

Einführung in Computer Microsystems

Sommersemester 2011

2. Block: Verilog Überblick



TECHNISCHE
UNIVERSITÄT
DARMSTADT





Advanced Digital Design with the Verilog HDL

von Michael D. Ciletti

Pearson Education, Inc. , 2003

- ▶ Guter Überblick über Verilog
- ▶ Viele Beispiele
- ▶ In Bibliothek vorhanden



Grundlagen

Grundlegender Baustein: Modul

Noch ganz ohne Hardware-Bezug



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

  initial // führe folgenden Block am Anfang aus
begin
  $display ("Beginn_der_Simulation...");
  for (l=1; l <= 3; l=l+1)
    $display ("Durchlauf_%"d", l);
  $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



```
// 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 ...

Was wird ausgegeben?

```
module two_blocks;
```

```
  initial
```

```
    begin
```

```
      $display ("Ja");
```

```
      $display ("Ja");
```

```
    end
```

```
  initial
```

```
    begin
```

```
      $display ("Nein");
```

```
      $display ("Nein");
```

```
    end
```

```
endmodule
```

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

```
module two_blocks;
```

```
  initial
```

```
  begin
```

```
    $display ("Ja");
```

```
    $display ("Ja");
```

```
  end
```

```
  initial
```

```
  begin
```

```
    $display ("Nein");
```

```
    $display ("Nein");
```

```
  end
```

```
endmodule
```

Einzig 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

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

Beispiel: Einfache ALU

Verhaltensmodell



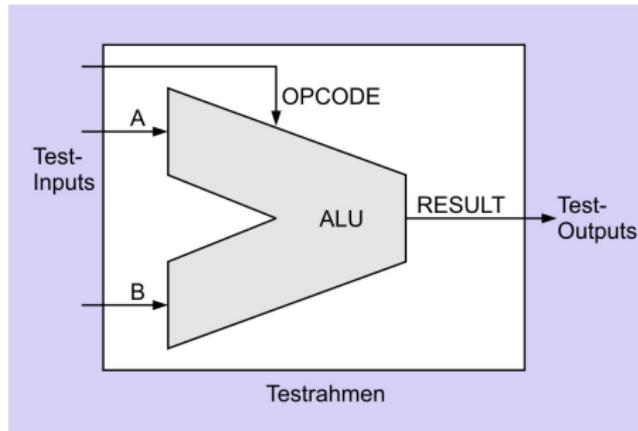
```
module alu (  
  input wire [2:0] OPCODE,  
  input wire [31:0] A,  
           B,  
  output reg [31:0] RESULT  
);  
  
'define ADD    3'b000 // 0    // nur zur Übung:  
'define MUL    'b001 // 1    // Konstanten auf  
'define AND    3'o2   // 2    // verschiedene Arten  
'define LOGAND 3'h3   // 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
```



Teststrahlen



- ▶ 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.



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

Testrahmen für die einfache ALU

```
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.



Verhalten und Struktur in Verilog

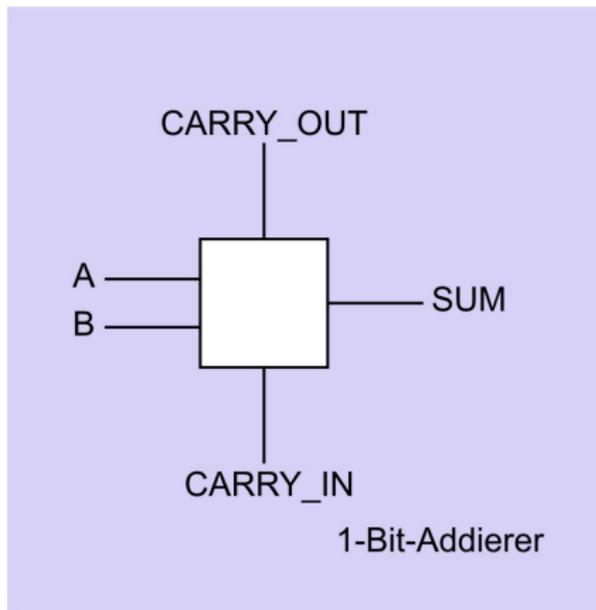


- ▶ Abbildung von Eingaben auf Ausgaben
- ▶ “was”, nicht “wie”
- ▶ Realisierung nicht von außen sichtbar (black box)
- ▶ Zur Modellierung reicht häufig ein einzelner always-Block



