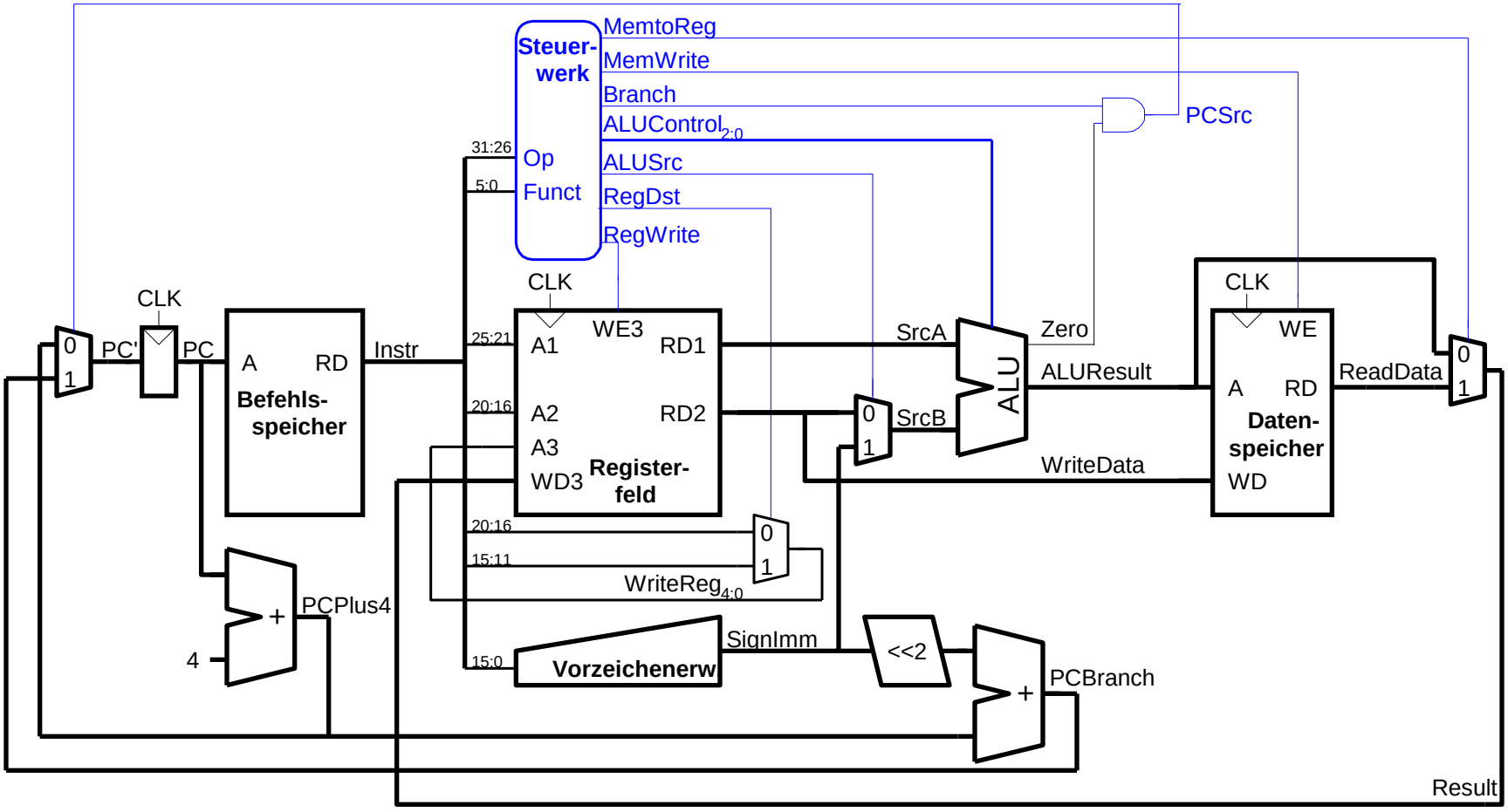
Organisatorisch

TECHNISCHE UNIVERSITÄT DARMSTADT

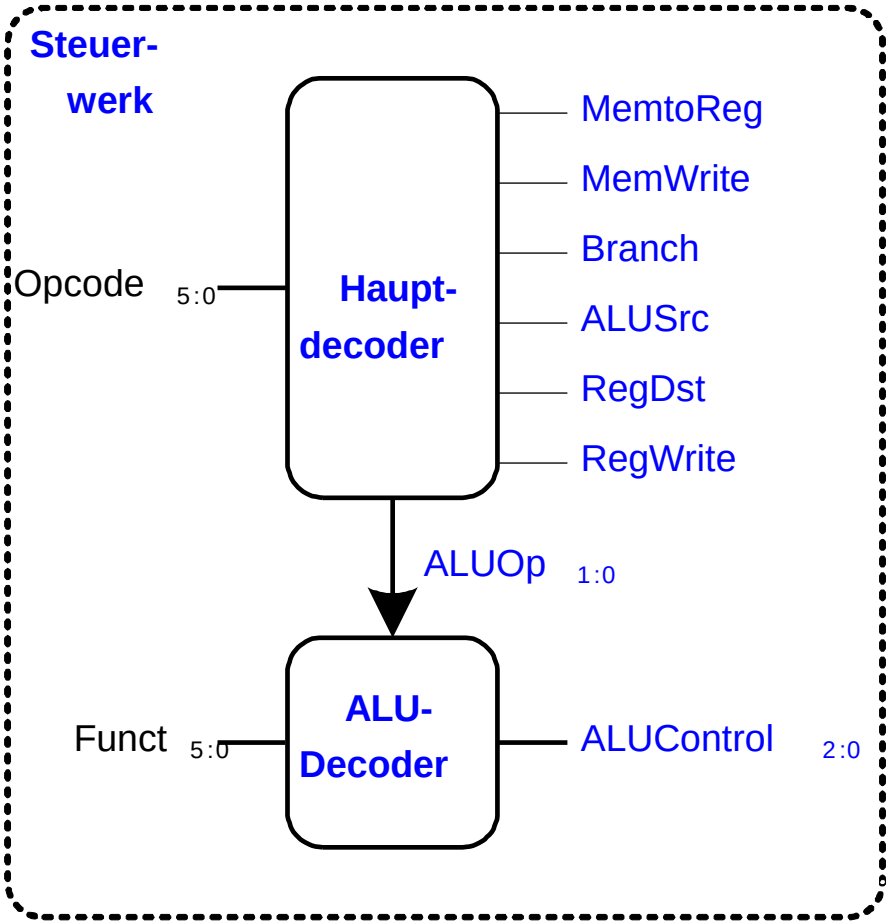
- **Gruppen:** sich bis heute Abend (21.04.) als Gruppen in Moodle eintragen
  - 2 Leute müssen zu dritt arbeiten, da wir nur 24 Boards haben und 50 Leute angemeldet sind
  - falls Sie keinen Partner/in haben, Sie können nach der Vorlesung nach vorne kommen, um einen Partner/in zu finden
  - **ab Lab 2** müssen alle Abgaben als Gruppe eingereicht
- **Vivado installieren** (Anleitung steht auf Moodle bei Lab 2)
- **Nexys4-DDR Boards** nächste Woche abholen:
  - Bei Frau Reimund (S2/02 E103)
  - **Do (28.04.) 13:00-15:30**
  - **Fr (29.04.) 09:30-12:30**
  - Nur ein Mitglied der Gruppe muss kommen
  - 50 Euro Pfand (wird Ihnen zurückgegeben mit der Ruckgabe des Boards am Ende des Semesters)

Embedded Systems & Applications

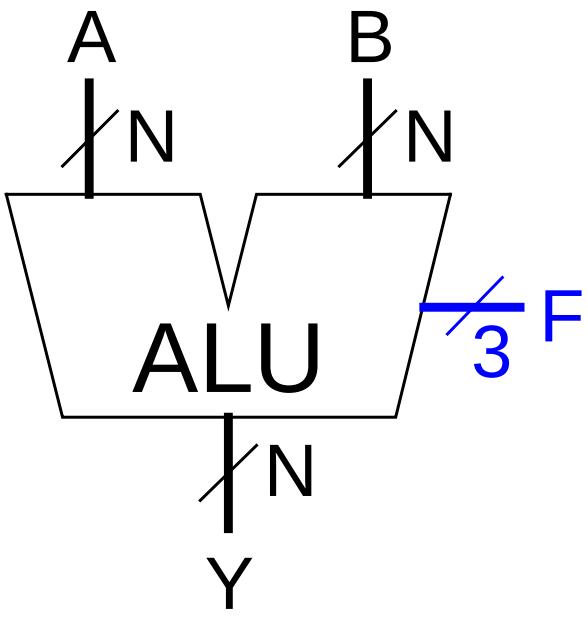
Vollständiger Ein-Takt-Prozessor



Steuerwerk

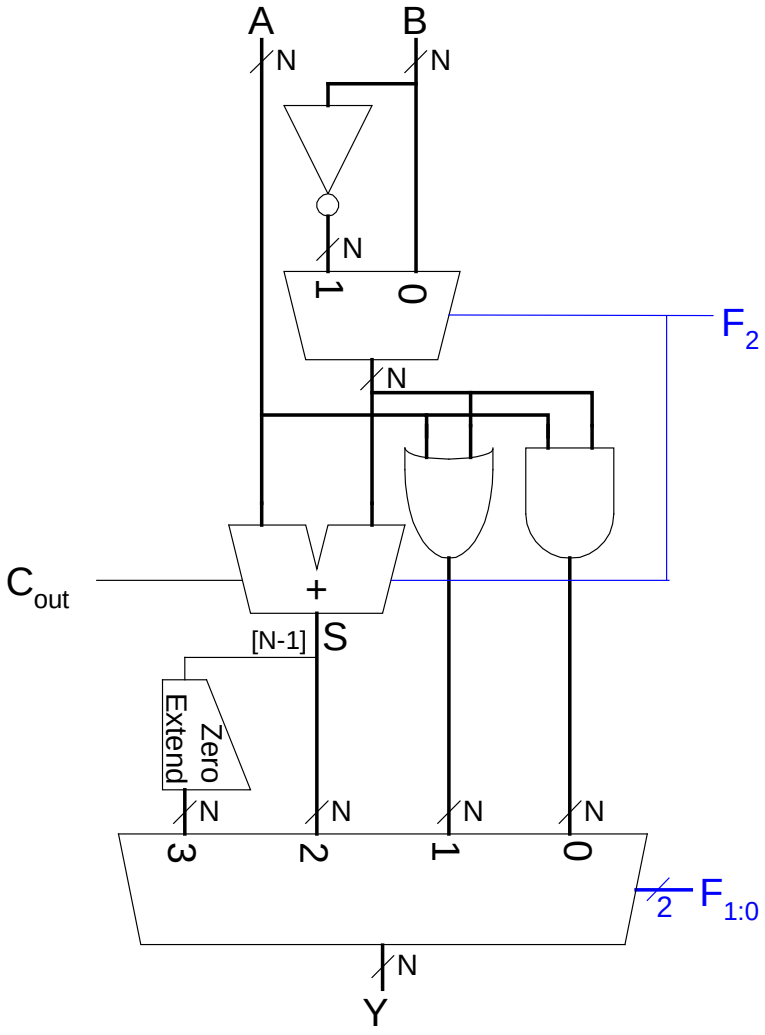

Steuer-
werk
Opcode
5:0
Haupt-
decoder
MemtoReg
MemWrite
Branch
ALUSrc
RegDst
RegWrite
ALUOp
1:0
Funct
5:0
ALU-
Decoder
ALUControl
2:0

Zur Erinnerung: ALU



$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B
011	unbenutzt
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Zur Erinnerung: ALU



$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B
011	unbenutzt
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Steuerwerk: ALU-Decoder

TECHNISCHE UNIVERSITÄT DARMSTADT

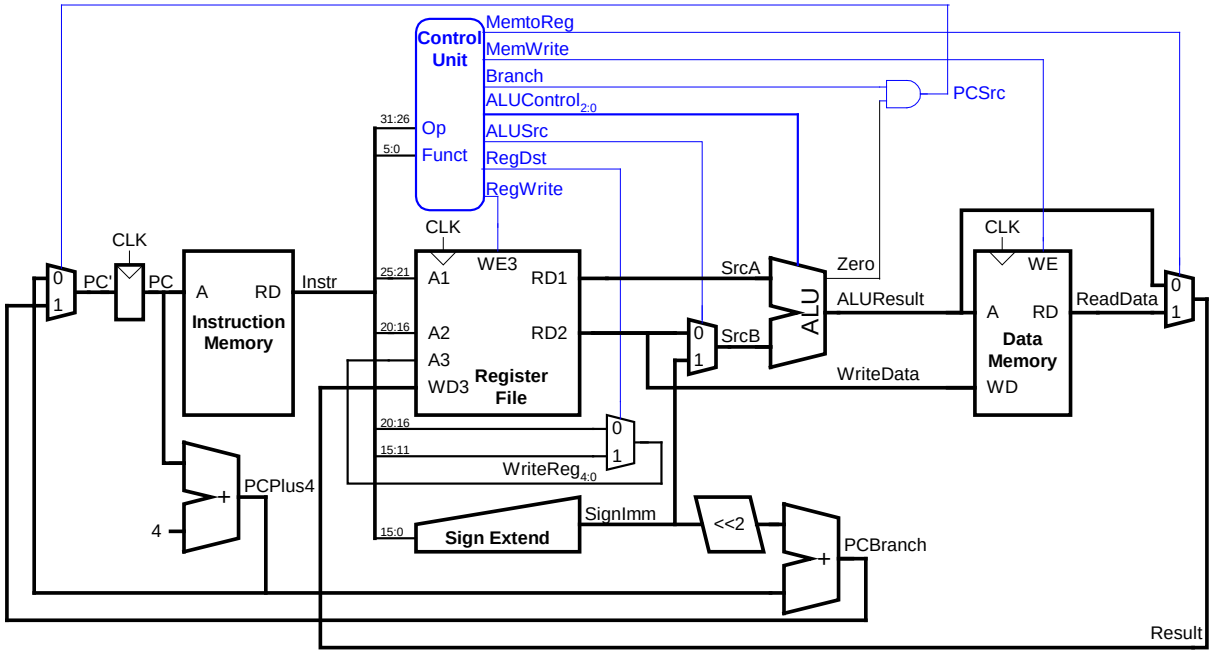
$ALUOp_{1:0}$	Bedeutung
00	Addiere
01	Subtrahiere
10	Werte Funct-Feld aus
11	unbenutzt

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Embedded Systems & Applications

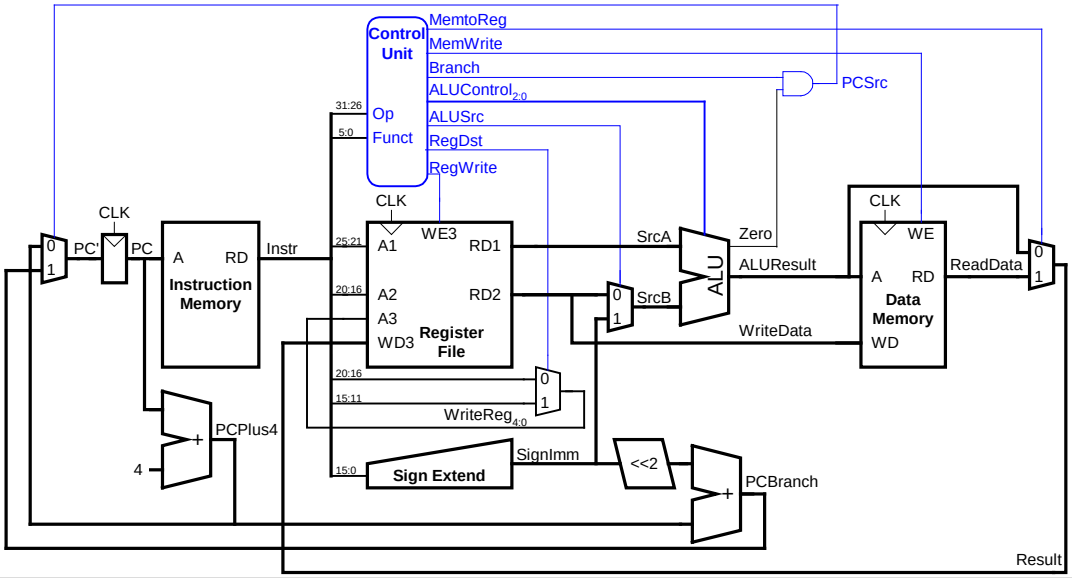
Steuerwerk: Hauptdecoder

Instruktion	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-Typ	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							

