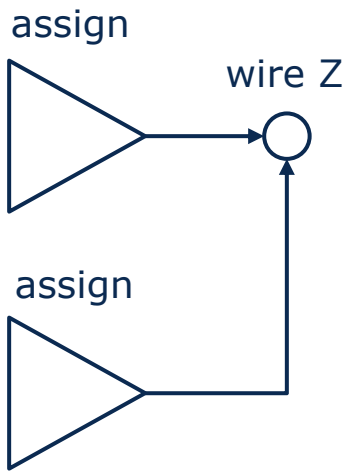
Verilog – Prozeduren

Prozedurale Zuweisungen

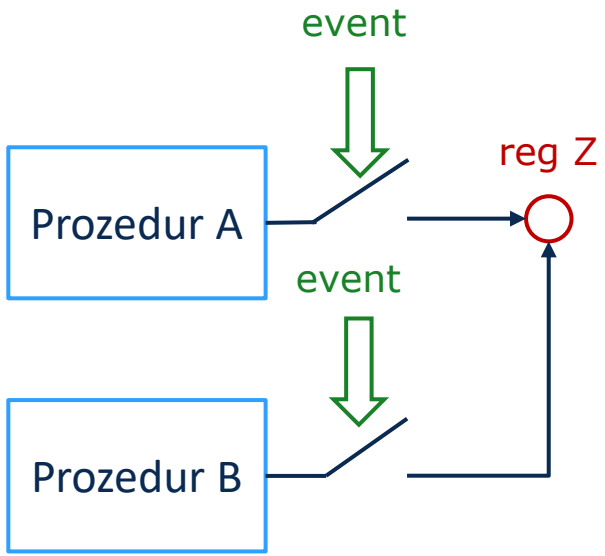
- Kontinuierliche Zuweisungen (assign)
  - → Treiben ein Netz (wire) permanent
  - → Wert ändert sich sobald die „inputs" der Zuweisung sich ändern

- Prozedurale Zuweisungen
  - → schreiben Wert auf einen Register-Datentyp (z.B. reg, integer)
  - → Register-Datentyp hält Wert bis zum nächsten prozeduralen Zugriff
  - → Zuweisungen werden getriggert durch die Control Flow Blöcke (Prozeduren)
    - initial
    - always
    - task
    - function
- Prozeduren können parallel und konkurrierend ablaufen

Kontinuierliche Zuweisung

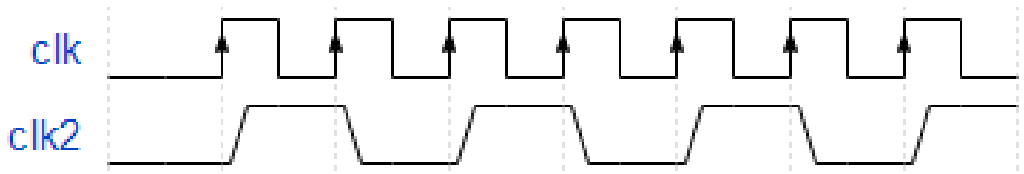


Prozedurale Zuweisung



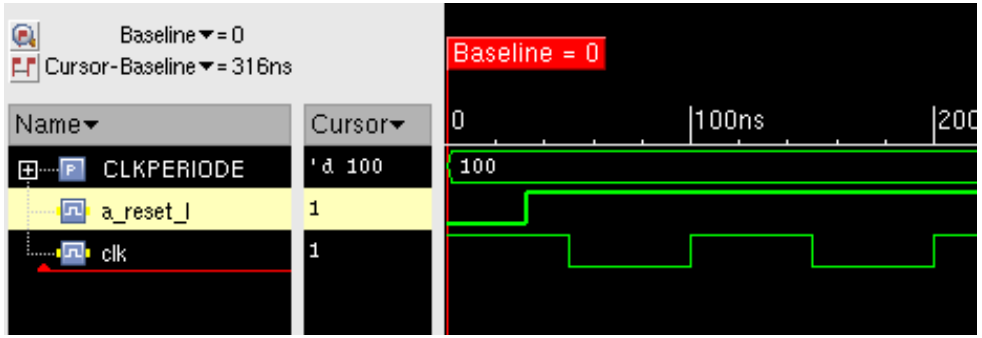
Beispiel: Prozedurale Zuweisungen

```
`timescale 1ns/1ps
module clock_div ();
   parameter PERIOD=10
   reg clk, clk2;
   initial begin
      clk=0;
      clk2=0;
   end

   always #PERIOD/2 clk=~clk;

   always @(posedge clk) begin
      clk2<=~clk2;
   end
endmodule
```



- **initial**
  - Ausführung zu Beginn der Simulation im Zeitschritt 0.
  - typische Anwendungen
    - Initialisieren von Registern
    - Starten von Stimuli (Waveforms) in Testbench Umgebungen
  - **nicht synthesefähig!**