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

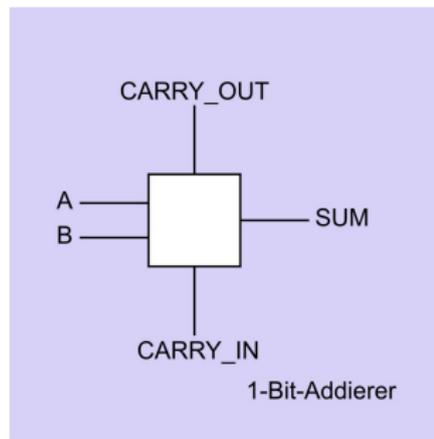
Beispiel: 1b-Addierer



Verhaltensbeschreibung des 1b-Addierers

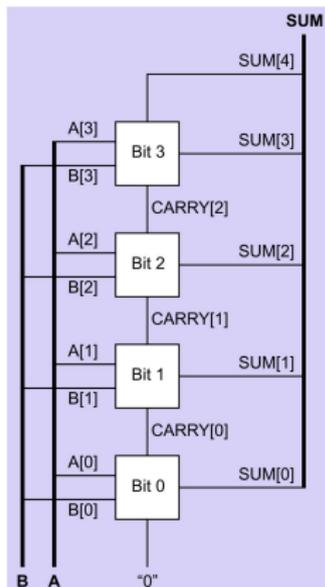
Konkreter Aufbau aus Gattern interessiert hier nicht

```
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
```

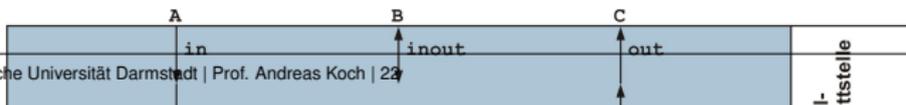
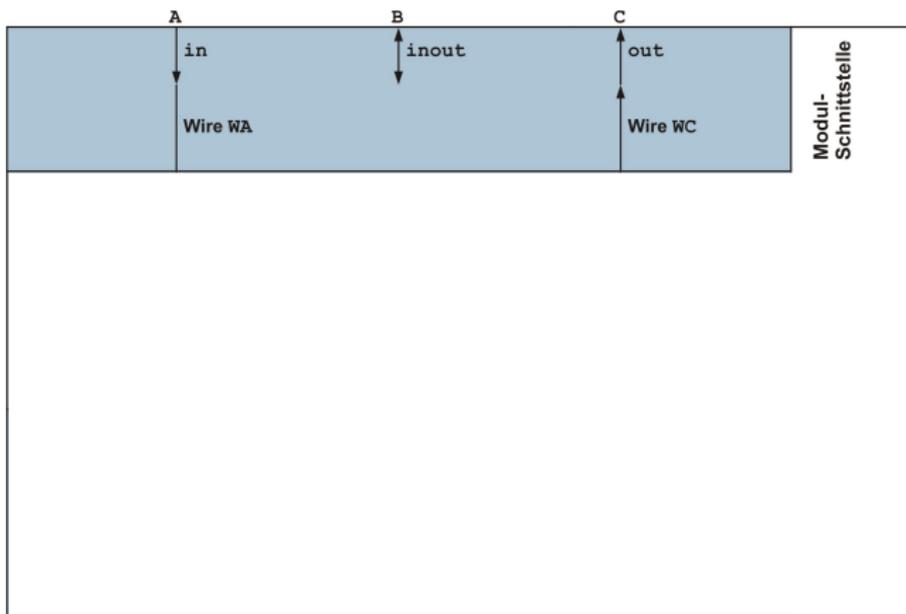


Elemente von Verilog-Modulen

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



Prozedurale Modellierung



- ▶ 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

- ▶ 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, Gemeinschaften 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 → 1 nicht sichtbar für always-Block
 - ▶ initial-Block läuft atomar ab

```
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

- ▶ # 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)

```
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
```

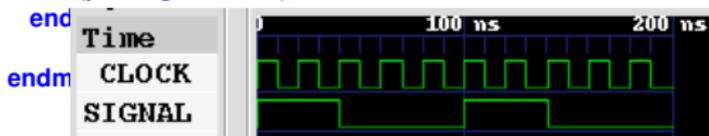
Alternative Benutzung von @

- ▶ @ 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);
```



