Einführung in Computer Microsystems
Sommersemester 2012
2. Block: Verilog Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches

TECHNISCHE UNIVERSITÄT DARMSTADT

Am 25.4.2012 keine reguläre Vorlesung!
Stattdessen: Vortrag von Vertreter der Fa. ARM, Inc.

- Marktführer auf dem Gebiet der eingebetteten Prozessoren
- Insbesondere in Smartphones
  - Apple iOS, Google Android, Microsoft Windows Phone: Alle auf ARM CPUs
- Themenauswahl
  - Geschäftsmodell und Anwendungsbereiche
  - ARM Prozessoren und Architekturen
  - Energiebedarf
  - Programmierung des ARM Cortex-M
  - Busse und Protokolle
  - ...
- Vortragssprache ist Englisch

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Grundlagen

Grundlegender Baustein: Modul

Noch ganz ohne Hardware-Bezug

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module count;          // dies ist ein Kommentar
integer I;

 initial   // führe folgenden Block am Anfang aus
 begin
   $display ("Beginn␣der␣Simulation...");
   for  (I=1; I <= 3; I=I+1)
     $display ("Durchlauf␣%d", I);
   $display ("Ende␣der␣Simulation");
 end

endmodule
```

```
Beginn der Simulation...
Durchlauf              1
Durchlauf              2
Durchlauf              3
Ende der Simulation
```

Modulschnittstelle

Ein- und Ausgänge, Register und Wire

TECHNISCHE UNIVERSITÄT DARMSTADT

```
 // Bestimmung des Maximums
module maximum (
  input  wire [31:0]  A,
                      B,
  output reg  [31:0]  MAX
);

always @(A, B) // führe Block aus, wann immer sich A oder B ändern
  begin
    if  (A > B)
      MAX = A;
    else
      MAX = B;
    $display ("new␣maximum␣is␣%d", MAX);
  end

endmodule
```

Eigentliche Funktion durchaus in Hardware synthetisierbar

Merkwürdiges Konstrukt: `initial`

Ging doch in Java auch ohne ...

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module two_blocks;

  initial
    begin
      $display ("Ja");
      $display ("Ja");
    end

  initial
    begin
      $display ("Nein");
      $display ("Nein");
    end

endmodule
```

Was wird ausgegeben?

```
Ja Ja Nein Nein
Nein Nein Ja Ja
Ja Nein Ja Nein
Nein Ja Nein Ja
NJeain NJeain
NeJainJaNein ...
```

Auflösung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module two_blocks;

  initial
    begin
      $display ("Ja");
      $display ("Ja");
    end

  initial
    begin
      $display ("Nein");
      $display ("Nein");
    end

endmodule
```

Einzige Möglichkeiten:

```
Ja Ja Nein Nein
```

oder

```
Nein Nein Ja Ja
```

“Oder” ???

Nachbildung von Parallelität

Gelegentlich als *Pseudo-Parallelität* bezeichnet

TECHNISCHE UNIVERSITÄT DARMSTADT

- Simulator läuft auf einzelnem Prozessor
  - Traditionell, könnte mittlerweile zwar anders sein (ist aber schwierig!)
  - . . . aber Modellierungskonzepte sind älter
- Simulator muß aber parallele Abläufe ausführen
- Vorgehen
  - Parallele Blöcke werden in beliebiger Reihenfolge nacheinander simuliert
  - Anweisungen innerhalb eines `begin`/`end`-Blocks laufen immer in der hingeschriebenen Reihenfolge ab, und zwar in der Regel ohne Unterbrechung (atomar)

➥**Eigenartige Auffassung von Parallelität?!?!?**

Stimmt! Vorgehen oben ist nur die halbe Miete, später mehr!

Kurze Wiederholung einiger Verilog-Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Operationsgruppe	Bedeutung
+ - * / %	Arithmetik
< <= > >=	Vergleich
== != === !==	Gleichheit
! && ||	logische Operatoren
~ & | ^	bit-weise Operatoren
? :	Auswahl
<< >>	Shift

Beispiel: Einfache ALU

Verhaltensmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module alu (
  input  wire  [2:0] OPCODE,
  input  wire [31:0] A,
                     B,
  output reg  [31:0] RESULT
);

`define ADD      3'b000  // 0      // nur zur Übung:
`define MUL     'b001    // 1      // Konstanten auf
`define AND      3'o2    // 2      // verschiedene Arten
`define LOGAND 3'h3      // 3
`define MOD      4       // 4
`define SHL      3'b101  // 5

  always @ (OPCODE, A, B)
    case (OPCODE)
      `ADD:    RESULT = A + B;
      `MUL:    RESULT = A * B;
      `AND:    RESULT = A & B;
      `LOGAND: RESULT = A && B;
      `MOD:    RESULT = A % B;
      `SHL:    RESULT = A << B;
      default: $display ("Unimplemented␣Opcode:␣%d!", OPCODE);
    endcase

endmodule
```

Testrahmen

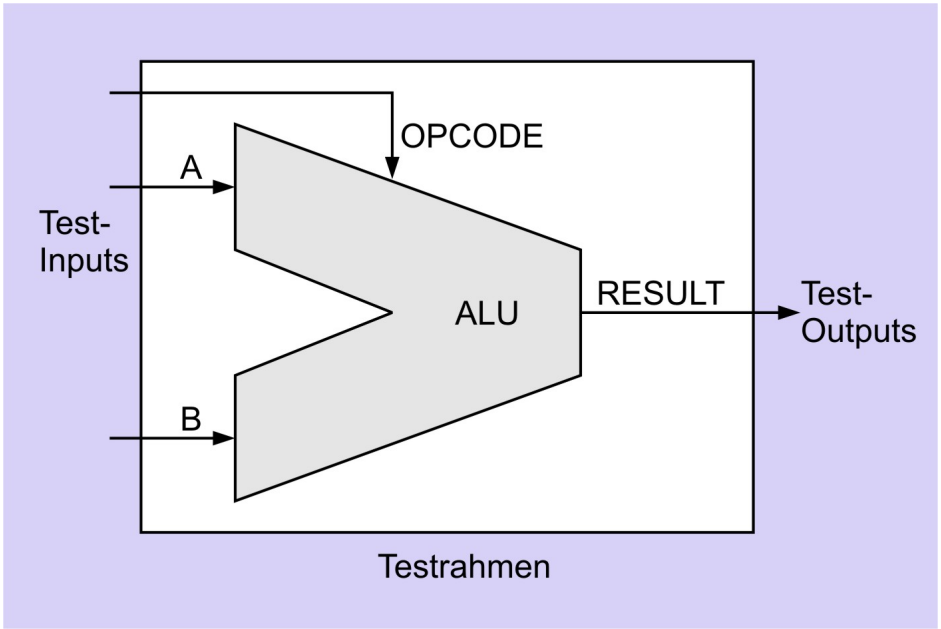
Wie ausprobieren?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Modul `alu` macht freiwillig überhaupt nichts
- Der Simulator prüft quasi nur die Syntax
- Lösung:
  - Von außen Daten an Moduleingänge anlegen
  - Sogenannte Stimuli
  - Dann beobachten, wie sich Modulausgänge verhalten
- Analog zu Unit Tests im Software-Bereich
  - JUnit etc.

Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT



- Saubere Trennung von
  - Prüfling (device under test, DUT)
  - Erzeugung von Eingabedaten
  - Auswertung der Ausgabedaten

Testrahmen für die einfache ALU

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module test;

  reg     [2:0] OPCODE; // Zuweisungsziele für Eingabedaten (Variablen)
  reg    [31:0] A,
                B;
  wire   [31:0] RESULT;  // Stück Draht (zum Lesen der Ausgabe)

`define ADD         0
`define MUL         1
`define AND         2
`define LOGAND      3
`define MOD         4
`define SHL         5

  alu AluDUT (OPCODE, A, B, RESULT); // ALU-Instanz

   initial  begin                          // Test-Inputs
    $display ("Simulation beginnt...");
    OPCODE = `ADD; A = 3; B = 2; #1; // <- Zeit vergehen lassen
    OPCODE = `SHL; A = 3; B = 2; #1;
    $display ("Simulation endet.");
    $finish;
  end

  always @ (RESULT)                    // Test-Outputs
    $display ("OPCODE = %d, A = %d, B = %d: RESULT = %d",
      OPCODE, A[5:0], B[5:0], RESULT[5:0]);

endmodule
```

```
Simulation beginnt...
OPCODE =  0, A =   3, B =   2: RESULT =   5
OPCODE =  5, A =   3, B =   2: RESULT =  12
Simulation endet.
```

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Verhalten und Struktur in Verilog

Verhalten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Abbildung von Eingaben auf Ausgaben
- “was”, nicht “wie”
- Realisierung nicht von außen sichtbar (black box)
- Zur Modellierung reicht häufig ein einzelner `always`-Block

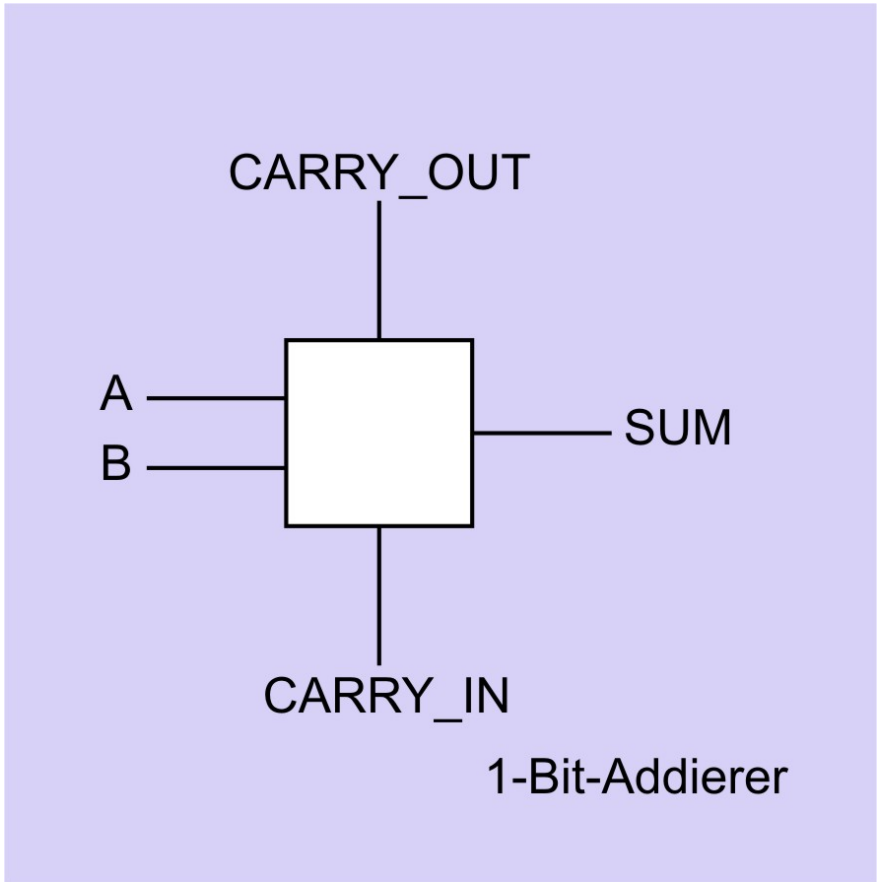
Struktur

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschreibe Einheit als
  - Untereinheiten
  - Verbindungen
- Im Extremfall
  - Keine `always` oder `initial`-Blöcke
  - Nur Modulinstanziierungen

Beispiel: 1b-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

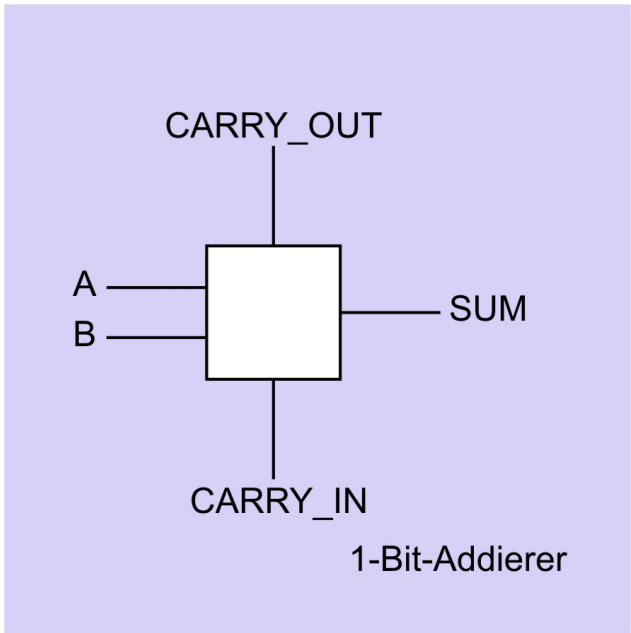


Verhaltensbeschreibung des 1b-Addierers

Konkreter Aufbau aus Gattern interessiert hier nicht

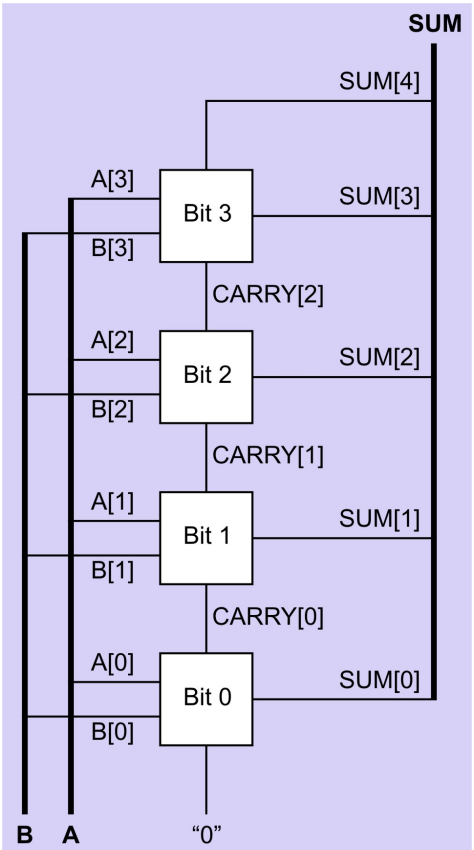
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module one_bit_adder(
  input       A,        // 1-Bit-Wires per Default
              B,
              CARRY_IN,
  output reg  SUM,
              CARRY_OUT
);

  // Verhalten des one_bit_adder
  always @ (A, B, CARRY_IN)
    {CARRY_OUT, SUM} = A + B + CARRY_IN;

endmodule // one_bit_adder
```



Struktur eines 4b-Addierers in Ripple-Carry-Technik

Aufgebaut aus 1b-Addierern



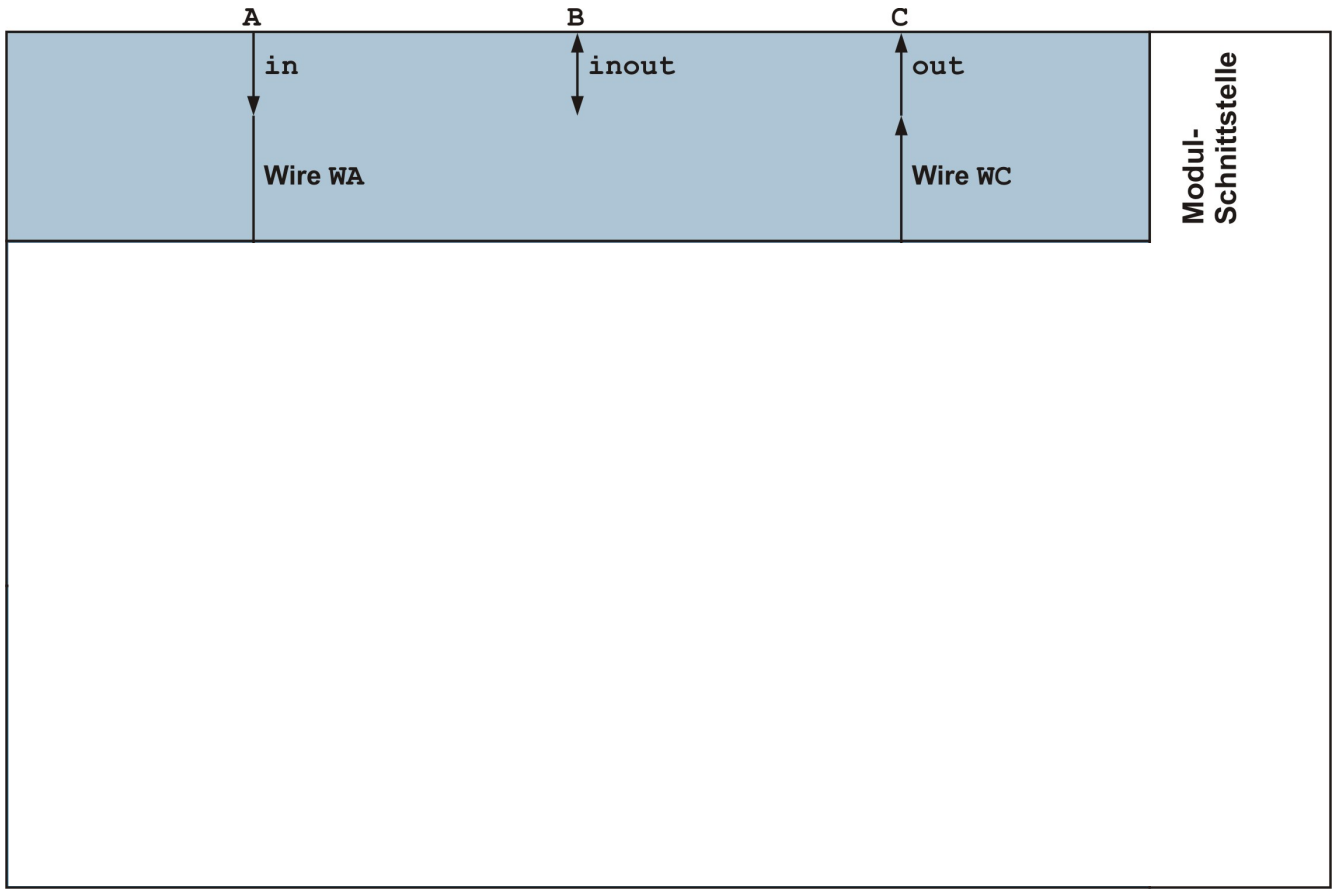

```
module four_bit_adder(
  input  wire [3:0]  A,
                     B,
  output wire [4:0]  SUM
);


  wire [2:0]  CARRY;


  // Struktur des four_bit_adder
  one_bit_adder Bit0 (A[0],  B[0],       1'b0, SUM[0], CARRY[0]);
  one_bit_adder Bit1 (A[1],  B[1],  CARRY[0], SUM[1], CARRY[1]);
  one_bit_adder Bit2 (A[2],  B[2],  CARRY[1], SUM[2], CARRY[2]);
  one_bit_adder Bit3 (A[3],  B[3],  CARRY[2], SUM[3], SUM[4] );

endmodule // four_bit_adder
```

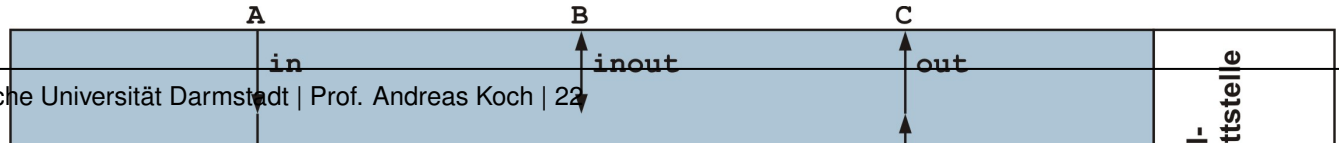
TECHNISCHE
UNIVERSITÄT
DARMSTADT

Elemente von Verilog-Modulen

Modulstruktur von Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT





Modulschnittstelle

`input`, `output`, `inout`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Verilog 2001
module maximum (
  input  wire [7:0] A,
                    B,
  output reg  [7:0] RESULT
);

  always @(A, B)
    if  (A > B)
      RESULT = A;
    else
      RESULT = B;

endmodule
```

```
// Verilog 1995
module maximum (A, B, RESULT);
  input       A,
              B;
  output      RESULT;

  wire [7:0]  A,
              B;
  reg  [7:0]  RESULT;

  always @(A or B)
    if  (A > B)
      RESULT = A;
    else
      RESULT = B;

endmodule
```

- `inout` für bidirektionale Datenbusse

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prozedurale Modellierung

`always` und `initial`-Blöcke

TECHNISCHE UNIVERSITÄT DARMSTADT

- Werden zueinander parallel ausgeführt (in beliebiger Reihenfolge)
- Werden im Inneren i.d.R. sequenziell ausgeführt
- Ausführung erfolgt ohne Unterbrechung
  - Falls keine Zeitkontrollanweisungen (`#` etc.) auftreten
- Eintrittsbedingungen mit `@` (gelesen: *at*)
- `always @(COUNTER)`: Bei Änderungen von `COUNTER`
- `always @(*)`: alle Lesevariablen eines Blockes
- Faustregel
  - `always`-Blöcke in Schaltungsteilen (synthetisierbar)
  - `initial`-Blöcke in Testmodulen (nicht synthetisierbar)

♯ *n*: *n* Zeiteinheiten warten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Explizite Modellierung von Zeit
- Andere parallele Prozesse laufen weiter

```
module time_delay;
  reg DATA;

  always @(DATA)
    $display ("Zeit:␣%2.0f,␣DATA␣=␣%d", $time, DATA);

  initial  begin
    DATA = 0;
    #10;
    DATA = #10 1;
    #10;
  end

endmodule
```

```
Zeit:  0, DATA = 0
Zeit: 20, DATA = 1
```

Sieht einfach aus, Gemeinheiten liegen tiefer

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module time_delay;
  reg DATA;

  always @(DATA)
    $display ("Zeit: %2.0f, DATA = %d", $time, DATA);

  initial  begin
    DATA = 0;
    DATA = 1;
  end

endmodule
```

```
Zeit:  0, DATA = 1
```

- Transition `DATA` 0 $\rightarrow$ 1 nicht sichtbar für `always`-Block
  - `initial`-Block läuft atomar ab

Extremfall

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module time_delay;
  reg DATA;

  always @(DATA)
    $display ("Zeit: %2.0f, DATA = %d", $time, DATA);

  initial  begin
    DATA = 0;
    #0;
    DATA = 1;
  end

endmodule
```

```
Zeit:   0, DATA = 0
Zeit:   0, DATA = 1
```

- Transition nun sichtbar
  - `#` unterbricht Ausführung von `initial`-Block
  - Erlaubt Reaktion durch `always`-Block
  - Es vergeht aber keine Zeit!

Diskussion von ♯

Bisher im wesentlichen Trickserei

TECHNISCHE UNIVERSITÄT DARMSTADT

- # lässt sich nicht synthetisieren
- Hat nur Effekte während der Simulation
- Dort benutzt zur Erzeugung von Testsignalen
- Kenntnisse aber manchmal bei Fehlersuche nützlich

```
module gen_clock;
  reg CLOCK;

  always @(CLOCK)
    $display ("Zeit : %2.0f, CLOCK = %d", $time, CLOCK);

  always begin
    CLOCK = 0;
    #10;
    CLOCK = 1;
    #10;
  end

endmodule
```

```
Zeit:  0, CLOCK = 0
Zeit: 10, CLOCK = 1
Zeit: 20, CLOCK = 0
Zeit: 30, CLOCK = 1
Zeit: 40, CLOCK = 0
Zeit: 50, CLOCK = 1
... bis zum Stromausfall
```

Warten mit @ (*at*)

Warten auf punktuelles Ereignis (Wertänderung, Flanke)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module atdemo;
  reg    CLOCK, SIGNAL1, SIGNAL2;

  always @(posedge CLOCK)
    $display ("Zeit: %2.0f: positive Flanke", $time);

  always @(negedge CLOCK)
    $display ("Zeit: %2.0f: negative Flanke", $time);

  always @(SIGNAL1, SIGNAL2) // oder @(SIGNAL1 or SIGNAL2)
    $display ("Zeit: %2.0f: SIGNAL1 oder SIGNAL2",$time);

   initial  begin
    CLOCK = 0; #10;
    CLOCK = 1; #10;
    SIGNAL2 = 0; #10;
    CLOCK = 0; #10;
    SIGNAL1 = 1; #10;
    $finish;          // Ende der Simulation erzwingen
  end

endmodule
```

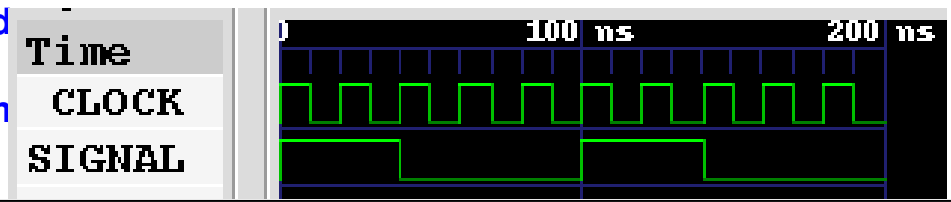
```
Zeit:  0: negative Flanke
Zeit: 10: positive Flanke
Zeit: 20: SIGNAL1 oder SIGNAL2
Zeit: 30: negative Flanke
Zeit: 40: SIGNAL1 oder SIGNAL2
```

Alternative Benutzung von @

TECHNISCHE UNIVERSITÄT DARMSTADT

- @ kann auch innerhalb eines Blocks benutzt werden
- Wartet bis zum Eintreten des Ereignisses
- Kann dort nicht synthetisiert werden
  - Lösung: Echte Zustandsautomaten explizit modellieren
- Aber gelegentlich nützlich für Stimuli-Erzeugung

```
module atdemo2;
  reg    CLOCK, SIGNAL;

  always begin
    CLOCK = 1; #10;
    CLOCK = 0; #10;
  end

  always begin
    // Erzeuge Signalmuster 11000 synchron zur
    // steigenden Flanke von CLOCK
    SIGNAL=1;
    @(posedge CLOCK);
    @(posedge CLOCK);
    SIGNAL=0;
    @(posedge CLOCK);
    @(posedge CLOCK);
    @(posedge CLOCK);
  end

endm
```



Sequentielle Anweisungen innerhalb von Blöcken

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if  (CONDITION) begin
   RESULT = 42
end else begin
   RESULT = 23;
end
```

```
// gleicher Effekt durch
RESULT = (CONDITION) ? 42 : 23;
```

Sequentielle Anweisungen innerhalb von Blöcken

TECHNISCHE UNIVERSITÄT DARMSTADT

```
case (OPCODE)
  'OP1: begin
          RESULT = A + B;
          FLAG = 0;
        end
  'OP2: begin
          RESULT = A − B;
          FLAG = 1;
        end
  default:  $display ("Unimplemented␣Opcode:␣%d!", OPCODE);
endcase
```

Sequentielle Anweisungen innerhalb von Blöcken

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while (REQUEST == 0) begin
  CLK = 0;
  # 10;
  CLK = 1;
  #10;
end

initial  begin : blockname // erforderlich für lokale Variablen
  integer i;                // integer: 32b, vorzeichenbehaftet
  for  (i=0; i<5; i  = i+1) begin
    CLOCK = 0;
    #10;
    CLOCK = 1;
    #10;
  end
end
```

Strukturierung von sequentiellen Blöcken

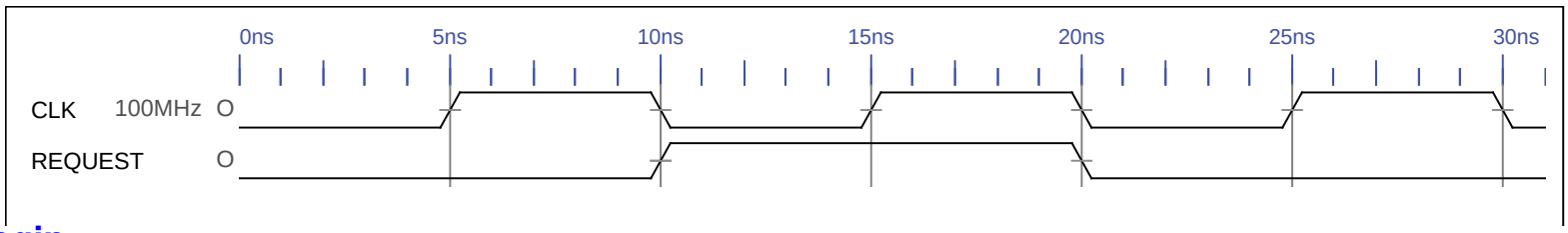
Nur für Testrahmen in der Simulation, bei Synthese Module verwenden!