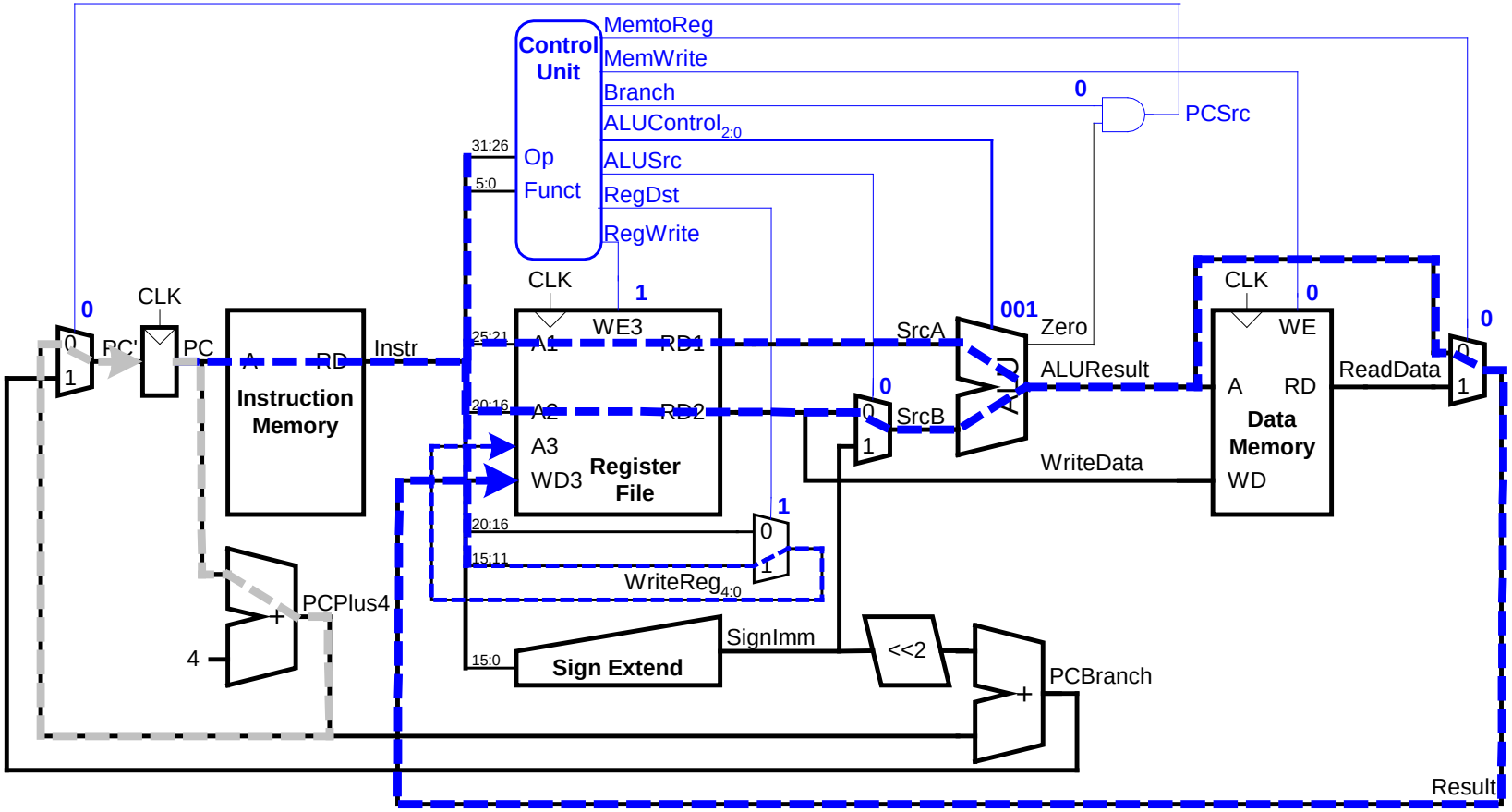


Steuerwerk: Hauptdecoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

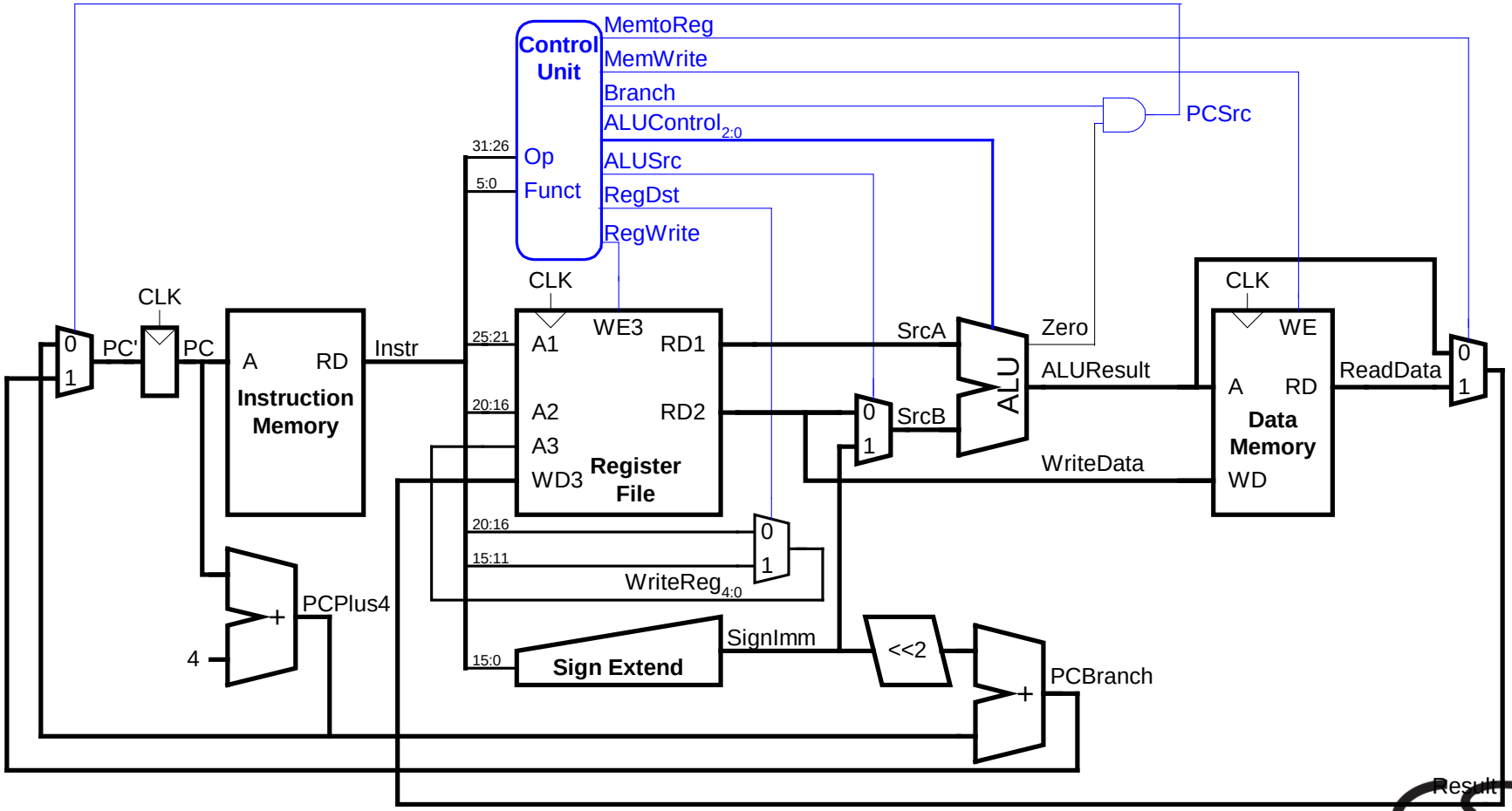


Beispiel im Ein-Takt Datenpfad: or



Erweitere Funktionalität: `addi`

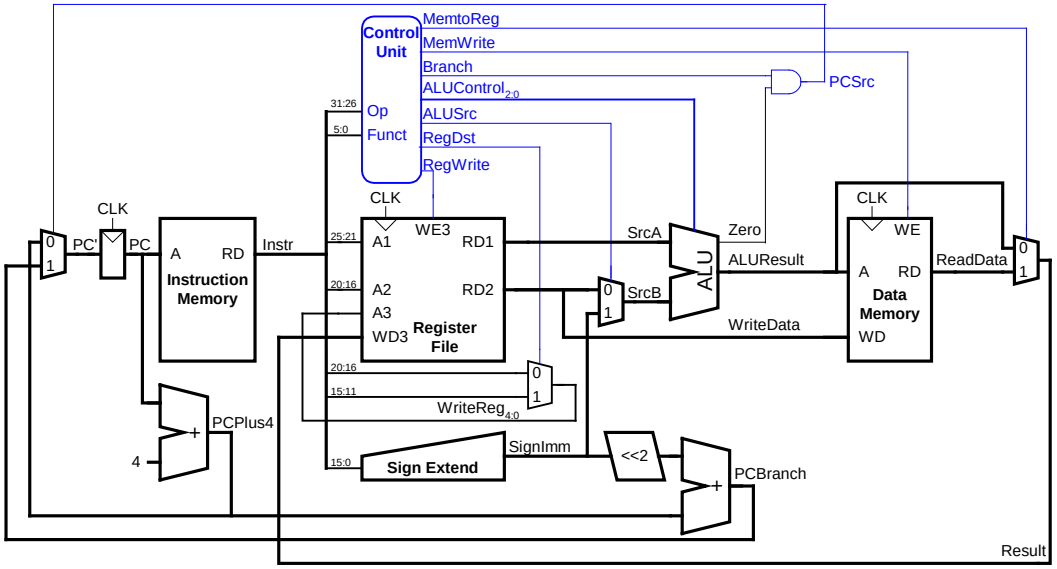
- Keine Änderung am Datenpfad nötig



Erweitere Steuerwerk: addi

TECHNISCHE UNIVERSITÄT DARMSTADT

Instruktion	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-Typ	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	**001000**							



Embedded Systems & Applications

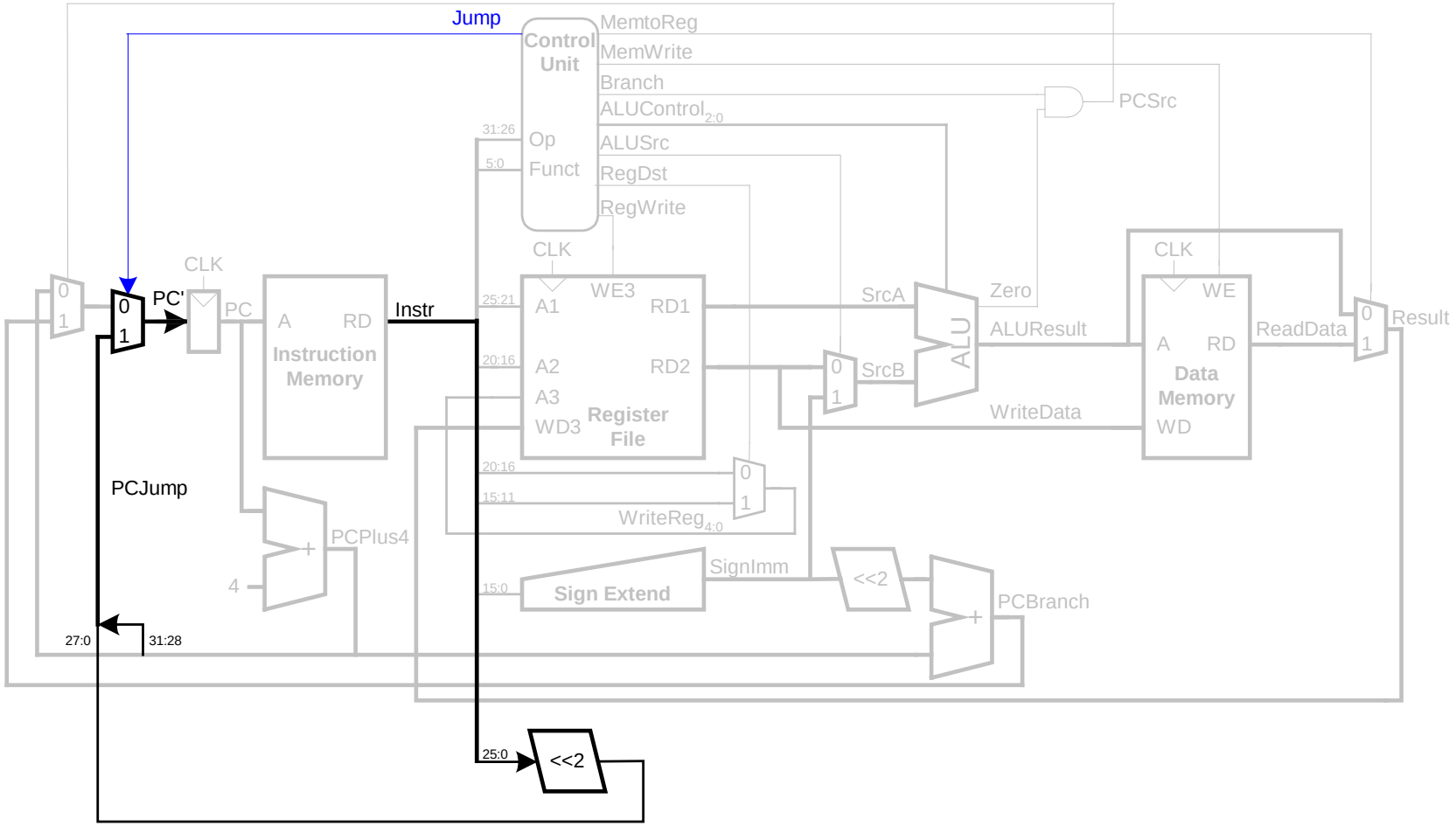
Erweitere Steuerwerk: `addi`

TECHNISCHE UNIVERSITÄT DARMSTADT

Instruktion	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-Typ	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`addi`	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**

Embedded Systems & Applications

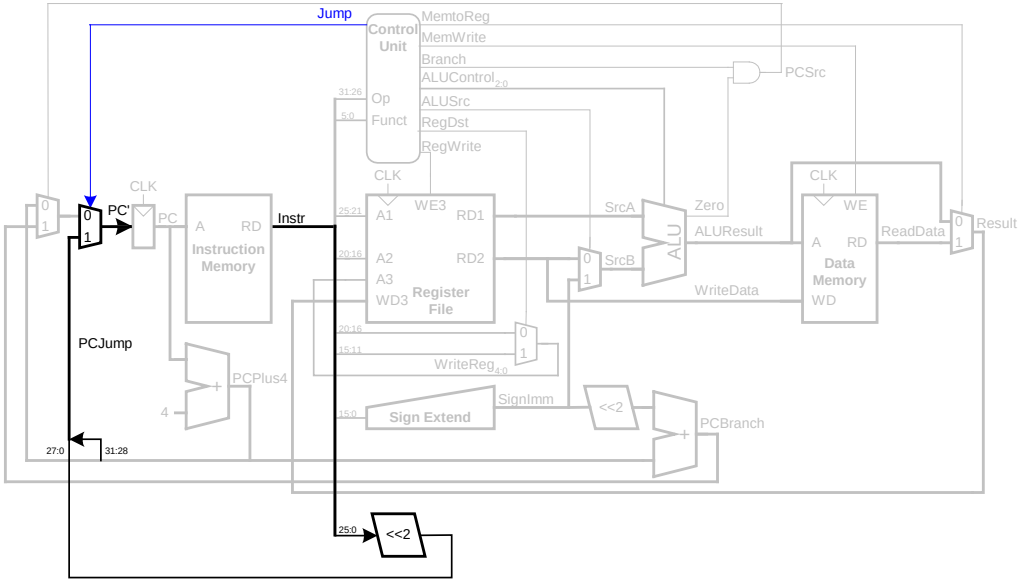
Erweitere Funktionalität: `j`



Steuerwerk: Hauptdecoder

TECHNISCHE UNIVERSITÄT DARMSTADT

Instruktion	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-Typ	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`j`	**000010**								



Embedded Systems & Applications

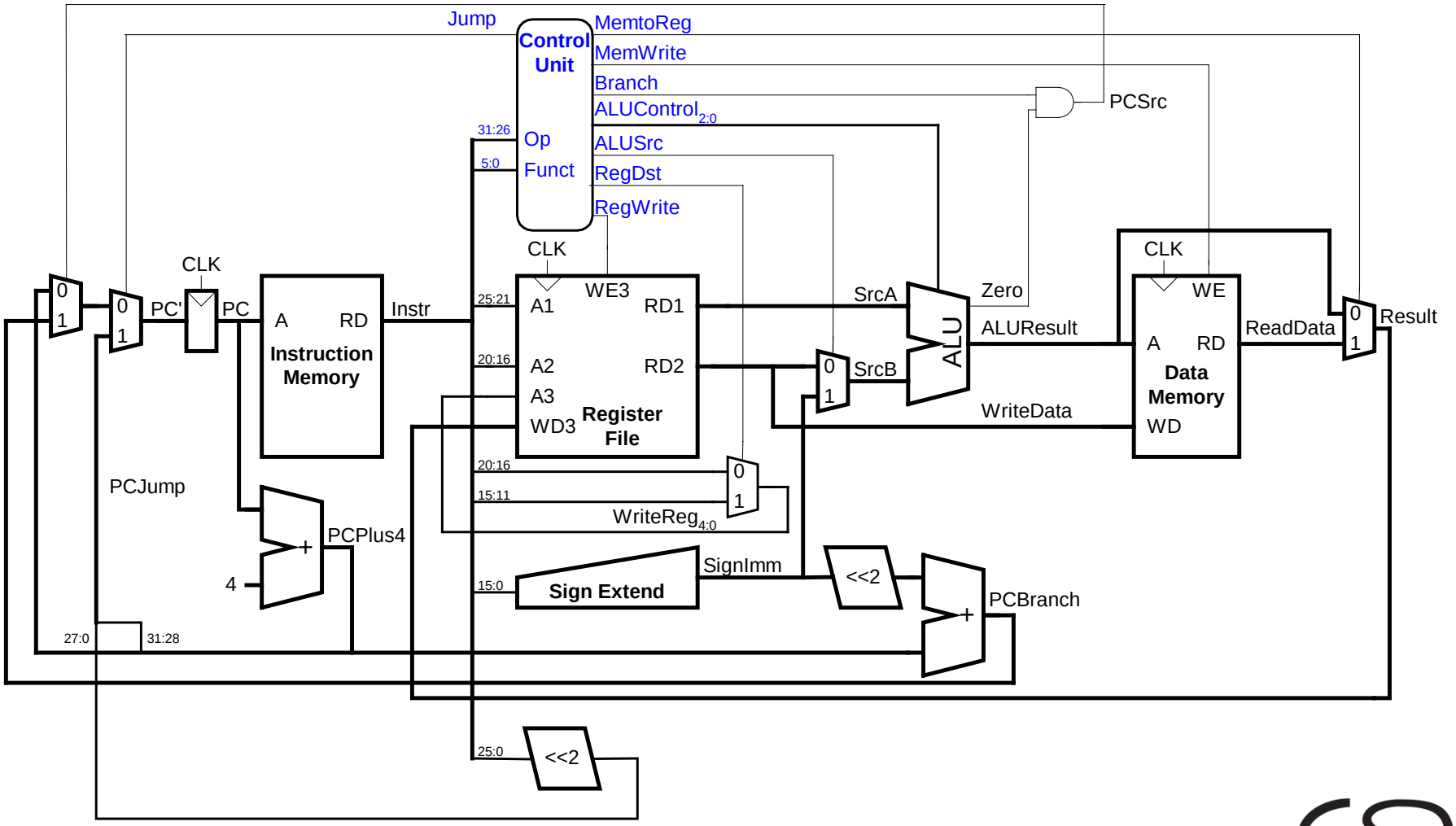
Steuerwerk: Hauptdecoder

TECHNISCHE UNIVERSITÄT DARMSTADT

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`j`	**000010**	**0**	**X**	**X**	**X**	**0**	**X**	**XX**	**1**

Embedded Systems & Applications

Rückblick: Ein-Takt MIPS Prozessor



MIPS Prozessor mit Pipelining

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Parallelität

Teile Ablauf im Ein-Takt-Prozessor in fünf Stufen:

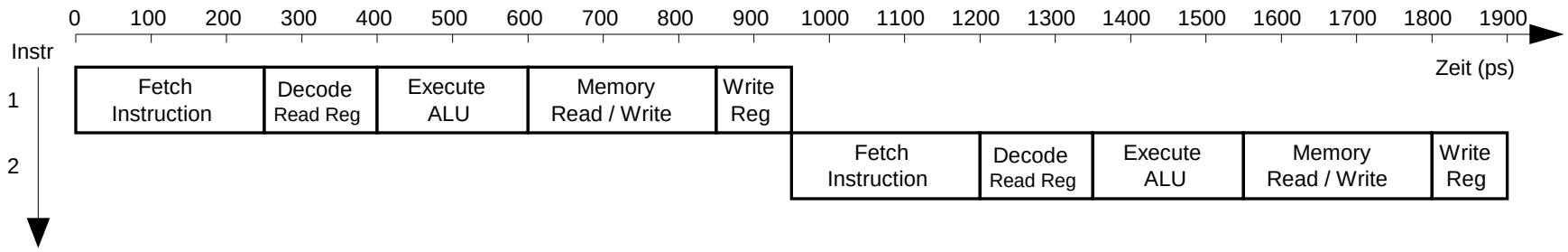
- Hole Instruktion (*Fetch*)
- Dekodiere Bedeutung von Instruktion (*Decode*)
- Führe Instruktion aus (*Execute*)
- Greife auf Speicher zu (*Memory*)
- Schreibe Ergebnisse zurück (*Writeback*)

