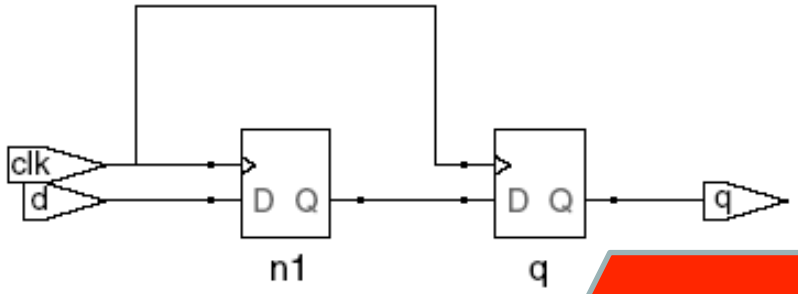

Danke für das Photo!
Habt Ihr überlebt?
Next person to talk in TGDI? Deal with me!

Nicht-blockende Zuweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

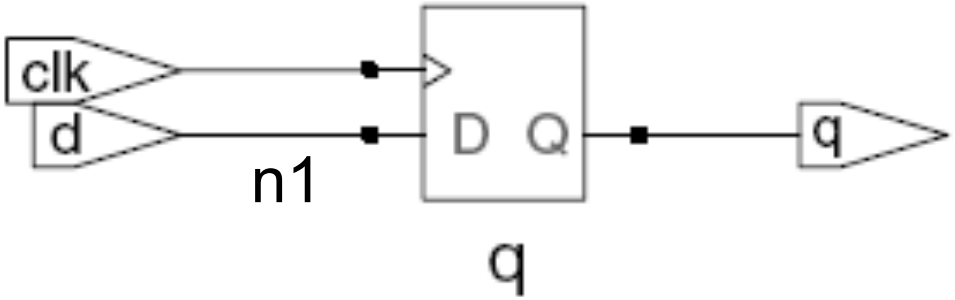


LEHRE WIKI TEST IN DREI FOLIEN

Blockende Zuweisung

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

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



Blocking vs. Non-Blocking

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Eingangswerte
// a = 0
// b = 1

always_comb
 begin

 a = b;

 c = a & b;

end

// Ausgangswerte
// a =
// b =
// c =
```

```
// Eingangswerte
// a = 0, b = 1

always_comb
 begin

 a <= b;

 c <= a & b;

