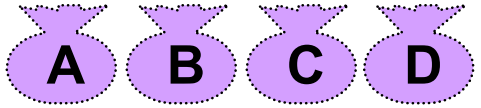
Inhaltsübersicht

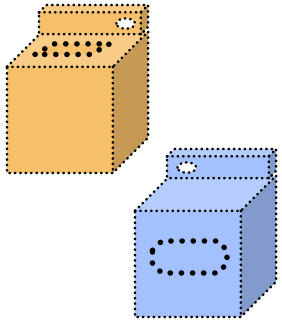
4. Pipelining

I. Einführung Pipelining
II. DLX-Pipeline
III. Pipeline-Hazards

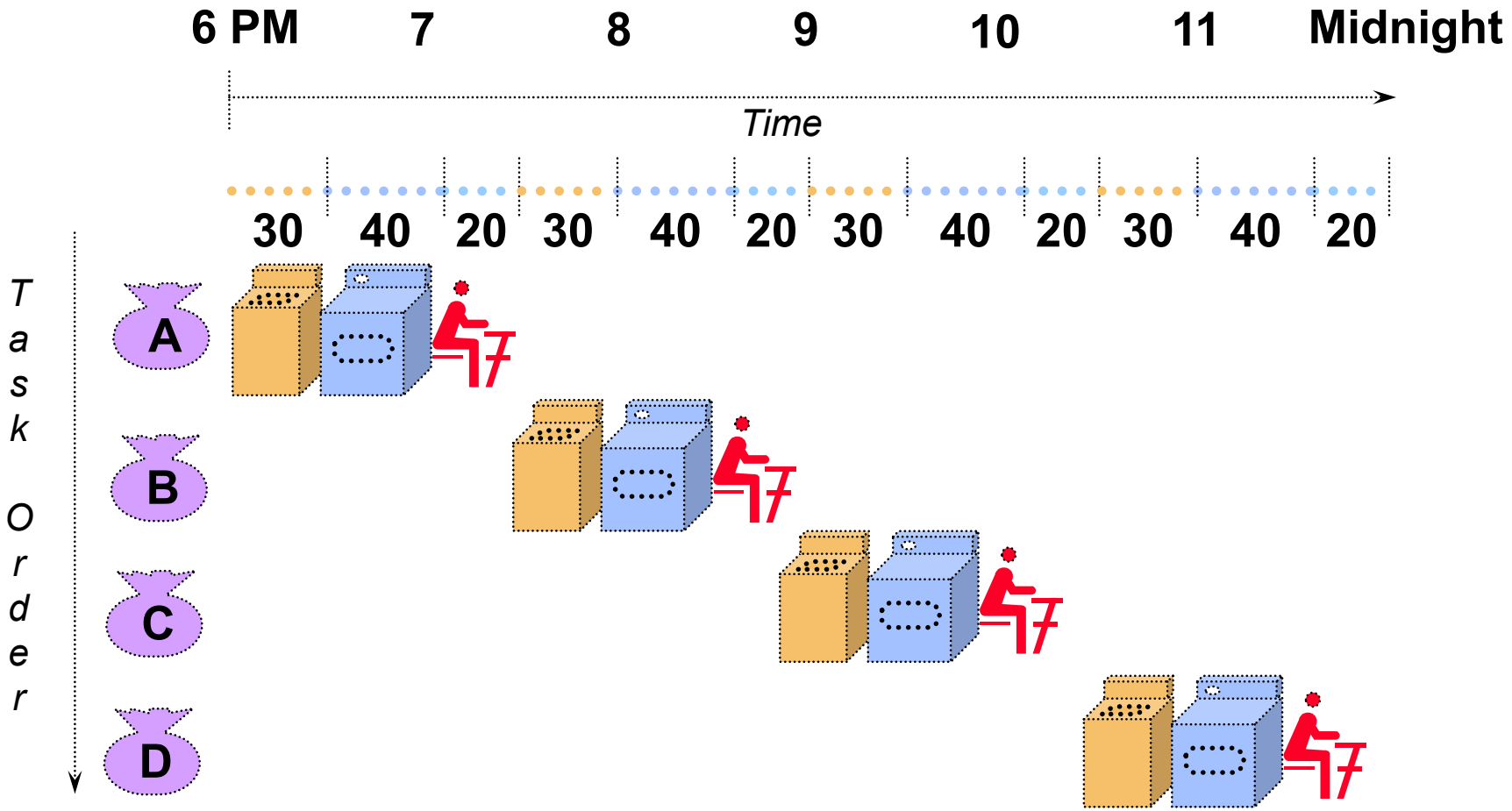
Pipelining – Eine nichttechnische Einführung

- Pipelining: Überlappung der Ausführung von mehreren Aufgaben.
- Beispiel Waschsalon:
  - Ann, Brian, Cathy, Dave: Jeder hat Wäsche zu waschen, trocknen und zusammenzulegen
  - Waschmaschine: 30 Min.
  - Trockner: 40 Min.
  - Falten: 20 Min.



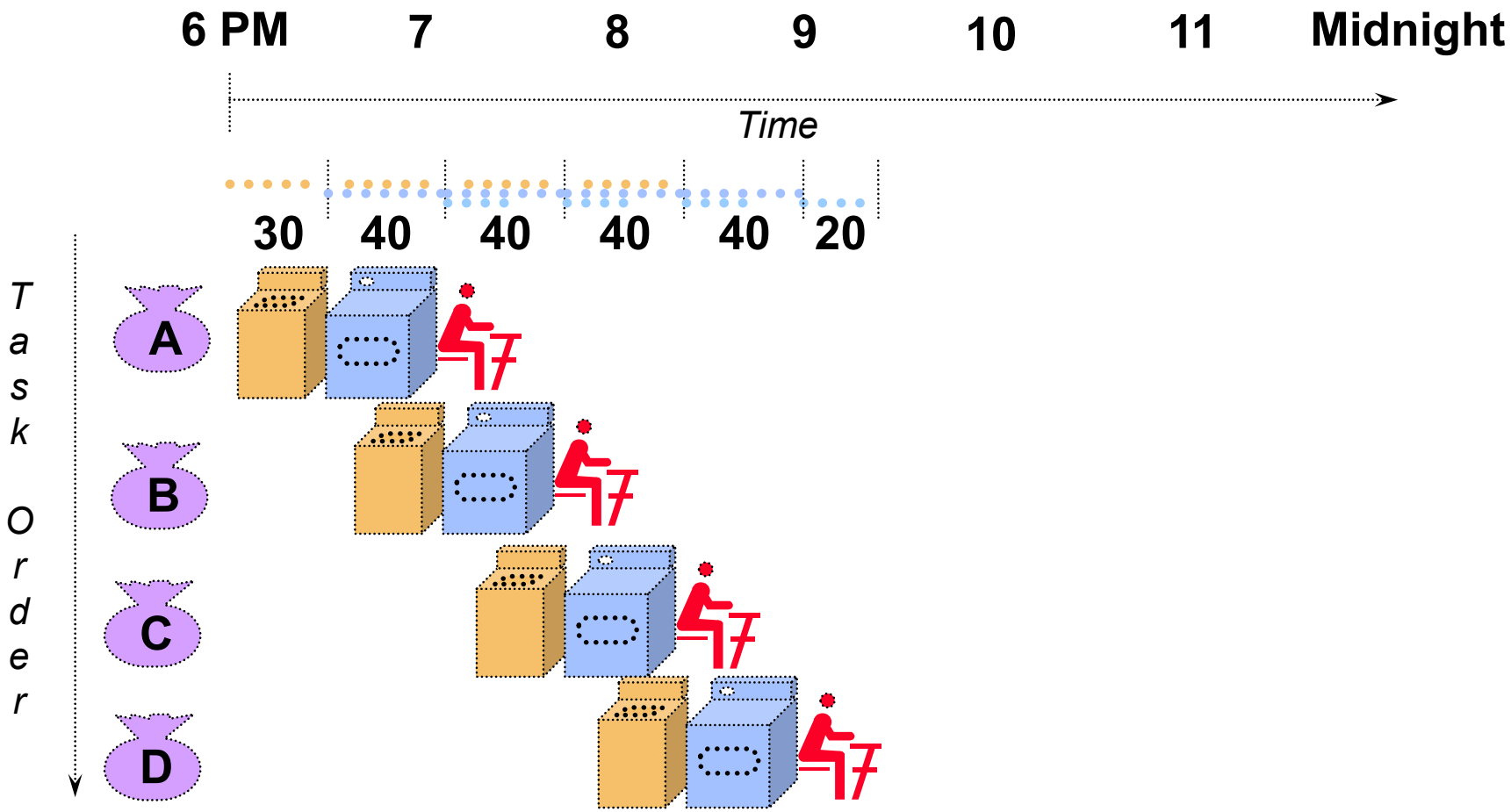


Beispiel : Sequentielle Wäsche



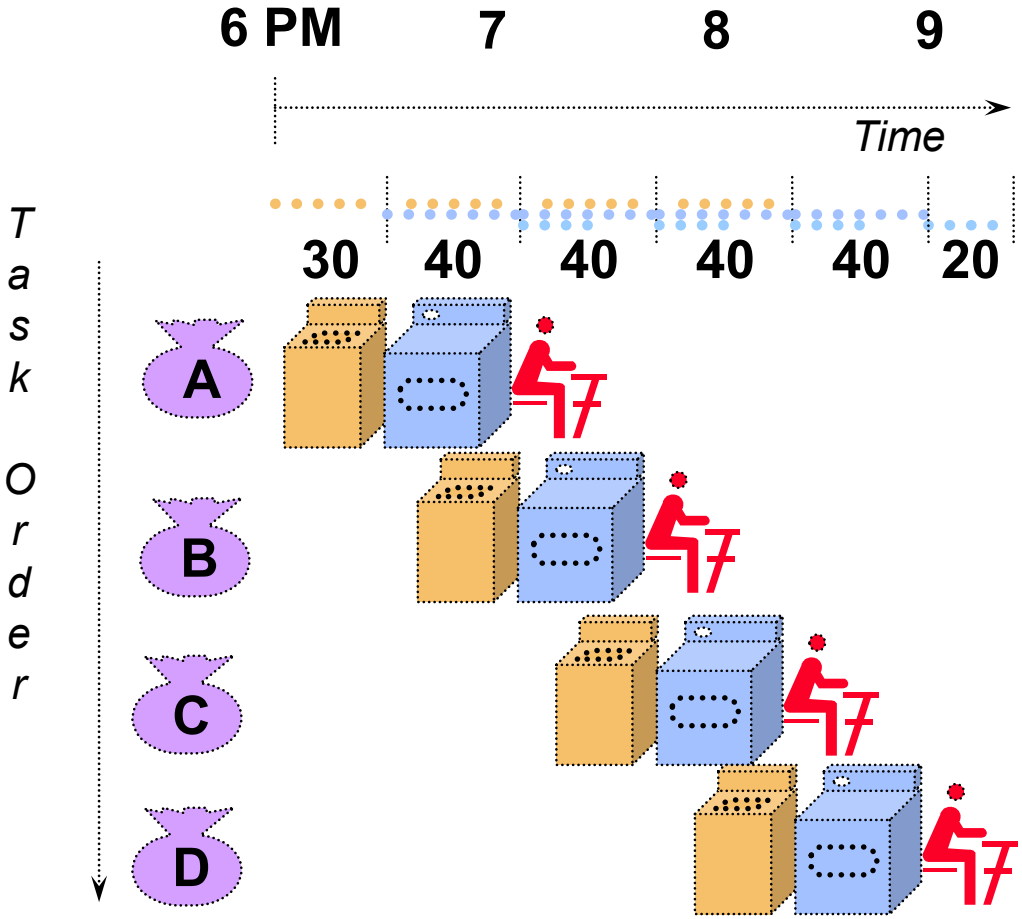
– Sequentielle Abarbeitung: 6 Stunden für 4 Vorgänge

Beispiel : Wäsche mit Pipelining



- Wäsche mit Pipelining: 3.5 Stunden für 4 Vorgänge
- Speedup = 6/3.5 = 1.7

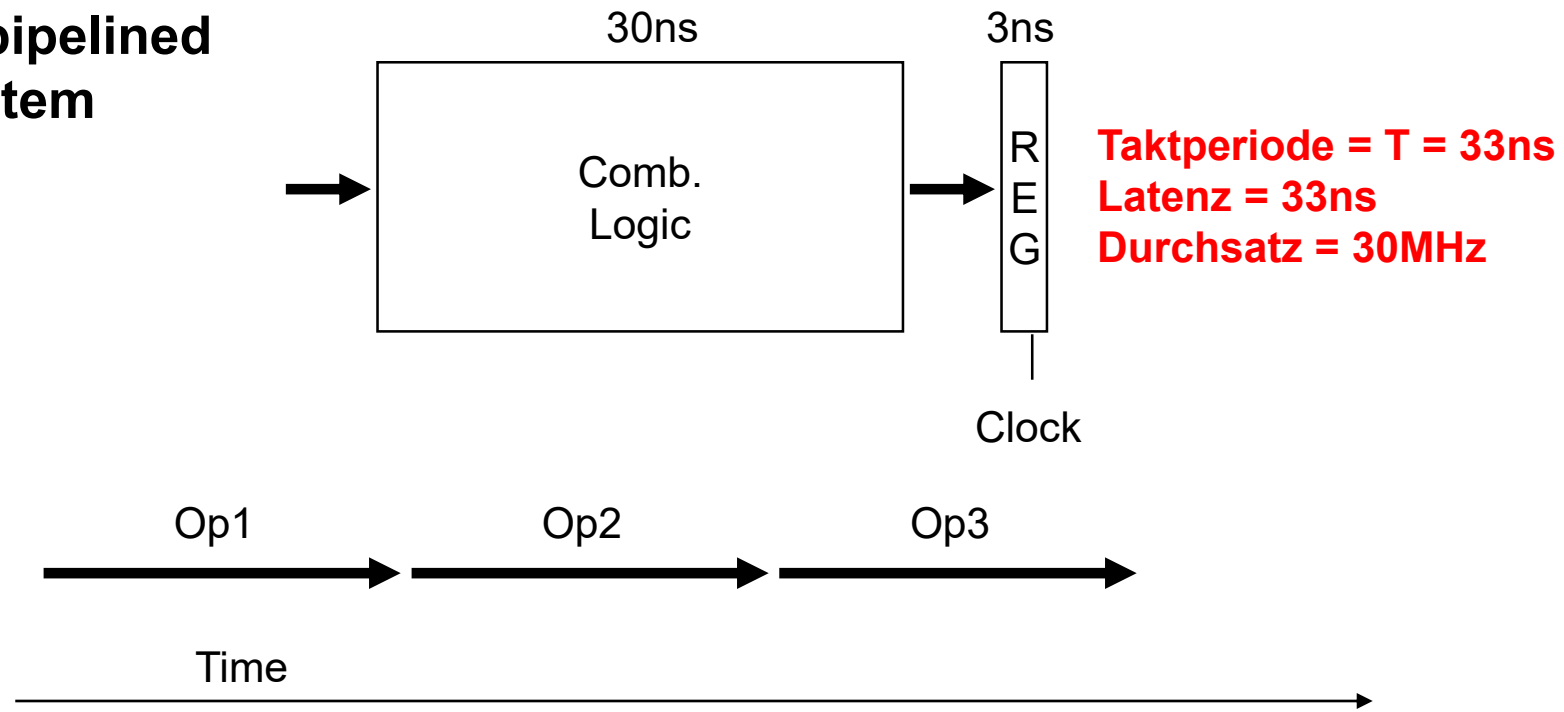
Was kann man von diesen Beispielen lernen?