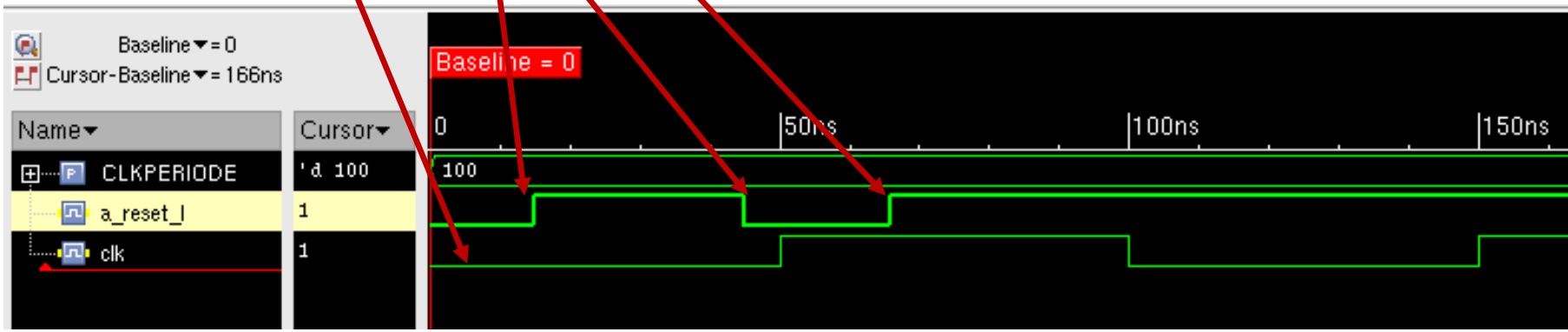
```
//Beispiel Initial Prozedur
reg clk;
reg a_reset_l;
initial clk = 1'b1;

always #(CLKPERIODE/2) clk = !clk;

initial begin
    a_reset_l = 1'b0;
    #33
    a_reset_l = 1'b1;
end
```



Konkurrierende Initial Prozeduren

```
reg clk;
reg a_reset_l;
initial clk = 1'b1;
initial clk = 1'b0;
always #(CLKPERIODE/2) clk = !clk;
initial begin
    #33 a_reset_l = 1'b0;
    a_reset_l = 1'b1;
    #33 a_reset_l =1'b1;
end
initial begin
     a_reset_l = 1'b0;
     #15 a_reset_l = 1'b1;
     #30 a_reset_l = 1'b0;
end
```

- Konkurrierende Ausführung von Prozeduren möglich
- Keine x-Werte wie bei kontinuierlicher Mehrfachzuweisung



Initialisierung von Register Datentypen

- Initialisierung von Register Datentypen ist bei Deklaration möglich
- Komfortabel für Testbenches
- **nicht synthesefähig!**

```
//Beispiel Initialisierung von Register Datentypen
reg clk=1'b1;
real a=3.245;
integer i=1;
```

always Prozedur

```
//Always Block
always @(…) begin
…
end
```

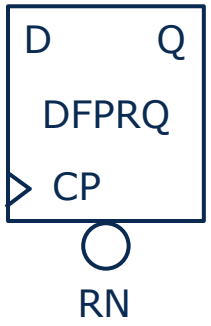
- Permanente Ausführung
- Sinnvolle Anwendung mittels „timing control“ Statements, z.B.
  - Verzögerte Zuweisungen
  - Event Steuerung
  - Sensitivity Liste
- geeignet zur Modellierung und Beschreibung von
  - sequentieller Logik
  - kombinatorischer Logik

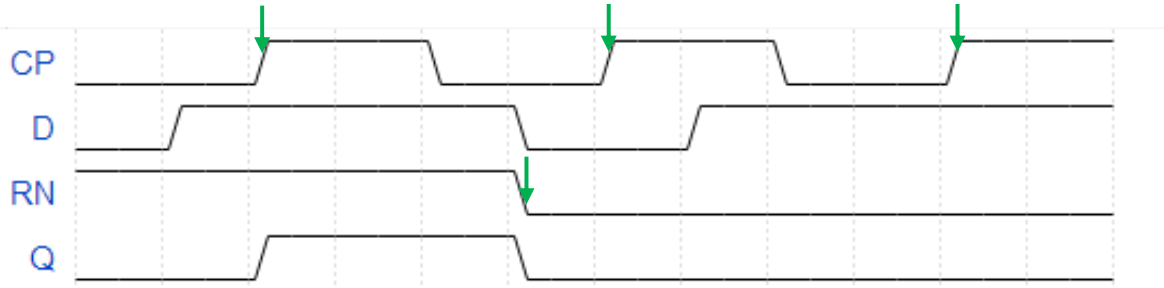
- Ausführung von always Prozeduren steuerbar durch „event control statements“ @
  - @ <identifier> → Signalwechsel (steigende oder fallende flanke)
  - @ posedge → steigende flanke
  - @ negedge → fallende flanke
- Kombination der events mit ‚OR‘ möglich

```
// Beispiele

//keine Event Control
always #(CLKPERIODE/2) clk = !clk;  //permanente Ausführung

always @(b) a=b;                    //Zuweisung getriggert durch b

always @(posedge clk) q <= d;       //steigende Taktflanke

always @(negedge clk) q <= d;       //fallende Taktflanke
```

Beispiel: Flip-Flop mit Asynchronem Reset

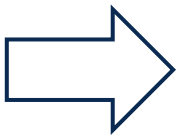
```
module DFPRQ (D,CP,Q,RN);

input D,CP,RN;
output Q;
reg q_reg;

always @(posedge CP or negedge RN) begin
   if (RN==1'b0) begin
      q_reg<=1'b0;
   end
   else begin
      q_reg<=D;
   end
end
assign Q=q_reg;

endmodule
```





- Ein kombinatorischer always Block wird durch eine Sensitivity Liste getriggert
- Coding Guidelines:
  - Alle rechtsseitigen Signale in Zuweisungen **müssen enthalten sein**!
  - Linksseitige Signale **sollen nicht enthalten** sein → kein „self-triggering"

```
// Beispiel Sensitivity Liste (XOR)

//vollständige Sensitivity Liste
always @(a or b) begin
   c1=a^b;
end
//fehlendes Signal in Sensitivity Liste
always @( a ) begin
   c2=a^b;
end
```



Mismatch zwischen RTL Simulation und Syntheseergebnis

„Blocking Assignments“

- Blocking Assignments (=) werden **vor** dem folgenden Statement in der Prozedur **unmittelbar** ausgeführt
- Unterbricht (blockt) den prozeduralen Ablauf

```
// Beispiel Blocking Assignment
initial begin
          a=0;
          b=0;
          a= #2 1;
          b= #4 1;
          a= #2 0;
          b=0;
end
```