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module task_example;
  reg    [7:0]  RESULT;

  task  add(
    input reg  [7:0]  A,
                      B
  );
    RESULT = A + B;
  endtask

  task  display_result ;
    $display ("Die␣Summe␣ist␣%d", RESULT);
  endtask

   initial  begin
    add (1,2);
    display_result ;
  end

endmodule
```

Beispiel für Anwendung in der Simulation

TECHNISCHE UNIVERSITÄT DARMSTADT




```
initial  begin
  CLK = 0;
  #5;
  CLK = 1;
  #5;
  REQUEST = 1;
  CLK = 0;
  #5;
  CLK = 1;
  #5;
  RESULT = 0;
  CLK = 0;
  #5;
  CLK = 1;
  #5
end
```

```
task DoClock;
  CLK = 0;
  #5;
  CLK = 1;
  #5;
endtask

initial  begin
  DoClock;
  REQUEST = 1;
  DoClock;
  RESULT = 0;
  DoClock;
end
```

Konstante Werte

Aufbau von Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Bit-Breite (dezimal), falls fehlend: nehme minimale Breite für Wert
2. `s` für vorzeichenbehaftet, falls fehlend: vorzeichenlos
3. Basis: 2 (`'b`), 8 (`'o`), 10 (`'d` oder nichts) sowie 16 (`'h`)
4. Eigentlicher Wert
   - Ziffern der Basis, optional getrennt durch _ (Lesbarkeit)
   - `x` (unbestimmt) oder `z` (hochohmig)

```
module constant_example;
  reg   [7:0]  DATA;

  initial  begin
    DATA =           0;    $display ("DATA␣=␣%b", DATA);
    DATA =          10;    $display ("DATA␣=␣%b", DATA);
    DATA =        'h10;    $display ("DATA␣=␣%b", DATA);
    DATA =        'b10;    $display ("DATA␣=␣%b", DATA);
    DATA =         255;    $display ("DATA␣=␣%b", DATA);
    DATA =       1'b1;     $display ("DATA␣=␣%b", DATA);
    DATA = 'bxxxxzzzz;     $display ("DATA␣=␣%b", DATA);
    DATA = 'b1010_1010;    $display ("DATA␣=␣%b", DATA);
    DATA =       1'bz;     $display ("DATA␣=␣%b", DATA);
    DATA =        'bz;     $display ("DATA␣=␣%b", DATA);
  end
endmodule
```

```
DATA = 00000000
DATA = 00001010
DATA = 00010000
DATA = 00000010
DATA = 11111111
DATA = 00000001
DATA = xxxxzzzz
DATA = 10101010
DATA = 0000000z
DATA = zzzzzzzz
```

Datentypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variablen vom Typ `wire` oder `reg`
  - Können beliebige Bit-Breite verwenden
    - z.B. `reg [7:0] result`
  - Jedes Bit kann einen der Werte 0, 1, x (unbekannt) oder z (hochohmig) haben
- Unterschied zwischen `wire` und `reg`

  reg kann Werte speichern (z.B. ein Flip-Flop)

  wire kann keine Werte speichern, aber übertragen (ein Draht)

Verfeinerung: Vorzeichenbehaftung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ohne Angaben: Vorzeichenlos
- Vorzeichenbehaftete Zahlen durch Schlüsselwort `signed`
  - `wire signed [7:0] op1`
  - `reg  signed [3:0] op2`
- Konstanten durch `s` vor Kennung für Basis
  - `4'she`: 4b breit, vorzeichenbehaftet, hexadezimal, Wert -2
  - `4'he`: 4b breit, vorzeichenlos, hexadezimal, Wert 14

Vorsicht beim Rechnen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nur wenn alle Teile eines Ausdrucks `signed` sind, ist Ergebnis `signed`
- Wenn auch nur ein Teil `unsigned` ist, wird Ergebnis `unsigned`
- Unabhängig von Vorzeichenbehaftung des Zuweisungsziels
- Ergebnis wird abhängig von seiner Vorzeichenbehaftung auf Breite von Ziel aufgefüllt
  - Bei `unsigned`: Mit Nullbits
  - Bei `signed`: Durch Vorzeichenerweiterung
    - *sign extension*, TGDI

Beispiel: Vorzeichen- und Breitenerweiterung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module sign_test;
    reg             [2:0]  u1 =  1;        // bitmuster 001 = 1
    reg             [2:0]  u2 = −2;        // bitmuster 110 = 6
    reg signed      [2:0]  s1 =  1;        // bitmuster 001 = 1
    reg signed      [2:0]  s2 = −2;        // bitmuster 110 = −2
    reg             [4:0]  u;
    reg signed      [4:0]  s;
    reg             [4:0]  u3 = 4'he;      // bitmuster 01110 = 14
    reg signed      [4:0]  s3 = 4'she;     // bitmuster 11110 = −2

     initial  begin
       u = u1 + u2; s = s1 + s2;
       $display("u=u1+u2=%b+%b=%b␣s=s1+s2=%b+%b=%b", u1, u2, u, s1, s2, s);

       u = u1 + s2; s = s1 + u2;
       $display("u=u1+s2=%b+%b=%b␣s=s1+u1=%b+%b=%b", u1, s2, u, s1, u2, s);

       u = s1 + s2; s = u1 + u2;
       $display("u=s1+s2=%b+%b=%b␣s=u1+u2=%b+%b=%b", s1, s2, u, u1, u2, s);

       u = u3 + u1; s = s3 + s1;
       $display("u=u3+u1=%b+%b=%b
    end
endmodule
```

```
u=u1+u2=001+110=00111   s=s1+s2=001+110=11111
u=u1+s2=001+110=00111   s=s1+u2=001+110=00111
u=s1+s2=001+110=11111   s=u1+u2=001+110=00111
u=u3+u1=01110+001=01111 s=s3+s1=11110+001=11111
```

Typkonvertierungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Implizite Konvertierung in vorzeichenlosen Typ

- Anwendung des Extraktionsoperators [*msb*:*lsb*]
- Auch bei Angabe des gesamten Wortes

```
reg signed [7:0] DATA;
... = DATA[7:0];
```

ist die rechte Seite immer vorzeichenlos

Explizite

- `$signed(`*v*`)` konvertiert *v* in vorzeichenbehafteten Typ
- `$unsigned(`*v*`)` konvertiert *v* in vorzeichenlosen Typ

Was ist mit `integer`?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nicht für Synthese verwenden!
- Nur ungenau definiert
  - 32b oder 64b vorzeichenbehaftete Zahl
  - Hängt von CAD-Werkzeugen ab!
- Aber nützlich für
  - Simulation
  - Schleifenzähler für `for` etc.

Verbinden von Elementen

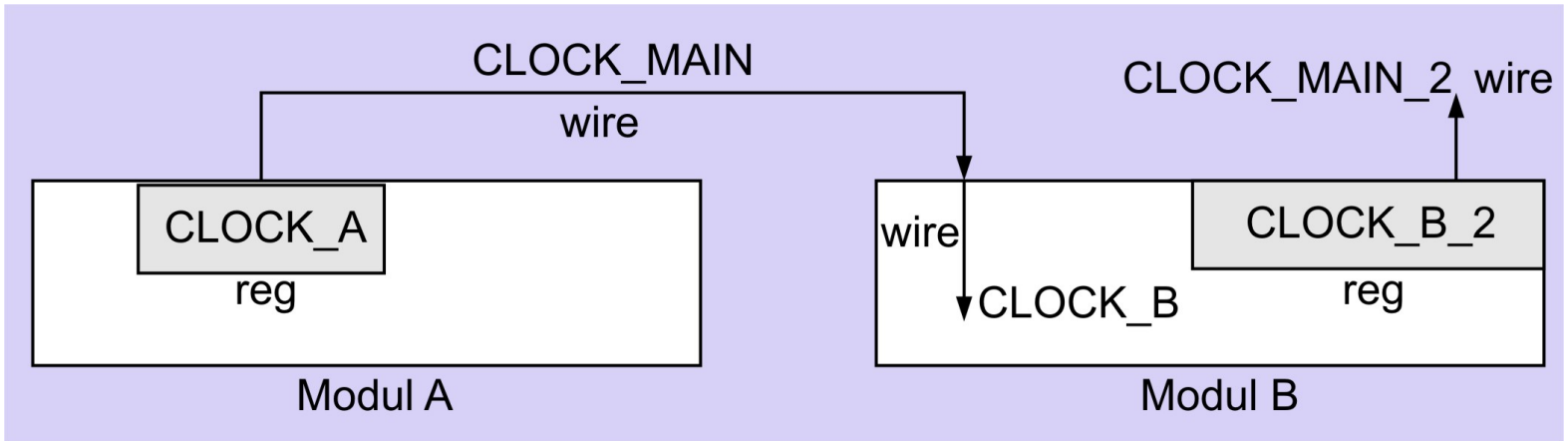
Verbindungen zwischen Registern und Wires

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Wire verbindet ein (oder mehrere) Register oder Wires mit irgendetwas
- Beispiel: R1 $\rightarrow$ W1 $\rightarrow$ W2 $\rightarrow$ R2
- Treiben eines Wires durch ständige Zuweisung
  - `assign W1 = R1`
  - “Draht W1 wird am Ausgang des Registers R1 festgelötet”
  - W1 spiegelt alle Änderungen von R1 wider
- Übernehmen eines Wertes in Register durch normale Zuweisung
  - `R2 = W2` oder `R2 <= W2`
  - Register ändert Wert nur bei Ausführung der Zuweisung

Beispiel: Wire und Register

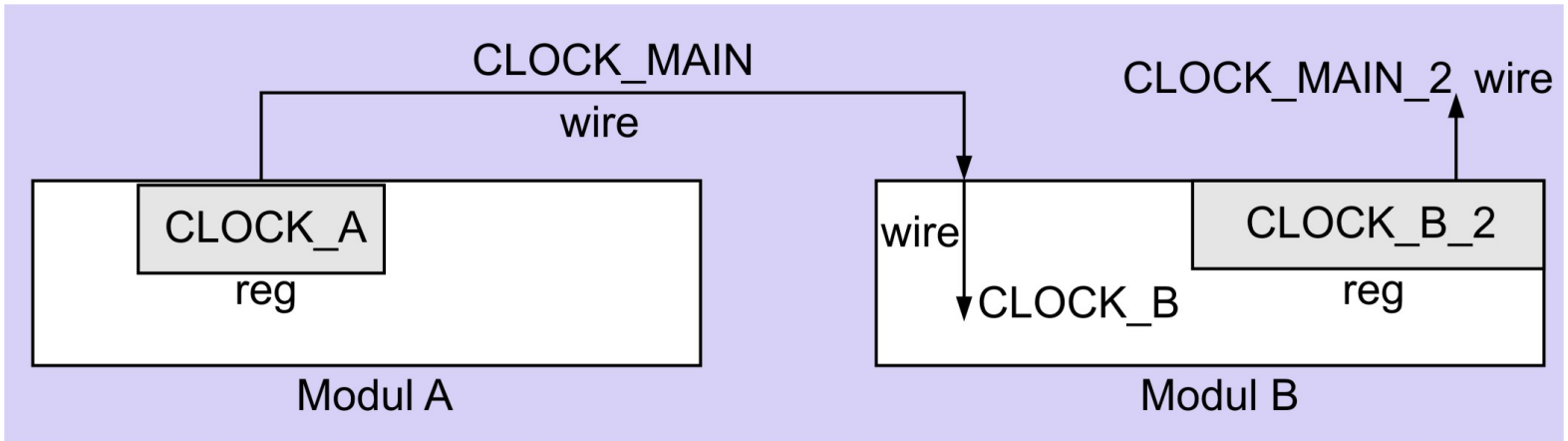
Hier gezeigt: Modul A




```
module a (
  output reg CLOCK_A
);

  always begin
    CLOCK_A = 0;          // Clock auf 0 setzen
    #10;                  // 10 Zeiteinheiten warten
    CLOCK_A = 1;          // Clock auf 1 setzen
    #10;                  // 10 Zeiteinheiten warten
  end
endmodule // a
```

Beispiel: Wire und Register

Hier gezeigt: Modul B

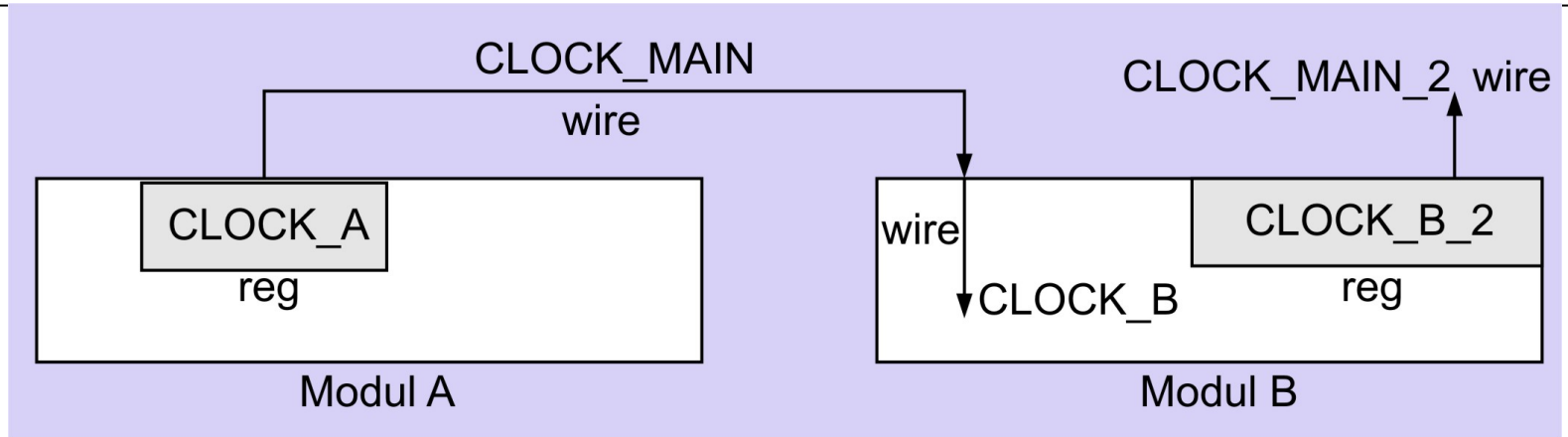



```
module b(
  input  wire CLOCK_B,        // Haupttakt
  output reg  CLOCK_B_2       // halb so schneller Takt
);

   initial  begin
     CLOCK_B_2 = 0;             // Startwert für CLOCK_B_2
  end

  always @(posedge CLOCK_B) begin
     CLOCK_B_2 = ~CLOCK_B_2;
  end

endmodule
```

Beispiel: Wire und Register

Hier gezeigt: Hauptmodul `main`

TECHNISCHE UNIVERSITÄT DARMSTADT




```
module main;
  wire CLOCK_MAIN,
       CLOCK_MAIN_2;

  a A (CLOCK_MAIN);              // Instanzen
  b B (CLOCK_MAIN, CLOCK_MAIN_2);

  always @(CLOCK_MAIN)
    $display ("CLOCK_MAIN");

  always @(CLOCK_MAIN_2)
    $display ("CLOCK_MAIN_2");

  initial  begin
    #100;        // 100 Zeiteinheiten warten
    $finish;     // und die Simulation beenden
  end

endmodule
```

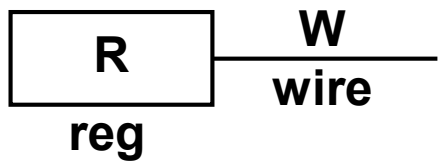
```
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
...
```

Ständige Zuweisung mit `assign`

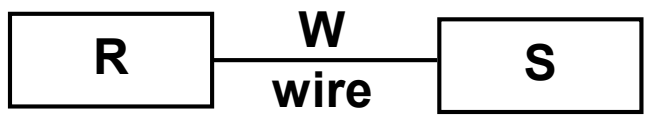
continuous assignment

TECHNISCHE UNIVERSITÄT DARMSTADT

- `assign W=R`: Änderungen an `R` werden auf `W` sichtbar



- Bei `assign W=R; assign W=S;`



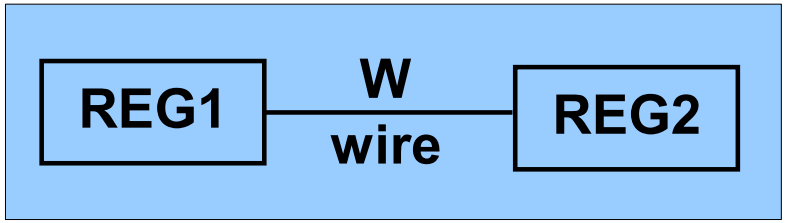
  ist W unbestimmt (`x`), wenn
  - `R` und `S` nicht den gleichen Wert haben ...
  - und keiner von beiden hochohmig (`z`) ist
- Ständige Zuweisung ist auch mit Ausdruck möglich

  ```
  assign W = R + 2*S;
  ```
- Ständige Zuweisungen laufen pseudo-parallel zu prozeduralen Blöcken ab

Mehrere Signale gleichzeitig auf einem Wire

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module constant_example_2;
  reg   [7:0]  REG1,
               REG2;
  wire  [7:0]  W = REG1;
  assign       W = REG2;

   initial   begin
                                        #1; $display ("W␣=␣%b", W);
     REG1 = 0; REG2 =                0; #1; $display ("W␣=␣%b", W);
     REG1 = 10; REG2 =            8'bz; #1; $display ("W␣=␣%b", W);
                 REG2 = 8'b11111111; #1; $display ("W␣=␣%b", W);
                 REG2 =            8'bx; #1; $display ("W␣=␣%b", W);
                 REG2 =            8'bz; #1; $display ("W␣=␣%b", W);
  end

endmodule
```




```
W = xxxxxxxx
W = 00000000
W = 00001010
W = xxxx1x1x
W = xxxxxxxx
W = 00001010
```

Felder von Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

- `reg A[1:1000]` oder `reg A[1000:1]`
  - Feld von 1000 Variablen, jede 1b breit
  - `RESULT = A[500]`
- `reg [15:0] B [1:1000]`
  - Feld von 1000 Variablen, jede 16b breit
  - `RESULT = A[500][8]`
- `reg [15:0] B [1:100][1:200][1:300]`
  - Feld von 6.000.000 Variablen, jede 16b breit
  - `RESULT = A[99][156][223][7]`

Operatoren

Arithmetik

TECHNISCHE UNIVERSITÄT DARMSTADT

- `+`, `-`: Kein Problem
- `*`: Nicht von allen Tools synthetisierbar
  - Kann sehr große Schaltungen nach sich ziehen
  - Hängt von Zieltechnologie ab
  - Hier bei uns aber grundsätzlich OK
  - Datentypen `signed` beachten!
- `/`, `%`: In der Regel nicht synthetisierbar
  - Ausnahme: Division durch Zweierpotenz
  - In allen anderen Fällen Modul aus Bibliothek instantiieren

Prüfung auf Gleichheit/Ungleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT

`==`, `!=` Logische Gleichheit/Ungleichheit

- Wenn beide Operanden einen Wert von 0 oder 1 haben ...
- liefere `1'b1` bei Gleichheit/Ungleichheit, `1'b0` sonst
- Falls einer der Operanden $\notin \{0, 1\}$, liefere `1'bx`

`===`, `!==` Wörtliche Gleichheit/Ungleichheit

- Liefere `1'b1`, wenn beide Operanden gleich/ungleich sind
- `1'b0` sonst
- Das gilt nun auch für die Werte `x` und `z`
- Nicht synthetisierbar, nur in Testrahmen sinnvoll

Arithmetische Vergleiche

TECHNISCHE UNIVERSITÄT DARMSTADT

`>`, `>`, `<=`, `>=` Arithmetische Vergleiche

- Wenn einer der Operanden $\notin \{0, 1\}$ ist, liefere `1'bx`
- Liefere `1'b1` wenn der Vergleich wahr ist, `1'b0` sonst
- Beachte korrekte Vorzeichenbehaftung der Operanden (`signed`)

Beispiele: Vergleiche

TECHNISCHE UNIVERSITÄT DARMSTADT

```
(1'bx  ==  1'b1 ) =  x
(1'bx  === 1'b1 ) =  0
(1'bx  ==  1'bx ) =  x
(1'bx  === 1'bx ) =  1
(1'bz  !=  1'b1 ) =  x
(1'bx  <=  1'b1 ) =  x
(2'bxx ==  2'b11) =  x
(3'h7  <=  3'h1 ) =  0
(3'sh7 <=  3'sh1) =  1
```

Logische Operatoren: `!`, `&&`, `||`, ^

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vergleichbar den entsprechenden Operatoren in C und Java
- Aber Hardware-Werte `x` und `z` beachten!

```
(! 1'b1          ) =  0
(! 1'bx          ) =  x
(! 1'bz          ) =  x
(1'b1  && 1'b0 ) =  0
(1'bx  && 1'b0 ) =  0
(1'bx  && 1'b1 ) =  x
(1'bx  || 1'b0 ) =  x
(1'bx  || 1'b1 ) =  1
(2'b00 || 2'bxx) =  x
```

Bit-weise Operatoren: ~, &, |

TECHNISCHE UNIVERSITÄT DARMSTADT

- Gleiche Ideen wie bei den logischen Operatorn
- Nun aber auf jedes einzelne Bit angewandt

```
(~ 4'bzx10          ) =   4'bxx01
(  4'b001x & 4'b0x10) =   4'b0010
(  2'b1x   | 2'b00  ) =   2'b1x
```

Konkatenation und Vervielfältigung mit { und }

TECHNISCHE UNIVERSITÄT DARMSTADT

Konkatenation Zusammensetzen von Signalen zu größeren Einheiten
`{3'b100, 4'bxxzz, 2'ha}` ergibt `100_xxzz_10`

Vervielfältigen von Signalen
`{ 3 { 4'b1010 } }` ergibt `12'b1010_1010_1010`

Kombination der beiden Operatoren ist möglich `{ 4 { 2'b00, 2'b11} }` ergibt
`16'b0011_0011_0011_0011`

Schieben

TECHNISCHE UNIVERSITÄT DARMSTADT

Logisches Schieben

```
module shift;

  initial begin
    $display ("%b", 8'b1111_0000 >> 4);  0000_1111
    $display ("%b", 8'b0000_1111 >> 4);  0000_0000
    $display ("%b", 8'b0000_1111 << 4);  1111_0000
  end

endmodule
```

Arithmetisches Shiften: Erhält Vorzeichen beim Rechts-Shift mit >>>, <<< verhält sich wie <<

```
  $display ("%b", 8'sb1111_0000 >>> 4);  1111_1111
  $display ("%b", 8'sb1111_0000 <<< 1);  1110_0000
  $display ("%b", 8'sb1111_0000 <<< 4);  0000_0000
```

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Feinheiten von Anweisungen

Blockende Zuweisung =

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wird immer zusammenhängend ausgeführt
- Auch wenn sie eine Zeitkontrolle #*n* enthält
- Nachfolgende Anweisungen starten erst nach Ende der blockenden Zuweisung
- Ablauf der blockenden Zuweisung
  1. Lese aktuelle Werte von Variablen und werte Ausdruck auf rechter Seite aus
  2. Warte evtl. mit # die angegebene Zeit ab
  3. Übernehme Wert in Zuweisungsziel auf linker Seite
  4. Mache mit nächster Anweisung weiter
- Benutzung
  - Zur Erzeugung von Stimuli in Simulation
  - In rein kombinatorischen Blöcken in der Synthese
    - Ohne `always @(posedge ...)`

Nichtblockende Zuweisung <=

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wird immer in zwei Phasen getrennt ausgeführt
- Ablauf der nichtblockenden Zuweisung
  1. Lese aktuelle Werte von Variablen und werte Ausdruck auf rechter Seite aus, merke Ergebnis
  2. Mache sofort mit nächster Anweisung im Block weiter
  3. Am Ende des Blockes
     - Übernehme gemerkte Werte in Zuweisungsziele auf linker Seite
     - Falls Zeitkontrolle: Verzögere obige Zuweisung auch noch (benutzen wir aber nicht!)
- Benutzung
  - In allen sequentiellen Blöcken während der Synthese

Niemals = und <= an eine Variable in einem Block mischen!

Beispiel: Blockende Zuweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module blocking_1;

  reg A, B;

  always @(A,B)
    $display("A=%b␣B=%b", A, B);

   initial  begin
    A = 0;
    B = 1;

    A = B;
    B = A;
  end

endmodule
```

```
A=1 B=1
```

Ausführung nacheinander.

Beispiel: Nichtblockende Zuweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module blocking_2;

  reg A, B;

  always @(A,B)
    $display("A=%b␣B=%b", A, B);

   initial  begin
    A = 0;
    B = 1;

    A <= B;
    B <= A;
  end

endmodule
```

```
A=1 B=0
```

Getrennte Ausführung von Lesen und Schreiben.

Beispiel: Zeitverhalten

Bei blockenden und nicht-blockenden Zuweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module blocking_3;

  reg A, B, C, D, E, F;

  // blockende Zuweisungen
  initial  begin
    A = #10 1;
    B = #2 0;
    C = #4 1;
  end

  // nichtblockende Zuweisungen
  initial  begin
    D <= #10 1;
    E <= #2 0;
    F <= #4 1;
  end

  always @(A,B,C,D,E,F)
    $display (
      "t=%2.0f   A=%b B=%b C=%b   D=%b E=%b F=%b",
      $time, A, B, C, D, E, F);

endmodule
```

```
t= 0   A=x B=x C=x   D=x E=x F=x
t= 2   A=x B=x C=x   D=x E=0 F=x
t= 4   A=x B=x C=x   D=x E=0 F=1
t=10   A=1 B=x C=x   D=1 E=0 F=1
t=12   A=1 B=0 C=x   D=1 E=0 F=1
t=16   A=1 B=0 C=1   D=1 E=0 F=1
```

Verbindungen über Port-Namen

Verbindungen über Port-Namen

TECHNISCHE UNIVERSITÄT DARMSTADT

Verbindung über Port-Reihenfolge

```
module topmod;
  wire [4:0] v;
  wire a,b,c,w;

  modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
  inout wa, wb;
  input c, d;

  ...
endmodule
```

Verbindungen über Port-Namen

```
module topmod;
  wire [4:0] v;
  wire a,b,c,w;

  modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule
...
```

- Unterschiedliche Reihenfolge
- Nichtangeschlossene Ports

Symbolische Konstanten

Makros

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ähnlich C-Präprozessor
- Simple Textersetzung, keine Typprüfung
- Über Modulgrenzen hinweg gültig bis zum Programmende

```
module module_1;

‘define TEXT "Hallo"
‘define TIMES 3

  reg [2:0] COUNTER;

  initial
    for (COUNTER = 1; COUNTER <= ‘TIMES; COUNTER = COUNTER + 1)
      $display (‘TEXT);

endmodule


module module_2;
  reg [2:0] COUNTER;

  initial
    for (COUNTER = 1; COUNTER <= ‘TIMES; COUNTER = COUNTER + 1)
      $display (‘TEXT);

endmodule
```

Modulparameter mit `parameter` und `defparam`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Übergebe Konstanten in eine Modulinstanz
  - `parameter` bei der Moduldefinition
  - `defparam` bei der Instanziierung

```
module counter #(
  parameter Width = 8
) (
  input  wire                 CLOCK,
  output reg    [Width-1:0]   COUNT
);

   initial
     COUNT = 0;
  always @(posedge CLOCK)
     COUNT = COUNT + 1;
endmodule // counter

module main;
  defparam Counter1.Width = 3; // Parameter explizit definiert
  wire [Counter1.Width-1:0]  C1;
  wire [3:0]                 C2;
  reg                        CLOCK;
 ...
// Takterzeugung & $display C1, C2 weggelassen
 ...
  counter       Counter1(CLOCK, C1);
  counter #(4)  Counter2(CLOCK, C2); // Parameter bei Instanzierung

endmodule // main
```

```
  0: C1=0 C2= 0
 10: C1=1 C2= 1
 30: C1=2 C2= 2
 50: C1=3 C2= 3
 70: C1=4 C2= 4
 90: C1=5 C2= 5
110: C1=6 C2= 6
130: C1=7 C2= 7
150: C1=0 C2= 8
170: C1=1 C2= 9
190: C1=2 C2=10
210: C1=3 C2=11
230: C1=4 C2=12
250: C1=5 C2=13
270: C1=6 C2=14
290: C1=7 C2=15
310: C1=0 C2= 0
```

Lokale Parameter mit `localparam`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Falls Parameter nicht von außen überschrieben werden dürfen
- Sonst Verhalten wie `parameter`

```
module RAM
 #(parameter ASIZE=10, DSIZE=8)
  (inout [DSIZE-1:0] data,
   input [ASIZE-1:0] addr,
   input             en, rw_n);

   // Speichertiefe ist 2**(ASIZE)
   localparam MEM_DEPTH = 1<<ASIZE;

   reg [DSIZE-1:0] mem [0:MEM_DEPTH-1];

    ...

endmodule
```

Zeiteinheiten mit `timescale`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Was bedeutet `#1` überhaupt?
  - Sekunden? Stunden? Wochen?
- Zuordnung durch ‘`timescale`-Direktive
  - Am Anfang des Verilog-Modells
- Zwei Parameter
  1. Maß für 1 Zeiteinheit
     - 1, 10, 100
     - Einheit s, ms, us, ns, ps, oder fs
  2. Auflösung der Simulation
     - 1, 10, 100
     - Einheit s, ms, us, ns, ps, oder fs
     - Muß kleiner gleich Zeiteinheit sein!
     - Genauer $\rightarrow$ langsamer
- Bei RTL-Simulation nicht so kritisch
- Bei uns oft ausreichend:
  - `‘timescale 1 ns / 1 ns`
  - `‘timescale 1 ns / 10 ps`

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Systemfunktionen

Ausgaben mit `$display` und `$write`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beide geben Text und formatierte Daten aus
- Formatierung

\n	neue Zeile
\t	Tabulator
\\	das Zeichen \
\"	Anführungszeichen
%%	das Zeichen %
%h, %H	Hexadezimalzahl
%d, %D	Dezimalzahl
%o, %O	Oktalzahl
%b, %B	Binärzahl
%f, %F	reelle Zahl
%c	einzelnes Zeichen
%s	Zeichenkette
%t	Zeit
%m	aktueller Modulname

- `$display` gibt immer Zeilenvorschub am Ende aus
- `$write` nicht
- `$display("Zur Zeit %t ist das A=%b und B=%d", $time, A, B);`

Lesen von Speicherdaten aus Datei

Mit `$readmemh`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module readmemh_demo;

// Speicher
reg [31:0]  Mem [0:11];

  // Lese Speicherdaten aus Datei
   initial
     $readmemh("data.txt",Mem);

  // Inhalt des Speichers anzeigen

   initial  begin : a_block
     integer k;
     $display("Inhalt von Mem:");
     for  (k=0; k<12; k=k+1)
       $display("%d:%h",k,Mem[k]);
   end

endmodule
```

```
// Inhalt von data.txt

02328020  // Kommentar
02328022
02328024
02328025
8e700002
ae700001
1232fffa
1210fff9
```

```
Inhalt von Mem:
 0:02328020
 1:02328022
 2:02328024
 3:02328025
 4:8e700002
 5:ae700001
 6:1232fffa
 7:1210fff9
 8:xxxxxxxx
 9:xxxxxxxx
10:xxxxxxxx
11:xxxxxxxx
```

Schreiben: Eigene Schleife implementieren

Simulation beenden

TECHNISCHE UNIVERSITÄT DARMSTADT

$finish beendet Simulation sofort

- Vorsicht in Xilinx ISE: Schließt auch Signaldiagramm!

$stop schaltet Simulator in interaktiven Modus

- Gelegentlich für Debugging nützlich:

