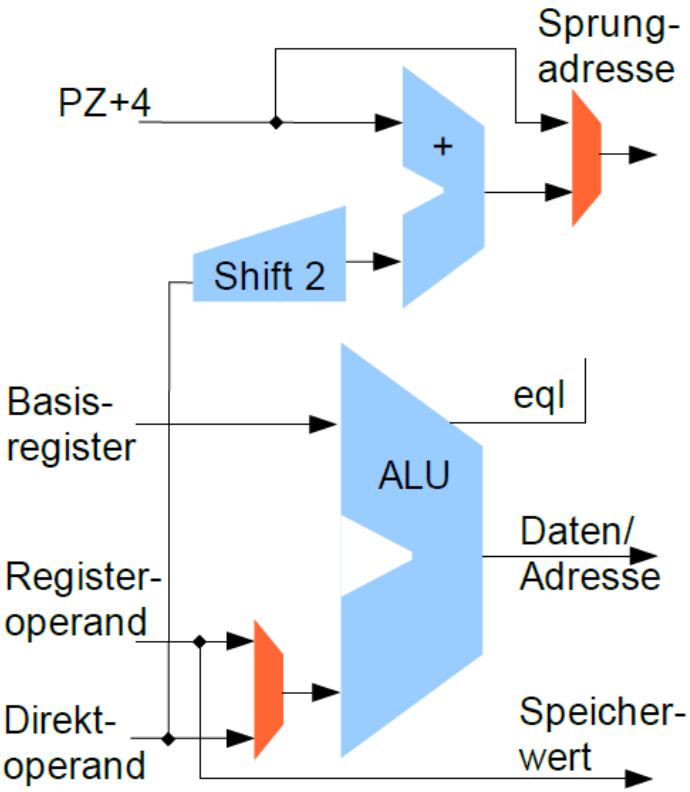
Ausführungsphase *(Execute)* (2)

***Load/Store*-Architektur**

- Speicheroperationen können keine Arithmetik
- also z.B. kein **`inc 0xaff0($7),0x0004`**
- ALU schon zur Adressberechnung benötigt

Separater Addierer zur Sprungzielberechnung

- Prüfschaltung auf Gleichheit zweier Register in der primären ALU („**`eql`**“)
- Bedingte Sprünge mit einem 16-bit Direktoperanden (**`beq $7,$8,loop`**)
- Maximal möglicher *Offset* von ±17 Bit nach einer 2-bit Verschiebung
- Unbedingte Sprünge mit 28-bit Adresse später

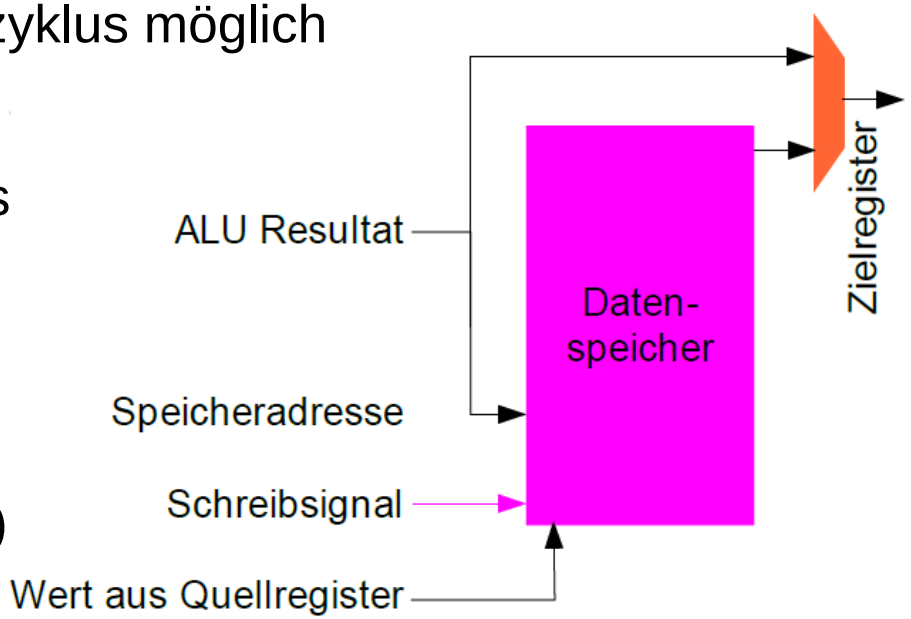


Speicherzugriff *(Memory Access)* (1)

Getrennte Speicher für Code & Daten
- Aktuelle Instruktion wird bis zum Ende des Gesamtzyklus gehalten
- Kein zweiter Zugriff im gleichen Taktzyklus möglich

Quellregister speichern, falls *Store*
- Speichersteuerung durch besonderes Schreibsignal

Zielregister laden
- Falls Ladebefehl aus dem Speicher
- Falls Rücksprungadresse (*PC-magic*)
- Falls Resultat aus ALU

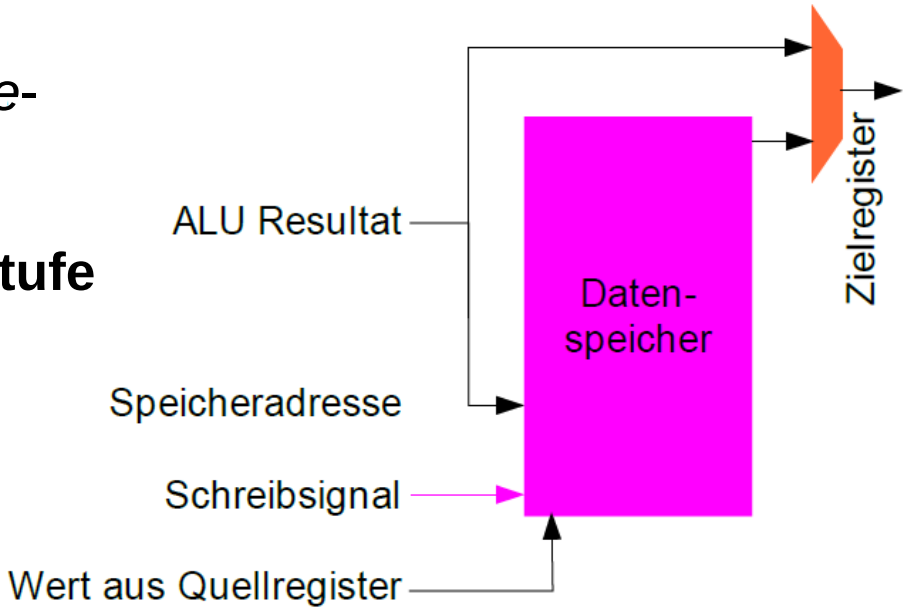


Speicherzugriff *(Memory Access)* (2)

ALU-Resultat nutzen

- Für „*Register Write-Back*“
- Als Datenspeicheradresse
- Nicht direkt speichern, wg. *Load/Store*-Architektur!

☞ **Multiplexer in nächste Prozessor-Stufe**



Register zurückschreiben *(Write Back)*

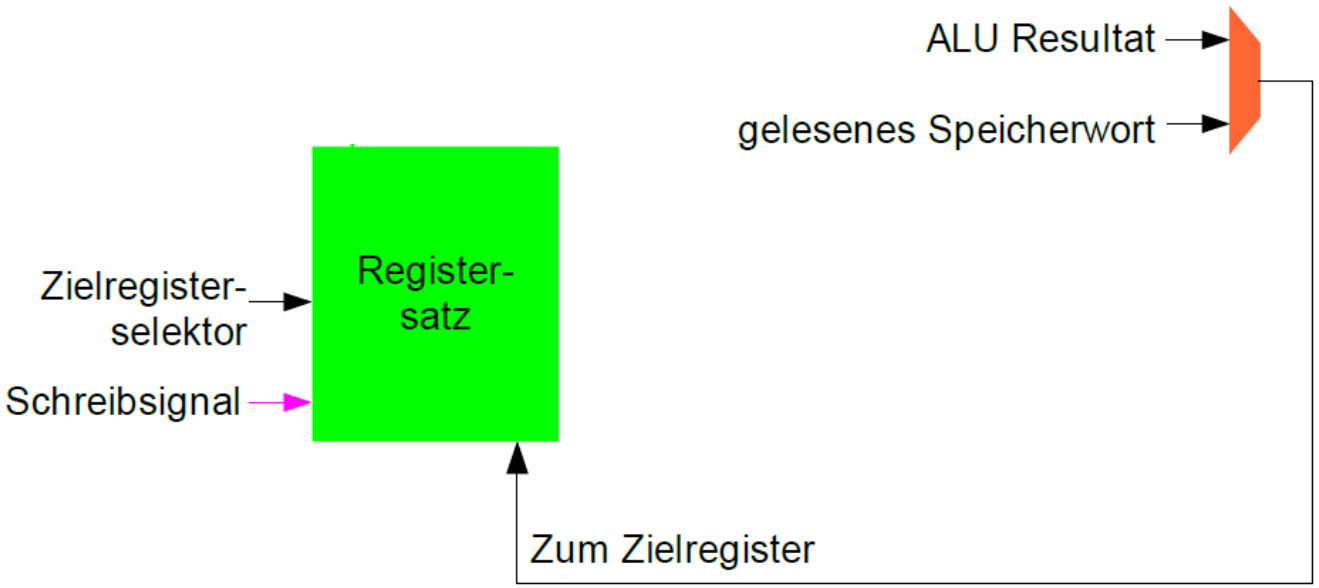
Nummer des Zielregisters (Zielregisterselektor)

– Stammt aus IR[15-11] oder IR[20-16], 5-bit Bereich für Werte 0-31

Steuersignal

– Zielregister zum Ende des Instruktionszyklus schreiben

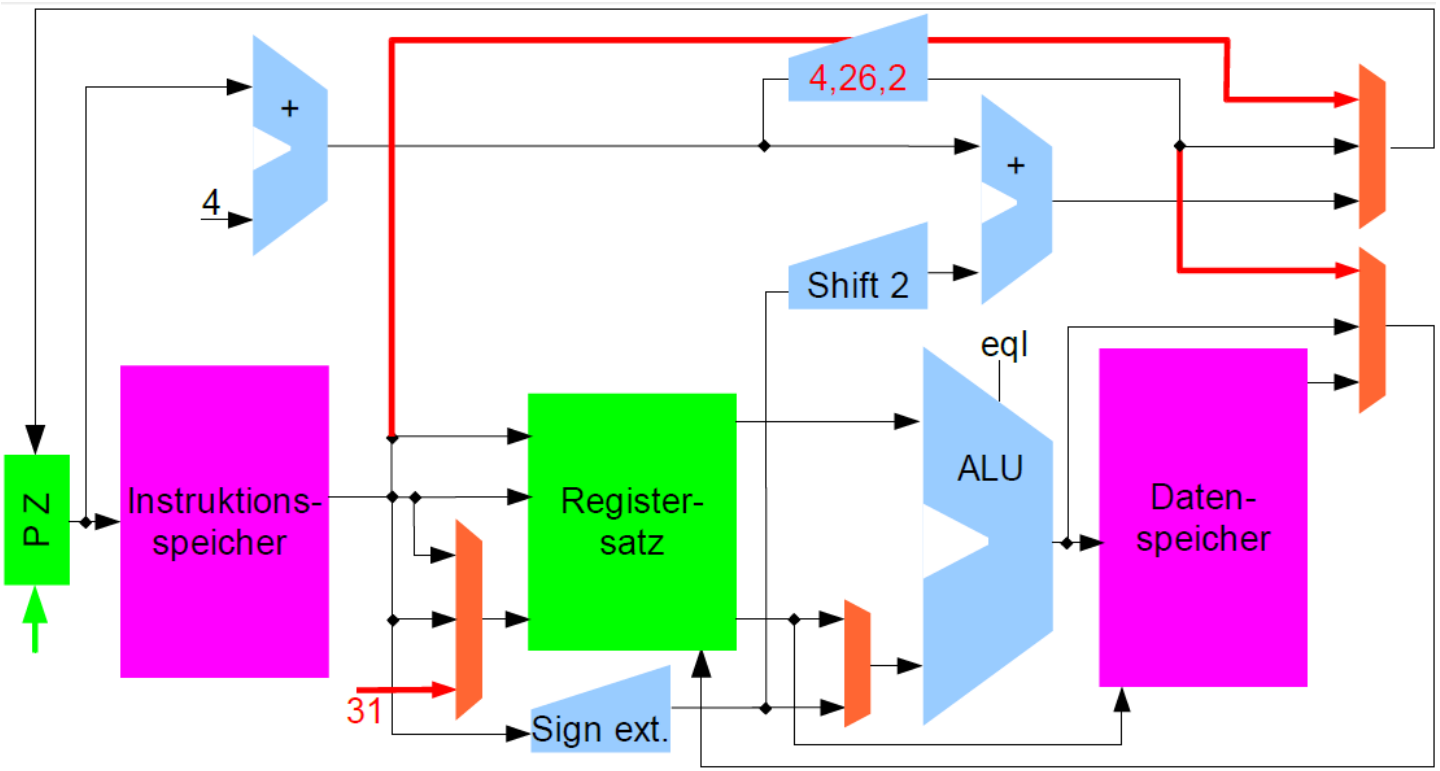
– Schreibsignal an Registersatz, falls nötig



Ergänzung um weitere Sprungbefehle

Pseudorelative Sprünge (jump xLabel)

- Kein separater Addierer erforderlich, nur ein zusätzlicher MUX-Eingang
- Oberste 4 Bits unverändert, untere 28 Bits werden ersetzt (4, 26, 2)
- *Jump-and-Link* (**`jal`**) sichert alten Programmzähler in **$31** (Subroutine)



Erforderliche Steuerleitungen (1)

Für Speicher

– 2-bit Steuersignal: 0/8/16/32 Bit zum Datenspeicher schreiben
– Instruktionsspeicher liest immer

Für Registersatz

– 2-bit Steuersignal: 0/8/16/32 Bit zum Registerfile schreiben

Für 4 Multiplexer

– 2-bit Steuersignal: Auswahl des Zielregisters (1 aus 3)
– 2-bit Steuersignal: Datenquelle für Zielregister
– 2-bit Steuersignal: Sprungziel wählen
– 1-bit Steuersignal: Direkt- oder Registeroperand für ALU wählen

Erforderliche Steuerleitungen (2)

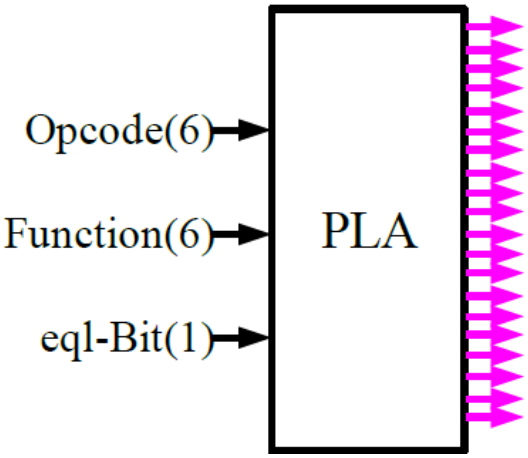
Für Arithmetik

– 1-bit Steuersignal: Vorzeichenerweiterung ja/nein

– 6-bit Steuersignal: ALU-Operation

Ca. 20 Steuersignale sind erforderlich

☞ Mittelgroßes PLA auf Chip



Provisorisches Fazit (1)

Programmzähler PC wird in jedem Taktzyklus neu gesetzt

Taktperiode muss so lange dauern wie die längste Instruktion benötigt

- *fetch*, *decode*, *execute*, *memory access*, *write-back*
- *load/store word*

Flexibler aber reduzierter Befehlssatz

- Keine Fließkomma-Arithmetik
- Keine Unterbrechungen
- Keine Teilwörter

Nur etwa 20 Steuersignale ☞ einfaches Steuerwerk!

Provisorisches Fazit (2)

Mehrheitlich ungetaktete Schaltnetze

– ALU, Speicher, Inkrementer, Vergleicher, Multiplexer, Vorzeichenerweiterung

Aber: Einzyklusmaschine ist unwirtschaftlich

– Komponenten arbeiten jeweils nur einen kleinen Teil der Gesamtzeit
– Zeitverlust bei potentiell kurzen Instruktionen

Empfehlung: überlappende Instruktionsausführung!

Roter Faden

6. Grundlagen der Rechnerarchitektur

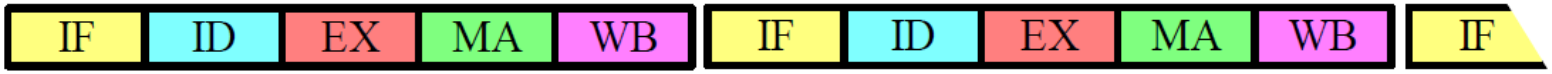
- Grundbegriffe der Rechnerarchitektur
- Programmiermodelle / die Befehlsschnittstelle
- Aufbau einer MIPS-Einzelzyklusmaschine
  - Befehlsholphase *(Instruction Fetch)*
  - Dekodierphase *(Instruction Decode)*
  - Ausführungsphase *(Execute)*
  - Speicherzugriff *(Memory Access)*
  - Register zurückschreiben *(Write Back)*
- Fließbandverarbeitung / *Pipelining*
- Dynamisches *Scheduling*

Überleitung: Mehrzyklen-CPU, *Pipelining* (1)

Gesamtzyklus der bisherigen MIPS

- Dauer des Instruktionszyklus ist die Summe der Einzelverzögerungen
- Unteraktivitäten müssen abwarten, bis die Parameter sicher vorliegen
- Anderenfalls können sich *„spurious writes“* ergeben
- z.B. in Registersatz oder in den Speicher

Mehrzyklen-CPU als Überleitung zum Fließbandprinzip

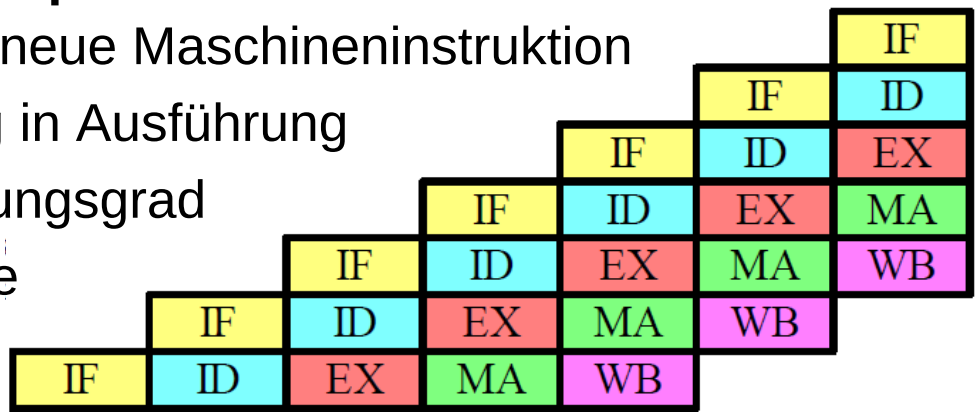


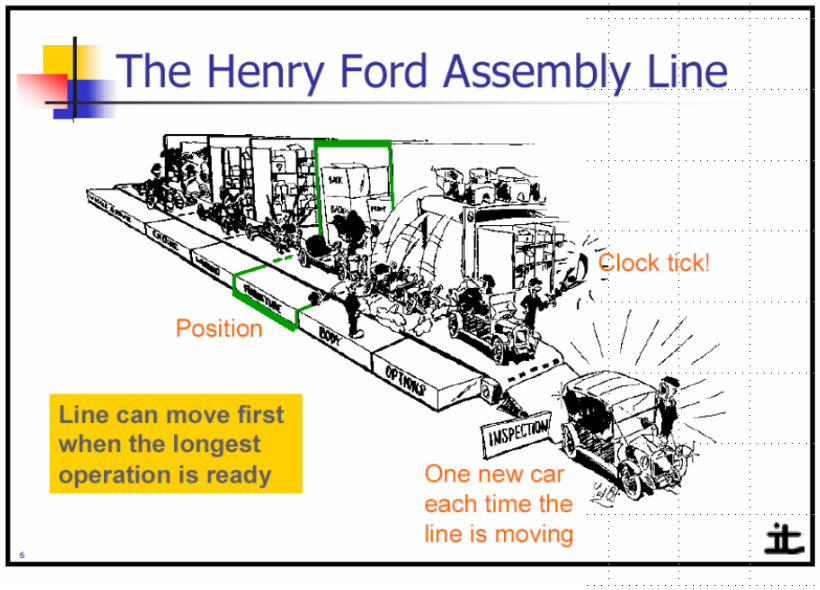
- Aufteilung der Befehlsausführung auf mehrere gleich lange Taktzyklen
- Einfügen von Registern für in den Stufen entstandene Zwischenresultate
- Noch immer nur eine Instruktion zu einem Zeitpunkt in Ausführung
- CPU-Zustand bezieht sich auf eine einzelne aktuelle Instruktion

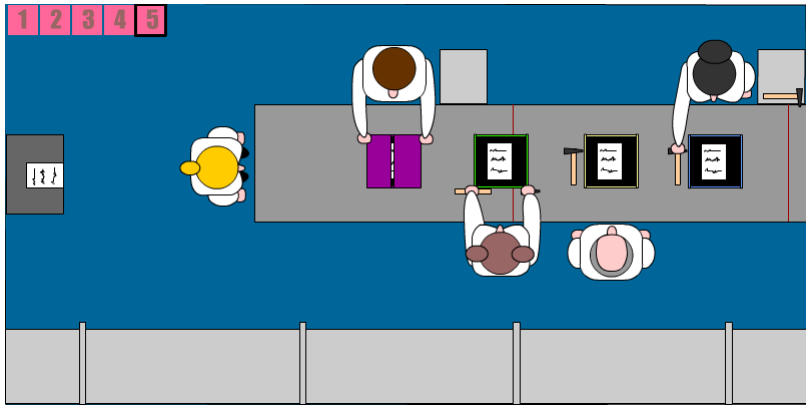
Überleitung: Mehrzyklen-CPU, *Pipelining* (2)

***Pipelined* CPU – mit Fließbandprinzip**

- In jedem Taktzyklus beginnt eine neue Maschineninstruktion
- Mehrere Instruktionen gleichzeitig in Ausführung
- Aber unterschiedlicher Fertigstellungsgrad
- Bessere Auslastung der Hardware
- Höherer Durchsatz





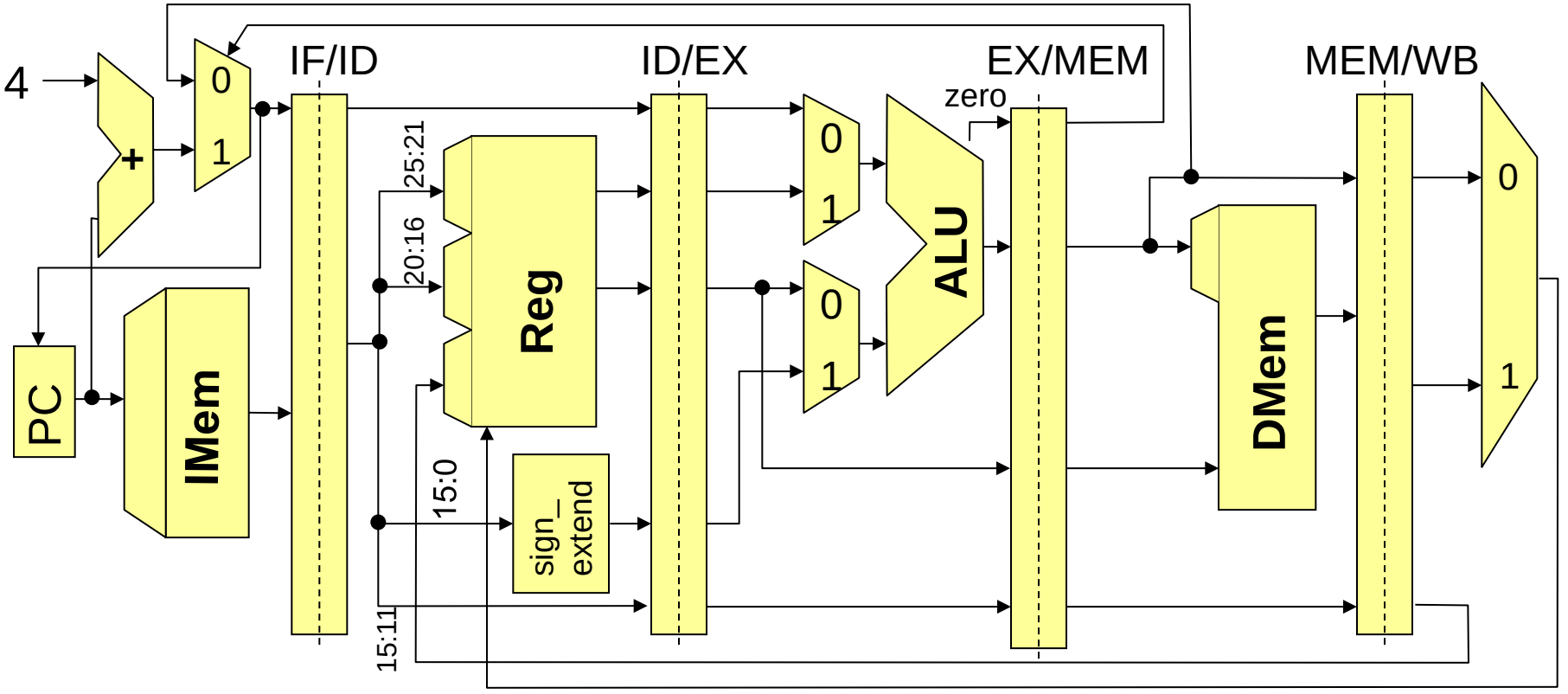


Fließbandverarbeitung

Alternative Darstellung!

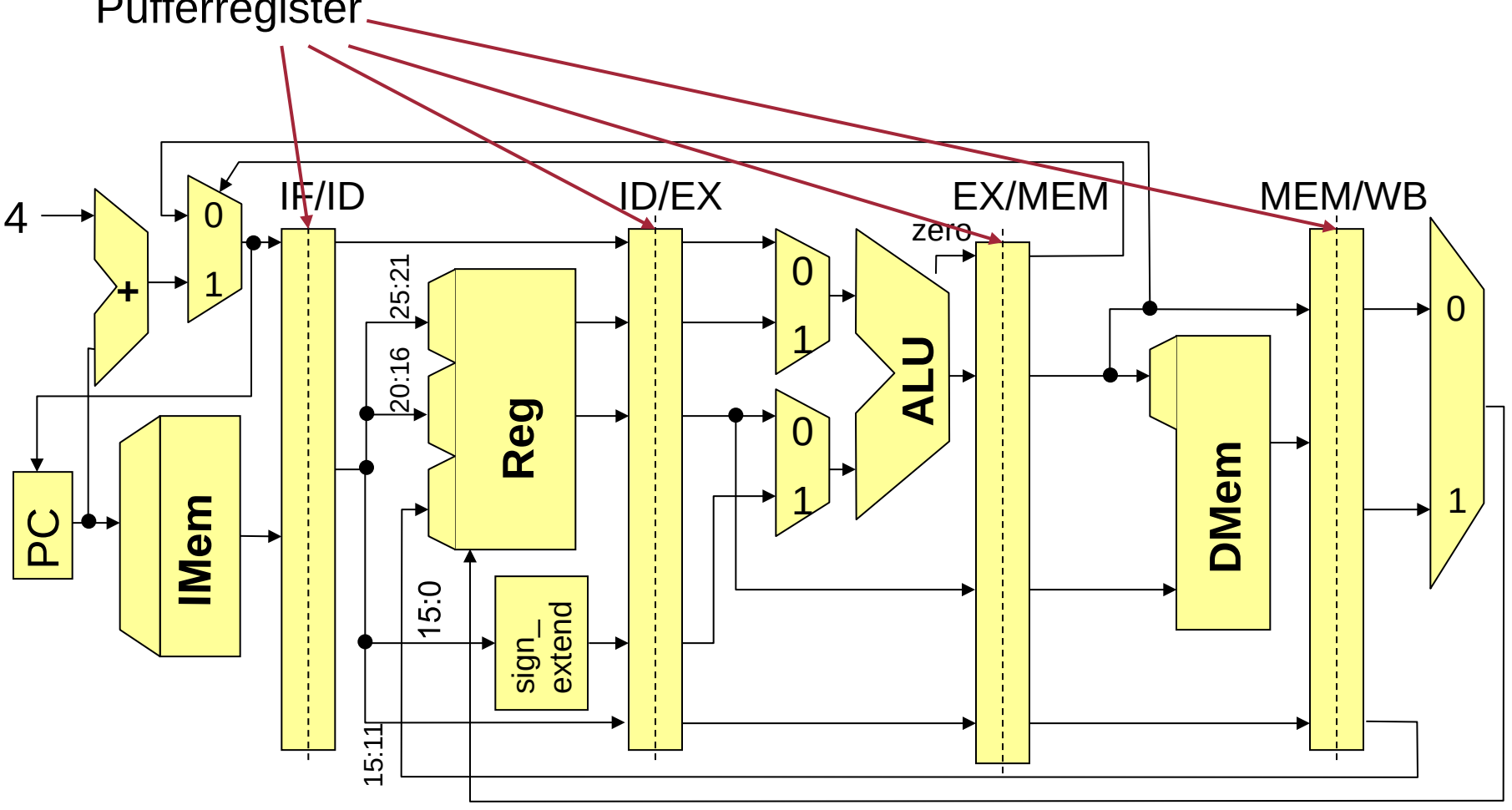
Fließband-Architektur (engl. *pipeline architecture*): Bearbeitung mehrerer Befehle gleichzeitig, analog zu Fertigungsfließbändern. Beispiel MIPS:

instruction fetch | *instruction decode/ register read* | *instruction execution/ address calculation* | *Memory access* | *(register) write-back*



Änderungen gegenüber der Struktur ohne Fließband

– Aufteilung des Rechenwerks in Fließbandstufen, Trennung durch Pufferregister



Aufgaben der einzelnen Phasen bzw. Stufen

Befehlsholphase
– Lesen des aktuellen Befehls; separater Speicher, zur Vermeidung von Konflikten mit Datenzugriffen

