Verilog – Hierarchisches Design

- Verilog bietet die Möglichkeit zur Partitionierung des Designs und zur Wiederverwendung von Designelementen
  - Module
  - Tasks
  - Funktionen
- Einbinden von Code Blöcken mit File Includes (´include)

Instanziierung von Modulen

Deklaration

```
module my_module (a, b, c);
 input a,b;
 output c;
…
endmodule
```

Instanziierung

```
…
reg a0;
wire b0,c0,c1;

my_module my_module_i0 (
        .a(a0),
        .b(b0),
        .c(c0)
);

my_module my_module_i1 (
        .a(a0),
        .b(c0),
        .c(c1)
);
```

- Module können instanziiert werden
- Inputs können von „reg“ und „wire“ Datentypen getrieben werden
- Outputs werden auf „wire“ Datentypen zugewiesen (kontinuierliche Zuweisung)

Parametrisierung bei Instanziierung

Deklaration

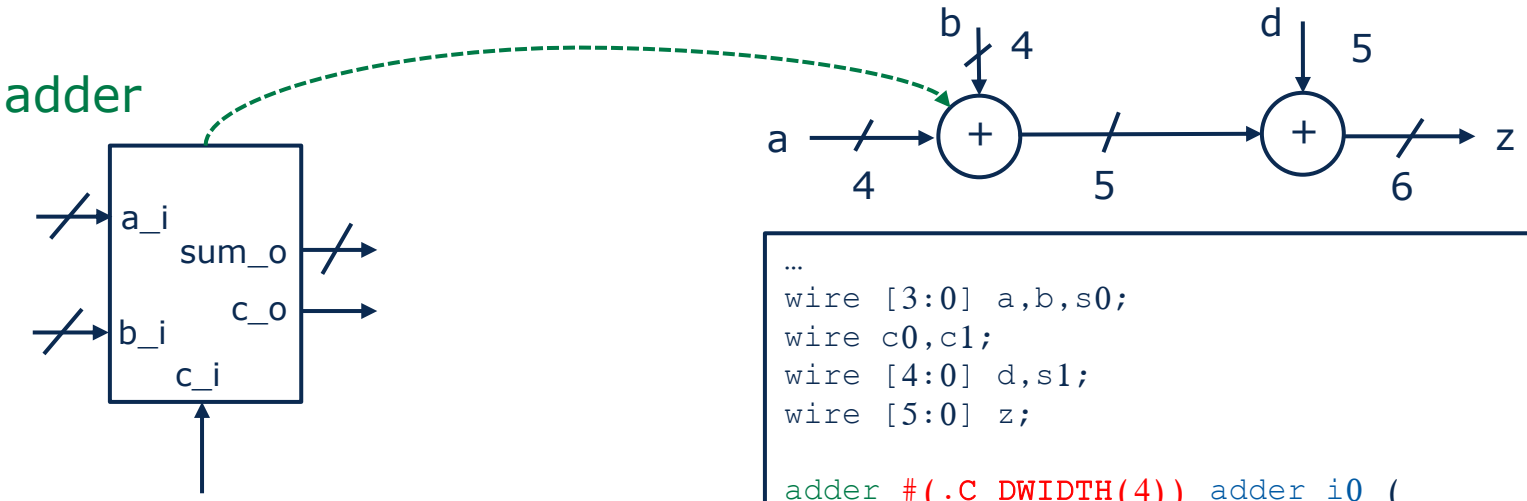
```
module my_module (a, b, c);

 parameter P0=1;
 parameter P1=2;

 input a,b;
 output c;
…
endmodule
```

Instanziierung

```
…
reg a0;
wire b0,c0,c1;

my_module #(.P0(2),.P1(3))
 my_module_i0 (
        .a(a0),
        .b(b0),
        .c(c0)
);
```

- Parameter von Modulen können bei der Instanziierung überschrieben werden
- Verschiedene Instanzen können individuell Parametrisiert werden