```
% show value Q -radix dec
42
% run
Simulation wird nun fortgesetzt
```

Verilog Sprachstandard

TECHNISCHE UNIVERSITÄT DARMSTADT

Wenn man es genau wissen möchte:

- Sprache: Standard IEEE 1364-2005 “Verilog Language Reference Manual”
- Syntheseregeln: Standard IEEE 1364.1 / IEC 62142-2005 “Verilog register transfer level synthesis”

Aus dem TU Darmstadt-Netz (ggf. via VPN) über ULB aus der IEEE Literaturdatenbank Xplore abrufbar.

Einführung in Computer Microsystems
Sommersemester 2012
3. Block: Modellierung in Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

Register-Transfer-Logik

Register-Transfer-Logik (RTL)

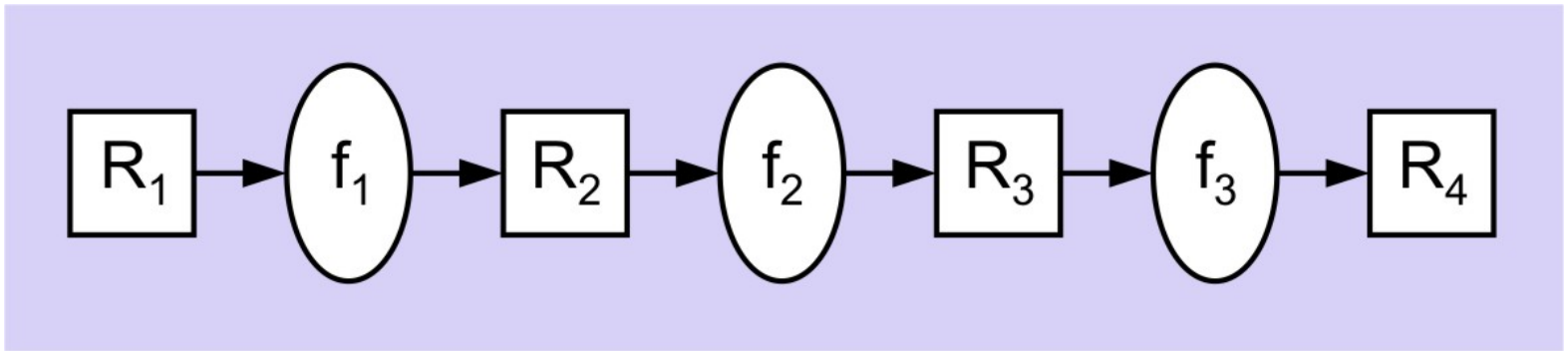
Einführung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ähnlichkeit zum Programmieren $y = f_3(f_2(f_1(x)))$
  - Aber räumlich parallel verteilt
- Synchron durch gemeinsamen Takt
- Gut testbar
- Sehr kompakt mit nichtblockender Zuweisung realisierbar

Pipeline in Register-Transfer-Logik

TECHNISCHE UNIVERSITÄT DARMSTADT



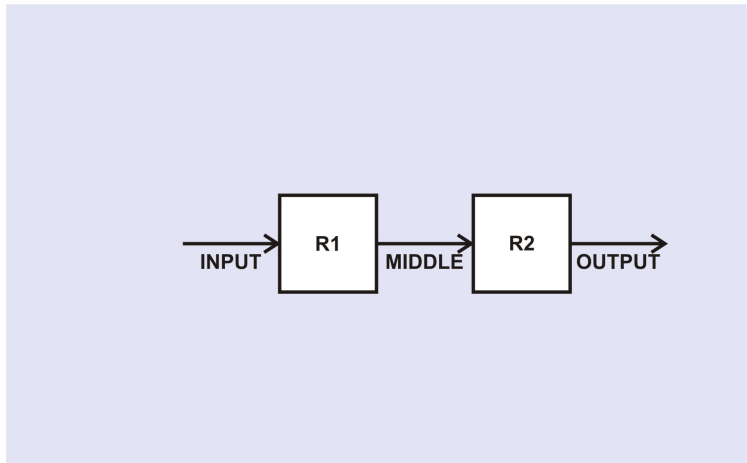
- Kombinatorische Logik zwischen den Registern
  - f_1: verdoppeln
  - f_2: plus 5
  - f_3: quadrieren
- Pipeline berechnet $R_4 = (2R_1 + 5)^2$
  - bearbeitet 3 Datensätze gleichzeitig
  - gibt pro Takt ein Ergebnis aus
  - Damit 3x schneller als sequentielle Berechung der drei Funktionen

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Konstruktion von Pipelines in RTL

1. Schritt: Flip-Flop-Kette

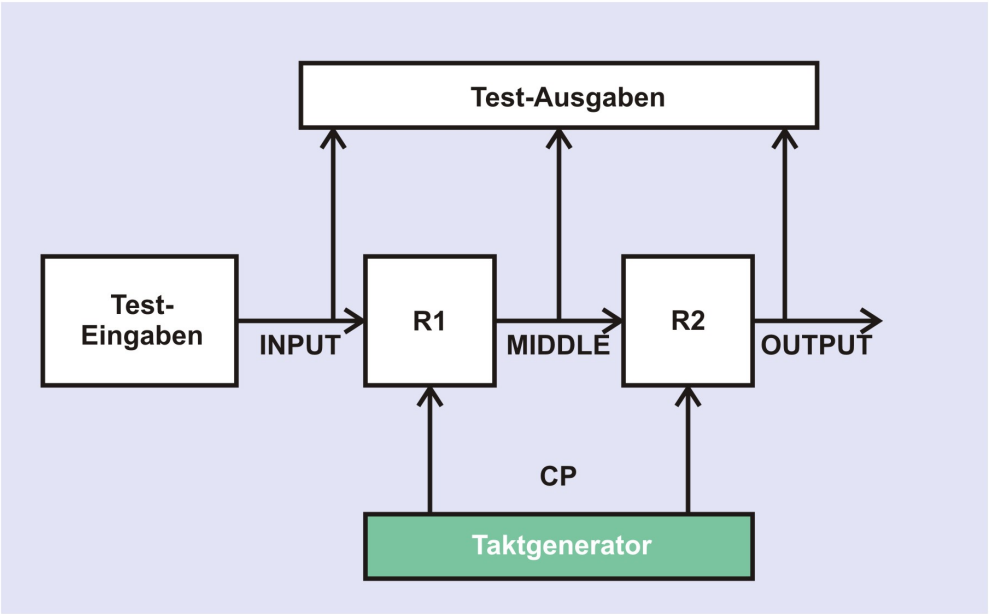
TECHNISCHE UNIVERSITÄT DARMSTADT



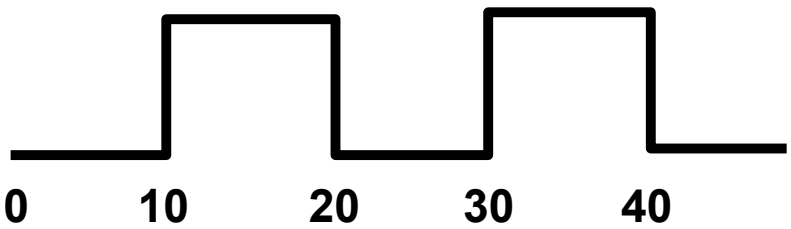
- Mini-Pipeline aus zwei Flip-Flops
- Flip-Flops sind flankengesteuert
  - Unterschied zu Latches (pegelgesteuert)
  - Aufbau z.B. aus Master-Slave-Latches (TGDI)
- Annahme hier: vorderflankengesteuert
  - **always** @(**posedge** CLOCK)

2. Schritt: Takterzeugung

TECHNISCHE UNIVERSITÄT DARMSTADT

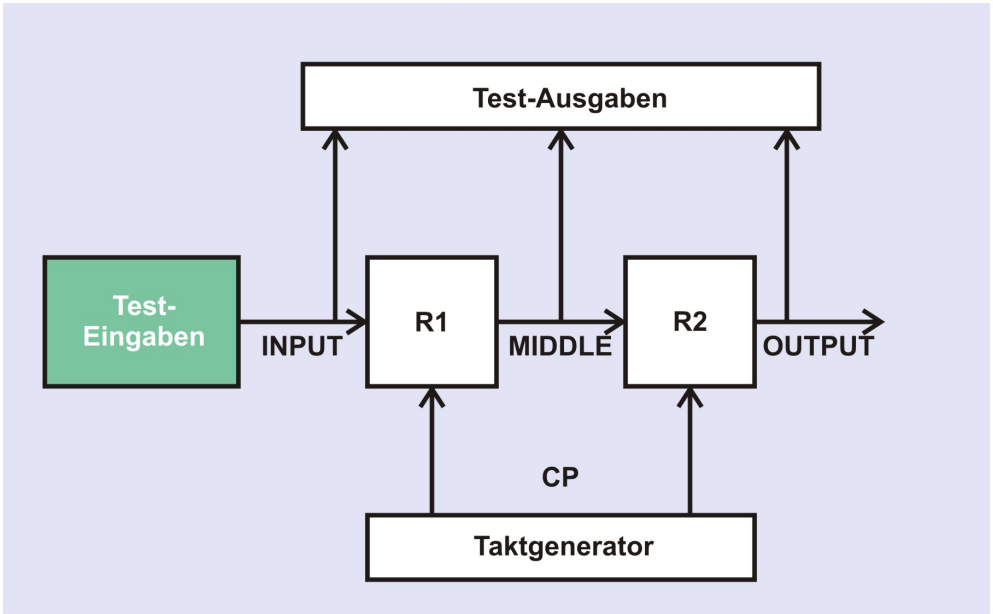



```
always begin
   CP = 0; #10;
   CP = 1; #10;
end
```

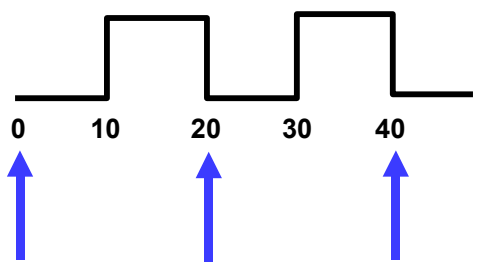


3. Schritt: Testeingaben (Stimuli)

TECHNISCHE UNIVERSITÄT DARMSTADT

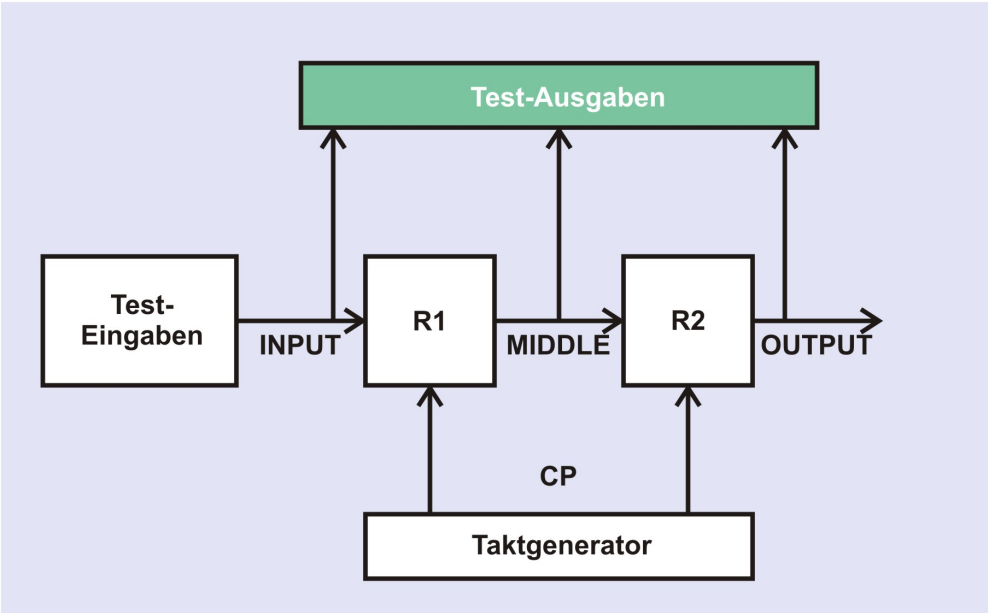



```
initial  begin
  INPUT = 0;      #20;
  INPUT = 255;  #20;
  INPUT = 8'haa; #20;
  $finish;
end
```



4. Schritt: Testausgaben

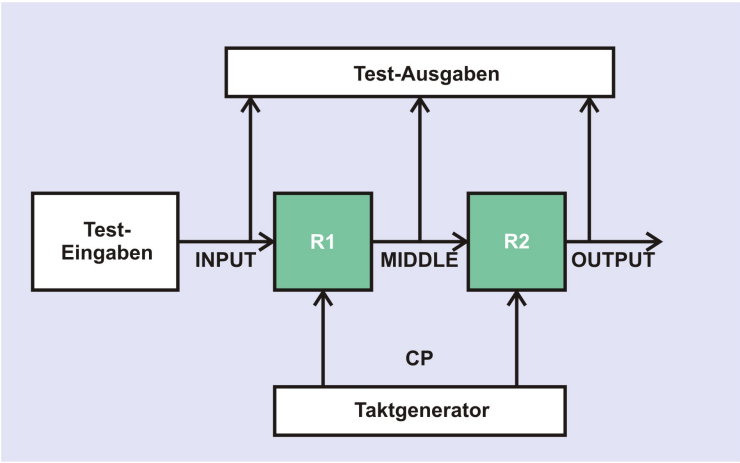
TECHNISCHE UNIVERSITÄT DARMSTADT

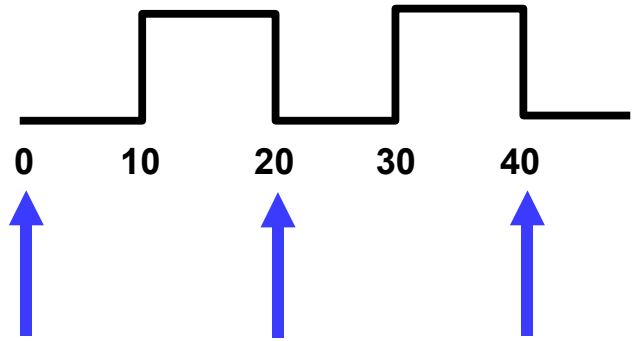



```
always @(INPUT, MIDDLE, OUTPUT)
  $display
    ("Zeit: %2.0f,  INPUT = %h,  MIDDLE = %h,  OUTPUT = %h",
    $time,  INPUT, MIDDLE, OUTPUT);
```

5. Schritt: Modellierung der Flip-Flops

TECHNISCHE UNIVERSITÄT DARMSTADT






```
always @(posedge CP)
  MIDDLE = INPUT; // Fehler!

always @(posedge CP)
  OUTPUT = MIDDLE; // Fehler!
```

```
Zeit:  0,  INPUT = 00,  MIDDLE = xx,  OUTPUT = xx
Zeit: 11,  INPUT = 00,  MIDDLE = 00,  OUTPUT = xx
Zeit: 20,  INPUT = ff,  MIDDLE = 00,  OUTPUT = xx
Zeit: 31,  INPUT = ff,  MIDDLE = ff,  OUTPUT = 00
Zeit: 40,  INPUT = aa,  MIDDLE = ff,  OUTPUT = 00
Zeit: 51,  INPUT = aa,  MIDDLE = aa,  OUTPUT = ff
```

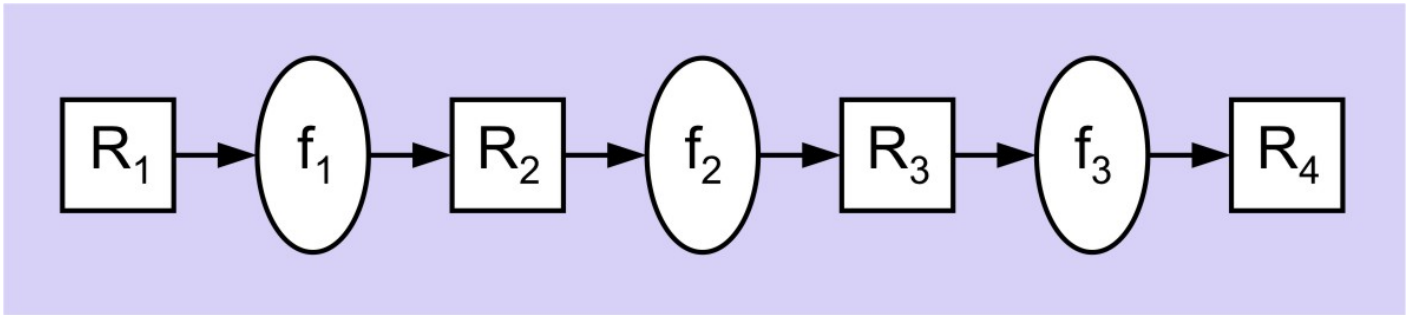
```
always @(posedge CP) //SIM
  MIDDLE = #1 INPUT;

always @(posedge CP)
  OUTPUT = #1 MIDDLE;
```

```
begin
  OUTPUT = MIDDLE;
  MIDDLE = INPUT;
end
```

```
  OUTPUT <= MIDDLE;
  MIDDLE <= INPUT;
end
```

Beispiel-Pipeline: Rahmenmodul

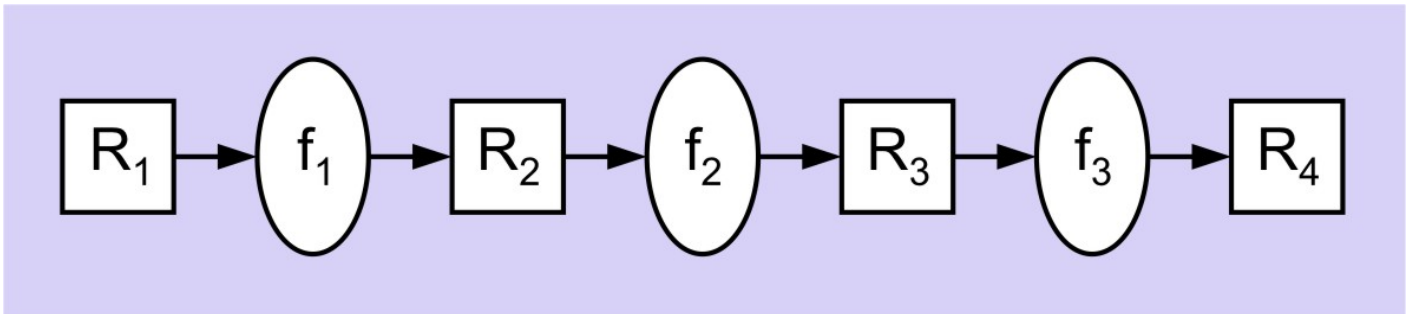
TECHNISCHE UNIVERSITÄT DARMSTADT




```
module pipeline #(

parameter    Low   = 10,        // Takt low
             High  =  5         // Takt high
);
reg          CLOCK;             // Takt

reg     [7:0] R1,               // Register 1
              R2,               // Register 2
              R3,               // Register 3
              R4;               // Register 4

integer      I;                 // Hilfsvariable
 ...
endmodule // pipeline
```

Beispiel-Pipeline: Takterzeugung

TECHNISCHE UNIVERSITÄT DARMSTADT

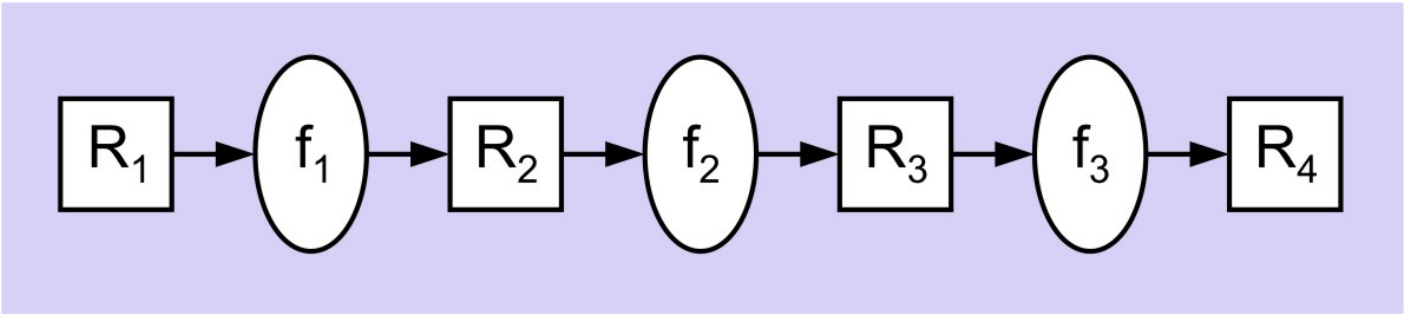



```
// Ein-Phasen-Takt
always begin
  #Low CLOCK <= 1;          // Takt low
  #High CLOCK <= 0;         // Takt high
end
```

Beispiel-Pipeline: Kombinatorische Logik

Führt eigentliche Rechnung aus

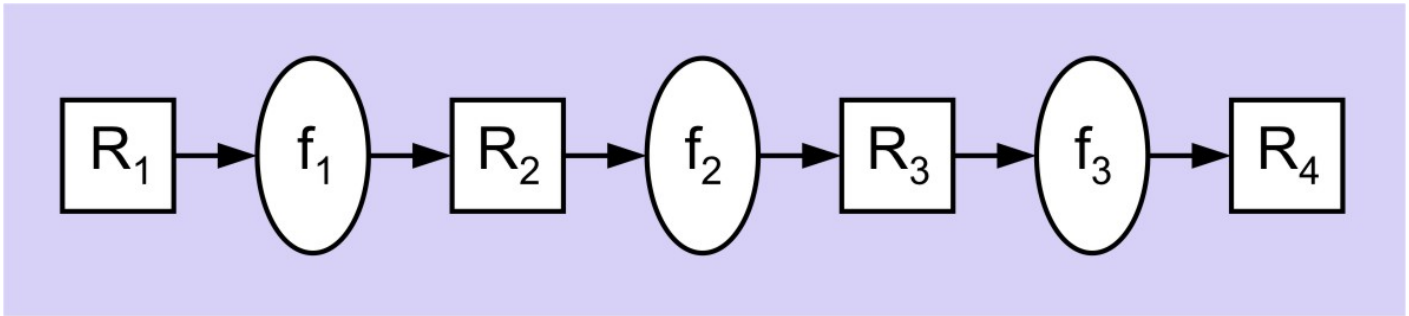
TECHNISCHE UNIVERSITÄT DARMSTADT




```
// Logik zwischen R1 und R2
function [7:0] f1 (input [7:0] IN);
  f1 = 2 * IN;
endfunction

// Logik zwischen R2 und R3
function [7:0] f2 (input [7:0] IN);
  f2 = IN + 5;
endfunction

// Logik zwischen R3 und R4
function [7:0] f3 (input [7:0] IN);
  f3 = IN * IN;
endfunction
```

Testrahmen

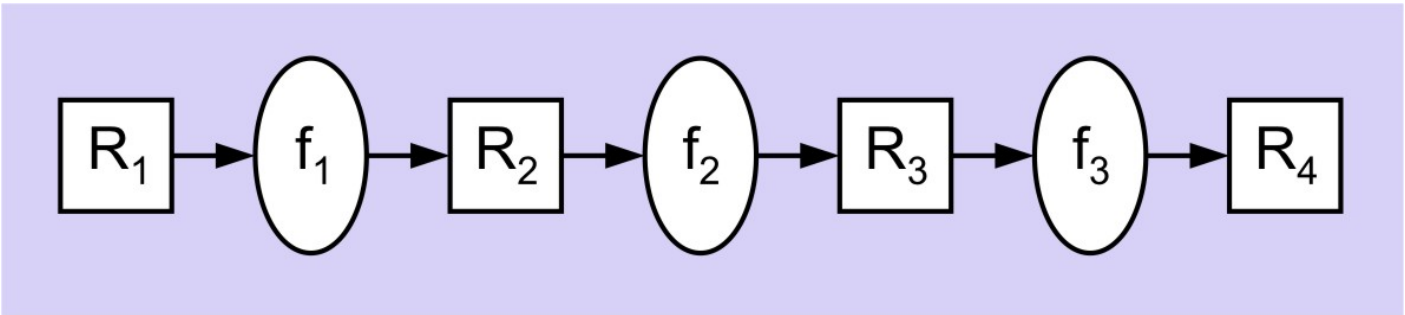
Hier in einem Modul (kürzer), besser: saubere Trennung in eigenem Modul




```
// Test-Ausgaben
always @(*)
  $display ("%4.0f   %b   %d   %d   %d   %d",
  $time, CLOCK, R1, R2, R3, R4);

// Ueberschrift, Test-Eingaben
 initial  begin
  $display ("Zeit  CLOCK   R1   R2   R3   R4\n");

  @(negedge CLOCK) R1 <= 1;      // R1 eingeben
  @(negedge CLOCK) R1 <= 2;      // R1 eingeben
  @(negedge CLOCK) R1 <= 3;      // R1 eingeben
  @(negedge CLOCK) R1 <= 4;      // R1 eingeben

  for(I=1;I<=5;I=I+1)            // Pipeline leeren
    @(posedge CLOCK);
  $finish;
end
```

Beispiel-Pipeline: Ablaufsteuerung

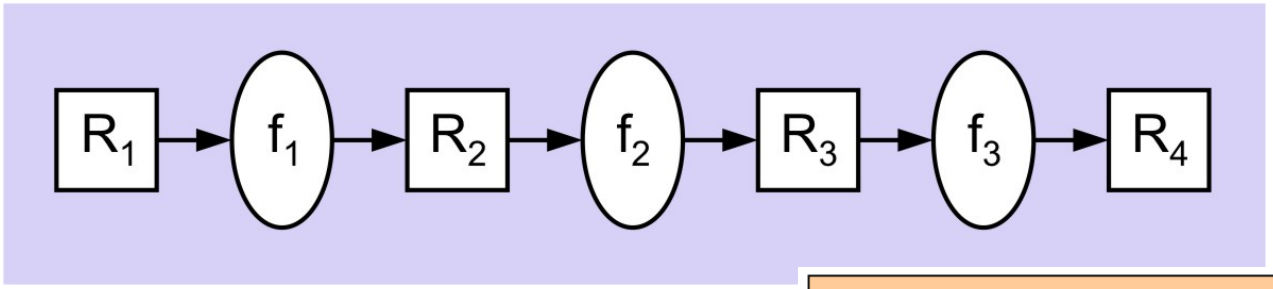
Hier in einem Modul (kürzer), besser: saubere Trennung in eigenem Modul



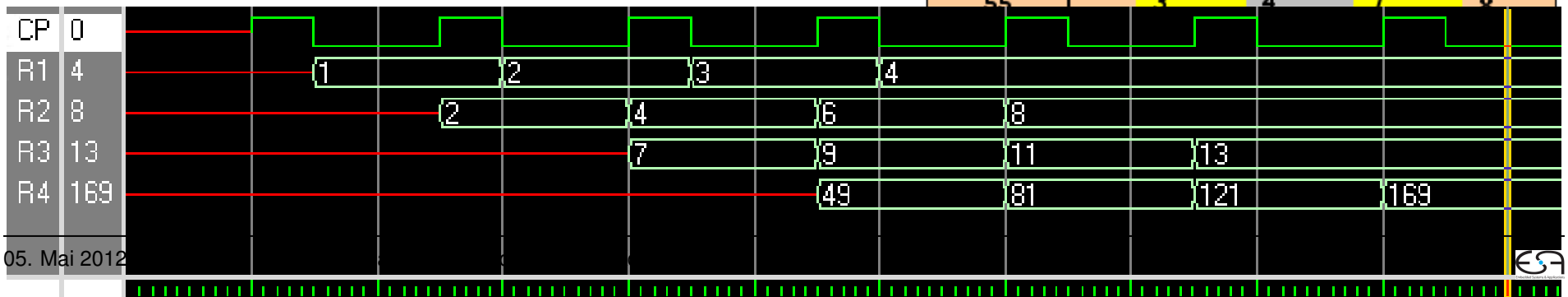