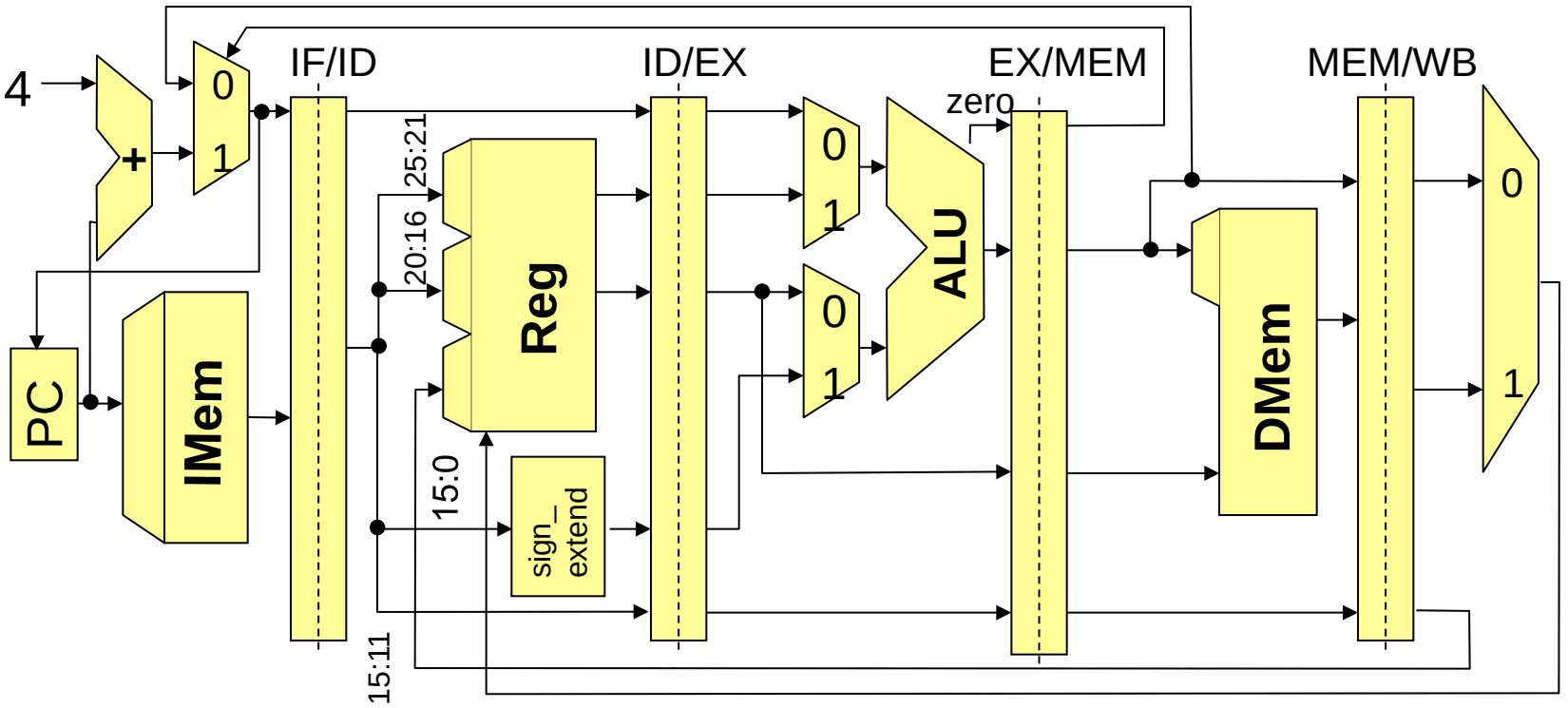
Dekodier- und Register-Lese-Phase
– Lesen der Register möglich wegen fester Plätze für Nr. im Befehlswort

Ausführungs- und Adressberechnungsphase
– Berechnung arithmetischer Funktion bzw. Adresse für Speicherzugriff

Speicherzugriffsphase
– Wird nur bei Lade- und Speicherbefehlen benötigt

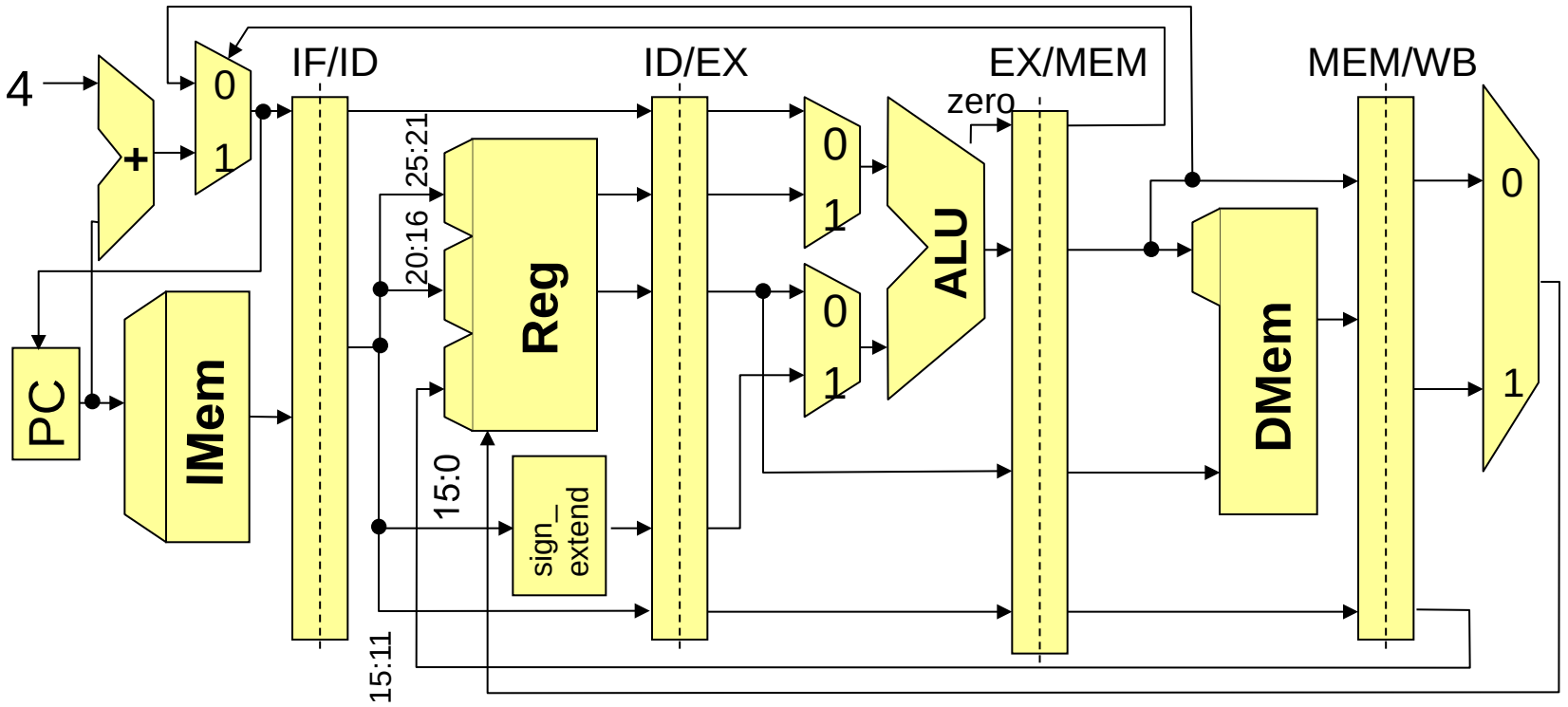
Abspeicherungsphase
– Speichern in Register, bei Speicherbefehlen nicht benötigt

Idealer Fließbanddurchlauf (1)



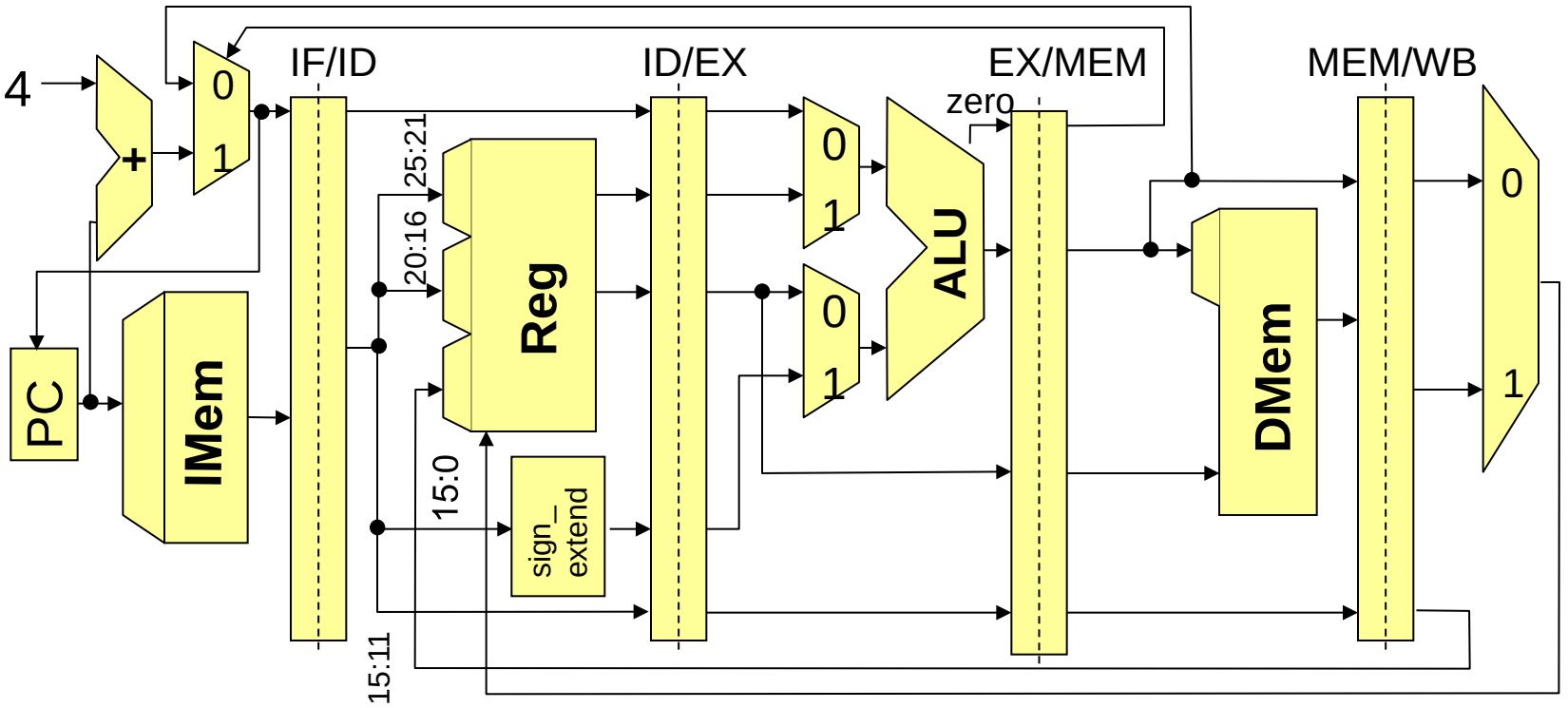
Zyklus 1				
Befehl 1				

Idealer Fließbanddurchlauf (2)



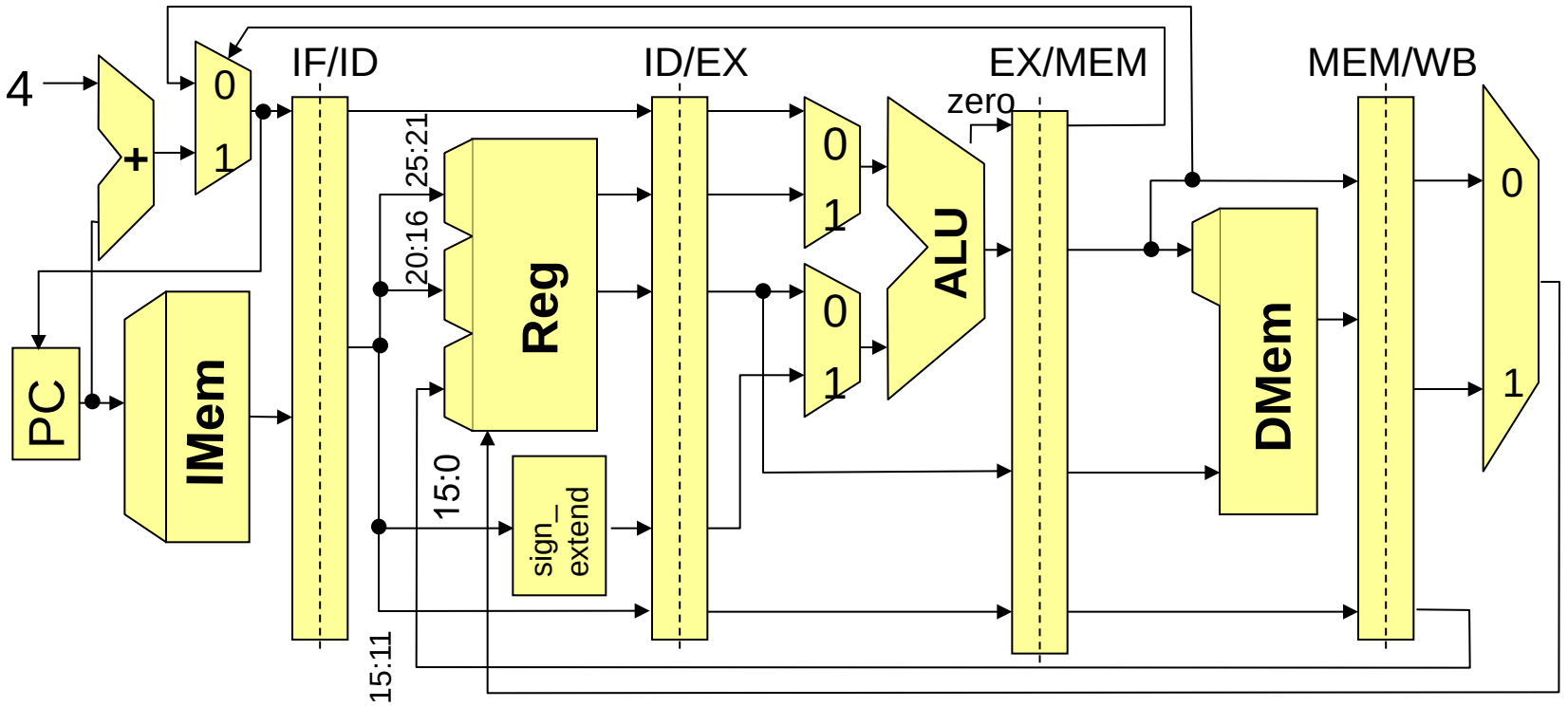
Zyklus 2				
Befehl 2	**Befehl 1**			

Idealer Fließbanddurchlauf (3)



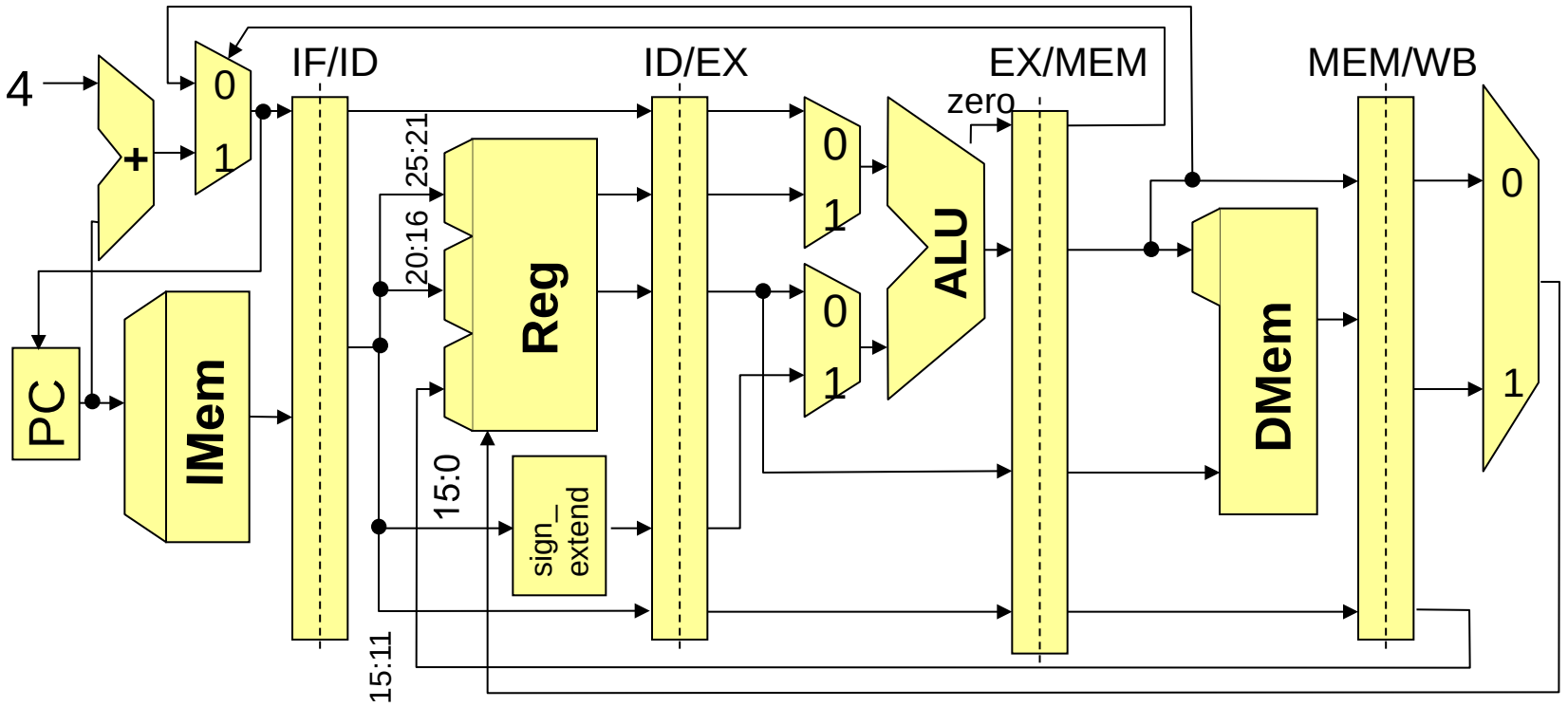
Zyklus 3				
Befehl 3	**Befehl 2**	**Befehl 1**		

Idealer Fließbanddurchlauf (4)



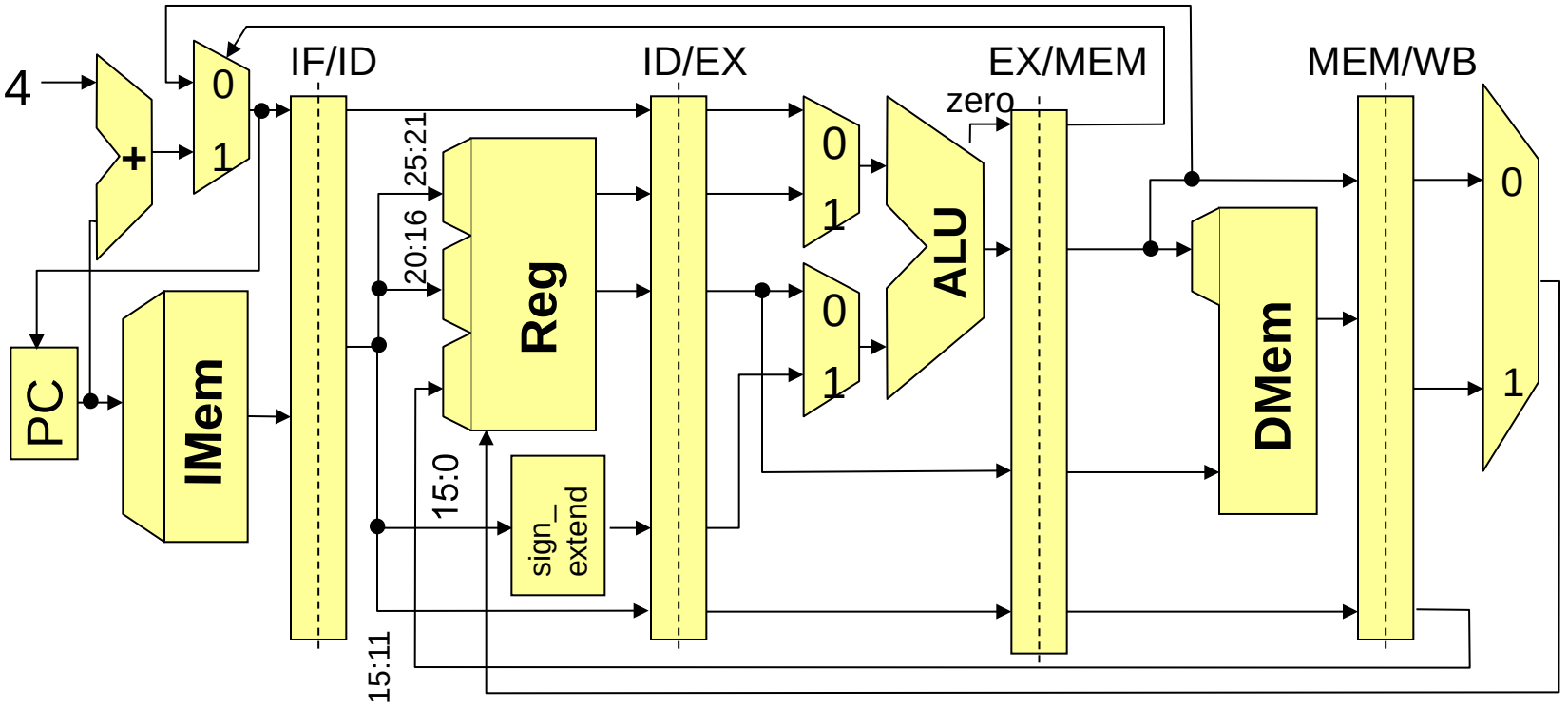
Zyklus 4				
Befehl 4	**Befehl 3**	**Befehl 2**	**Befehl 1**	

Idealer Fließbanddurchlauf (5)



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

Idealer Fließbanddurchlauf (6)



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 Maschinenbefehlen.

Definition: 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  $10,$12,$9
xor $8,$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 Maschinenbefehlen.

Definition: 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 $8,$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 Maschinenbefehlen.

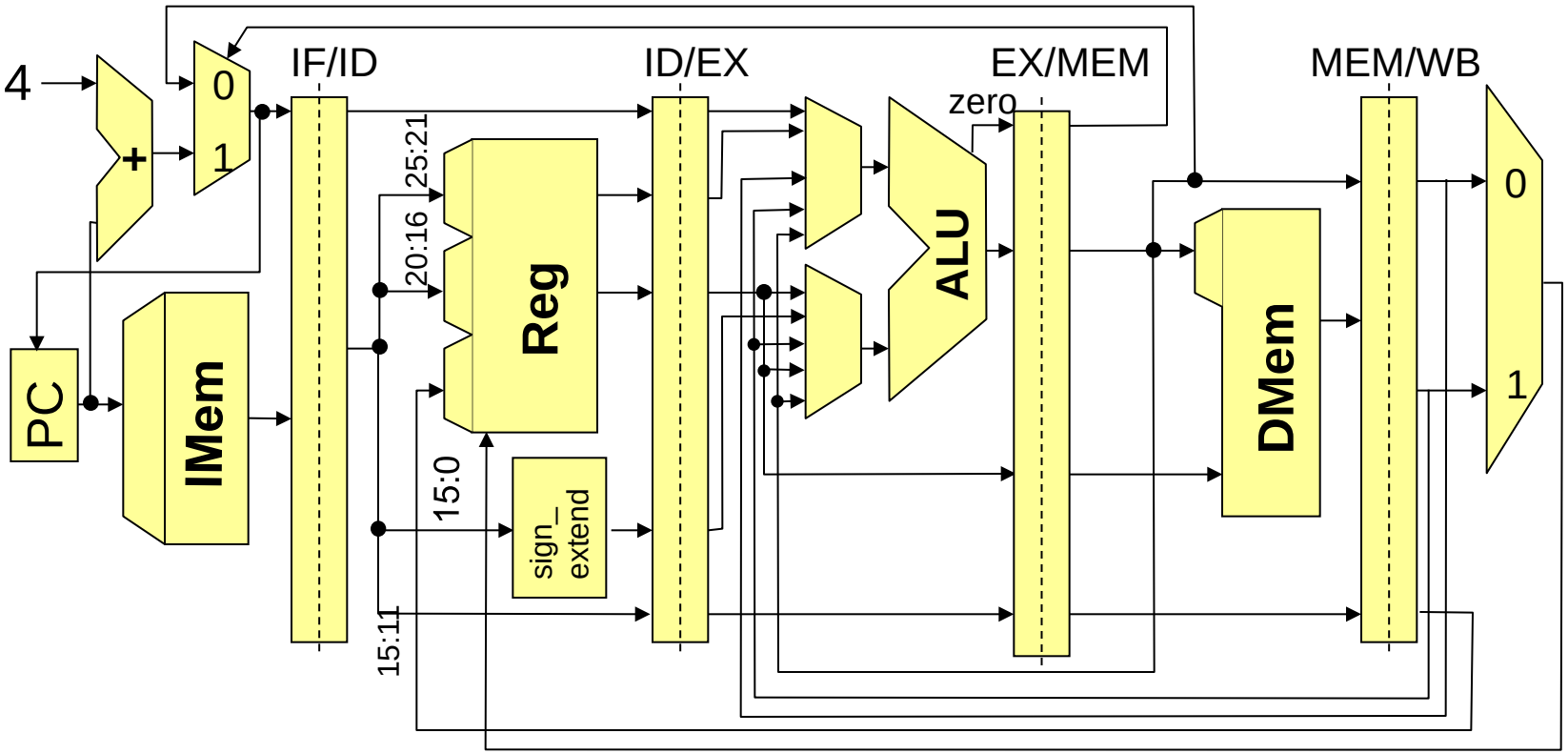
Definition: Zwei Befehle *i* und *j* heißen voneinander *Ausgabe-abhängig*, falls *i* und *j* die selbe Speicherzelle beschreiben.

Beispiel

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

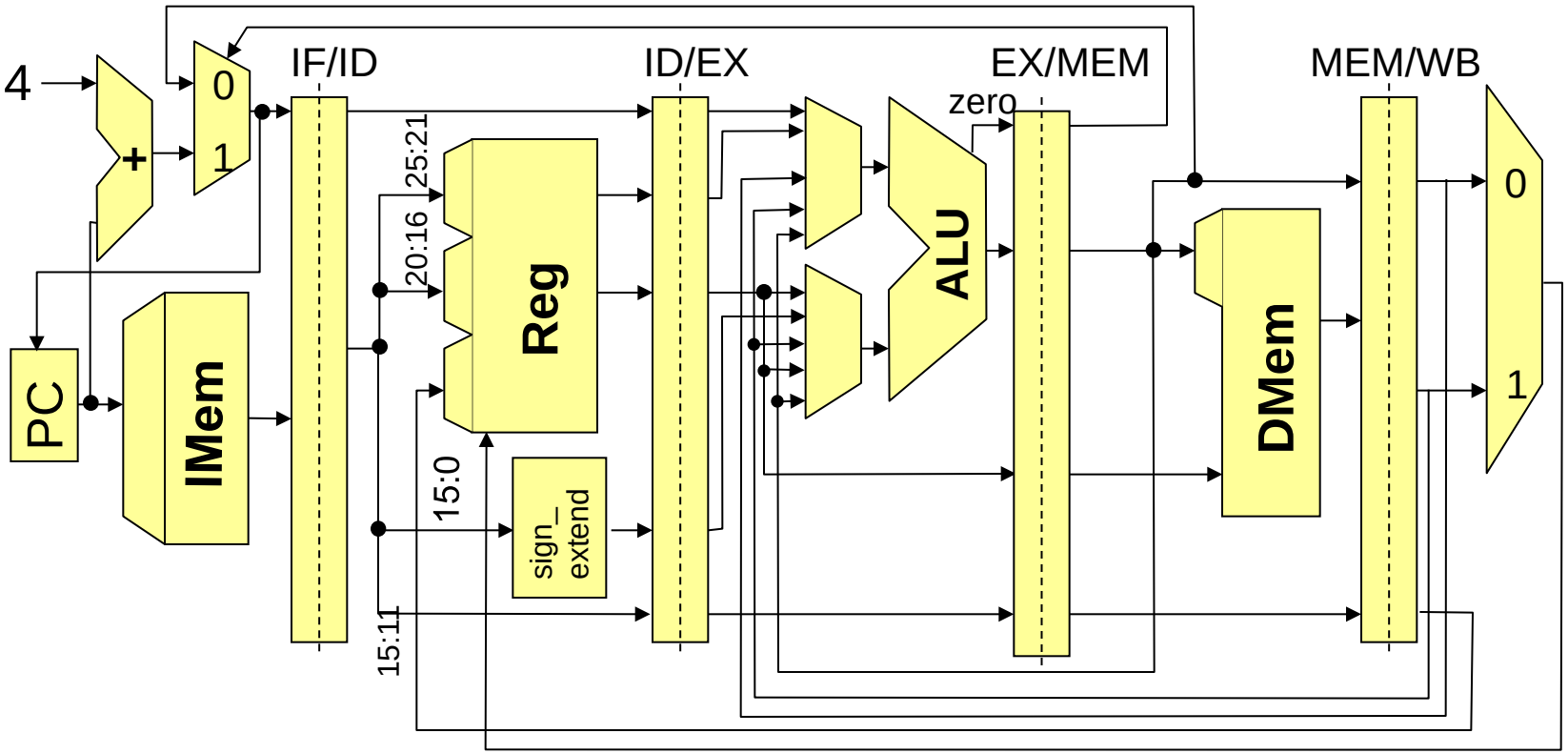
Diese Art der Abhängigkeit heißt (bei Hennessy und anderen) *write after write*- (oder WAW-) Abhängigkeit

Bypässe, *forwarding* – Behandlung des *data hazards* bei add und sub (1)



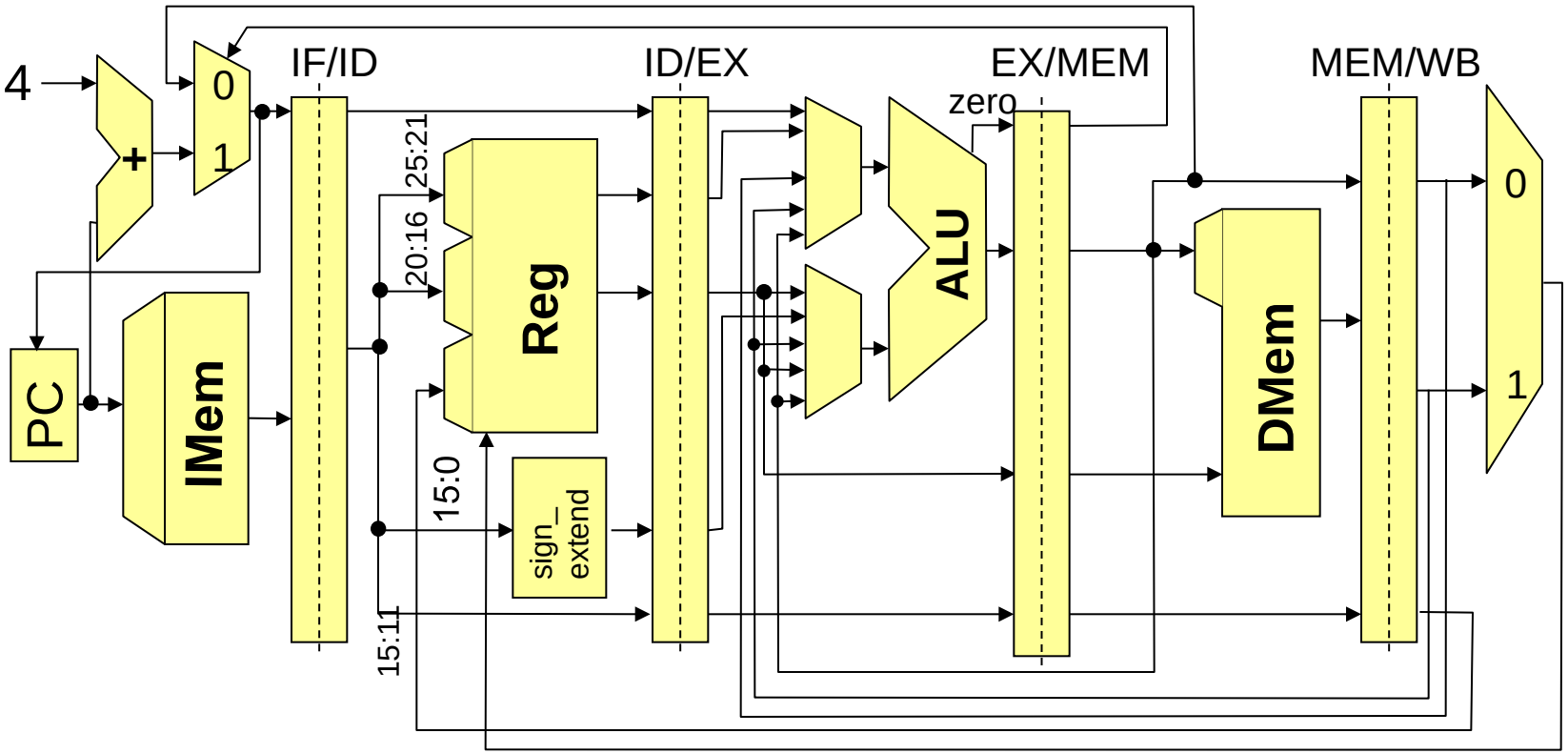
Zyklus 1				
`add $12,$2,$3`				

Bypässe, *forwarding* – Behandlung des *data hazards* bei add und sub (2)