- Pipelining verbessert nicht die Latenz (latency) eines Vorgangs, es verbessert den Durchsatz (throughput) der gesamten Arbeitslast
- Pipeline-Rate ist limitiert durch die langsamste Pipeline-Stufe
- Mehrere Stufen arbeiten gleichzeitig, d.h. Pipelining ist Parallelverarbeitung
- Potentieller Speedup = Anzahl der Pipeline-Stufen
- Unbalancierte Stufen verringern Speedup
- Die Zeit für das Füllen der Pipeline verringert ebenfalls den Speedup

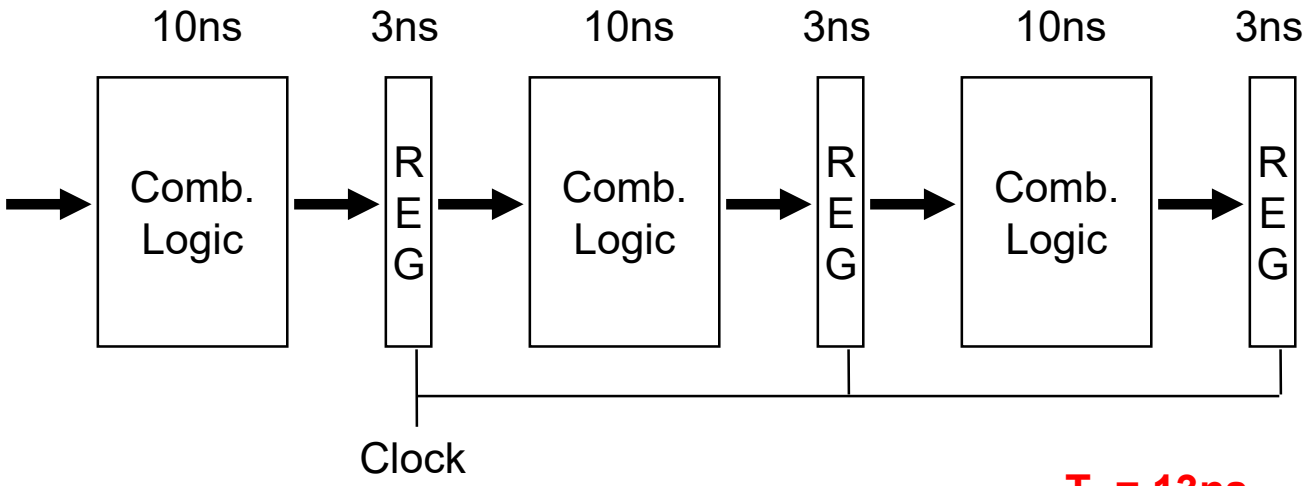
Pipelining bei digitalen Schaltungen

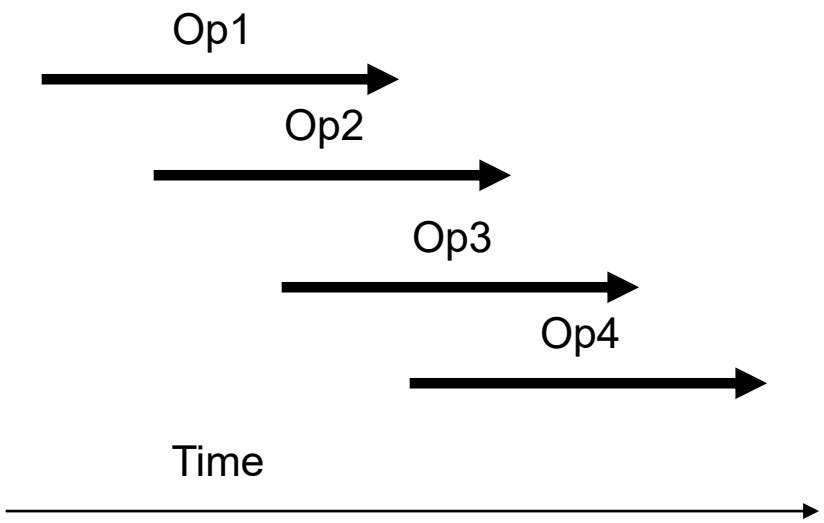
Unpipelined System



- Neue Werte können angelegt werden, wenn vorhergehende Operation zu Ende ist.
- Die zeitliche Distanz der Operationen ist daher 33 ns

3-stufige Pipeline

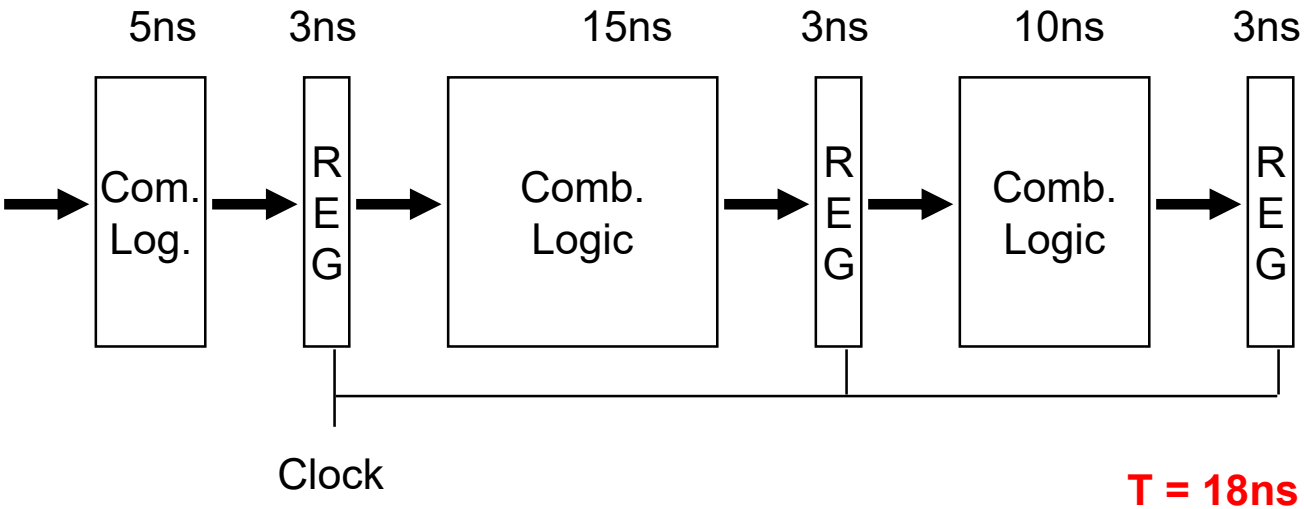




T = 13ns
Latenz = 39ns
Durchsatz = 77MHz

– 3 (kleinere) Operationen laufen nun gleichzeitig

Limitierung: Unterschiedlich schnelle Stufen



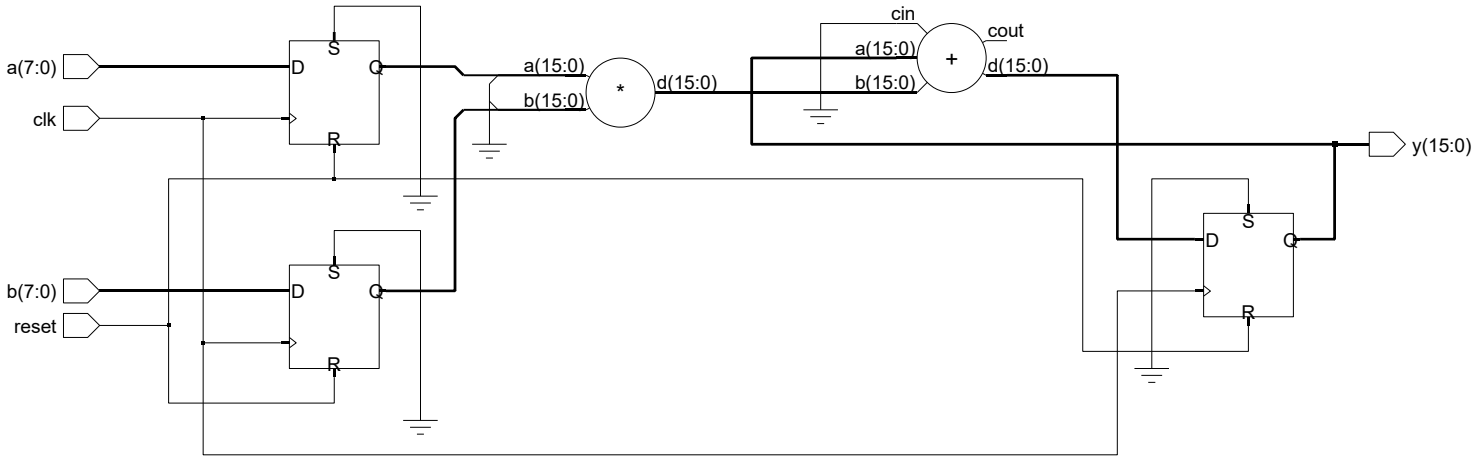
T = 18ns
Latenz = 18 * 3 = 54 ns
Durchsatz = 55MHz

- Durchsatz ist limitiert durch langsamste Stufe
  - Latenz ist bestimmt durch Taktperiode * Stufenzahl
- Balancierung der Stufen notwendig

Beispiel: MAC-Einheit in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY alu IS
   PORT(
      a    : IN   std_logic_vector(7 DOWNTO 0);
      b    : IN   std_logic_vector(7 DOWNTO 0);
      reset: IN         std_logic;
      clk  : IN         std_logic;
      y    : OUT  std_logic_vector(15 DOWNTO 0));
END alu ;


ARCHITECTURE beh OF alu IS
     SIGNAL a_i, b_i : unsigned(7 downto 0);
     SIGNAL accu: unsigned(15 downto 0);
BEGIN
     p0: PROCESS(clk, reset)
     BEGIN
            IF reset = '1' THEN
                        a_i <= (OTHERS=>'0');
                        b_i <= (OTHERS=>'0');
                        accu <= (OTHERS=>'0');
            ELSIF clk = '1' AND clk'event THEN
                        a_i <= unsigned(a);
                        b_i <= unsigned(b);
                        accu <= accu + (a_i*b_i);   --MAC: Multiply-Accumulate
            END IF;
     END PROCESS;
     y <= std_logic_vector(accu);
END beh;
```

Syntheseergebnis MAC-Einheit

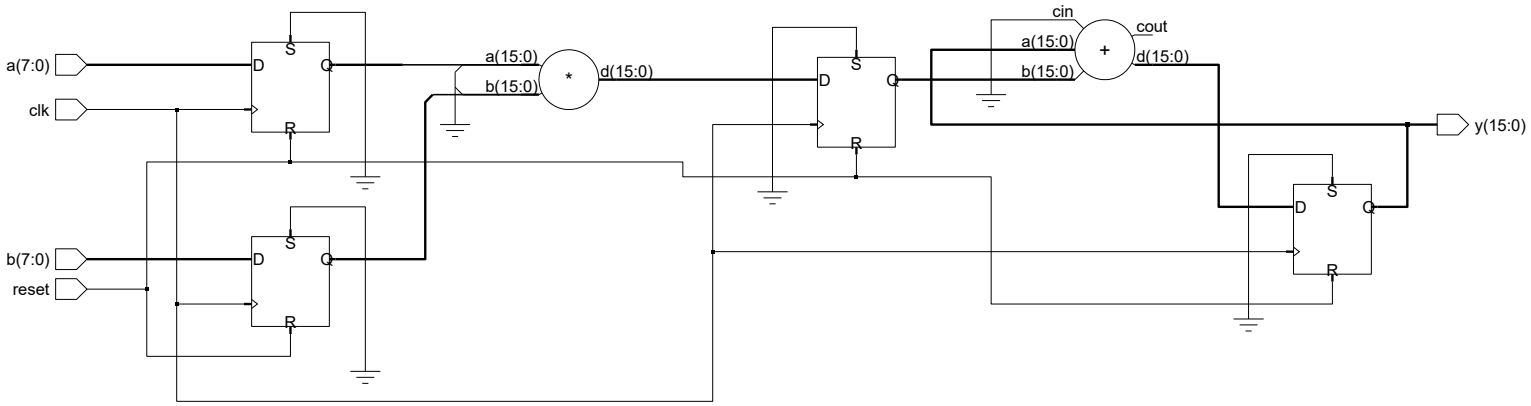


fmax = 118 MHz (Xilinx Spartan3 FPGA)

MAC-Einheit: Einführung von Pipelining