Füge Pipeline-Register zwischen den Stufen ein

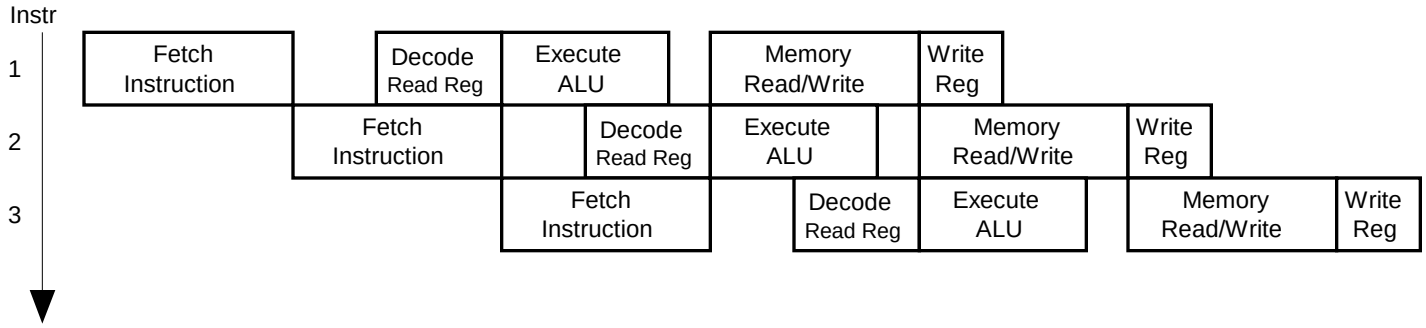
Rechenleistung: Ein-Takt und Pipelined

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein-Takt

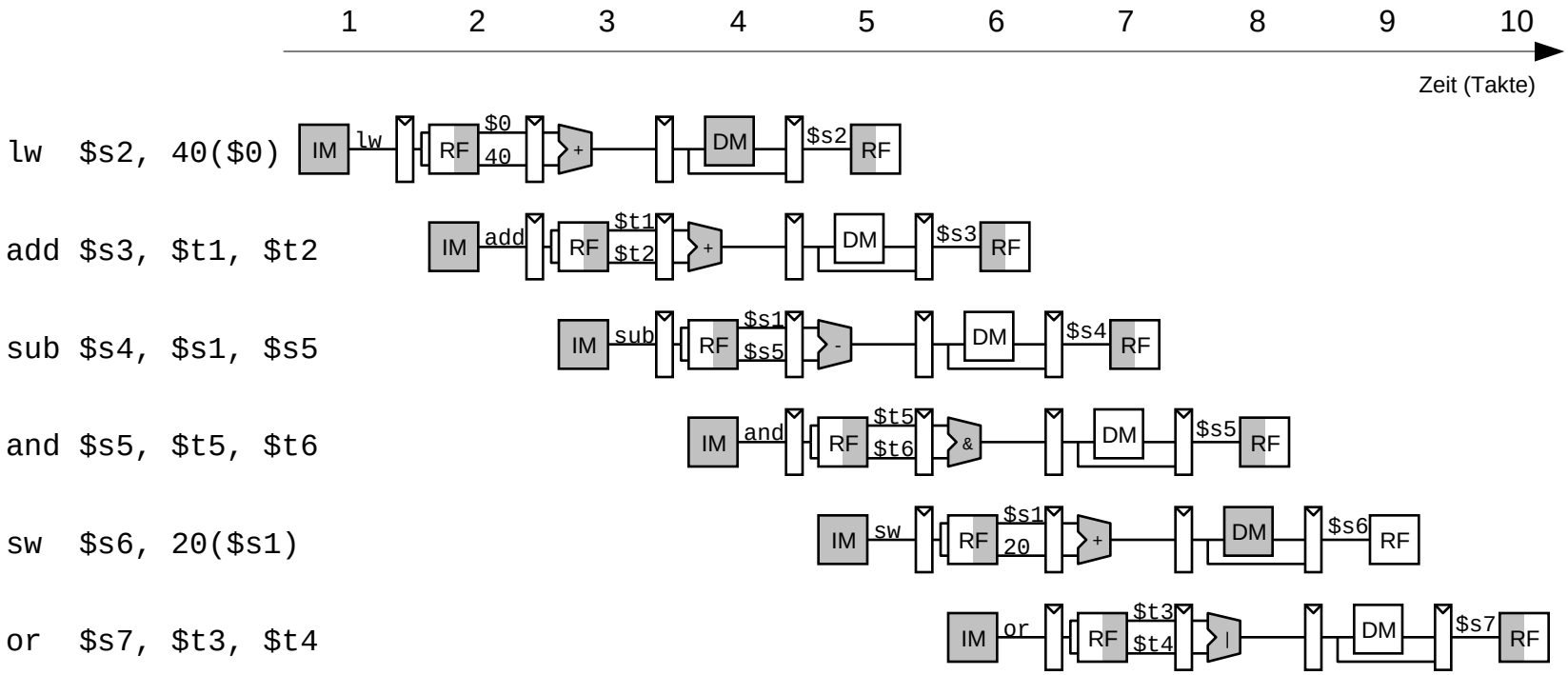


Pipelined

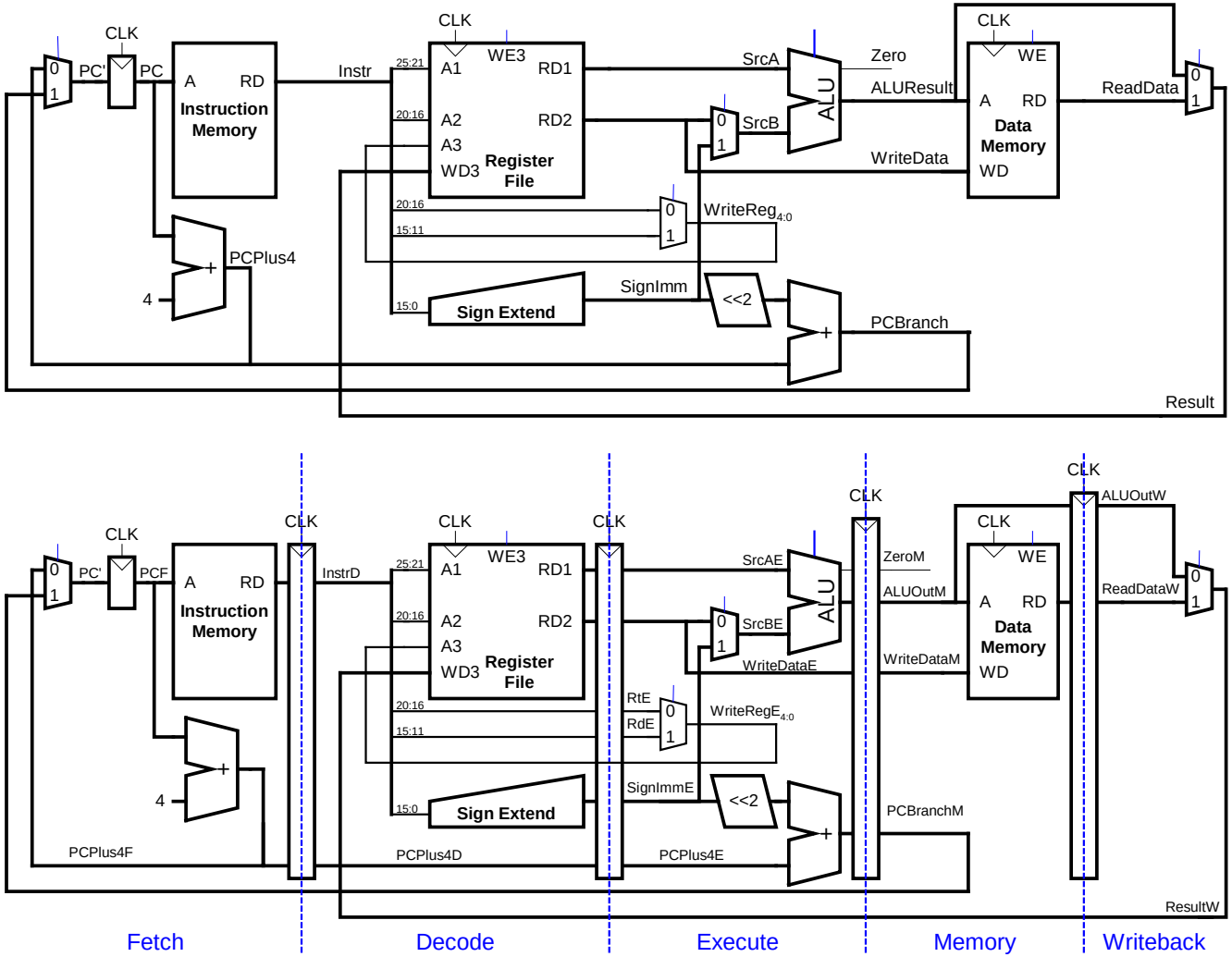


Embedded Systems & Applications

Abstraktere Darstellung des Pipelinings

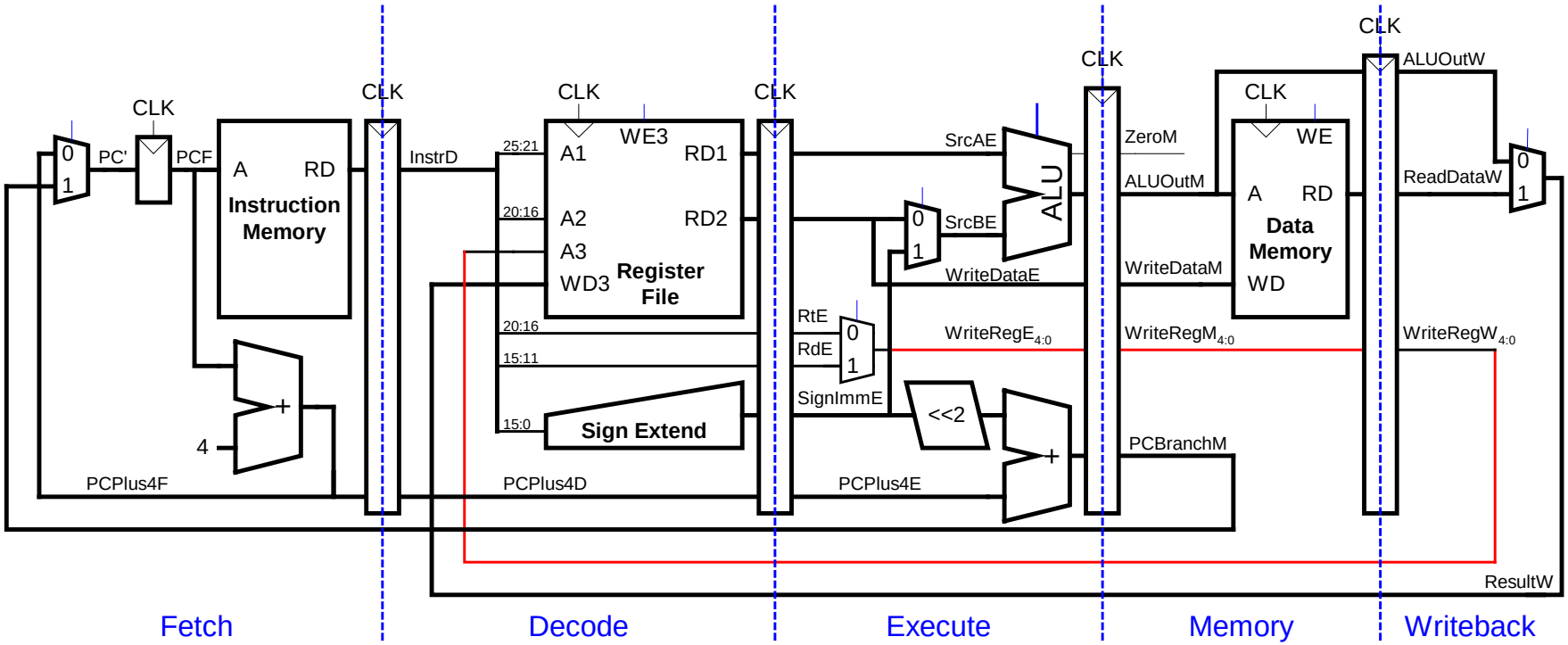


Ein-Takt- und Pipelined-Datenpfad

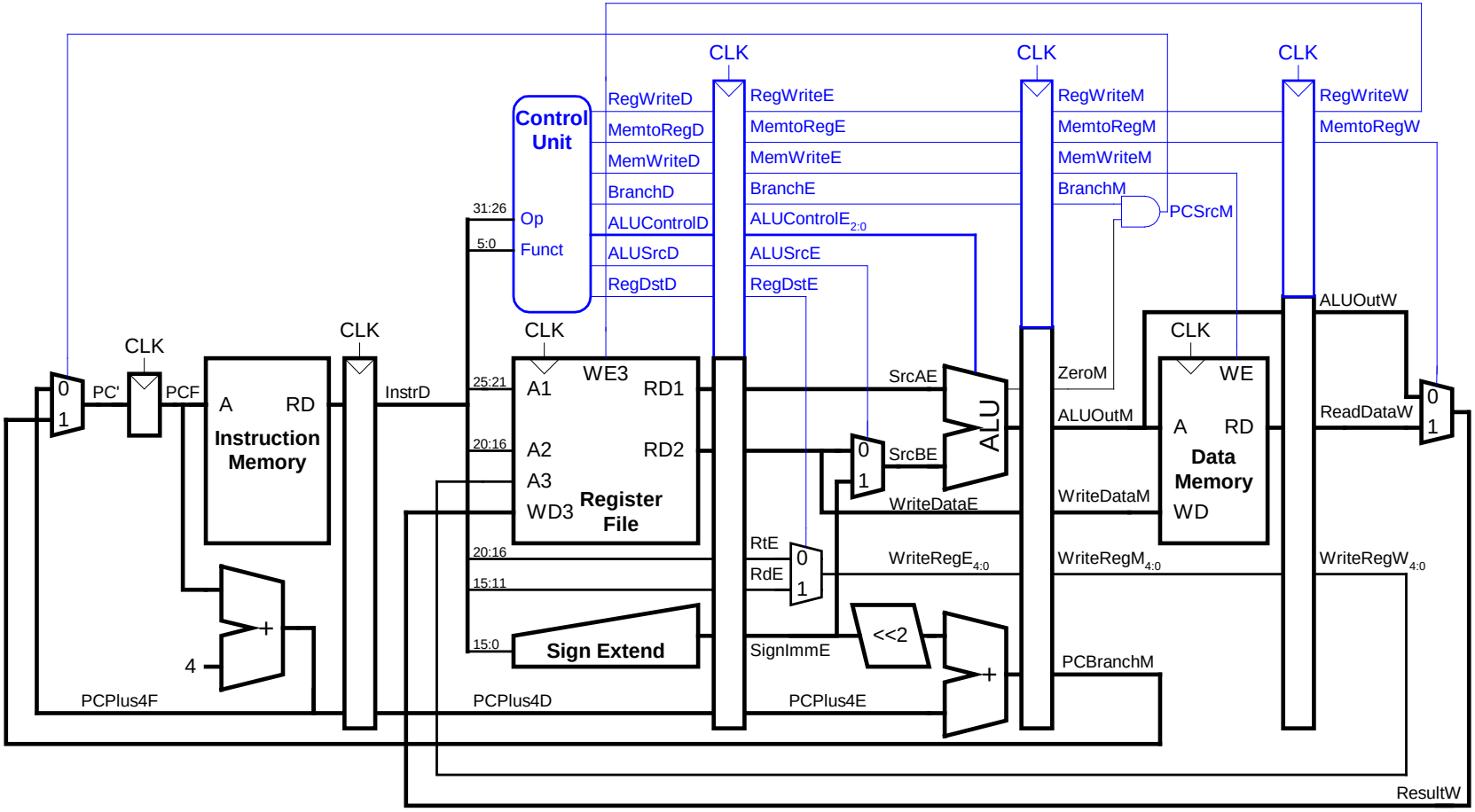


Korrigierter Pipelined-Datenpfad

- **WriteReg** muss zur gleichen Zeit am Registerfeld ankommen wie **Result**



Steuersignale für Pipelined-Datenpfad



Identisch zu Ein-Takt-Steuerwerk, aber Signale verzögert über Pipeline-Stufen

Abhängigkeiten zwischen Pipeline-Stufen (*hazards*)

TECHNISCHE UNIVERSITÄT DARMSTADT

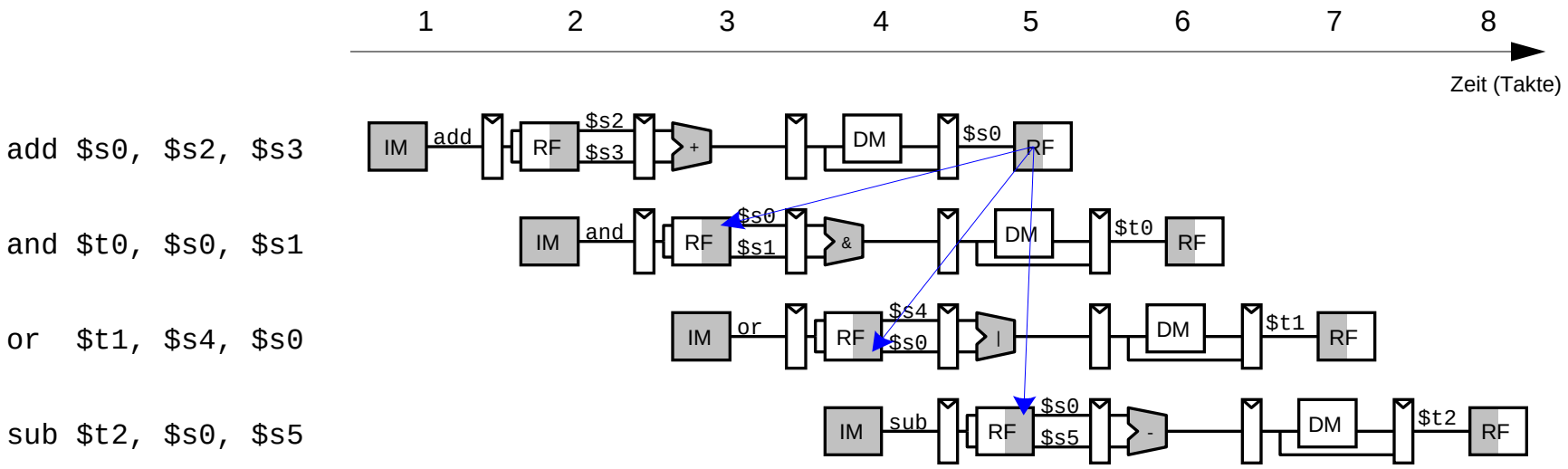
Treten auf wenn eine

- Instruktion vom Ergebnis einer vorhergehenden abhängt
- … diese aber noch kein Ergebnis geliefert hat

Arten von Hazards

- **Data Hazard:** z.B. Neuer Wert von Register noch nicht in Registerfeld eingetragen
- **Control Hazard:** Unklar welche Instruktion als nächstes ausgeführt werden muss
  - Tritt bei Verzweigungen auf

Data Hazard



Hier: **Read-after-Write** Hazard (RAW)
- $s0 „muss vor Lesen geschrieben werden“

Umgang mit Data Hazards

TECHNISCHE UNIVERSITÄT DARMSTADT

Möglichkeiten:

Umgang mit Data Hazards

TECHNISCHE UNIVERSITÄT DARMSTADT

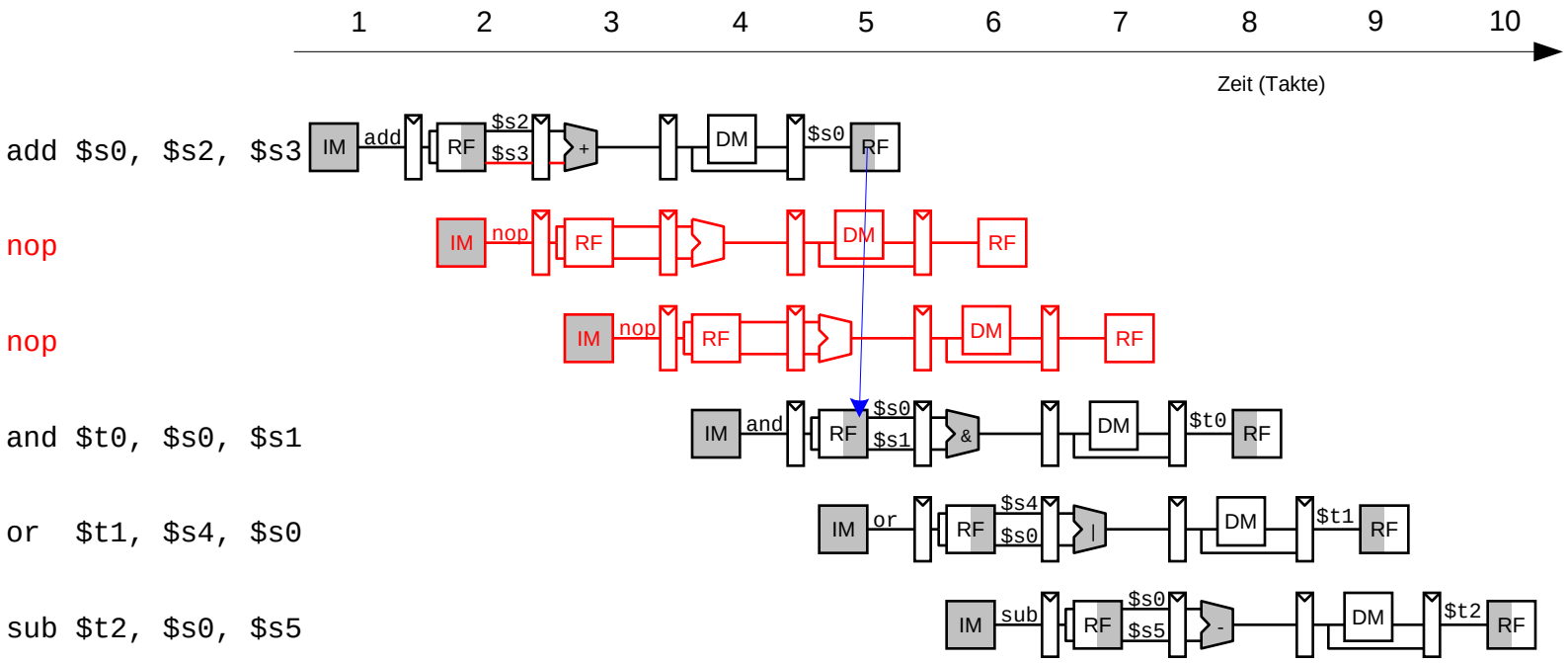
Möglichkeiten:

1. Plane Wartezeiten von Anfang an ein
   - Füge nops zur Compile-Zeit ein
   - *scheduling*
2. Stelle Maschinencode zur Compile-Zeit um
   - *scheduling / reordering*
3. Leite Daten zur Laufzeit schneller über Abkürzungen weiter
   - *bypassing / forwarding*
4. Halte Prozessor zur Laufzeit an bis Daten da sind
   - *stalling*

Beseitigung von Data Hazards zur Compile-Zeit

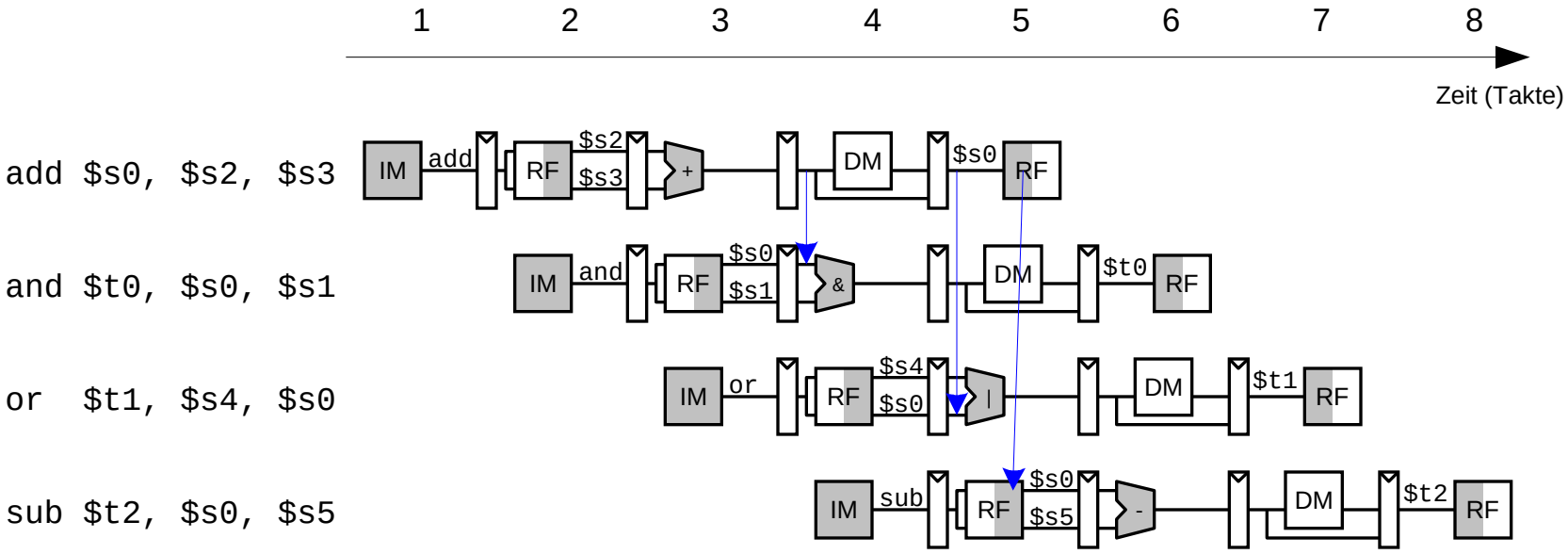
TECHNISCHE UNIVERSITÄT DARMSTADT

- Füge ausreichend viele nops ein bis Ergebnis bereitsteht
- Oder schiebe unabhängige Instruktionen nach vorne (statt nops)