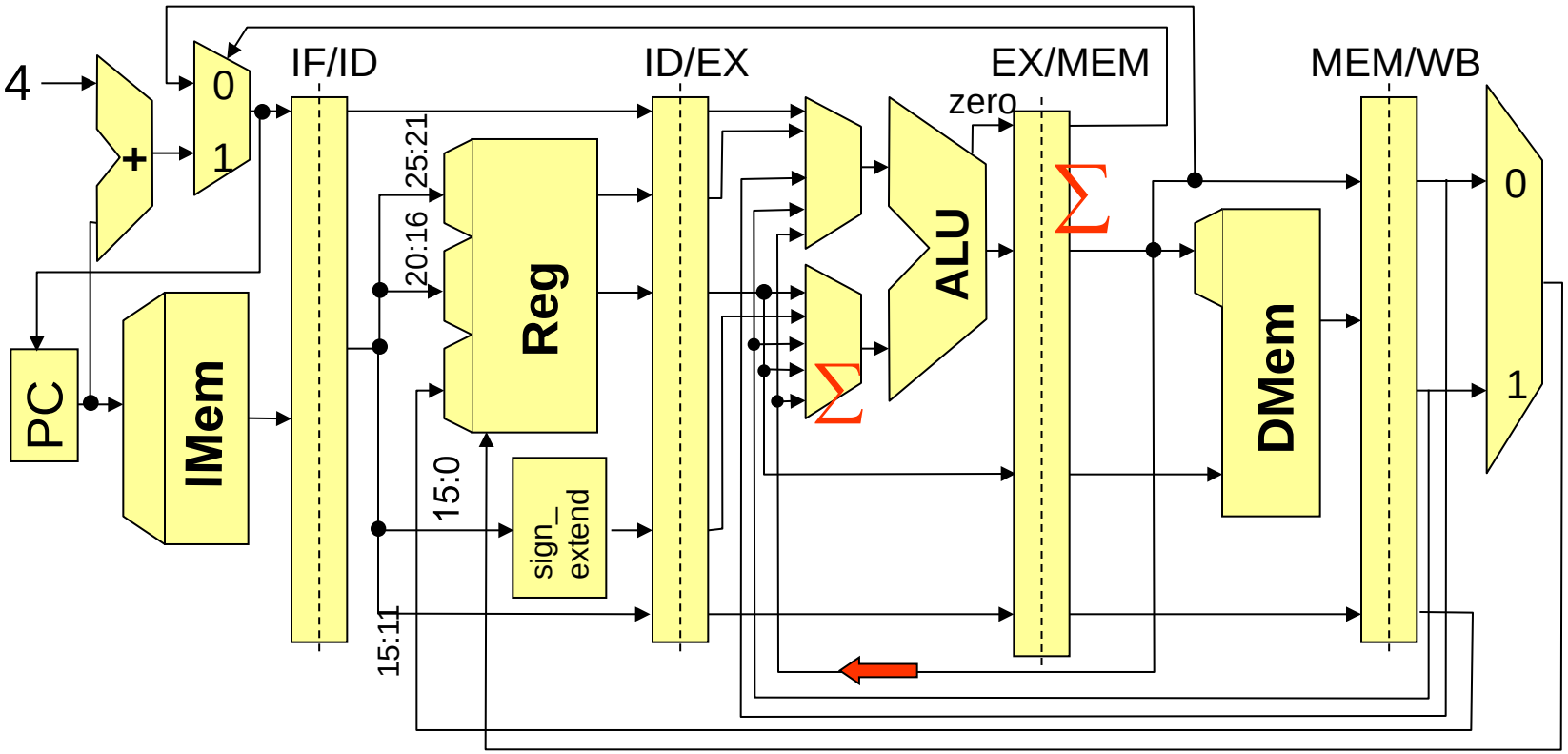
Zyklus 2				
`sub $4,$5,$12`	`add $12,$2,$3`			

Bypässe, *forwarding* – Behandlung des *data hazards* bei add und sub (3)



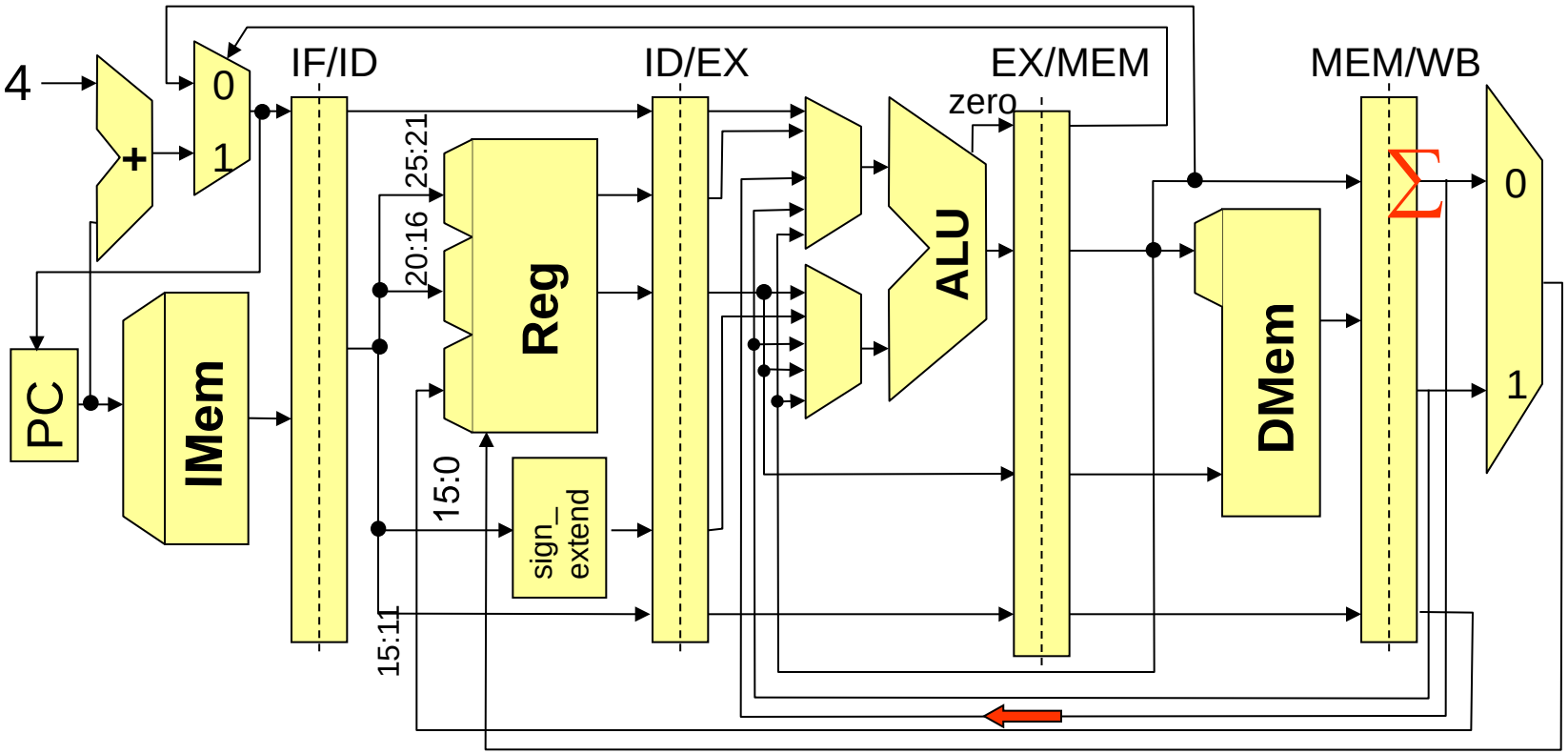
Zyklus 3				
`and $6,$12,$7`	`sub $4,$5,$12`	`add $12,$2,$3`		

Bypässe, *forwarding* – Behandlung des *data hazards* bei add und sub (4)



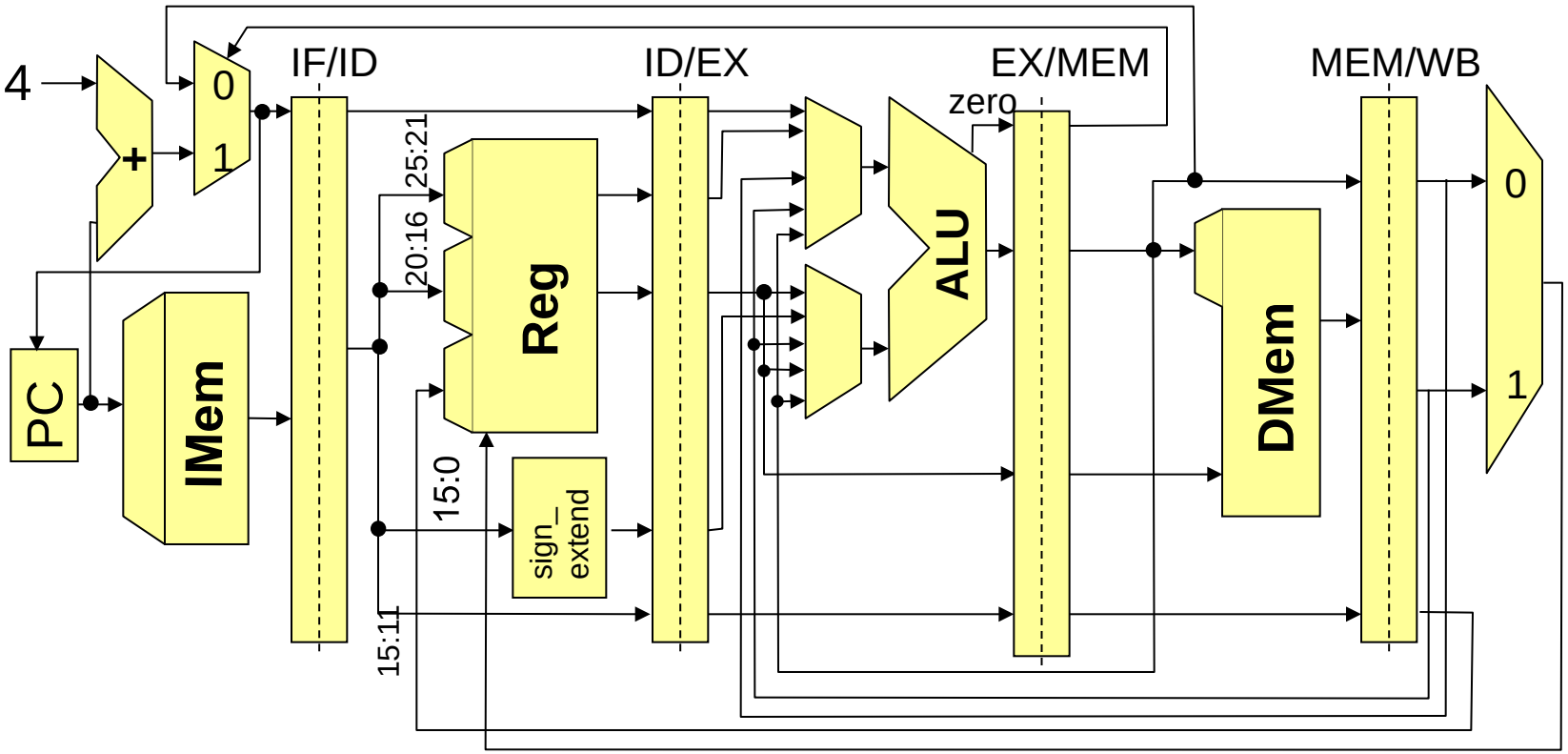
Zyklus 4				
`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`add $12,$2,$3`	

Bypässe, *forwarding* – Behandlung des *data hazards* bei add und sub (5)



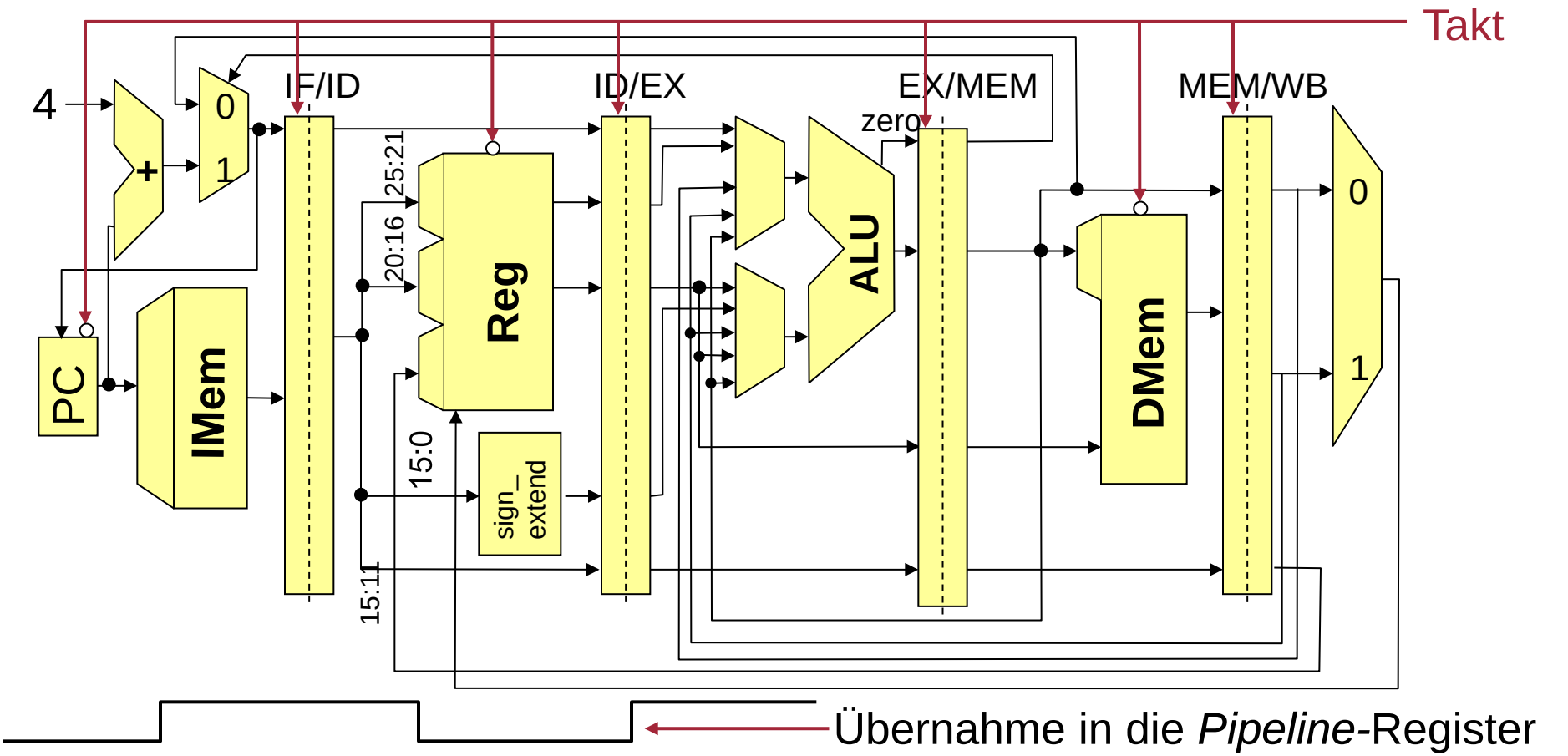
Zyklus 5				
`xor $8,$12,$11`	`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`add $12,$2,$3`

Bypässe, *forwarding* – Behandlung des *data hazards* bei add und sub (6)



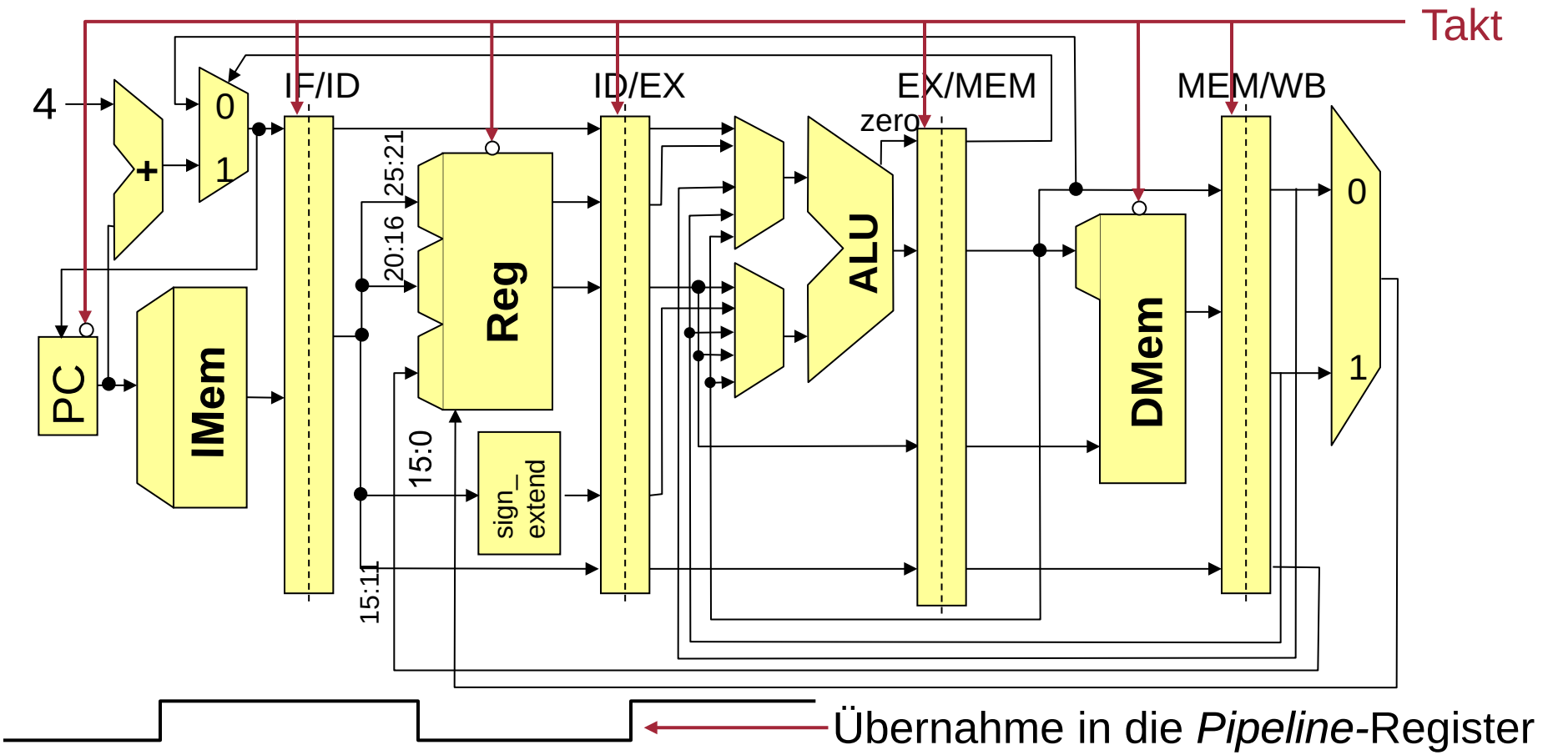
Zyklus 6				
`...`	`xor $8,$12,$11`	`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`

Taktung zur Behandlung des *data hazards* bei or (1)



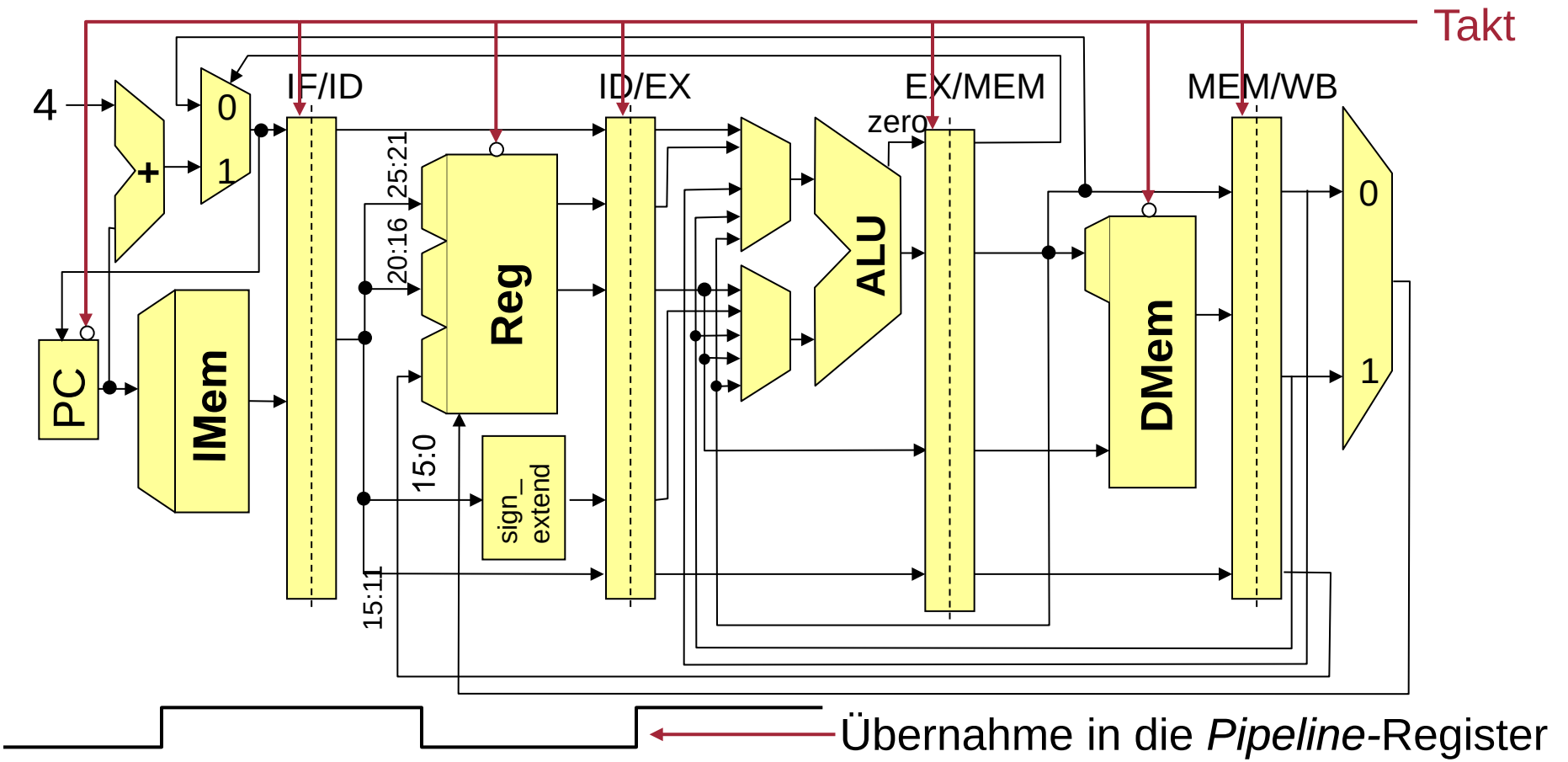
Zyklus 1				
`add $12,$2,$3`				

Taktung zur Behandlung des *data hazards* bei or (2)



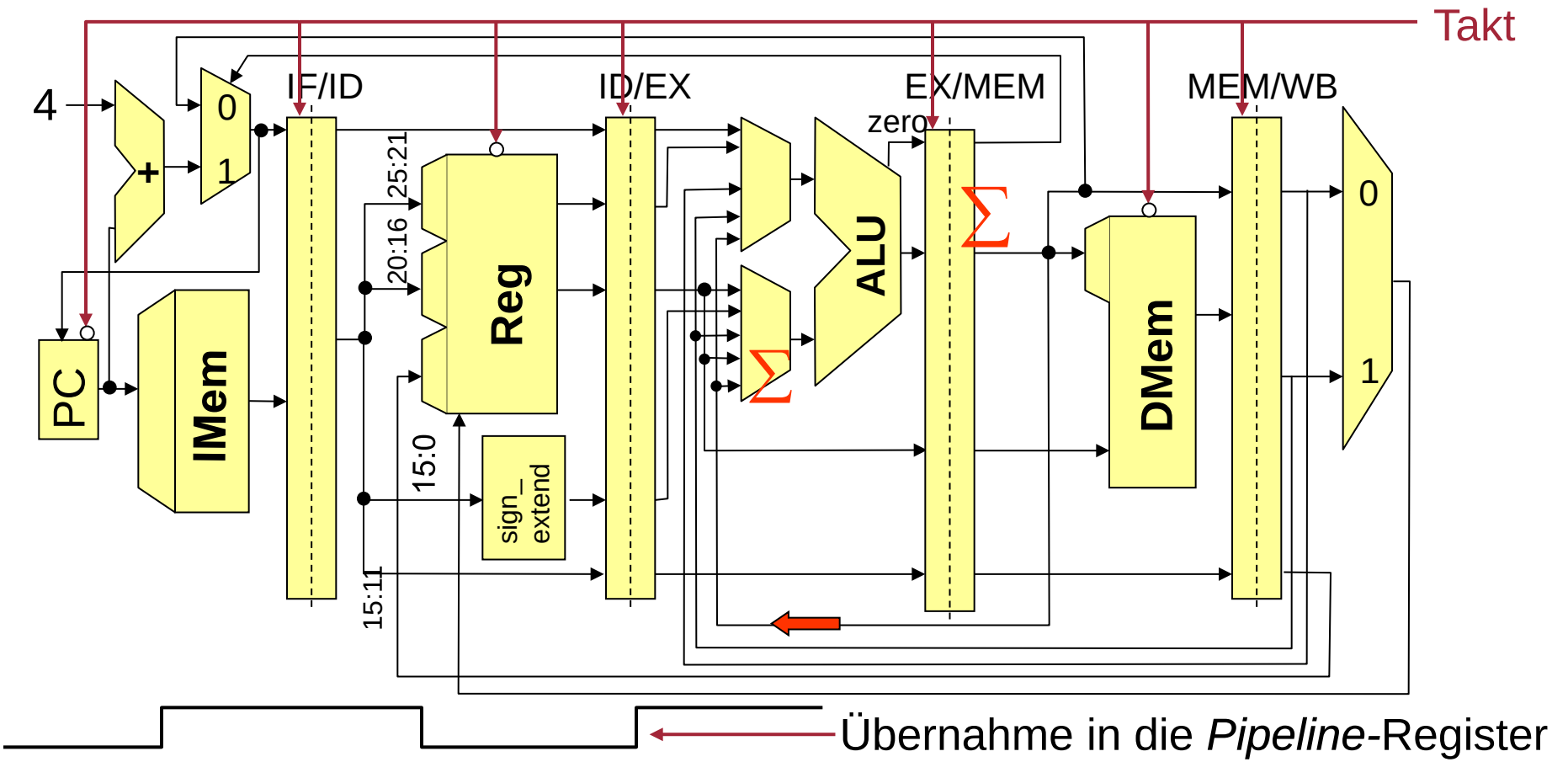
Zyklus 2				
`sub $4,$5,$12`	`add $12,$2,$3`			

Taktung zur Behandlung des *data hazards* bei or (3)



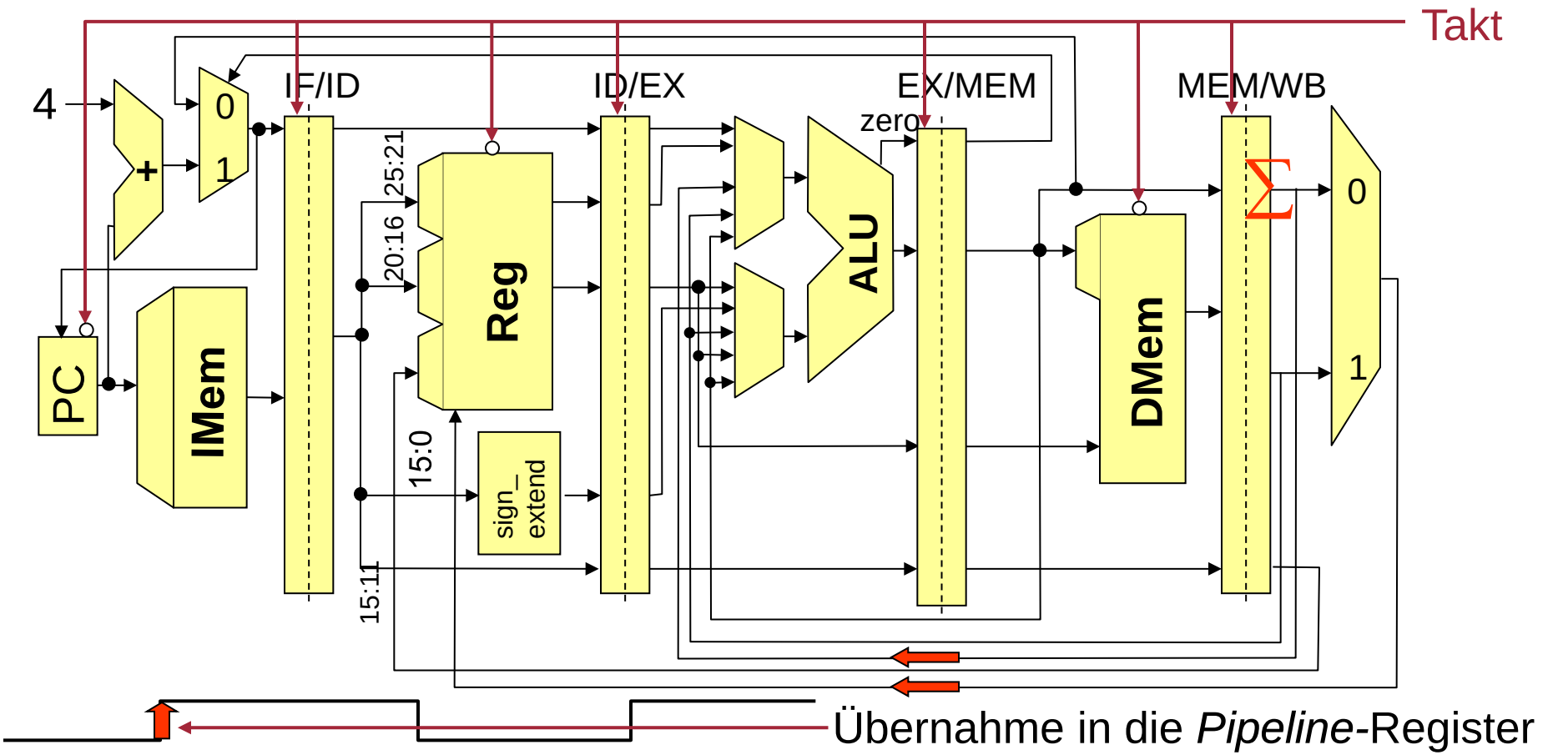
Zyklus 3				
`and $6,$12,$7`	`sub $4,$5,$12`	`add $12,$2,$3`		