```
ARCHITECTURE beh OF alu IS
    SIGNAL a_i, b_i : unsigned(7 downto 0);
    SIGNAL accu, x_i : unsigned(15 downto 0);
BEGIN
    p0: PROCESS(clk, reset)
    BEGIN
          IF reset = '1' THEN
                a_i <= (OTHERS=>'0');
                b_i <= (OTHERS=>'0');
                x_i <= (OTHERS=>'0');
                accu <= (OTHERS=>'0');
          ELSIF clk = '1' AND clk'event THEN
                a_i <= unsigned(a);
                b_i <= unsigned(b);
                x_i <= a_i * b_i;
                accu <= accu + x_i;
          END IF;
    END PROCESS;
    y <= std_logic_vector(accu);
END beh;
```

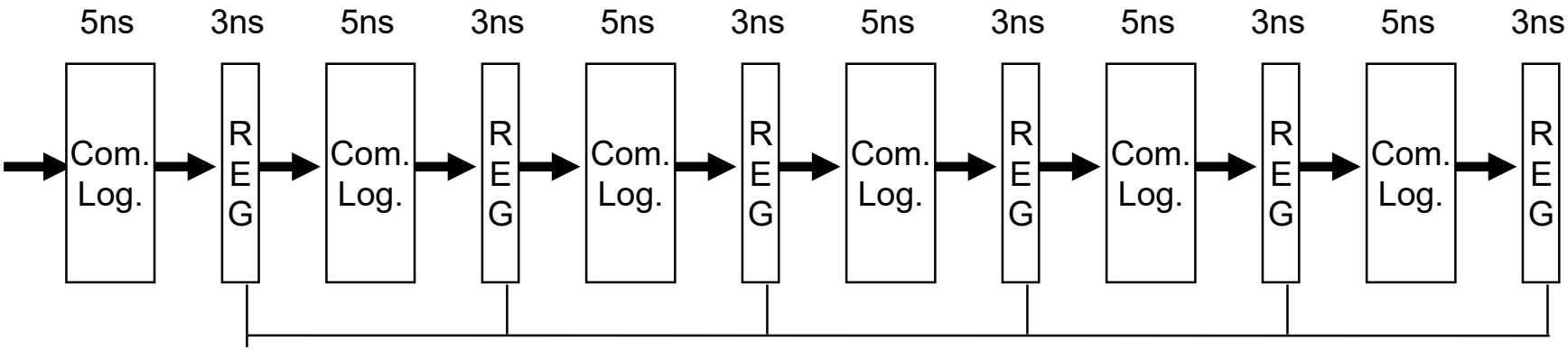
Syntheseergebnis MAC-Einheit



fmax = 161 MHz (Xilinx Spartan3 FPGA)

Unbalanciert:
Addierer: T = 6,2 ns
Multiplizierer: T = 5,2 ns (192,3 MHz)

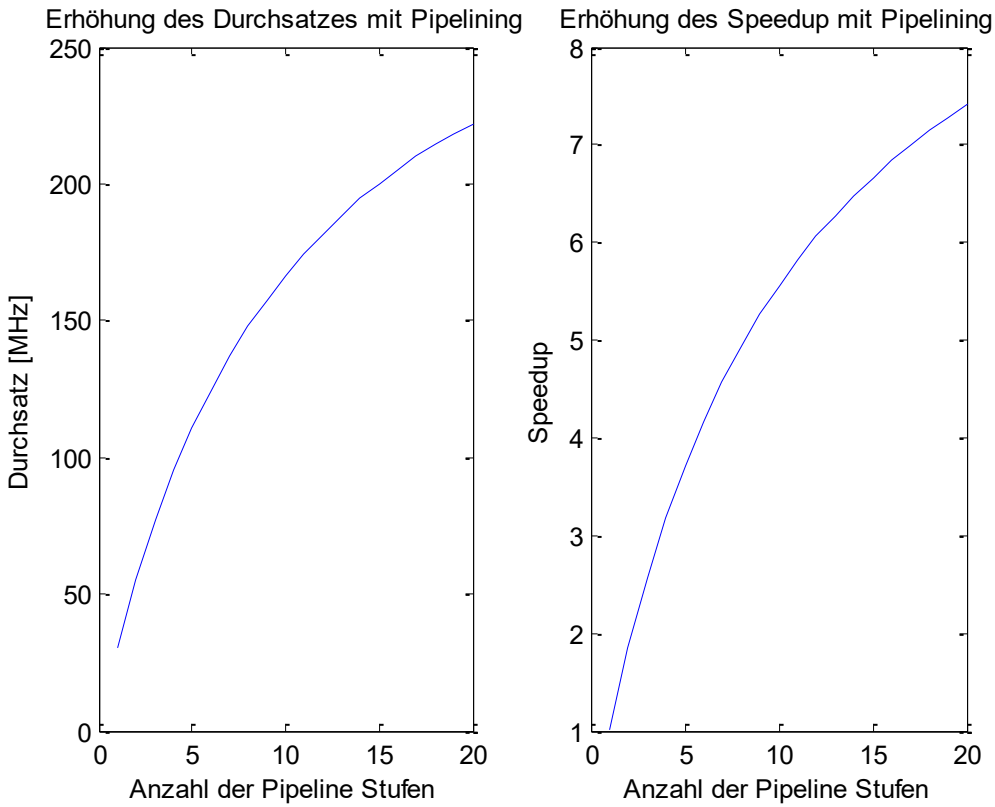
Tiefes Pipelining



T = 8ns
Latenz = 48ns
Durchsatz = 125MHz

- Pipelining-Gewinne werden mit steigender Stufenzahl immer geringer
- Limitierender Faktor ist die Verzögerungszeit T_r der Register

Pipelining Speedup



$$Durchsatz = \frac{1}{\frac{T}{x} + T_r} = \frac{1}{\frac{30ns}{x} + 3ns}$$

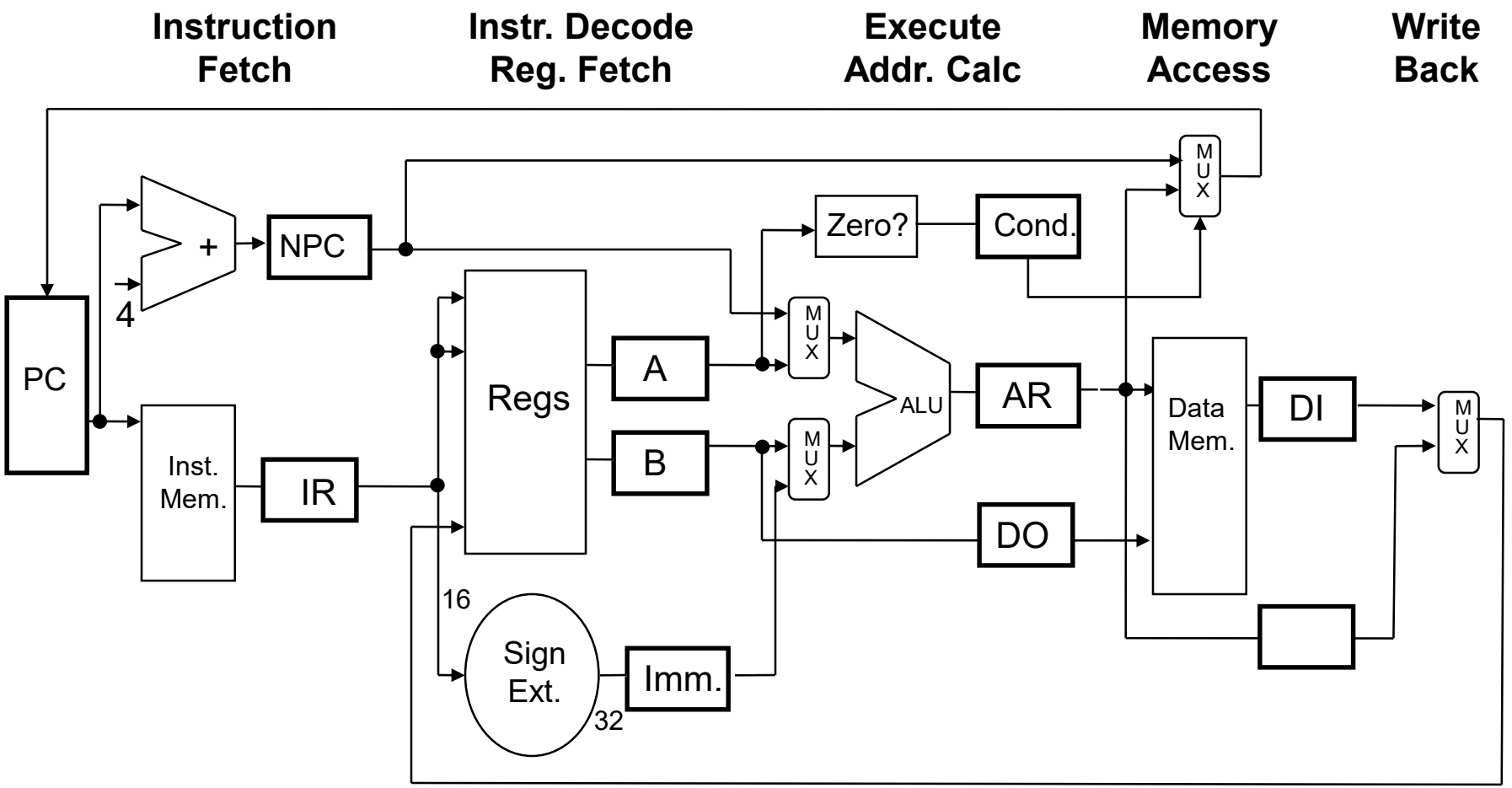
4. Pipelining

I. Einführung Pipelining

II. DLX-Pipeline

III. Pipeline-Hazards

Die 5-stufige DLX-Pipeline



DLX Stufen: Stufen 1 und 2

1. Instruction Fetch Stufe (IF)

- IR <= Mem[PC] (Instruktion laden)
- NPC <= PC+4 (Neuer Wert Program Counter)

2. Instruction Decode / register fetch Stufe (ID)

- A <= Regs[$IR_{6...10}$] (Quellregister 1)