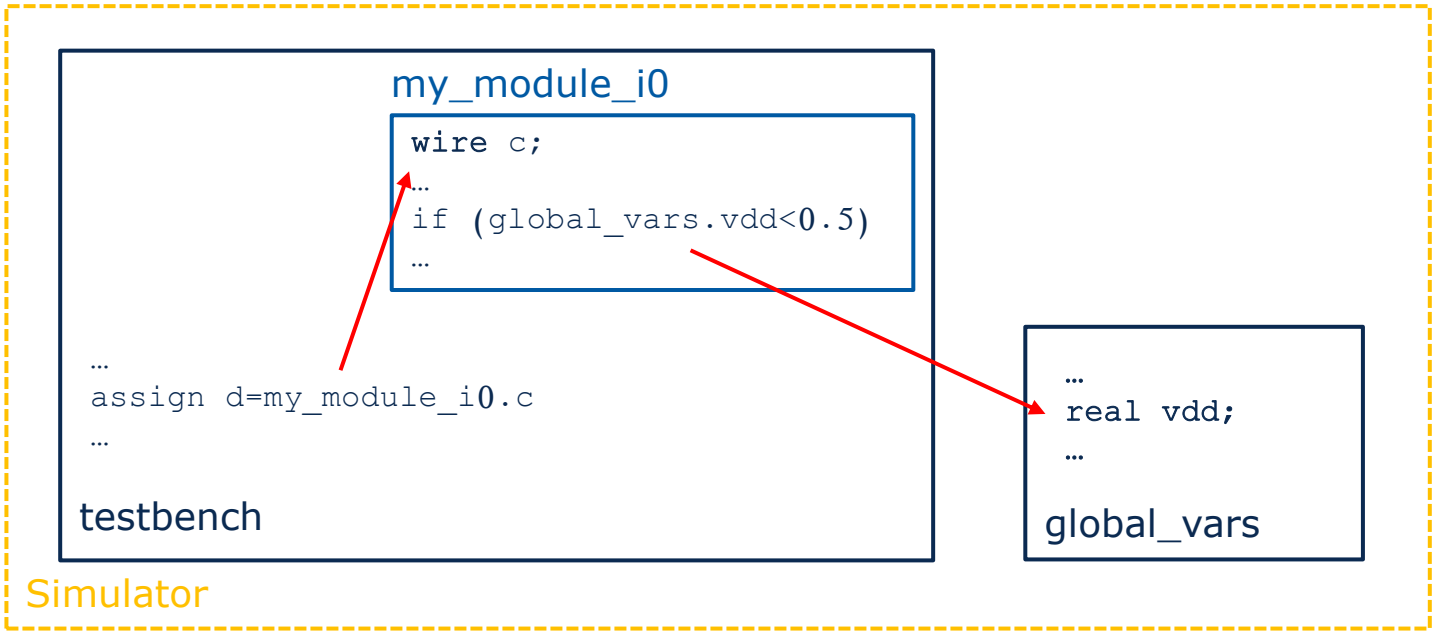
Beispiel: Instanziierung und Parametrisierung Addierer Datenpfad




```
//Addierer
module adder (sum_o, c_o, c_i, a_i, b_i) ;
   parameter C_DWIDTH=4;

   input [C_DWIDTH-1:0]  a_i, b_i;
   input c_i;
   output [C_DWIDTH-1:0] sum_o;
   output c_o;

   assign {c_o, sum_o} = a_i + b_i + c_i;
endmodule
```

```
…
wire [3:0] a,b,s0;
wire c0,c1;
wire [4:0] d,s1;
wire [5:0] z;

adder #(.C_DWIDTH(4)) adder_i0 (
    .sum_o(s0),
    .c_o(c0),
    .c_i(1'b0),
    .a_i(a),
    .b_i(b)
) ;
adder #(.C_DWIDTH(5)) adder_i1 (
    .sum_o(s1),
    .c_o(c1),
    .c_i(1'b0),
    .a_i({c0,s0}),
    .b_i(d)
) ;
assign z={c1,s1};
```

- Signale in Verilog sind global verfügbar
- Instanz-Hierarchien werden mit „ . „ im Signalnamen getrennt.
- **Nicht synthesegerecht!**
- Anwendungen:
  - Monitoring von Signalen in Testbenches zum Debugging und zur Verifikation
  - Für globale Netze die keine Signalpins sind (z.B. Versorgungsspannungen)
- ACHTUNG: Signale, die nicht über die Ports laufen sind ist auch in der implementierten Schaltung von außen nicht schreibbar/lesbar!