Taktung zur Behandlung des *data hazards* bei or (4)



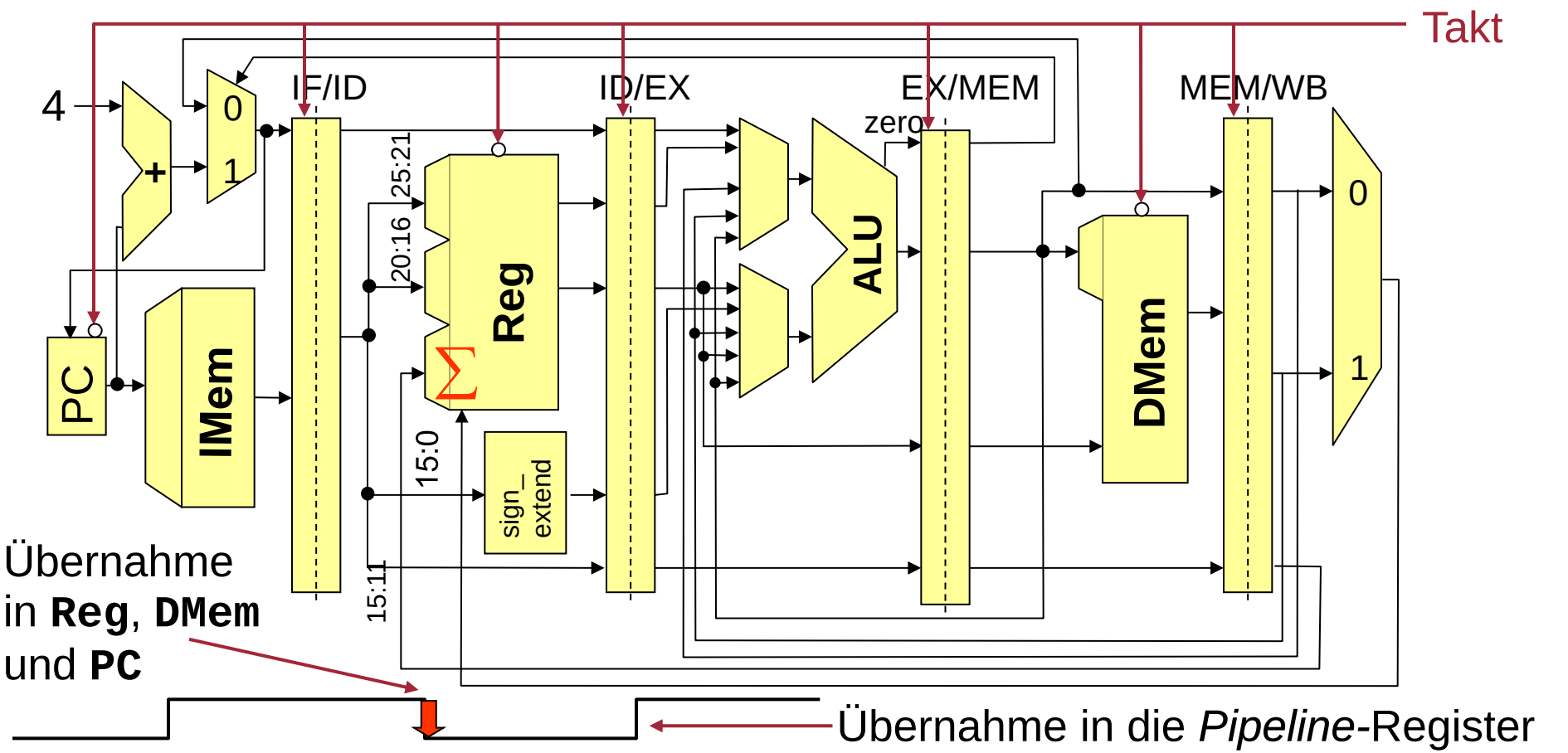
Zyklus 4				
`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`add $12,$2,$3`	

Taktung zur Behandlung des *data hazards* bei or (5)



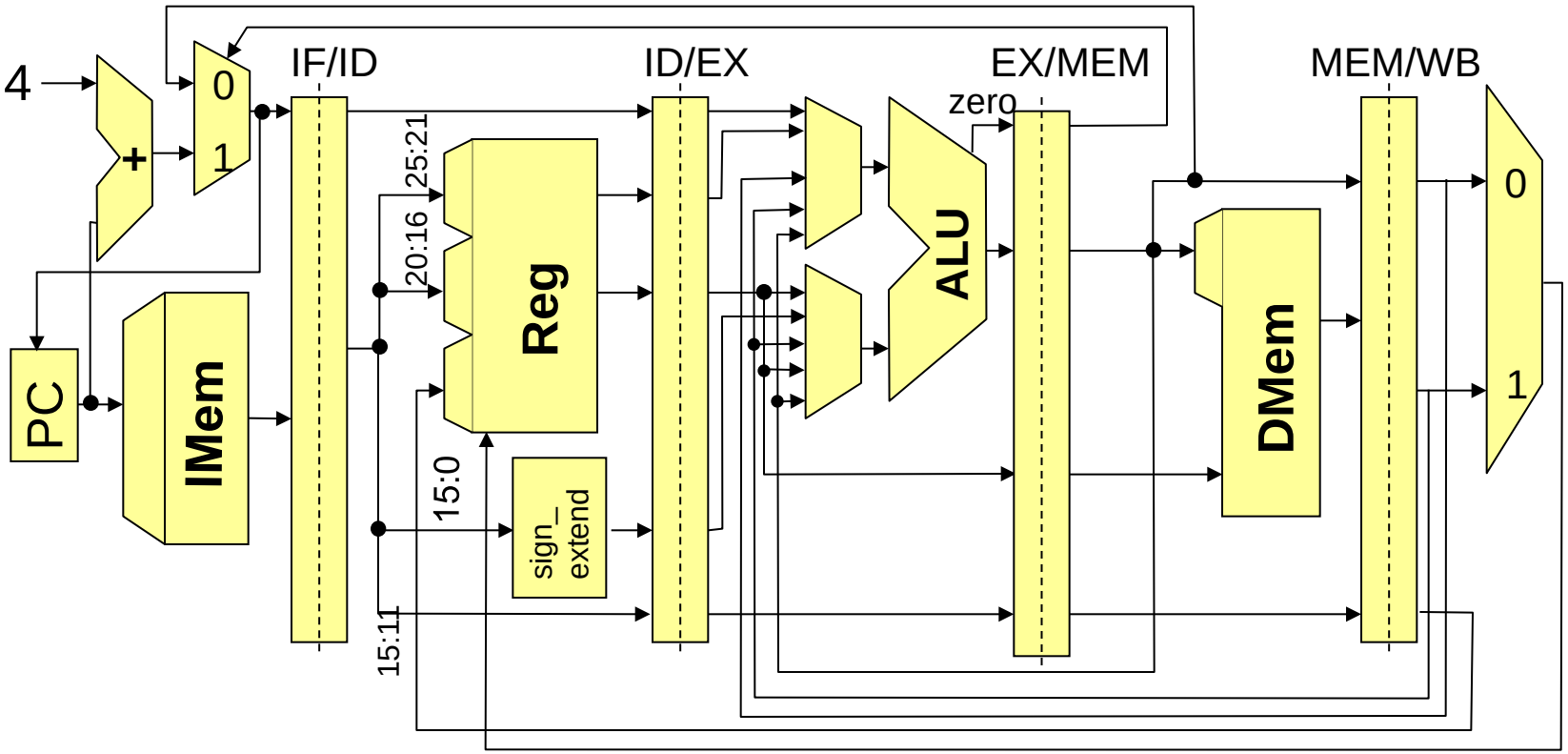
Zyklus 5				
`xor $8,$12,$11`	`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`add $12,$2,$3`

Taktung zur Behandlung des *data hazards* bei or (6)



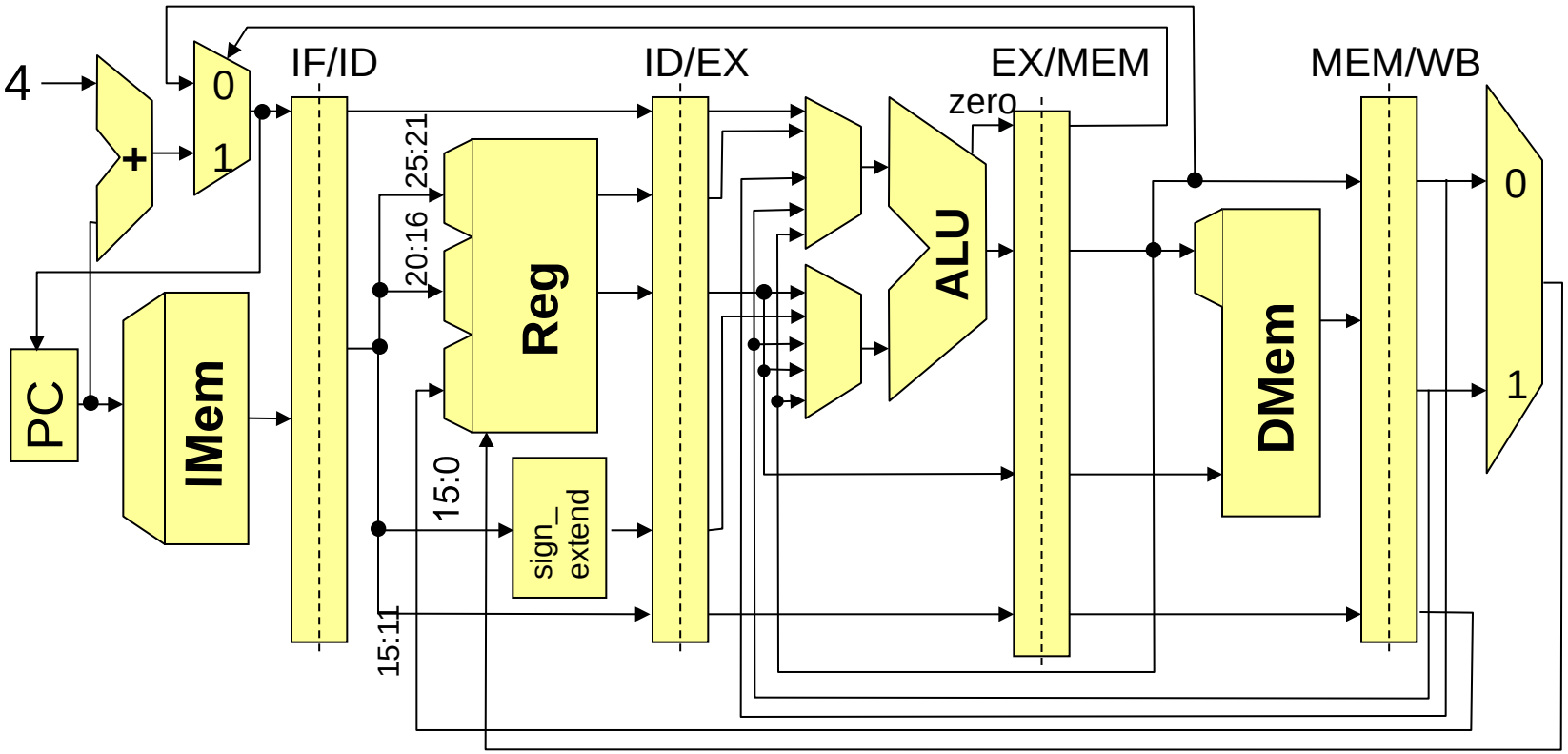
Zyklus 6				
`...`	`xor $8,$12,$11`	`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`

Alle *data hazards* durch Bypässe behandelbar? (1)



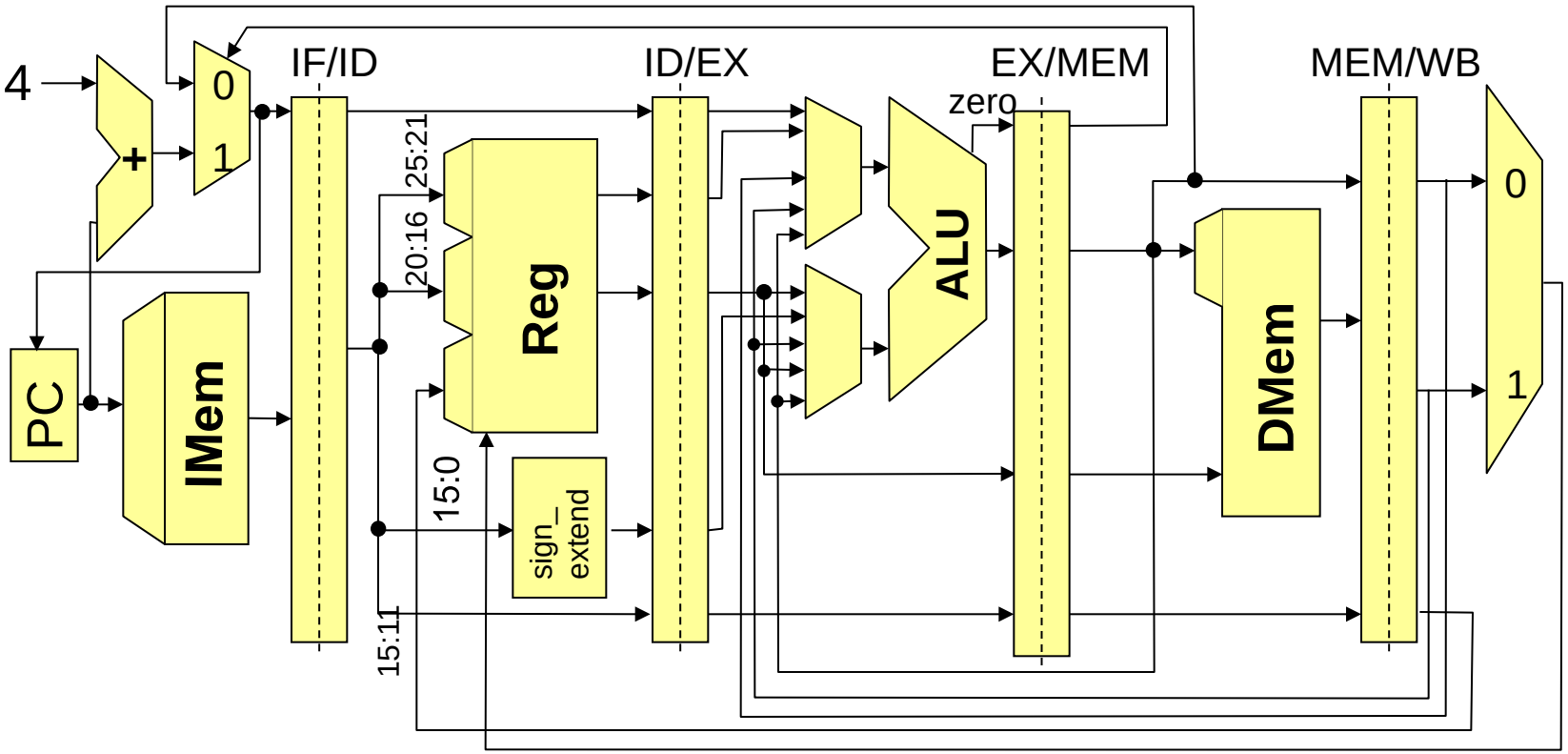
Zyklus 1				
`lw $12,0($2)`				

Alle *data hazards* durch Bypässe behandelbar? (2)



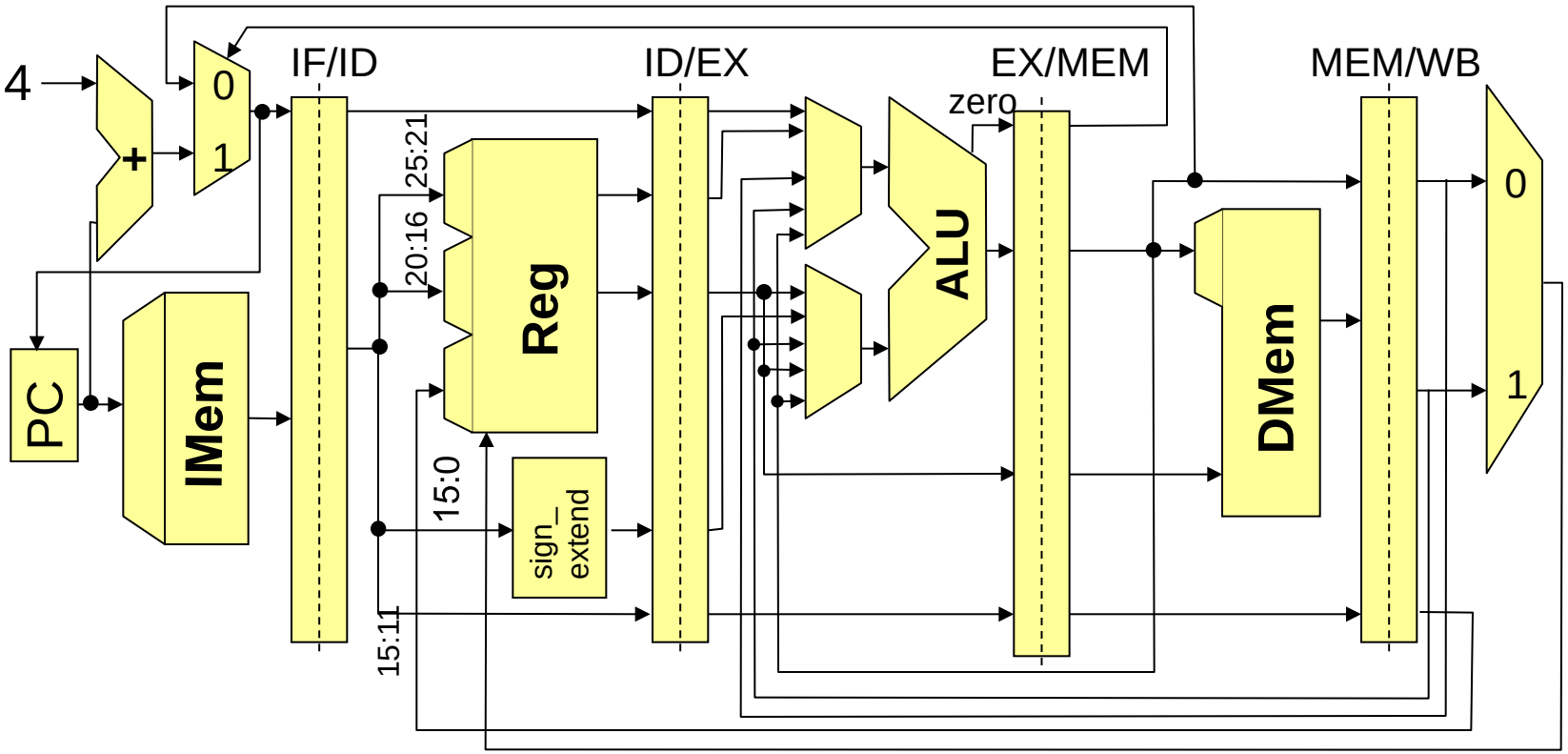
Zyklus 2				
`sub $4,$5,$12`	`lw $12,0($2)`			

Alle *data hazards* durch Bypässe behandelbar? (3)

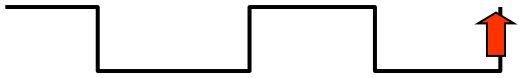


Zyklus 3				
`and $6,$12,$7`	`sub $4,$5,$12`	`lw $12,0($2)`		

Alle *data hazards* durch Bypässe behandelbar? (4)

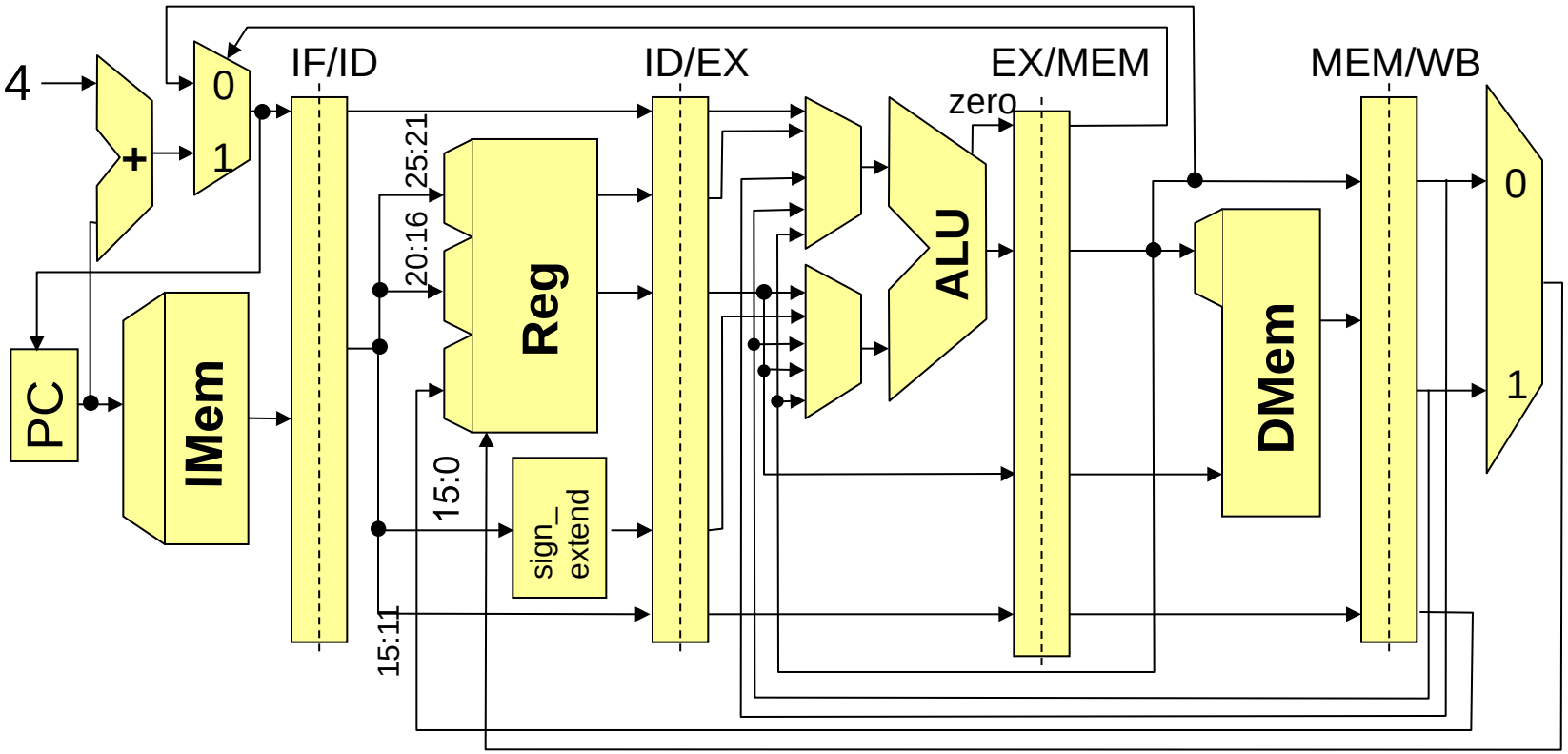


Zyklus 4				
`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`lw $12,0($2)`	



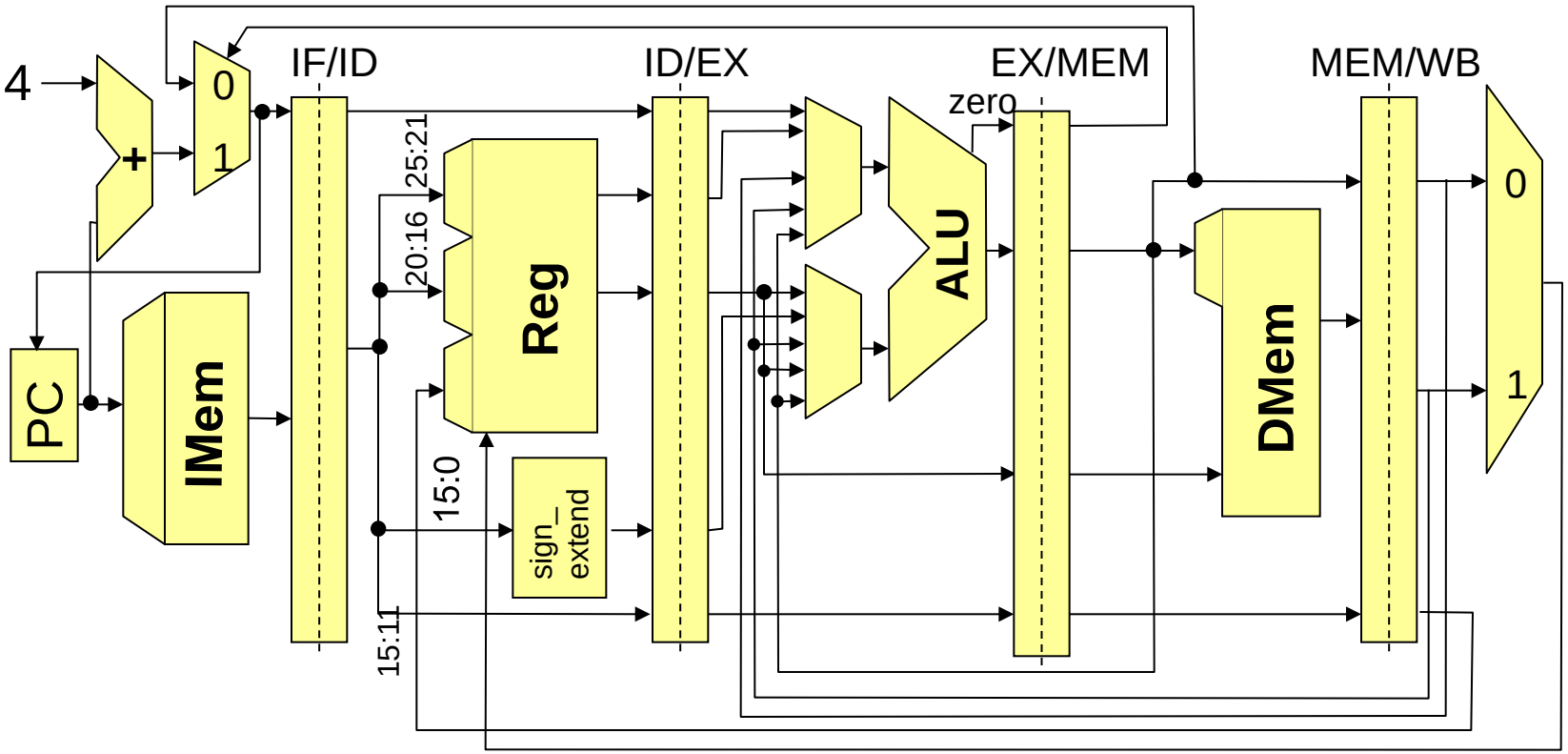
*Geladenes Speicherwort steht am Ende von Zyklus 4 in Pipeline-Register, steht für **sub** noch nicht bereit!*

Lösung durch Anhalten des Fließbands *(pipeline stall, hardware interlocking, bubbles)* (1)



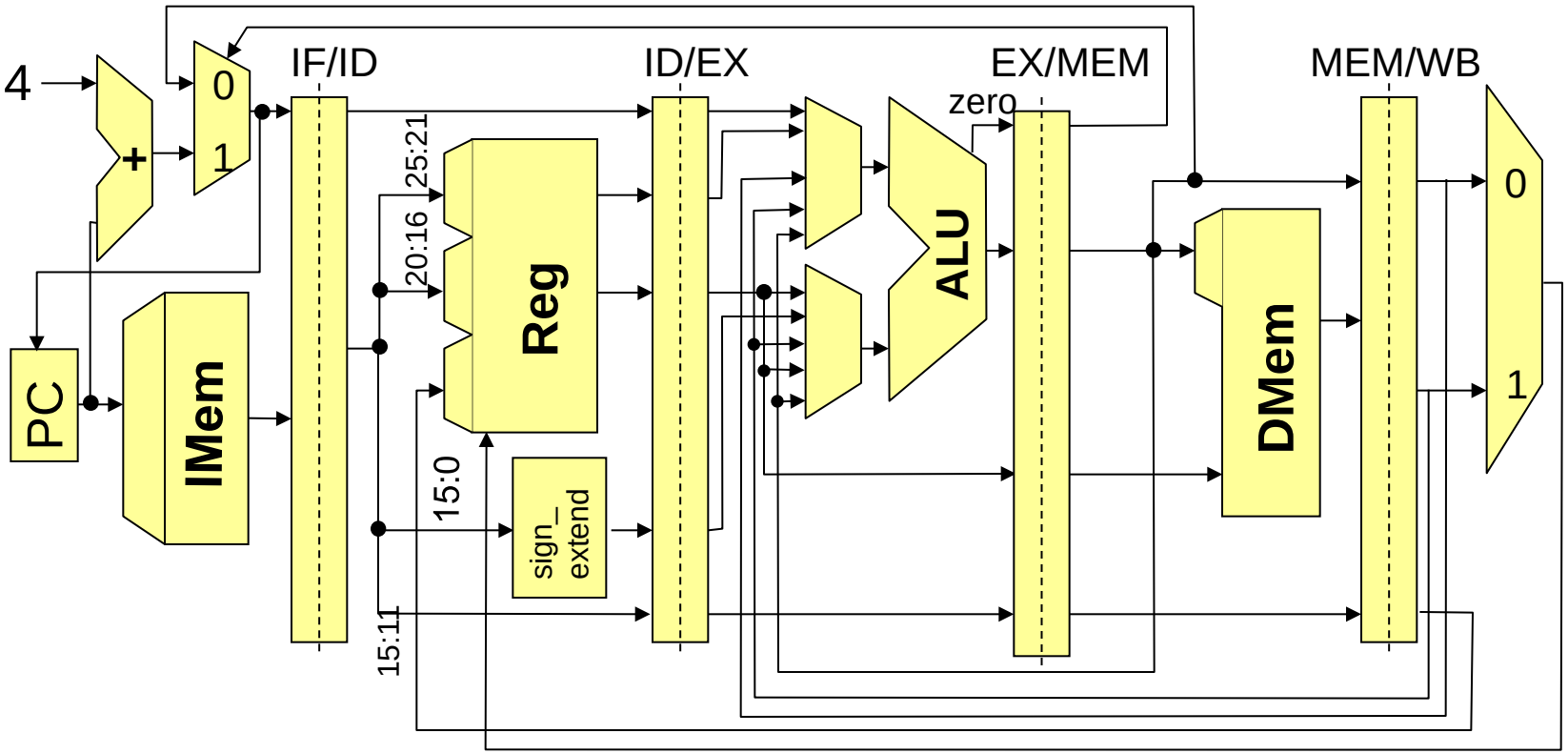
Zyklus 1				
`lw $12,0($2)`				

Lösung durch Anhalten des Fließbands *(pipeline stall, hardware interlocking, bubbles)* (2)