- B <= Regs[$IR_{11...15}$] (Quellregister 2)
- Imm <= $((IR_{16})^{16}$ ## $IR_{16..31})$ (Erw. Immediate 32 Bit)

DLX Stufen: Stufe 3

3. Execute / effective address Stufe (EX):

- *Speicherzugriff* (z.B., LW R1, 30 (R2))
  AR <= A + Imm (Effektive Adresse)
- *Register-Register ALU op.* (z.B., ADD R1, R2, R3)
  AR <= A func B (ALU Operation)
- *Register-Immed. ALU op.* (z.B., ADDI R1, R2, #3)
  AR <= A func Imm (ALU Operation)
- *Branch* (z.B., BEQZ R4, next)
  AR <= NPC + Imm (Sprungziel)
  Cond <= (A op 0) (Vergleich Reg. A auf 0)

DLX Stufen: Stufe 4

4. Memory Access / branch completion (MEM):

- *Loads* (z.B., LW R1, 30 (R2))

  DI <= Mem[AR] (Speicherinhalt in DI laden)

- *Stores* (z.B., SW 500(R4), R3)

  Mem[AR] <= B (Inhalt B in Speicher laden)

- *Branch* (z.B., BEQZ R4, next)

  if (cond) PC <= AR

  else PC <= NPC (Bedingter Sprung)

DLX Stufen: Stufe 5

5. Write-back cycle (WB):

- *Register-Register ALU op.* (z.B., ADD R1, R2, R3)

  $Regs[IR_{16....20}] <= AR$

- *Register-Immed. ALU op* (z.B., ADD R1, R2, #3)

  $Regs[IR_{11...15}] <= AR$

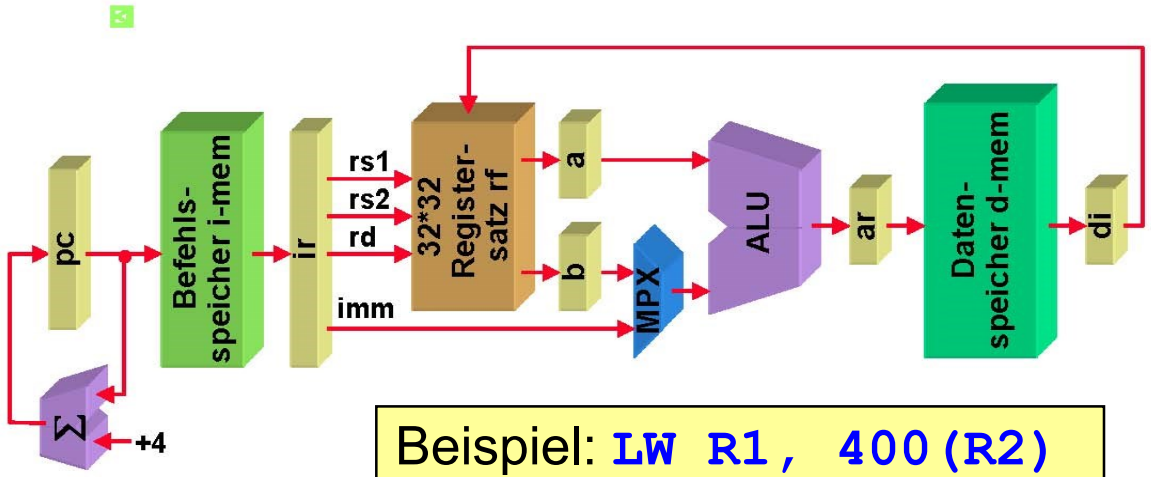
- *Loads* (z.B., LW R1, 30 (R2))

  $Regs[IR_{11...15}] <= DI$

Abarbeitung von Befehlen: Load

I-Format

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

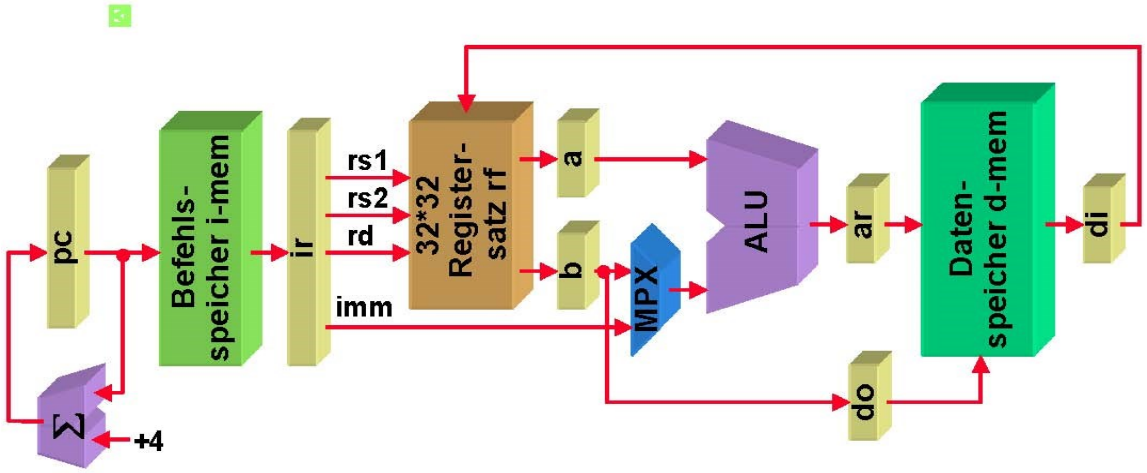


Beispiel: `LW R1, 400(R2)`
Wirkung: R1:=MEM(R2+400)

Abarbeitung von Befehlen: Store

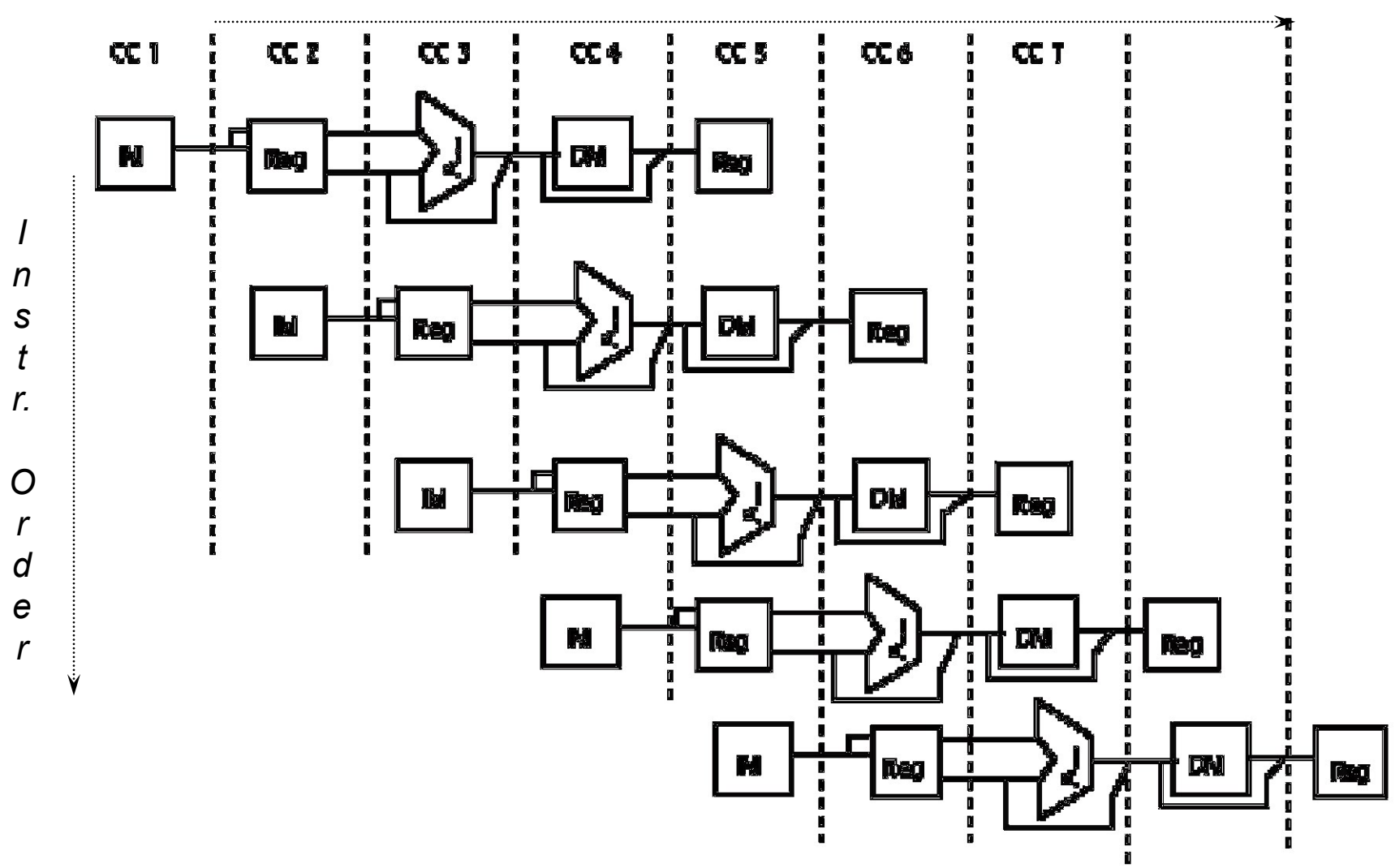
I-Format

	IF:	ID:	EX:	MEM:	