- Blocking Assignments eigenen sich zur Beschreibung von Signalabläufen (Waveforms, Stimuli)

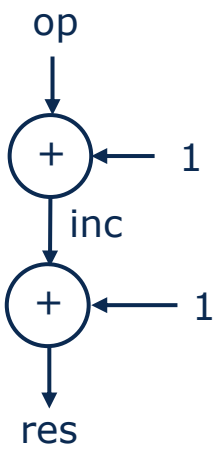
- Non-Blocking Assignments (<=) unterbrechen den prozeduralen Ablauf nicht
- Ausführung durch den Simulator in 2 Schritten:
  1. Auswertung der rechten Seite, **Vormerken** der Zuweisung auf die linke Seite
  2. **Durchführen** der Zuweisung zum Zeitpunkt des Event Control Statements

```
// Beispiel Non-Blocking Assignment
initial begin
        a <= 0;
        b <= 0;
        a <= #2 1;
        b <= #4 1;
        a <= #2 0;
        b <= 0;
end
```

- Non-Blocking Assignments eigenen sich zur Beschreibung von sequentiellen Logikblöcken deren Timing durch die Event Control Statements definiert ist (z.B. @(posedge clk))

Beispiel 1 zu Blocking und Non-Blocking Assignments

Kombinatorische Logik

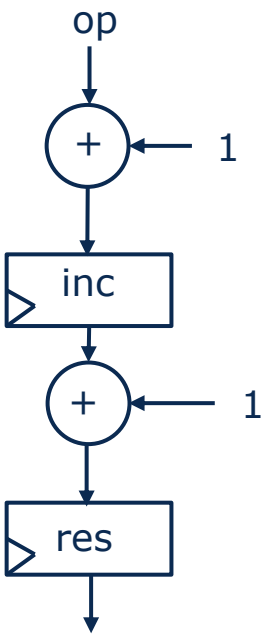



```
// Blocking Assignment
always @(op) begin
          inc1=op+1;
          res1=inc1+1;
end
```

```
// Non-Blocking Assignment
always @(op) begin
          inc2<=op+1;
          res2<=inc2+1;
end
```

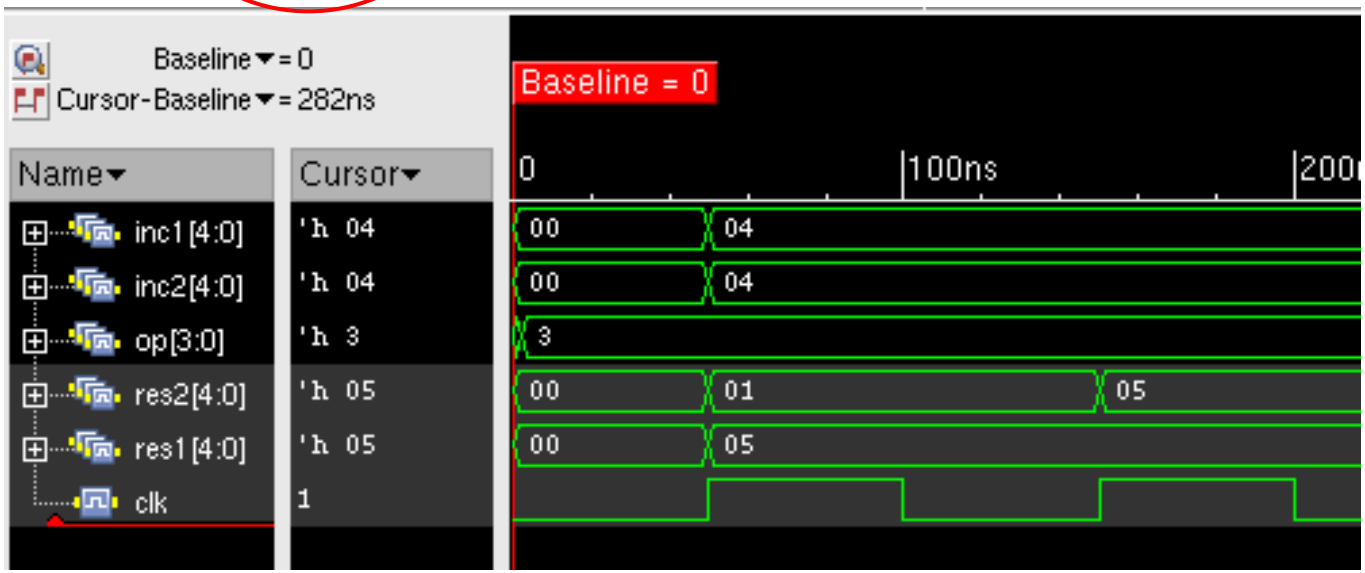
- Nutzung von **Blocking Assignments** zur Beschreibung **kombinatorischer Logik**!

Sequentielle Logik




```
// Blocking Assignment
always @(posedge clk) begin
   inc1=op+1;
   res1=inc1+1;
end
```

```
// Non-Blocking Assignment
always @(posedge clk) begin
   inc2<=op+1;
   res2<=inc2+1;
end
```

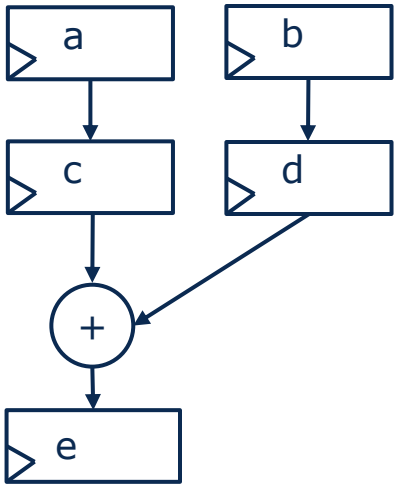


- Nutzung von **Non-Blocking Assignments** zur Beschreibung **sequentieller Logik**!