end

// Ausgangswerte
// a =
// b =
// c =
```

Regeln für Zuweisungen von Signalen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Um synchrone sequentielle Logik zu beschreiben, benutzt immer
  - `always_ff @(posedge clk)`
  - Nicht-blockende Zuweisungen

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

- Um einfache kombinatorische Logik zu beschreiben, benutzt immer
  - Ständige Zuweisung (*continuous assignment*)

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

- Um komplexere kombinatorische Logik zu beschreiben, benutzt immer
  - `always_comb begin ... end`
  - Blockende Zuweisungen

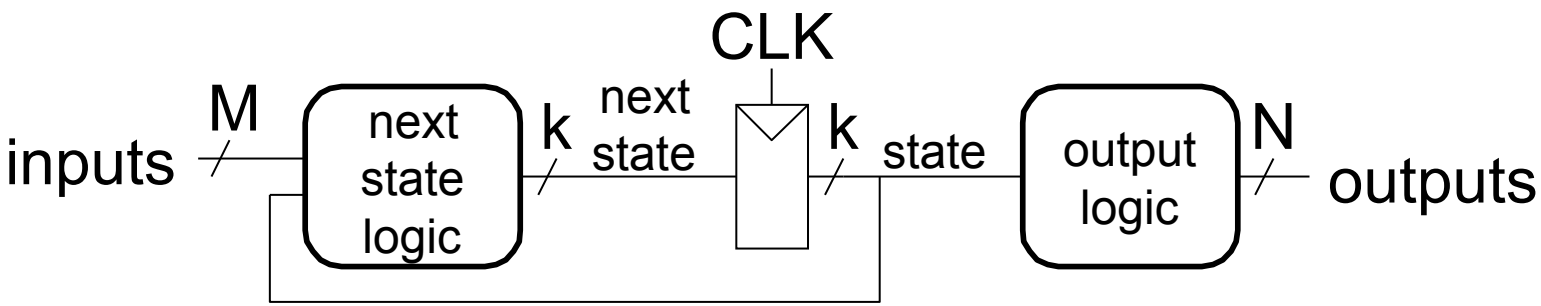
- Weisen Sie nicht an ein Signal
  - … in mehreren `always_XXX`-Blöcken zu
  - … in einem `always_XXX`-Block gemischt mit = und <= zu

LEHRE WIKI FRAGE

- Bitte jetzt auf LEHRE WIKI eine Frage beantworten!

Endliche Zustandsautomaten (FSM)

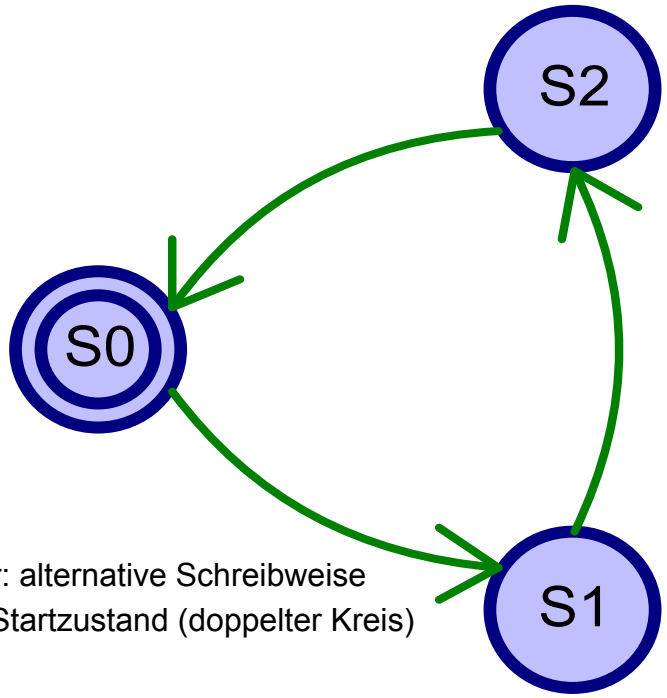
- Drei Blöcke:
  - Zustandsübergangslogik (*next state logic*)
  - Zustandsregister (*state register*)
  - Ausgangslogik (*output logic*)



Beispiel-FSM: Dritteln der Taktfrequenz

- Eingabe:
  - Explizit kein Signal
  - Implizit den Schaltungstakt
    - Mit Frequenz *f*

- Ausgabe
  - Signal `q` mit Frequenz *f*/3



Hier: alternative Schreibweise
für Startzustand (doppelter Kreis)

FSM in Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module divideby3FSM(input logic clk, input logic reset, output logic q);
   logic[1:0] state, nextstate;
   parameter S0 = 2'b00;      // Kodierung der Zustände
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
   always_ff @ (posedge clk, posedge reset)//Zustandsregister
      if (reset) state <= S0;
      else       state <= nextstate;
  always_comb begin // Zustandsübergangslogik
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
    end
   assign q = (state == S0);                  // Ausgangslogik
endmodule
```

Parametrisierte Module

TECHNISCHE UNIVERSITÄT DARMSTADT

2:1 Multiplexer:

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

Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
mux2 mux1(d0, d1, s, out);
```

Instanz mit 12-bit Busbreite: `mux2 #(12) lowmux(d0, d1, s, out);`

Aber besser (falls mehrere Parameter auftreten sollten): `mux2 #(.WIDTH(12)) lowmux(d0, d1, s, out);`

Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

Der Modulname sei **`sillyfunction`**

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

Der Modulname sei **`sillyfunction`**

SystemVerilog

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

Einfacher Testrahmen für Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

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

  sillyfunction dut(a, b, c, y);//Instanz des zu testenden Mod.

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

Selbstprüfender Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module testbench2 ();
 logic  a, b, c; logic y;

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

 // Eingangswerte anlegen, warten,
 // Ausgang mit erwartetem Wert prüfen
 initial begin
   a = 0; b = 0; c = 0; #10;
   if(y !== 1) $display("000 Fehler");

   c = 1; #10;
   if(y !== 0) $display("001 Fehler");

   b = 1; c = 0; #10;
   if(y !== 0) $display("010 Fehler");

   c = 1; #10;
   if (y !== 0) $display("011 Fehler");

   a = 1; b = 0; c = 0; #10;
   if (y !== 1) $display("100 Fehler");

   c = 1; #10;
   if(y !== 1) $display("101 Fehler");

   b = 1; c = 0; #10;
   if(y !== 0) $display("110 Fehler");

   c = 1; #10;
   if (y !== 0) $display("111 Fehler");
 end
endmodule
```