```
// Pipeline steuern und Funktionen berechnen
always @(posedge CLOCK) begin
   R2 <= f1(R1);
   R3 <= f2(R2);
   R4 <= f3(R3);
end
```

Beispiel-Pipeline: Ergebnisse

TECHNISCHE UNIVERSITÄT DARMSTADT

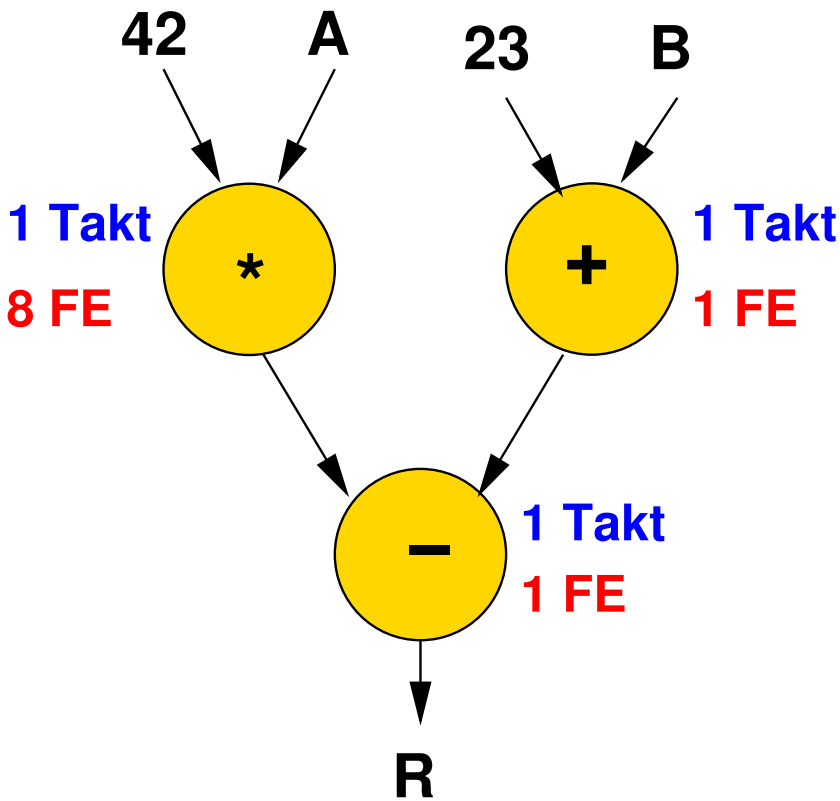


Zeit	CLOCK	R1	R2	R3	R4
10	1	x	x	x	x
15	0	x	x	x	x
15	0	1	x	x	x
25	1	1	x	x	x
25	1	1	2	x	x
30	0	1	2	x	x
30	0	2	2	x	x
40	1	2	2	x	x
40	1	2	4	7	x
45	0	2	4	7	x
45	0	3	4	7	x
55	1	3	4	7	x



Pipeline mit parallelen Operatoren

Berechne $R = 42 \cdot A - (23 + B)$

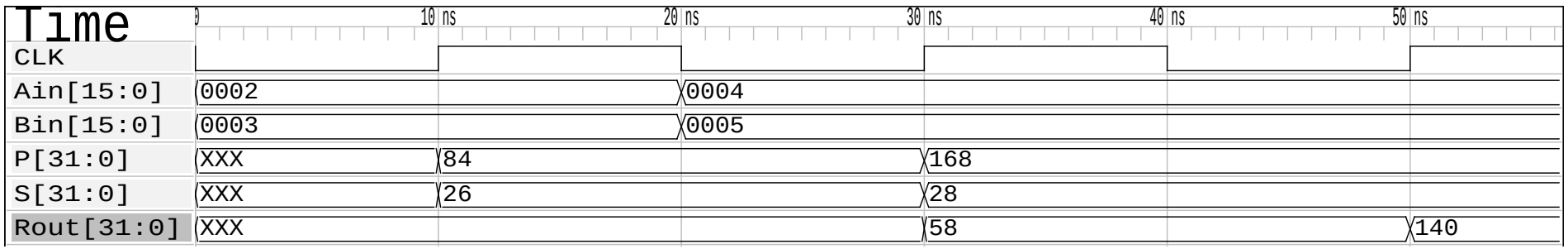



```
module compute
  (input              CLK,
   input       [15:0] A, B,
   output reg  [31:0] R);

   reg [31:0] P, S;

   always @(posedge CLK) begin
    P <= 42 * A;
    S <= 23 + B;
    R <= P - S;
   end

endmodule
```

- FE = Flächeneinheit
- Durchsatz: 1 Datum pro Takt
- Latenz: 2 Takte
- Fläche: 10 FE

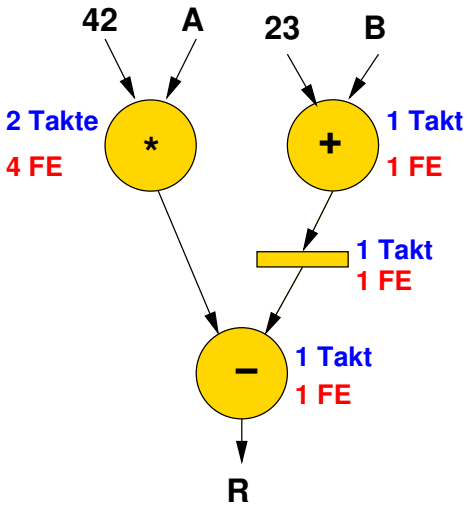
Pipeline mit parallelen Operatoren

Signalverlaufsdiagramm



Pipeline mit parallelen Operatoren

Berechne $R = 42 \cdot A - (23 + B)$



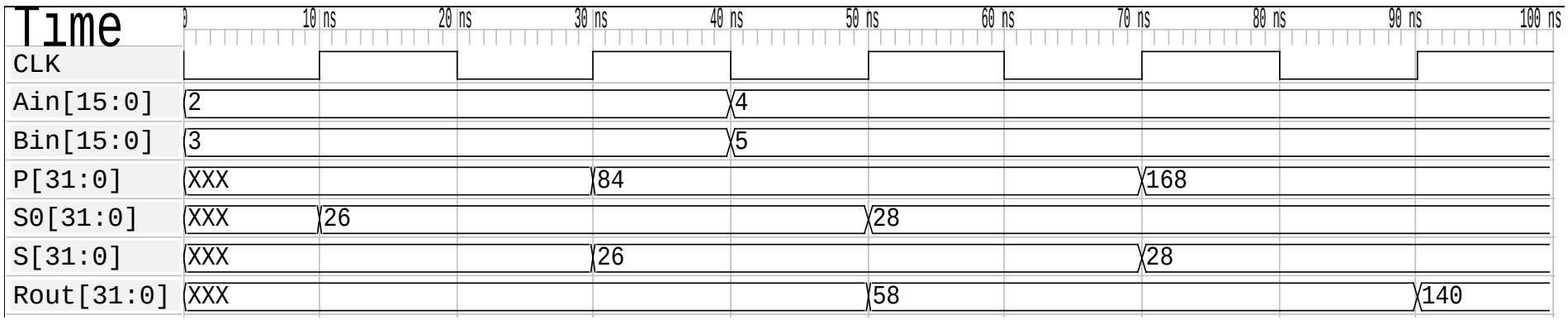

```
module compute
  (input               CLK,
   input        [15:0] A, B,
   output reg [31:0] R);

   reg  [31:0] S, S0;
   wire [31:0] P;

   // Langsamer und kleiner Multiplizierer
   mul2 MUL2 (CLK, A, 42, P);

   always @(posedge CLK) begin
    S0 <= 23 + B;
    S <= S0;       // Verzögerungsregister
    R <= P - S;
   end

endmodule
```

- Multiplizierer nun langsamer und kleiner
  - Braucht zwei Takte
  - Verzögerungsregister zum Synchronisieren
- Durchsatz: 1 Datum pro 2 Takte
- Latenz: 3 Takte
- Fläche: 7 FE

Pipeline mit parallelen Operatoren

Signalverlaufsdiagramm

TECHNISCHE UNIVERSITÄT DARMSTADT



Einführung in Computer Microsystems
Sommersemester 2012
4. Block: Einführung in die Logiksynthese

TECHNISCHE UNIVERSITÄT DARMSTADT

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Einführung

Logiksynthese

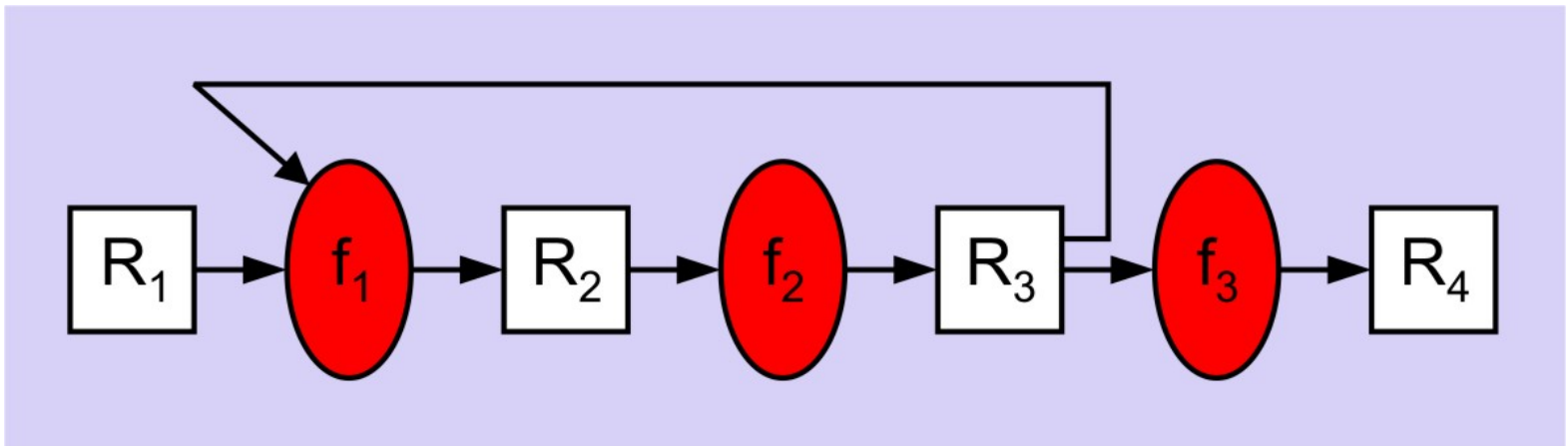
TECHNISCHE UNIVERSITÄT DARMSTADT

- Abbildung von RTL-Modell auf Gattermodell
  - Register-Transfer-Ebene auf Logikebene
- Wichtige Hersteller von Entwurfswerkzeugen
  - Für ASICs: Synopsys, Cadence
  - Für FPGAs: Synopsys, Mentor Graphics
  - Gibt aber auch noch diverse andere Anbieter

Logiksynthese

TECHNISCHE UNIVERSITÄT DARMSTADT

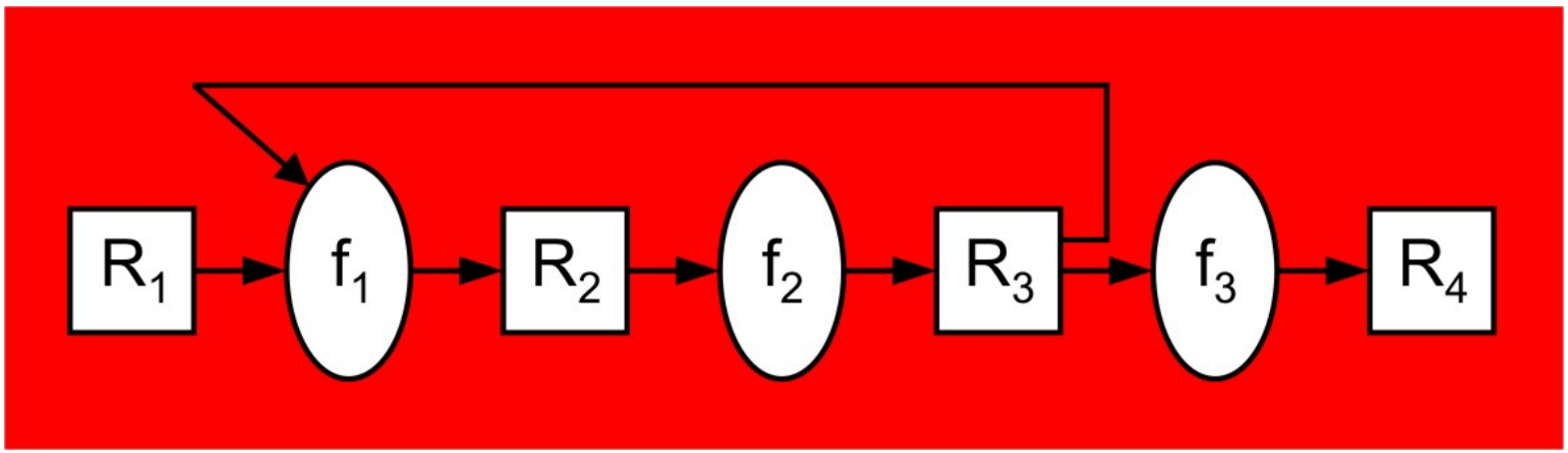
Optimiert im wesentlichen Logik zwischen getakteten Registern



Vergleich: High-Level-Synthese

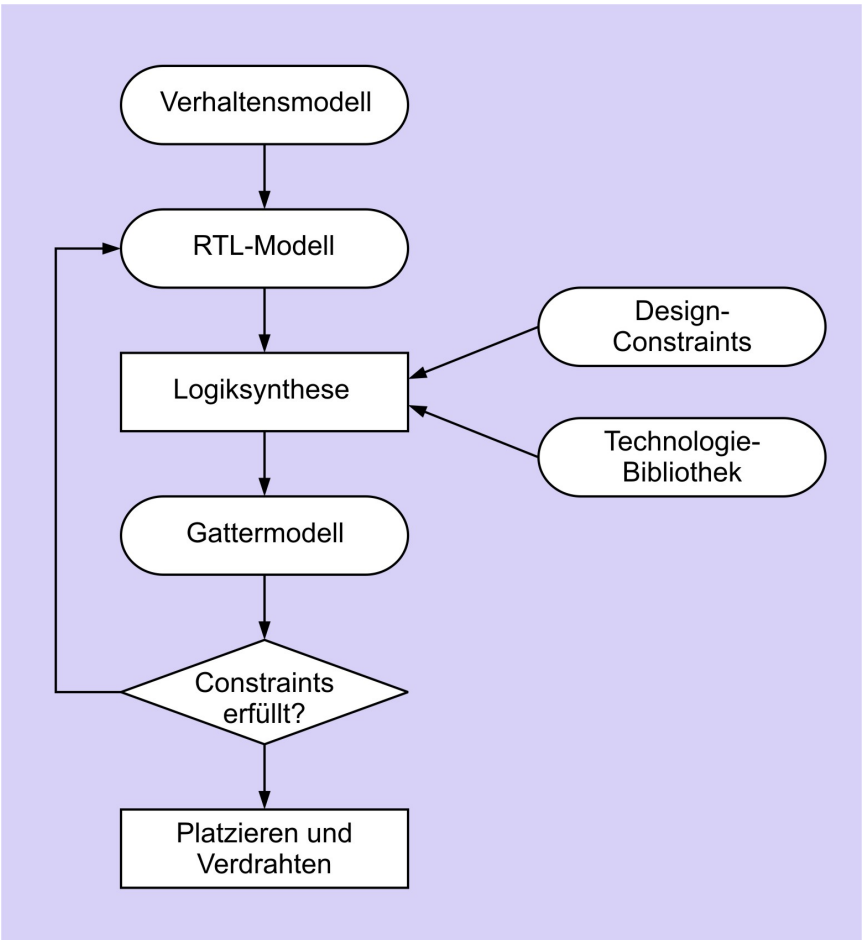
TECHNISCHE UNIVERSITÄT DARMSTADT

Beginnt oberhalb von RTL und optimiert über Taktgrenzen hinweg



Noch experimentell, nur eingeschränkte praktische Bedeutung

Ablauf der Logiksynthese



- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Vorteile der Logiksynthese

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kürzere Entwurfszeit
- Weniger fehleranfällig
- Anforderungen an Zeit und Fläche aufstellbar
- Portabilität zwischen verschiedenen Chip-Herstellern
- Leichtere Exploration des Entwurfsraumes
  - Wieviel langsamer, wenn 25% kleiner?
- Einheitlicher Entwurfsstil bei Team-Arbeit
- Leichtere Wiederverwendung von (Teil-)Entwürfen

Wichtigste synthetisierbare Verilog-Konstrukte

TECHNISCHE UNIVERSITÄT DARMSTADT

Signale und Variablen	`wire, reg`

prozedural	`always, begin, end,`
	`if, else,`
	`case,`
	`function, task, =, <=`

Struktur	`module,`
	`input, inout, output,`
	`parameter,`
	
	`assign`

Eingeschränkt: `for`

Einige nichtsynthetisierbare Verilog-Konstrukte

TECHNISCHE UNIVERSITÄT DARMSTADT

- initial: Stattdessen explizites Reset-Verhalten beschreiben
- Zeitkontrolle: `#` und `@` innerhalb von Block
  - Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie

➥Prä- und Post-Synthese-Simulationen können differieren

Synthetisierbare Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

arithmetisch	`*, /, +, -, %`
logisch	`!, &&, ||`
bit-weise	`~, &, |, ^, ^~, ~^`
Reduktion	`&, ~&, |, ~|, ^, ^~, ~^`
Relation	`>, <, >=, <=`
Gleichheit	`==, !=` aber kein `===` und `!==` mit `x` und `z`
Shift	`>>, <<, <<<, >>>`
Konkatenation	`{ }`
bedingt	`?:`

Syntheseergebnisse

Vorgehen

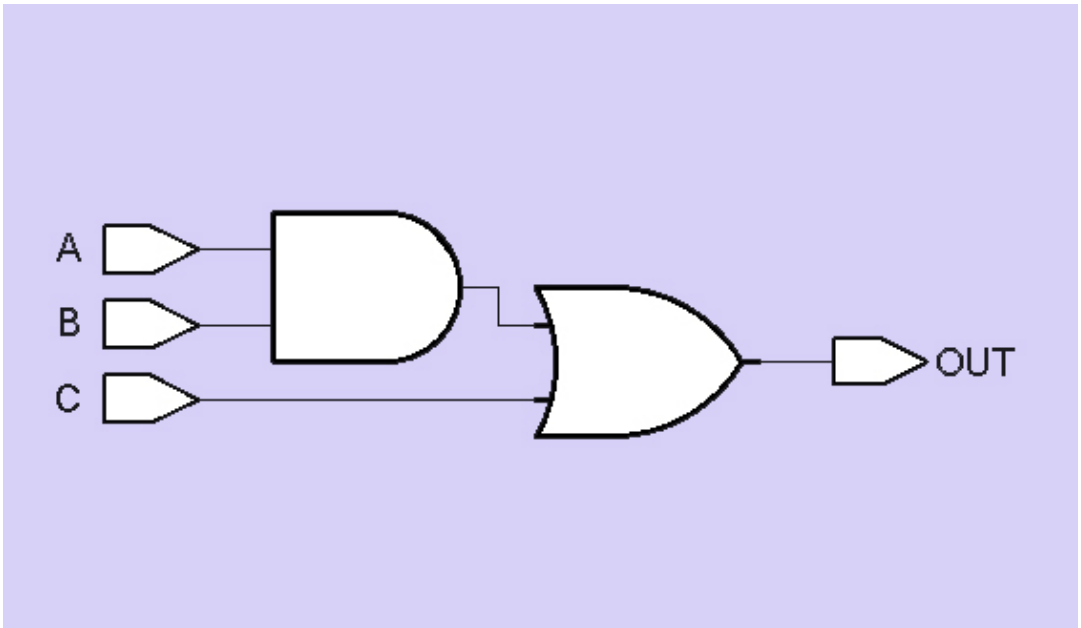
TECHNISCHE UNIVERSITÄT DARMSTADT

- Zunächst Abbildung auf allgemeines Gattermodell
- Noch weitgehend ohne Berücksichtigung der Zieltechnologie
- Reine Zwischendarstellung

Synthese einer `assign`-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

assign OUT = (A & B) | C;



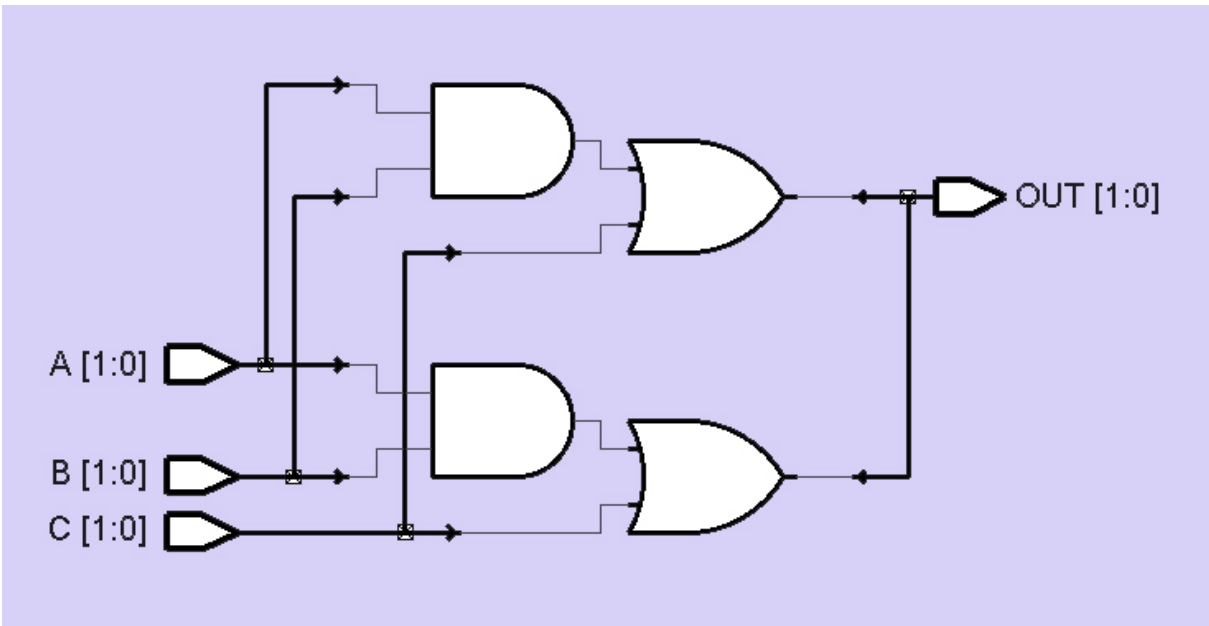
Annahme hier: Alle Signale 1b breit

Synthese einer `assign`-Anweisung

Hier: 2b breite Signale

TECHNISCHE UNIVERSITÄT DARMSTADT

```
assign OUT = (A & B) | C;
```

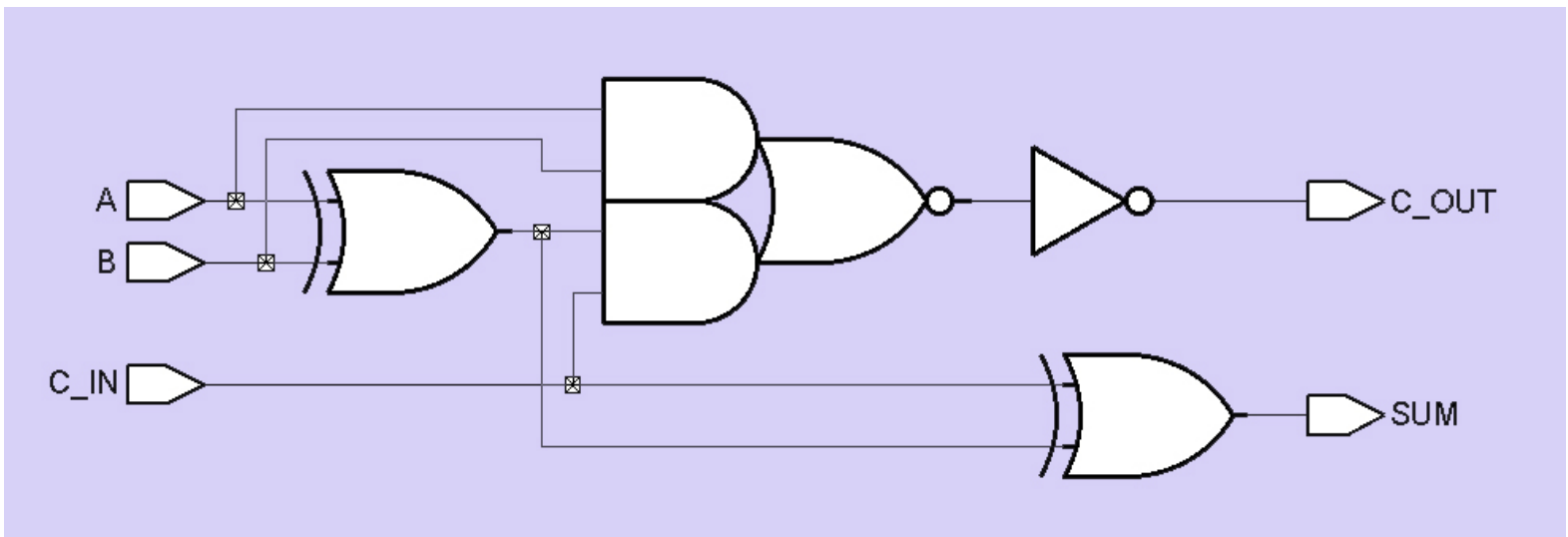


Synthese einer `assign`-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

assign {C_OUT, SUM} = A + B + C_IN

1b-Volladdierer



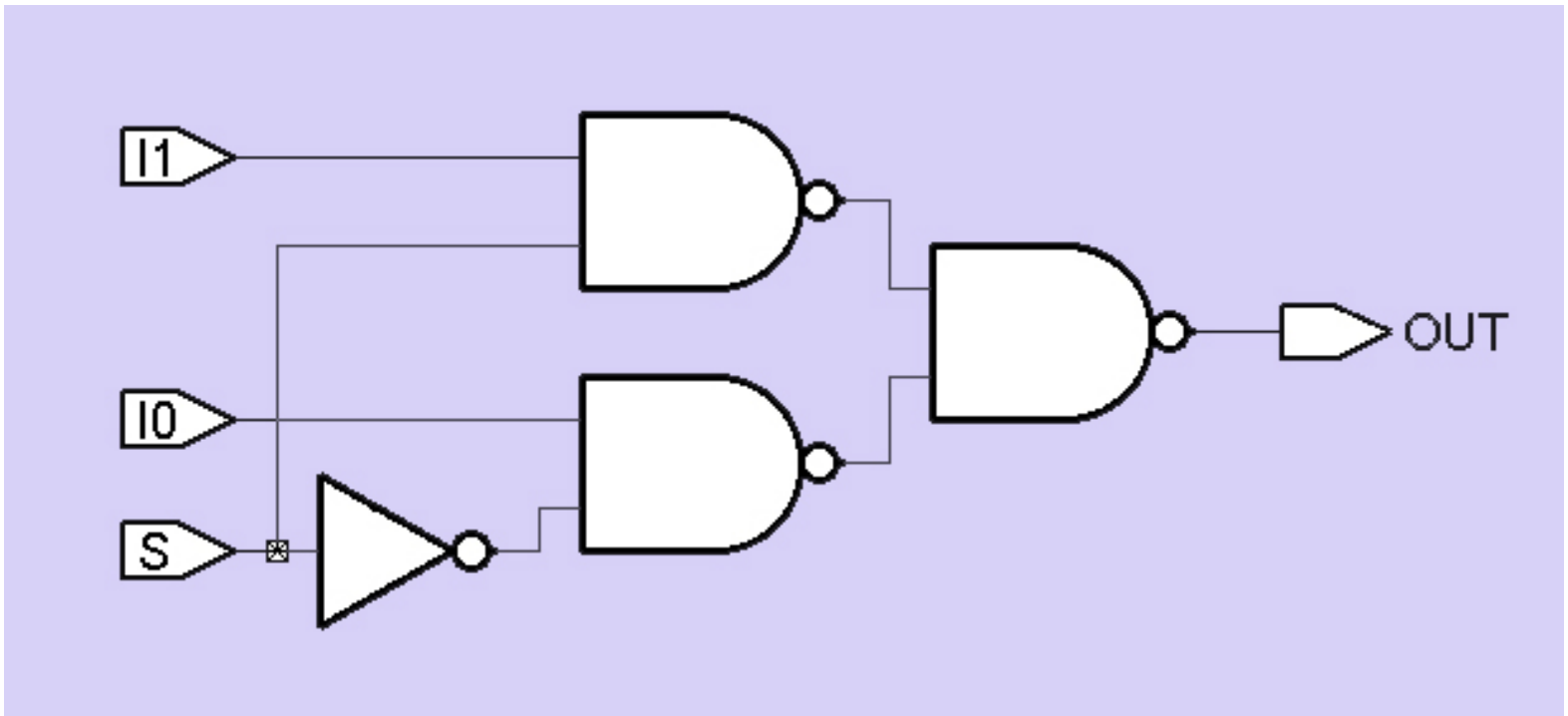
Merkwürdiges Gatter in der Mitte: AND-OR-INVERT (AOI),
sehr effizient in ASIC-Technologie realisierbar

Synthese einer `assign`-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

assign OUT = (S) ? I1 : I0

Multiplexer



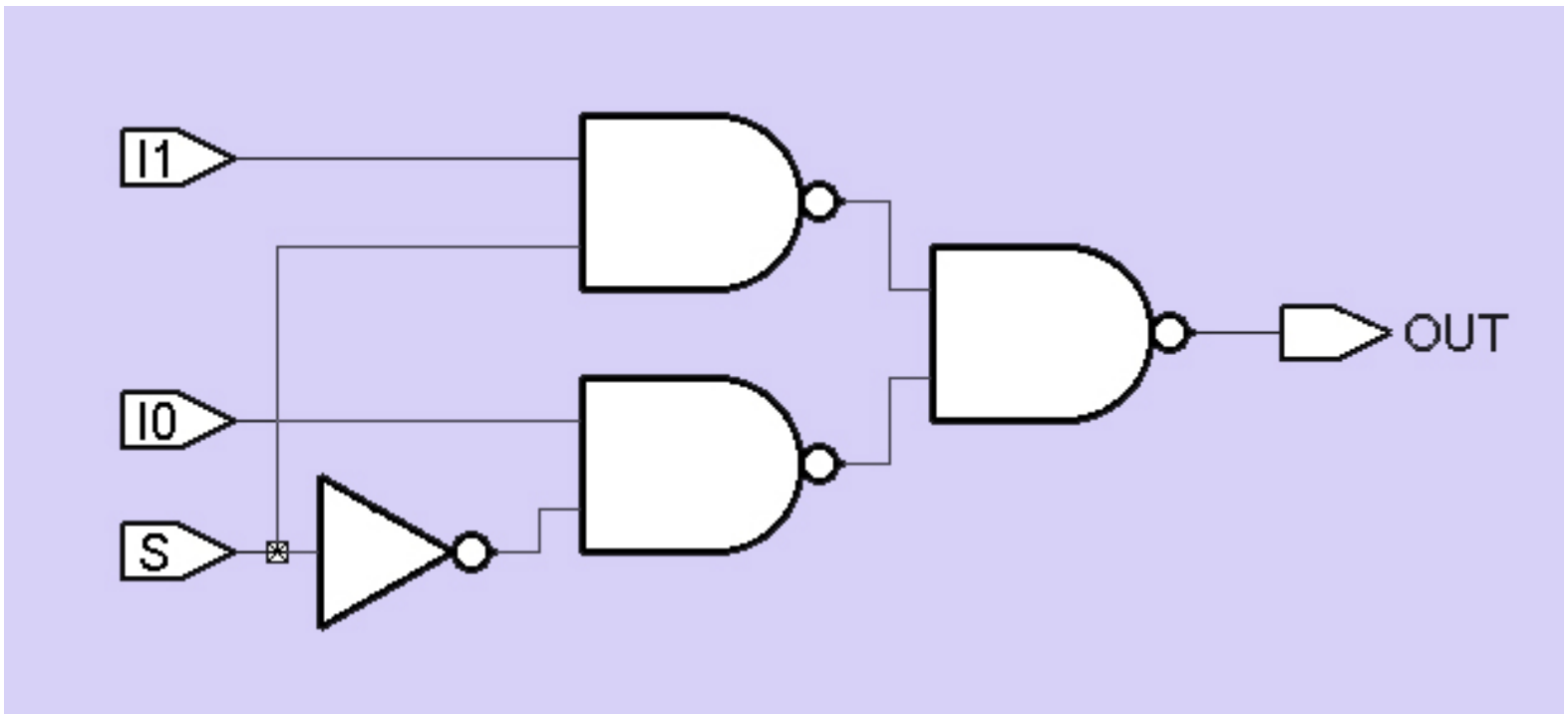
Synthese von `if/else` und `case`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if (S) OUT = I1;
else   OUT = I0;
```

```
case (S)
  0: OUT = I0;
  1: OUT = I1;
endcase
```