Beispiel 3 zu Blocking und Non-Blocking Assignments

- Nur eine Zuweisung pro always Block
- Ist die Nutzung von blocking oder non-blocking assignments egal?

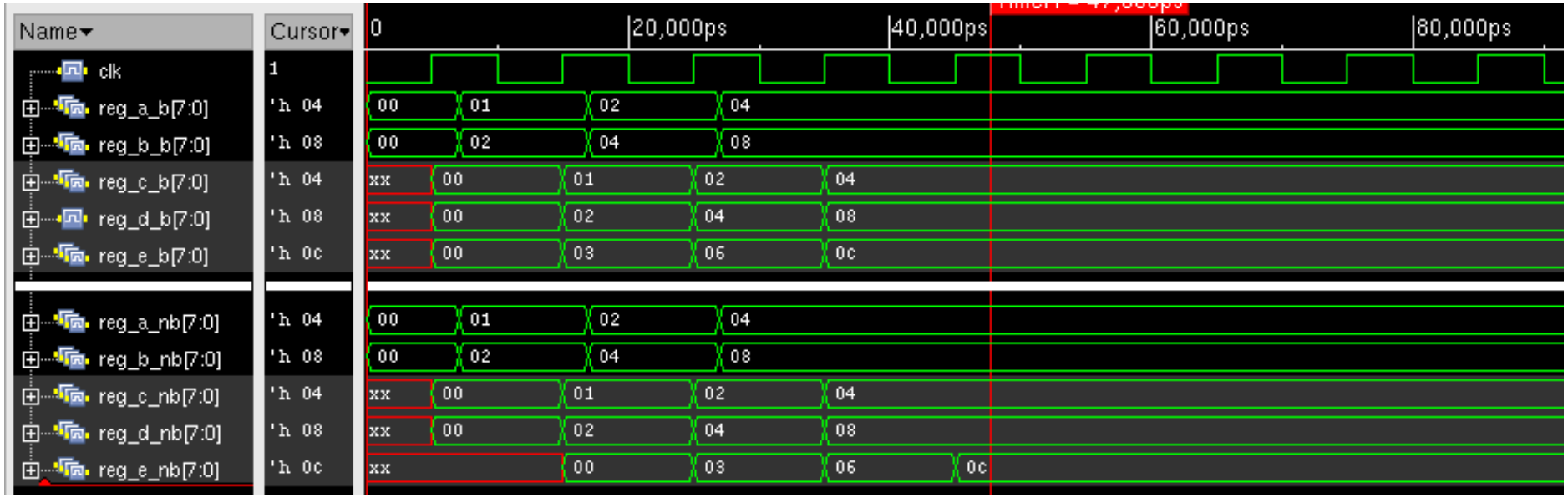
- Verwendung mehrerer dieser Blöcke (z.B. Instanzen eines Registers)
- Beispiel:




```
always @(posedge clk) begin
          reg_c_b=reg_a_b;
end
always @(posedge clk) begin
          reg_d_b=reg_b_b;
end
always @(posedge clk) begin
          reg_e_b=reg_c_b+reg_d_b;
end
```

```
always @(posedge clk) begin
          reg_c_nb<=reg_a_nb;
end
always @(posedge clk) begin
          reg_d_nb<=reg_b_nb;
end
always @(posedge clk) begin
          reg_e_nb<=reg_c_nb+reg_d_nb;
end
```

Beispiel 3 zu Blocking und Non-Blocking Assignments



- Nutzung von **Non-Blocking Assignments** zur Beschreibung **sequentieller Logik**!
- Auch bei nur **einer Zuweisung** pro always Block

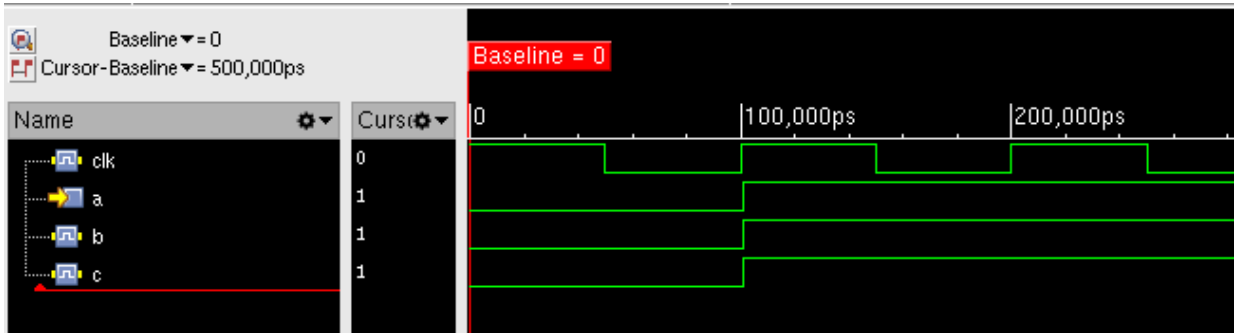
- **Delta Cycle:** Konzept in HDL Simulationen um Events zu ordnen, welche mit dem Zeitabstand von 0 (null) stattfinden
- **Zeitschritt:** Fortschreiten der Simulationszeit

```
input clk;
input a;
reg b,c;

always @(a)
begin
    b = a;
end

always @(b)
begin
    c = b;
end
```