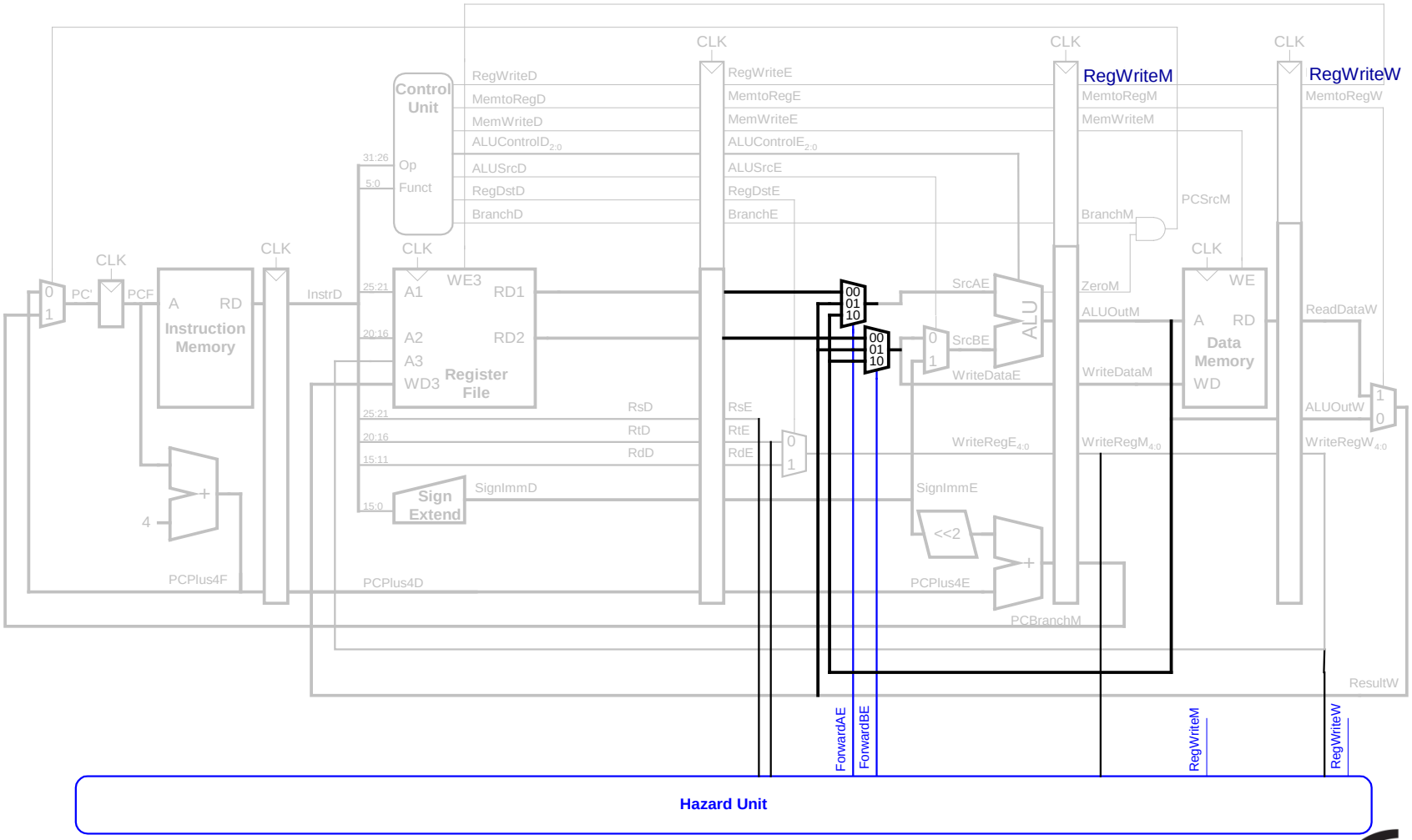


Embedded Systems & Applications

Data Forwarding: “Abkürzungen” einbauen



Data Forwarding: “Abkürzungen” einbauen



Data Forwarding: “Abkürzungen” einbauen

TECHNISCHE UNIVERSITÄT DARMSTADT

“Abkürzung” zur Execute-Stufe von

- Memory-Stufe oder
- Writeback-Stufe

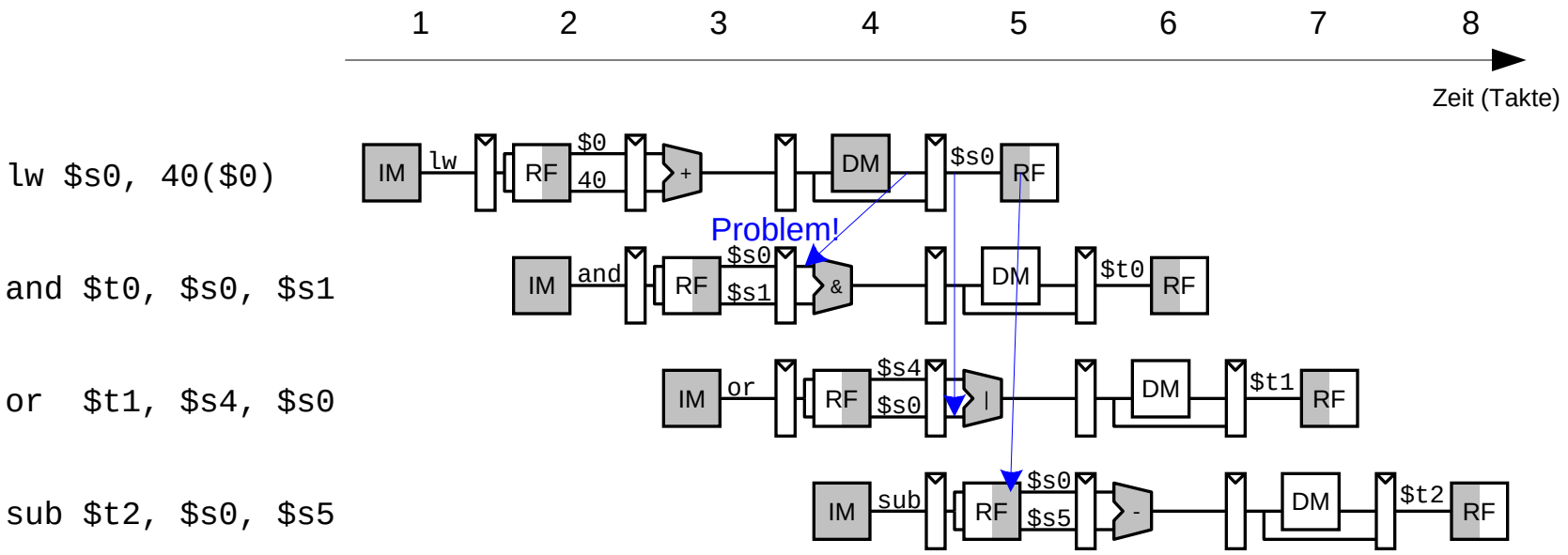
Forwarding-Logik für Signal *ForwardAE* (Weiterleiten von Operand A):

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
          ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
          ForwardAE = 01
else
          ForwardAE = 00
```

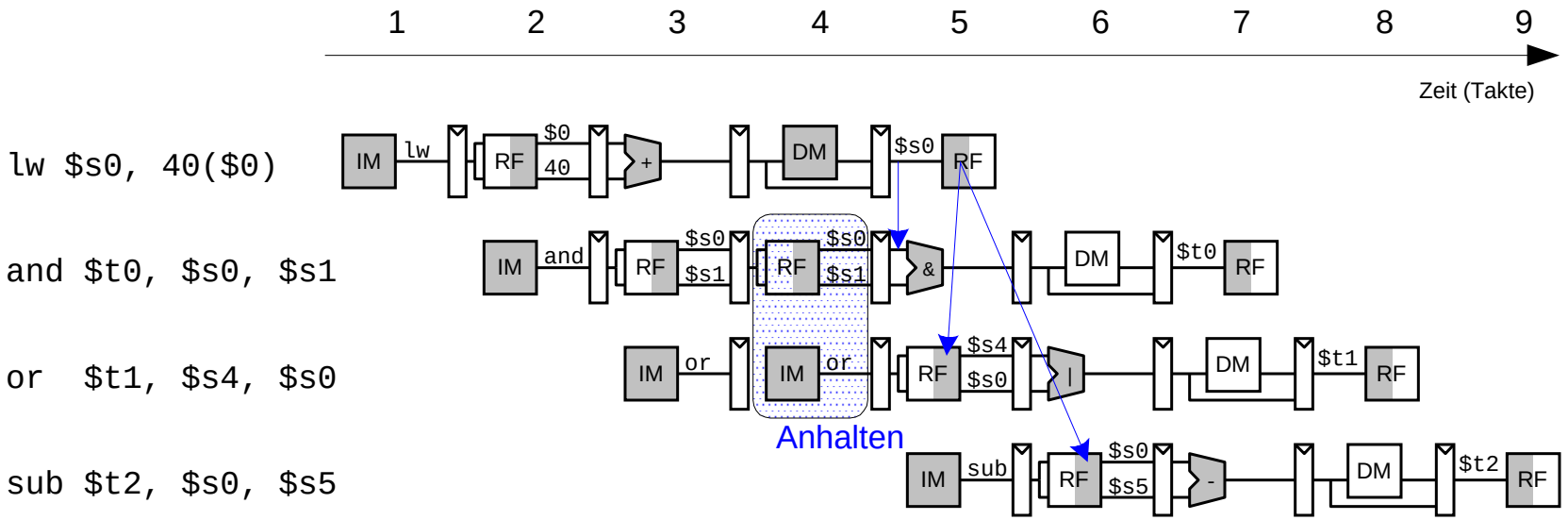
Forwarding-Logik für Signal *ForwardBE* (Weiterleiten von Operand B) analog: ersetze *rsE* durch *rtE*

Embedded Systems & Applications

Anhalten des Prozessors (*stalling*)

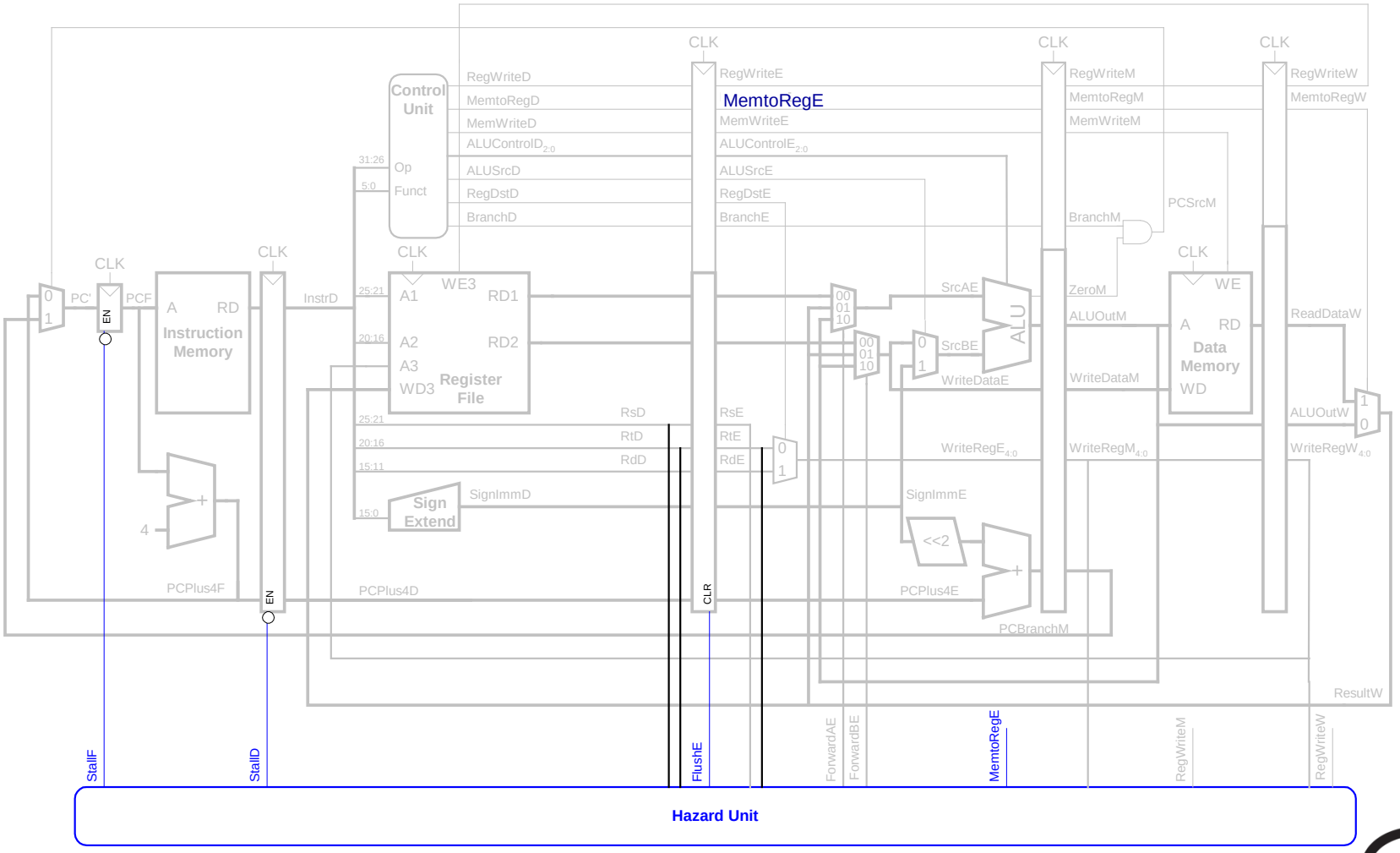


Anhalten des Prozessors (*stalling*)



Erweiterung der Hazard-Einheit für Stalling

TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

Behandlung von Stalling in Hazard-Einheit

TECHNISCHE UNIVERSITÄT DARMSTADT

Stalling-Logik:

```
lwstall = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall
```


Embedded Systems & Applications

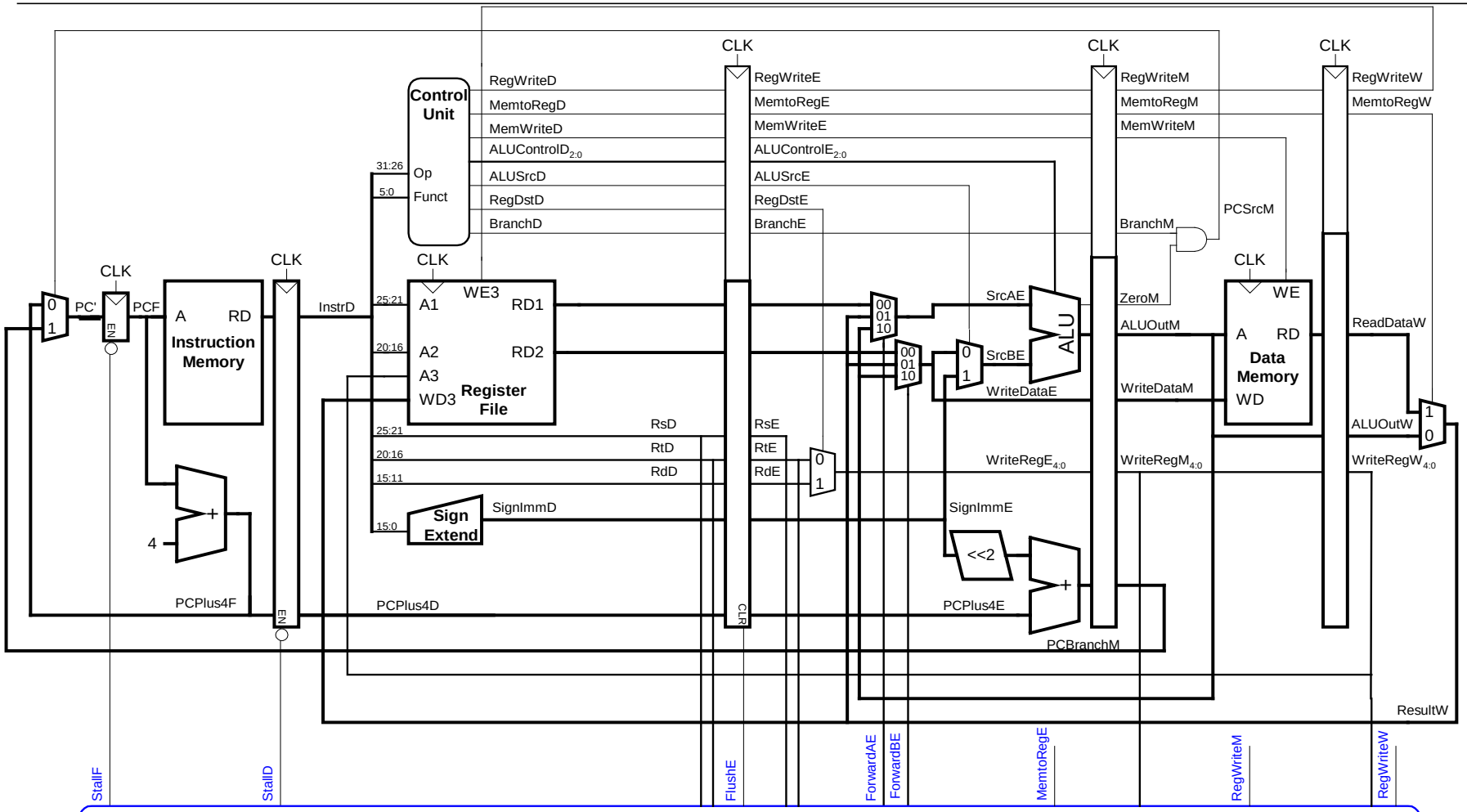
Control Hazards

TECHNISCHE UNIVERSITÄT DARMSTADT

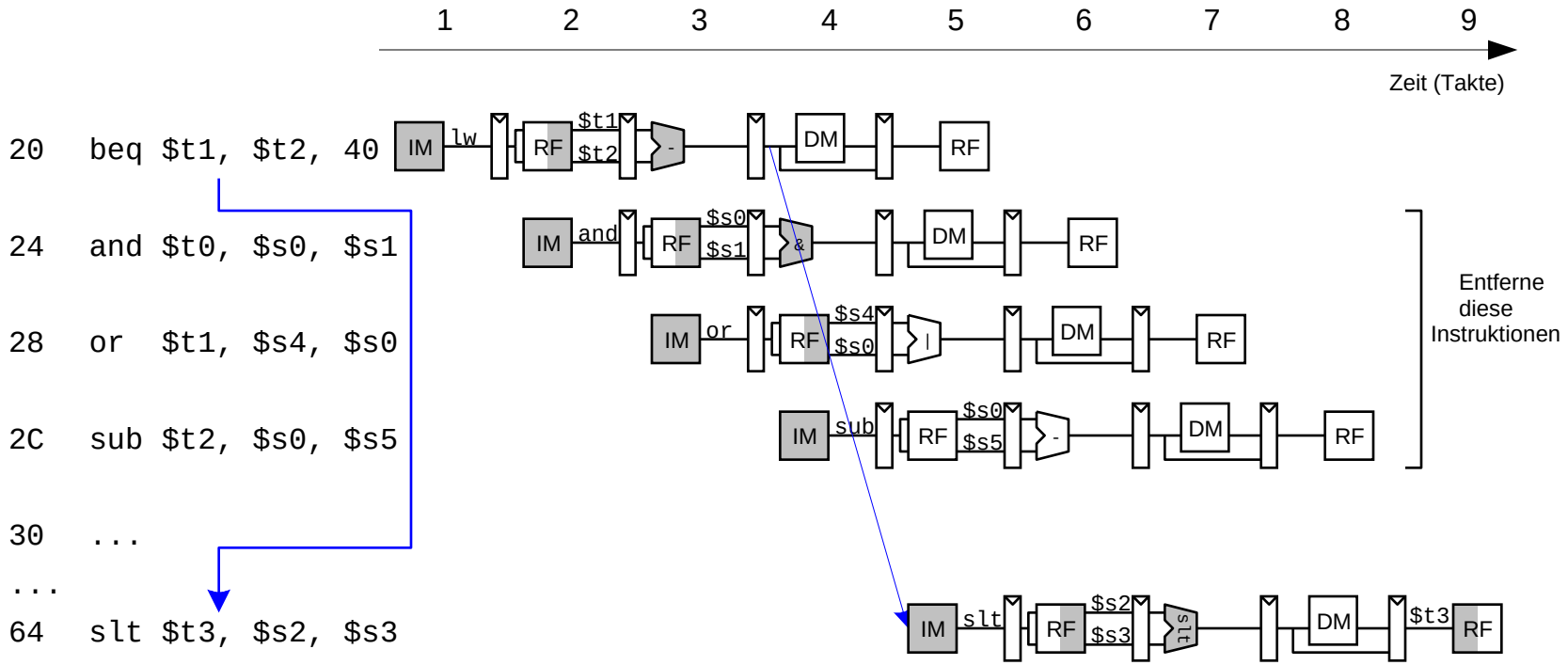
beq:

- Entscheidung zu Springen wird erst in vierter Stufe der Pipeline (M) getroffen
- Neue Instruktionen werden aber bereits geholt
  - Im einfachsten Fall: Von PC+4, +8, +12, …
- Falls zu springen ist, müssen diese Instruktionen aus der Pipeline entfernt werden
  - … das Programm wäre ja woanders (am Sprungziel) weitergegangen
  - "Spülen" (*flush*)
- Kosten eines solchen falsch vorhergesagten Sprunges:
  - Anzahl von zu entfernenden Instruktion falls Sprung genommen
  - Könnte reduziert werden, wenn Sprung in früherer Pipeline-Stufe entschieden würde

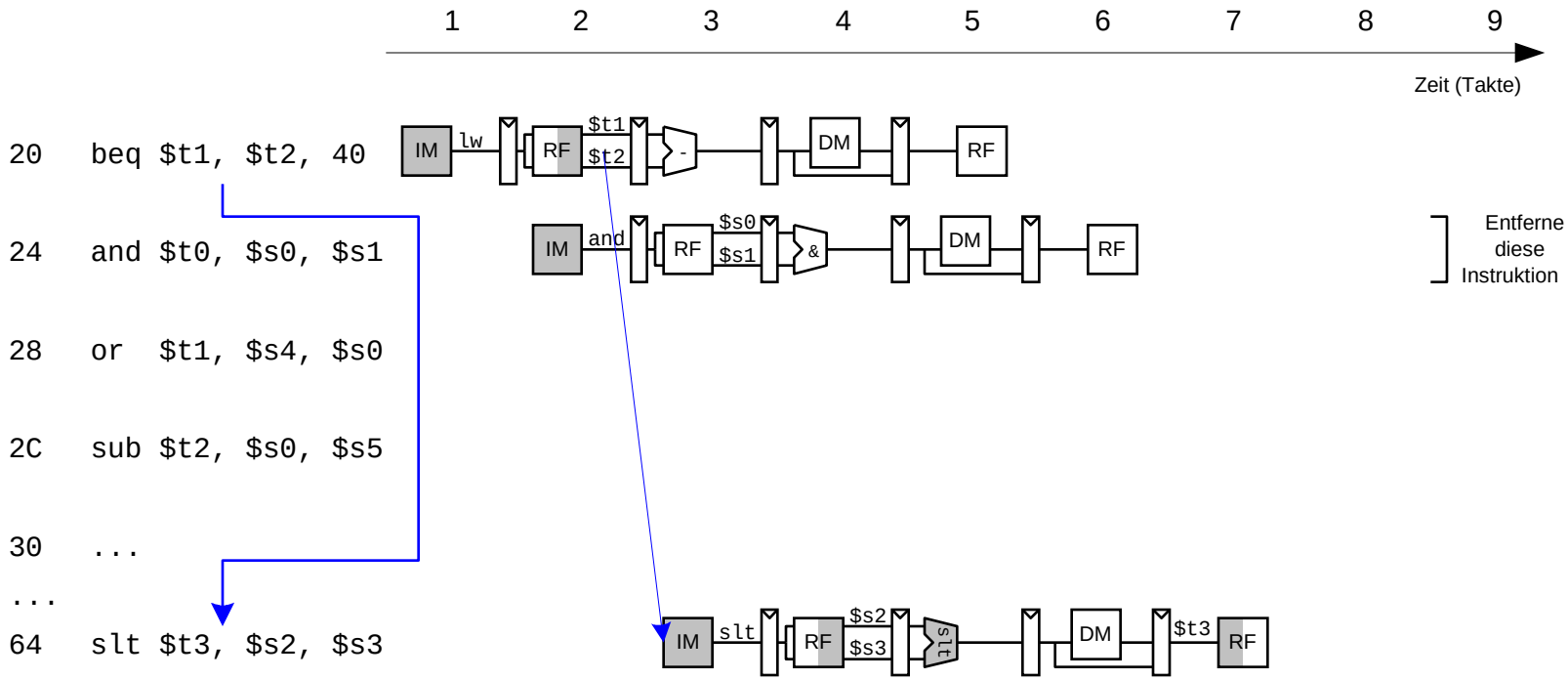
Control Hazards: Ursprüngliche Pipeline



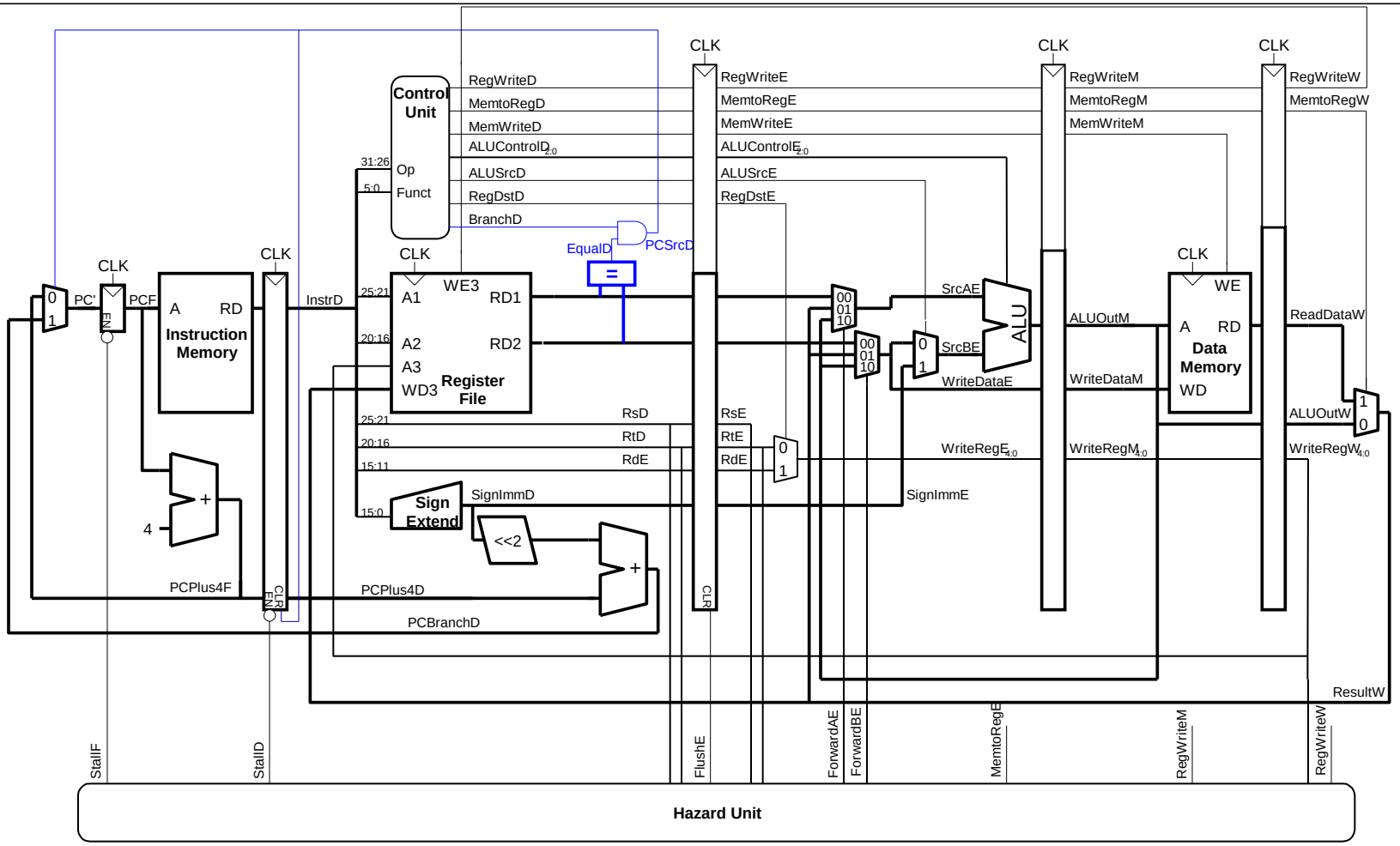
Beispiel: Control Hazards



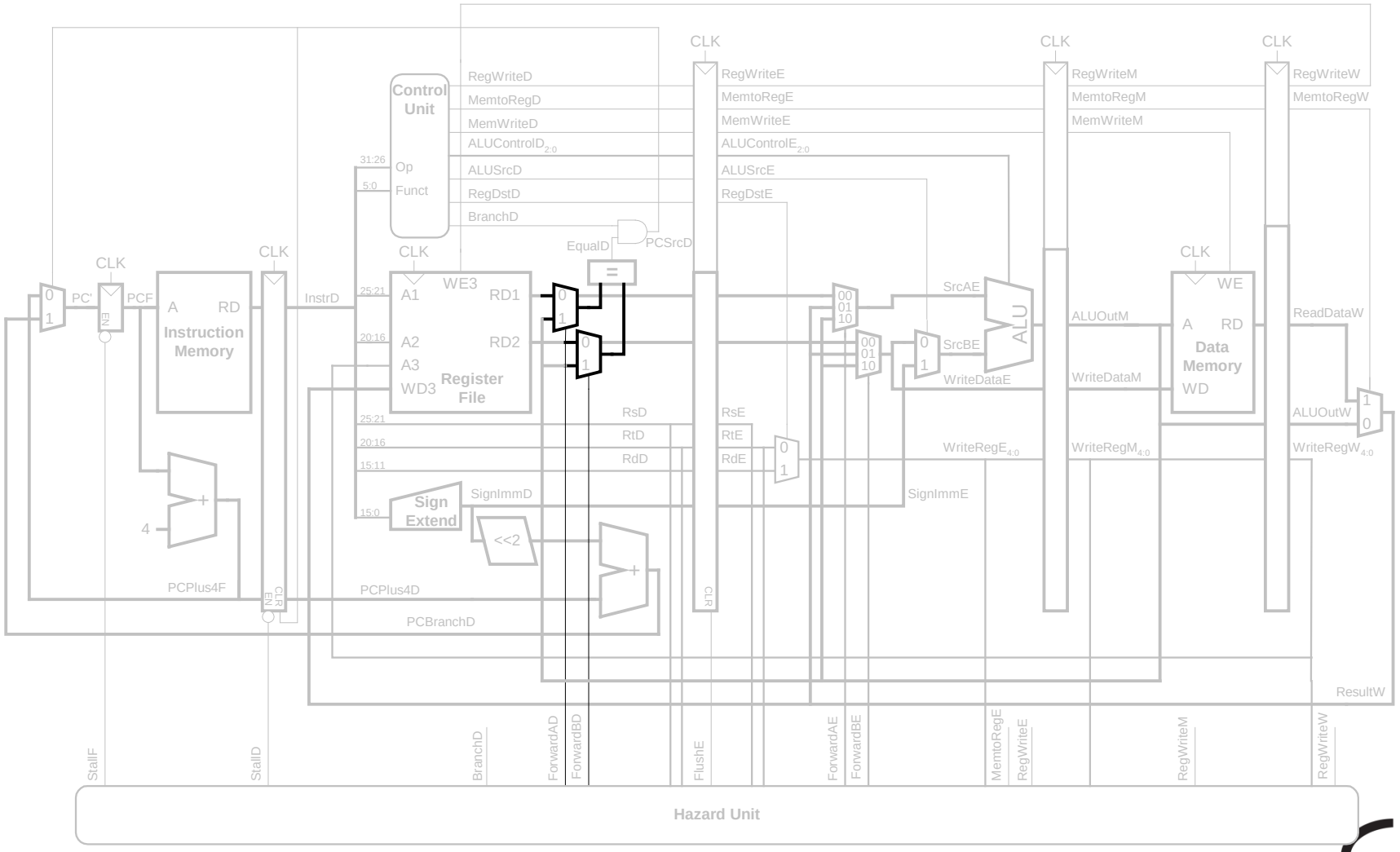
Auflösen von Control Hazards durch frühere Sprungentscheidung



Control Hazards: Ansatz “Frühere Sprungentscheidung”



Berücksichtige neue Data Hazards



Frühe Sprungentscheidung: Benötigte Logik für Forwarding und Stalling

- Forwarding-Logik:

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

- Stalling-Logik:

```
branchstall = BranchD AND RegWriteE AND
                  (WriteRegE == rsD OR WriteRegE == rtD)
                OR
                    BranchD AND MemtoRegM AND
                  (WriteRegM == rsD OR WriteRegM == rtD)

StallF = StallD = FlushE = lwstall OR branchstall
```

Orthogonaler Ansatz: Sprungvorhersage

TECHNISCHE UNIVERSITÄT DARMSTADT

Versuche vorherzusagen, ob ein Sprung genommen wird

- Dann können Instruktionen von der richtigen Stelle geholt werden
- Rückwärtssprünge werden üblicherweise genommen (Schleifen!)
- Genauer: Für jeden Sprung Historie führen, ob er die letzten Male genommen wurde
  - … dann wird jetzt vermutlich auch wieder genommen

Eine gute Vorhersage reduziert die Zahl der Sprünge, die ein Flush der Pipeline erforderlich machen

Embedded Systems & Applications

SystemVerilog

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Hardware-Beschreibungssprachen**
  - ***H**ardware **D**escription **L**anguages (HDL)*
- Erlauben **textuelle** Beschreibung von Schaltungen
  - Auf verschiedenen Abstraktionsebenen
    - **Struktur** (z.B. Verbindungen zwischen Gattern)
    - **Verhalten** (z.B. Boole‘sche Gleichungen)
- **Entwurfswerkzeuge** erzeugen Schaltungsstruktur daraus automatisch
  - Computerprogramme
  - Computer-Aided Design (CAD) oder Electronic Design Automation (EDA)
  - Schaltungssynthese
    - Grob vergleichbar mit Übersetzung (Compilieren) von konventionellen Programmiersprachen

Schreiben von HDL

TECHNISCHE UNIVERSITÄT DARMSTADT

Denken Sie daran:

- Was für Hardware erwarte ich?
- Auf papier skizzieren
- Was für Ergebnisse erwarte ich beim Simulation?
- Auf RTL Schematic nachschauen

SystemVerilog Vorschläge

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Vivado Project Anleitung** steht auf Moodle (unter Lab 2)
- **Die Beispiele** erst angucken und unter Vivado herausprobieren (XSIM Simulator, RTL Schematic)
- **Danach, Lab 2** (komplizierteres System)

Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

Verilog ist im Rahmen der Veranstaltung sehr ähnlich zu SystemVerilog

Verilog ist teilweise komplizierter

- Verwendet separate Datentypen **`wire`** und **`reg`** statt **`logic`** Typ
- Benutzt keine `always` Variationen (stattdessen `always @ …`)
  - **Flip-Flops:** **`always @(posedge clk)`** statt **`always_ff @(posedge clk)`**
  - **Latches:** **`always @(clk, d)`** statt **`always_latch`**
  - **Kombinatorische Logik:** **`always @(*)`** statt **`always_comb`**

Übersicht der Unterschiede im Buch (2. Auflage) im Kapitel 4.7.1

Embedded Systems & Applications

Einleitung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Fast alle kommerziellen Hardware-Entwürfe mit HDLs realisiert

- Zwei HDLs haben sich durchgesetzt
  - SystemVerilog
  - VHDL

SystemVerilog

TECHNISCHE UNIVERSITÄT DARMSTADT

- 1984 von der Firma Gateway Design Automation entwickelt
- Seit 1995 ein IEEE Standard (1364)
  - Überarbeitet 2001 und 2005
  - Neuer Dialekt SystemVerilog (Obermenge von Verilog-2005)
- Weit verbreitet in zivilen US-Firmen
- In Darmstadt im Fachbereich Informatik
  - Rechnerorganisation (ESA, Prof. Koch – Prof Harris, Sommer ‘16)
  - Eingebettete Systeme und ihre Anwendungen (ESA, Prof. Koch)
- In Darmstadt im Fachbereich Elektrotechnik
  - Rechnersysteme (RS, Prof. Hochberger)

VHDL

TECHNISCHE UNIVERSITÄT DARMSTADT

- ***Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language***
- Entwickelt 1981 durch das US Verteidigungsministerium
  - Inspiriert durch konventionelle Programmiersprache Ada
- Standardisiert in 1987 durch IEEE (1076)
  - Überarbeitet in 1993, 2000, 2002, 2006, 2008
- Weit verbreitet in
  - US-Rüstungsfirmen
  - Vielen europäischen Firmen
- In Darmstadt im Fachbereich Elektrotechnik
  - Integrierte elektronische Systeme (IES, Prof. Hofmann)

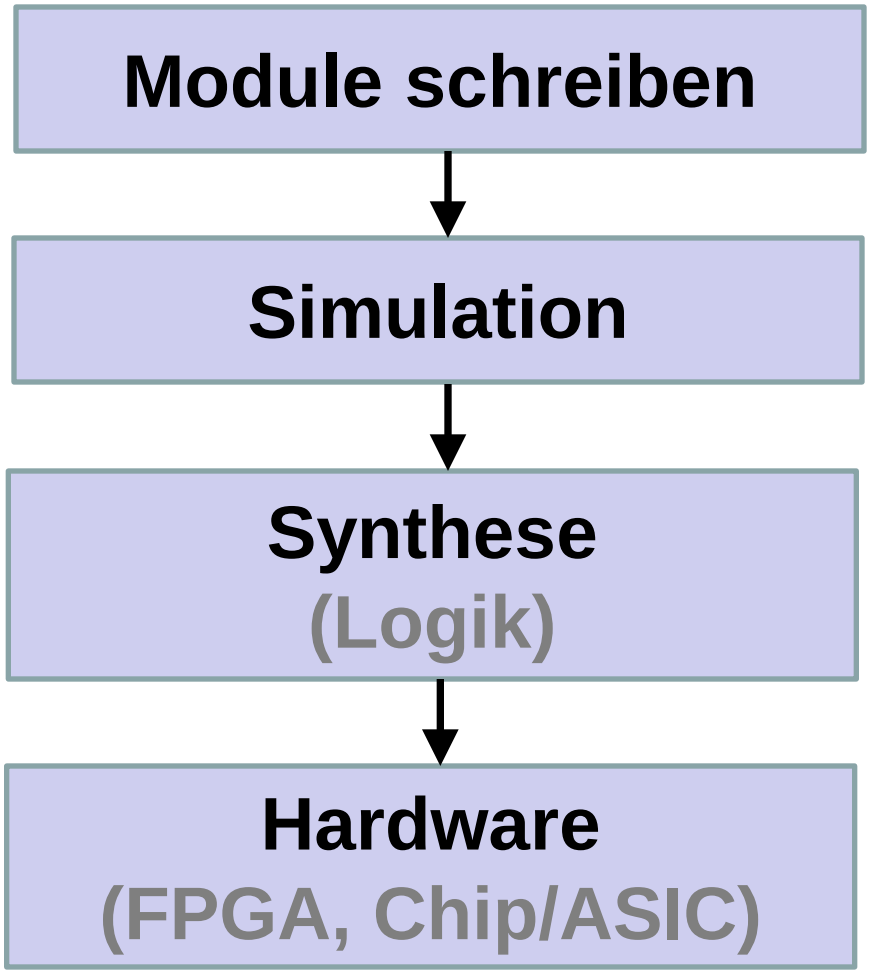
In dieser Iteration der Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

- In den Vorlesungen SystemVerilog
  - Häufig kompakter zu schreiben
  - Eher auf Einzelfolien darstellbar
- Hier gezeigte Grundkonzepte sind in beiden Sprachen identisch
- Nur andere Syntax
  - VHDL-Beschreibung ist aber in der Regel länger
- Im Buch werden beide Sprachen nebeneinander gezeigt
  - Kapitel 4
  - Moderne Entwurfswerkzeuge können in der Regel beide Sprachen

Von einer HDL zu Logikgattern

TECHNISCHE UNIVERSITÄT DARMSTADT



Von einer HDL zu Logikgattern

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Simulation**
  - Eingangswerte werden in HDL-Beschreibung eingegeben
    - Beschriebene Schaltung wird stimuliert
  - Berechnete Ausgangswerte werden auf Korrektheit geprüft
  - Fehlersuche viel einfacher und billiger als in realer Hardware
- **Synthese**
  - Übersetzt HDL-Beschreibungen in Netzlisten
    - **Logikgatter** (Schaltungselemente)
    - **Verbindungen** (Verbindungsknoten)

Von einer HDL zu Logikgattern

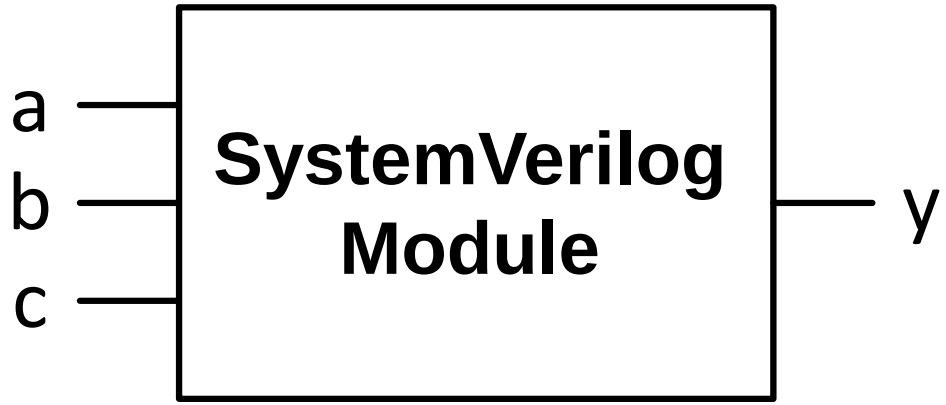
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Simulation**
  - Eingangswerte werden in HDL-Beschreibung eingegeben
    - Beschriebene Schaltung wird stimuliert
  - Berechnete Ausgangswerte werden auf Korrektheit geprüft
  - Fehlersuche viel einfacher und billiger als in realer Hardware
- **Synthese**
  - Übersetzt HDL-Beschreibungen in Netzlisten
    - **Logikgatter** (Schaltungselemente)
    - **Verbindungen** (Verbindungsknoten)

WICHTIG:

Beim Verfassen von HDL-Beschreibungen ist es essentiell wichtig,
immer die vom Programm beschriebene **Hardware** im Auge zu behalten!

SystemVerilog Module



Zwei Arten von Beschreibungen in Modulen:

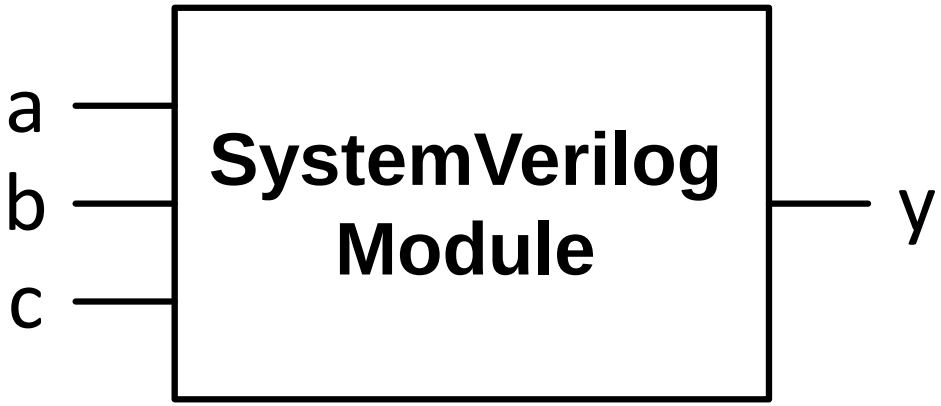
- **Verhalten:** Was tut die Schaltung?
- **Struktur:** Wie ist die Schaltung aus Untermodulen aufgebaut?

Beispiel für Verhaltensbeschreibung

TECHNISCHE UNIVERSITÄT DARMSTADT

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```



Embedded Systems & Applications

Beispiel für Verhaltensbeschreibung

TECHNISCHE UNIVERSITÄT DARMSTADT

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

- `module/endmodule`: Anfang/Ende der Module
- `example`: Namen der Module
- Operatoren:
  - ~: NOT (nicht)
  - &: AND (und)
  - |: OR (oder)

Embedded Systems & Applications

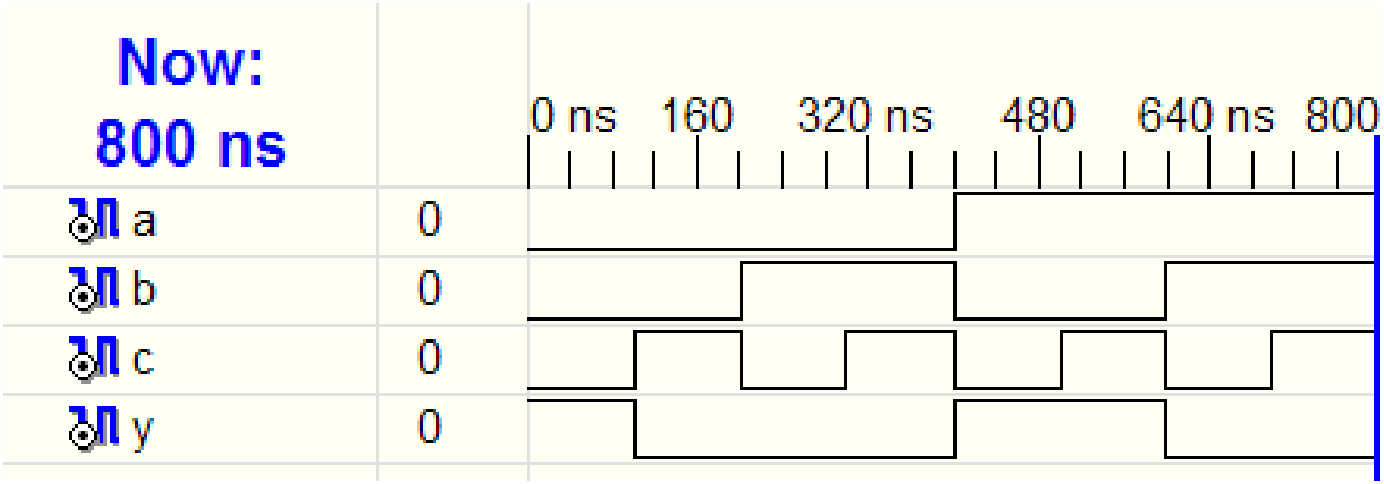
Simulation von Verhaltensbeschreibungen

TECHNISCHE UNIVERSITÄT DARMSTADT

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Signalverlaufsdiagramm (*waves*):



Embedded Systems & Applications

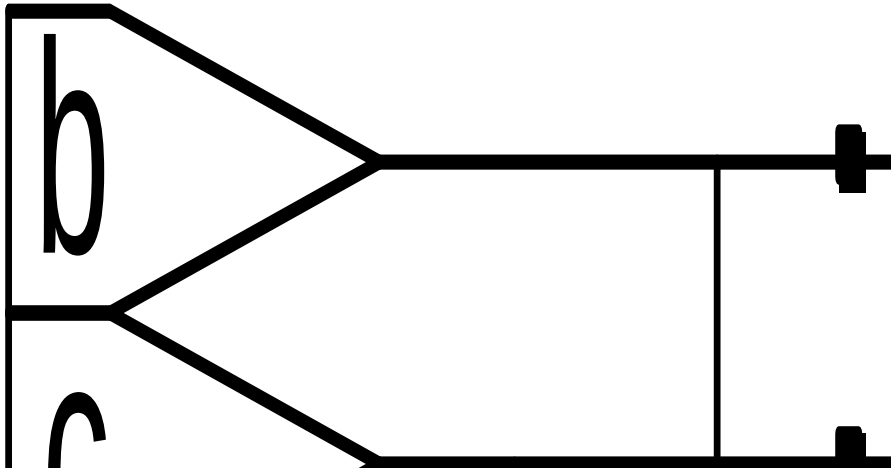
Synthese von Verhaltensbeschreibungen

TECHNISCHE UNIVERSITÄT DARMSTADT

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

SystemVerilog Syntax

TECHNISCHE UNIVERSITÄT DARMSTADT

- Unterscheidet Groß- und Kleinschreibung
  - **Beispiel:** `reset` und `Reset` sind nicht das gleiche Signal
- Namen dürfen nicht mit Ziffern anfangen
  - **Beispiel:** `2mux` ist ein ungültiger Name
- Anzahl von Leerzeichen, Leerzeilen und Tabulatoren irrelevant
- Kommentare:

```
// bis zum Ende der Zeile
/* über mehrere
   Zeilen */
```

SystemVerilog Syntax

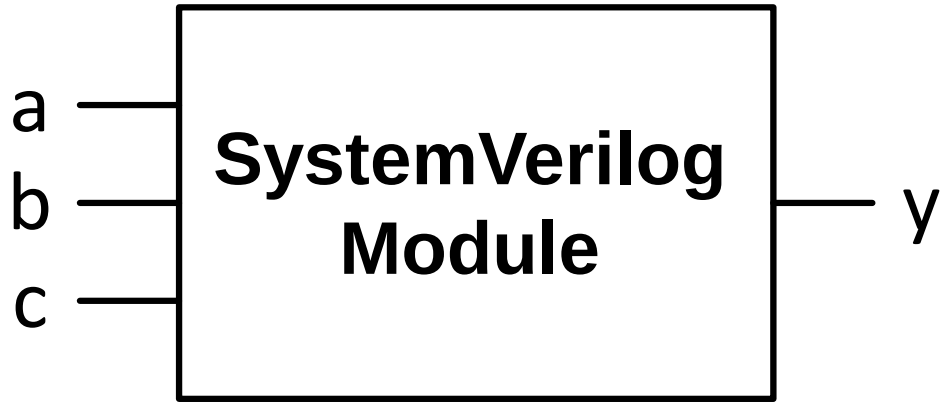
TECHNISCHE UNIVERSITÄT DARMSTADT

- Unterscheidet Groß- und Kleinschreibung
  - **Beispiel:** `reset` und `Reset` sind nicht das gleiche Signal
- Namen dürfen nicht mit Ziffern anfangen
  - **Beispiel:** `2mux` ist ein ungültiger Name
- Anzahl von Leerzeichen, Leerzeilen und Tabulatoren irrelevant
- Kommentare:

```
// bis zum Ende der Zeile
/* über mehrere
   Zeilen */
```

Sehr ähnlich zu C und Java!

SystemVerilog Module



Zwei Arten von Beschreibungen in Modulen:

- **Verhalten:** Was tut die Schaltung?
- **Struktur:** Wie ist die Schaltung aus Untermodulen aufgebaut?

Strukturelle Beschreibung: Modulhierarchie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule


module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule

module nand3 (input  logic a, b, c
              output logic y);
  logic n1;  // internes Signal(Verbindungsknoten)

  and3 andgate(a, b, c, n1); // Instanz von and3 namens andgate
  inv  inverter(n1, y);      // Instanz von inv namens inverter
endmodule
```


Embedded Systems & Applications

Strukturelle Beschreibung: Modulhierarchie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule


module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule

module nand3 (input  logic d, e, f
              output logic w);
  logic n1;  // internes Signal(Verbindungsknoten)

  and3 andgate(d, e, f, n1); // Instanz von and3 namens andgate
  inv  inverter(n1, w);      // Instanz von inv namens inverter
endmodule
```


Embedded Systems & Applications

Strukturelle Beschreibung: Modulhierarchie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module nand3 (input  logic d, e, f
              output logic w);
  logic n1;  // internes Signal(Verbindungsknoten)

  and3 andgate(d, e, f, n1); // Instanz von and3 namens andgate
  inv  inverter(n1, w);      // Instanz von inv namens inverter
endmodule


module nand3 (input  logic d, e, f
              output logic w);
  logic n1;

  and3 andgate(.a(d), .b(e), .c(f), .y(n1));
  inv  inverter(.a(n1), .y(w));
endmodule
```

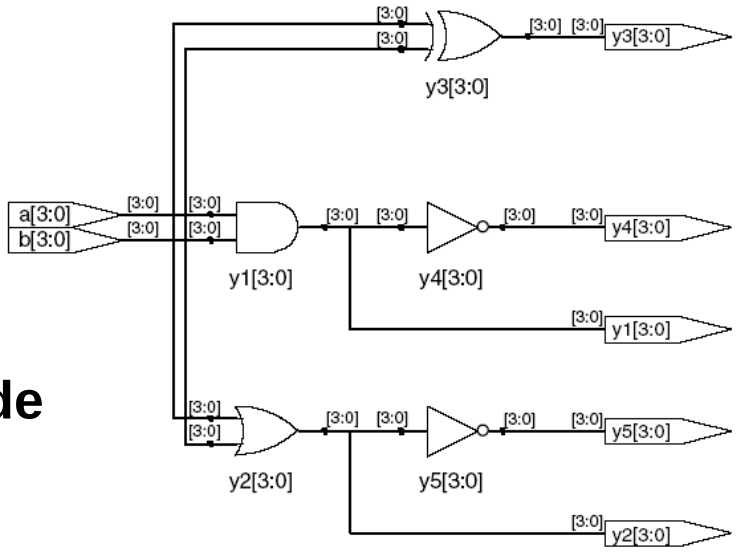

Embedded Systems & Applications

Bitweise Verknüpfungsoperatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module gates (input  logic [3:0] a, b,
              output logic [3:0] y1, y2, y3, y4, y5);
   /* Fünf unterschiedliche Logikgatter
      mit zwei Eingängen, jeweils 4b Busse */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

Syntheseergebnis:



`//` **Kommentar bis zum Zeilenende**

`/*…*/` **Mehrzeiliger Kommentar**

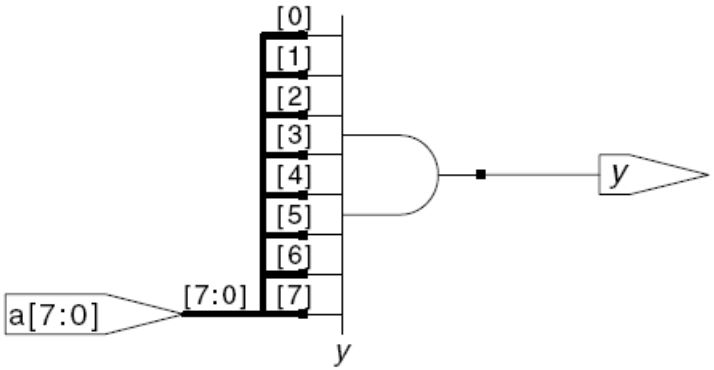
Embedded Systems & Applications

Reduktionsoperatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module and8 (input  logic [7:0] a,
             output logic       y);
   assign y = &a;
   // &a ist Abkürzung für
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```

Syntheseergebnis:

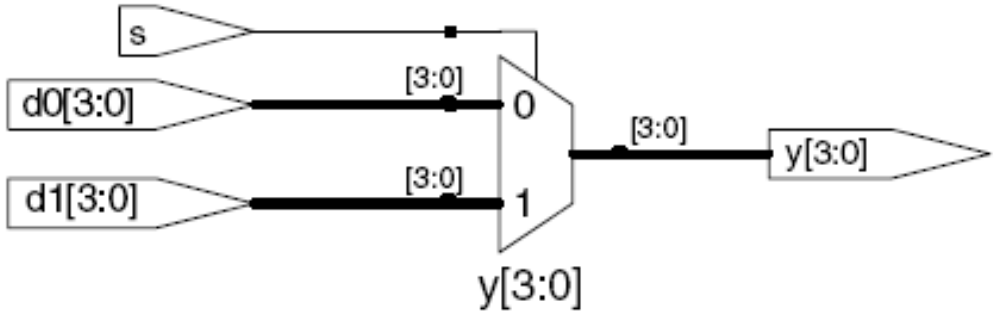


Embedded Systems & Applications

Bedingte Zuweisung

```
module mux2(input  logic[3:0]  d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```

Syntheseergebnis:



? : ist ein ternärer Operator, da er drei Operanden miteinander verknüpft: `s`, `d1`, und `d0`.

Interne Verbindungsknoten oder Signale

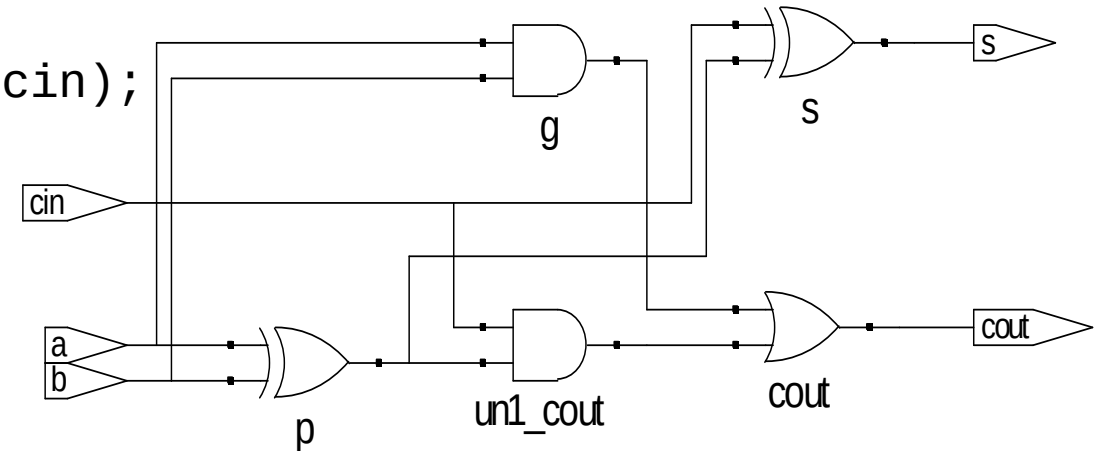
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;    // interne Verbindungsknoten

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

Bindung von Operatoren (Präzedenz)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bestimmt Ausführungsreihenfolge

Höchste

`~`	NOT
`*, /, %`	Multiplikation, Division, Modulo
`+, -`	Addition, Subtraktion
`<<, >>`	Schieben (logisch)
`<<<, >>>`	Schieben (arithmetisch)
`<, <=, >, >=`	Vergleiche
`==, !=`	gleich, ungleich
`&, ~&`	AND, NAND
`^, ~^`	XOR, XNOR
`|, ~|`	OR, NOR
`?:`	Ternärer Operator

Niedrigste

Embedded Systems & Applications

Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntax: ***N'Bwert***

N = Breite in Bits, **B** = Basis

N'B ist optional, sollte der Konsistenz halber aber immer geschrieben werden

wenn weggelassen: Dezimalsystem

Zahl	Bitbreite	Basis	entspricht Dezimal	Darstellung im Speicher
3'b101	3	binär	5	101
'b11	Nicht vorgegeben	binär	3	00…0011
8'b11	8	binär	3	00000011
8'b1010_1011	8	binär	171	10101011
3'd6	3	dezimal	6	110
6'o42	6	oktal	34	100010
8'hAB	8	hexadezimal	171	10101011
42	Nicht vorgegeben	dezimal	42	00…0101010

Embedded Systems & Applications

Operationen auf Bit-Ebene: Beispiel 1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// wenn y ein 12-bit Signal ist, hat die
// Anweisung diesen Effekt:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

Unterstriche (_) in numerischen Konstanten dienen nur der besseren Lesbarkeit, sie werden von SystemVerilog ignoriert

Operationen auf Bit-Ebene: Beispiel 2

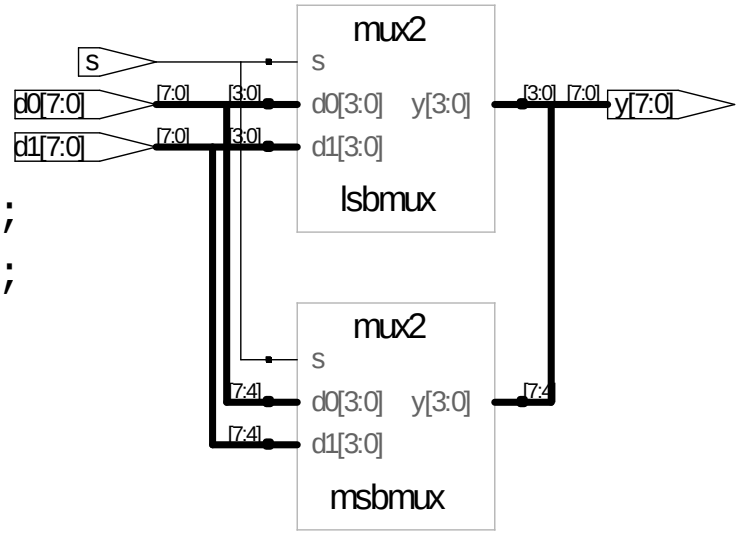
TECHNISCHE UNIVERSITÄT DARMSTADT

SystemVerilog:

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

Syntheseergebnis:



Benutzt mux2 module von Folie 26.

Embedded Systems & Applications

Operationen auf Bit-Ebene: Beispiel 2

TECHNISCHE UNIVERSITÄT DARMSTADT

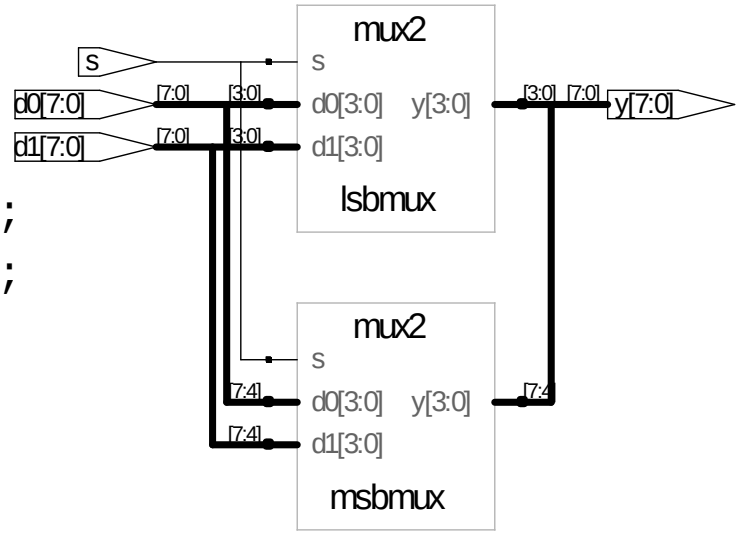
SystemVerilog:

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule


...
  mux2 lsbmux(.d0(d0[3:0]), .d1(d1[3:0]),
              .s(s), .y(y[3:0]));
  mux2 msbmux(.d0(d0[7:4]), .d1(d1[7:4]),
              .s(s), .y(y[7:4]));
...
```

Syntheseergebnis:



Benutzt mux2 module von Folie 26.

Embedded Systems & Applications

Operationen auf Bit-Ebene: Beispiel 2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);
  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);
  mux2 lsbmux(.d0(d0[3:0]), .d1(d1[3:0]),
              .s(s), .y(y[3:0]));
  mux2 msbmux(.d0(d0[7:4]), .d1(d1[7:4]),
              .s(s), .y(y[7:4]));
endmodule
```

```
module mux2(input  logic[3:0]  d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```

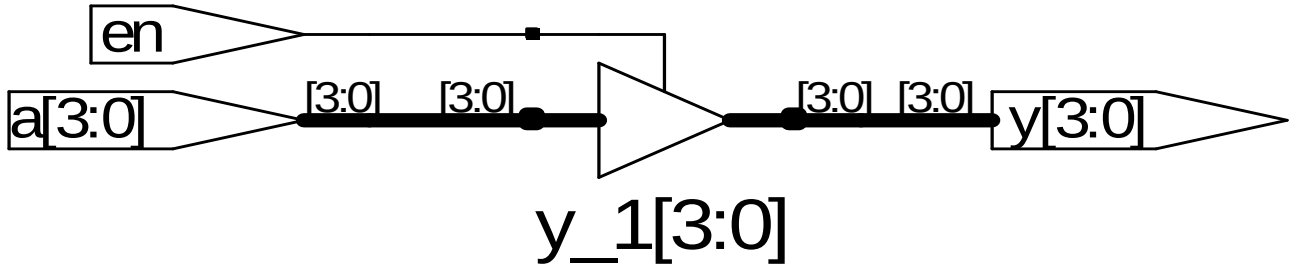
Hochohmiger Ausgang: Z

TECHNISCHE UNIVERSITÄT DARMSTADT

SystemVerilog:

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output logic [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

Syntheseergebnis:

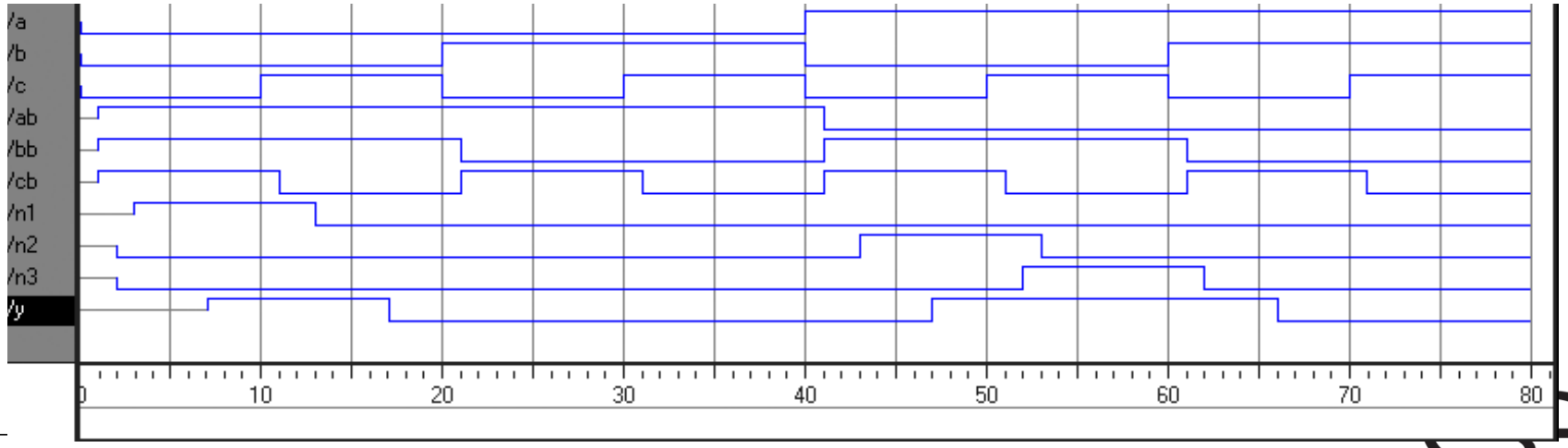


Embedded Systems & Applications

Verzögerungen: # *Zeiteinheiten*

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



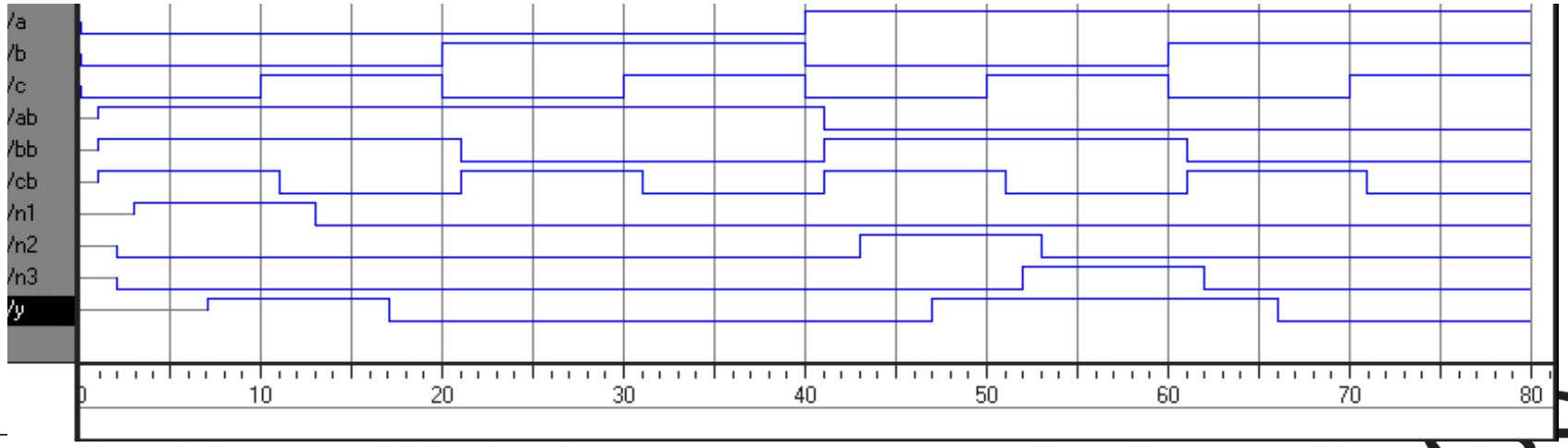
Embedded Systems & Applications

Verzögerungen: # *Zeiteinheiten*

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

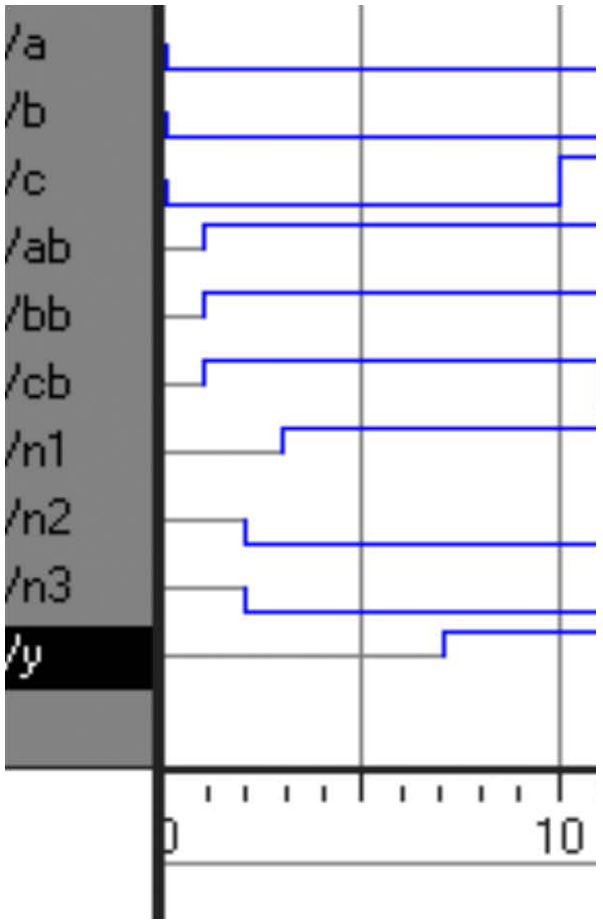
Nur für die Simulation, *#n* werden für die Synthese **ignoriert**!