Multiplexer

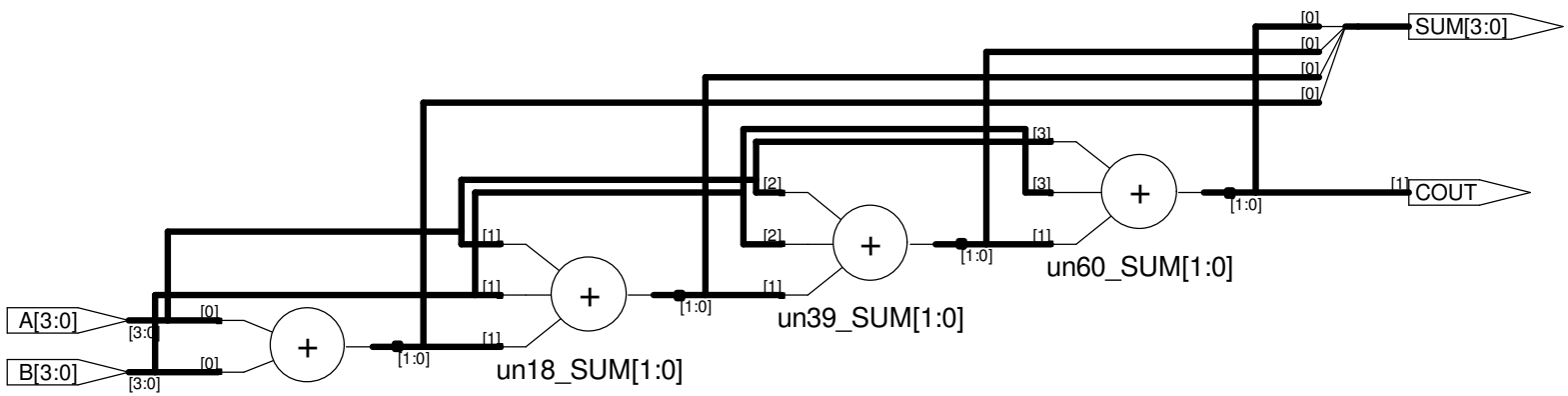


Synthese von `for`-Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

```
module unrolled_for (input      [3:0] A, B,
                     output reg [3:0] SUM,
                     output reg       COUT);
integer I;
reg C;

always @(*) begin
  C = 0;
  for (I = 0; I < 4; I = I + 1) begin
    {C, SUM[I]} = A[I] + B[I] + C;
  end
  COUT = C;
end
```



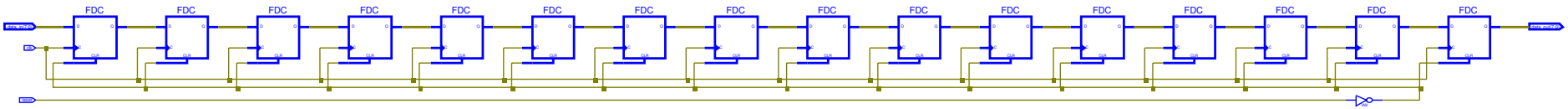
Besser: `generate` / `genvar`-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module generated_arraya_pipeline(data_out,data_in,clk,reset);
 parameter width = 8;
 parameter length = 16;
 output [width-1:0] data_out;
 input  [width-1:0] data_in;
 input clk,reset;
 reg [width-1:0] pipe [0:length-1];
 wire [width-1:0] d_in [0:length-1];
 assign d_in[0] = data_in;
 assign data_out = pipe[length-1];
 generate
  genvar k;
  for (k=1;k<=length-1;k=k+1) begin: W
   assign d_in[k] = pipe[k-1]; end
 endgenerate
 generate
  genvar j;
  for (j=0;j<=length-1;j=j+1)
   begin: stage
    always @(posedge clk or negedge reset) begin
     if (reset == 0) pipe[j] <= 0; else pipe[j] <= d_in[j]; end
   end
 endgenerate
endmodule
```

Ergebnis der Generierung

TECHNISCHE UNIVERSITÄT DARMSTADT



`generate` / `genvar` mit Instanziierungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module top_pads2 (pdata, paddr, ...);
  input [15:0]  pdata;                    // pad data bus
  inout [31:0]  paddr;                    // pad addr bus
  wire  [15:0]  data;                     // data bus
  wire  [31:0]  addr;                     // addr bus
  wire          wr;                       // Schreibsignal (gibt addr auf paddr-Pads aus)
   ...
  genvar i;
   ...                                                  // Erzeugt Instanznamen
                                                        //   dat[0].i1 bis dat[15].i1
  generate for (i=0; i<16; i=i+1) begin: dat
    IBUF i1 (.O(data[i]),  .I(pdata[i]));  end
  endgenerate

  generate for (i=0; i<32; i=i+1) begin: adr            // Erzeugt Instanznamen
    BIDIR b1 (.N2(addr[i]),  .N1(paddr[i]),  .WR(wr)); end   //  adr[0].b1 to adr[31].b1
  endgenerate
endmodule
```

Geht aber noch einfacher ...

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module top_pads3 (pdata, paddr, pctl1, pctl2,    pctl3,  pclk);
  input [15:0]  pdata;                          // pad data bus
  inout [31:0]  paddr;                          // pad addr bus
  wire  [15:0]  data;                           // data bus
  wire  [31:0]  addr;                           // addr bus
  wire          wr;                             // Schreibsignal (gibt addr auf paddr-Pads aus)
                                                //  Array-Instanznamen
                                                //  i[15] bis i[0]
  IBUF  i[15:0]  (.O(data), .pI(pdata));
  BIDIR b[31:0] (.N2(addr), .pN1(paddr), .WR(wr));          // Array-Instanznamen
                                                //  b[31] bis b[0]
endmodule
```

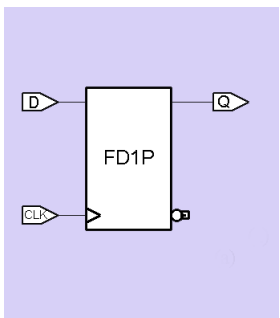
Speicherelemente

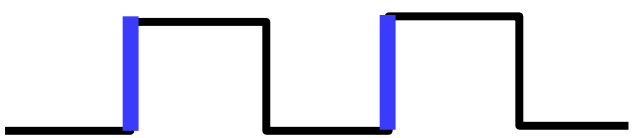
Synthese von Speicherelementen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
always @ (posedge CLK)
  Q <= D;
```

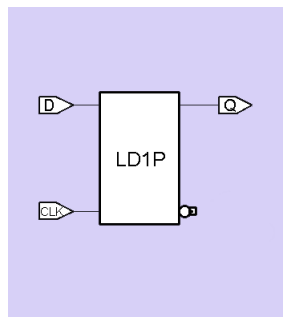
Vorderflankengesteuertes Flip-Flop

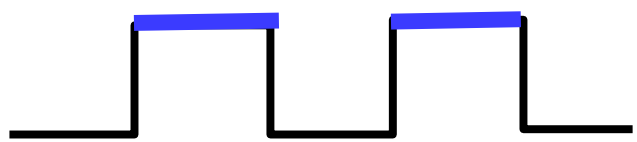




```
always @ (CLK or D)
  if  (CLK) Q = D;
```

Pegelgesteuertes Latch

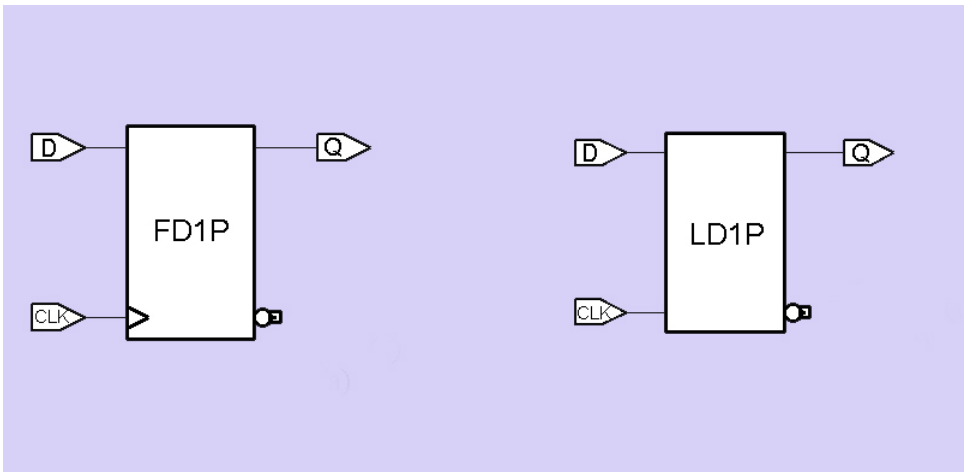




Wichtig: Unterschiede

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hardware-Register
  - (flankengesteuertes) Flip-Flop
  - (pegelgesteuertes) Latch



- Verilog-Datentyp `reg`

➥Nicht identisch

Register-Synthese

Grundlage der Modellierung: `always`-Blöcke

TECHNISCHE UNIVERSITÄT DARMSTADT

- flankengesteuerte always-Blöcke

```
always @(posedge CLK)
    ...

always @(posedge CLK, negedge nRESET)
    ...
```

- flankenfreie always-Blöcke

```
always @(CLK, D)
    ...

always @ (A, B, C_IN)
    ...
```

TECHNISCHE UNIVERSITÄT DARMSTADT

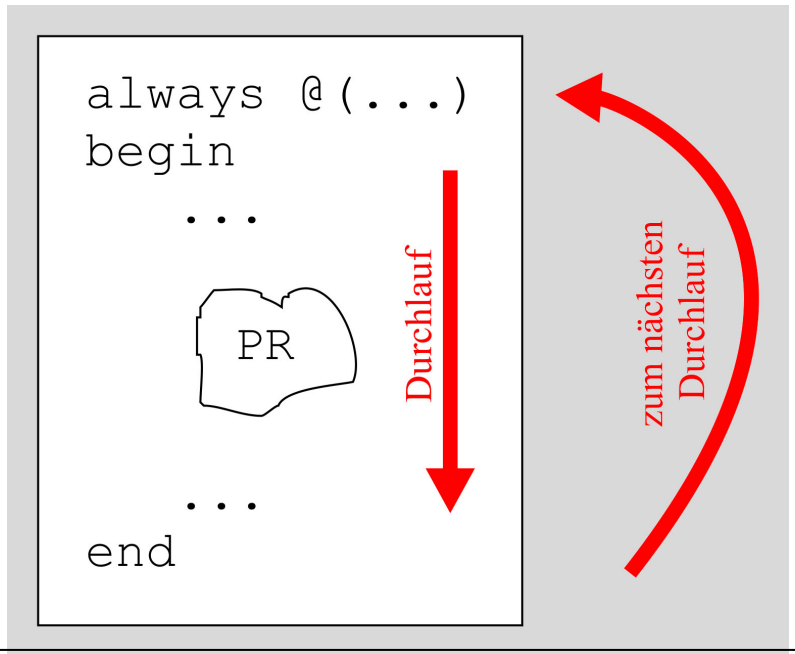
Potenzielle Register

Eine Variable `Q` ist ein potenzielles Register (PR), wenn sie in einem always-Block geschrieben wird (`Q = ...`, `Q <= ...`)..

Vollständigkeit

Ein potenzielles Register ist vollständig, falls es bei jedem Durchlauf des always-Blocks nicht-redundant geschrieben wird.

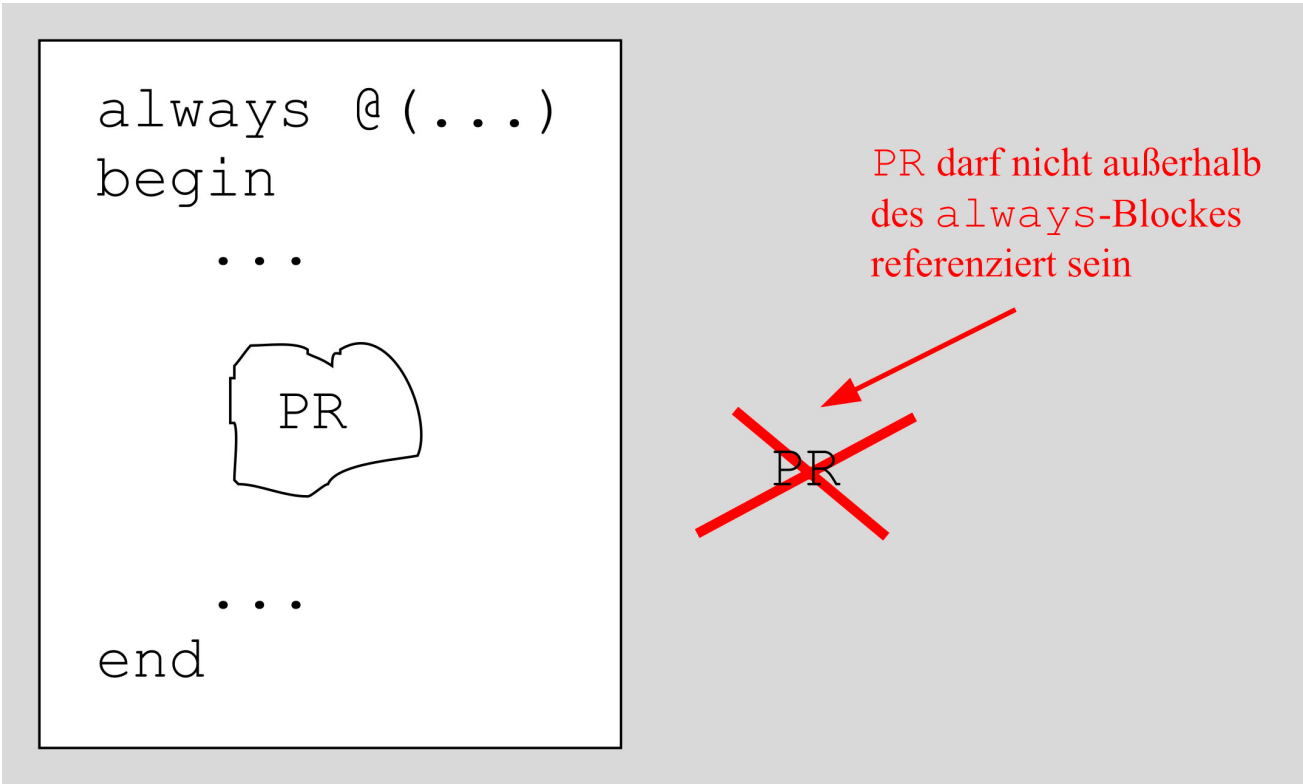
Nicht-redundant: Nicht mit sich selber, also nicht `Q = Q`



TECHNISCHE UNIVERSITÄT DARMSTADT

Räumliche Lokalität

Ein potenzielles Register ist räumlich lokal, wenn es nur innerhalb eines always-Blocks verwendet wird (lesend oder schreibend).



Beispiele: Räumliche Lokalität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module r_lokal_1 (
  input  wire A,
  output reg  C);

reg B;

always @ (A, B)
begin
  B = ~A;
  if  (B) C = A;
  else    C = ~A;
end
endmodule
```

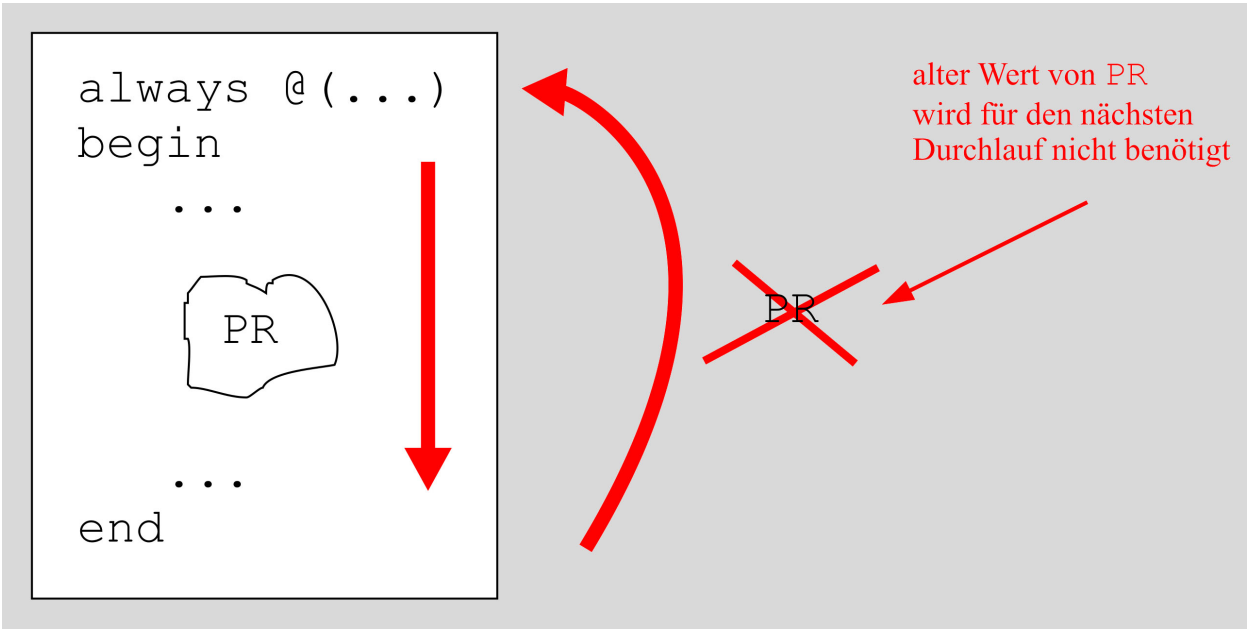
```
module r_lokal_2 (
  input wire A);

reg B, C;

always @ (A)
  C = A;

always @ (C)
  B = ~C;

endmodule
```

In beiden Modulen ist `B` räumlich lokal, nicht aber `C`.

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Lokalität

Ein potenzielles Register ist zeitlich lokal (kurz: lokal), falls es räumlich lokal ist und nie vor dem Schreiben gelesen wird.



Der alte Wert braucht also nicht zwischengespeichert zu werden.

Beispiele: Zeitliche Lokalität

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module z_lokal_1 (
  input  wire A,
  output reg  C);

reg TMP;

always @ (A, TMP)
begin
  TMP = ~A;
  C   = TMP;
end
endmodule
```

`TMP` ist zeitlich lokal

```
module z_lokal_2 (
  input  wire A, B,
  output reg  C);

reg TMP;

always @ (A, B, TMP)
begin
  if  (B) TMP = ~A;
  C   = TMP;
end
endmodule
```

`TMP` ist nicht zeitlich lokal

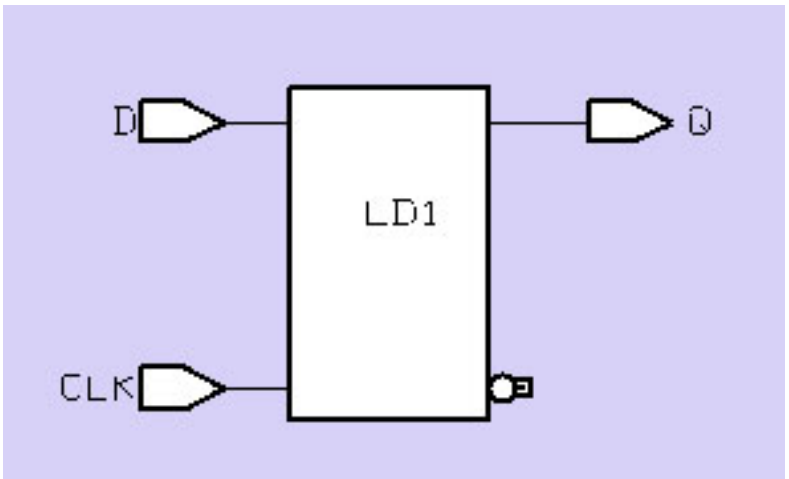
Flankenfreier Always-Block

Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
always @(CLK, D)
   if  (CLK) Q = D;
```



Latch wegen unvollständigem `Q`

Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

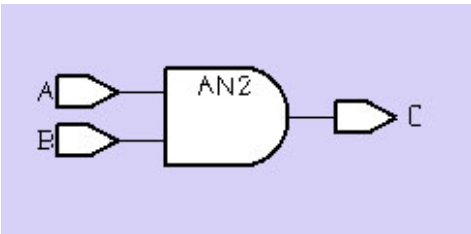
TECHNISCHE UNIVERSITÄT DARMSTADT

```
always @(CLK, D)
  if  (CLK) Q = D;
  else      Q = 0;
```

Latch vermieden da `Q` nun vollständig

```
always @(A, B)
  if  (A) C = B;
  else    C = 0;
```

Signalnamen wie `CLK` irrelevant. Kombinatorische Logik!



Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module decoder (
  input  wire [3:0]  I;
  output reg  [9:0]  DECIMAL);
always @(I)
  case (I)
    4'h0: DECIMAL = 10'b0000000001;
    4'h1: DECIMAL = 10'b0000000010;
    4'h2: DECIMAL = 10'b0000000100;
    4'h3: DECIMAL = 10'b0000001000;
    4'h4: DECIMAL = 10'b0000010000;
    4'h5: DECIMAL = 10'b0000100000;
    4'h6: DECIMAL = 10'b0001000000;
    4'h7: DECIMAL = 10'b0010000000;
    4'h8: DECIMAL = 10'b0100000000;
    4'h9: DECIMAL = 10'b1000000000;
  endcase
endmodule
```

- Latch wegen unvollständigem `case`
- Wie vollständig formulieren?
- Durch Angabe von `default`

Diskussion

TECHNISCHE UNIVERSITÄT DARMSTADT

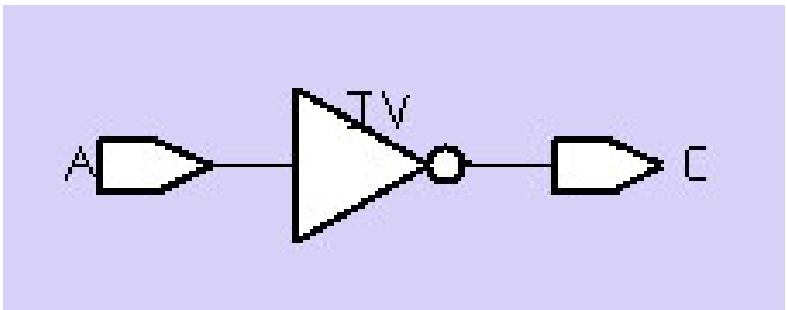
- Ein unvollständiges potenzielles Register (PR) erzeugt zunächst ein Latch
- Ist das PR jedoch lokal, wird das Latch aber anschließend wegoptimiert
- Auf Nummer sicher gehen!
- Damit PR kein Latch wird
  - PR vollständig beschreiben
  - oder PR nur lokal verwenden

Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module z_lokal (
   input  wire A,
   output reg  C);

reg TMP;

always @ (A, TMP)
begin
   TMP = ~A;
   C   = TMP;
end
endmodule
```



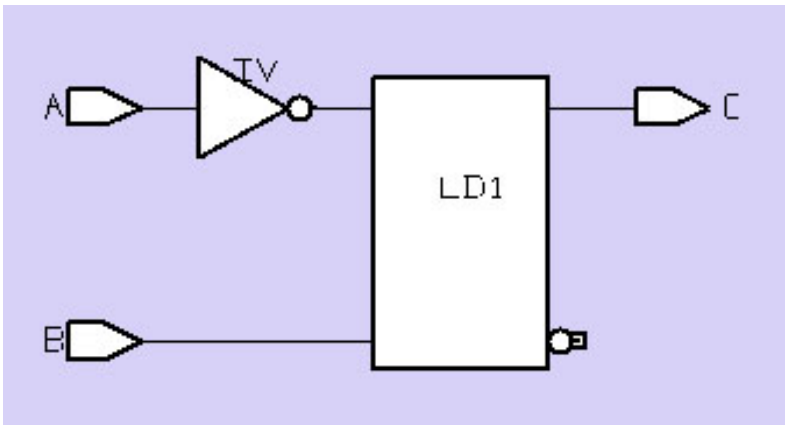
Latch wird vermieden, da `TMP` vollständig ist (und zeitlich lokal).

Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module lokal (
   input  wire A, B,
   output reg C);

reg TMP;

always @ (A, B, TMP)
begin
   if  (B) TMP = ~A;
   C    = TMP;
end
endmodule
```



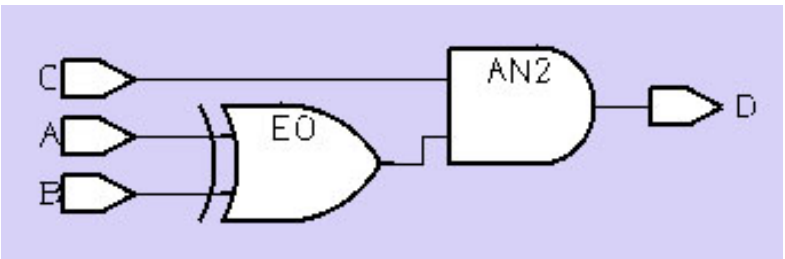
Latch entsteht, da `TMP` unvollständig ist und nicht zeitlich lokal.

Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module lokal(
  input  wire A, B, C,
  output reg  D);

reg TMP;

always @ (TMP, A, B, C)
  if  (C) begin
    TMP = A + B;
    D = TMP;
  end
  else D = 0;

endmodule
```



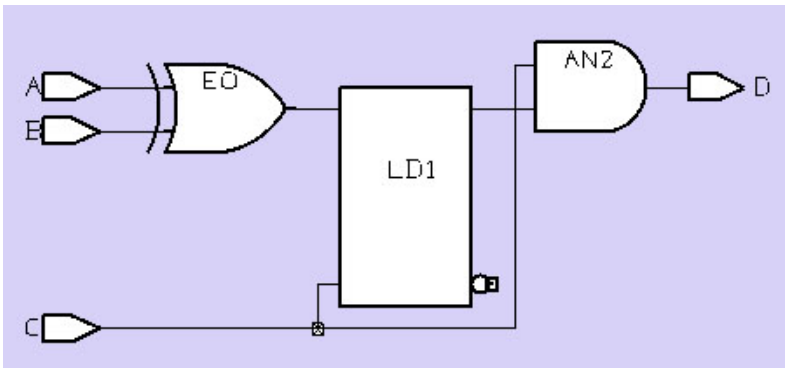
Latch wird vermieden, da `TMP` zwar unvollständig ist, aber räumlich und zeitlich lokal.

Beispiele: Logik oder Latch?

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module lokal (
  input  wire A, B, C,
  output reg  D);

reg TMP;

always @ (TMP, A, B, C)
  if  (C) begin
    D = TMP;
    TMP = A + B;
  end
  else D = 0;

endmodule
```



Latch entsteht, da `TMP` unvollständig ist und nur räumlich, nicht aber zeitlich lokal ist.

Richtlinien

Flankenfreier `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zur Vermeidung von Latches
  - PR vollständig beschreiben
  - oder nur lokal verwenden
- Verilog-Funktionen ergeben kein Latch
  - Sie haben keinen internen Zustand
  - Keine globalen oder static-Variablen wie in z.B. in Java
- In flankenfreien `always`-Blöcken immer alle Lesevariablen in Aktivierungsliste
  - **always** @(*)
- Hier immer die blockende Zuweisung = verwenden!