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



```
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;

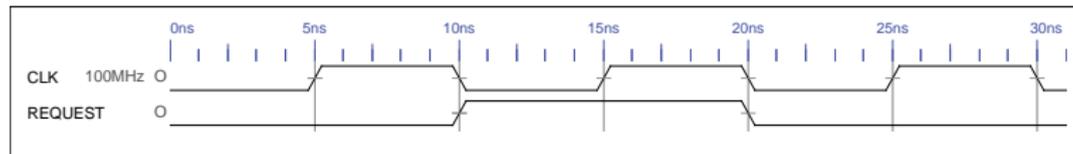
    task add(
        input reg [7:0] A,
            B
    );
        RESULT = A + B;
    endtask

    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



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

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 = xxxzzzzz  
DATA = 10101010  
DATA = 0000000z  
DATA = zzzzzzzz
```

- ▶ 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)



- ▶ 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

- ▶ 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 Breiterweiterung

```
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_s=s1+s2=001+110=00111    s=s1+s2=001+110=11111
             u=u1+s2=001+110=00111    s=s1+u1=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");
  end
endmodule
```



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`?

- ▶ 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.



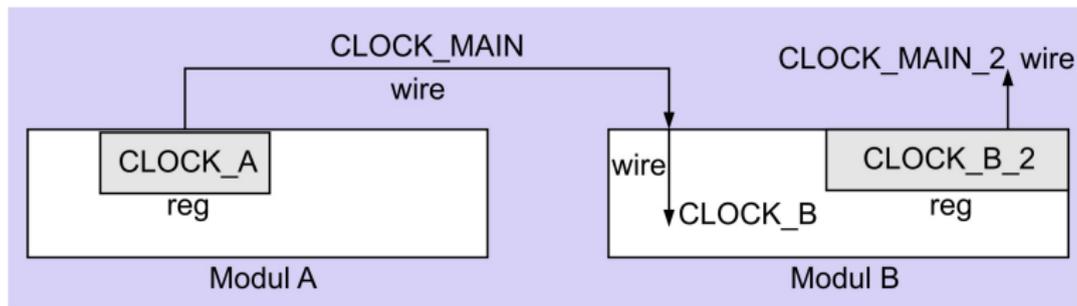
Verbinden von Elementen



- ▶ 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
```

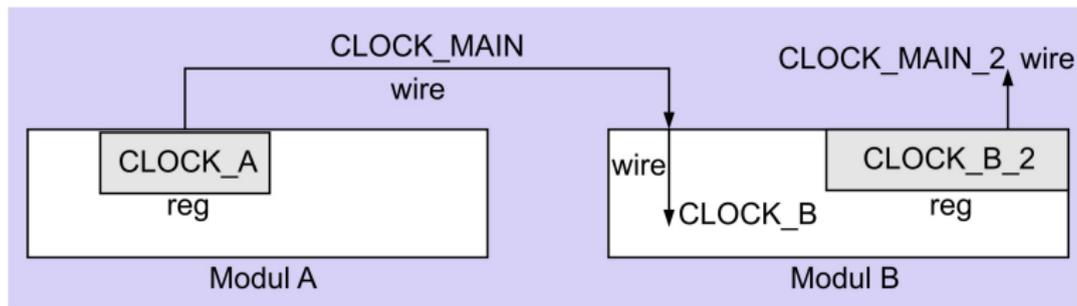
```
  CLOCK_A = 0;           // Clock auf 0 setzen  
  #10;                  // 10 Zeiteinheiten warten  
  CLOCK_A = 1;          // Clock auf 1 setzen  
  #10;                  // 10 Zeiteinheiten warten
```

```
end
```

```
endmodule // a
```

Beispiel: Wire und Register

Hier gezeigt: Modul B



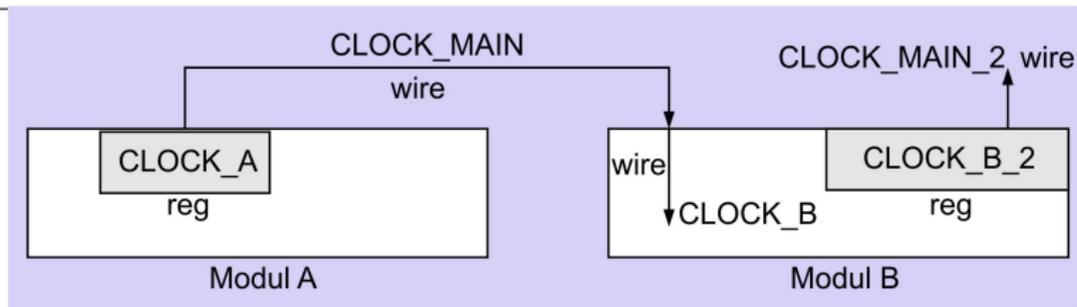
```
module b(  
  input wire CLOCK_B,    // Haupttakt  
  output reg CLOCK_B_2  // halb so schneller Takt  
);  
  