SW	ir<=i-mem(pc), pc<=pc+4	a<=rf(ir[6:10]), b<=rf(ir[11:15])	ar<=a+ir[16:31], do<=b	d-mem(ar)<= do	



Beispiel: `SW 400(R2), R1`
Wirkung: MEM(R2+400):=R1

Zeitlicher Ablauf der Befehlsabarbeitung



Effizienz des Pipelinings

- Um Pipelining wirkungsvoll anwenden zu können, müssen eine Reihe von Voraussetzungen erfüllt sein, z.B.:
  - Die Anzahl der nacheinander zu verarbeitenden Schritte muß hoch sein, da das Füllen der Pipeline aufwendig ist
  - Die Pipeline muß füllbar sein
    - Beispiel Speicherzugriffszeit : Um in jedem Takt eine Instruktion neu beginnen zu können, sind Daten-und Instruktions-Caches (Harvard-Architektur) mit Zugriffszeit = Prozessortaktzeit notwendig. Das heißt, daß es unsinnig ist, einen Pipeline-Prozessor ohne Caches zu bauen (siehe Kapitel 5).

4. Pipelining

I. Einführung Pipelining
II. DLX-Pipeline
III. Pipeline-Hazards

Wesentliches Pipelining Problem: Pipeline-Konflikte

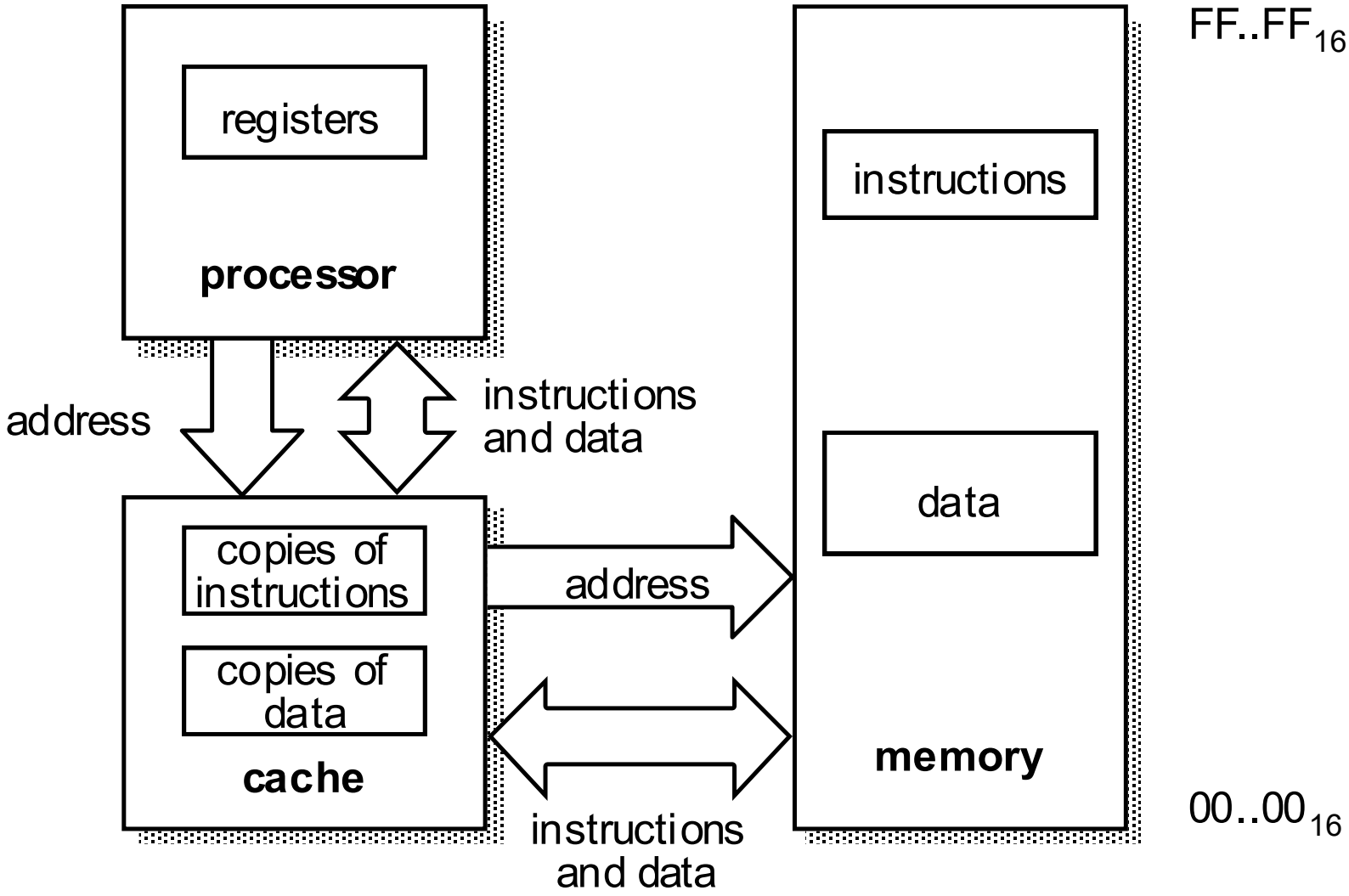
- Pipeline-Konflikte oder Pipeline-Hazards verhindern, dass die nächste Instruktion im dafür vorgesehenen Zyklus ausgeführt wird:
  - Struktur-Hazards: Ressourcenkonflikt
  - Daten-Hazards: Datenabhängigkeiten von Instruktionen
  - Steuerfluss-Hazards: Verändern des linearen, sequentiellen Programmablauf durch Sprünge

- Einfachste Lösung: Anhalten der Pipeline ("pipeline stall") bis der Konflikt aufgelöst ist.

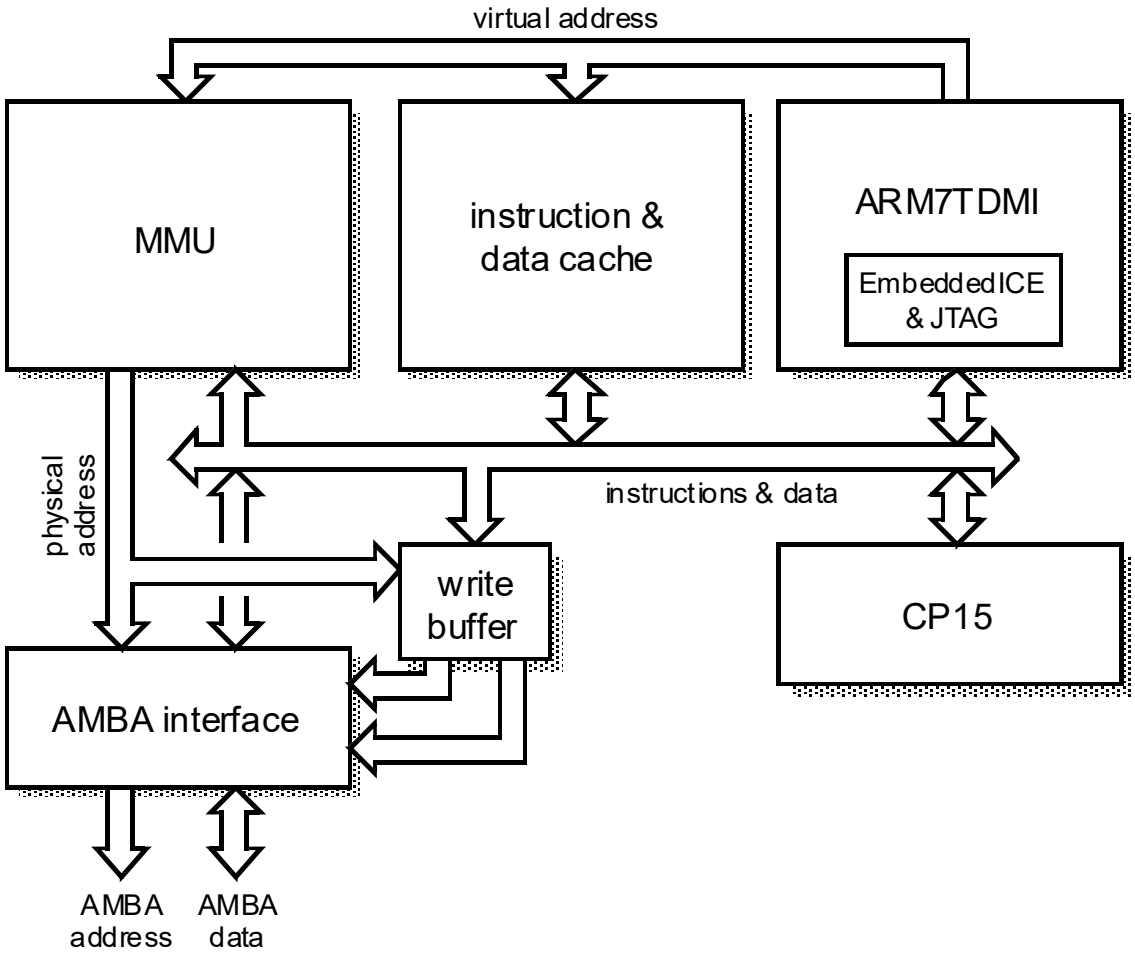
Struktur-Hazards

- Wenn eine Kombination von Instruktionen nicht ausgeführt werden kann aufgrund von Ressourcenkonflikten, dann wird dies als Struktur-Hazard bezeichnet.