Delta cycle:

```
b <- a
c <- b
```

nächster Zeitschritt: *keine Zuweisung*



→ **sofortige** Signalzuweisung

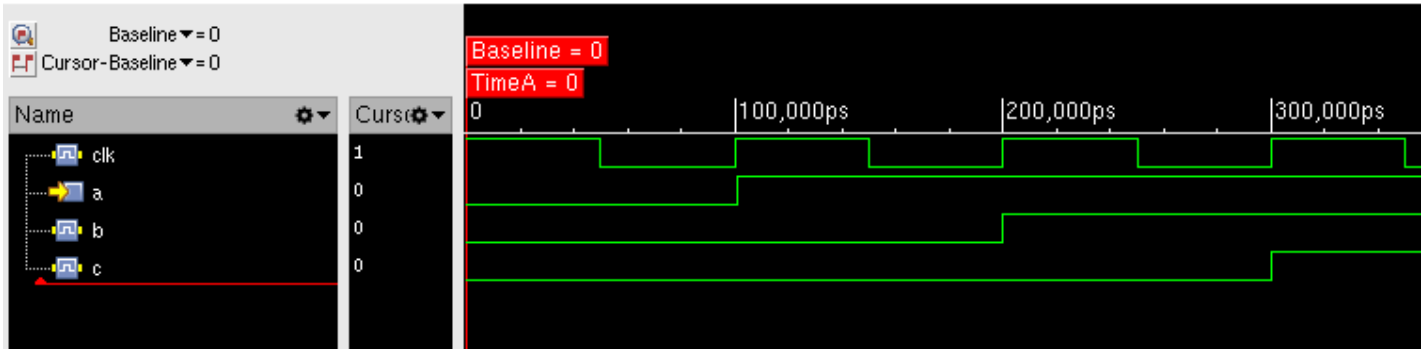
```
input clk;
input a;
reg b,c;

always @(posedge clk)
begin
    b <= a;
end

always @(posedge clk)
begin
    c <= b;
end
```

Delta cycle:

```
tmp_b <- a
tmp_c <- b
```

nächster Zeitschritt:

```
b <- tmp_b
c <- tmp_c
```



→ Signalzuweisung **bei Zeitschritt**

- Innerhalb von prozeduralen Zuweisungen können Control Statements verwendet werden zur bedingten Ausführung von Blöcken
  - if/else , case

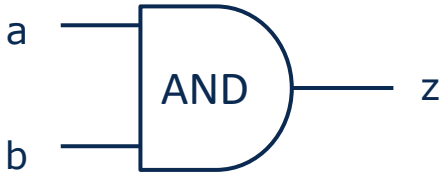
```
// Beispiel IF/ELSE
wire [1:0] a;
always @(a) begin
  if (a==0) begin
      b=1;
  end
  else if (a==1) begin
      b=2;
  end
  else begin
      b=3;
  end
end
```

```
// Beispiel case 1
wire [1:0] a;
always @(a) begin

case (a)
    2'b00:  b=1;
    2'b01:  b=2;
    2'b10:  b=3;
    2'b11:  b=3;
endcase
```

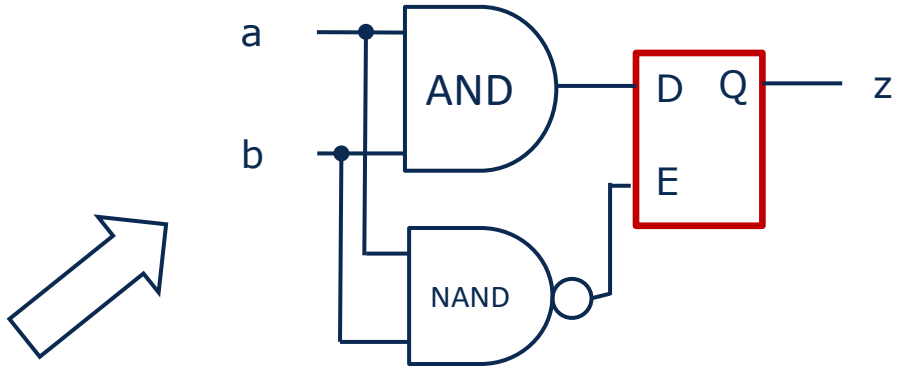
```
// Beispiel case 2
wire [1:0] a;
always @(a) begin

case (a)
    2'b00:  b=1;
    2'b01:  b=2;
    default: b=3;
endcase
```

Unerwünschte Beschreibung von Latches in kombinatorischer Logik

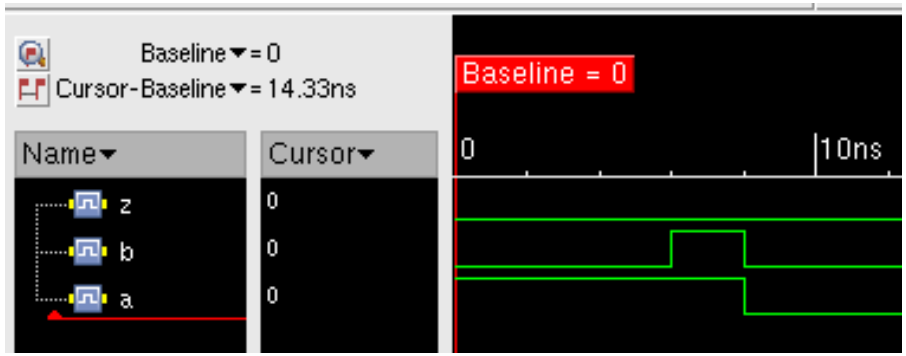
- Bei Control Statements zur Beschreibung kombinatorischer Logik sind Default Zuweisungen zwingend!
- Andernfalls werden sequentielle Elemente (Latches) beschrieben




```
// Beispiel IF/ELSE
// ohne default
always @(a or b) begin
  if ((a==0)||(b==0)) begin
      z=0;
  end
end
```

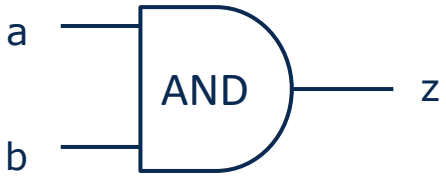


Latch eingefügt!