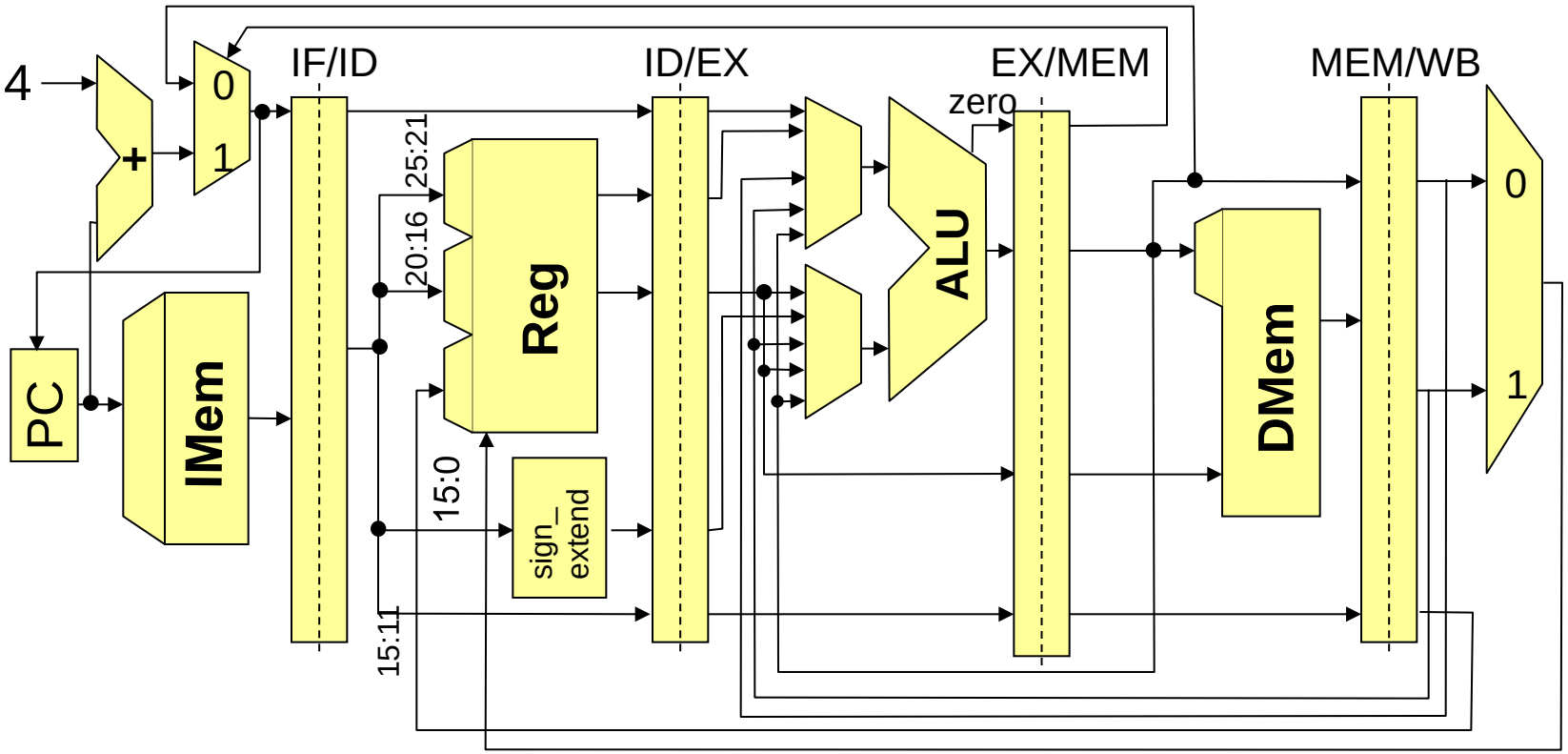
Zyklus 2				
`sub $4,$5,$12`	`lw $12,0($2)`			

Lösung durch Anhalten des Fließbands *(pipeline stall, hardware interlocking, bubbles)* (3)



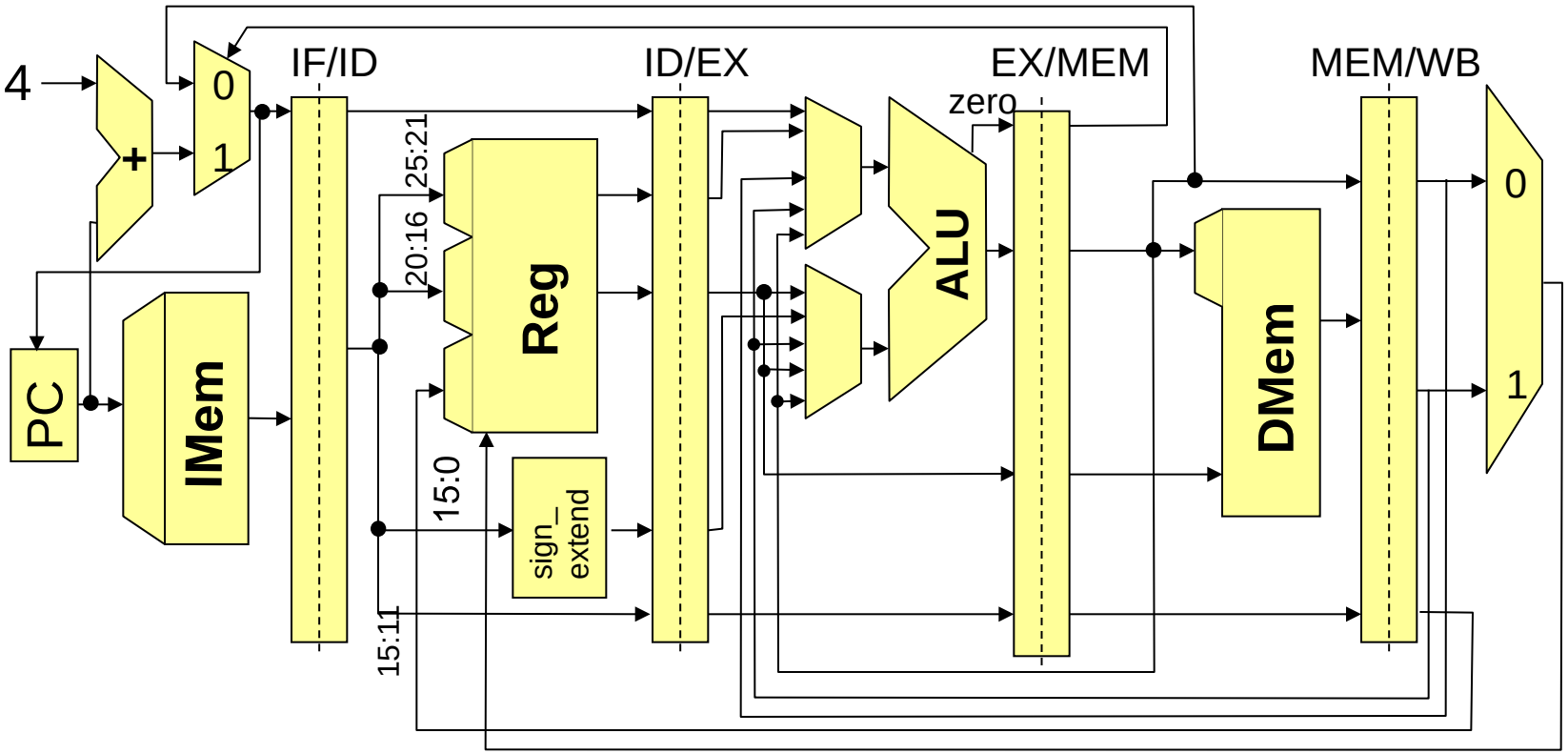
Zyklus 3				
`and $6,$12,$7`	`sub $4,$5,$12`	`lw $12,0($2)`		

Lösung durch Anhalten des Fließbands *(pipeline stall, hardware interlocking, bubbles)* (4)



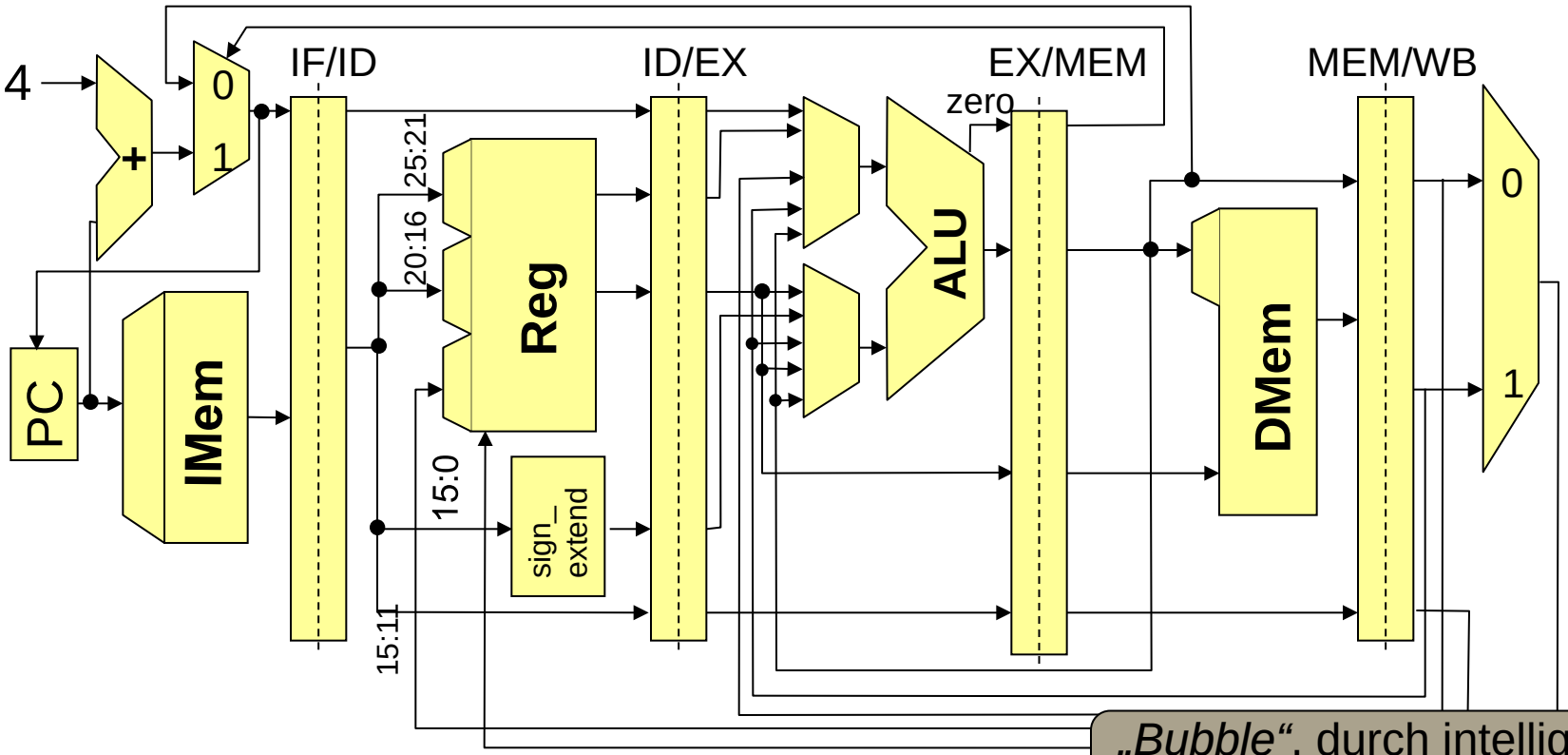
Zyklus 4			
or $10,$12,$9 NOOP	and $6,$12,$7 NOOP	sub $4,$5,$12 NOOP	lw $12,0($2)	

Lösung durch Anhalten des Fließbands *(pipeline stall, hardware interlocking, bubbles)* (5)



Zyklus 5				
`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`nop`	`lw $12,0($2)`

Lösung durch Anhalten des Fließbands *(pipeline stall, hardware interlocking, bubbles)* (6)



„*Bubble*“, durch intelligente Compiler vermeiden!

Zyklus 6				
`xor $8,$12,$11`	`or $10,$12,$9`	`and $6,$12,$7`	`sub $4,$5,$12`	`nop`

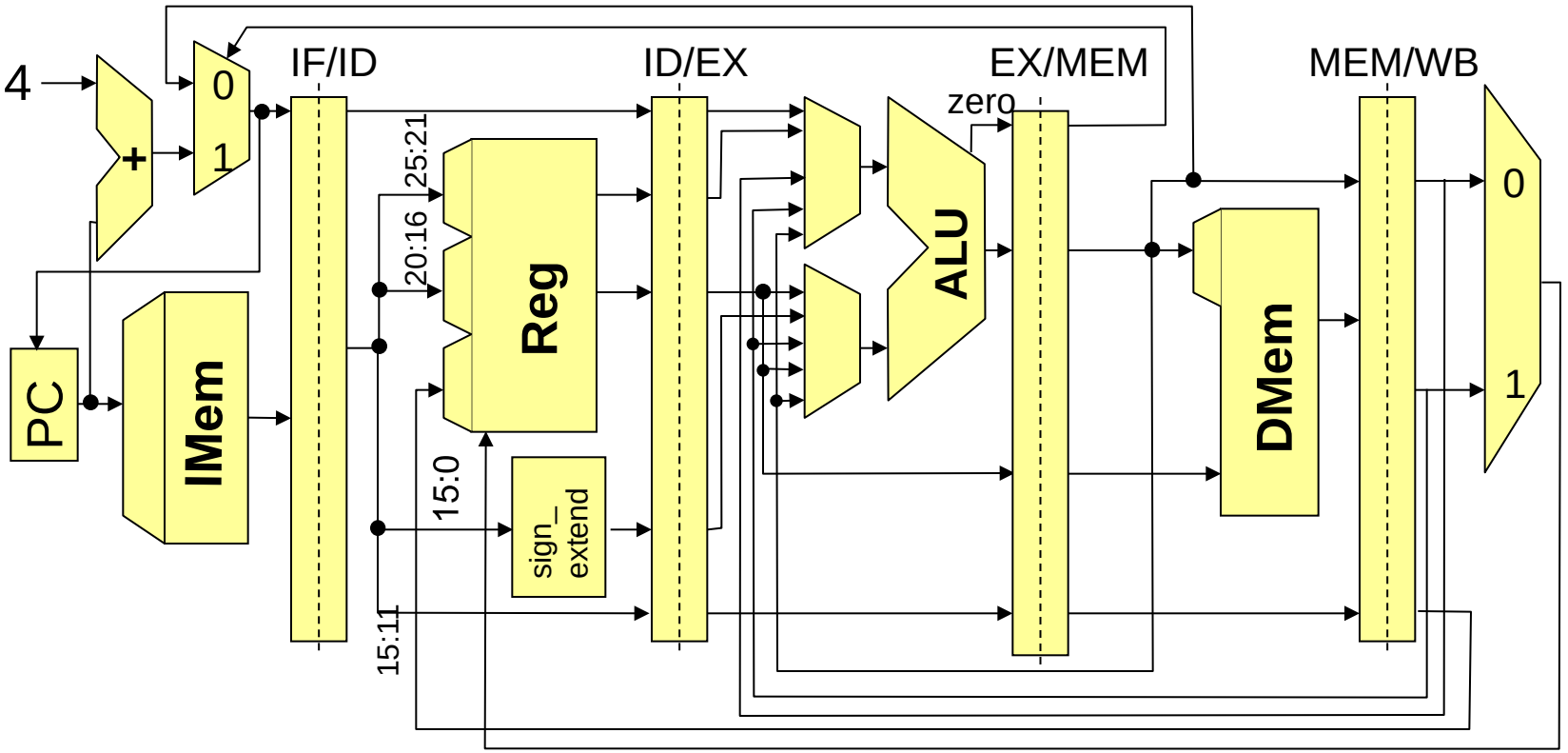
Kontrollabhängigkeiten *(control hazards)* (1)

Beispielprogramm

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

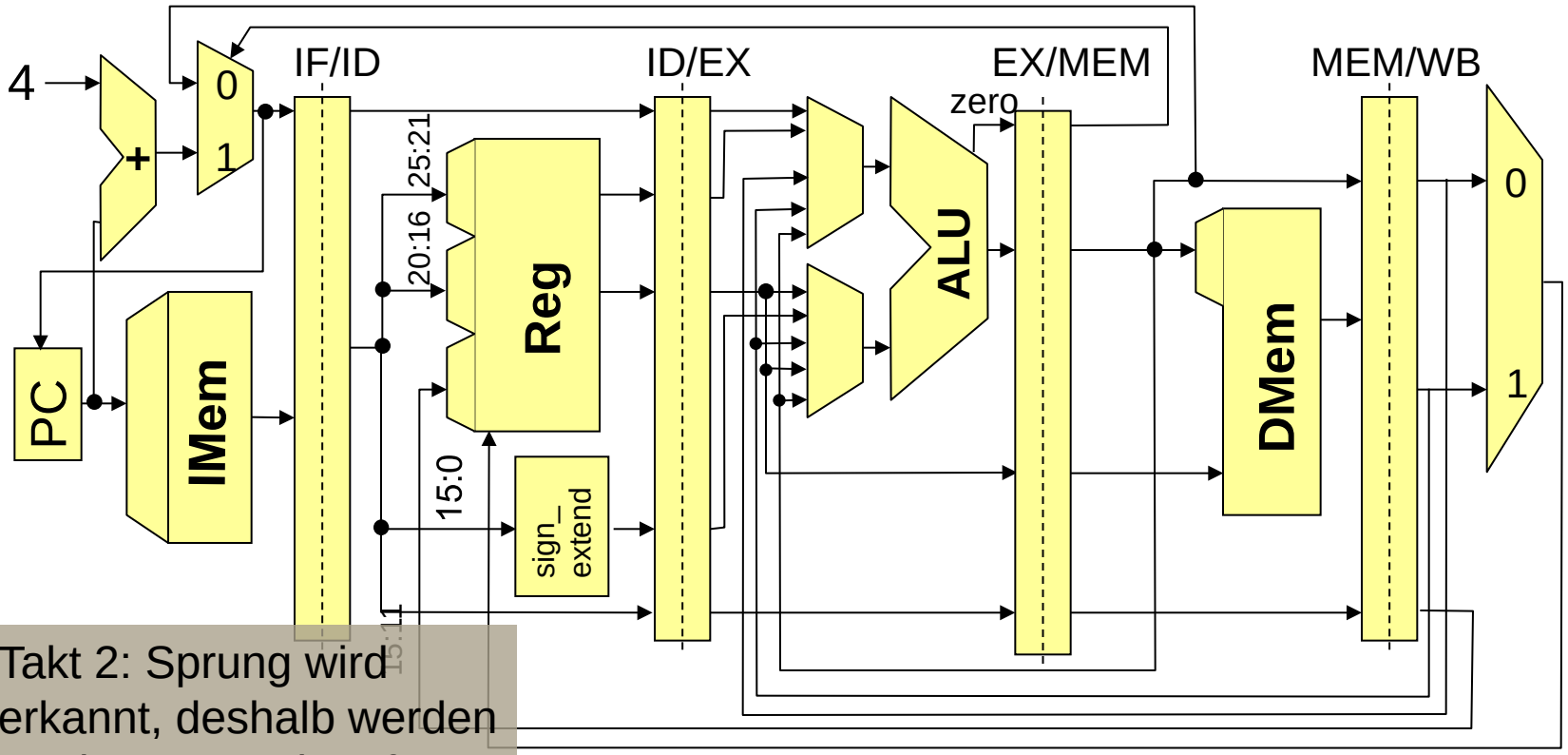
– Wir versuchen zunächst, durch Einfügen von NOOPs die intuitive Bedeutung des Programms zu realisieren…

Kontrollabhängigkeiten *(control hazards)* (2)



Zyklus 1				
`beq $12,$2,t`				

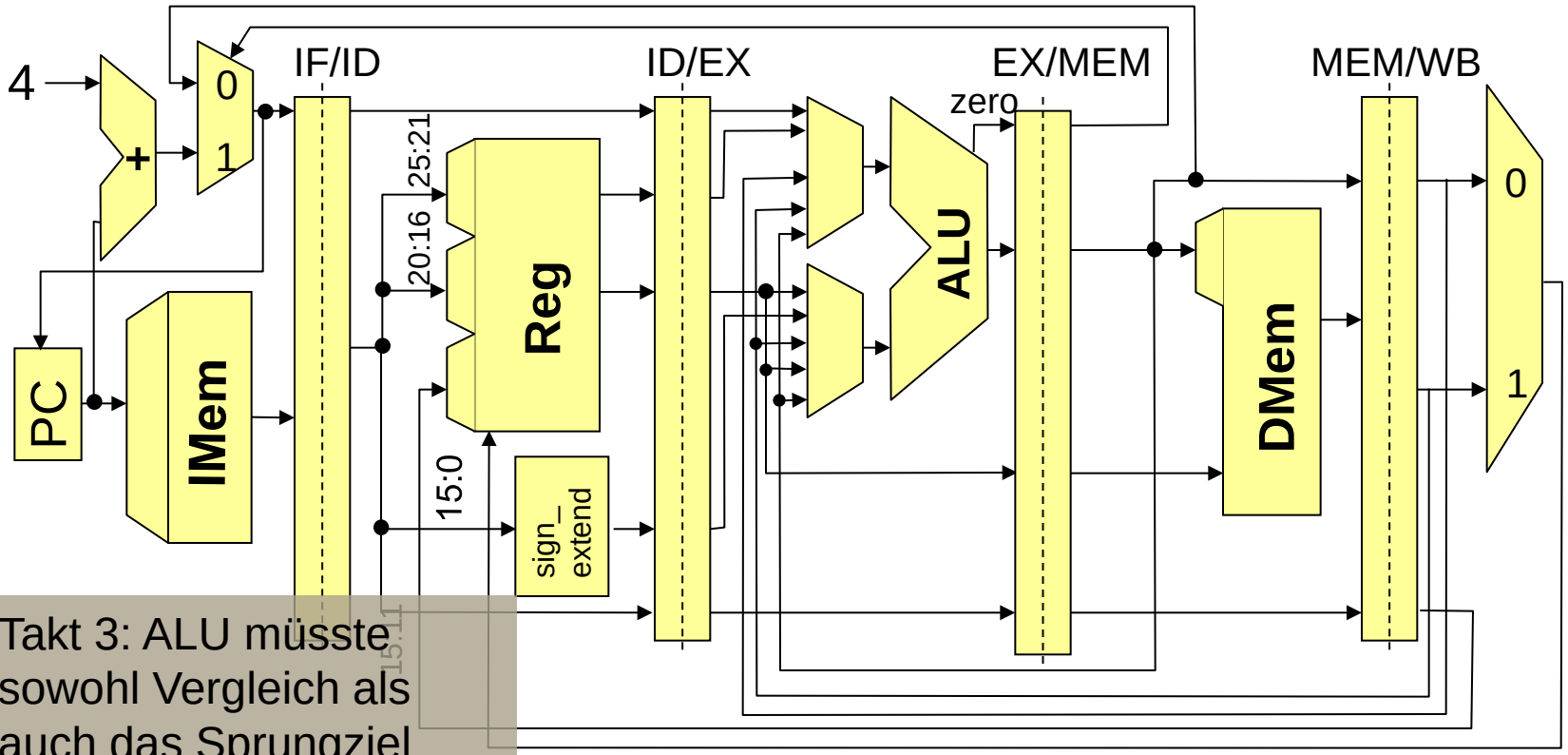
Kontrollabhängigkeiten *(control hazards)* (3)



Takt 2: Sprung wird erkannt, deshalb werden zwei NOOPs eingefügt.

Zyklus 2				
`sub ...`	`beq $12,$2,t`			

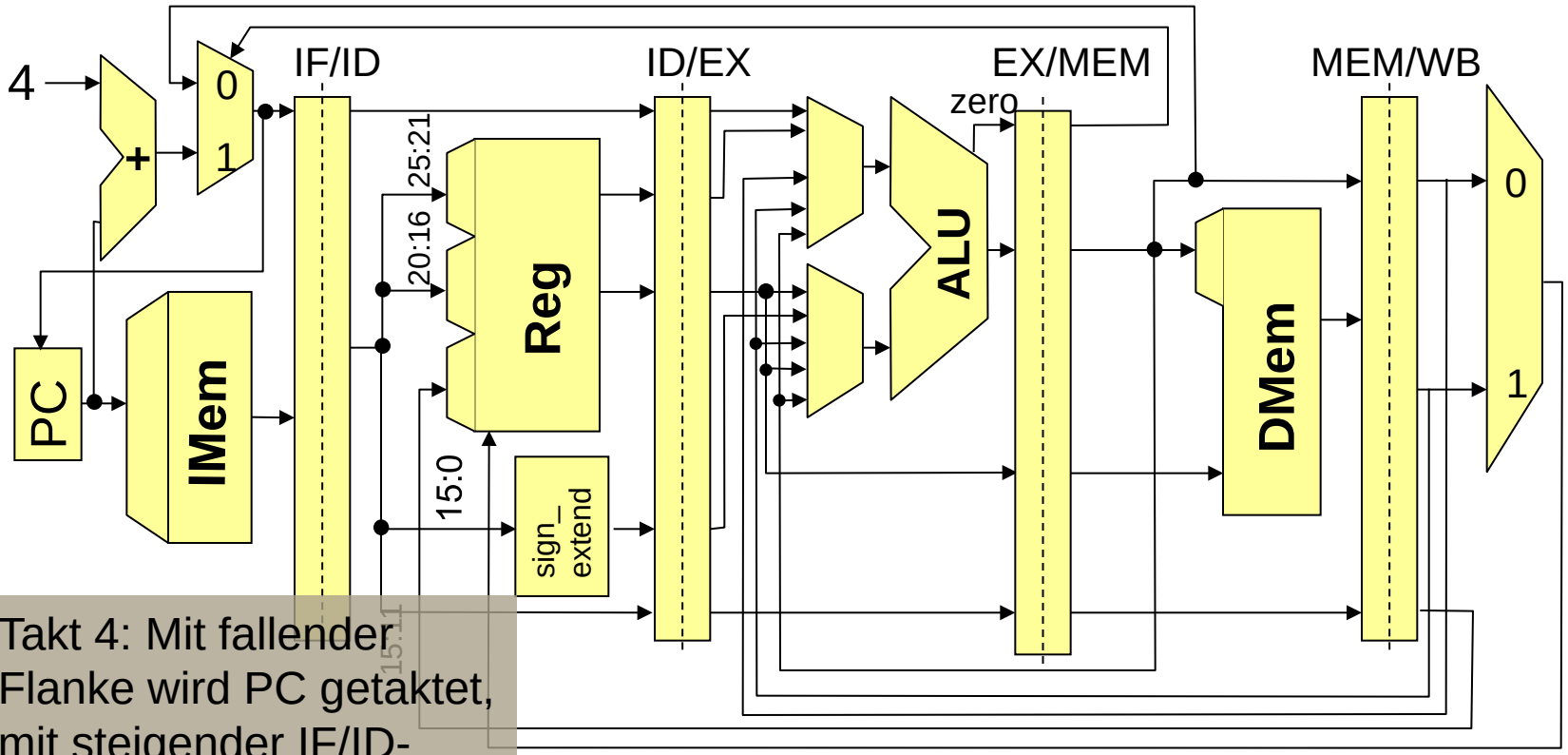
Kontrollabhängigkeiten *(control hazards)* (4)



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

Zyklus 3				
nop	**nop**	**beq $12,$2,t**		

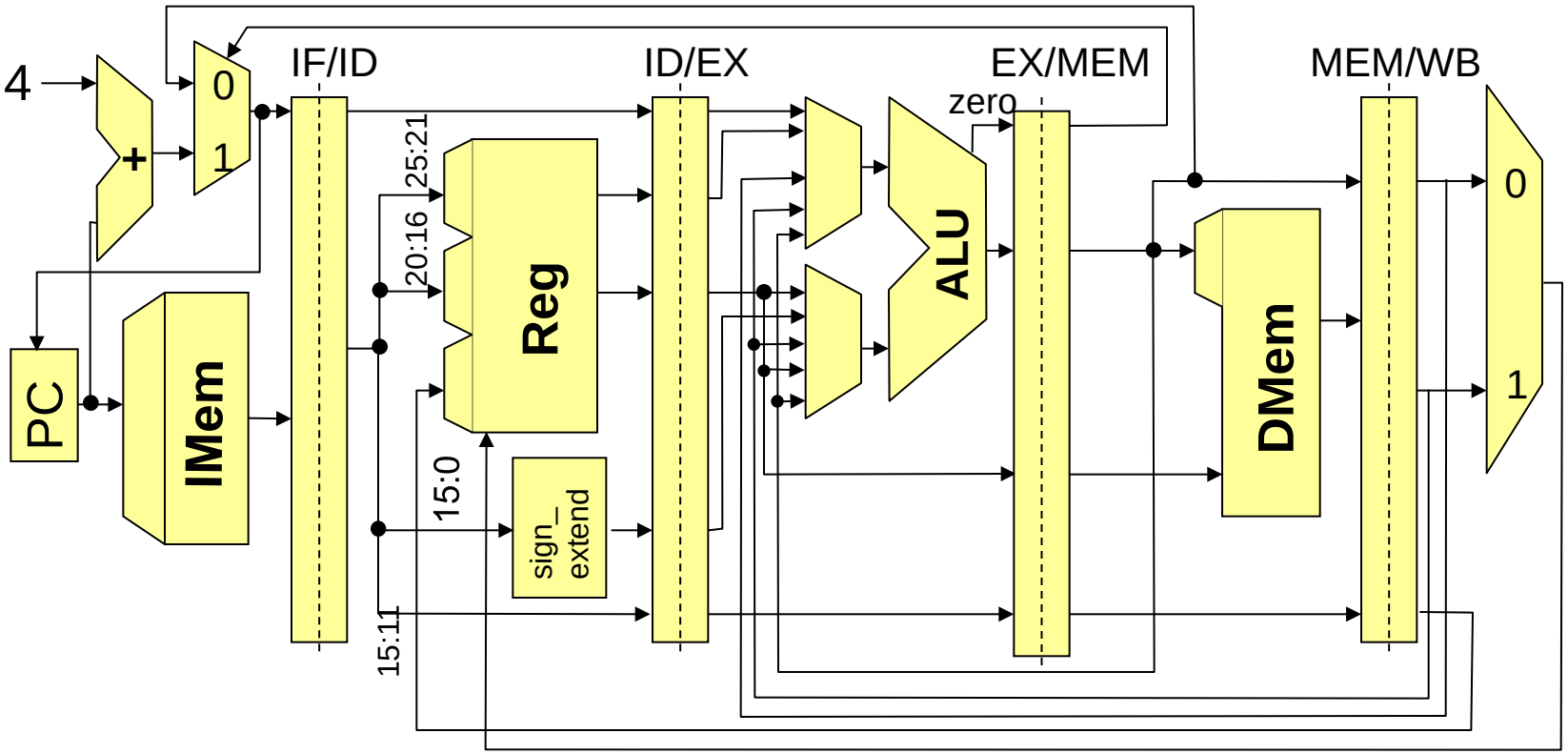
Kontrollabhängigkeiten *(control hazards)* (5)