Getakteter Always-Block

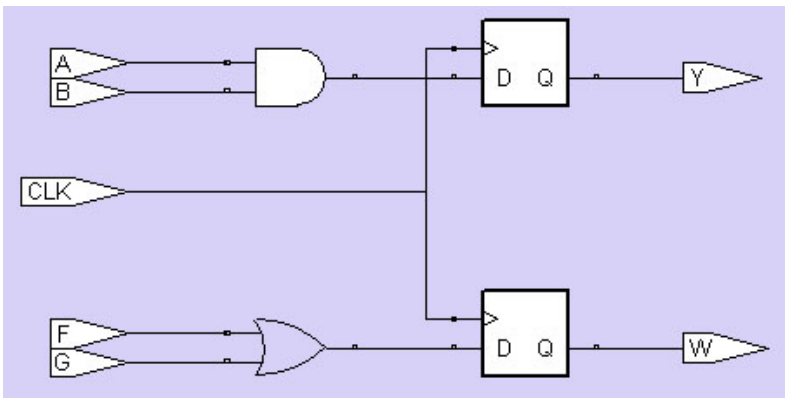
Flip-Flop

Getakteter `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mit `posedge` oder `negedge` in der Aktivierungsliste
- Jedes nicht-lokale potenzielle Register wird Flip-Flop
- Vollständigkeit ist nun irrelevant.

```
always @(posedge CLK)
begin
  Y <= A & B;
  W <= F | G;
end
```

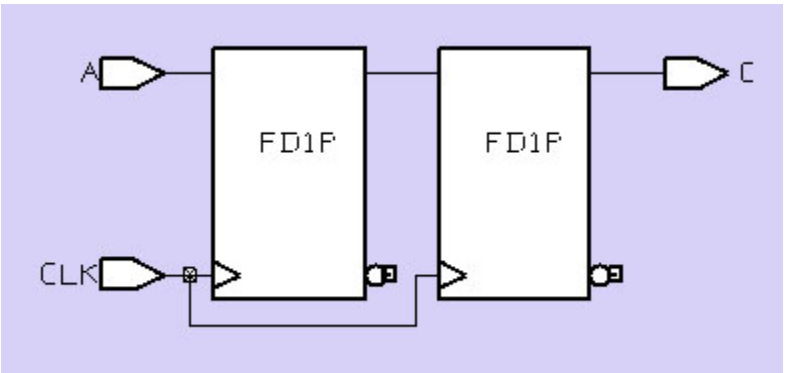


Flip-Flop

Getakteter `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module lokal (
   input  wire CLK, A,
   output reg  C);

reg B;

always @ (posedge CLK)
begin
   B <= A;
   C <= B;
end

endmodule
```



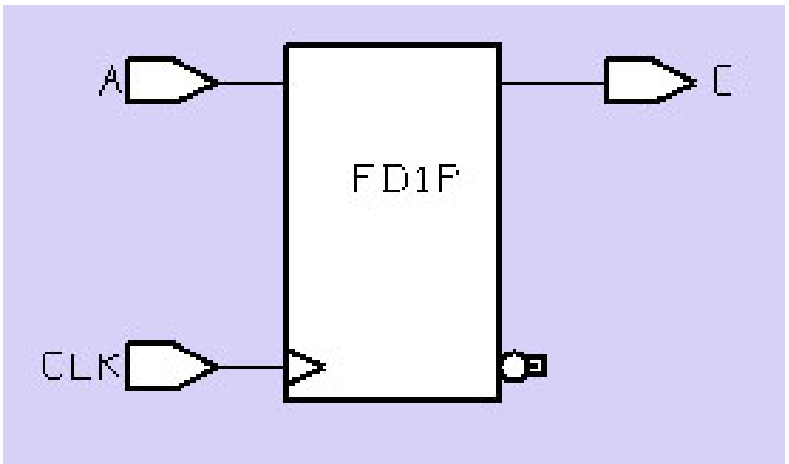
Für `B` entsteht ein Flip-Flop, da B zwar räumlich, nicht aber zeitlich lokal ist.

Flip-Flop

Getakteter `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module lokal (
   input  wire CLK, A,
   output reg  C);

reg B;

always @ (posedge CLK)
begin
   B  = A;
   C <= B;
end

endmodule
```



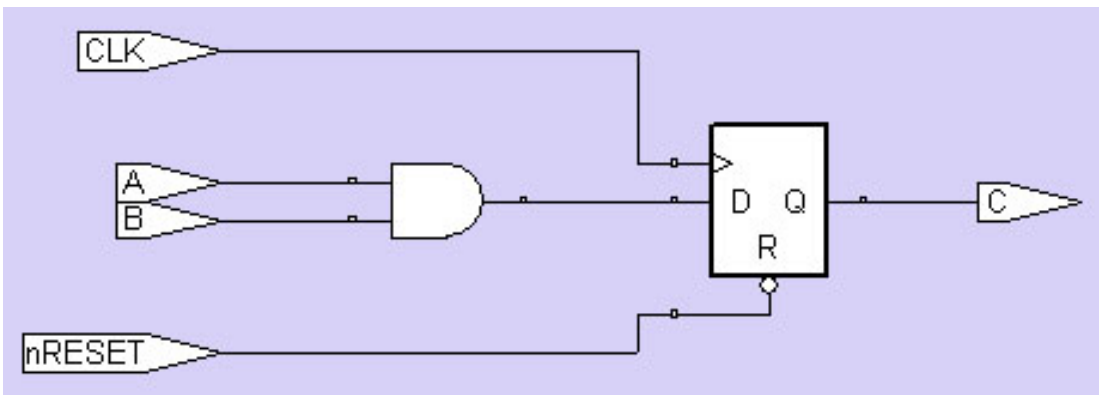
Für `B` entsteht kein Flip-Flop, da B räumlich und zeitlich lokal ist.

Flip-Flop mit Reset

Getakteter `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module FF (
  input  wire CLK, nRESET, A, B,
  output reg  C);

always @(  posedge CLK,
           negedge nRESET)
  if  (!nRESET)   C <= 0;
  else            C <= A & B;

endmodule
```



Flip-Flop mit asynchronem Reset

Zuweisungen

Richtlinien für Zuweisungen

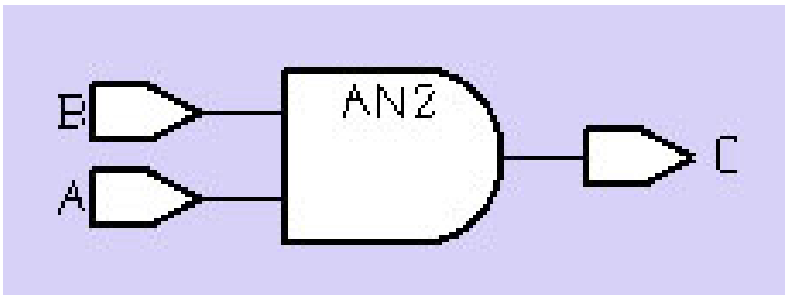
TECHNISCHE UNIVERSITÄT DARMSTADT

- Für spätere Flip-Flops
  - nichtblockende Zuweisung
- Für kombinatorische Logik, lokale Hilfsvariablen und Latches
  - blockende Zuweisung
- Für die Synthese aber unerheblich
  - Es gelten die Regeln von Vollständigkeit und Lokalität
- Trotzdem Richtlinien befolgen
  - Sonst Prä-Synthese und Post-Synthese Simulation nur schwer vergleichbar

So nicht: Nichtblockende Zuweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
always @ (A, B)
begin
   C <= 0;
   if  (B) C <= A;
end
```

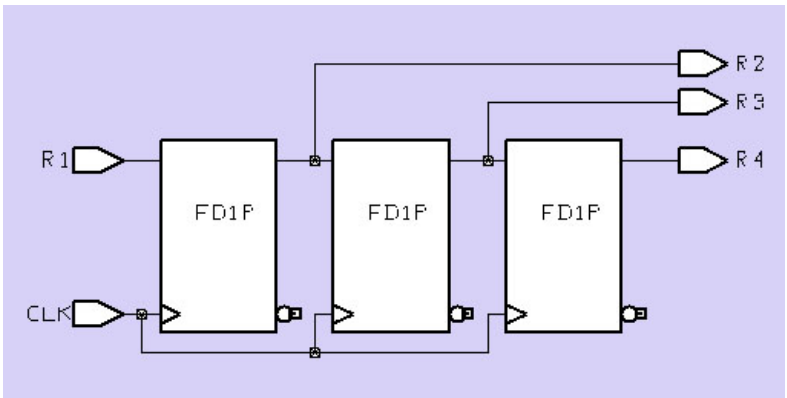


Trotz <= kombinatorische Logik, da c vollständig beschrieben und lokal verwendet.

So nicht: Blockende Zuweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
always @ (posedge CLK)
  R2 = R1;

always @ (posedge CLK)
  R3 = R2;

always @ (posedge CLK)
  R4 = R3;
```



- In Simulation: Zufällige Ausführungsreihenfolge
  - R2=, R3=, R4 oder R4=, R2=, R3= ...
- In Synthese: Immer Schieberegister aus Flip-Flops
  - R2, R3, R4 nicht nur lokal verwendet

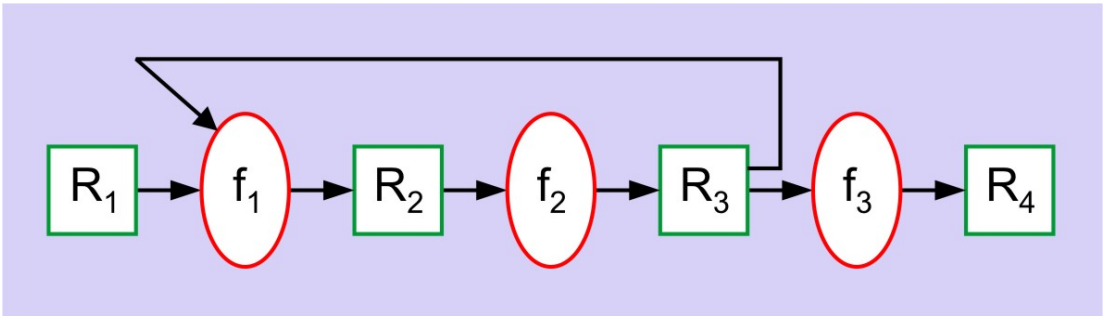
TECHNISCHE
UNIVERSITÄT
DARMSTADT

Kombinatorische Logik und Register

Trennung von kombinatorischer Logik und Registern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Werden in RTL ja getrennt betrachtet
- Also auch im Verilog-Modell sauber trennen
- Beispiel
  - Diesmal keine Pipeline!
  - Teil eines digitalen Weckers
  - Benutzer soll Anzahl von Wecktönen wählen können (zwischen 1 und 100)



1. Möglichkeit: Getrennte `always`-Blöcke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module alarm_count (
  input  wire       CLOCK,              // Takt
                    RESET,              // Reset
                    INCALARM,           // =1 wenn Taste gedrückt
  output reg  [6:0] CURRENTCOUT);       // jetziger Zustand (Alarmanzahl bis 100)

// interne Variable
reg [6:0] NEXTCOUNT;                    // naechster Zustand

// kombinatorische Uebergangsfunktion, die bei jeder Aenderung
// des jetzigen Zustandes CURRENTCOUNT oder der Eingabe INCALARM
// den naechsten Zustand NEXTCOUNT vorlaeufig berechnet

always @(CURRENTCOUT, INCALARM) begin
  NEXTCOUNT = CURRENTCOUT;
  if  (INCALARM == 1)          NEXTCOUNT = CURRENTCOUT+1;
  if  (NEXTCOUNT > 100)        NEXTCOUNT = 100;
end

// naechsten Zustand mit jedem Takt endgueltig zum jetzigen machen;
// synchrones Reset

always @(posedge CLOCK)
if    (RESET == 1)    CURRENTCOUT <= 0;
else                  CURRENTCOUT <= NEXTCOUNT;

endmodule
```

2. Möglichkeit: Logik als Funktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module alarm_count (
  input  wire       CLOCK,              // Takt
                    RESET,              // Reset
                    INCALARM,           // =1 wenn Taste gedrückt
  output reg  [6:0] CURRENTCOUT);       // jetziger Zustand (Alarmanzahl bis 100)

// kombinatorische Uebergangsfunktion, die bei jeder Aenderung des jetzigen
// Zustandes CURRENTCOUT oder der Eingabe INCALARM
// den naechsten Zustand NEXTCOUNT vorlaeufig berechnet

function [6:0] NEXTCOUNT(
  input  [6:0] CURRENTCOUT,
  input        INCALARM);

begin
  NEXTCOUNT = CURRENTCOUT;
  if  (INCALARM == 1)       NEXTCOUNT = CURRENTCOUT+1;
  if  (NEXTCOUNT > 100)     NEXTCOUNT = 100;
end
endfunction

// naechsten Zustand mit jedem Takt endgueltig zum jetzigen machen;
// synchrones Reset

always @(posedge CLOCK)
if    (RESET == 1)  CURRENTCOUT <= 0;
else                CURRENTCOUT <= NEXTCOUNT (CURRENTCOUT, INCALARM);

endmodule
```

3. Möglichkeit: Keine saubere Trennung

Alles in einem `always`-Block

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module alarm_count (
  input  wire       CLOCK,              // Takt
                    RESET,              // Reset
                    INCALARM,           // =1 wenn Taste gedrückt
  output reg  [6:0] CURRENTCOUT);       // jetziger Zustand (Alarmanzahl bis 100)

// interne Variable
reg [6:0] NEXTCOUNT;                    // naechster Zustand

// mit jedem Takt naechsten Zustand kombinatorisch in NEXT berechnen
// und dann in PRESENT zum jetzigen machen;

always @(posedge CLOCK) begin
  NEXTCOUNT = CURRENTCOUT;
  if  (INCALARM == 1)       NEXTCOUNT = CURRENTCOUT+1;
  if  (NEXTCOUNT > 100)     NEXTCOUNT = 100;

  if  (RESET == 1)          CURRENTCOUT <= 0;
  else                      CURRENTCOUT <= NEXTCOUNT;
end

endmodule
```

- Kürzer, aber nicht mehr sauber getrennt
- Nachteil: Wird bei Erweiterung leicht unübersichtlich
- In der Industrie verpönt, dort i.d.R. strikte Trennung

Zusammenfassung: Potenzielle Register

Zusammenfassung Definitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- getakteter `always`-Block: **always** @(**posedge** CLK)...
- flankenfreier `always`-Block: **always** @(CLK, D)...
- `PR` ist potenzielles Register, falls `PR` in einem `always`-Block geschrieben wird
- `PR` ist vollständig, wenn es in jedem Durchlauf eines `always`-Blockes geschrieben wird
- `PR` ist räumlich lokal, wenn es nur innerhalb eines `always`-Blockes auftritt
- Ein räumlich lokales `PR` ist zeitlich lokal (kurz: lokal), falls es nie vor dem Schreiben gelesen wird

Zusammenfassung: Getaktete `always`-Blöcke

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jedes nicht-lokale PR wird ein getaktetes Flip-Flop
- An solche Flip-Flops wird mit <= zugewiesen
- An kombinatorische Hilfsvariablen mit =
- Spätere Flip-Flops werden an nur einer Stelle geschrieben
  - Ausgenommen der Reset, der steht extra
- Jedes Flip-Flop bekommt bei Reset einen definierten Wert

Zusammenfassung: Flankenfreie `always`-Blöcke

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein vollständiges oder lokales PR wird kombinatorische Logik
- Ein nicht-lokales und unvollständiges PR wird ein Latch
- Es wird stets blockend mit = zugewiesen
- Die Aktivierungsliste enthält alle Lesevariablen des `always`-Blockes
- Ein Takt in der Aktivierungsliste wird in der Synthese wie jede Variable auch behandelt

Parallelität

Parallelität

TECHNISCHE UNIVERSITÄT DARMSTADT

- In konventionellen Programmiersprachen wie z.B. Pascal, C
  - Anweisungen werden der Reihe nach bearbeitet
  - Programmzähler zeigt auf aktuelle Anweisung
  - Es gibt nur einen Kontrollfluß
- In HDLs und realen Schaltungen
  - Alle Komponenten arbeiten parallel
  - Z.B. kann eine Taktflanke eine Vielzahl von gleichzeitigen Aktionen auslösen
    - Modelliert durch parallele `always`-Blöcke

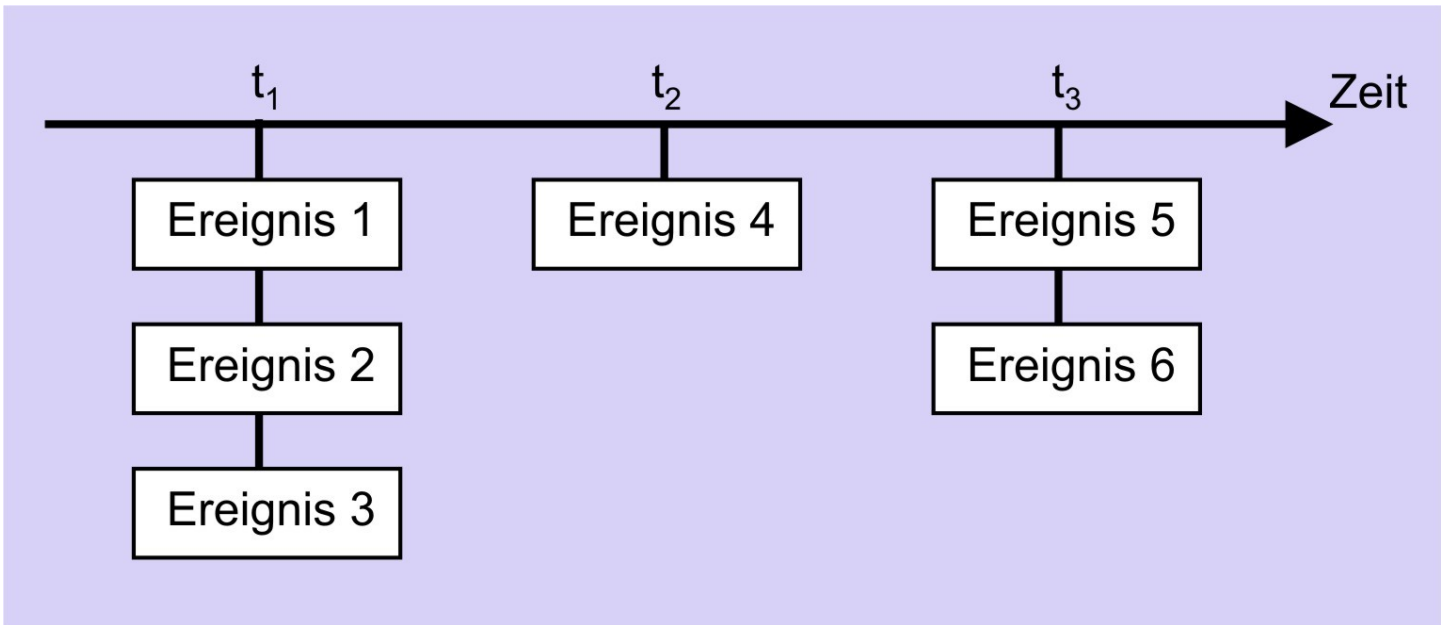
Modellierung von Parallelität in Verilog

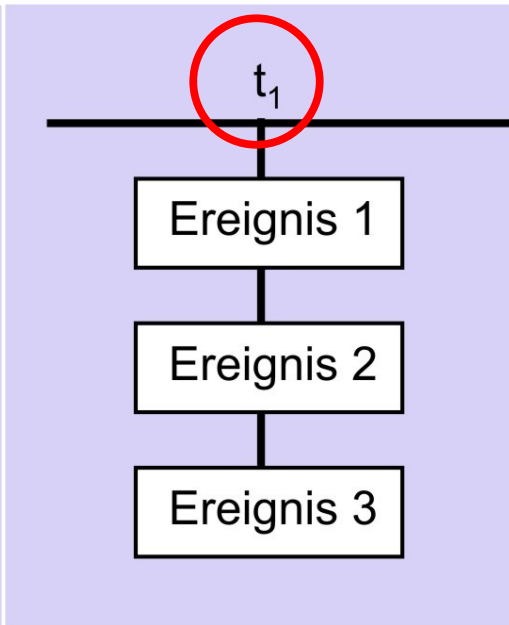
TECHNISCHE UNIVERSITÄT DARMSTADT

- Instanzen von Modulen
- `always`- und `initial`-Blöcke
- ständige Zuweisungen (*continuous assignments*)
- nichtblockende Zuweisungen
- Mischformen

Ereignisgesteuerte Simulation der Parallelität

TECHNISCHE UNIVERSITÄT DARMSTADT





- globale Simulations-Zeitpunkte t_1, t_2, ...
- ein oder mehrere Ereignisse sollen jeweils parallel ausgeführt werden
- Ereignis-Scheduler wählt eines zufällig aus
- wenn bei t_1 nichts mehr zu tun, gehe zu t_2 weiter

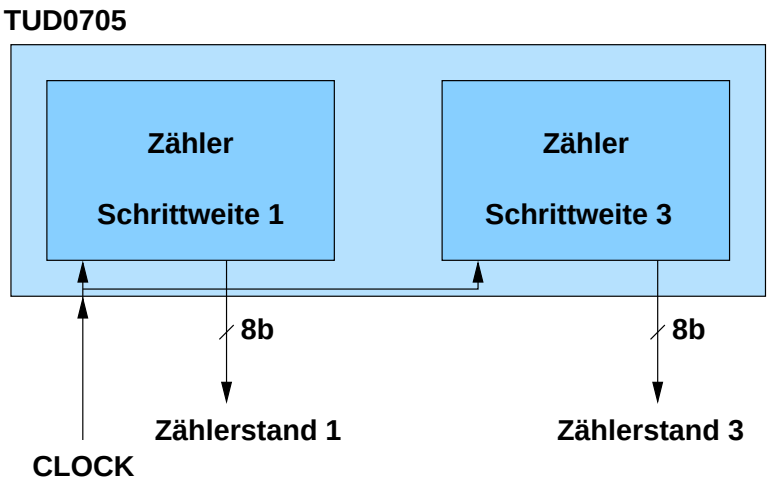
Parallelität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kein Verlass auf bestimmte Reihenfolge
  - Kann zwischen Simulatoren variieren
  - Kann auch durch Simulationsoptionen beeinflußt werden
- parallel = nicht-deterministisch
  - ein richtiges Ergebnis garantiert nicht allgemeine Korrektheit
  - exponenziell viele Ergebnisse möglich
- Unwägbarkeiten können durch Entwurfsstile reduziert werden
  - Synchrone Register-Transfer-Logik
  - Designer legt Zeitablauf explizit im Modell fest
  - Unterschiedliche Ereignisse finden in unterschiedlichen Takten statt

Busse

Beispiel-Chip TUD0705: Großer Verkaufserfolg!

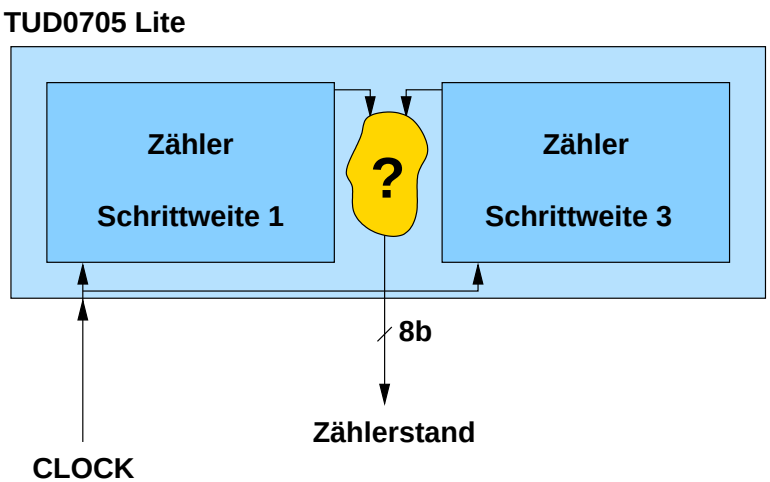


- Zwei synchrone 8b-Zähler
- Schrittweiten 1 und 3
- Beide parallel auslesbar

Problem: Zu teuer!

TECHNISCHE UNIVERSITÄT DARMSTADT

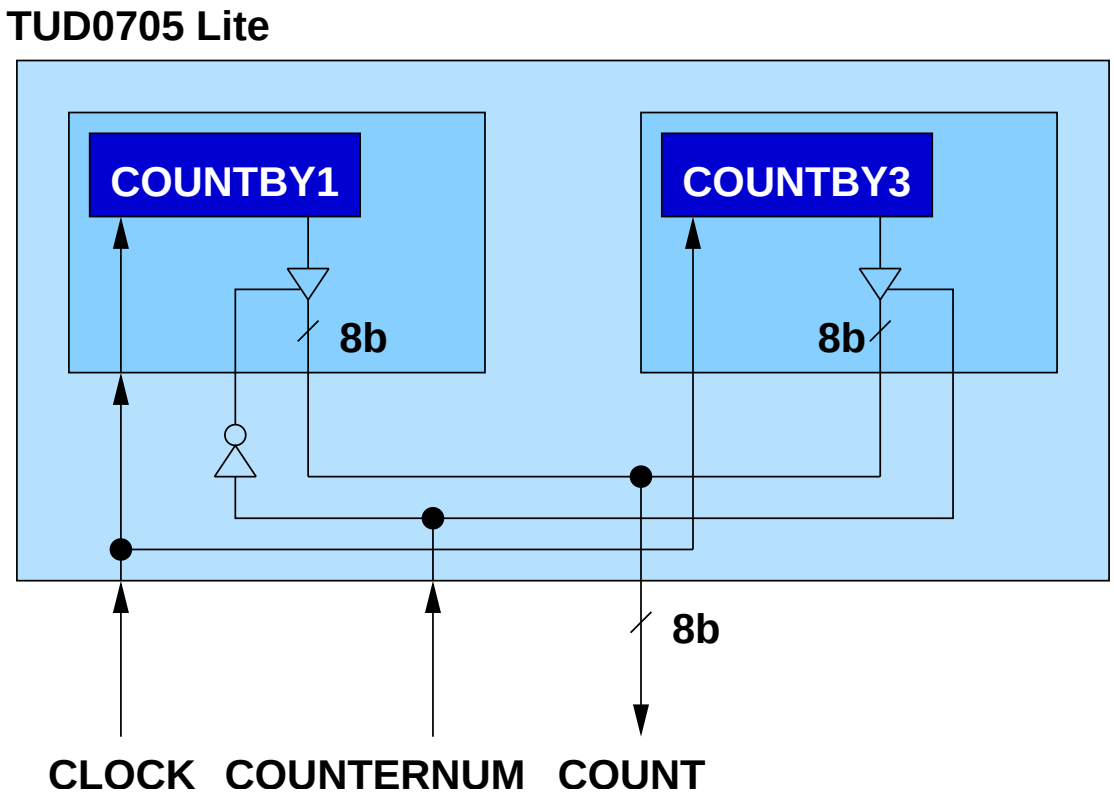
- Wo Geld sparen?
- Anforderung: Es wird nur jeweils einer der beiden Werte gebraucht
- Ausgangs-Pins sparen (kosten extra)
- Beide Zähler über die gleichen Pins nach aussen leiten



➥Wie beide Werte auseinanderhalten?