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Trennen von HDL-Programm und Testdaten

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

- Eigene Datei für Vektoren

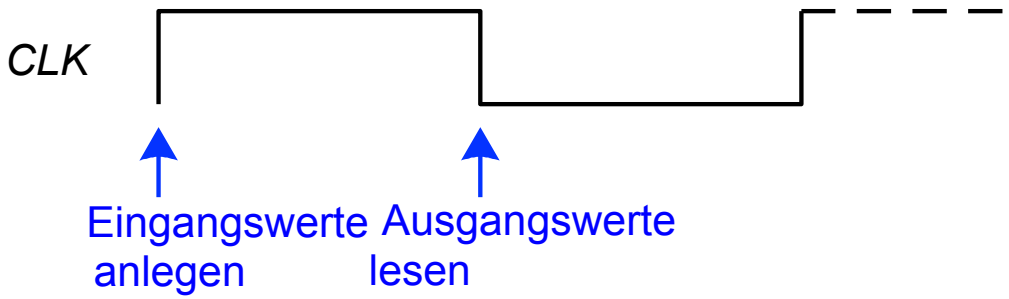
- Dann HDL-Programm für universellen Testrahmen

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

Selbstprüfender Testrahmen mit Testvektoren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Im Testrahmen erzeugter Takt legt zeitlichen Ablauf fest
  - Steigende Flanke: Eingabewerte aus Testvektor an Eingänge anlegen
  - Fallende Flanke: Aktuelle Werte an Ausgängen lesen



- Takt kann auch als Takt für sequentielle synchrone Schaltungen verwendet werden

Einfaches Textformat für Testvektordateien

TECHNISCHE UNIVERSITÄT DARMSTADT

Datei: `example.tv`

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

Aufbau:

Eingangsdaten "_" erwartete Ausgangsdaten

Testrahmen: 1. Erzeuge Takt

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

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

2. Lese Testvektordatei in Array ein

TECHNISCHE UNIVERSITÄT DARMSTADT

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

 initial      // Block wird genau einmal ausgeführt
    begin
      $readmemb("example.tv", testvectors);

      vectornum = 0; errors = 0; // Verwaltungsdaten initial.

      reset = 1; #27; reset = 0; // Reset-Impuls erzeugen

    end
…
```

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

3. Lege Testdaten an Eingänge an

TECHNISCHE UNIVERSITÄT DARMSTADT

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

`a,b,c` sind Eingänge der DUT

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

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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Hinweis: Um Werte hexadezimal auszugeben, Formatkennung %h verwenden

Beispiel:

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

6. Sind noch weitere Testvektoren abzuarbeiten?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
…
      // Array-Index zum Zugriff auf nächsten Testvektor erhöhen
      vectornum = vectornum + 1;

      // Ist der nächste schon ein ungültiger Testvektor?
      if (testvectors[vectornum] === 4'bx) begin

        $display("%d Tests bearbeitet mit %d Fehlern”, // Endmeldung
                       vectornum, errors);              // ausgeben
        $finish;       // Simulation anhalten
      end
    end
endmodule
```

Hinweis: Zum Vergleichen auf X und Z müssen die Operatoren

```
=== und !==
```

benutzt werden

SystemVerilog Sprachkonstrukte in TGDI

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vor Testrahmen alle für die Beschreibung von echter Hardware relevanten eingeführt
  - Schaltungssynthese
- Verilog kann viel mehr
  - Angedeutet beim Testrahmen (Dateioperationen, Ein/Ausgabe, …)
  - Aber in der Regel nicht mehr in Hardware synthetisierbar
  - Nicht Schwerpunkt dieser Veranstaltung
- Mehr Details in Kanonik Computer Microsystems
  - Im Sommersemester, dann SystemVerilog und BlueSpec-Erweiterungen
- In TGDI soll dieser Kurzüberblick reichen
  - Bei akutem Bedarf werden noch weitere Konstrukte eingeführt