Beispiel: FlipFlop Modell mit Power-Shut-Off

Testbench

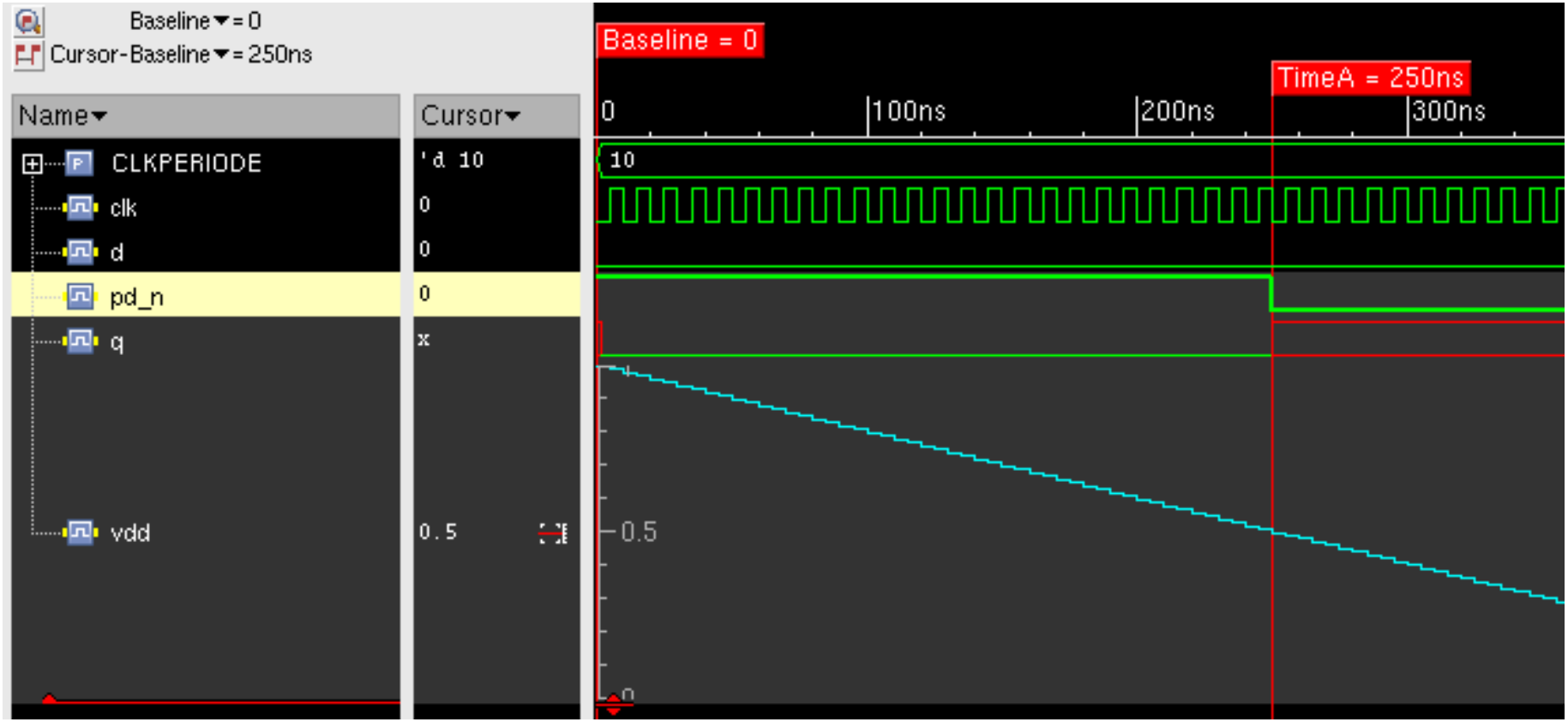
```
…
reg d,clk;
wire q;
wire pd_n;

reg_pd reg_i0 (
          .d(d),
          .cp(clk),
          .q(q)
);

//Observe internal power down signal
assign pd_n=reg_i0.pd_n;

//intialize FF without reset
initial #2 reg_i0.q_reg=$random;
```

Debug Probe

zufällige Initialisierung

```
module reg_pd (d, cp, q);

input d, cp;
output q;

wire pd_n;

reg q_reg;

assign pd_n=(global_vars.vdd>0.4);

always @(posedge cp or negedge pd_n) begin
   if (pd_n==1'b0) q_reg <=1'bx;
   else q_reg<=d;
end

assign q=q_reg;
endmodule
```