- Die Pipeline wird dann angehalten, bis die fragliche Ressource verfügbar ist.
- Beispiel: Gemeinsamer Speicher für Programm und Daten (von Neumann-Architektur) führt zu Ressourcenkonflikt

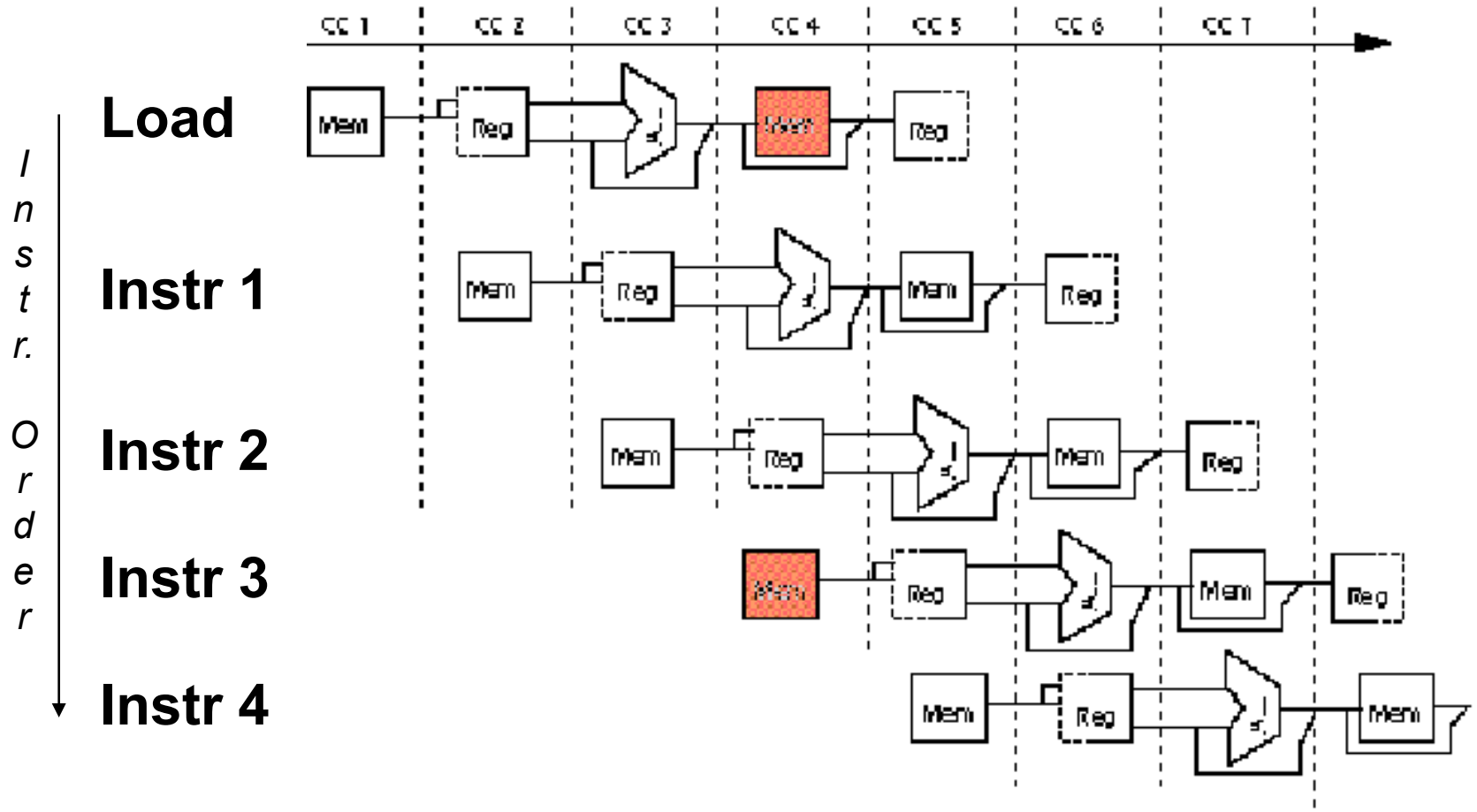
Beispiel: „Unified Cache“ (von Neumann)



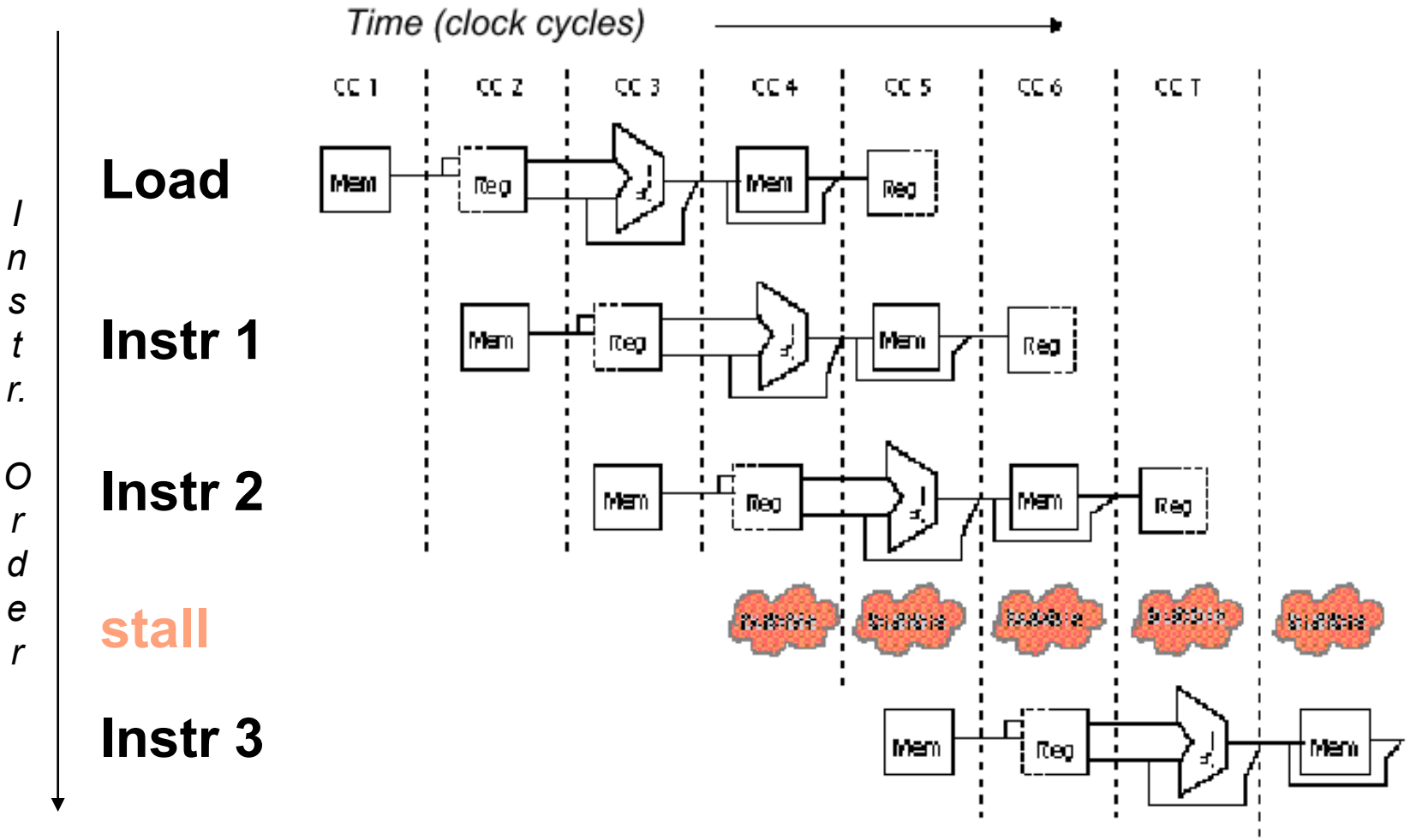
Beispiel: ARM7



Beispiel Struktur-Hazard (1)



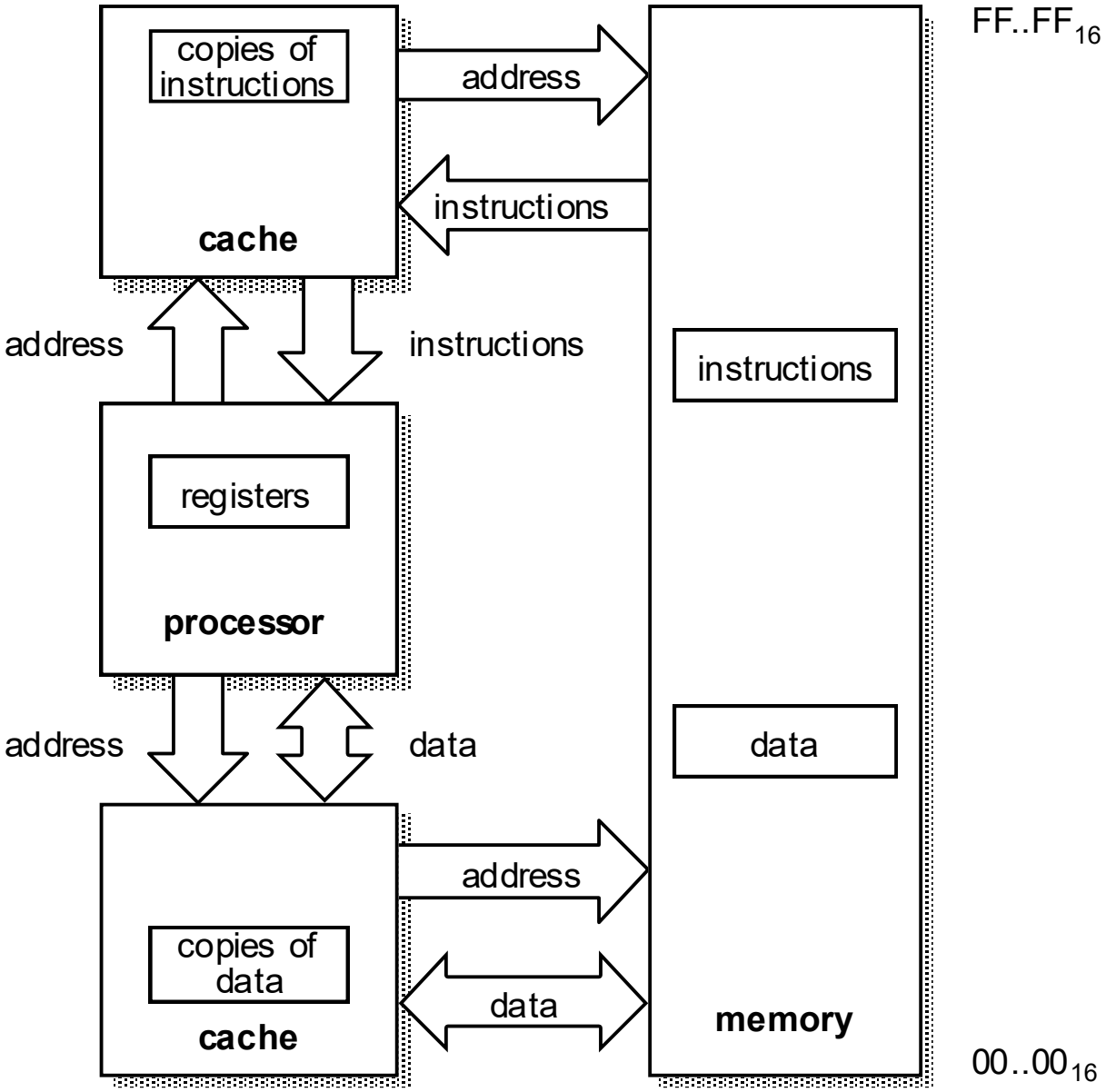
Beispiel Struktur-Hazard (2)



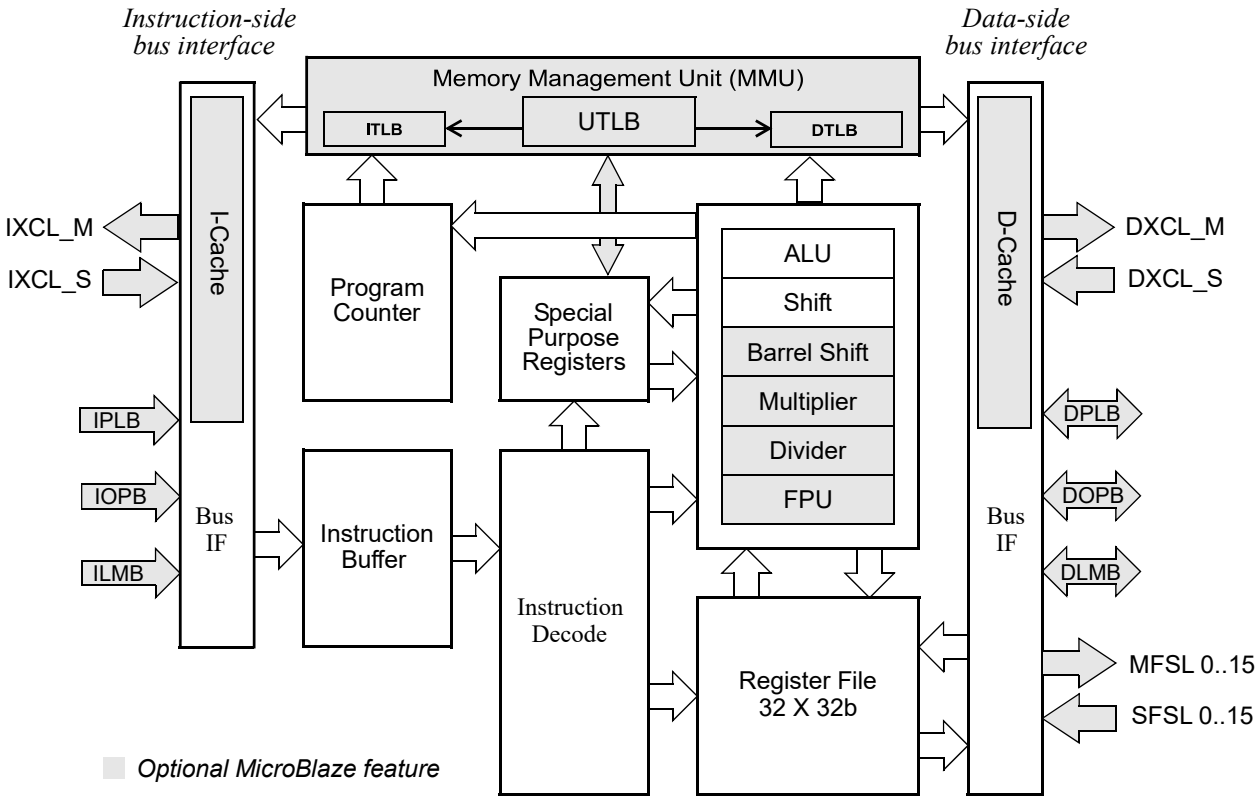
Beispiel Struktur-Hazard (3)

Inst.	1	2	3	4	5	6	7	8	9	10
Load Inst	IF	ID	EX	MEM	WB					
Intst i+1		IF	ID	EX	MEM	WB				
Intst i+2			IF	ID	EX	MEM	WB			
Intst i+3				STALL	IF	ID	EX	MEM	WB	
Intst i+4						IF	ID	EX	MEM	WB
Intst i+5							IF	ID	EX	MEM
Intst i+6								IF	ID	EX

Lösung durch „Harvard“-Architektur



Beispiel: Xilinx MicroBlaze

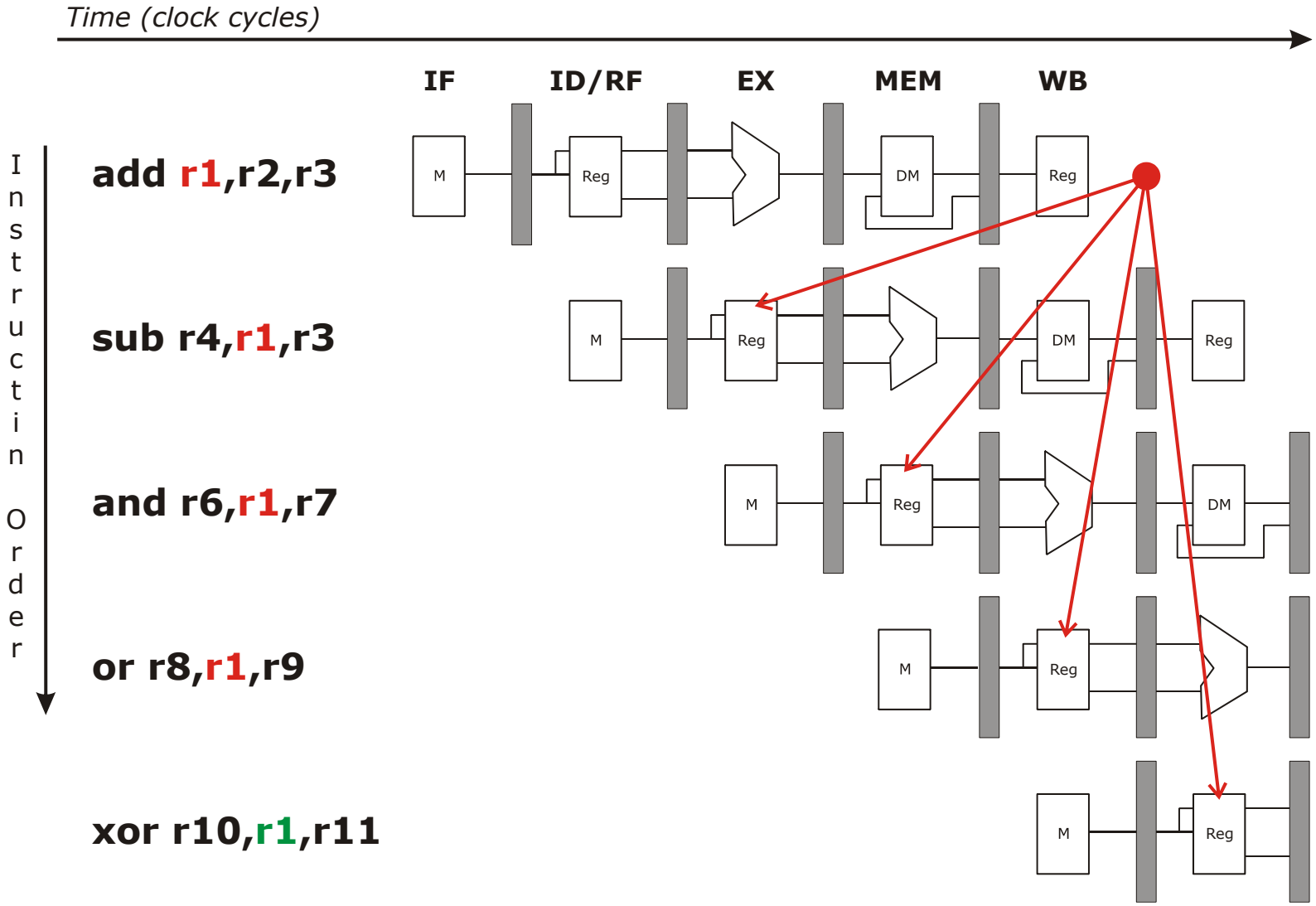


MicroBlaze Core Block Diagram

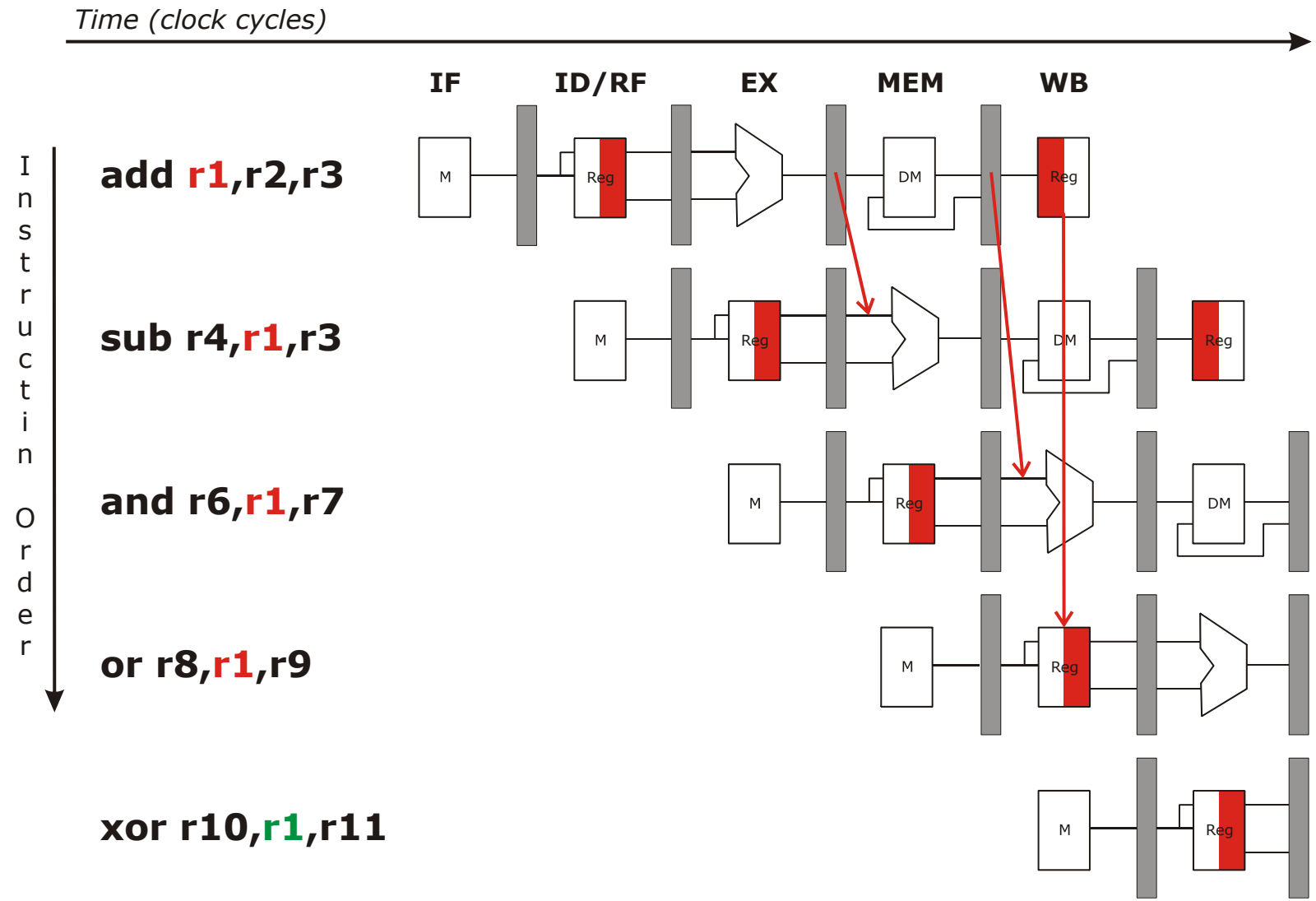
Daten-Hazards

- Bei einem Daten-Hazard verändert die Abarbeitung in der Pipeline die Reihenfolge der Schreib-/Lesezugriffe auf die Operanden im Vergleich zur Reihenfolge im Quellcode oder im Vergleich zur sequentiellen Abarbeitung in einer Maschine ohne Pipelining.

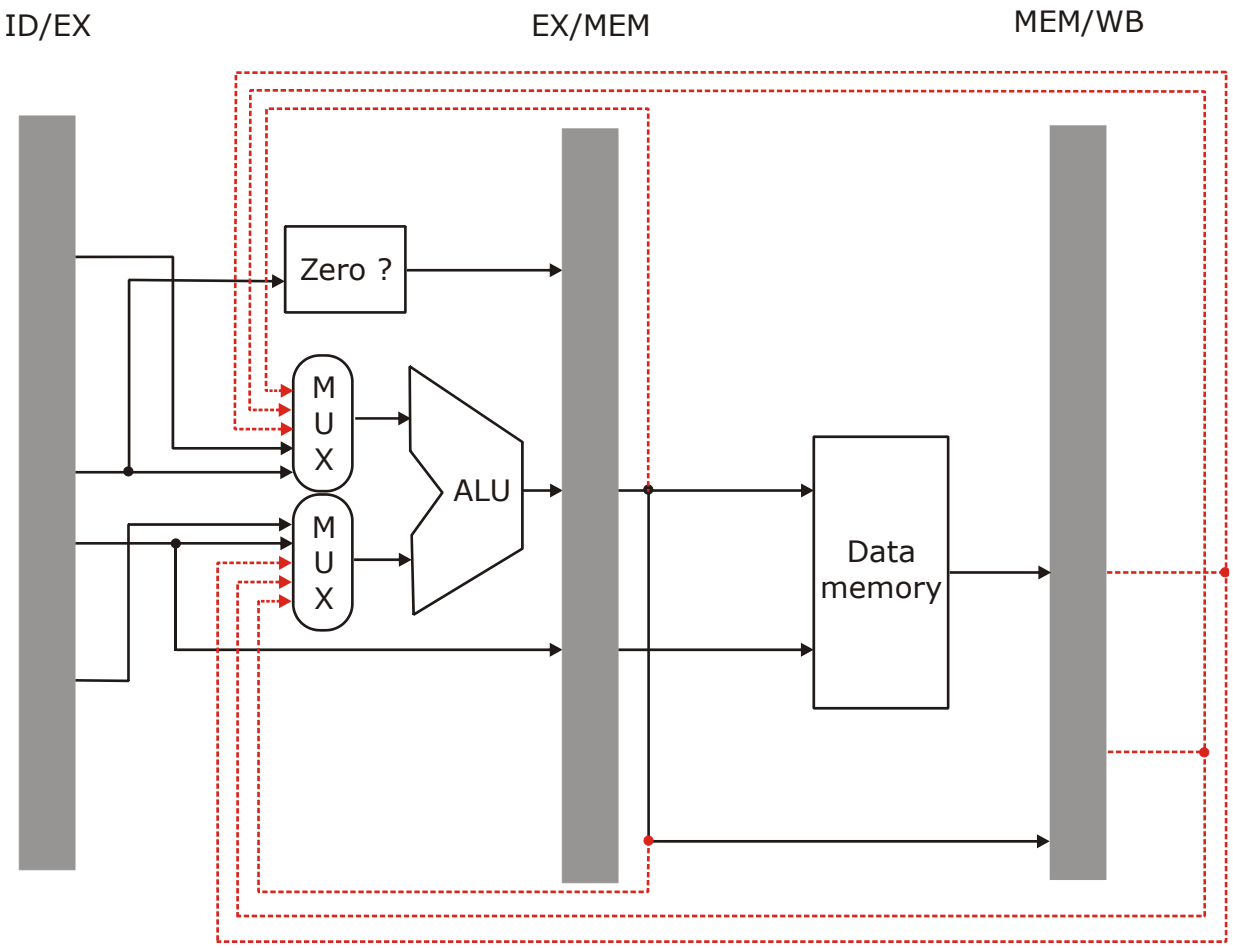
Beispiel Daten-Hazard



Lösung: Forwarding



Hardware Änderungen für Forwarding



Klassifikation von Daten-Hazards (1)

- Annahme: **$Instr_I$** vor **$Instr_J$** im Quellcode

- Read After Write (RAW)
  $Instr_J$ liest Operand bevor **$Instr_I$** diesen schreibt:

```
I:   ADD R1, R2, R3     IF  ID  EX  MEM WB
J:   SUB R4, R1, R5         IF  ID  EX  MEM  WB
```

Klassifikation von Daten-Hazards (2)

- Write After Write (WAW):
  $Instr_J$ schreibt Operand vor **$Instr_I$** :

I:	**LW R1, 0(R2)**	**IF**	**ID**	**EX**	**MEM1**	**MEM2**	**WB**
J:	**ADD R1, R2, R3**		**IF**	**ID**	**EX**	**WB**	

- Kann in DLX nicht passieren:
  - Alle Instruktionen benötigen 5 Stufen und
  - WB ist immer die Stufe 5

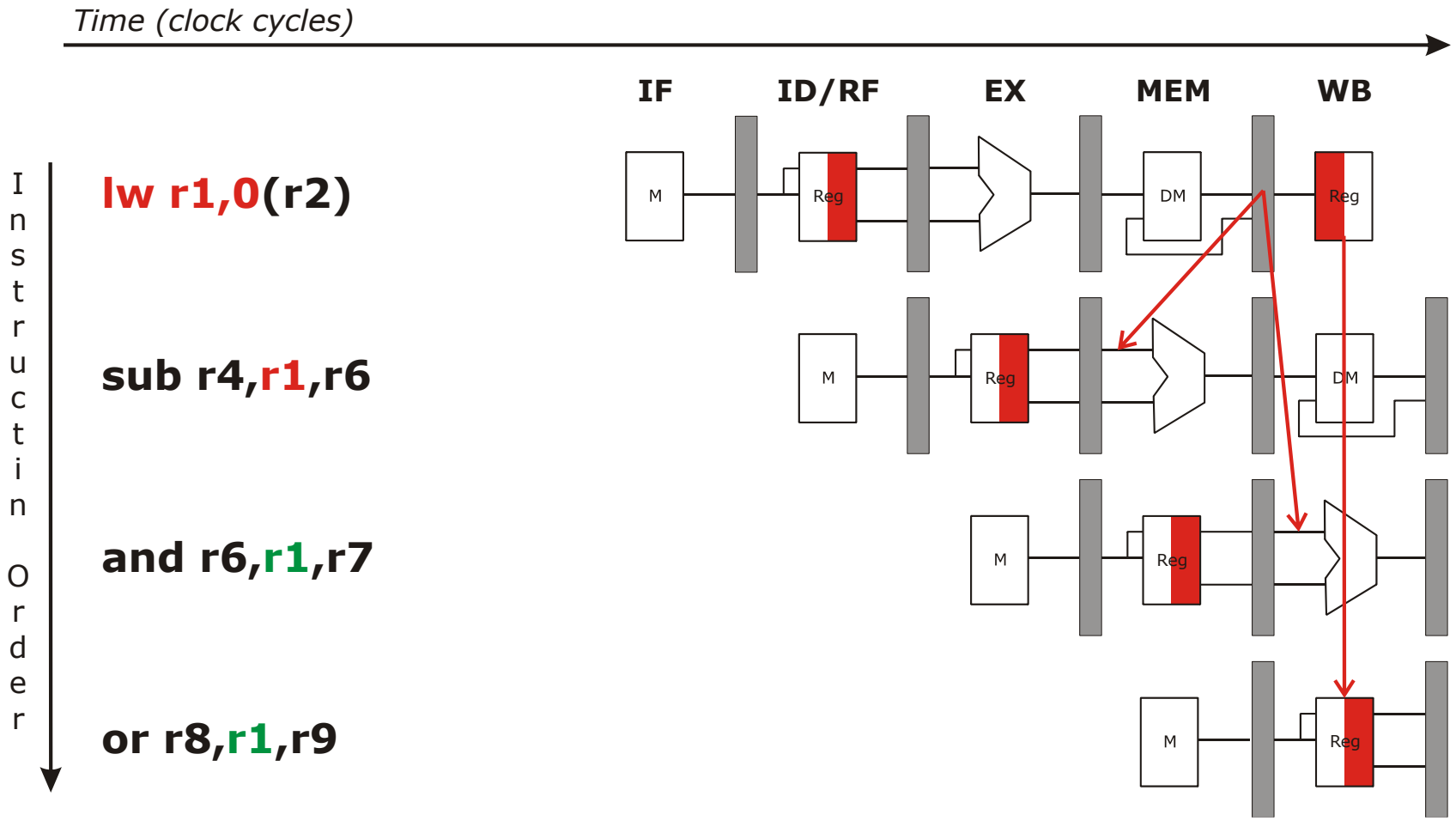
Klassifikation von Daten-Hazards (3)

- Write After Read (WAR):
  $Instr_J$ schreibt Operand bevor **$Instr_I$** diesen liest:

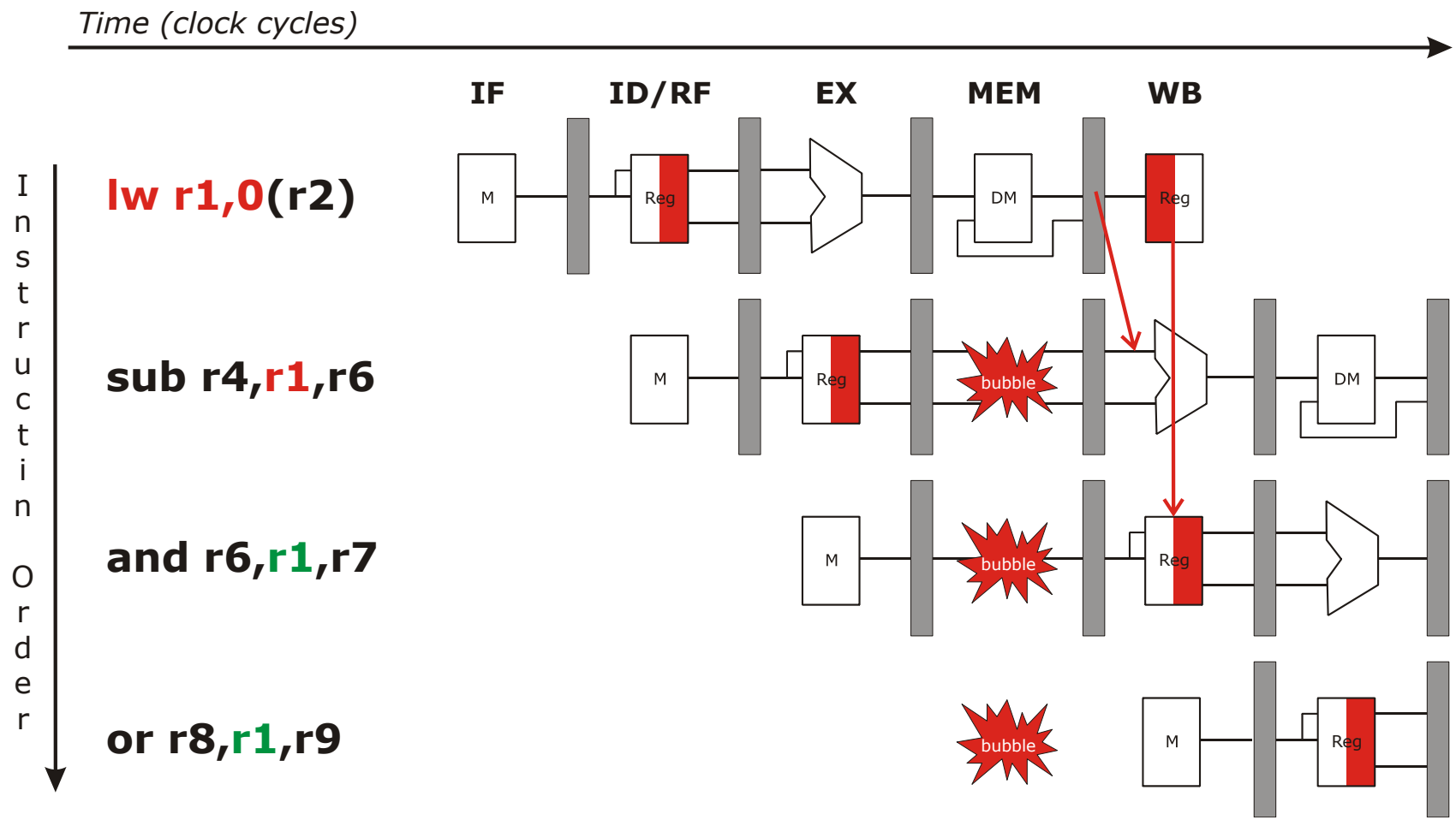
I:	**SW 0(R1), R2**	**IF**	**ID**	**EX**	**MEM1**	**MEM2**	**WB**
J:	**ADD R2, R3, R4**		**IF**	**ID**	**EX**	**WB**	

- Kann in DLX nicht passieren:
  - Alle Instruktionen benötigen 5 Stufen und
  - Lesen immer in Stufe 2 und
  - WB ist immer die Stufe 5

Nicht-auflösbare Hazards



Automatischer Stall durch „Pipeline Interlock“



Compiler Scheduling für Daten-Hazards

- Statt die Pipeline anzuhalten, kann der Compiler versuchen die Hazards durch Umändern der Codesequenz zu vermeiden. Dies wird als „Pipeline Scheduling“ oder „Instruction scheduling“ bezeichnet.

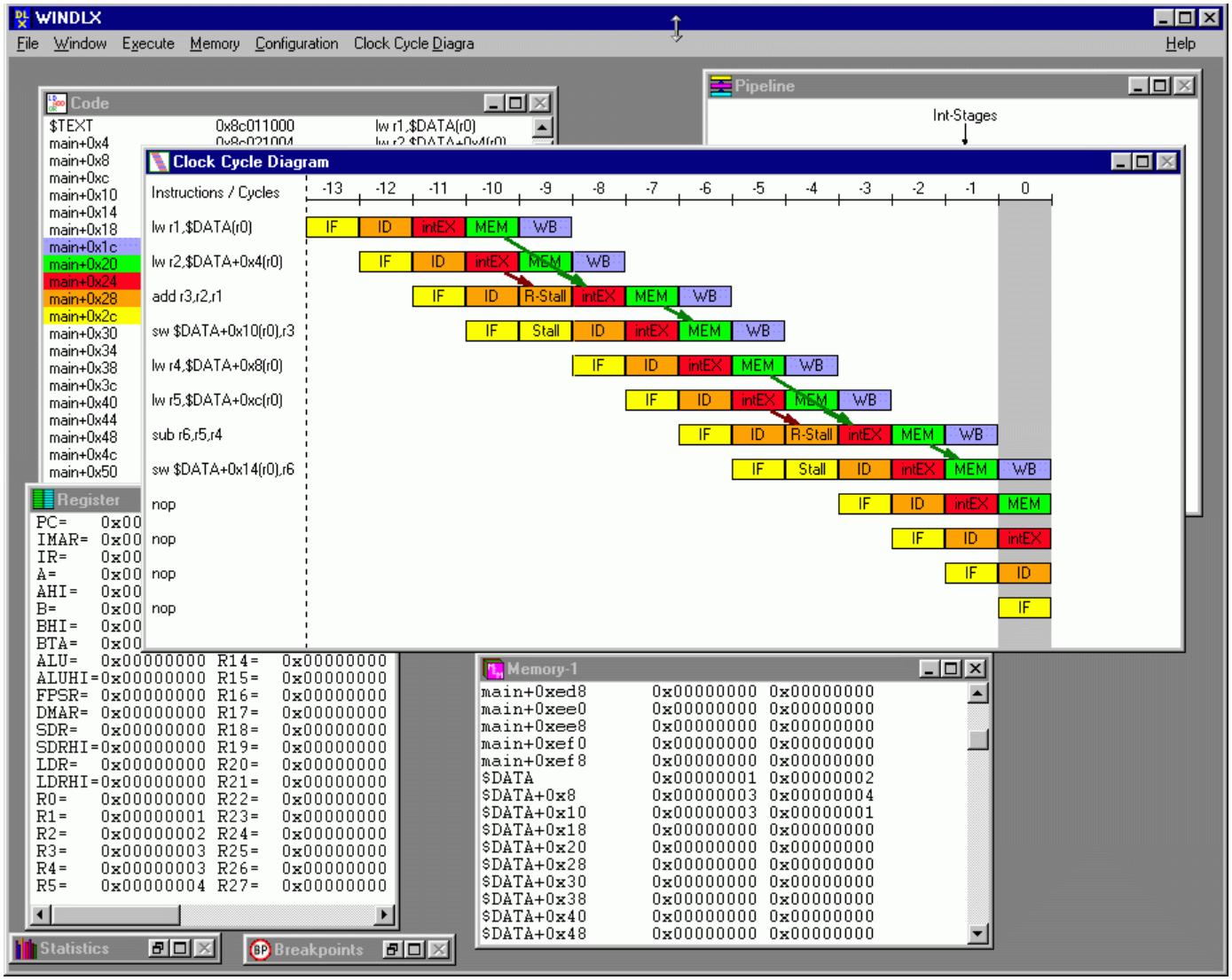
Instruction Scheduling : Beispiel (1)

- a = b + c; d = e - f;

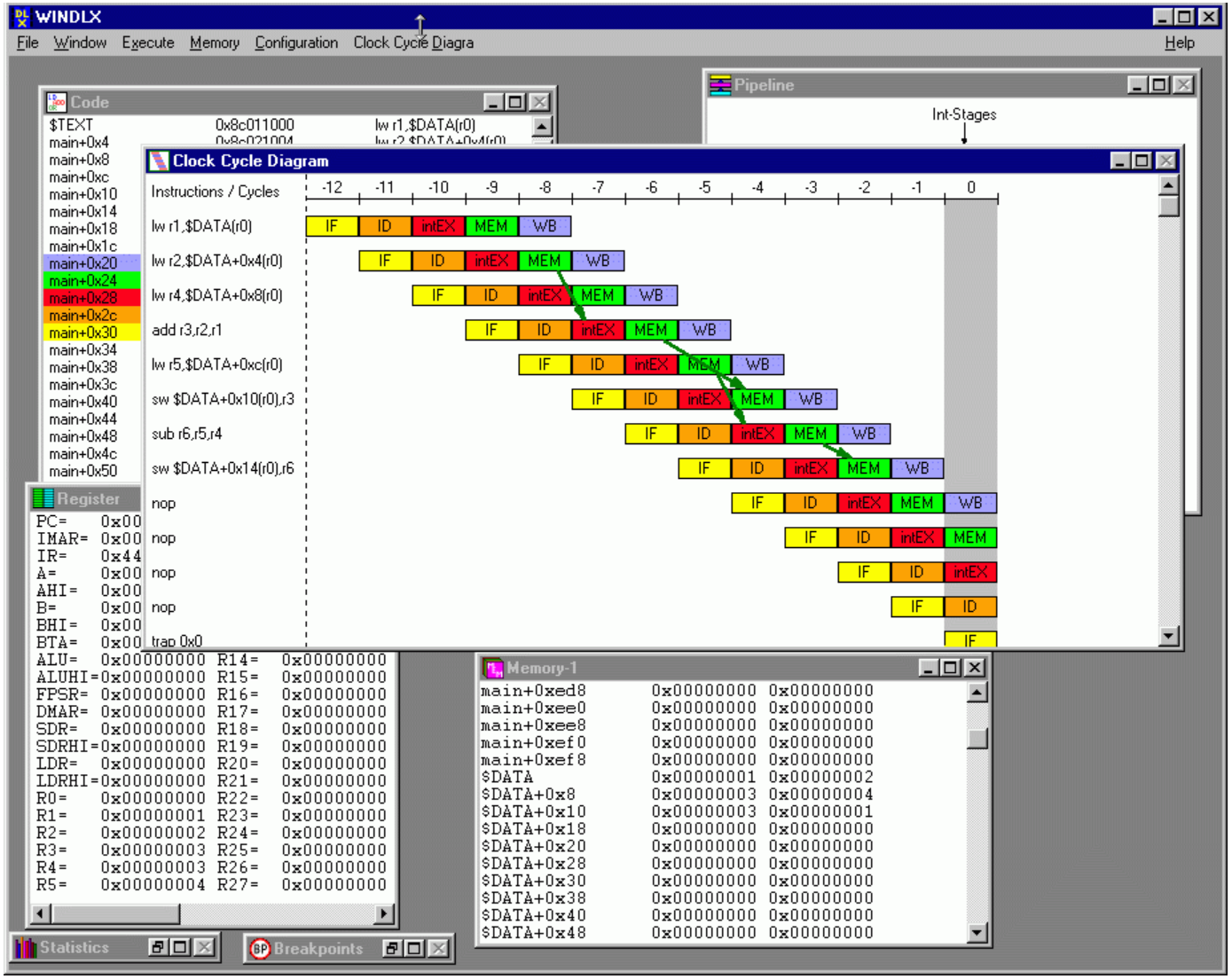
```
                    ;** Store data in data
area
                    .data
                    .word 0x1, 0x2, 0x3,
0x4