  initial begin  
    CLOCK_B_2 = 0;      // Startwert für CLOCK_B_2  
  end  
  
  always @(posedge CLOCK_B) begin  
    CLOCK_B_2 = ~CLOCK_B_2;  
  end  
  
endmodule
```

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
...
```

- ▶ `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]`

Mehrere Signale gleichzeitig auf einem Wire

```
module constant_example_2;
```

```
  reg [7:0] REG1,  
        REG2;
```

```
  wire [7:0] W = REG1;
```

```
  assign W = REG2;
```

```
  initial begin
```

```
    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
```



Operatoren



- ▶ +, -: 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 instantiiieren



$==, !=$ Logische Gleichheit/Ungleichheit

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

$===, !==$ Wörtliche Gleichheit/Ungleichheit

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



>, >, <=, >= Arithmetische Vergleiche

- ▶ Wenn einer der Operanden $\notin \{0, 1\}$ ist, liefere $1'b_x$
- ▶ Liefere $1'b_1$ wenn der Vergleich wahr ist, $1'b_0$ sonst
- ▶ Beachte korrekte **Vorzeichenbehaftung** der Operanden (signed)

Beispiele: Vergleiche

$(1'bx == 1'b1) = x$

$(1'bx === 1'b1) = 0$

$(1'bx == 1'bx) = x$

$(1'bx === 1'bx) = 1$

$(1'bz != 1'b1) = x$

$(1'bx <= 1'b1) = x$

$(2'bxx == 2'b11) = x$

$(3'h7 <= 3'h1) = 0$

$(3'sh7 <= 3'sh1) = 1$

Logische Operatoren: $!$, $\&\&$, $||$, \wedge

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

$$(! 1'b1) = 0$$

$$(! 1'bx) = x$$

$$(! 1'bz) = x$$

$$(1'b1 \&\& 1'b0) = 0$$

$$(1'bx \&\& 1'b0) = 0$$

$$(1'bx \&\& 1'b1) = x$$

$$(1'bx || 1'b0) = x$$

$$(1'bx || 1'b1) = 1$$

$$(2'b00 || 2'bxx) = x$$

- ▶ Gleiche Ideen wie bei den **logischen** Operatoren
- ▶ Nun aber auf jedes **einzelne** Bit angewandt

$$(\sim 4'bzx10) = 4'bxx01$$

$$(4'b001x \& 4'b0x10) = 4'b0010$$

$$(2'b1x | 2'b00) = 2'b1x$$

Konkatenation und Vervielfältigung mit { und }



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Konkatenation Zusammensetzen von Signalen zu größeren Einheiten

`{3'b100, 4'bxxxx, 2'ha}` ergibt `100_xxxx_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`

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
```



Feinheiten von Anweisungen

Blockende Zuweisung =

- ▶ 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 ...)`

- ▶ 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
 - ▶ Übernahme 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



```
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



```
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



```
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);
```

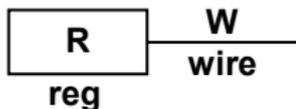
```
endmodule
```

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

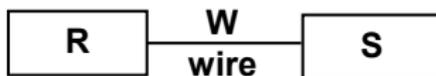
Ständige Zuweisung mit `assign`

continuous assignment

- ▶ `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

- ▶ 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
```

- ▶ 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 Instanziierung
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
```

- ▶ 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 → langsamer
- ▶ Bei RTL-Simulation nicht so kritisch
- ▶ Bei uns oft ausreichend:
 - ▶ `'timescale 1 ns / 1 ns`
 - ▶ `'timescale 1 ns / 10 ps`



Systemfunktionen

- ▶ Beide geben Text und formatierte Daten aus

- ▶ Formatierung

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



```
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 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



`$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