Korrekte Default Zuweisungen

- Korrekte Beschreibungsvarianten:




```
// Beispiel AND case 1
always @(a or b) begin
  case {a,b}
     2'b00: z=0;
     2'b01: z=0;
     2'b10: z=0;
     2'b11: z=1;
  endcase
 end
```

```
// Beispiel AND case 2
always @(a or b) begin
  case {a,b}
     2'b11:    z=1;
     default:  z=0;
   endcase
 end
```

```
// Beispiel AND IF/ELSE 1
always @(a or b) begin
  if ((a==0)||(b==0)) begin
       z=0;
  end
  else begin
     z=1;
  end
end
```

```
// Beispiel AND IF/ELSE 2
always @(a or b) begin
  z=1;
  if ((a==0)||(b==0)) begin
       z=0;
  end
end
```

- Blöcke in prozeduralen Zuweisungen können in Schleifen wiederholt werden
  - forever → Endloswiederholung
  - repeat → Wiederholungen gegebener Anzahl
  - while → Wiederholung solange Bedingung erfüllt
  - for → Klassische for loop

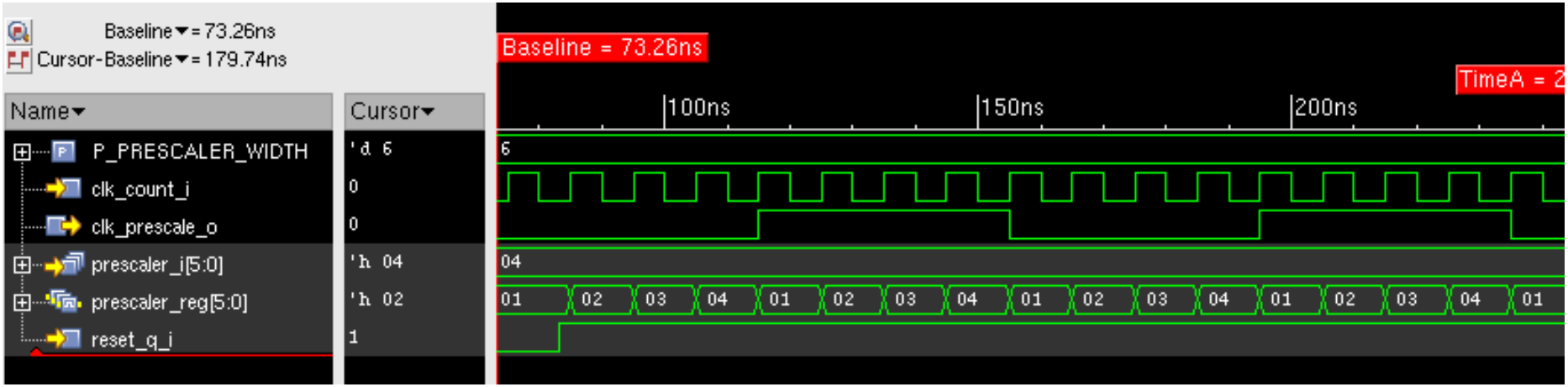
Beispiel: Kombinatorische Logik Thermometer Decoder

```
module thermo_decoder (bin_i, thermo_o);

   parameter BINBITS=5;
   parameter THERMOBITS=32;

   input   [BINBITS-1:0] bin_i;
   output  [THERMOBITS-1:0] thermo_o;

   reg [THERMOBITS-1:0] thermo_o;
   //thermo decoder block
   integer i;

   always @(bin_i) begin
      for (i=0;i<THERMOBITS;i=i+1) begin
         if (bin_i[BINBITS-1:0]>i) thermo_o[i]=1'b1;
         else thermo_o[i]=1'b0;
      end
   end
endmodule
```

Beispiel: Sequentielle Logik Frequenzteiler

```
module clk_prescaler (reset_q_i, clk_count_i, prescaler_i, clk_prescale_o);

parameter P_PRESCALER_WIDTH=6;
input     reset_q_i;
input     clk_count_i;
input     [P_PRESCALER_WIDTH-1:0]       prescaler_i;
output    clk_prescale_o;
reg [P_PRESCALER_WIDTH-1:0] prescaler_reg;
reg       clk_prescale_reg;

always @(posedge clk_count_i or negedge reset_q_i) begin
   if (reset_q_i == 1'b0) begin
      prescaler_reg<=1;
      clk_prescale_reg<=0;
   end
   else begin
      if (prescaler_reg==prescaler_i) begin
          clk_prescale_reg<=~clk_prescale_reg;
          prescaler_reg<=1;
      end
      else begin
          prescaler_reg<=prescaler_reg+1;
      end
   end
end
assign clk_prescale_o=clk_prescale_reg;
endmodule
```