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

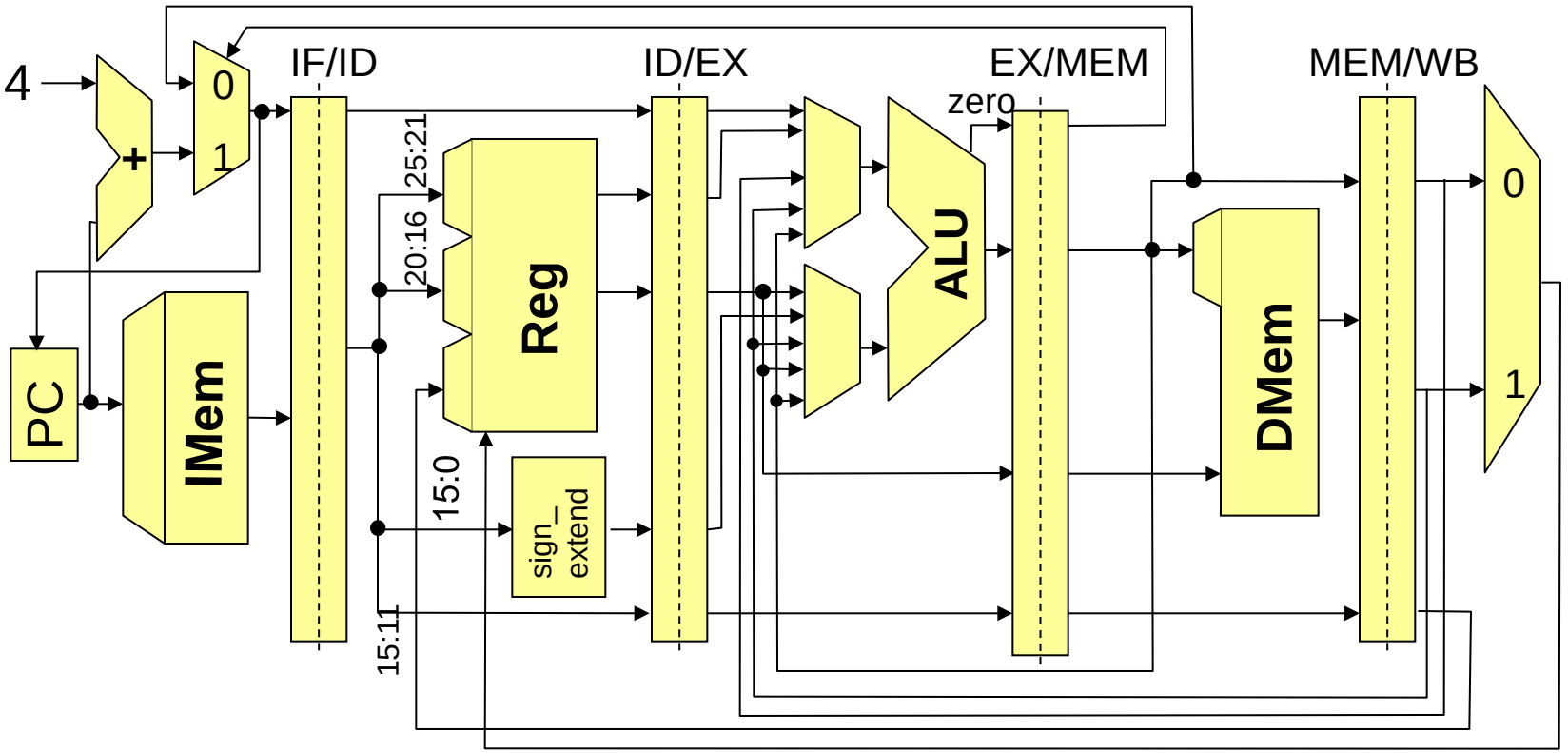
Zyklus 4				
sub oder **add**	**nop**	**nop**	**beq $12,$2,t**	

Kontrollabhängigkeiten *(control hazards)* (6)



Zyklus 5				
`...`	**`sub`** oder **`add`**	**nop**	**nop**	**`beq $12,$2,t`**

Kontrollabhängigkeiten *(control hazards)* (7)



Zyklus 6				
. . .	. . .	**sub** oder **add**	**nop**	**nop**

Kontrollabhängigkeiten *(control hazards)* (8)

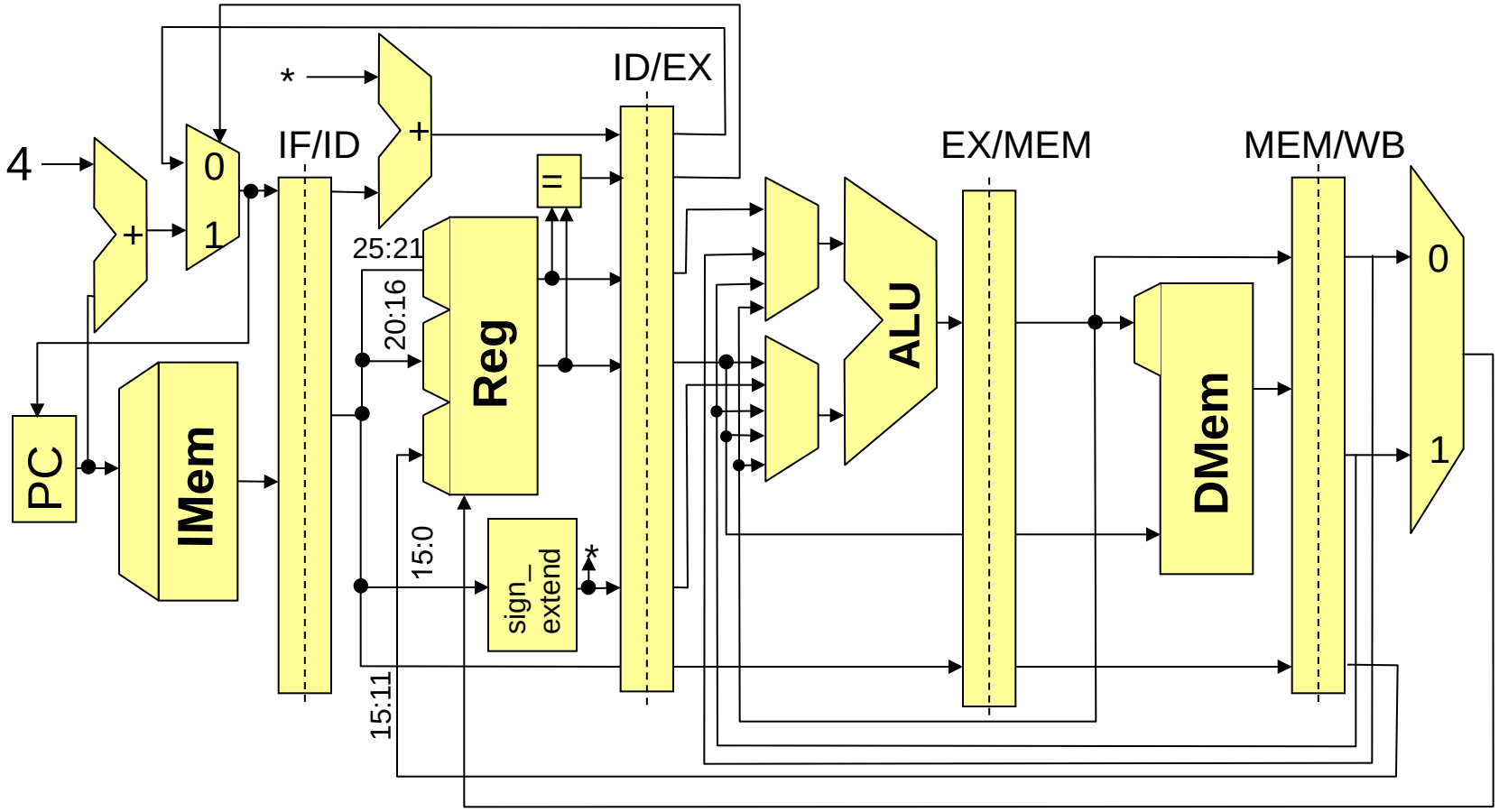
Probleme beim gezeigten Ansatz

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

Lösungsansatz

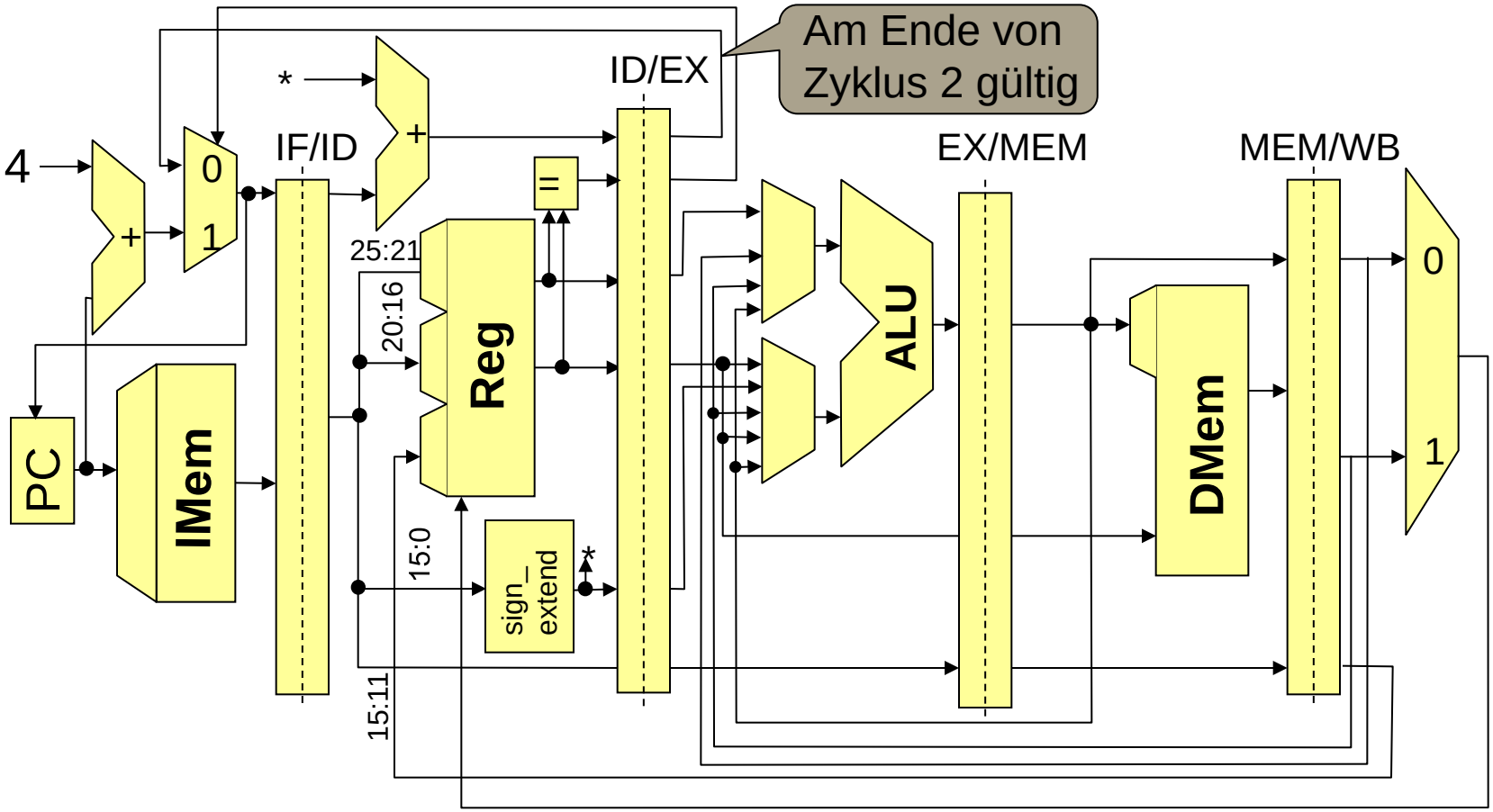
- Gleichheit der Register wird schon in der *instruction decode*-Stufe geprüft
- Sprungziel wird in separatem Adressaddierer ebenfalls bereits in der *instruction decode*-Stufe berechnet
  (☞ ähnlich zu Folie 95)
- Sofern weiterhin noch Verzögerungen auftreten:
  - nächsten Befehl einfach ausführen *(delayed branch)*
  - oder weiterhin NOOP(s) einfügen *(stall)*

Reduktion der *branch delay penalty – delayed branch* (1)



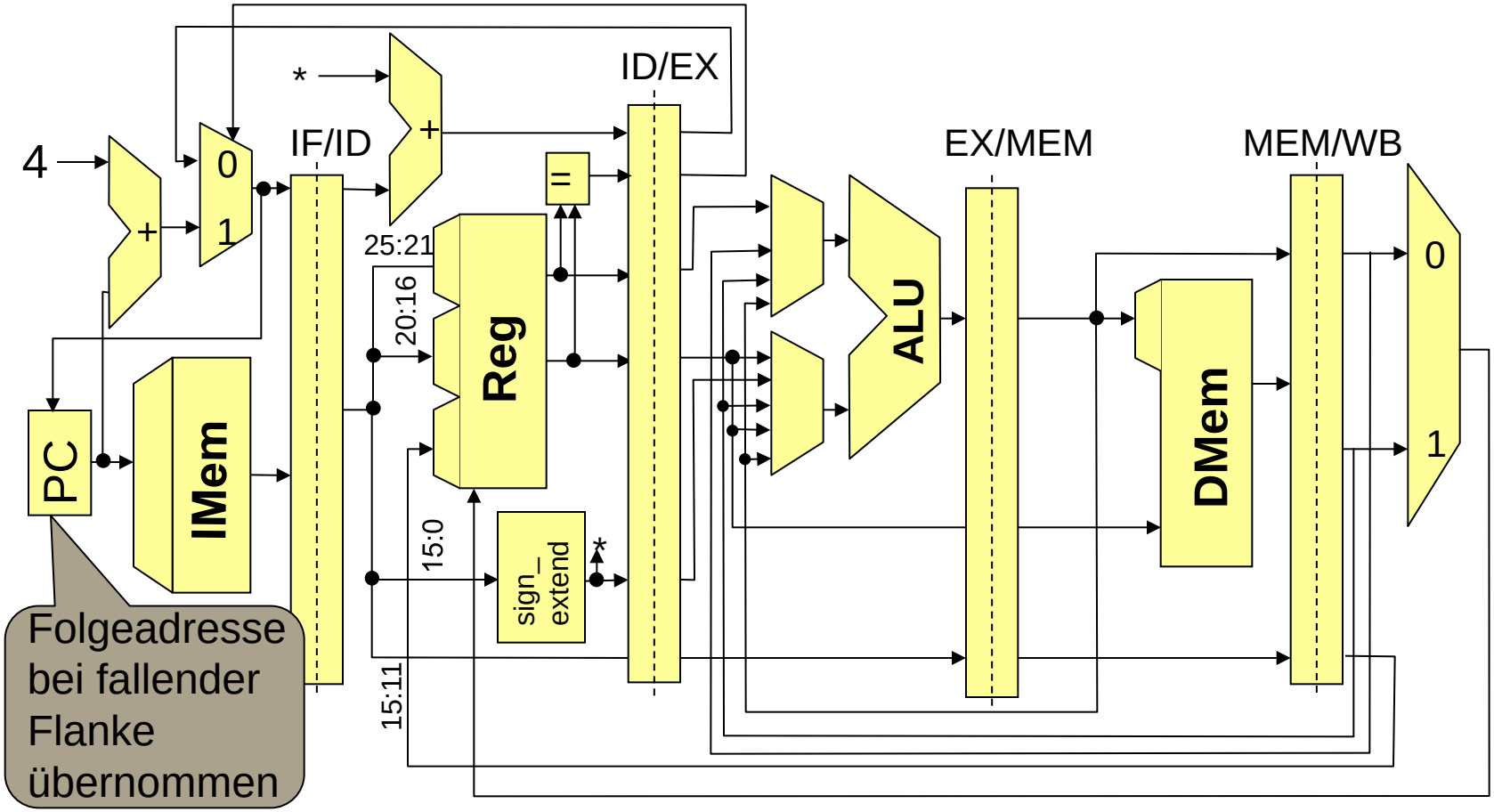
Zyklus 1				
`beq $12,$2,t`				

Reduktion der *branch delay penalty – delayed branch* (2)



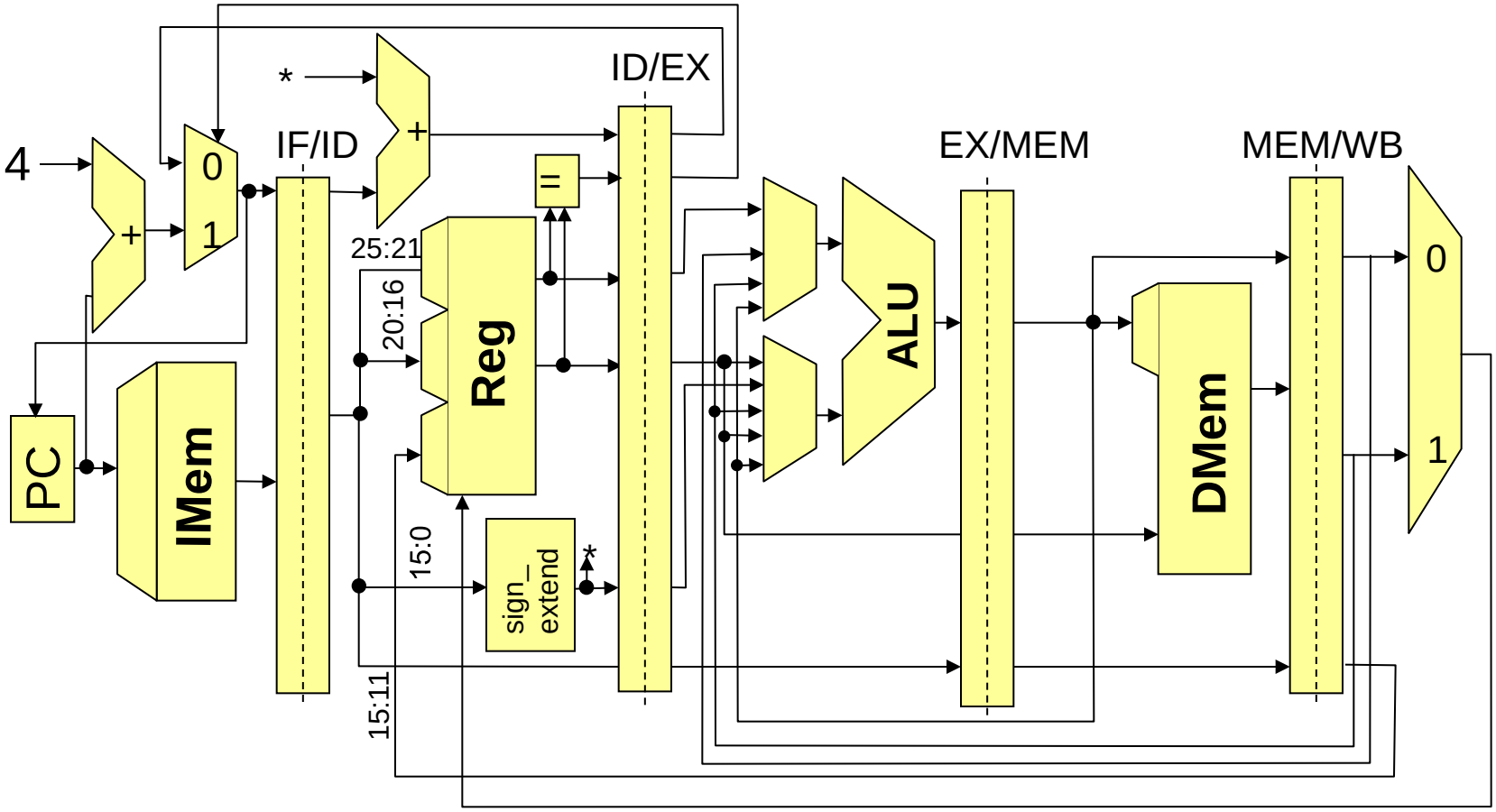
Zyklus 2				
`sub ...`	`beq $12,$2,t`			

Reduktion der *branch delay penalty – delayed branch* (3)



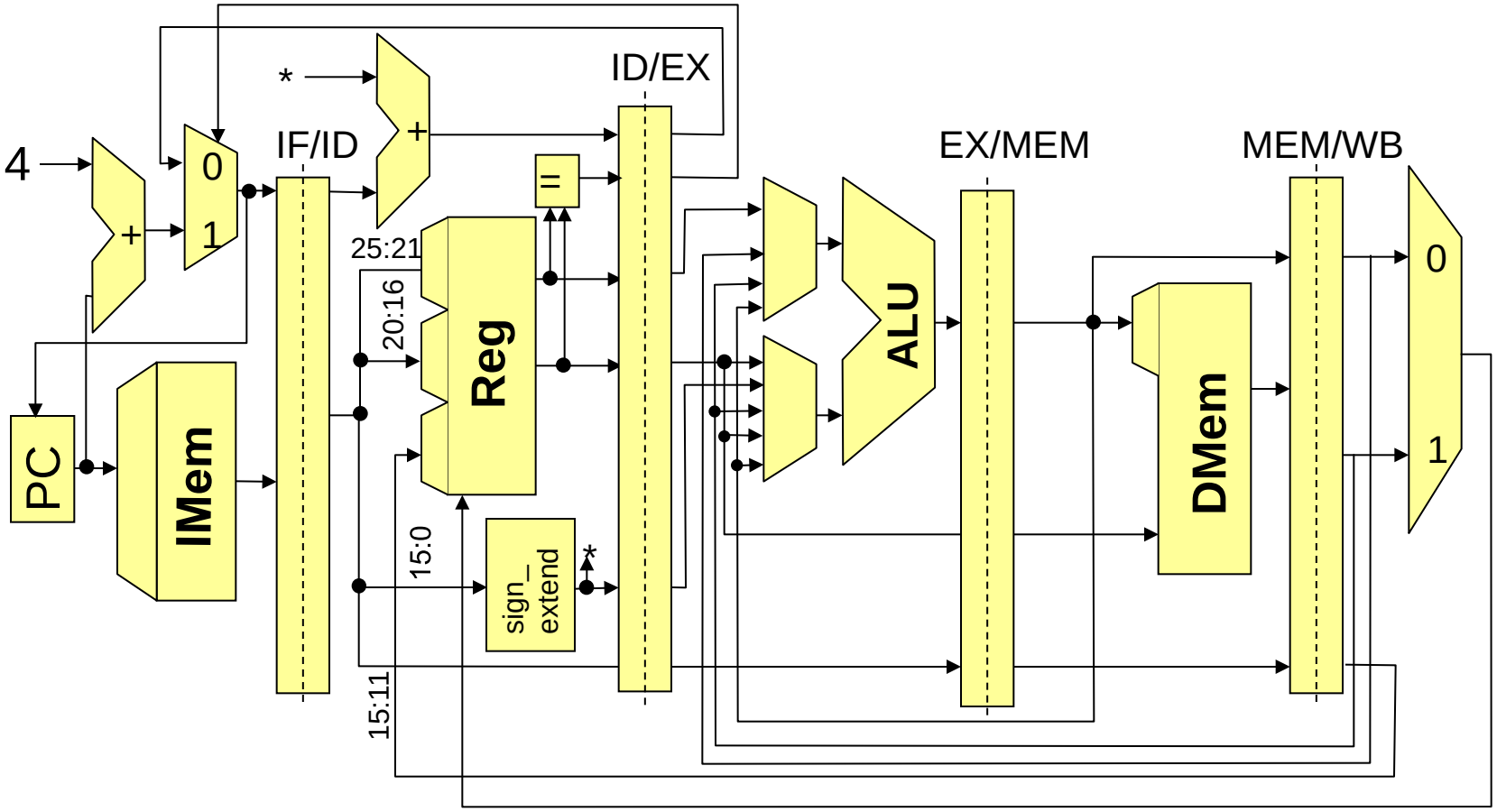
Zyklus 3				
`add ...`	`sub ...`	`beq $12,$2,t`		

Reduktion der *branch delay penalty – delayed branch* (4)



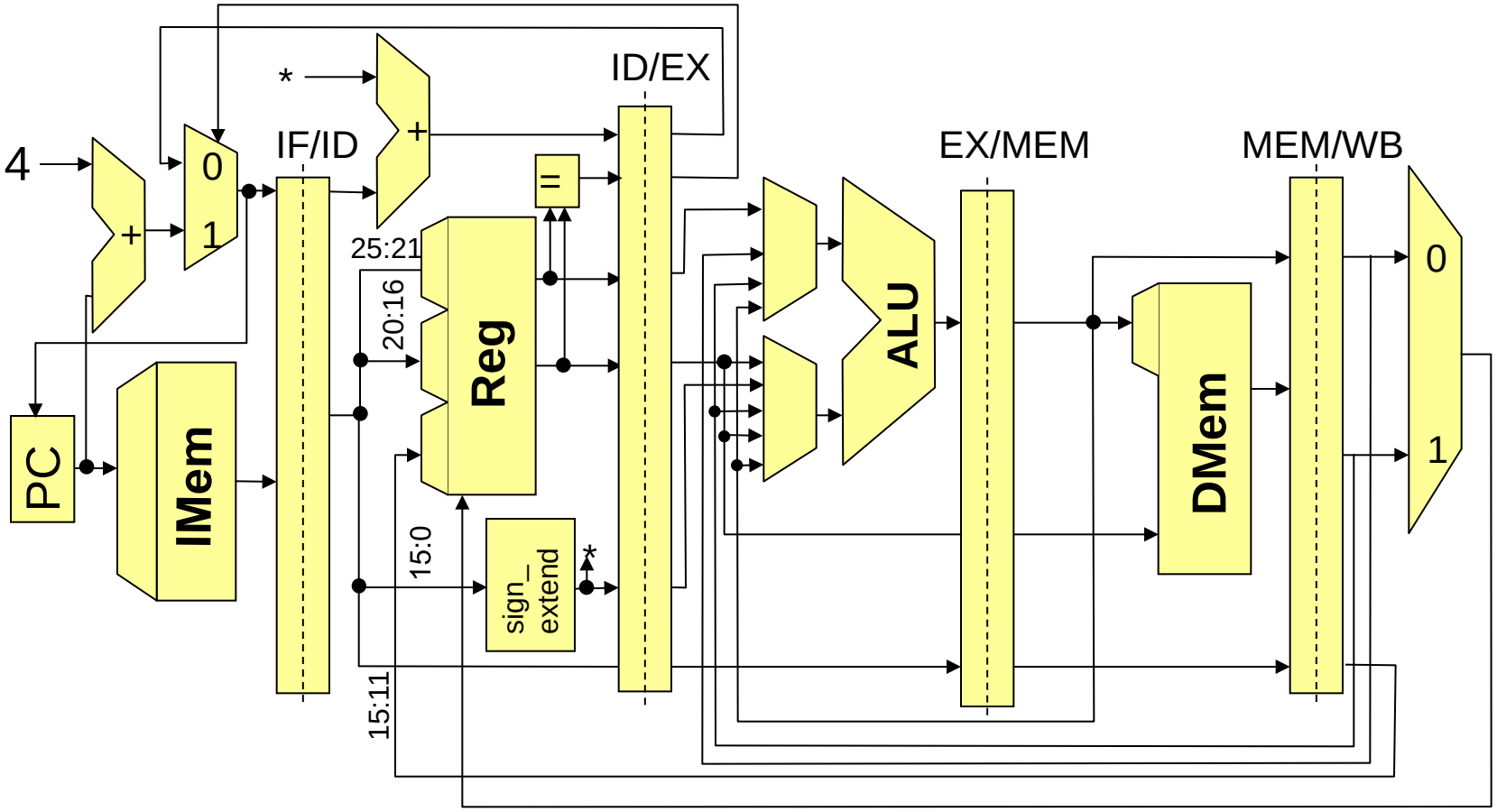
Zyklus 4				
`...`	`add ...`	`sub ...`	`beq $12,$2,t`	

Reduktion der *branch delay penalty – delayed branch* (5)



Zyklus 5				
`...`	`...`	`add ...`	`sub ...`	`beq $12,$2,t`

Reduktion der *branch delay penalty – delayed branch* (6)