Beispiel: FlipFlop Modell mit Power-Shut-Off



- Funktionen können komplexe Zuweisungen kapseln:
  - → übersichtlicher Code
  - → Wiederverwendbarkeit
- Funktionen haben **einen** Rückgabewert.
- Eine Funktionsdefinition muss eine **Zuweisung** auf die Funktion beinhalten.
- Funktionen werden in **einem** Zeitschritt ausgeführt
- Eine Funktionsdefinition darf **keine zeitlichen Abläufe** (Delays, Event Control Statements) enthalten.
- Funktionen können keine Tasks aufrufen.
- Eine Funktion muss **mindesten ein** Input Argument haben.
- Eine Funktionsdefinition darf **keine** Argumente der Typen **output** oder **inout** besitzen.

- → Funktionen eignen sich zur Beschreibung kombinatorischer Logik

Beispiel Funktion

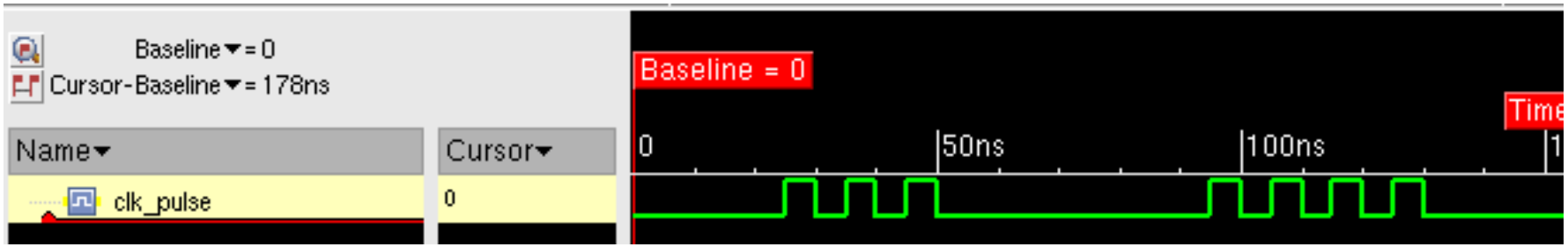
```
//Beispiel Funktionen
…
//Definition
function [5:0] adder_func;
   input [3:0] a,b;
   input [4:0] d;
   begin
      adder_func=a+b+d;
   end
endfunction
…
//Aufruf
assign result_1=adder_func(e,f,g);
…
always @(opa or opb or opd) begin
   result_2=adder_func(opa,opb,opd);
end
…
```

- Tasks können komplexe Zuweisungen einschließlich zeitlicher Abfolgen kapseln:
  - → übersichtlicher Code
  - → Wiederverwendbarkeit
- Tasks können **beliebig viele Inputs und Outputs** haben.
- Task können **zeitliche Abläufe** (Delays, Event Control Statements) enthalten.
- Tasks können Funktionen und andere Tasks aufrufen.
- Tasks können auf globale Variablen zugreifen.

- Tasks eignen sich zur Beschreibung kombinatorischer Logik, sequentieller Logik und Stimuli

Beispiel: Task zur Takt-Puls Erzeugung

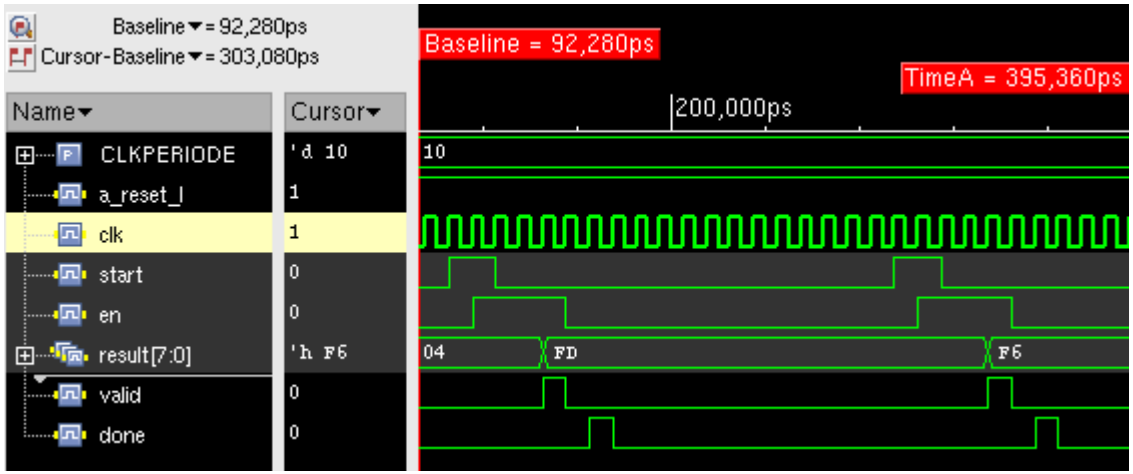
```
reg clk_pulse;
initial clk_pulse=1'b0;

task clk_pulses;
   input integer nr;
   integer i;
   begin
      for (i=0;i<nr;i=i+1) begin
         #(CLKPERIODE/2) clk_pulse=1'b1;
         #(CLKPERIODE/2) clk_pulse=1'b0;
      end
    end
endtask

initial begin
          #20 clk_pulses(3);
          #40 clk_pulses(4);
end
```