Beispiel: Sequentielle Logik Frequenzteiler Waveform



Zusammenfassung: Modellierung kombinatorischer Logik

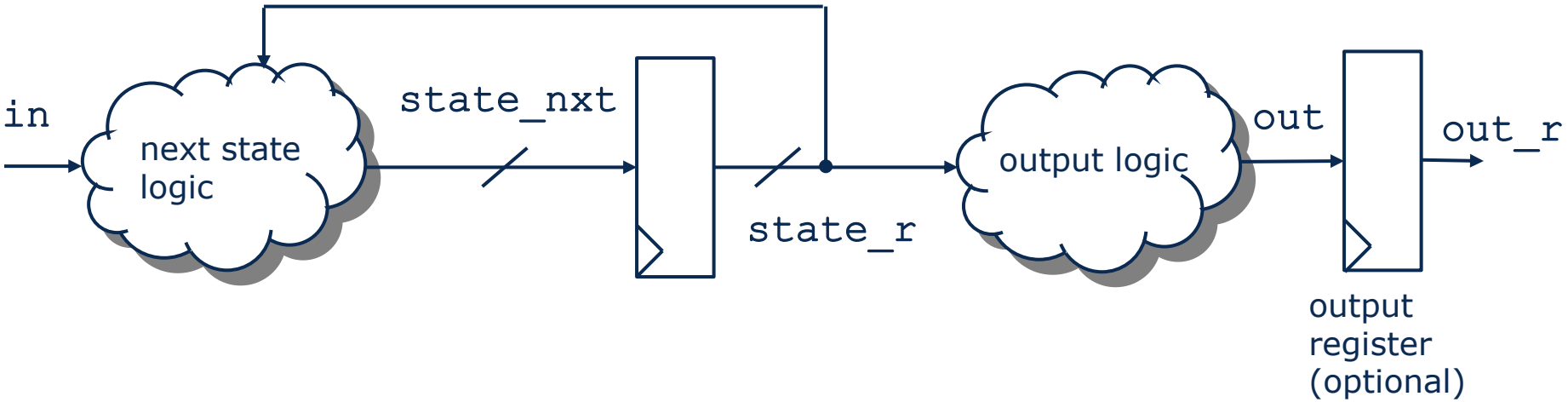
- Nutzung von always Blöcken möglich
- Vollständige Sensitivity Liste nötig
- Nutzung von Blocking Assignments (=)
- Default Zuweisungen oder vollständige Auscodierung von Control-Statements (if/else, case) zur Vermeidung von ungewollten Latches
- Zuweisen von kombinatorischen Signalen in nur **einem** always Block (keine konkurrierenden Zuweisungen)

Zusammenfassung: Modellierung sequentieller Logik

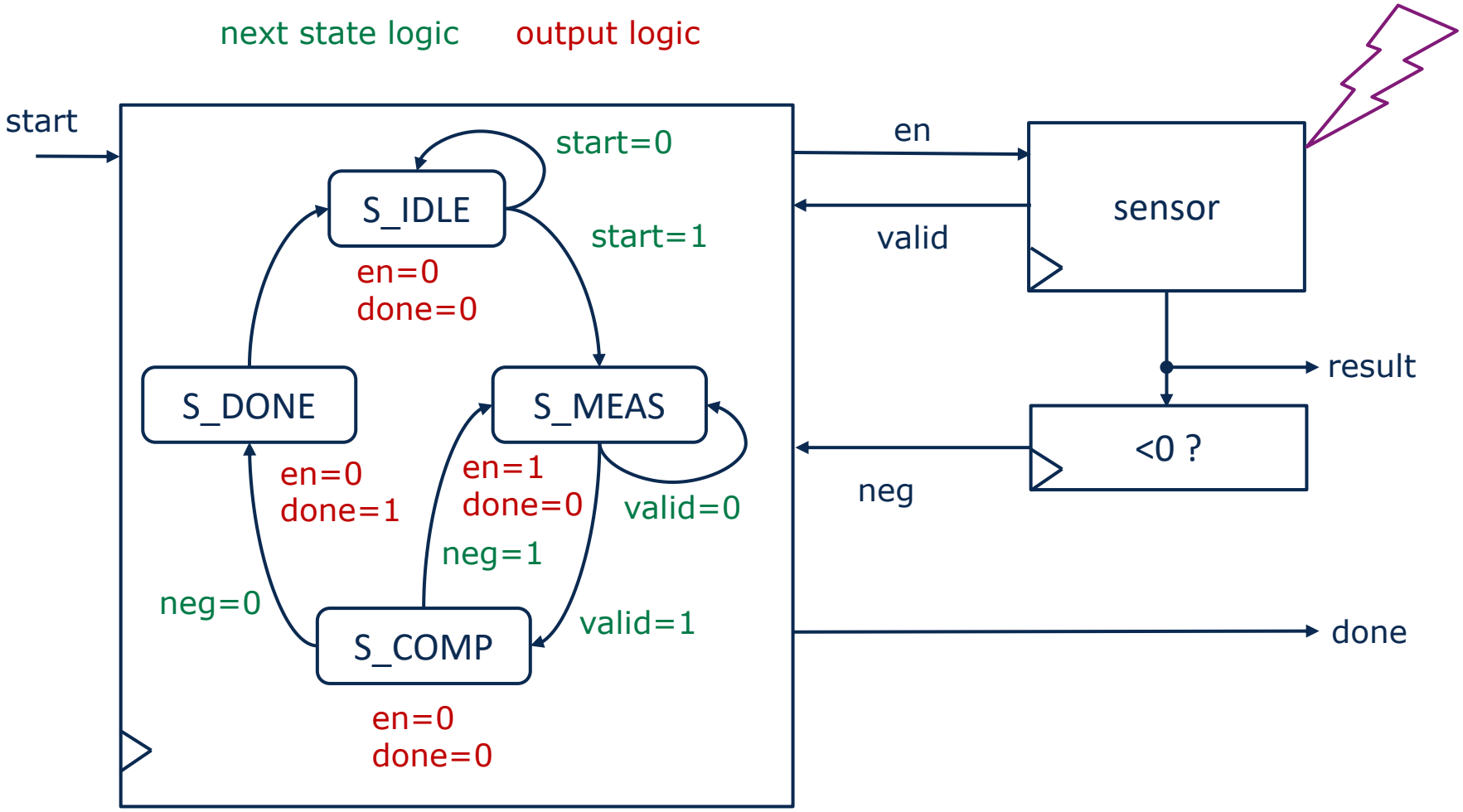
- Nutzung von always Blöcken zwingend
- Flankensensitives Event Control Statement als Trigger (z.B. @(posedge clk))
- Asynchrones set/reset möglich
- Always Blöcke mit asynchronem set/reset müssen alle Register Signale im Reset Block und im funktionalen Block enthalten
- Nutzung von non-blocking assignments (<=)
- Schreiben von Signalen in nur **einem** always block (keine konkurrierenden Zuweisungen)

Beschreibung von Zustandsautomaten (FSM)