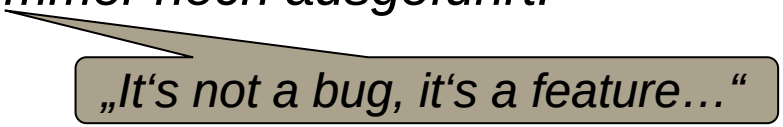


Zyklus 6				
. . .	. . .	. . .	**add . . .**	**sub . . .**

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 ...
```



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 Compilern mit nützlichen Befehlen gefüllt werden.

Nur notfalls sollte es ein NOOP sein.

Die MIPS-Maschine hat einen *delay slot*, der aber vom Assembler verdeckt wird.

Pipelining – Zusammenfassung

- Die Fließbandverarbeitung (engl. *pipelining*) ermöglicht es, in jedem Takt die Bearbeitung eines Befehls abzuschließen, selbst wenn die Bearbeitung eines Befehls ≥ 1 Takte dauert
- Mehrere *Pipelines* ☞ 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*
  - *delayed branches*
  - *branch prediction*
  - *out-of-order execution, dynamic scheduling*

Roter Faden

6. Grundlagen der Rechnerarchitektur

- Grundbegriffe der Rechnerarchitektur
- Programmiermodelle / die Befehlsschnittstelle
- Aufbau einer MIPS-Einzelzyklusmaschine
- Fließbandverarbeitung / *Pipelining*
  - Fließband-Architektur
  - *Pipeline-Hazards*
    - Datenabhängigkeiten (RAW, WAR, WAW)
    - Bypässe / *Forwarding*
    - *Pipeline Stalls*
    - Kontrollabhängigkeiten, *branch delay penalty*
    - *Delayed Branches*
- Dynamisches *Scheduling*

Dynamisches Befehlsscheduling – *in-order execution*

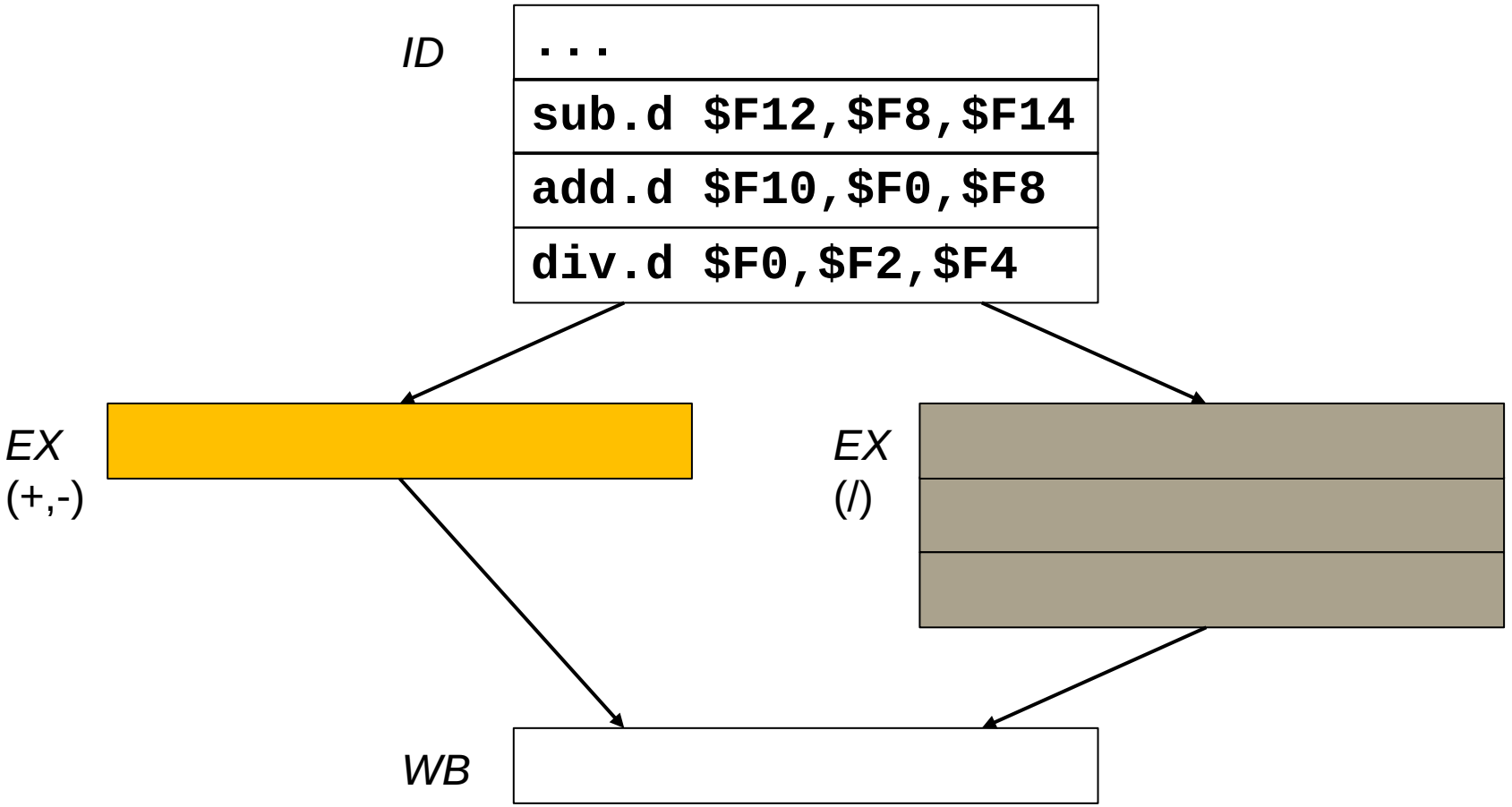
Bislang

– Reihenfolge der Befehlsabarbeitung =
  Reihenfolge der Befehle im Speicher, abgesehen von Sprüngen
– Behindert schnelle Ausführung

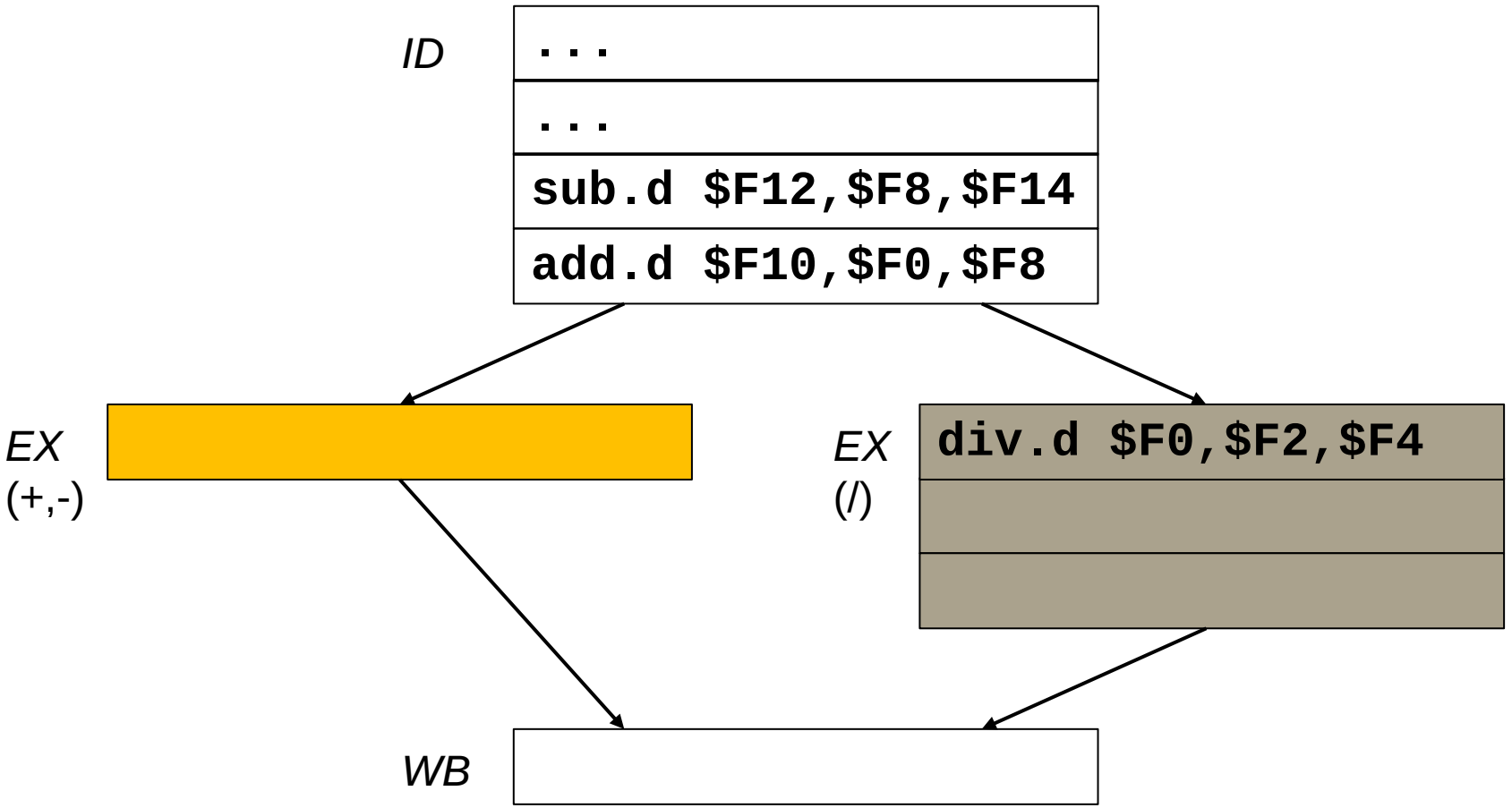
Beispiel

```
sub.d $F12,$F8,$F14
add.d $F10,$F0,$F8
div.d $F0,$F2,$F4
```

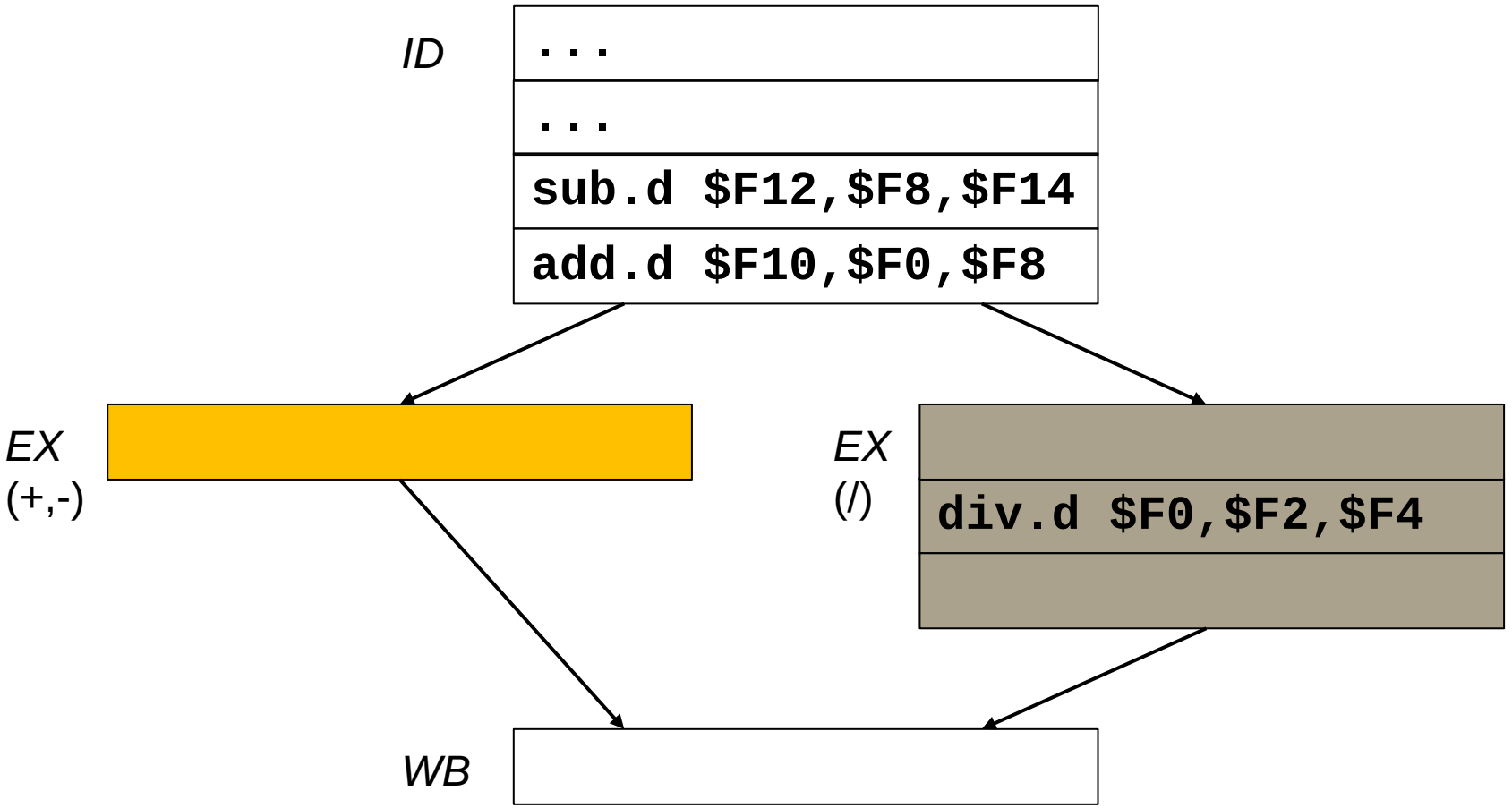
In-order execution (1)



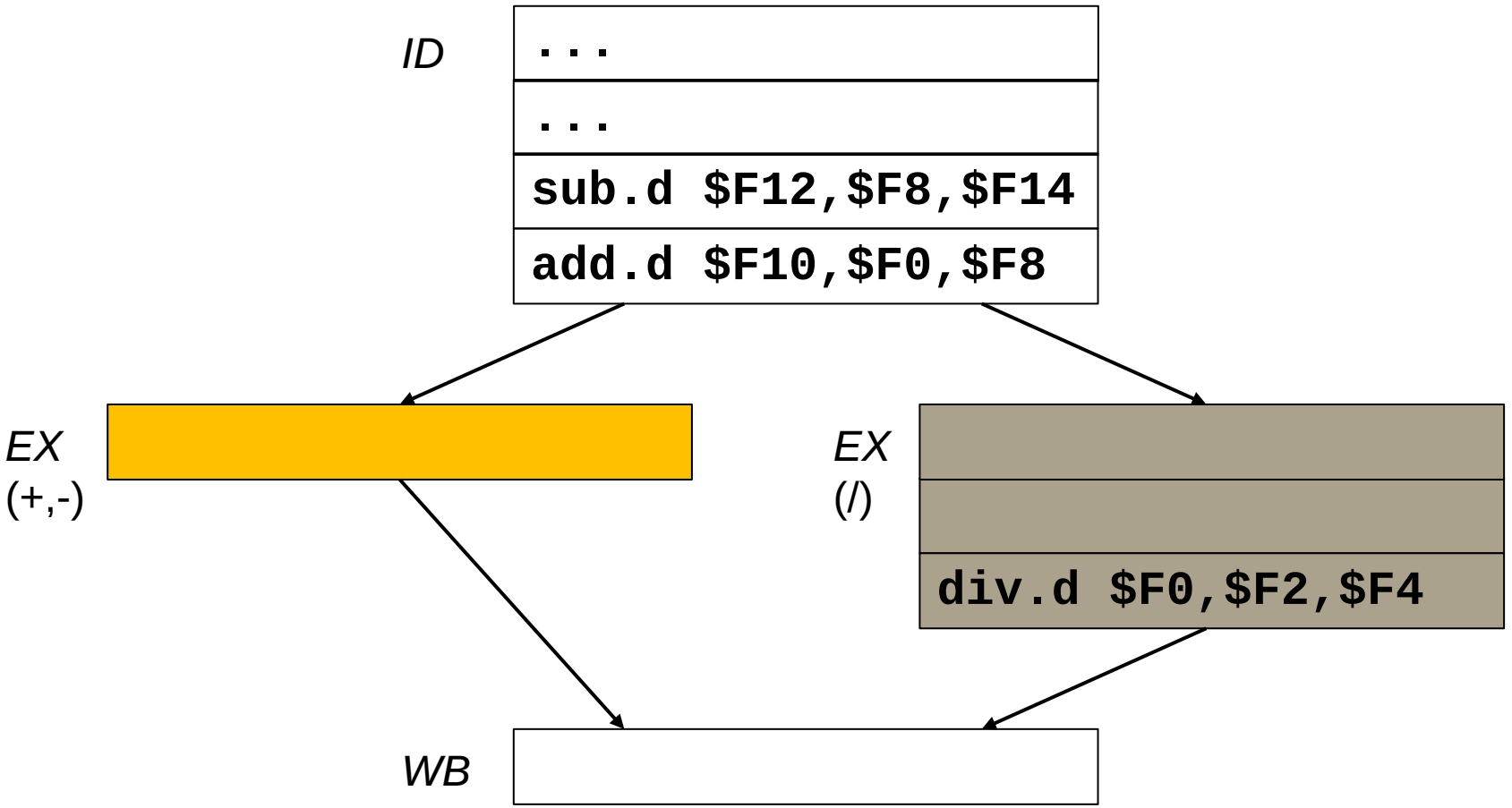
In-order execution (2)



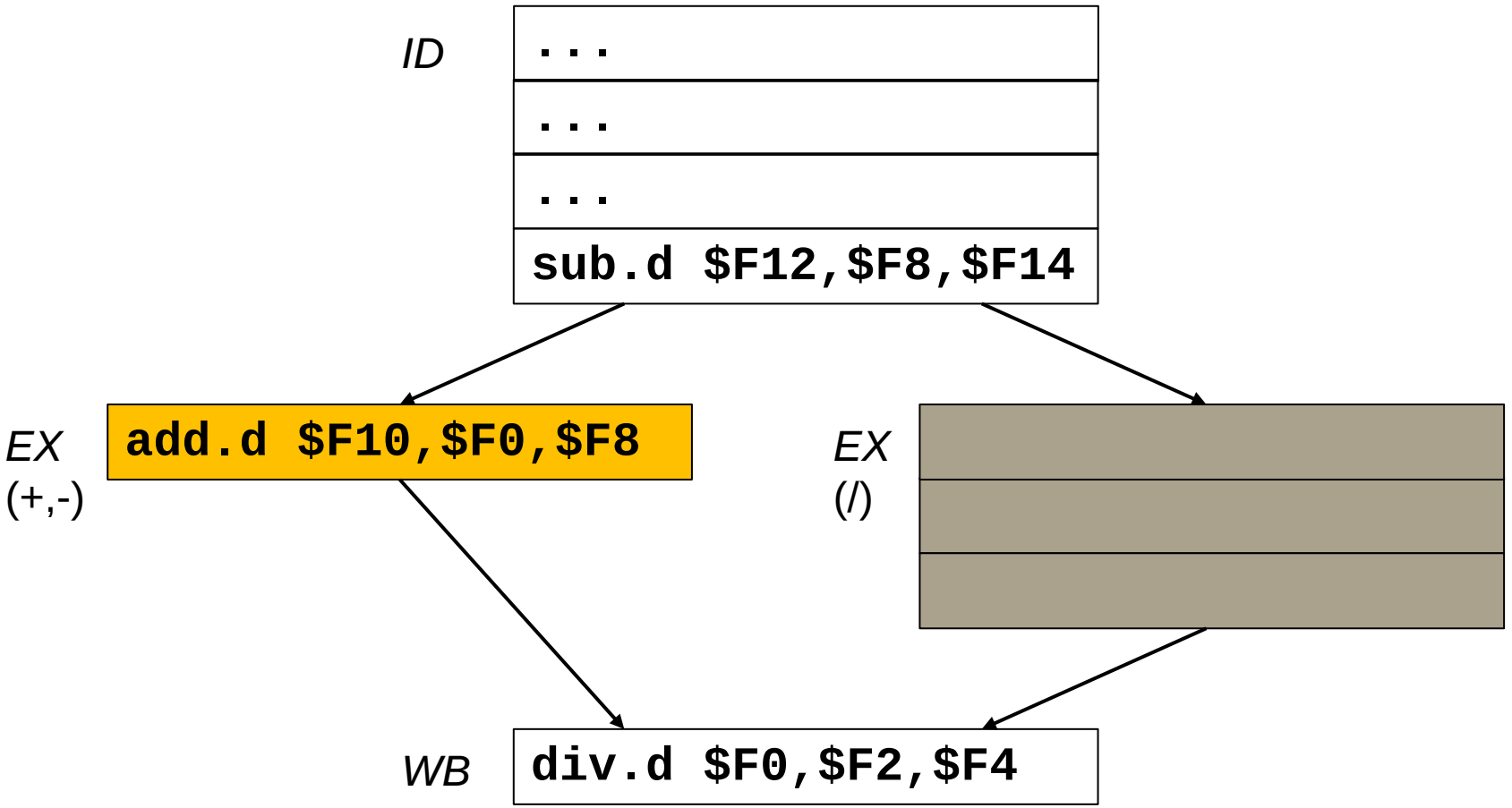
In-order execution (3)



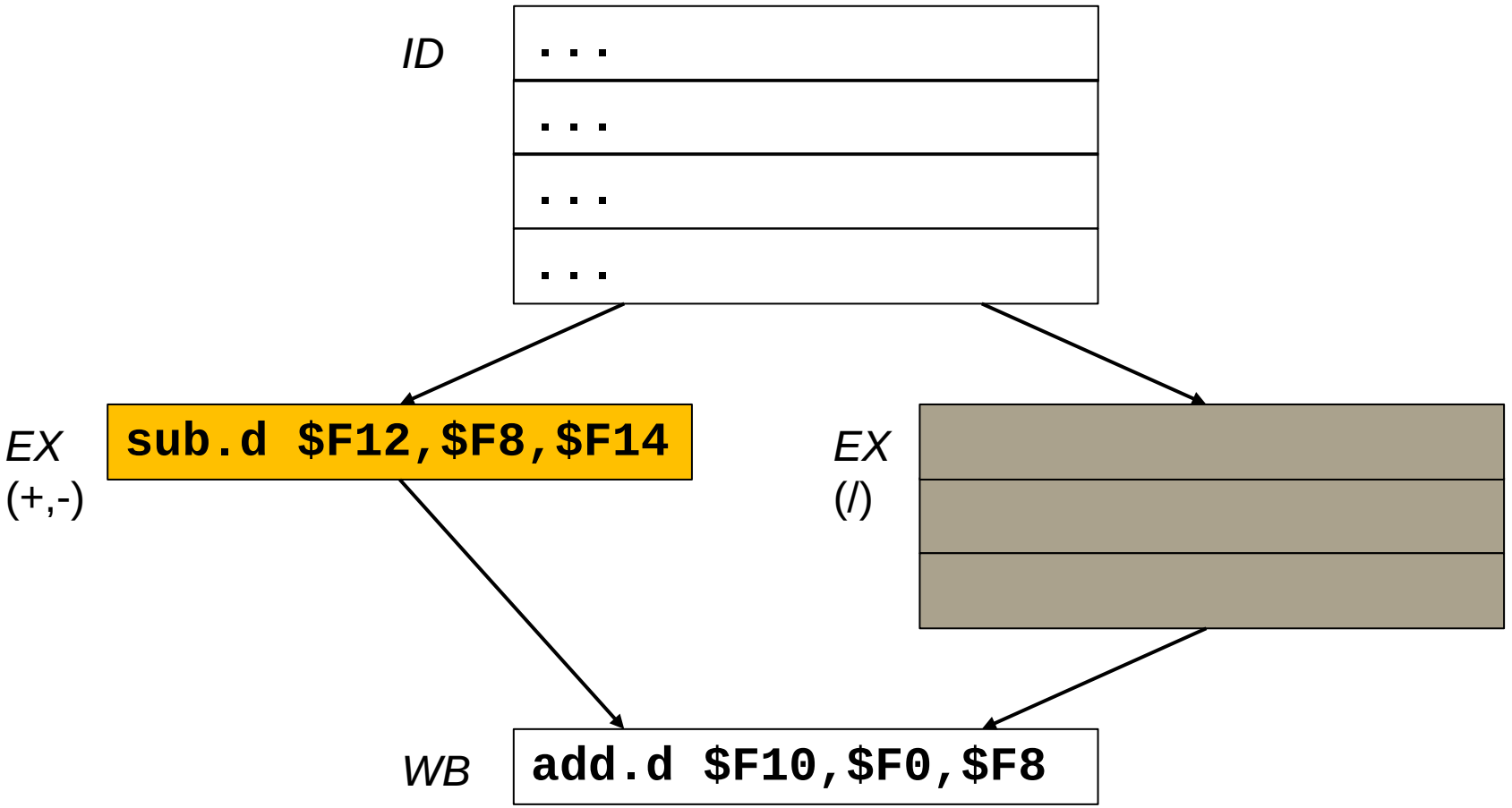
In-order execution (4)



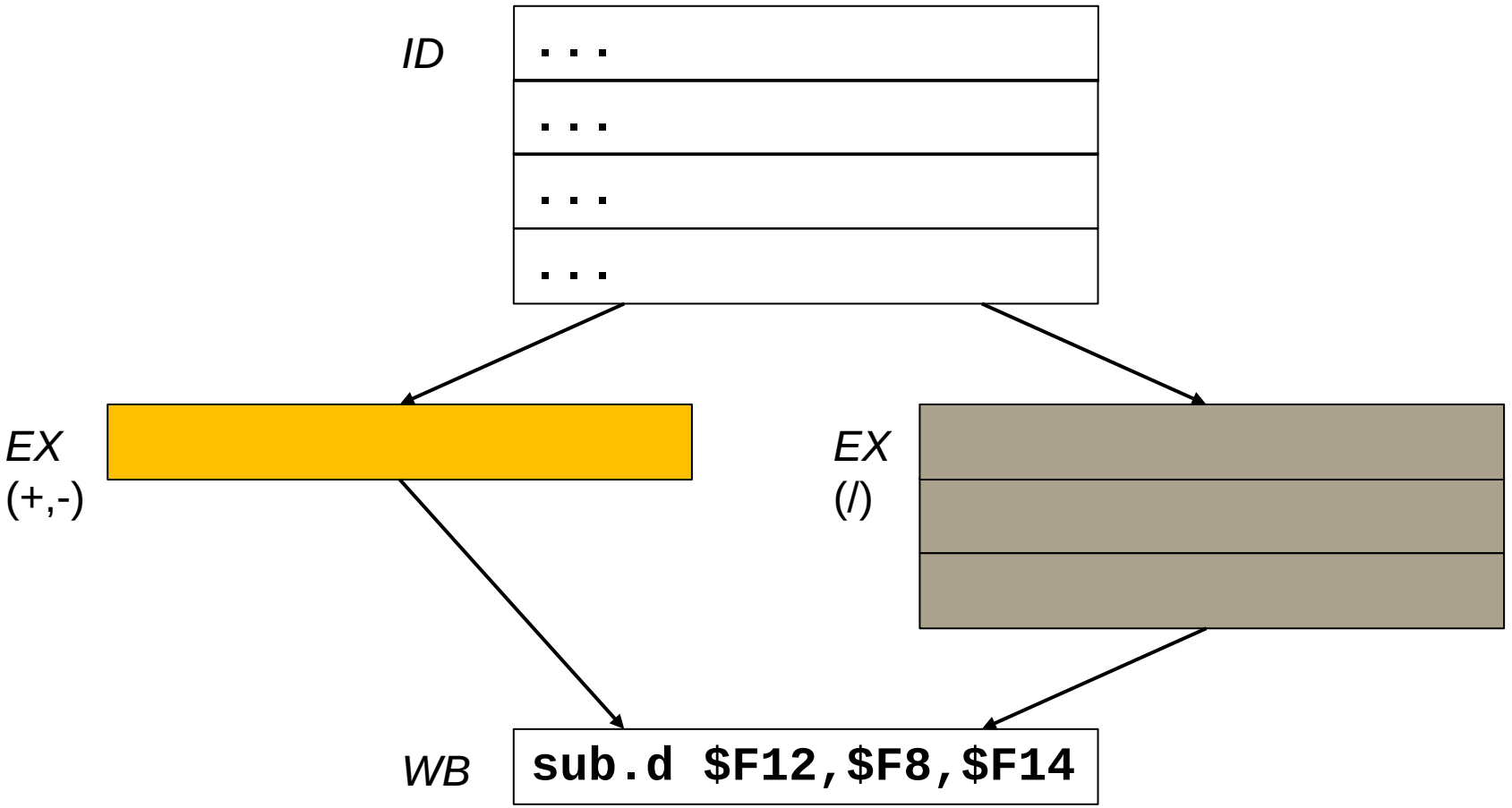
In-order execution (5)



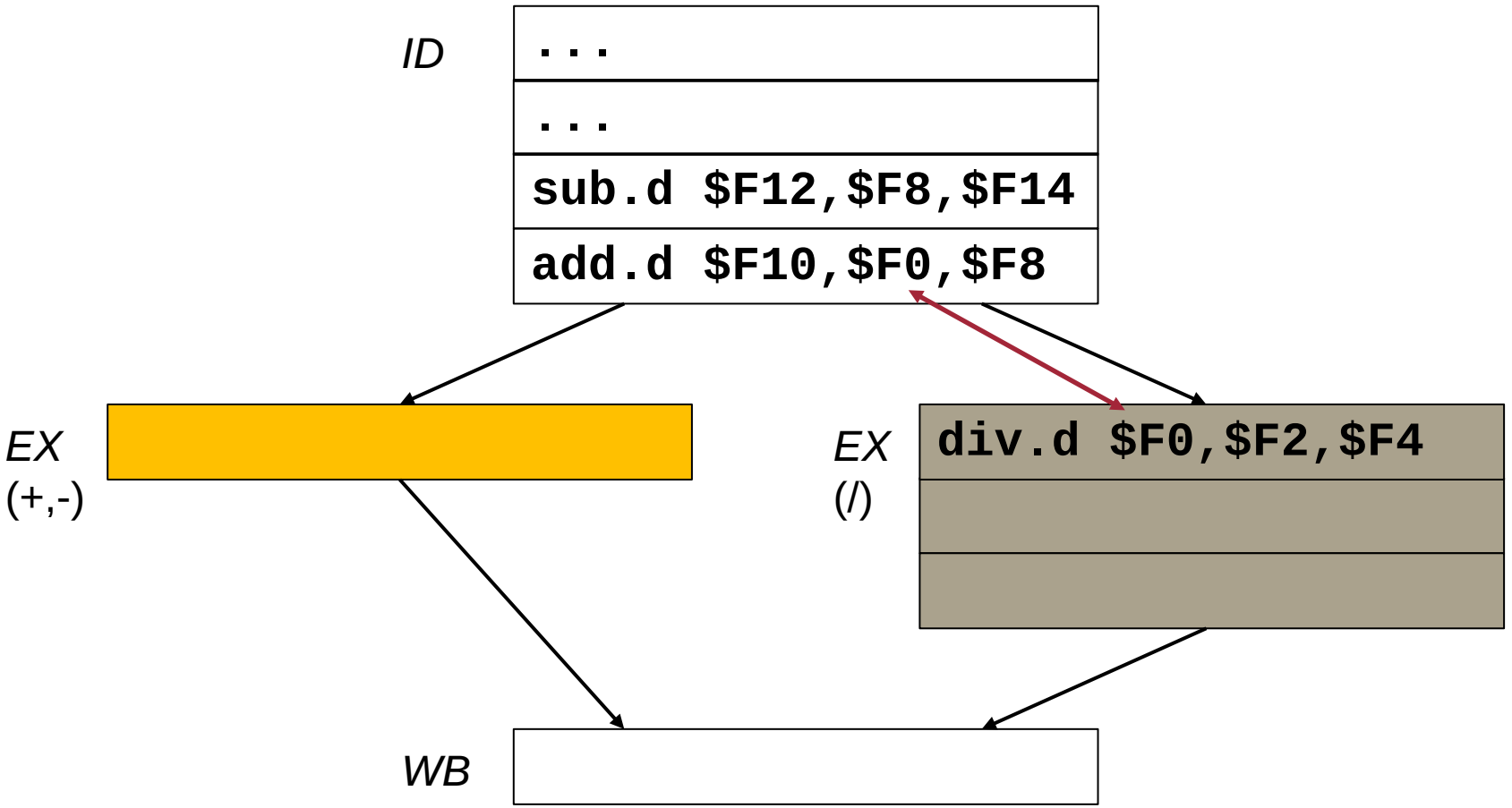
In-order execution (6)



In-order execution (7)

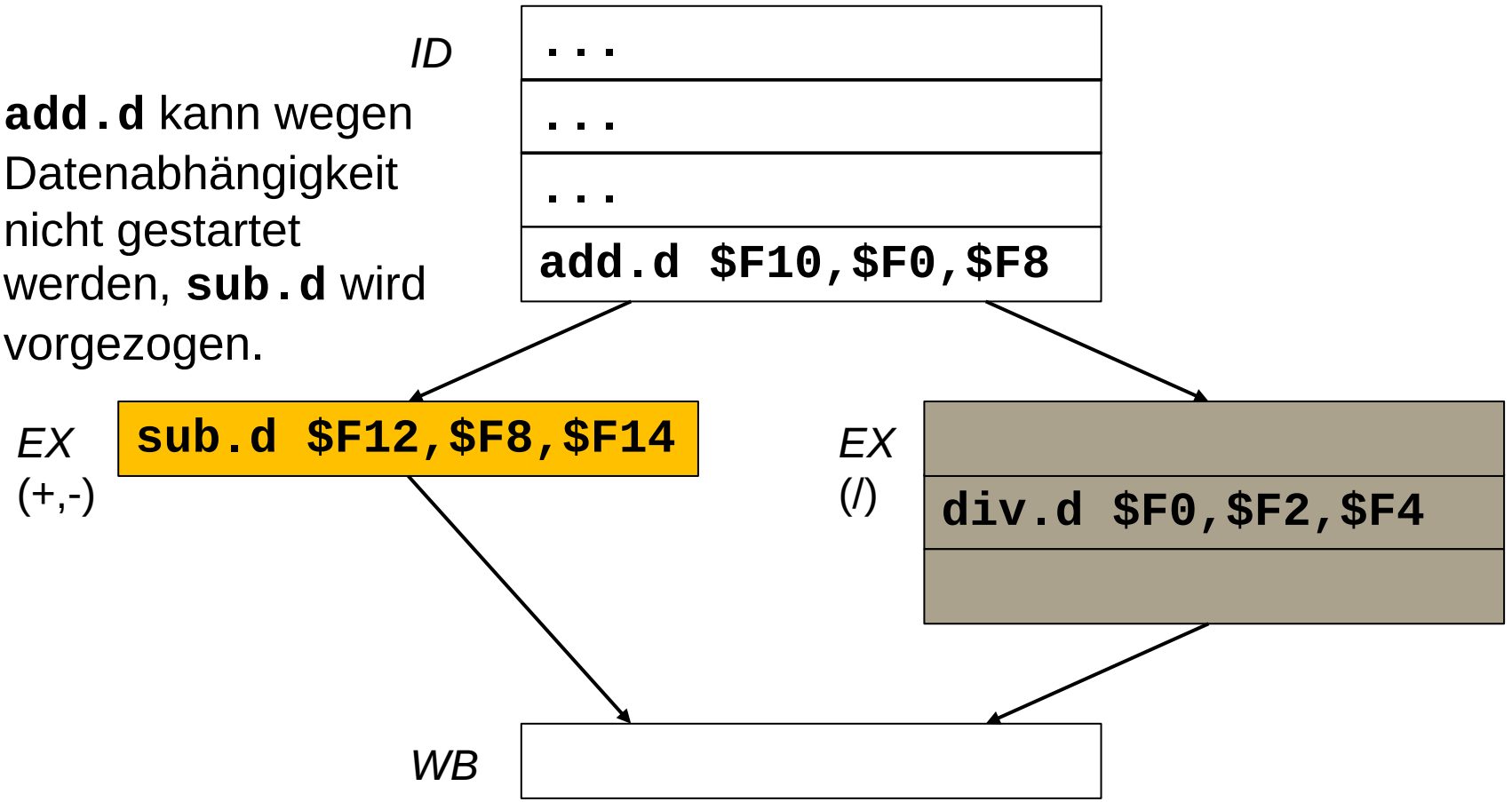


Dynamisches *Scheduling – out-of-order execution* (1)



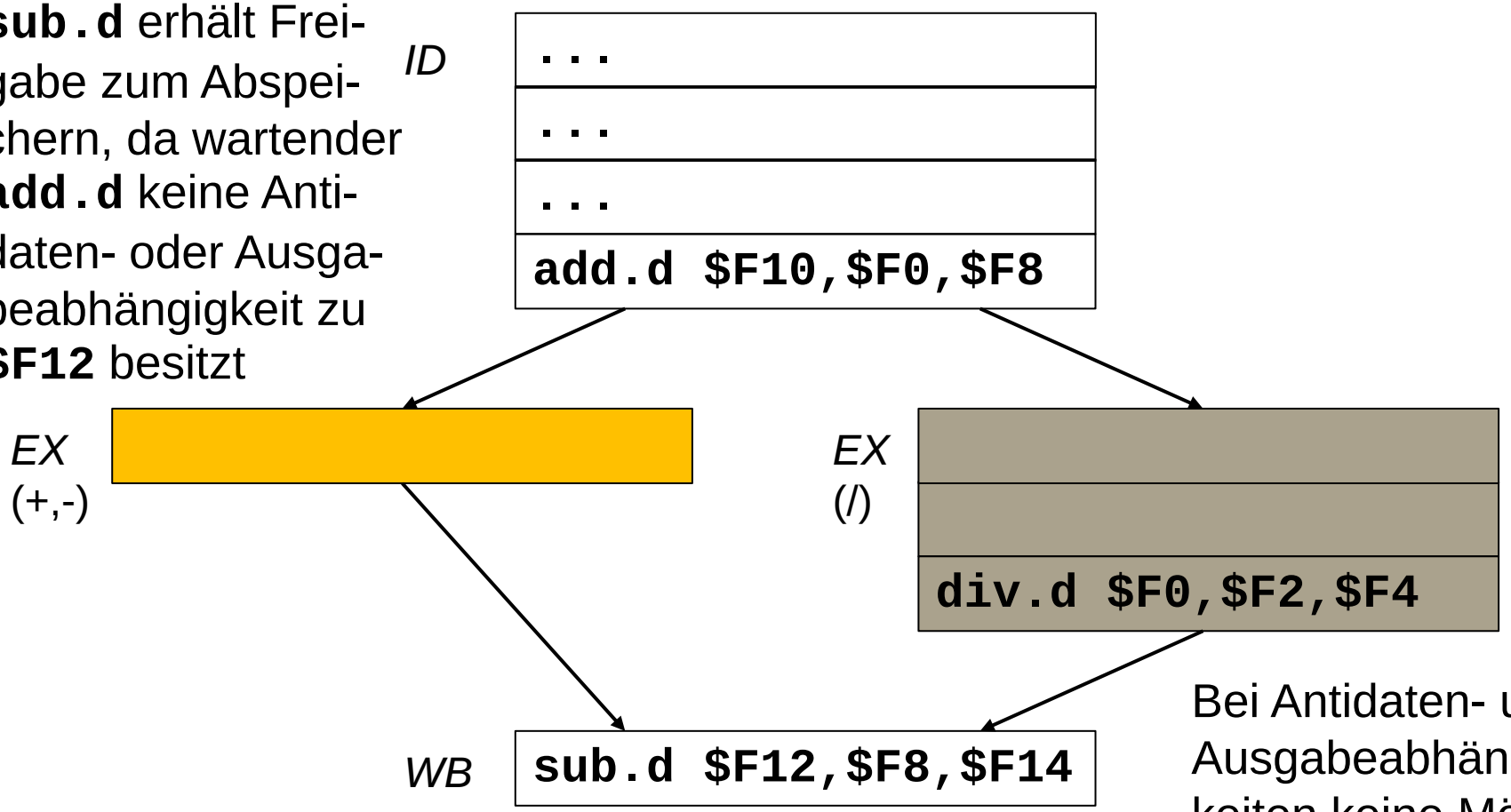
Dynamisches *Scheduling – out-of-order execution* (2)

add.d kann wegen Datenabhängigkeit nicht gestartet werden, **sub.d** wird vorgezogen.



Dynamisches *Scheduling – out-of-order execution* (3)

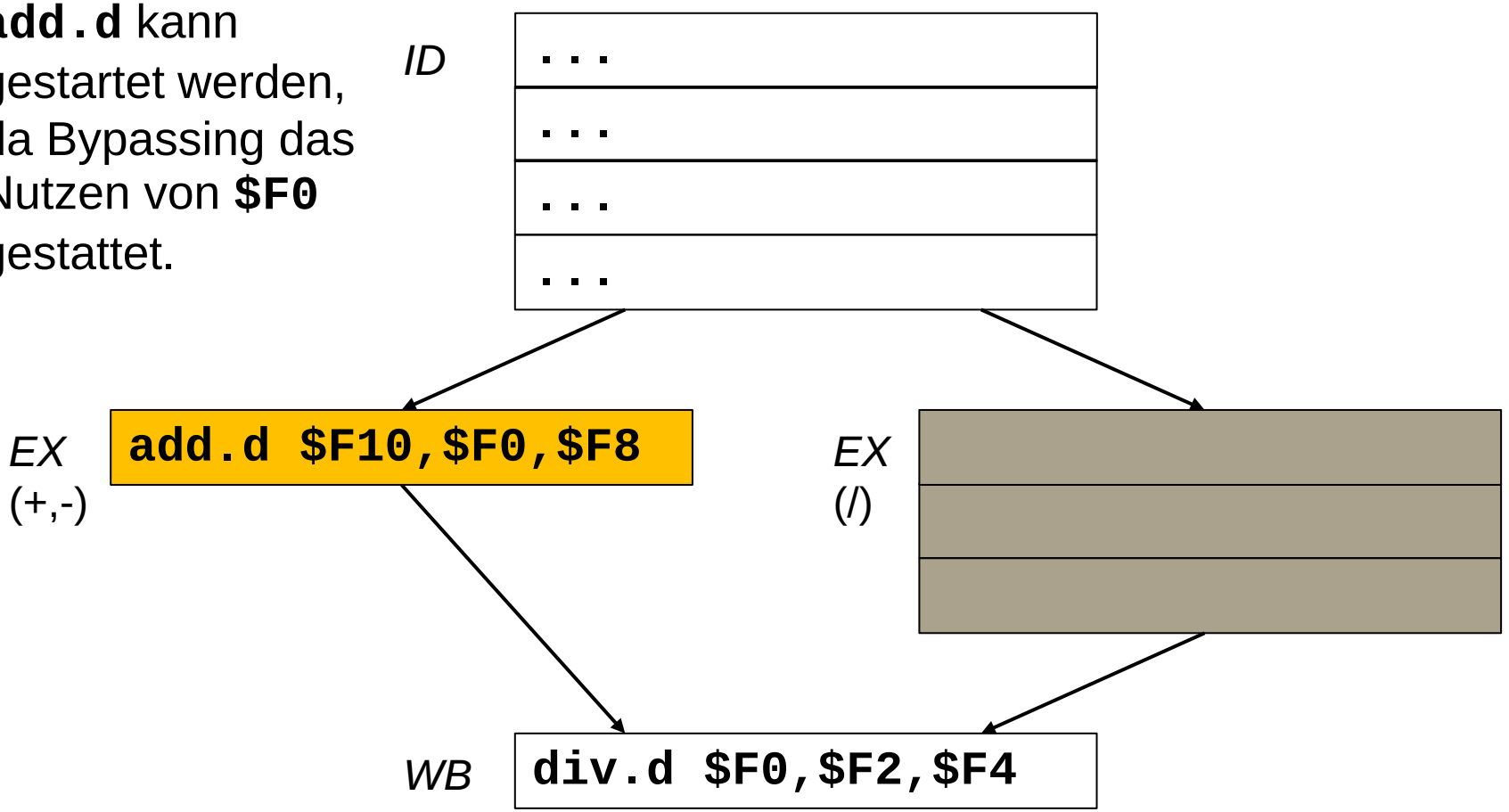
sub.d erhält Freigabe zum Abspeichern, da wartender **add.d** keine Antidaten- oder Ausgabeabhängigkeit zu **$F12** besitzt



Bei Antidaten- und Ausgabeabhängigkeiten keine Möglichkeit, andere Befehle zu überholen.

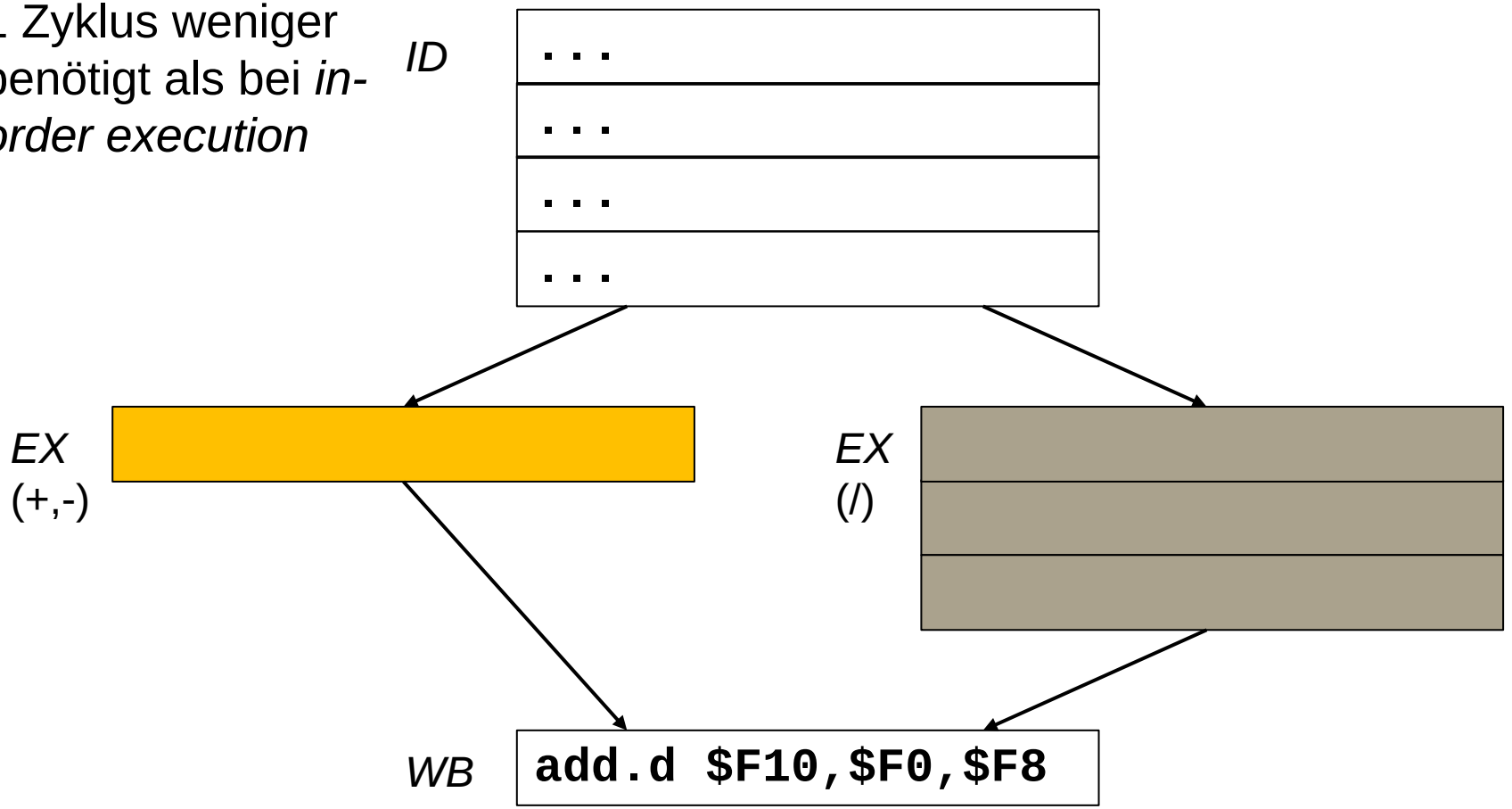
Dynamisches *Scheduling – out-of-order execution* (4)

`add.d` kann gestartet werden, da Bypassing das Nutzen von **`$F0`** gestattet.



Dynamisches *Scheduling – out-of-order execution* (5)

1 Zyklus weniger benötigt als bei *in-order execution*



Scoreboarding (1)

Jeder Befehl, der aus der *Instruction fetch*-Einheit kommt, durchläuft das *Scoreboard*.

Wenn für einen Befehl alle Daten/Operanden bekannt sind und die Ausführungseinheit frei ist, wird der Befehl gestartet.

Alle Ausführungseinheiten melden abgeschlossene Berechnungen dem *Scoreboard*.

Dieses erteilt Befehlen die Berechtigung zum Abspeichern von Ergebnissen, sofern

– Speichereinheit frei ist und
– Antidaten- und Ausgabeabhängigkeiten berücksichtigt sind

und prüft, ob dadurch neue Befehle ausführbereit werden.

Scoreboarding (2)

Zentrale Datenstruktur hierfür: *Scoreboard* (deutsch etwa „Anzeigetafel“ [für Befehlsstatus])

Ursprünglich realisiert für CDC 6600 (1964):

– *load/store*-Architektur
– mehrere funktionale Einheiten (4xFP, 6xMem, 7xInteger ALU)

Scoreboarding für MIPS nur sinnvoll

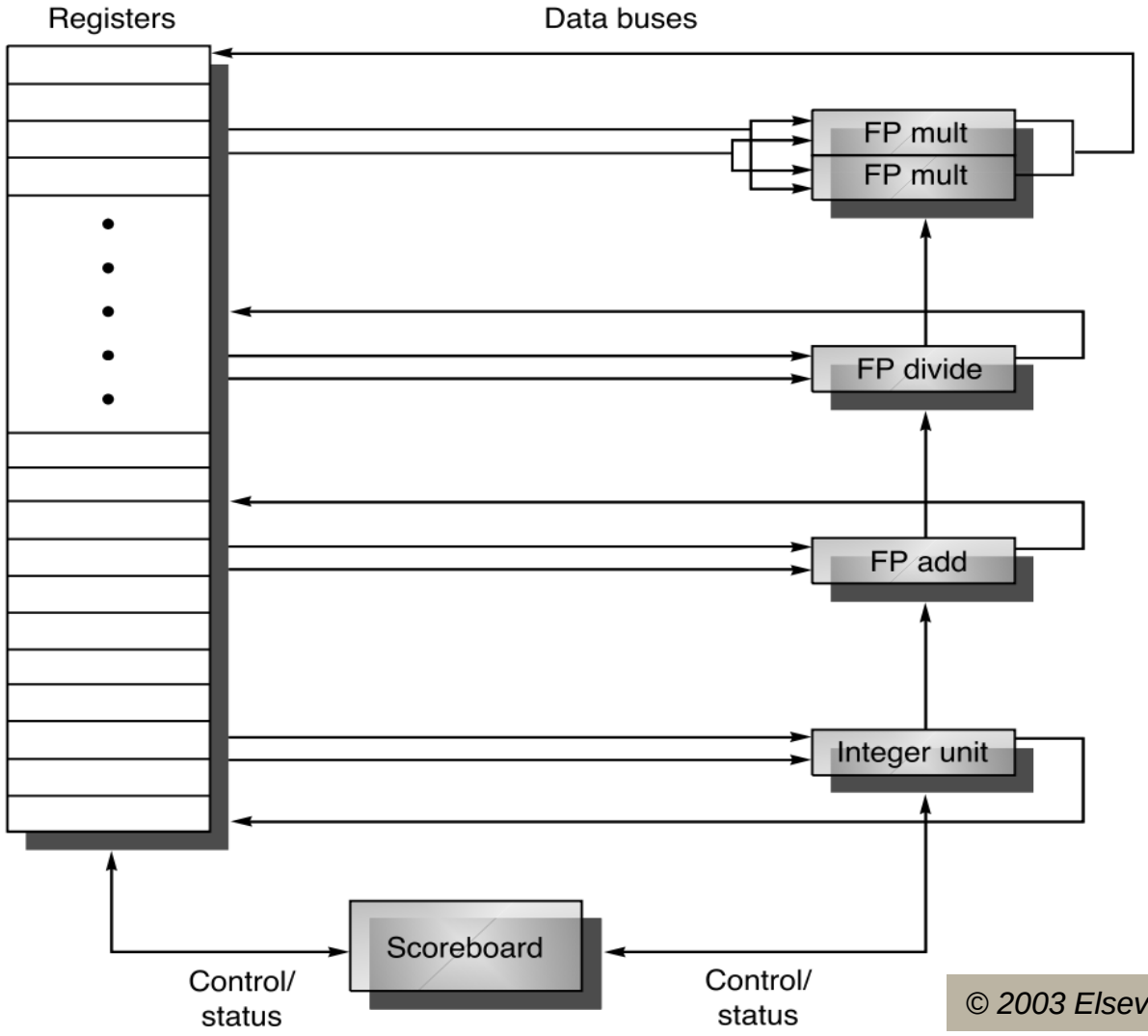
– für FP-*Pipeline* (Operationen mit mehreren Taktzyklen)
– und mehrere funktionale Einheiten
  (hier: 2 x Mult, Div, Add, Int)

Scoreboarding (3)

Struktur einer MIPS-*Pipeline* mit *Scoreboard*

Scoreboard kontrolliert funktionale Einheiten

Bussysteme erforderlich



Funktion des *Scoreboards* (1)

Scoreboard kontrolliert Befehlsausführung ab Befehlsholphase (IF) in 4 Phasen:

1. *Issue*
   - Befehl wird ausgegeben, falls
     - benötigte funktionale Einheit frei ist und
     - kein anderer aktiver Befehl (d.h. im *Scoreboard*) schreibt gleiches Zielregister ☞ verhindert WAW-Konflikte
   - Auswertung beginnt *nicht*, Befehl nur an funktionale Einheit geleitet!
     - Ersetzt Teil der ID-Phase in klassischer *Pipeline*
     - Anhalten der *Pipeline (stall)* erfolgt bei
       - strukturellen Konflikten oder
       - WAW-Konflikt
       - ☞ Kein weiterer Befehl wird ausgegeben!

Funktion des *Scoreboards* (2)

2. *Read Operands* (RO)
   - *Scoreboard* beobachtet Verfügbarkeit von Quelloperanden
     - Sind verfügbar, wenn kein früher ausgegebener Befehl, der noch aktiv ist, diese schreiben wird (Datenflussprinzip)
   - Sobald Operanden verfügbar: *Scoreboard* signalisiert betreffender funktionaler Einheit Beginn der Ausführungsphase
   - RAW-Konflikte (d.h. echte Datenabhängigkeiten) werden hier auf diese Weise dynamisch aufgelöst
   - Befehle werden ggf. *out-of-order* zur Ausführung gebracht (*Issue* erfolgt noch strikt *in-order*!)
   - *Scoreboard* verwendet kein *Forwarding* (Komplexität!)

     ☞ Operanden werden bei Verfügbarkeit aus Register gelesen

Funktion des *Scoreboards* (3)

3. *Execution*
   - Fkt. Einheit beginnt Befehlsausführung nach Erhalt der Operanden
   - Entspricht „klassischer“ EX-Phase, ggf. > 1 Taktzyklen lang
   - *Scoreboard* wird informiert, sobald Ergebnis vorliegt
     (noch in funktionaler Einheit, d.h. noch nicht im Zielregister!)
4. *Write Results* (WR)
   - *Scoreboard* erteilt Berechtigung zum Abspeichern, wenn Zielregister nicht von aktiver Operation (in RO) noch benötigt wird
     ☞ Lösung von WAR-Konflikten!

Abspeichern macht ggf. neue Befehle ausführbereit

WR und RO dürfen nicht überlappen!

Beispiel