Beispiel: Task zum Starten einer FSM

- Starten der FSM aus Beispiel ab Folie 81

```
task run_meas;
   begin
      #(CLKPERIODE/2) start=1'b1;
      #(2*CLKPERIODE) start=1'b0;
      wait(done==1'b1);
      #(2.5*CLKPERIODE)
      $display("DONE");
   end
endtask

…

initial begin
          #100 run_meas;
          //do results processing
          #100 run_meas;
          //do results processing
end
```



File Includes

- Funktionen und Tasks müssen in dem Modul definiert werden, in dem sie genutzt werden.
- File includes erlauben die Verwendung vordefinierter Code-Fragmente

```
//Beispiel File Include
module testbench ()

   `include "my_tasks.v"

   `include "my_functions.v"

   `include „stim_clk_gen.v"

   initial begin
   …
   //testcase code
   …

end
endmodule
```

```
//my_tasks.v
task my_task1;
…
endtask

task my_task2;
…
endtask
```

```
//my_functions.v
function my_func1;
…
endfunction

function my_func2;
…
endfunction
```

```
//stim_clk_gen.v
parameter CLKPERIODE = 10;
reg clk;
initial clk = 1'b0;
always #(CLKPERIODE/2) clk = !clk;
```

Systemfunktionen und -tasks

- Verilog besitzt eingebaute Systemfunktionen und -tasks
- Hier einige wichtige:
  - $display
    - Formatierte Ausgabe auf die Kommandozeile
  - $fopen, $fwrite, $fflush, $fclose
    - Schreiben in eine Datei
  - $readmemh, $readmemb
    - Laden eines Speichers aus einer Textdatei
  - $random
    - Erzeugen einer Zufallszahl
  - $dist_normal
    - Erzeugung einer Gauss-Verteilten Zufallszahl
  - $realtime
    - Rückgabe der aktuellen Simulationszeit als real
  - $finish
    - Beenden der Simulation

Beispiel: Laden eines Speichers in der Simulation

mem_content.txt

```
12abc  34def  1dead 2bee1
```

```
module mem_init;

reg [19:0] memory [0:3];

initial $readmemh("mem_content.txt", memory);

integer i;
initial begin
   $display("rdata:");
   for (i=0; i < 4; i=i+1)   begin
     $display("%d:%h",i, memory[i]);
   end
end
endmodule
```

```
rdata:
0:12abc
1:34def
2:1dead
3:2bee1
```

simulation output

Beispiel: Oszillator mit Jitter

```
parameter NOISE_SEED=321;
parameter SIGMA_T=0.5;

reg clk_jitter=0;
real period_jitter=0;
real period=CLKPERIODE;
integer seed=NOISE_SEED, mean=0, stdev=1000000, random_value_int;

always @(period_jitter)  begin
           period=CLKPERIODE+period_jitter;
           if (period<=0) period=0.001;
  end
always @(posedge clk_jitter) begin
          random_value_int=$dist_normal(seed,mean,stdev);
          period_jitter=random_value_int/1000000.0*SIGMA_T*1.4142135;
end
always #(period/2)  clk_jitter=~clk_jitter;

real period_meas=0;
real prev_event_time=0;
real event_time=0;
always @(posedge clk_jitter) begin
          prev_event_time=event_time;
          event_time=$realtime;
          period_meas=event_time-prev_event_time;
end
```

Beispiel: Oszillator mit Jitter