- FSM: Finite State Machine
- Modelliert als Moore-Automat
  - Zustandsübergangsfunktion: state_nxt=G(state_r,in)
  - Ausgabefunktion: out=F(state_r)



Beispiel: FSM



Beispiel: FSM Sourcecode Header

```
// Company           :   tud
// Author            :   hoeppner
// E-Mail            :   <email>
//
// Filename          :   fsm.v
// Project Name      :   p_ice
// Subproject Name   :   s_verilog
// Description       :   <short description>
//
// Create Date       :   Wed Jan 22 10:16:59 2014
// Last Change       :   $Date$
// by                :   $Author$

//-----------------------------------------------------------------
module fsm (reset_q_i,clk_i,start_i,valid_i,neg_i,en_o,done_o);

input     reset_q_i;
input     clk_i;
input     start_i;
input     valid_i;
input     neg_i;
output    en_o;
output    done_o;
…
```

Beispiel: FSM Sourcecode Parameter und Signale

```
…
parameter S_IDLE=0;
parameter S_MEAS=1;
parameter S_COMP=2;
parameter S_DONE=3;

//registers
reg [1:0] state_r;

//comb signals
reg [1:0]         state_nxt;
reg               en;
reg               done;
…
```

Beispiel: FSM Sourcecode Zustandsübergangslogik

```
//next state logic
always @(state_r or start_i or valid_i or neg_i)
begin
case (state_r)
          S_IDLE: begin
                    if (start_i==1'b1) begin
                              state_nxt=S_MEAS;
                    end
                    else begin
                              state_nxt=S_IDLE;
                    end
          end
          S_MEAS: begin
                    if (valid_i==1'b1) begin
                              state_nxt=S_COMP;
                    end
                    else begin
                              state_nxt=S_MEAS;
                    end
          end
          …
```

```
…
S_COMP: begin
          if (neg_i==1'b1) begin
                    state_nxt=S_MEAS;
          end
          else begin
                    state_nxt=S_DONE;
          end
end
S_DONE: begin
          state_nxt=S_IDLE;
end
default: begin
          state_nxt=S_IDLE;
end
endcase
end
…
```

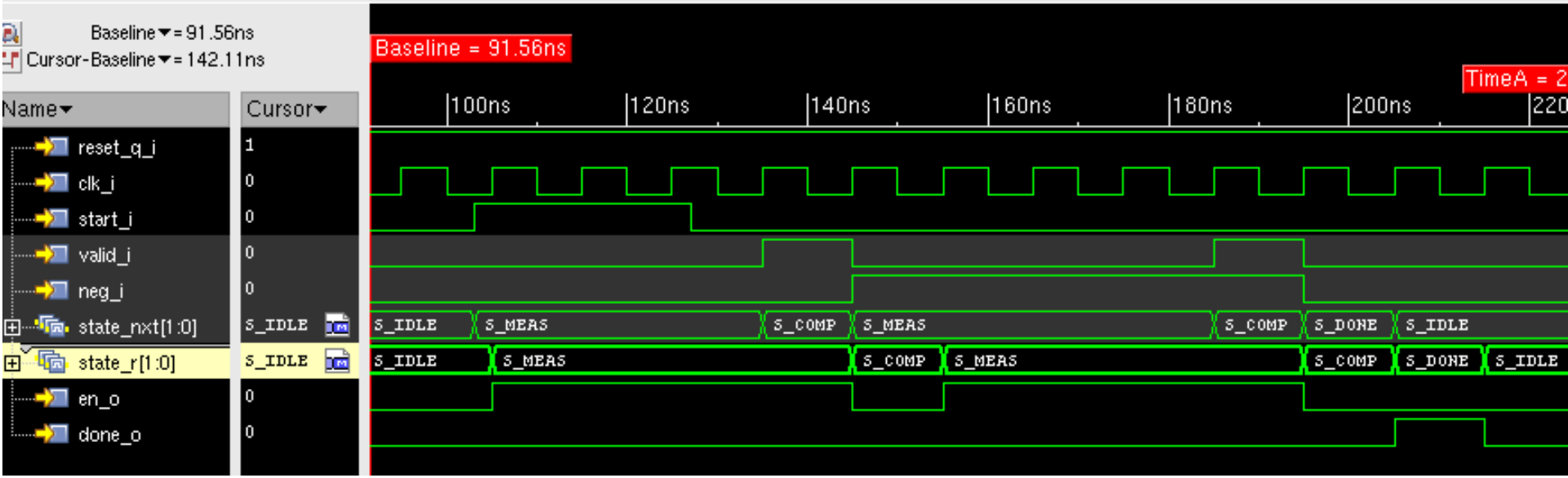
Beispiel: FSM Sourcecode Zustandsregister

```
…
//state register
always @(posedge clk_i or negedge reset_q_i)
begin
        if (reset_q_i==1'b0) begin
                state_r<=S_IDLE;
        end
        else begin
                state_r<=state_nxt;
        end
end
…
```

Beispiel: FSM Sourcecode Ausgangslogik

```
…
//output logic
always @(state_r) begin
        //default assignment
        en=0;
        done=0;
        if (state_r==S_MEAS) begin
                en=1;
        end
        if (state_r==S_DONE) begin
                done=1;
        end
end

//output assignment
assign en_o       = en;
assign done_o     = done;

endmodule
```

Beispiel: FSM Waveform



Guidelines zur Beschreibung von FSMs

- Nutzung von Parametern zur Benennung der Zustände
- Separierung von kombinatorischen und sequentiellen Schaltungsblöcken
  - Register
  - Zustandsübergangsfunktion
  - Ausgabefunktion
- Keine Zustandsübergangsfunktionen im sequentiellen Teil beschreiben
  - Ausnahme: synchrones Reset
- Separate Zuweisung der Ausgangssignale auf die „outputs“