```
div.d $F0,...          # längste Latenz
mul.d ...,$F0,$F8      # datenabhängig von div.d
add.d $F8,...          # antidatenabhängig von mul.d
                       # sonst unabhängig
```

Annahme für Latenzen: DIV: 40, MUL: 10, ADD: 2

Timing-Schema bei *Scoreboarding*:

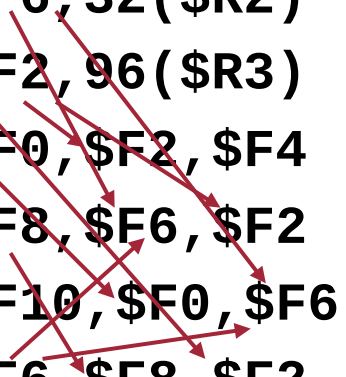
```
div.d     IF IS RO EX ..... EX WR
mul.d        IF IS (RO) ---------- RO EX .... EX WR
add.d           IF IS RO EX EX (WR)--- WR
```

- Alle Befehle *in-order* ausgegeben *(issue)*
- Reihenfolgeänderung bei Ausführung via *Read Operands*
- WAR-Konflikt durch *stall* in *Write Results* aufgelöst

Datenstrukturen

1. **Befehlsstatus:** Phase (*Issue*, …, *Write Results*), in der sich Befehl befindet
2. **Status der funktionalen Einheiten**, unterteilt in
   - *Busy* Einheit arbeitend oder nicht
   - *Op* Aktuelle Operation
   - F_i Zielregister
   - F_j, F_k Quellregister
   - Q_j, Q_k Funktionale Einheiten, die ggf. Quellregisterinhalt erzeugen
   - R_j, R_k Flag, ob Quelloperanden verfügbar, aber noch nicht in *Read Operands* gelesen
3. **Register-Ergebnis-Status:** Welche funktionale Einheit verfügbare Register schreibt (ggf. leer)

Codebeispiel (1)



```
l.d   $F6,32($R2)
l.d   $F2,96($R3)
mul.d $F0,$F2,$F4     # datenabhängig vom 2. l.d
sub.d $F8,$F6,$F2     # datenabhängig vom 1. & 2. l.d
div.d $F10,$F0,$F6    # datenabh. von mul.d und 1. l.d
add.d $F6,$F8,$F2     # datenabh. von sub.d und 2. l.d
                      # antidatenabh. v. sub.d & div.d
```

Betrachten Situation im *Scoreboard*:

– Beide *Loads* (**$F2**, **$F6**) fertig bearbeitet
– Weitere Befehle soweit möglich in Bearbeitung fortgeschritten
– Letzter Schritt: Ergebnis von 1. *Load* (**$F6**) gespeichert

Codebeispiel (2)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	
`mul.d`	`$F0,$F2,$F4`	+			
`sub.d`	`$F8,$F6,$F2`	+			
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`				

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	ja	`l.d`	F2	R3				nein	
Mult1	ja	`mul.d`	F0	F2	F4	Integer		nein	ja
Mult2	nein								
Add	ja	`sub.d`	F8	F6	F2		Integer	ja	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1	Integer			Add	Divide	

Codebeispiel (3)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+			
`sub.d`	`$F8,$F6,$F2`	+			
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`				

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein	–	–	–				–	
Mult1	ja	`mul.d`	F0	F2	F4	–		ja	ja
Mult2	nein								
Add	ja	`sub.d`	F8	F6	F2		–	ja	ja
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1	–			Add	Divide	

Codebeispiel (4)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+		
`sub.d`	`$F8,$F6,$F2`	+	+		
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`				

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			**nein**	**nein**
Mult2	nein								
Add	ja	`sub.d`	F8	F6	F2			**nein**	**nein**
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*…*
Einheit	Mult1				Add	Divide	

Codebeispiel (5)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	1	
`sub.d`	`$F8,$F6,$F2`	+	+	1	
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`				

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`sub.d`	F8	F6	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*…*
Einheit	Mult1				Add	Divide	

Codebeispiel (6)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	2	
`sub.d`	`$F8,$F6,$F2`	+	+	+	
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`				

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`sub.d`	F8	F6	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1				Add	Divide	

Codebeispiel (7)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	**3**	
`sub.d`	`$F8,$F6,$F2`	+	+	+	**+**
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`				

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	**nein**	–	–	–	–			–	–
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1				–	Divide	

Codebeispiel (8)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	**4**	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	**+**			

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	**ja**	**`add.d`**	**F6**	**F8**	**F2**			**ja**	**ja**
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1			**Add**		Divide	

Codebeispiel (9)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	**5**	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	**+**		

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			**nein**	**nein**
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*…*
Einheit	Mult1			Add		Divide	

Codebeispiel (10)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	6	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	+	1	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1			Add		Divide	

Codebeispiel (11)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	7	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	+	+	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1			Add		Divide	

Codebeispiel (12)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	8	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	+	+	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1			Add		Divide	

Codebeispiel (13)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	9	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	+	+	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1			Add		Divide	

Codebeispiel (14)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	+	+	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	ja	`mul.d`	F0	F2	F4			nein	nein
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	Mult1			Add		Divide	

Codebeispiel (15)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+			
`add.d`	`$F6,$F8,$F2`	+	+	+	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	**nein**	–	–	–	–			–	–
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6	–		**ja**	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit	–			Add		Divide	

Codebeispiel (16)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+	+		
`add.d`	`$F6,$F8,$F2`	+	+	+	

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	nein								
Mult2	nein								
Add	ja	`add.d`	F6	F8	F2			nein	nein
Divide	ja	`div.d`	F10	F0	F6			**nein**	**nein**

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit				Add		Divide	

Codebeispiel (17)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+	+	**1**	
`add.d`	`$F6,$F8,$F2`	+	+	+	**+**

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	nein								
Mult2	nein								
Add	**nein**	–	–	–	–			–	–
Divide	ja	`div.d`	F10	F0	F6			nein	nein

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit				–		Divide	

Codebeispiel (18)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+	+	**2**	
`add.d`	`$F6,$F8,$F2`	+	+	+	+

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	nein								
Mult2	nein								
Add	nein								
Divide	ja	`div.d`	F10	F0	F6			nein	nein

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*…*
Einheit						Divide	

Codebeispiel (19)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+	+	**3, …, 39**	
`add.d`	`$F6,$F8,$F2`	+	+	+	+

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	nein								
Mult2	nein								
Add	nein								
Divide	ja	`div.d`	F10	F0	F6			nein	nein

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit						Divide	

Codebeispiel (20)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+	+	+	
`add.d`	`$F6,$F8,$F2`	+	+	+	+

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	nein								
Mult2	nein								
Add	nein								
Divide	ja	`div.d`	F10	F0	F6			nein	nein

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*…*
Einheit						Divide	

Codebeispiel (21)

Befehls-Status

Befehl		*Issue*	*Read Op.*	*Exec. complete*	*Write Res.*
`l.d`	`$F6,32($R2)`	+	+	+	+
`l.d`	`$F2,96($R3)`	+	+	+	+
`mul.d`	`$F0,$F2,$F4`	+	+	+	+
`sub.d`	`$F8,$F6,$F2`	+	+	+	+
`div.d`	`$F10,$F0,$F6`	+	+	+	**+**
`add.d`	`$F6,$F8,$F2`	+	+	+	+

Status der Funktionseinheiten

Name	*Busy*	*Op*	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	nein								
Mult1	nein								
Mult2	nein								
Add	nein								
Divide	**nein**	**–**	**–**	**–**	**–**			**–**	**–**

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	…
Einheit						**–**	

Bewertung (1)

Leistungssteigerung von *Scoreboarding* (für CDC 6600)

- um Faktor 1,7 (FORTRAN-Programme) bzw.
- um Faktor 2,5 (handcodierter Assembler)
- Bewertung vor neueren RISC-Entwicklungen bzw. optimierten Speicherhierarchien!

Hardwareaufwand

- *Scoreboard* umfasst Logik vergleichbar einer der funktionalen Einheiten, d.h. relativ wenig
- Hauptaufwand durch Bussysteme zwischen funktionalen Einheiten und Registern, bzw. zum *Scoreboard*
  (4x höher gegenüber Prozessor mit sequentieller Dekodierung)

Bewertung (2)

Nützt verfügbare (!) Parallelität aus, um *stalls* aufgrund von echten Datenabhängigkeiten zu reduzieren.

Limitierungen entstehen durch

– Umfang der Parallelität auf Instruktionsebene *(instruction-level parallelism, ILP)*
– Anzahl der Einträge im *Scoreboard*
  ☞ Bestimmt, wie weit voraus die *Pipeline* nach unabhängigen Befehlen suchen kann
– Anzahl und Typ der funktionalen Einheiten
  Stall bei strukturellen *Hazards* hält alle weiteren Befehle an!

Zusammenfassung (1)

Klassifikation von Rechnern – die Befehlsschnittstelle

- Interne / Externe Rechnerarchitektur
- Adressierungsarten: Referenzstufen 0, ..., *n*
- *n*-Adressmaschinen
- Programmiermodelle: CISC, RISC, DSP, MMX, VLIW, NPU

Aufbau einer einfachen MIPS-Maschine

- Phasen der Befehlsabarbeitung:
  Instruction Fetch, Instruction Decode, Execute, Memory Access, Write Back
- Einzel- und Mehrzyklen CPUs

Zusammenfassung (2)

Fließbandverarbeitung / *Pipelining*

– Trennung der Phasen durch Pufferregister
– *Pipeline-Hazards*: Datenabhängigkeiten, Kontrollabhängigkeiten
– Bypässe, *pipeline stalls*, verzögerte Sprünge

Dynamisches *Scheduling*

– *Scoreboard* als zentrale Datenstruktur, die Abarbeitung von Befehlen und Status von funktionalen Einheiten überwacht
– *Pipeline*-Phasen mit *Scoreboarding*:
Instruction Issue, Read Operands, Execution, Write Results