                    ;** Store code in code
area
                    .text

                    .global   main
          main:

                    lw  R1, $DATA
                    lw  R2, $DATA+0x4
                    add R3, R2, R1
                    sw  $DATA+0x10, R3
                    lw  R4, $DATA+0x8
                    lw  R5, $DATA+0xc
                    sub R6, R5, R4
                    sw  $DATA+0x14, R6

                    nop
```

```
                    ;** Store data in data
area
                    .data
                    .word 0x1, 0x2, 0x3,
0x4

                    ;** Store code in code
area
                    .text

                    .global   main
          main:

                    lw  R1, $DATA
                    lw  R2, $DATA+0x4
                    lw  R4, $DATA+0x8
                    add R3, R2, R1
                    lw  R5, $DATA+0xc
                    sw  $DATA+0x10, R3
                    sub R6, R5, R4
                    sw  $DATA+0x14, R6

                    nop
```

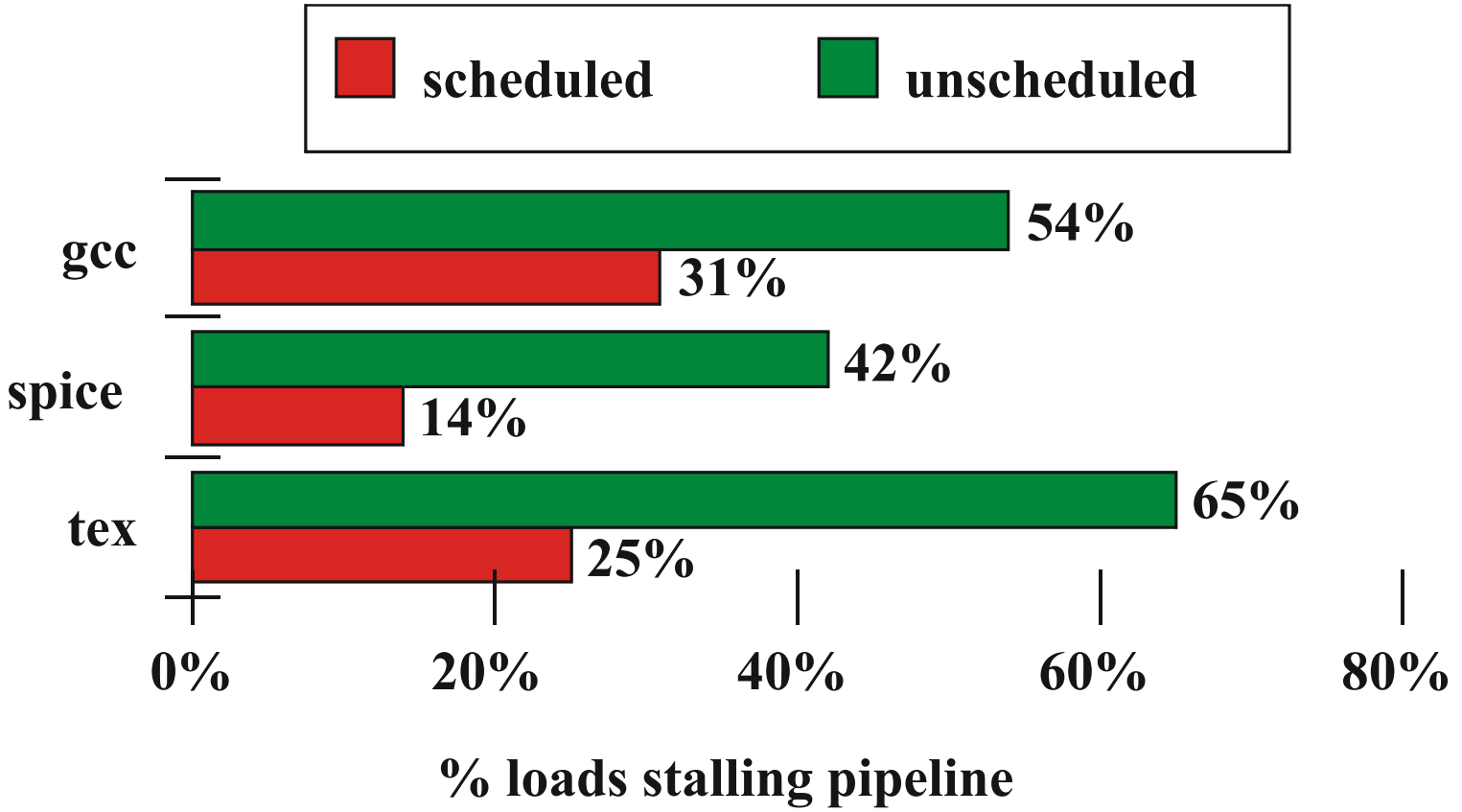
Instruction Scheduling : Beispiel (2)



Instruction Scheduling : Beispiel (3)



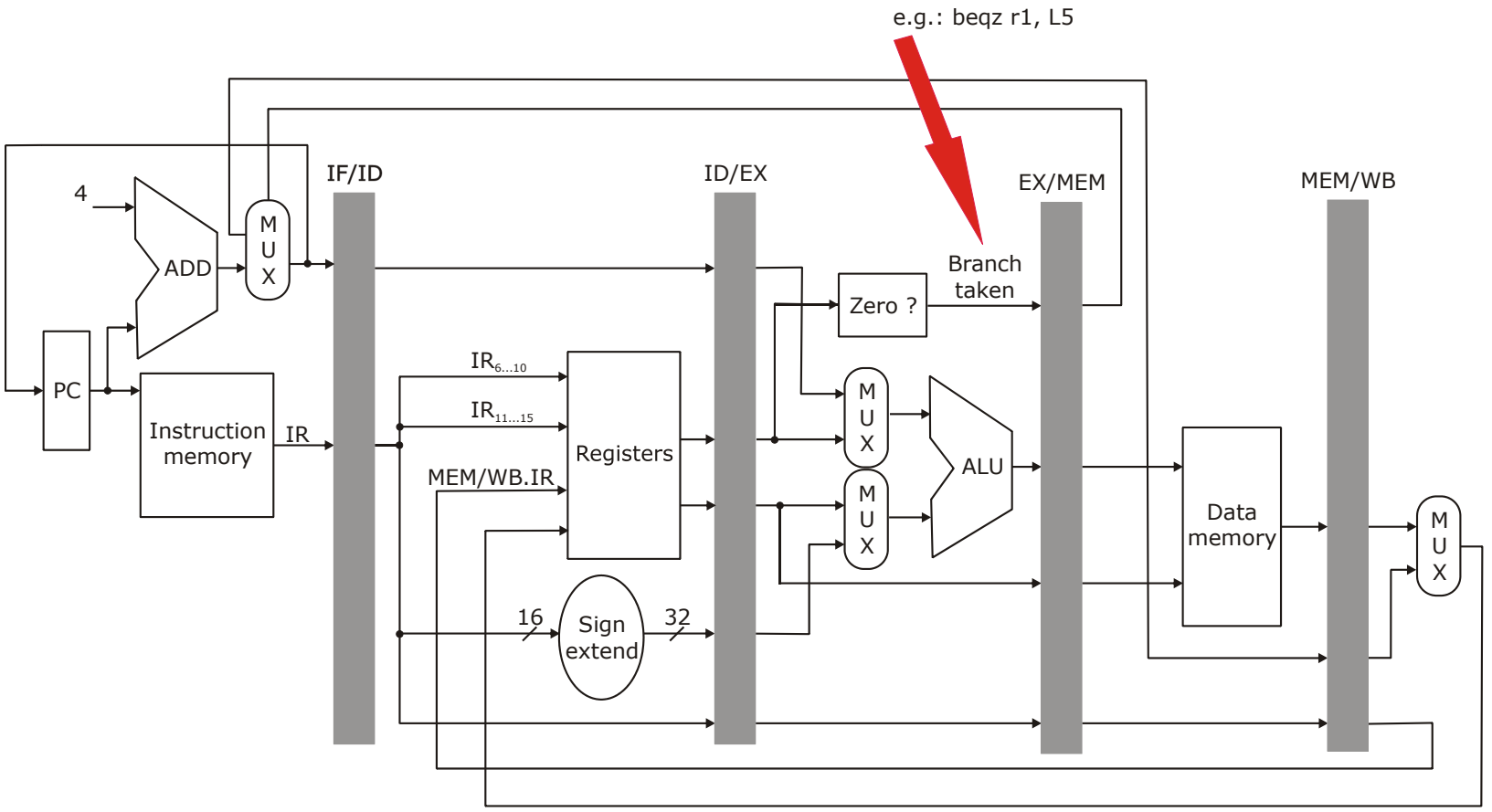
Anteil der Load-Befehle die zu einem „Stall“ führen



Steuerfluss-Hazards

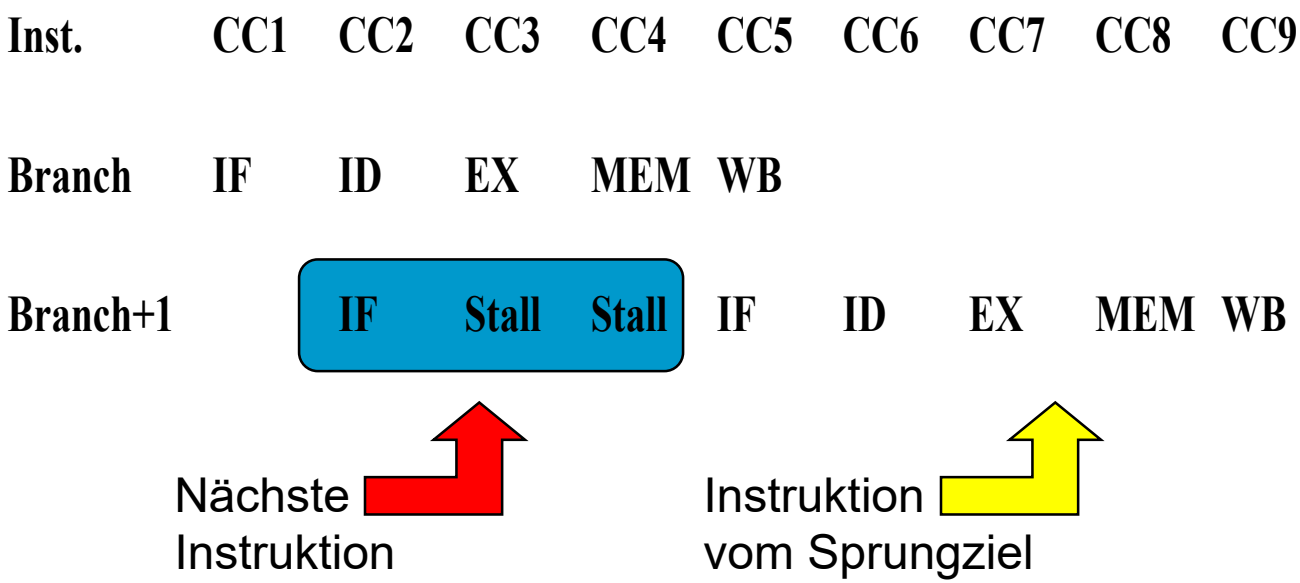
- Veränderungen des linearen Programmablaufs durch bedingte Sprungbefehle (Branch) oder unbedingte Sprungbefehle (Jump) führen zu Steuerfluss-Hazards.
- Ein “Branch” wird entweder
  - durchgeführt (Taken): `PC <= PC + 4 + Imm.` oder
  - nicht durchgeführt (Not Taken): `PC <= PC + 4`
- Einfachste Lösung: Anhalten der Pipeline, sobald ein “Branch” oder “Jump” erkannt wird.
  - Erkennen des “Branch” oder “Jump” in der ID Stufe
  - Aber: Ob ein bedingter Sprung ausgeführt wird, wird erst in der EX Stufe entschieden

Ausführen eines Branch in der DLX Pipeline



Auswirkungen eines Steuerfluss-Hazard

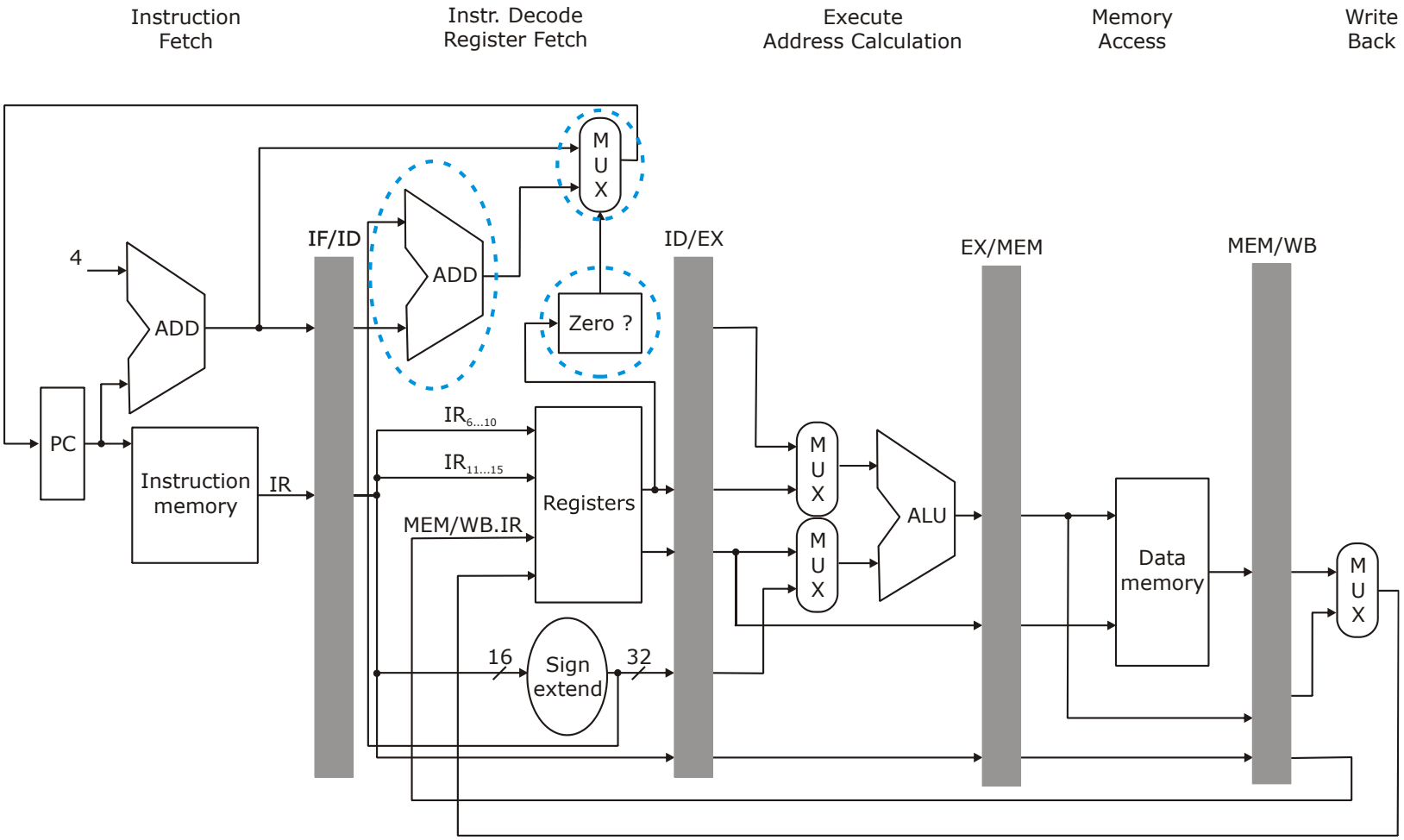
- Wenn der Sprung ausgeführt wird, muss die nächste Instruktion vom Sprungziel geholt werden, d.h. die Stufen IF und ID müssen erneut ausgeführt werden und es entstehen drei Leerzyklen.



Optimierung der Sprungausführung

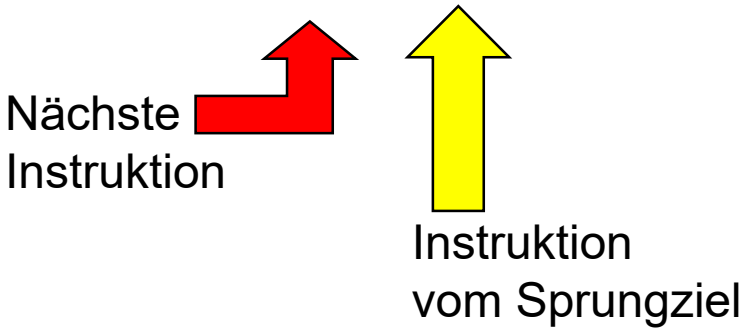
- Zwei Schritte:
  - Frühere Entscheidung ob Sprung auszuführen ist, UND
  - Frühere Berechnung der Adresse
- Sprungbedingung: Rx = 0 oder Rx /= 0 (BEQZ, BNEZ)
- Lösung im DLX:
  - Registertest in ID-Stufe
  - Zusätzlicher Addierer um Adresse in ID-Stufe zu berechnen
  - Ergebnis: Nur noch ein Leerzyklus

Optimierung der Sprungausführung (2)



Optimierung der Sprungausführung (3)

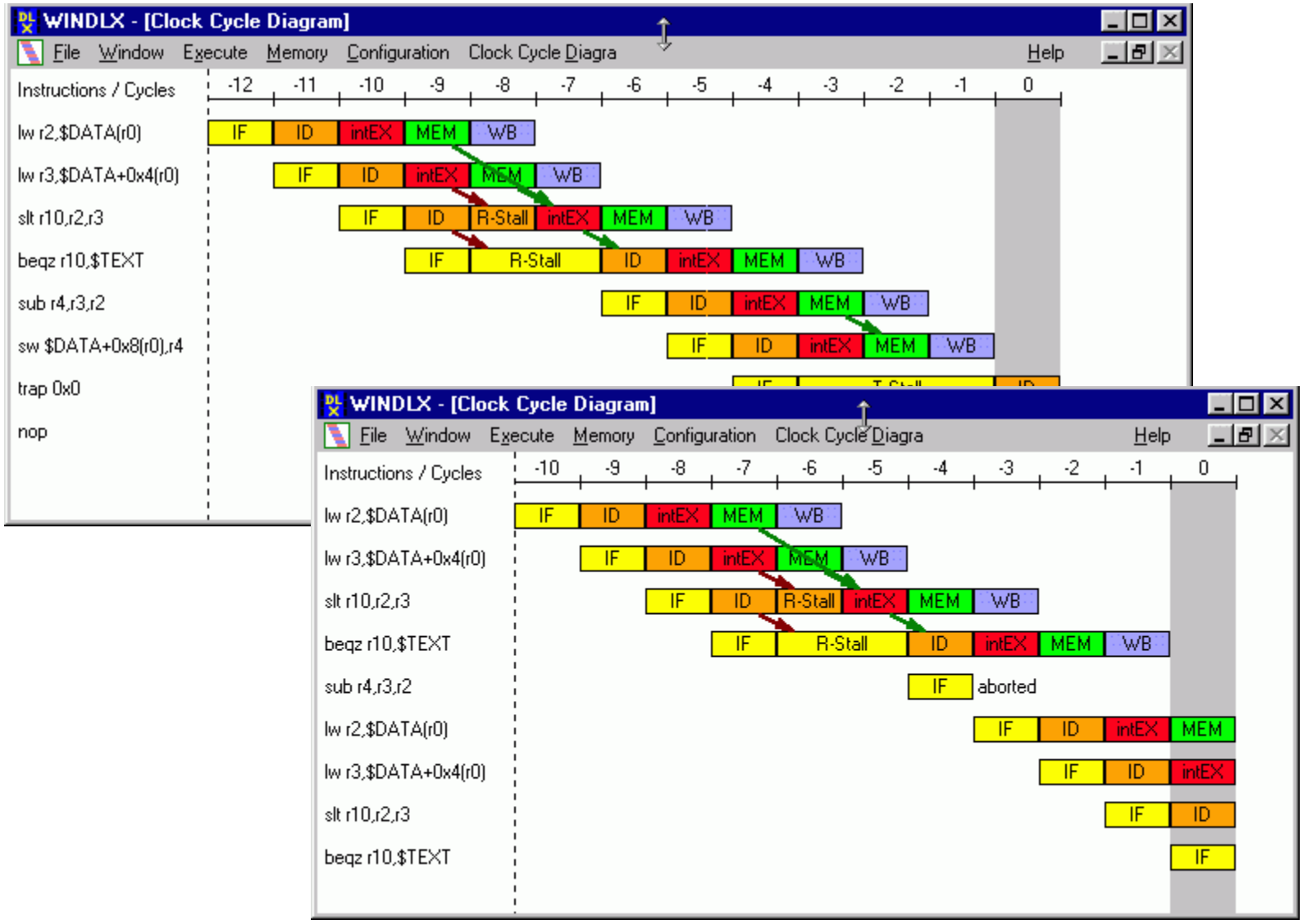
Inst.	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Branch	IF	ID	EX	MEM	WB				
Branch+1		IF	IF	ID	EX	MEM	WB		



Beispiel (1)

```
          ;** Store data in data area
          .data
          .word 0x5, 0x2

          .text
          .global  main
main:
          lw  R2, $DATA
          lw  R3, $DATA+0x4
          slt R10, R2, R3            ;R2<R3?
          beqz R10, main             ;Repeat if R2>R3
          sub R4, R3, R2             ;R4=R3-R2
          sw  $DATA+0x8, R4

          ;*** end
          trap
```

Beispiel (2)


WINDLX - [Clock Cycle Diagram]
File Window Execute Memory Configuration Clock Cycle Diagra Help
Instructions / Cycles
-12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0
lw r2,$DATA(r0)
lw r3,$DATA+0x4(r0)
slt r10,r2,r3
beqz r10,$TEXT
sub r4,r3,r2
sw $DATA+0x8(r0),r4
trap 0x0
nop
IF ID intEX MEM WB
R-Stall
WINDLX - [Clock Cycle Diagram]
File Window Execute Memory Configuration Clock Cycle Diagra Help
Instructions / Cycles
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0
lw r2,$DATA(r0)
lw r3,$DATA+0x4(r0)
slt r10,r2,r3
beqz r10,$TEXT
sub r4,r3,r2
aborted
lw r2,$DATA(r0)
lw r3,$DATA+0x4(r0)
slt r10,r2,r3
beqz r10,$TEXT

Statistiken

- SPEC Benchmarks für DLX:
  - Häufigkeit von Branches:
    - 14% bis 16% in Integer-Programmen
    - 3 % bis 12% in Floating-Point-Programmen.
  - ~75% davon sind Vorwärts-Verzweigungen
  - ~60% der Vorwärts-Verzweigungen werden ausgeführt
  - ~80% der Rückwärtsverzweigungen werden ausgeführt (z.B. Schleifen)
  - Daher: 65% aller bedingten Sprünge werden ausgeführt

Einfluss von Pipeline Stalls auf die Performance

- Verringerung des Speedups:

$$\text{Speedup from pipelining} = \frac{\text{Ave Instr Time unpipelined}}{\text{Ave Instr Time pipelined}} = \frac{\text{CPI}_{\text{unpipelined}} \times \text{Clock Cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock Cycle}_{\text{pipelined}}}$$

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall CPI} = 1 + \text{Pipeline stall CPI}$$

$$\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}}$$

Zusammenfassung Pipelining

- Pipelining verbessert den Durchsatz des Programms, nicht die Latenz eines Befehls
- Hazards begrenzen die möglichen Leistungsverbesserungen durch Pipelining
  - Struktur-Hazards: mehr HW-Ressource, z.B. Harvard-Architektur
  - Daten-Hazards: Forwarding, Compiler-Scheduling
  - Steuerfluss-Hazards: HW-Verbesserungen, Sprungvorhersage
- Erhöhung der Pipeline-Tiefe erhöht insbesondere das Problem der Steuerfluss-Hazards
- Compiler spielt eine wesentliche Rolle bei der Optimierung von Programmen für RISC-Maschinen.