Embedded Systems & Applications

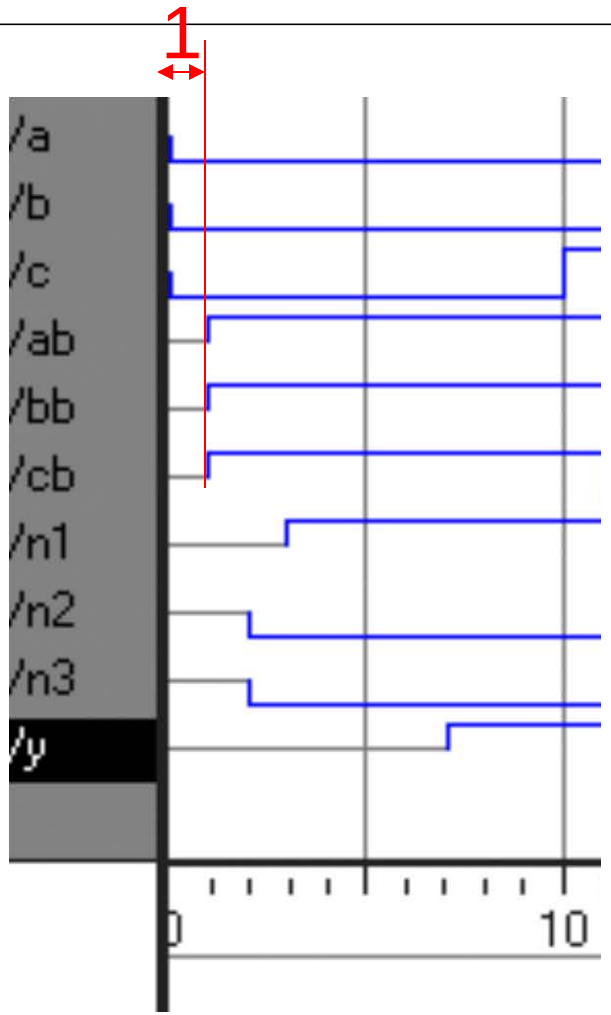
Verzögerungen

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



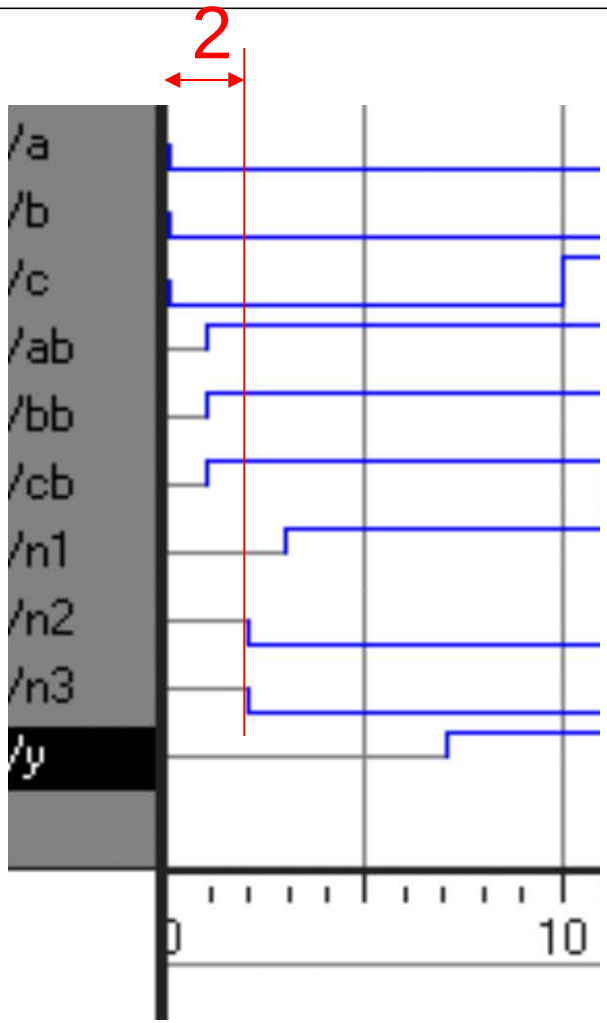
Verzögerungen

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



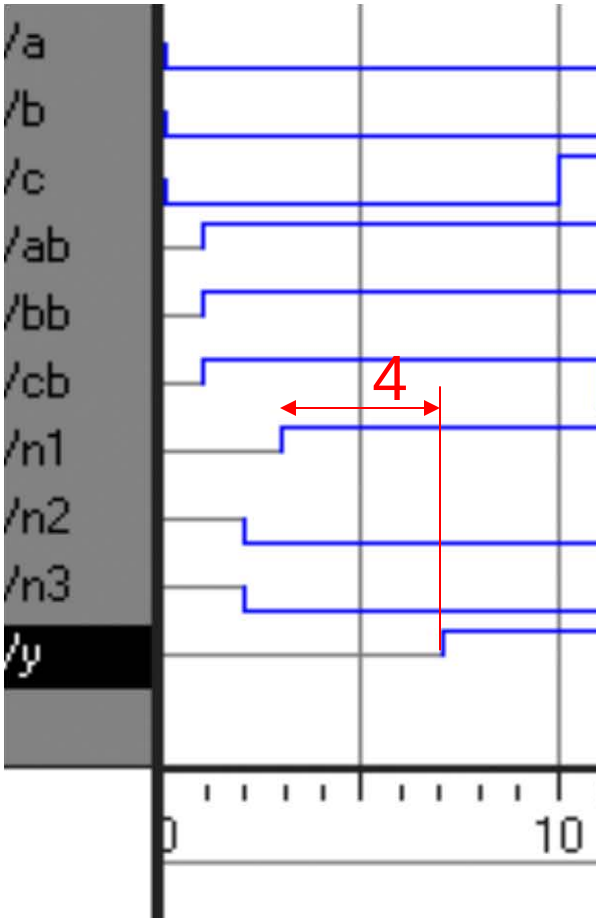
Verzögerungen

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Verzögerungen

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Sequentielle Schaltungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschreibung basiert auf Verwendung fester "Redewendungen"
  - Idiome
- Feststehende Idiome für
  - Latches
  - Flip-Flops
  - Endliche Zustandsautomaten (FSM)
- Vorsicht beim Abweichen von Idiomen
  - Wird möglicherweise noch richtig simuliert
  - Könnte aber fehlerhaft synthetisiert werden

Sequentielle Schaltungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschreibung basiert auf Verwendung fester "Redewendungen"
  - Idiome
- Feststehende Idiome für
  - Latches
  - Flip-Flops
  - Endliche Zustandsautomaten (FSM)
- Vorsicht beim Abweichen von Idiomen
  - Wird möglicherweise noch richtig simuliert
  - Könnte aber fehlerhaft synthetisiert werden

➔ Halten Sie sich an die Konventionen!

always-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

Allgemeiner Aufbau:

```
always @(sensitivity list)
  statement;
```

Interpretation:

Wenn sich die in der `sensitivity list` aufgezählten Werte ändern, wird die Anweisung `statement` ausgeführt.

Werte: In der Regel Signale, manchmal noch erweitert

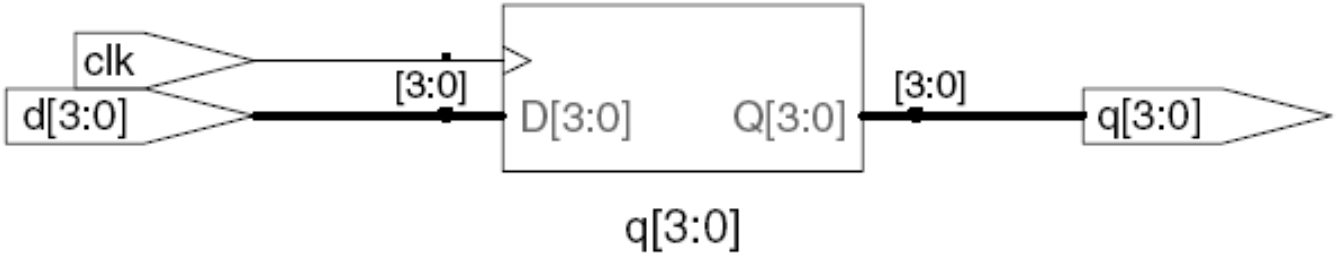
D Flip-Flop

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;  // gelesen als “q übernimmt d”
             // auf Englisch: “q gets d”
endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

Rücksetzbares D Flip-Flop

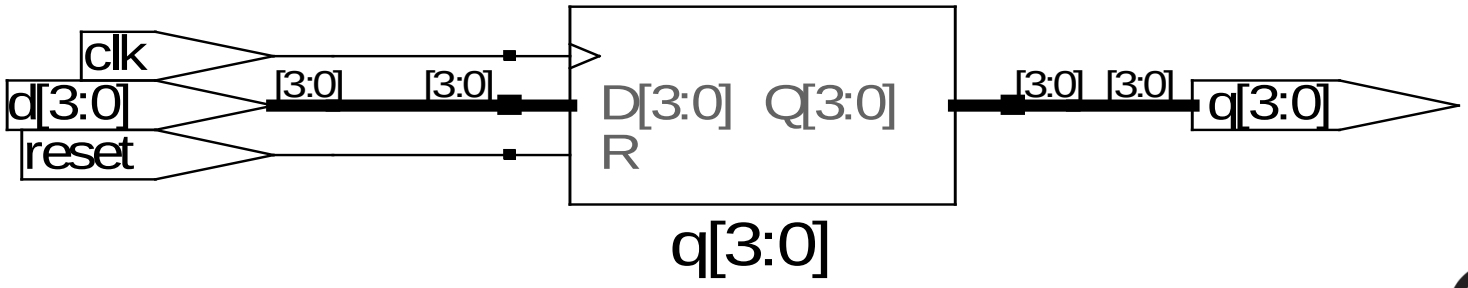
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchroner Reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

Rücksetzbares D Flip-Flop

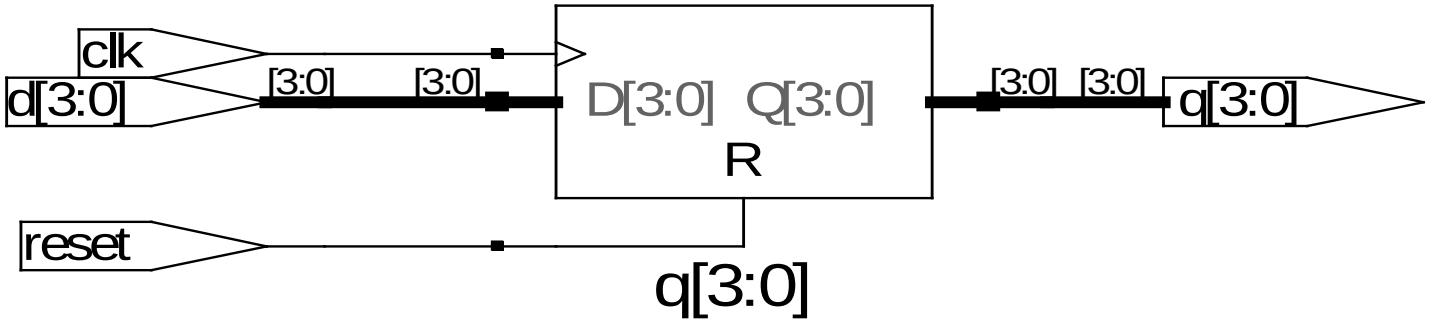
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchroner Reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

Rücksetzbares D Flip-Flop mit Taktfreigabe

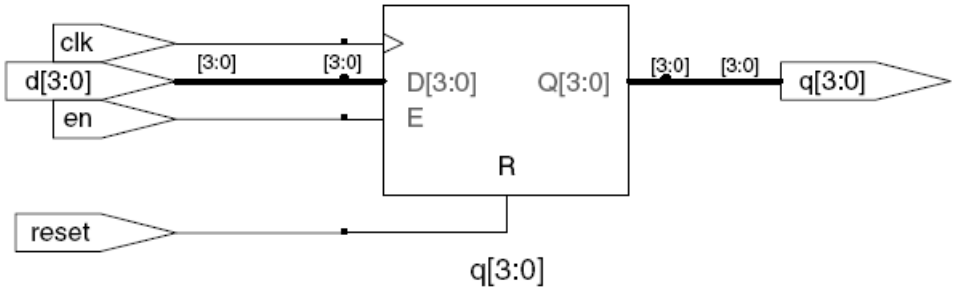
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module flopren(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchroner Reset mit Clock Enable
  always @ (posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```

Syntheseergebnis:



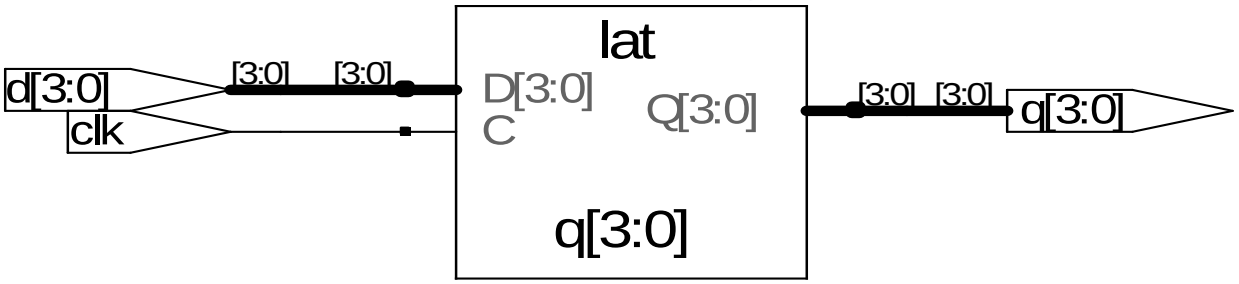
Embedded Systems & Applications

Latch

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

 always_latch
    if (clk) q <= d;
endmodule
```

Syntheseergebnis:



Achtung: In dieser Veranstaltung werden Latches nur selten (wenn überhaupt) gebraucht werden.

Sollten sie dennoch in einem Syntheseergebnis auftauchen, ist das in der Regel auf Fehler in Ihrer HDL-Beschreibung zurückzuführen (z.B. Abweichen von Idiomen)!

Wiederholung

TECHNISCHE UNIVERSITÄT DARMSTADT

Allgemeiner Aufbau:

```
always @(sensitivity list)
  statement;
```

- **Flip-flop:** `always_ff`
- **Latch:** `always_latch` **(nicht benutzen)**

Weitere Anweisungen zur Verhaltensbeschreibung

- Dürfen nur innerhalb von `always`-Anweisungen benutzt werden
  - `if` / `else`
  - `case`, `casez`

Kombinatorische Logik als `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

  always_comb // wann immer sich irgendein gelesenes Signal ändert
    begin     // bei mehr als einer Anweisung: begin/end
      y1 = a & b;          // AND
      y2 = a | b;          // OR
      y3 = a ^ b;          // XOR
      y4 = ~(a & b);       // NAND
      y5 = ~(a | b);       // NOR
    end
endmodule
```

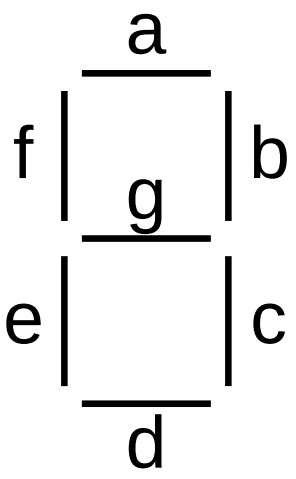
Hätte einfacher durch fünf `assign`-Anweisungen beschrieben werden können.

Kombinatorische Logik mit `case`

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  ...

endmodule
```

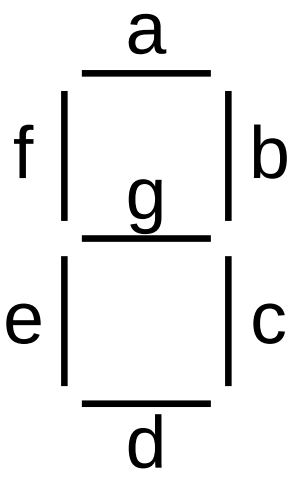


Kombinatorische Logik mit `case`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  always_comb  // kombinatorische Logik …
    case (data)
      //               abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000; // alle Fälle abgedeckt!
    endcase
endmodule
```



Embedded Systems & Applications

Kombinatorische Logik mit `case`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  always_comb  // kombinatorische Logik …
    case (data)
      //               abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000; // alle Fälle abgedeckt!
    endcase
endmodule
```

So einfach nicht als `assign` formulierbar

Embedded Systems & Applications

Kombinatorische Logik mit `case`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Um kombinatorische Logik zu beschreiben, muss ein `case`-Block **alle** Möglichkeiten abdecken
  - Entweder **explizit** angeben
  - Oder einen **`default`-Fall** angeben
    - Tritt in Kraft, wenn sonst keine andere Alternative passt
    - Im Beispiel verwendet

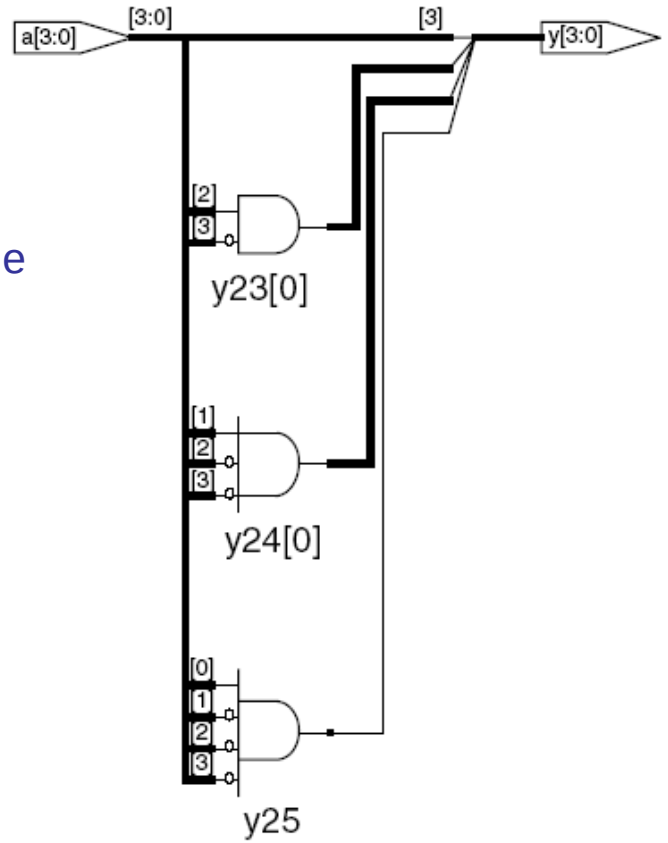
Kombinatorische Logik mit casez

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);

  always_comb // kombinatorische Logik …
    casez(a)
      4'b1???: y = 4'b1000;  // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;  // alle Fälle
                             // abgedeckt
   endcase
endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

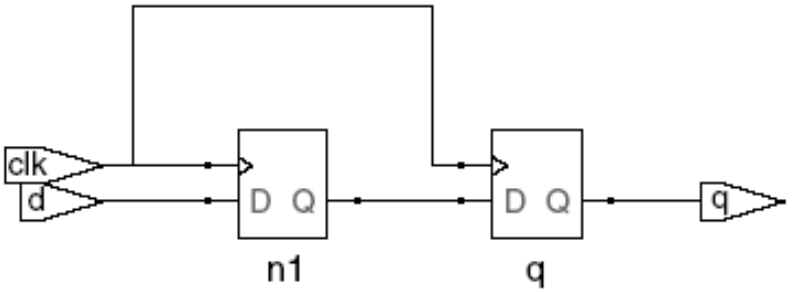
Nicht-blockende Zuweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

- <= steht für eine "**nicht-blockende** Zuweisung"
- Wird parallel mit allen anderen nicht-blockenden Zuweisungen ausgeführt
  - 1. Schritt: Alle „rechten Seiten" werden berechnet
  - 2. Schritt: Alle Berechnungsergebnisse werden an „linke Seiten" zugewiesen

```
// Synchronisierer mit nicht-blockenden
// Zuweisungen
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff(posedge clk)
    begin
      n1 <= d;  // nicht-blockend
      q  <= n1; // nicht-blockend
    end
endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