Idee: Nicht gebrauchten Zählerausgang hochohmig schalten

TECHNISCHE UNIVERSITÄT DARMSTADT



- Neuer Steuereingang `COUNTERNUM`
  - Bei `COUNTERNUM=0` Wert des ersten Zählers ausgeben
  - Bei `COUNTERNUM=1` Wert des zweiten Zählers ausgeben

Verilog: Modellierung der Zähler

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beliebige Schrittweite
- Hochohmig-schaltbarer Ausgang

```
module COUNTER
#(
  parameter stepsize = 1        // Schrittweite
 )
 (
  input   wire        CLOCK,
  input   wire        SELECT, // Wert ausgeben?
  output  wire [7:0]  OUT
 );

reg [7:0]  COUNT = 0;           // Nur für Simulation!

always @(posedge CLOCK)
  COUNT <= COUNT + stepsize;

assign OUT = (SELECT) ? COUNT : 8'bz; // Tri-State Treiber

endmodule
```

Verilog: Modellierung des Gesamt-Chips

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module TUD0705Lite
  (
    input    wire       CLOCK,
    input    wire       COUNTERNUM,
    output   wire [7:0] COUNT
  );

COUNTER #(1) COUNTBY1(CLOCK, COUNTERNUM == 0, COUNT);
COUNTER #(3) COUNTBY3(CLOCK, COUNTERNUM == 1, COUNT);

endmodule
```

Für Input-Wire `SELECT` direkt Ausdruck angegeben, statt:

```
wire SELECTBY1, SELECTBY3;

assign SELECTBY1 = COUNTERNUM == 0;
assign SELECTBY3 = COUNTERNUM == 1;

counter #(1)  COUNTBY1(CLOCK, SELECTBY1, COUNT);
counter #(3)  COUNTBY3(CLOCK, SELECTBY3, COUNT);
```

Verilog: Modellierung des Testrahmens

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module TESTFRAME;

wire [7:0] COUNT;
reg        CLOCK;
reg        COUNTERNUM;

TUD0705Lite DUT(CLOCK, COUNTERNUM, COUNT);
    always begin // Takt erzeugen
      CLOCK = 0;
      #10;
      CLOCK = 1;
      #10;
    end

initial  begin     // Stimuli
  COUNTERNUM = 0; // 1. Zähler
  #60;
  COUNTERNUM = 1; // 2. Zähler
  #60;
  COUNTERNUM = 0; // 1. Zähler
  #60;
  $finish;
end

always @(COUNT) // Ausgaben überwachen
  $display("%2.0f: COUNTERNUM=%b COUNT=%d",
           $time, COUNTERNUM, COUNT);
endmodule
```

Ergebnisse: Textausgabe

TECHNISCHE UNIVERSITÄT DARMSTADT

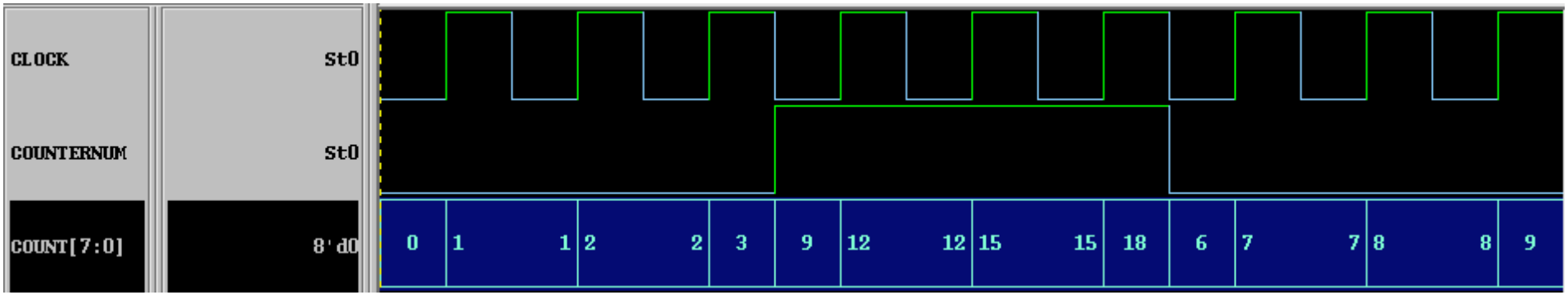
```
always begin // Takt erzeugen
  CLOCK = 0;
  #10;
  CLOCK = 1;
  #10;
end
```

```
  0: COUNTERNUM=0 COUNT=  0
 10: COUNTERNUM=0 COUNT=  1
 30: COUNTERNUM=0 COUNT=  2
 50: COUNTERNUM=0 COUNT=  3
 60: COUNTERNUM=1 COUNT=  9
 70: COUNTERNUM=1 COUNT= 12
 90: COUNTERNUM=1 COUNT= 15
110: COUNTERNUM=1 COUNT= 18
120: COUNTERNUM=0 COUNT=  6
130: COUNTERNUM=0 COUNT=  7
150: COUNTERNUM=0 COUNT=  8
170: COUNTERNUM=0 COUNT=  9
```

```
initial begin    // Stimuli
  COUNTERNUM = 0; // 1. Zähler
  #60;
  COUNTERNUM = 1; // 2. Zähler
  #60;
  COUNTERNUM = 0; // 1. Zähler
  #60;
  $finish;
end
```

Ergebnisse: Waves

TECHNISCHE UNIVERSITÄT DARMSTADT




```
  0: COUNTERNUM=0 COUNT=  0
 10: COUNTERNUM=0 COUNT=  1
 30: COUNTERNUM=0 COUNT=  2
 50: COUNTERNUM=0 COUNT=  3
 60: COUNTERNUM=1 COUNT=  9
 70: COUNTERNUM=1 COUNT= 12
 90: COUNTERNUM=1 COUNT= 15
110: COUNTERNUM=1 COUNT= 18
120: COUNTERNUM=0 COUNT=  6
130: COUNTERNUM=0 COUNT=  7
150: COUNTERNUM=0 COUNT=  8
170: COUNTERNUM=0 COUNT=  9
```

Busse

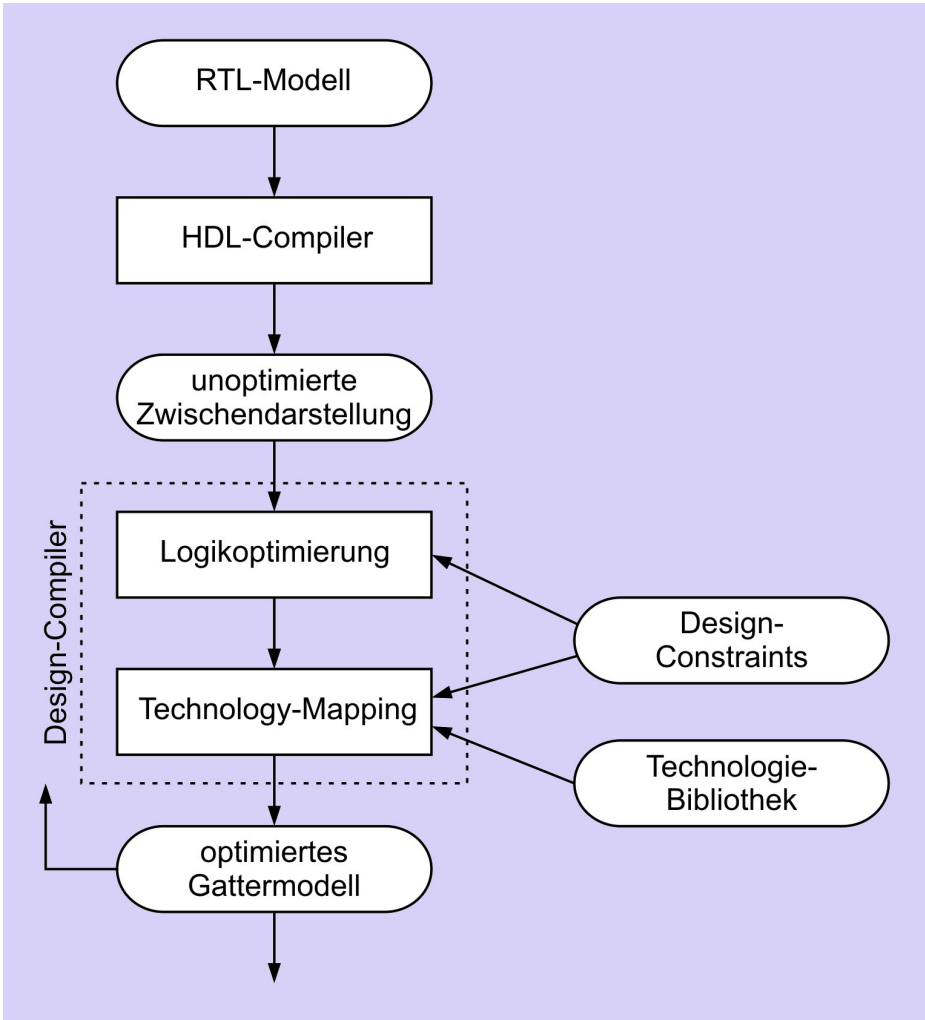
TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehrere Quellen/Senken auf einer Leitung: Bus
- Mehrere Senken: Unkritisch, fan-out immer konfliktfrei
- Quellen realisierbar durch Tri-State-Treiber
  - 0, 1, Z (hochohmig)
- Nur eine Quelle darf gleichzeitig aktiv sein
- Andere hochohmig schalten
- Benötigt Steuerung: Welche Quelle soll aktiv sein?
- Verschiedenste Möglichkeiten
- Hier gezeigt: Slave-Mode
  - Quellen wird von aussen mitgeteilt, ob Sie aktiv sein dürfen

Verfeinerter Ablauf der Synthese

Verfeinerter Entwurfsablauf der Synthese

TECHNISCHE UNIVERSITÄT DARMSTADT

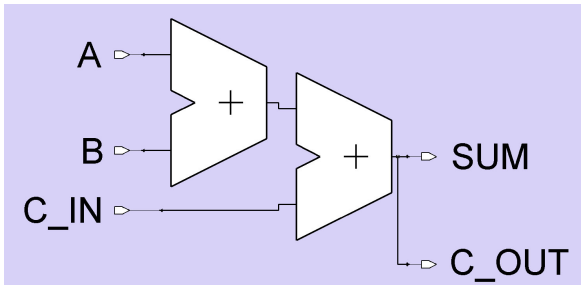


Syntheseablauf: Technologieabbildung

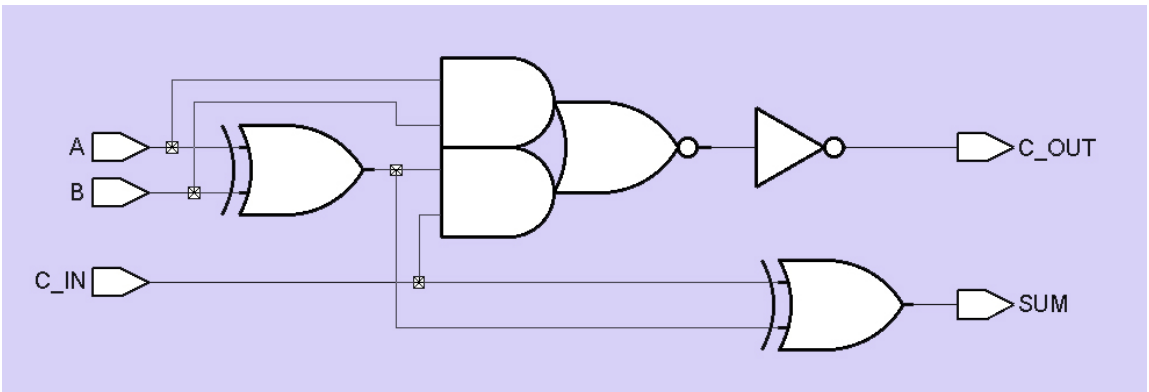
technology mapping

TECHNISCHE UNIVERSITÄT DARMSTADT

Unoptimierte Zwischendarstellung eines 1b-Addierers



Ergebnis für ASIC-Zielbibliothek (LSI 10K)

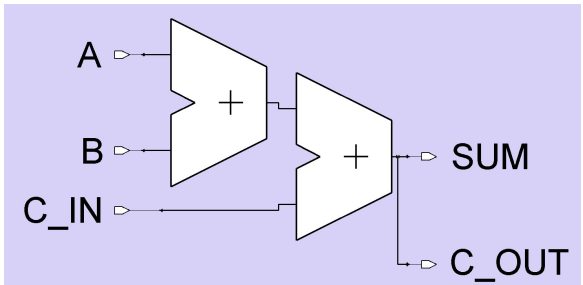


Syntheseablauf: Technologieabbildung

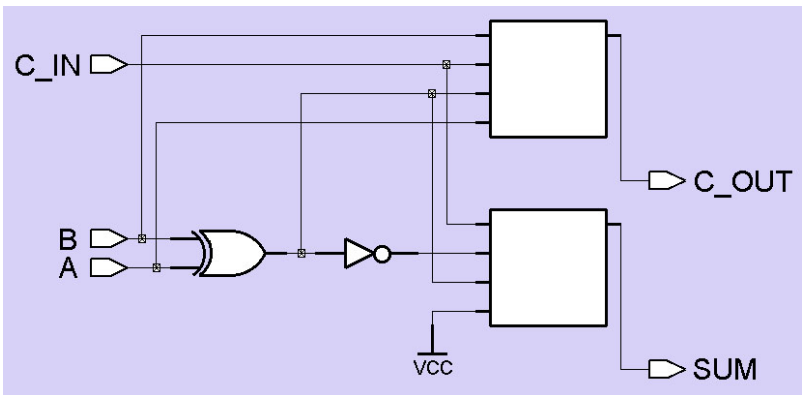
technology mapping

TECHNISCHE UNIVERSITÄT DARMSTADT

Unoptimierte Zwischendarstellung eines 1b-Addierers



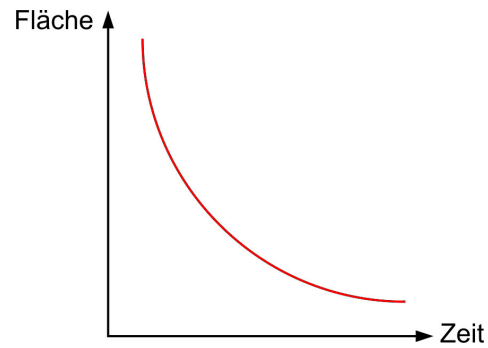
Ergebnis für FPGA (Xilinx XC4000)



Design-Constraints

TECHNISCHE UNIVERSITÄT DARMSTADT

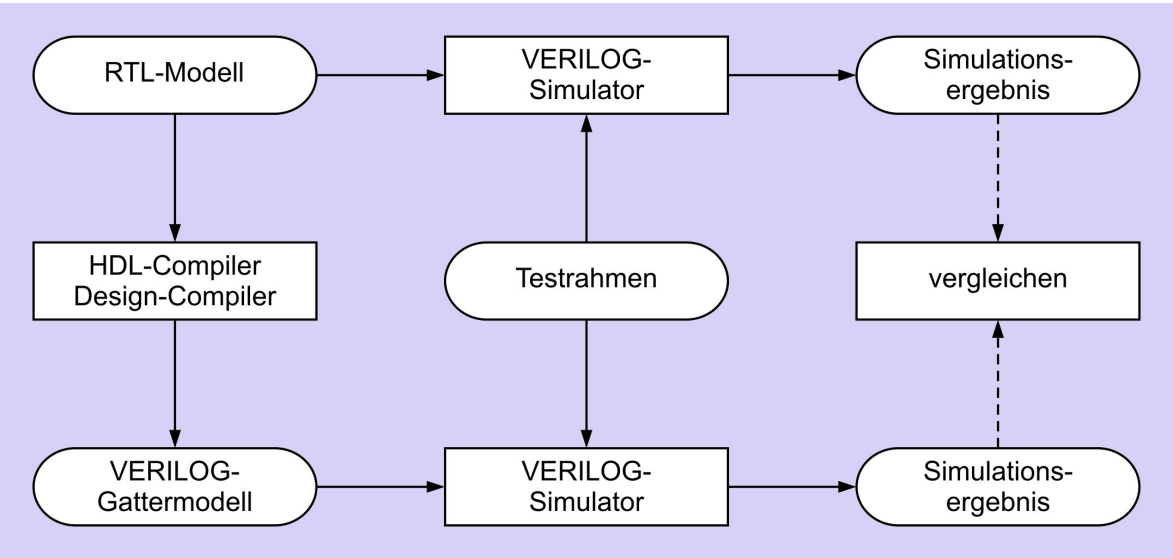
- Zeit
  - Timing-Analyse
  - Geschätzt nach Synthese (ohne Verdrahtungsverzögerung)
  - Exakt nach Platzieren und Verdrahten
  - ➥Layout-Ebene
- Fläche
  - Geschätzt nach Synthese (ohne Verdrahtungsfläche!)
  - Exakt nach Platzieren und Verdrahten
- Elektrische Leistungsaufnahme
  - Simulation auf Layout-Ebene
  - Bestimmung von Umschaltfrequenzen (*toggle frequencies*) von Signalen



Verifikation

Verhält sich synthetisierte Schaltung noch so wie Simulations-modell?

TECHNISCHE UNIVERSITÄT DARMSTADT



- Prä-Synthese ./. Post-Synthese-Simulation
- Gleiche Testdaten
  - Bei Post-Synthese aber genauere Tests möglich
  - Hier nun genaueres Zeitverhalten
- Differenzen in Ergebnissen durch anderes Zeitverhalten
- Vergleich etwas aufwendiger

Beispiel: 4b-Vergleicher

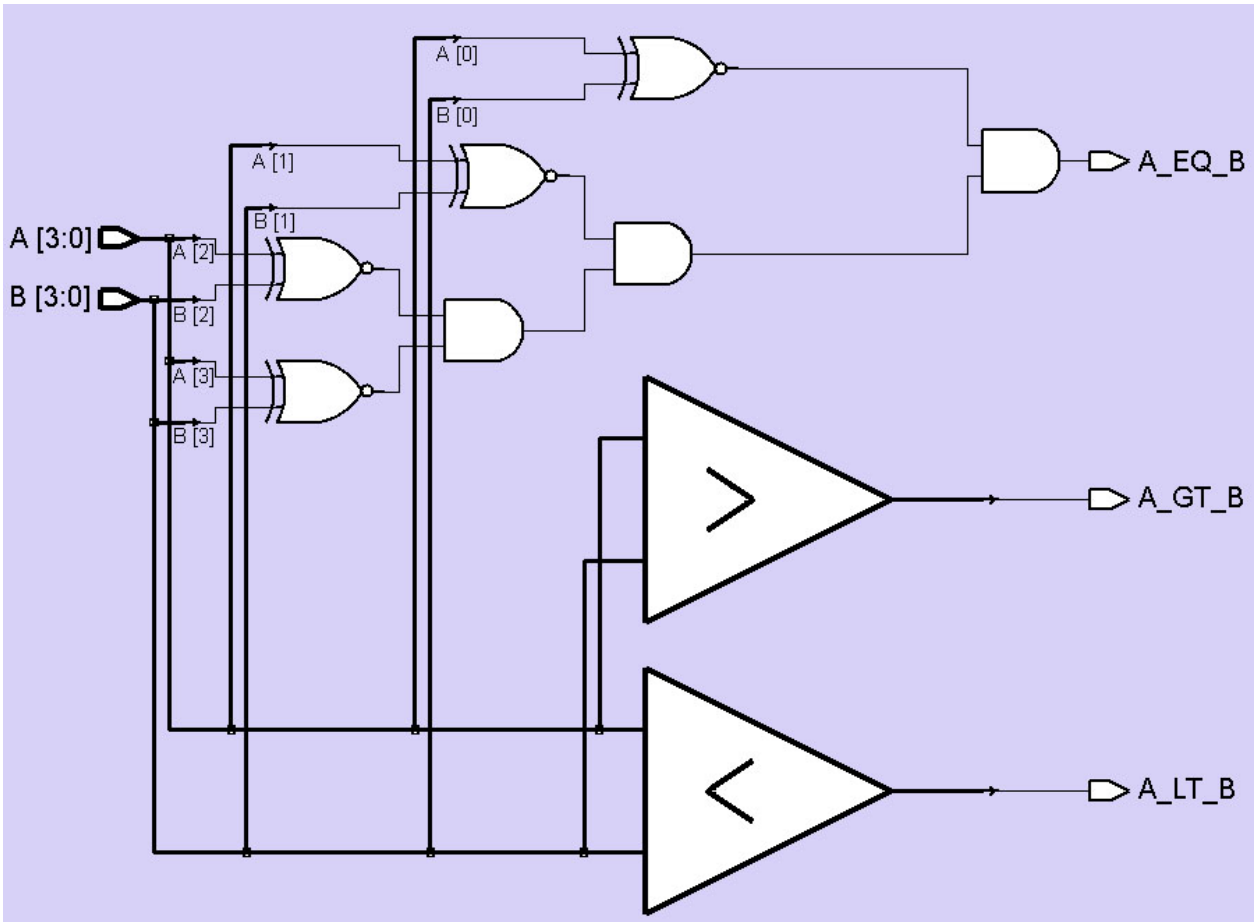
TECHNISCHE UNIVERSITÄT DARMSTADT

- Größenvergleich von zwei 4b breiten Eingabewerten `A` und `B`
- Bestimmt Flags für $<$, $>$, und =
- Erstes Ziel: Möglichst schnelle Schaltung, Fläche egal

```
// Vergleicher
module mag_comp (
  input  wire [3:0]  A, B,
  output wire        A_GT_B, A_LT_B, A_EQ_B);

assign A_GT_B = (A > B);       // A groesser B
assign A_LT_B = (A < B);       // A kleiner B
assign A_EQ_B = (A == B);      // A gleich B

endmodule
```

Zwischendarstellung des 4b-Vergleichers

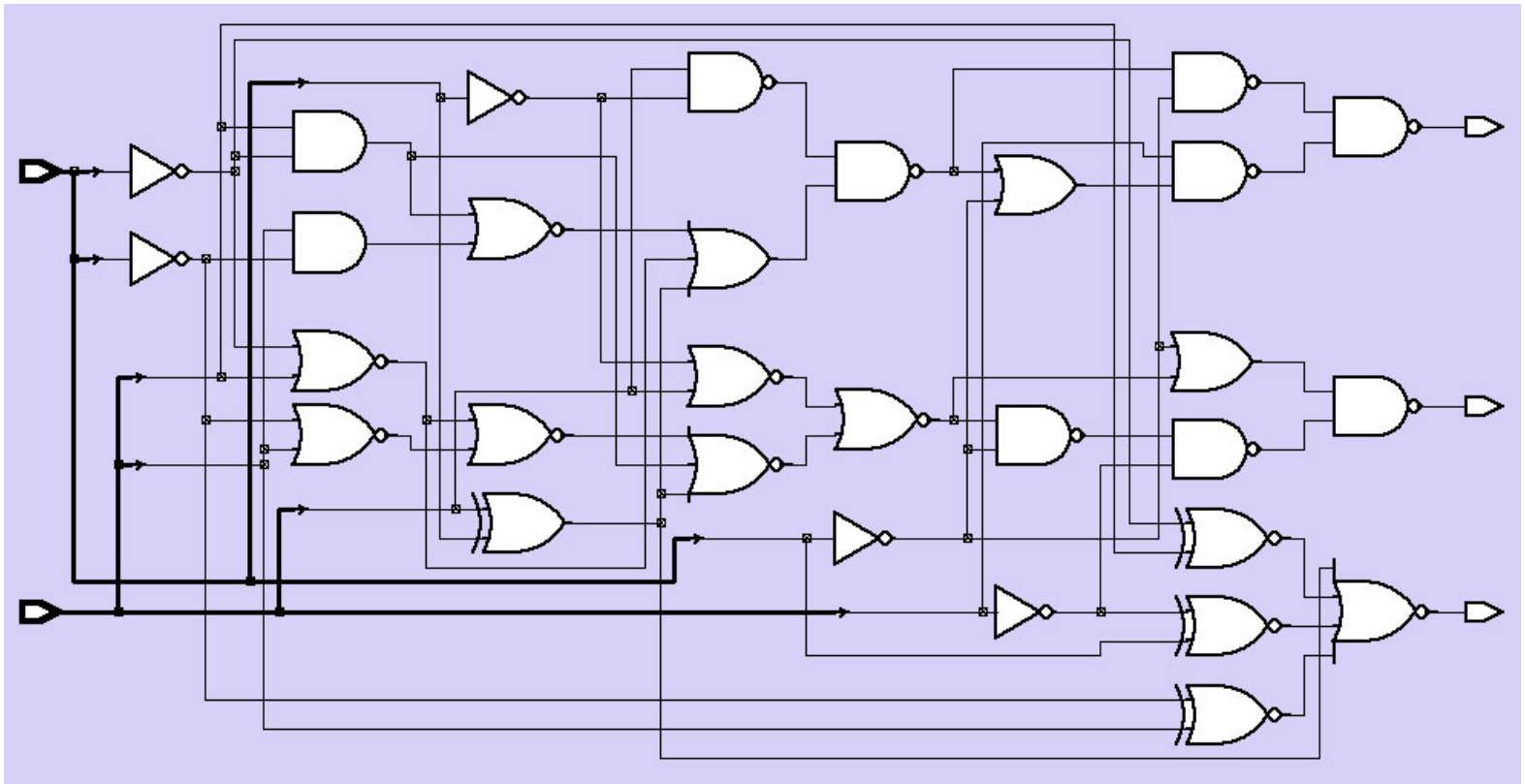
TECHNISCHE UNIVERSITÄT DARMSTADT



4b-Vergleicher in LSI Logic 10K ASIC-Technologie

Benutzt um 1993

TECHNISCHE UNIVERSITÄT DARMSTADT

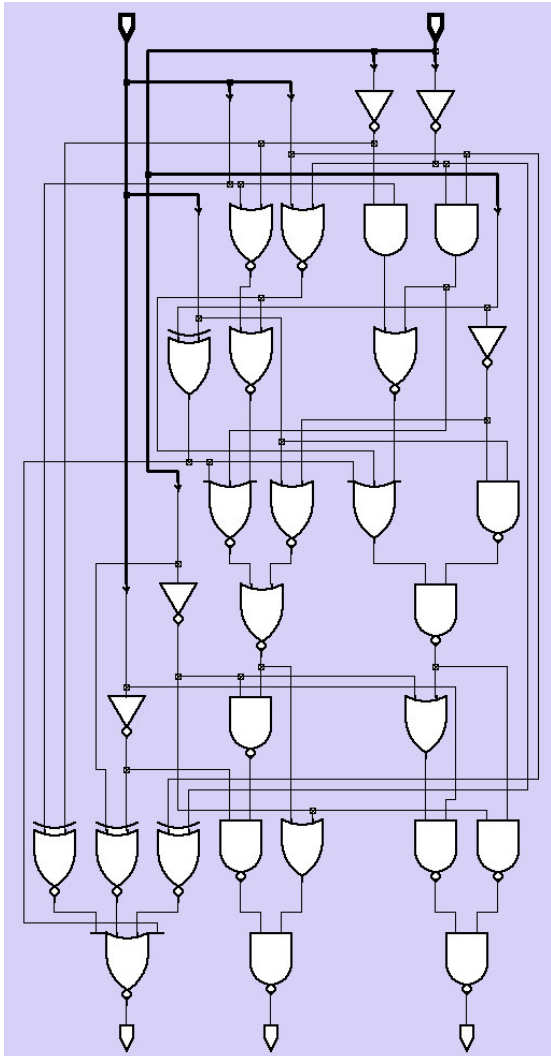


Gatternetzliste des 4b-Vergleichers

Als strukturelles Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mag_comp_gate (
  input  wire [3:0]  A,
                     B,
  output wire A_GT_B, A_LT_B, A_EQ_B);

wire
  N107, N108, N109, N110, N111, N112, N113, N114, N115,
  N116, N117, N118, N119, N120, N121, N122, N123, N124,
  N125, N126, N127, N128, N129, N130, N131, N132, N133;

  nr4   U89   (.A(N107), .B(N108), .C(N109), .D(N110), .Z(A_EQ_B));
  nd2   U90   (.A(N111), .B(N112), .Z(A_GT_B));
  nd2   U91   (.A(N113), .B(N114), .Z(A_LT_B));
  iv    U92   (.A(B[0]),  .Z(N115));
  iv    U93   (.A(B[1]),  .Z(N116));
  iv    U94   (.A(B[2]),  .Z(N117));
  nr2   U95   (.A(N119), .B(N120), .Z(N118));
  iv    U96   (.A(B[3]),  .Z(N121));
  nd2   U97   (.A(N123), .B(N124), .Z(N122));
  iv    U98   (.A(A[3]),  .Z(N125));
  en    U99   (.A(N116), .B(A[1]),  .Z(N108));
  en    U100  (.A(N125), .B(B[3]),  .Z(N109));
  en    U101  (.A(N115), .B(A[0]),  .Z(N110));
  an2   U102  (.A(A[1]),  .B(N116), .Z(N126));
  nr2   U103  (.A(N116), .B(A[1]),  .Z(N127));
  nr2   U104  (.A(N115), .B(A[0]),  .Z(N128));
  nr2   U105  (.A(N127), .B(N128), .Z(N129));
  nr3   U106  (.A(N129), .B(N126), .C(N107), .Z(N120));
  nr2   U107  (.A(N117), .B(A[2]),  .Z(N119));
  nd2   U108  (.A(N118), .B(N121), .Z(N130));
  nd2   U109  (.A(N130), .B(N125), .Z(N114));
  or2   U110  (.A(N121), .B(N118), .Z(N113));
  an2   U111  (.A(A[0]),  .B(N115), .Z(N131));
  ...
endmodule
```



Schlüssellochblick in Technologie-Bibliothek

an2: AND-Gatter mit zwei Eingängen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// 2-Input-AND-Gatter

// physikalische Zeiteinheit / Simulationsschrittweite
‘timescale 100 ps / 10 ps
‘celldefine
module an2 (Z, A, B);
  output Z;
  input  A, B;

  // Instanz einer VERILOG-Primitive (in KCMS nicht weiter behandelt!)
  and And1 (Z, A, B);

  // Verzögerungszeiten fuer steigende und fallende Flanken
  specify
    (A *> Z) = (1, 1);
    (B *> Z) = (1, 1);
  endspecify

endmodule
‘endcelldefine
```

Test des 4b-Vergleichers: Prä-Synthese

TECHNISCHE UNIVERSITÄT DARMSTADT

Testrahmen

```
module stimulus;

reg [3:0] A, B;
wire A_GT_B, A_LT_B, A_EQ_B;

// Instanz des Vergleichers
mag_comp Mag_comp (A, B, A_GT_B, A_LT_B, A_EQ_B);

 initial
   $monitor ($time,
     "␣A=%d,␣B=%d,␣A_GT_B=%b,␣A_LT_B=%b,␣A_EQ_B=%b",
       A,     B,     A_GT_B,  A_LT_B,   A_EQ_B);

// Testmuster
 initial  begin
       A = 10; B =  9;
  #10 A = 14; B = 15;
  #10 A =  0; B =  0;
  #10 A =  8; B = 12;
  #10 A =  6; B = 14;
  #10 A = 14; B = 14;
end

endmodule
```

```
 0 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Prä-Synthese-Ergebnis

Vergleich: Prä-Synthese mit Post-Synthese

Gleiche Stimuli wie eben

TECHNISCHE UNIVERSITÄT DARMSTADT

Prä-Synthese