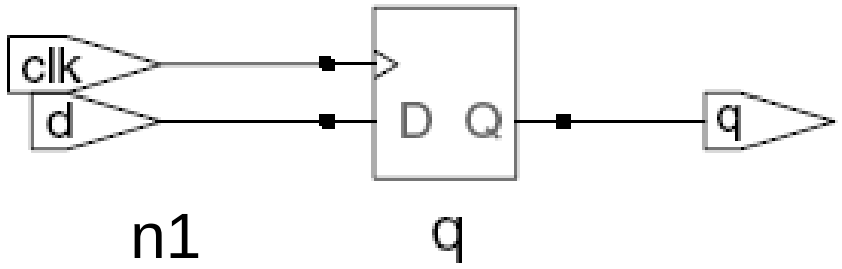
Blockende Zuweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

- = steht für eine "**blockende** Zuweisung"
- Wird hintereinander (seriell) in Reihenfolge im Programmtext ausgeführt
  - Solange eine blockende Zuweisung abläuft
  - … werden andere Anweisungen blockiert
  - Jede Anweisung für sich berechnet „rechte Seite" und weist an „linke Seite" zu

```
// Fehlerhafter Synchronisierer
// mit blockenden Zuweisungen
module syncbad(input  logic clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff(posedge clk)
    begin
      n1 = d;  // blockend
      q  = n1; // blockend
    end
endmodule
```

Syntheseergebnis:



Embedded Systems & Applications

Regeln für Zuweisungen von Signalen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Um **synchrone sequentielle** Logik zu beschreiben, benutzen Sie immer **`always_ff @(posedge clk)`** und nicht-blockende Zuweisungen (**`<=`**)

```
always_ff @(posedge clk)
  q <= d; // nicht-blockend
```


Embedded Systems & Applications

Regeln für Zuweisungen von Signalen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Um **synchrone sequentielle** Logik zu beschreiben, benutzen Sie immer **`always_ff @(posedge clk)`** und nicht-blockende Zuweisungen (**`<=`**)

```
always_ff @(posedge clk)
  q <= d; // nicht-blockend
```

- Um **einfache kombinatorische** Logik zu beschreiben, benutzen Sie immer ständige Zuweisung (*continuous assignment*) (**`assign ...`**)

```
assign y = a & b;
```

Regeln für Zuweisungen von Signalen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Um **synchrone sequentielle** Logik zu beschreiben, benutzen Sie immer **`always_ff @(posedge clk)`** und nicht-blockende Zuweisungen (**`<=`**)

```
always_ff @(posedge clk)
  q <= d; // nicht-blockend
```

- Um **einfache kombinatorische** Logik zu beschreiben, benutzen Sie immer ständige Zuweisung (*continuous assignment*) (**`assign ...`**)

```
assign y = a & b;
```

- Um **komplexere kombinatorische** Logik zu beschreiben, benutzen Sie immer **`always_comb`** und blockende Zuweisungen (**`=`**)

Regeln für Zuweisungen von Signalen

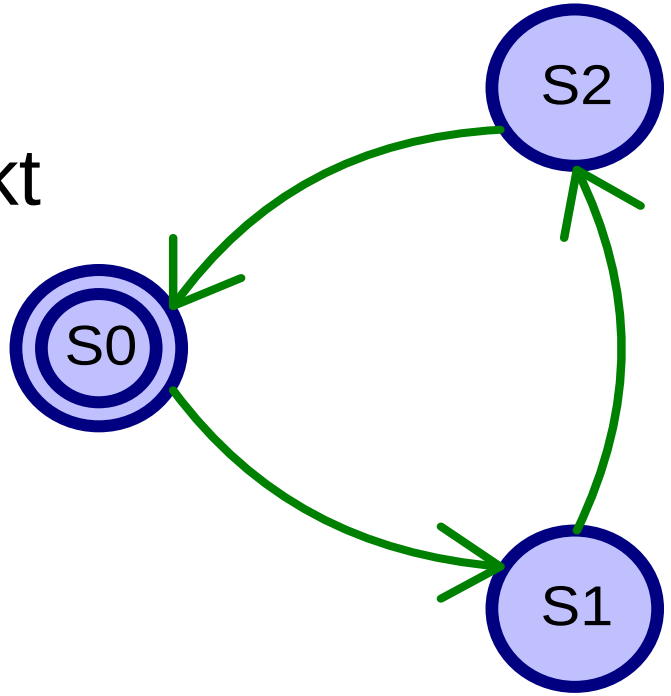
TECHNISCHE UNIVERSITÄT DARMSTADT

- Weisen Sie **nicht** an ein Signal
  - … in mehreren `always`-Blöcken zu
  - … in einem `always`-Block gemischt mit = und <= zu
  - ... in einem `always`-Block **und** in einer ständigen Zuweisung (*continuous assignment*) (**`assign ...`**)

Beispiel-FSM: Dritteln der Taktfrequenz

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Eingabe:**
  - Explizit kein Signal
  - Implizit den Schaltungstakt
    - Mit Frequenz *f*
- **Ausgabe:**
  - Signal q mit Frequenz *f*/3



Hier: alternative Schreibweise für Resetzustand (doppelter Kreis)

Embedded Systems & Applications

FSM in Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   always_ff @(posedge clk, posedge reset) // Zustandsregister
      if (reset) state <= S0;
      else       state <= nextstate;
   always_comb                             // Zustandsübergangslogik
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);               // Ausgangslogik
endmodule
```

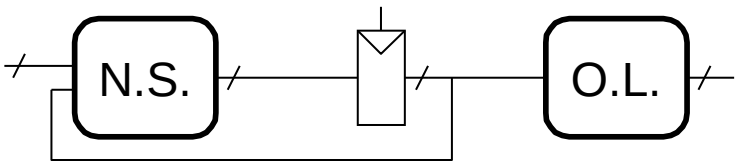
FSM in Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   always_ff @(posedge clk, posedge reset) // Zustandsregister
      if (reset) state <= S0;
      else       state <= nextstate;
   always_comb                             // Zustandsübergangslogik
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);               // Ausgangslogik
endmodule
```



Embedded Systems & Applications

FSM in Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic q);

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   logic [1:0] state, nextstate;

   always_ff @(posedge clk, posedge reset) // Zustandsregister
      if (reset) state <= S0;
      else       state <= nextstate;
   always_comb                             // Zustandsübergangslogik
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
          ...
```


Embedded Systems & Applications

FSM in Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

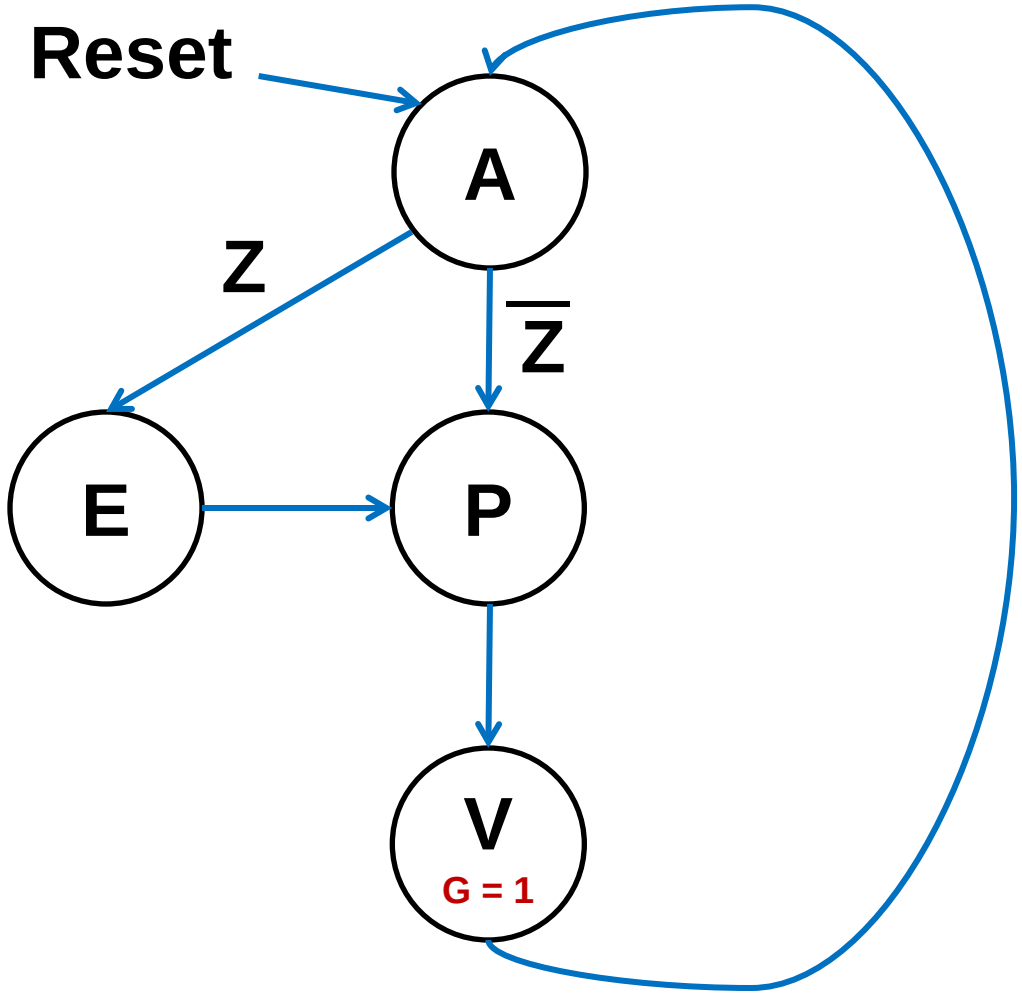
```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   always_ff @(posedge clk, posedge reset) // Zustandsregister
      if (reset) state <= S0;
      else       state <= nextstate;
   always_comb                             // Zustandsübergangslogik
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);               // Ausgangslogik
endmodule
```


Embedded Systems & Applications

FSM Beispiel 2: Zeitplan FSM



Zustände

A = aufwachen
E = essen
P = Zähne putzen
V = Vorlesung

Eingang

Z = Zeit

Ausgang

G = glücklich

FSM Beispiel 2: Zeitplan FSM

TECHNISCHE UNIVERSITÄT DARMSTADT

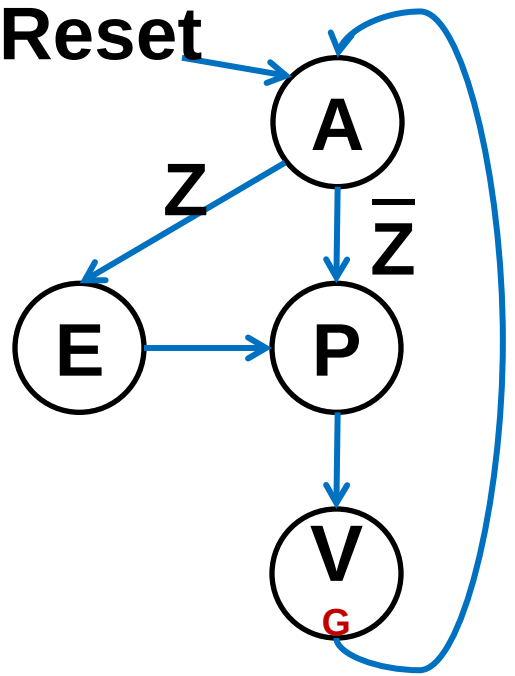
SystemVerilog

FSM Beispiel 2: Zeitplan FSM

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module zeitplanFSM(input  logic clk, reset, z,
                   output logic g);
   typedef enum logic [1:0] {A, E, P, V} statetype;
   statetype state, nextstate;

   always_ff @(posedge clk, posedge reset)
      if (reset) state <= A;
      else       state <= nextstate;
   always_comb
      case (state)
         A:       if (z) nextstate = E;
                  else   nextstate = P;
         E:       nextstate = P;
         P:       nextstate = V;
         default: nextstate = A;
      endcase
   assign g = (state == V);
endmodule
```



Embedded Systems & Applications

FSM Beispiel 2: Zeitplan FSM

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module zeitplanFSM(input  logic clk, reset, z,
                   output logic g);
   typedef enum logic [1:0] {A, E, P, V} statetype;
   statetype state, nextstate;

   always_ff @(posedge clk, posedge reset) // Zustandsregister
      if (reset) state <= A;
      else       state <= nextstate;
   always_comb                             // Zustandsübergangslogik
      case (state)
         A:       if (z) nextstate = E;
                  else   nextstate = P;
         E:       nextstate = P;
         P:       nextstate = V;
         default: nextstate = A;
      endcase
   assign g = (state == V);                // Ausgangslogik
endmodule
```

Parametrisierte Module

TECHNISCHE UNIVERSITÄT DARMSTADT

2:1 Multiplexer:

```
module mux2
  #(parameter WIDTH = 8)  // Parameter: Name und Standardwert
   (input  logic [WIDTH-1:0] d0, d1,
    input  logic             s,
    output logic [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
  mux2 mymux(data0, data1, select, out);
```

Instanz mit 12-bit Busbreite:

```
  mux2 #(12) lowmux(data0, data1, sel, out);
```

Aber **besser** (falls mehrere Parameter auftreten sollten):

```
  mux2 #(.WIDTH(12)) lowmux(data0, data1, sel, out);
```

Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **HDL-Programm zum Testen eines anderen HDL-Moduls**
  - Im Hardware-Entwurf schon lange üblich
  - … seit einigen Jahren auch im Software-Bereich (JUnit etc.)
- **Getestetes Modul**
  - *Device under test (DUT), Unit under test (UUT)*
- **Testrahmen werden nicht synthetisiert**
  - Nur für Simulation benutzt
- **Arten von Testrahmen**
  - **Einfach:** Legt nur feste Testdaten an und zeigt Ausgaben an
  - **Selbstprüfend:** Prüft auch noch, ob Ausgaben den Erwartungen entsprechen
  - **Selbstprüfend mit Testvektoren:** Auch noch mit variablen Testdaten

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Verfasse Verilog-Code um die folgende Funktion in Hardware zu berechnen:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

Der Modulname sei **`sillyfunction`**

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Verfasse Verilog-Code um die folgende Funktion in Hardware zu berechnen:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

Der Modulname sei **`sillyfunction`**

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

Einfacher Testrahmen für Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module testbench1();
  logic a, b, c;
  logic y;

  // Instanz des zu testenden Moduls erzeugen
  sillyfunction dut(a, b, c, y);

  // Eingangswerte anlegen und warten
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

Selbstprüfender Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module testbench2();
  logic  a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y);  // Instanz des zu testenden Module erzeugen
  initial begin                   // Eingangswerte anlegen, warten
                                  // Ausgang mit erwartetem Wert überprüfen
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```


Embedded Systems & Applications

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Trennen von HDL-Programm und Testdaten

- Eingaben
- Erwartete Ausgaben
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Trennen von HDL-Programm und Testdaten

- Eingaben
- Erwartete Ausgaben
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

Eigene Datei für Vektoren

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Trennen von HDL-Programm und Testdaten

- Eingaben
- Erwartete Ausgaben
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

Eigene Datei für Vektoren

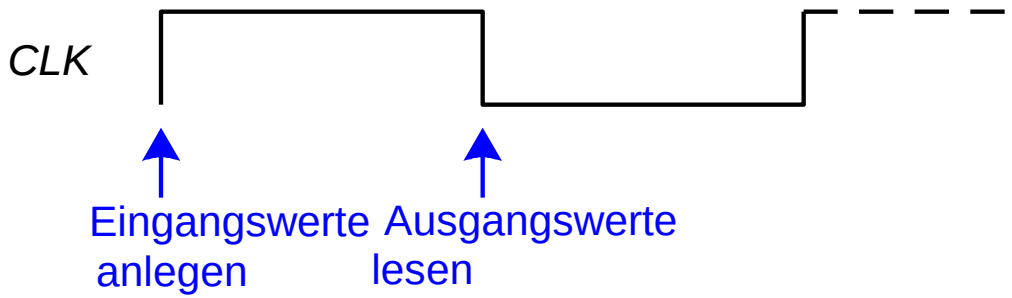
Dann HDL-Programm für universellen Testrahmen

1. Erzeuge Takt zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. Lese Vektordatei in SystemVerilog Array
3. Lege Eingangsdaten an
4. Warte auf Ausgabedaten, werte Ausgabedaten aus
5. Vergleiche aktuelle mit erwarteten Ausgabedaten, melde Fehler bei Differenz
6. Noch weitere Testvektoren abzuarbeiten?

Selbstprüfender Testrahmen mit Testvektoren

Im Testrahmen erzeugter Takt legt zeitlichen Ablauf fest

- Steigende Flanke: Eingabewerte aus Testvektor an Eingänge anlegen
- Fallende Flanke: Aktuelle Werte an Ausgängen lesen

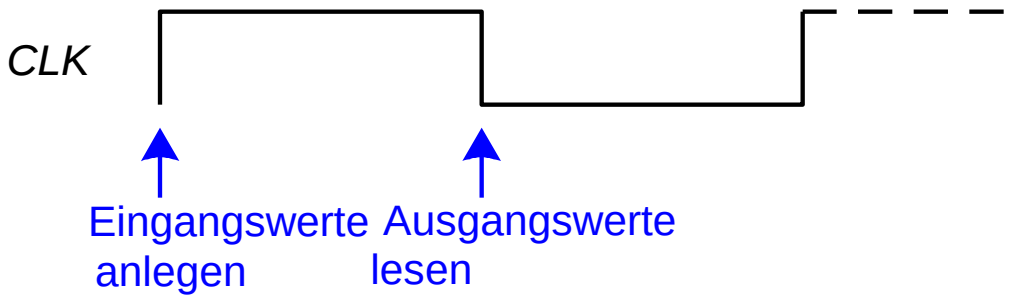


Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Im Testrahmen erzeugter Takt legt zeitlichen Ablauf fest

- Steigende Flanke: Eingabewerte aus Testvektor an Eingänge anlegen
- Fallende Flanke: Aktuelle Werte an Ausgängen lesen



Takt kann auch als Takt für sequentielle synchrone Schaltungen verwendet werden

Embedded Systems & Applications

Einfaches Textformat für Testvektordateien

TECHNISCHE UNIVERSITÄT DARMSTADT

Datei: `example.tv`

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Aufbau:

Eingangsdaten “_” erwartete Ausgangsdaten

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

HDL-Programm für universellen Testrahmen

1. Erzeuge Takt zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. Lese Vektordatei in SystemVerilog Array
3. Lege Eingangsdaten an
4. Warte auf Ausgabedaten, werte Ausgabedaten aus
5. Vergleiche aktuelle mit erwarteten Ausgabedaten, melde Fehler bei Differenz
6. Noch weitere Testvektoren abzuarbeiten?

Testrahmen: 1. Erzeuge Takt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // Verwaltungsdaten
  logic [3:0]  testvectors[10000:0]; // Array für Testvektoren

  // Instanz der Testschaltung erzeugen
  sillyfunction dut (a, b, c, y);

  // Takterzeugung
  always     // Hängt von keinen anderen Signalen ab:
             // Wird immer ausgeführt!
    begin
      clk = 1; #5; clk = 0; #5;
    end
```


Embedded Systems & Applications

2. Lese Testvektordatei in Array ein

TECHNISCHE UNIVERSITÄT DARMSTADT

…

```
// Zu Beginn der Simulation:
// Testdaten einlesen und einen Reset-Impuls erzeugen

initial  // Block wird genau einmal ausgeführt
   begin
     $readmemb("example.tv", testvectors);
     vectornum = 0; errors = 0; // Verwaltungsdaten initialisieren
     reset = 1; #27; reset = 0; // Reset-Impuls erzeugen
   end
```

…

Hinweis: Falls hexadezimale Testvektoren verwendet werden sollen, statt `$readmemb` den Aufruf `$readmemh` verwenden

3. Lege Testdaten an Eingänge an

TECHNISCHE UNIVERSITÄT DARMSTADT

…

```
// zur steigenden Taktflanke (genauer: kurz danach!)
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end
```

…

`a,b,c` sind **Eingänge** der DUT

`yexpected` ist eine **Hilfsvariable**, die nun den erwarteten Ausgangswert dieses Vektors enthält.

4. Warte auf Ausgabedaten, lese Ausgabedaten
5. Vergleiche aktuelle Ausgaben mit erwarteten Werten

TECHNISCHE UNIVERSITÄT DARMSTADT

...

```
// warte auf fallende Flanke zum Lesen der Ausgabedaten der DUT
always @(negedge clk)
  // nur Prüfen, nachdem Schaltung schon initialisiert
  if (~reset) begin
    // vergleiche aktuelle Ausgabe mit erwartetem Wert
    if (y !== yexpected) begin
      // Fehlermeldung
      $display("Fehler: Eingänge = %b", {a, b, c});
      $display("  Ausgänge = %b (%b erwartet)", y, yexpected);
      errors = errors + 1;    // zähle Fehler
    end
```

...

Hinweis: Um Werte hexadezimal auszugeben, Formatkennung %h verwenden

Beispiel: `$display("Error: Eingänge = %h", {a, b, c});`

Embedded Systems & Applications

6. Sind noch weitere Testvektoren abzuarbeiten?

TECHNISCHE UNIVERSITÄT DARMSTADT

...

```
        // Array-Index zum Zugriff auf nächsten Testvektor erhöhen
        vectornum = vectornum + 1;
        // Ist der nächste schon ein ungültiger Testvektor?
        if (testvectors[vectornum] === 4'bx) begin
          // Endmeldung ausgeben
          $display("%d Tests bearbeitet mit %d Fehlern",
                   vectornum, errors);
          $finish;     // Simulation anhalten
        end
    end
endmodule
```

Hinweis: Zum Vergleichen auf X und Z müssen die Operatoren **===** und **!==** benutzt werden

Embedded Systems & Applications

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

HDL-Programm für universellen Testrahmen

1. Erzeuge Takt zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. Lese Vektordatei in SystemVerilog Array
3. Lege Eingangsdaten an
4. Warte auf Ausgabedaten, werte Ausgabedaten aus
5. Vergleiche aktuelle mit erwarteten Ausgabedaten, melde Fehler bei Differenz
6. Noch weitere Testvektoren abzuarbeiten?

SystemVerilog Sprachkonstrukte

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vor Testrahmen alle für die Beschreibung von **echter** Hardware relevanten eingeführt
  - **Schaltungssynthese**

- SystemVerilog kann viel **mehr**
  - Angedeutet beim Testrahmen (Dateioperationen, Ein/Ausgabe, …)
  - Aber in der Regel nicht mehr in Hardware synthetisierbar