```
 0 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Post-Synthese

```
 0 A=10, B= 9, A_GT_B=x, A_LT_B=x, A_EQ_B=x
 3 A=10, B= 9, A_GT_B=x, A_LT_B=x, A_EQ_B=0
 6 A=10, B= 9, A_GT_B=x, A_LT_B=0, A_EQ_B=0
 8 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=1, A_LT_B=0, A_EQ_B=0
16 A=14, B=15, A_GT_B=1, A_LT_B=1, A_EQ_B=0
18 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=1, A_EQ_B=0
23 A= 0, B= 0, A_GT_B=0, A_LT_B=1, A_EQ_B=1
28 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=0, A_EQ_B=1
32 A= 8, B=12, A_GT_B=1, A_LT_B=0, A_EQ_B=0
34 A= 8, B=12, A_GT_B=0, A_LT_B=0, A_EQ_B=0
35 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
53 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Annahme: Jedes Gatter hat 1 Zeiteinheit Verzögerung

Diskussion: Prä- ./. Post-Synthese-Simulation

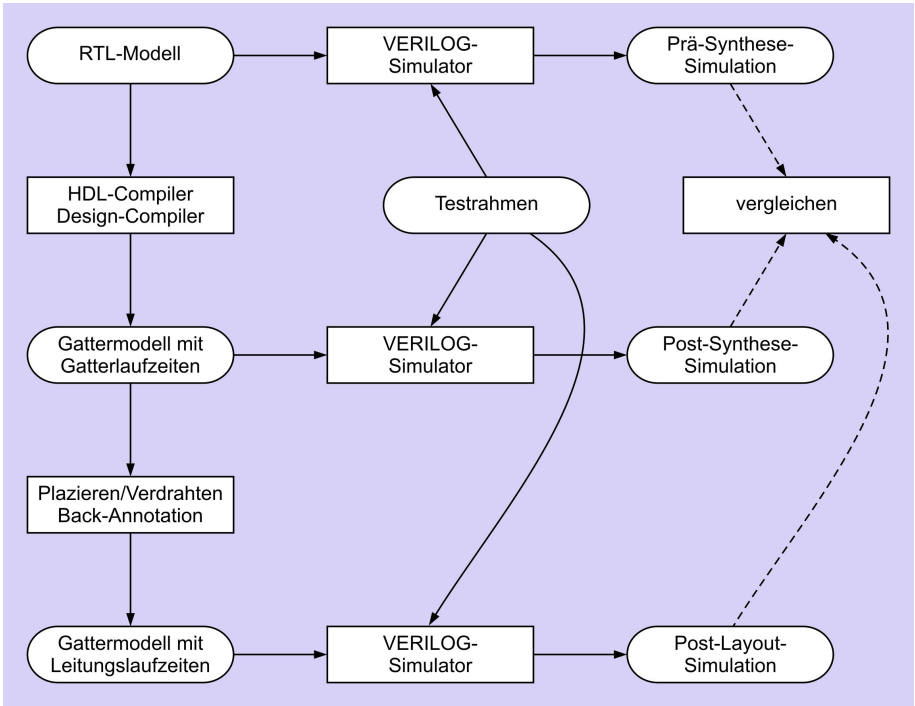
TECHNISCHE UNIVERSITÄT DARMSTADT

- Unterschiedlich viele Ergebnisse
- Verschiedene Werte
- Unterschiedliche Zeiten
  - Vorher gar keine ausser den im Testrahmen
- Manche Ergebnisse schlicht falsch (z.B. bei t=32)
- Interpretation nötig
  “Wenn man lange genug wartet, ist das Ergebnis richtig”!
- Was ist “lange genug”?
- Antwort: Kritischer Pfad (TGDI)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Weitere Verfeinerung der Verifikation

Nun bis auf Layout-Ebene

TECHNISCHE UNIVERSITÄT DARMSTADT



- Post-Layout-Simulation schliesst ein
  - Gatterverzögerungen
  - Leitungsverzögerung
  - Kann umfassen: Widerstände, Kapazitäten, Induktivitäten

Synthesebeispiel: Zero-Counter

Synthesebeispiel: Zero-Counter

TECHNISCHE UNIVERSITÄT DARMSTADT

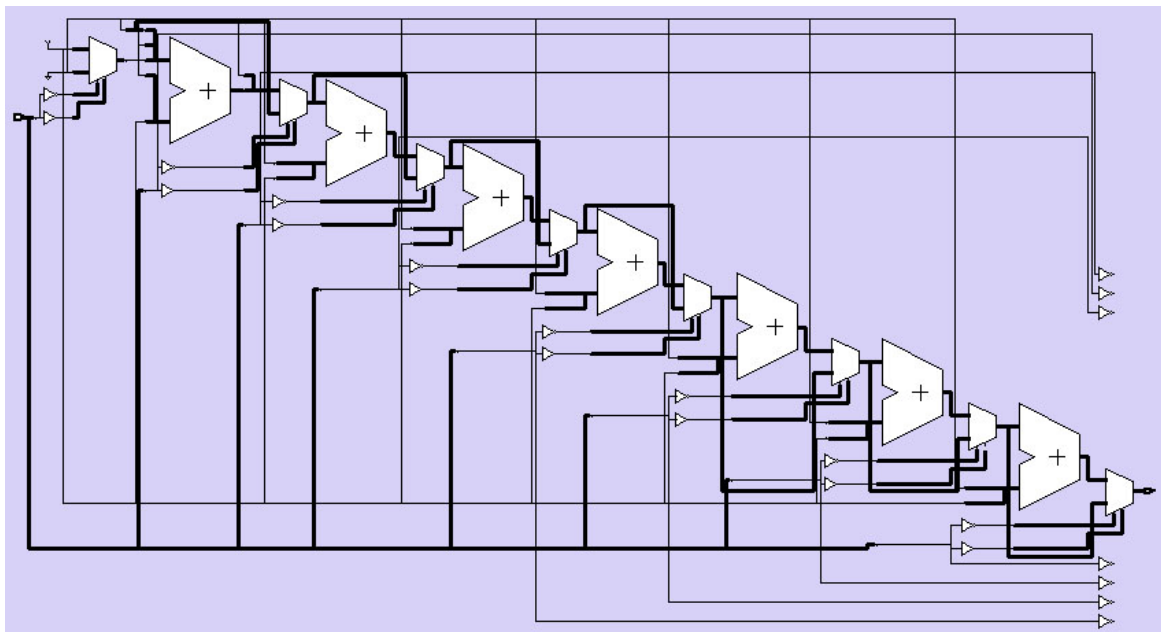
- Spezifikation
  - Eingabe ist ein 8b Datenwort
  - Ausgabe soll sein die Anzahl der Null-Bits in der Eingabe
- Genauer betrachtet
  - RTL-Modell kann Synthese-Ergebnis direkt beeinflussen
  - Wirkung von Design-Constraints
- Nicht mehr so relevant
  - Konkrete Umsetzung in Gatter-Modell
  - Bei größeren Schaltungen oft schlicht zu unübersichtlich

Zero-Counter: Intuitive Lösung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module count(
  input  wire [7:0]  IN,
  output reg  [3:0]  OUT);

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird

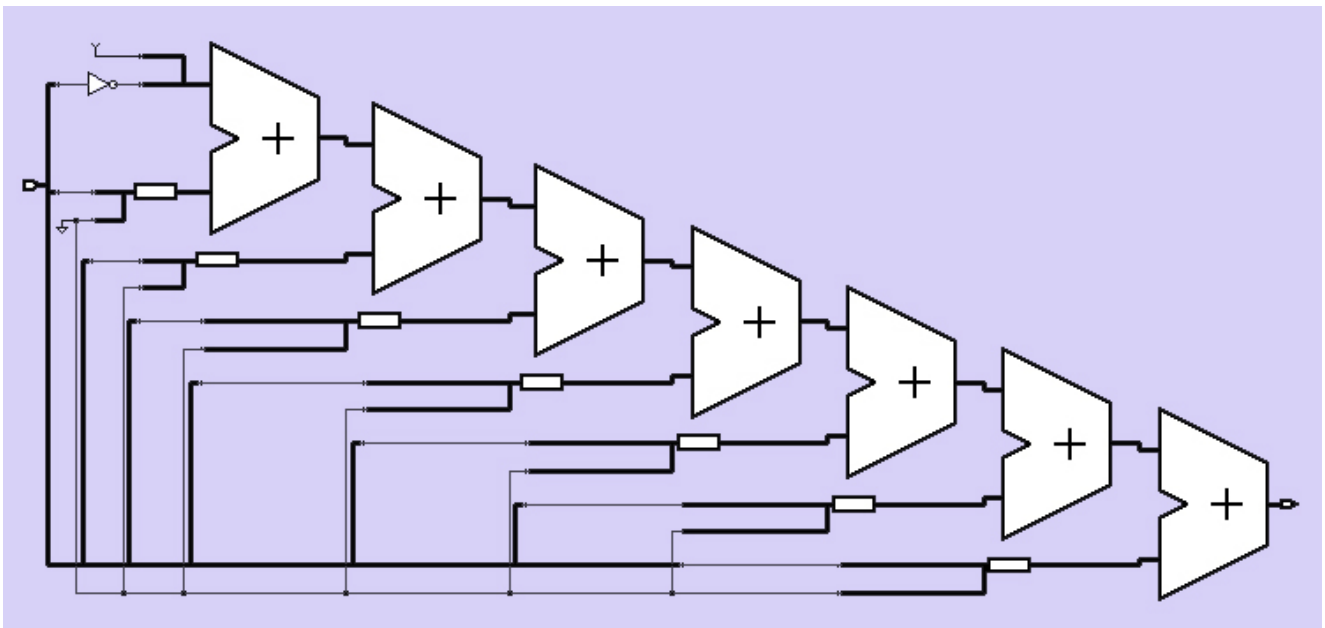


Zero-Counter: Vereinfachte Lösung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module count(
  input  wire [7:0]  IN,
  output reg  [3:0]  OUT);

integer       I;

always @(IN) begin
  OUT = ~IN[0];
  for  (I=1; I<8; I=I+1)
    OUT = OUT + ~IN[I];
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer entfallen, Bits werden direkt aufaddiert

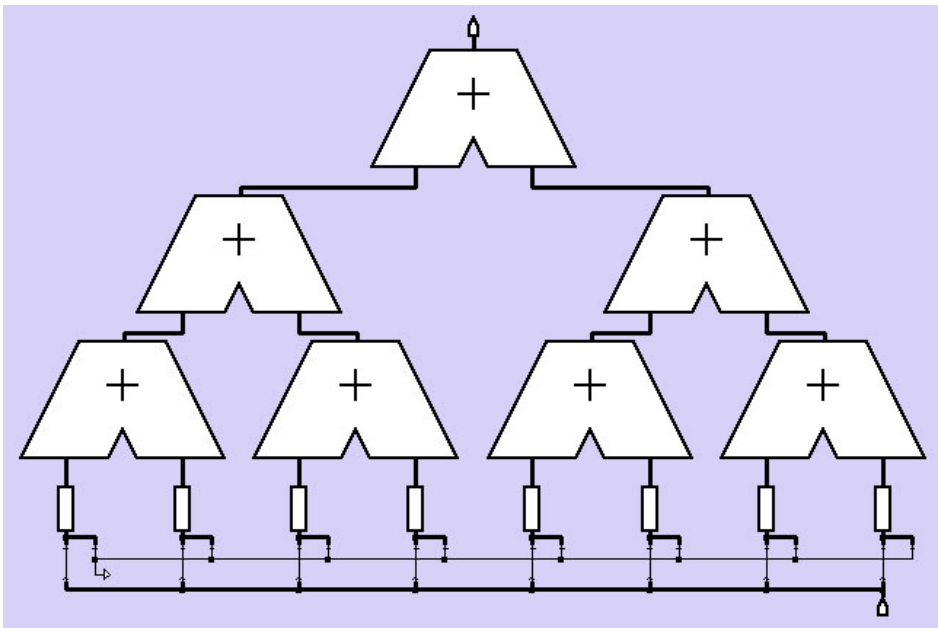


Zero-Counter: Schlaue Lösung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module count(
  input  wire [7:0] IN,
  output reg  [3:0] OUT);

always @(IN)
  OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))
      + ((~IN[4]+~IN[5])  + (~IN[6]+~IN[7]));

endmodule
```

- Bits werden direkt aufaddiert
- Jetzt aber hierarchische Klammerung
- Damit parallele Berechnung
- Addierer-Baum



Einfluß von Design-Constraints

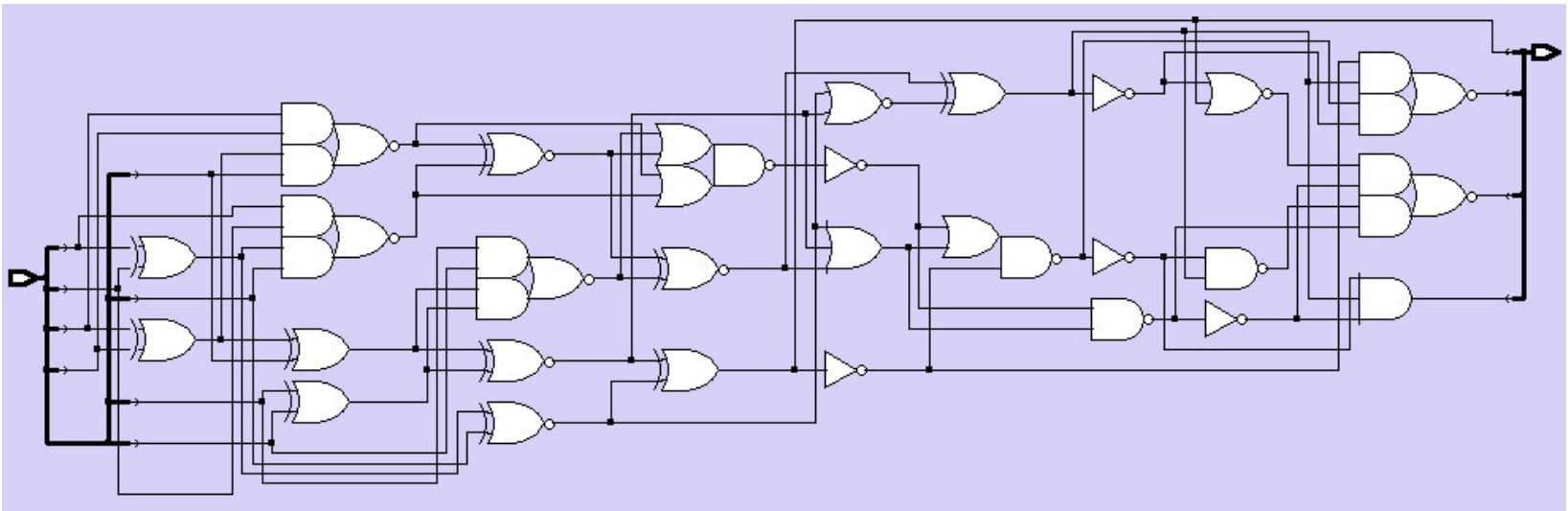
TECHNISCHE UNIVERSITÄT DARMSTADT

- Festlegen unterschiedlicher Optimierungsziele
- Üblich
  - Flächenbedarf
  - Geschwindigkeit (niedrige Verzögerung)
- Noch seltener
  - Energieverbrauch
  - Ausfallsicherheit

Optimierung auf Fläche

8b-Zero-Counter

TECHNISCHE UNIVERSITÄT DARMSTADT

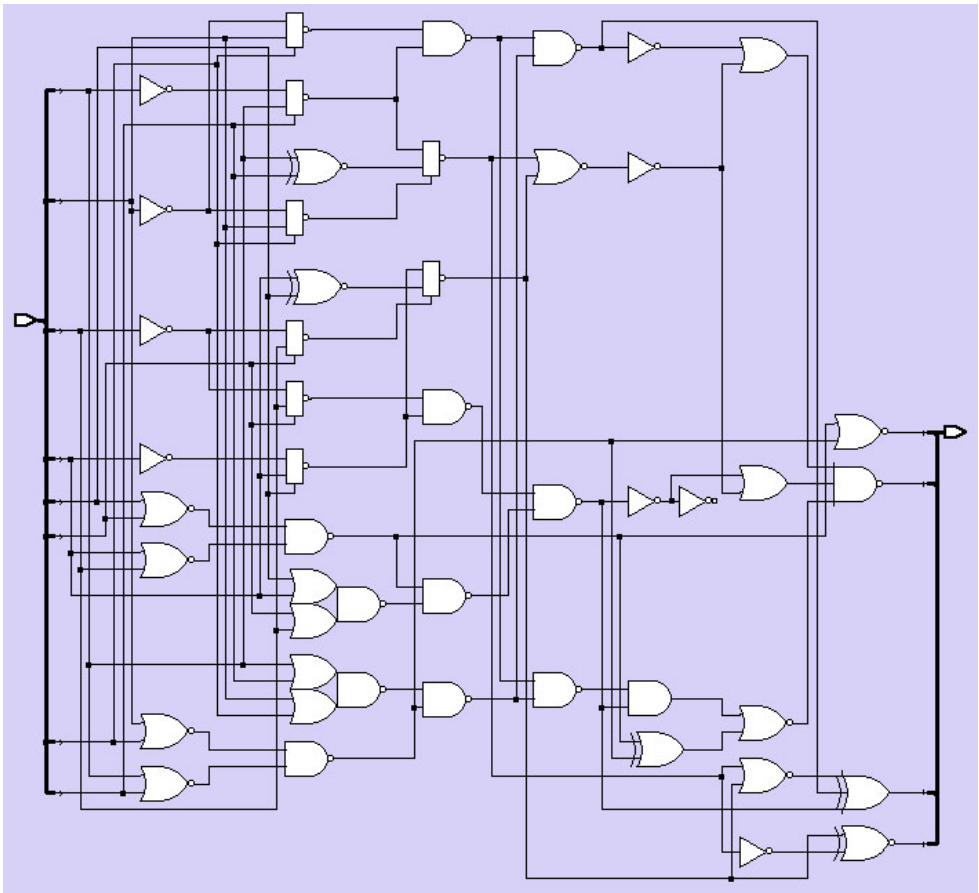


52 Gatter, 17.8ns Verzögerung

Optimierung auf Geschwindigkeit

8b-Zero-Counter

TECHNISCHE UNIVERSITÄT DARMSTADT



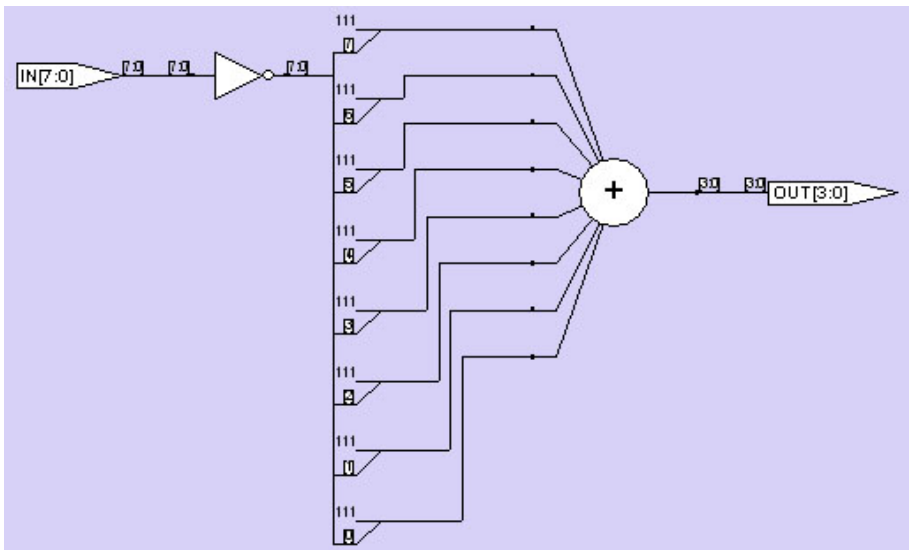
99 Gatter, 8.2ns Verzögerung

Zero-Counter: Noch bessere Lösung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module count(
  input  wire [7:0] IN,
  output reg  [3:0] OUT);

always @(IN)
  OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))
      + ((~IN[4]+~IN[5])  + (~IN[6]+~IN[7]));

endmodule
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer



Synthesebeispiel: Schaltwerk

Entwurfsbeispiel: Verkaufsautomat

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bisher kombinatorische Beispiele
  - 4b-Vergleicher
  - 8b-Zero-Counter
- Nun sequentielle Logik
  - Mit Zustandsgedächtnis und Takt
  - Schaltwerk

Spezifikation des Verkaufsautomaten

TECHNISCHE UNIVERSITÄT DARMSTADT

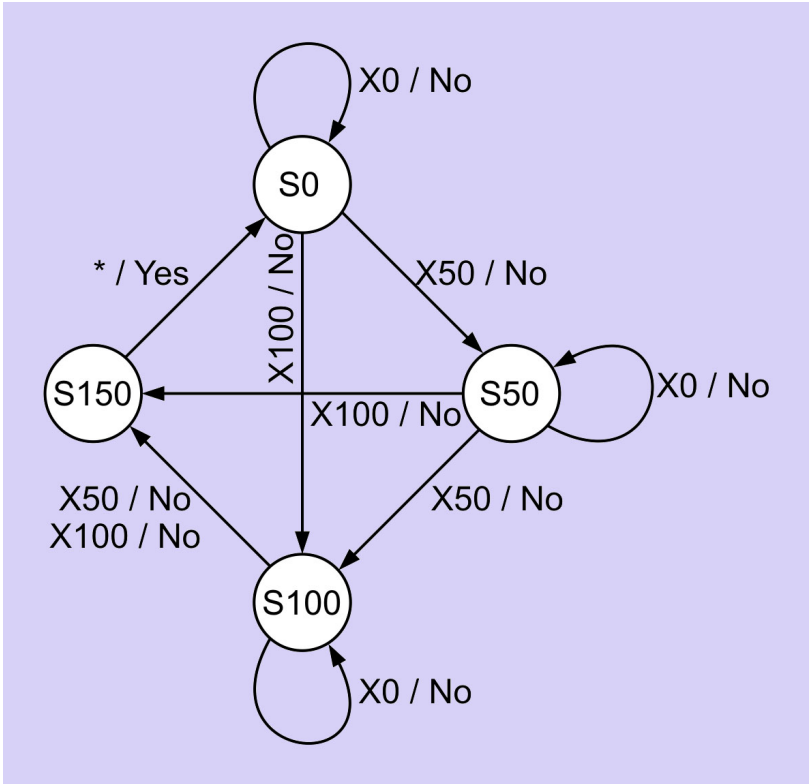
- Hier: Entwurf der Steuerung
- Eingabe: Münzen im Wert von 1 Euro und 50 Cent
- Ausgabe: Ein Eis für 1,50 EUR
- Eigentlicher Verdienst
  - Überzahlung möglich, aber kein Wechselgeld
- Reihenfolge von 50 Cent und 1 Euro beliebig
- Vereinfachung hier: Kurze Pause zwischen zwei Münzeinwürfen

Grobentwurf der Steuerung des Verkaufsautomaten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Periodisches Eingabesignal `COIN` aus zwei Bits
  - Kann symbolische Werte `X0` (=keine neue Münze), `X50`, `X100` annehmen
- Ausgabesignal `ICE` steuert Eisausgabe
  - Symbolische Werte `Yes` und `No`
- Zustände `S0`, `S50`, `S100`, `S150`

➥Zustandsautomat
in Mealy-Form



RTL-Modell: Modulkopf

Verwaltungskram, hier passiert noch nichts

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Verkaufsautomat
module iglu (
  input  wire       CLOCK,   // Takt
                    RESET,   // Reset
  input  wire [1:0] COIN,    // eingeworfene Muenze
  output reg        ICE);    // Warenausgabe

// interne Variable
reg  [1:0] PRESENT,          // jetziger Zustand
           NEXT;             // naechster Zustand
// Zustandscodierung
parameter S0   = 2'b00,      // bisher nichts eingeworfen
          S50  = 2'b01,      // bisher 50 Cent
          S100 = 2'b10,      // bisher 100 Cent
          S150 = 2'b11;      // bisher 150 Cent oder 200
// Eingabecodierung
parameter X0   = 2'b00,      // kein Einwurf
          X50  = 2'b01,      // Fuenfziger
          X100 = 2'b10;      // Euro
// Ausgabecodierung
parameter Yes = 1'b1,        // Warenausgabe
          No  = 1'b0;        // keine Ausgabe
```

RTL-Modell: Kombinatorische Logik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
always @(PRESENT, COIN)        // <-- flankenfrei!
  case (PRESENT)

  S0:                          // bisher nichts eingeworfen
    if  (COIN == X0)           // kein Einwurf
      {NEXT,ICE} = {S0, No};
    else if  (COIN == X50)     // Fuenfziger
      {NEXT,ICE} = {S50, No};
    else                       // Euro
      {NEXT,ICE} = {S100,No};

  S50:                         // bisher 50 Cent
    if  (COIN == X0)           // kein Einwurf
      {NEXT,ICE} = {S50, No};
    else if  (COIN == X50)     // Fuenfziger
      {NEXT,ICE} = {S100,No};
    else                       // Euro
      {NEXT,ICE} = {S150,No};

  S100:                        // bisher 100 Cent
    if  (COIN == X0)           // kein Einwurf
      {NEXT,ICE} = {S100,No};
    else                       // Fuenfziger oder Euro
      {NEXT,ICE} = {S150,No};  // Ueberzahlung wird dankend
                               //  ignoriert

  S150:                        // genug bezahlt
      {NEXT,ICE} = {S0, Yes} ; // Warenausgabe

  endcase
```

Berechne aus dem aktuellen Zustand `PRESENT` und eventuell eingeworfenen Münzen `COIN` vorläufig den nächsten Zustand `NEXT` und die Ausgabe `ICE`.

Beachte: Blockende Zuweisungen

RTL-Modell: Zustandsspeicher

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// naechsten Zustand mit jedem Takt endgueltig zum jetzigen machen;
// synchrones Reset
//
always @(posedge CLOCK)                    // <-- flankengesteuert!
  if  (RESET == 1'b1)   PRESENT <= S0;     // Anfangszustand
  else                  PRESENT <= NEXT;

endmodule // iglu
```

Testrahmen für Verkaufsautomat

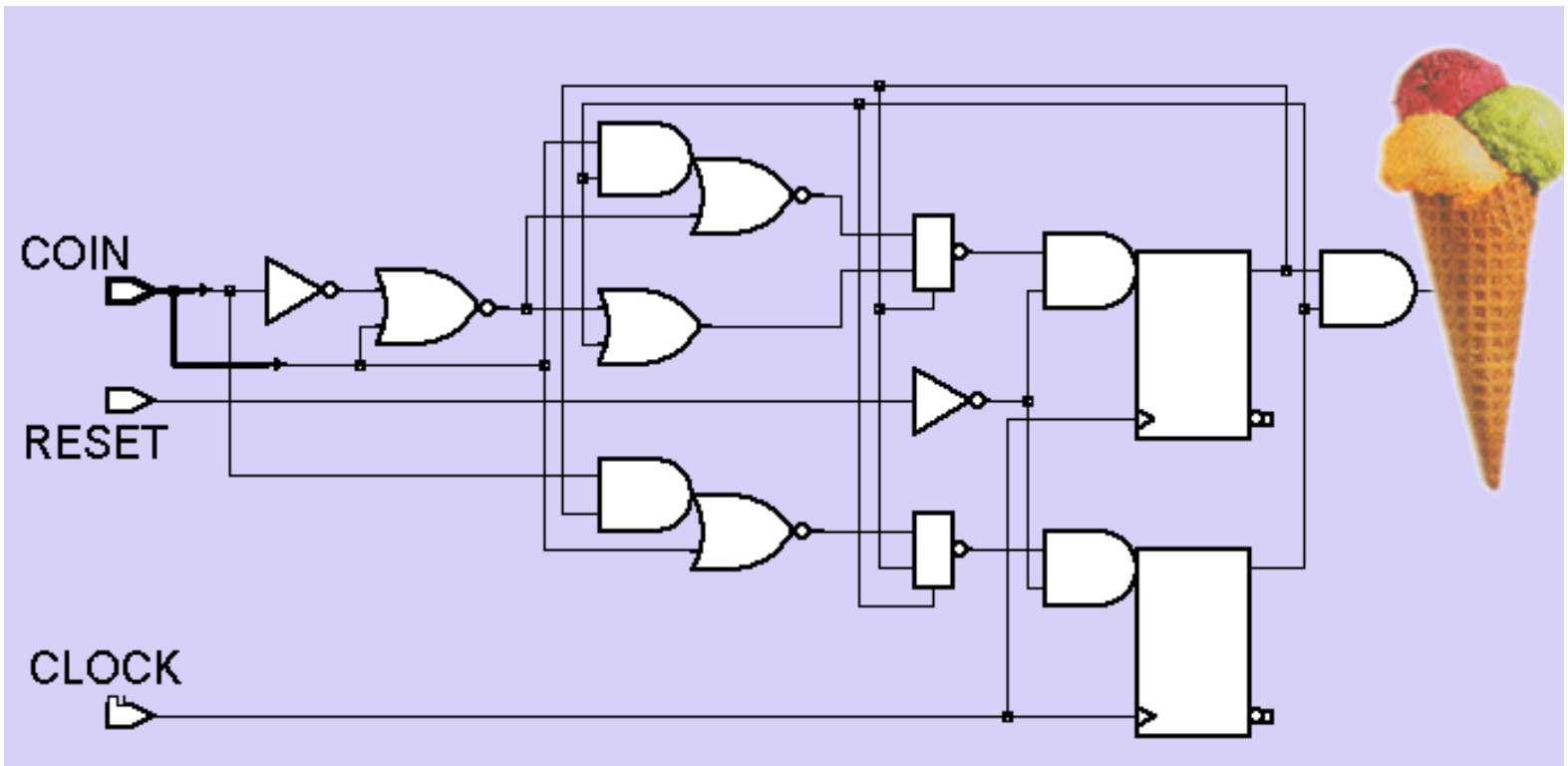
TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Instanz des Verkaufsautomaten
iglu  Iglu  (CLOCK, RESET, COIN, ICE);
// Takt
always
  #20 CLOCK = ~CLOCK;
// Ergebnis drucken
 initial  begin
  $display ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Zeit␣Reset␣Eisausgabe\n");
  $monitor ("%d␣␣␣%d␣␣␣␣␣%d", $time, RESET, ICE);
end
// Stimuli anlegen: Muenzen eingeben
 initial  begin
  CLOCK = 0; COIN = X0; RESET = 1'b1;
  #50 RESET = 1'b0;
  @(negedge CLOCK);

  // drei Fuenfziger
  #80  $display ("Einwurf␣␣50␣Ct"); COIN = X50; #40 COIN = X0;
  #80  $display ("Einwurf␣␣50␣Ct"); COIN = X50; #40 COIN = X0;
  #80  $display ("Einwurf␣␣50␣Ct"); COIN = X50; #40 COIN = X0;
  // einen Fuenfziger, dann einen Euro
  #160 $display ("Einwurf␣␣50␣Ct"); COIN = X50; #40 COIN = X0;
  #80  $display ("Einwurf␣100␣Ct"); COIN = X100; #40 COIN = X0;
  // zwei Euro (keine Rueckgabe!)
  #160 $display ("Einwurf␣100␣Ct"); COIN = X100; #40 COIN = X0;
  #80  $display ("Einwurf␣100␣Ct"); COIN = X100; #40 COIN = X0;
  // einen Euro, dann einen Fuenfziger
  #160 $display ("Einwurf␣100␣Ct"); COIN = X100; #40 COIN = X0;
  #80  $display ("Einwurf␣␣50␣Ct"); COIN = X50; #40 COIN = X0;

  #80 $finish;
 ...
```

```
                    Zeit Reset Eisausgabe

                       0   1      x
                      20   1      0
                      50   0      0
Einwurf  50 Ct
Einwurf  50 Ct
Einwurf  50 Ct
                     420   0      1
                     460   0      0
Einwurf  50 Ct
Einwurf 100 Ct
                     740   0      1
                     780   0      0
Einwurf 100 Ct
Einwurf 100 Ct
                    1060   0      1
                    1100   0      0
Einwurf 100 Ct
Einwurf  50 Ct
                    1300   0      1
```

Syntheseergebnis

TECHNISCHE UNIVERSITÄT DARMSTADT



Modellierungsstil

TECHNISCHE UNIVERSITÄT DARMSTADT

- RTL-Modellierungsstil beeinflußt Gattermodell
  - effizient oder schlecht, im schlimmsten Fall falsch
  - Designer muss eingesetzten Stil genau beobachten
- Zielkonflikt
  - Abstraktes RTL-Modell
    - angenehm, Synthesewerkzeug soll sich um Details kümmern
    - Produziert aber gelegentlich unerfreuliche Hardware
  - Hardware-nahes RTL-Modell
    - Mühsam, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigkeit von Zieltechnologie kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Spezialblöcken