Digitaltechnik
Wintersemester 2017/2018
11. Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

Inhalt

TECHNISCHE UNIVERSITÄT DARMSTADT

Einleitung

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zusammenlegung von Übungsgruppen ab KW 02
  - G20 $\rightarrow$ G12
    - Do 13:30-15:10 S103/313
    - Tobias Stöckert
    - Michael Tilli
- Klausurvorbereitung
  - Anmeldung für Fachprüfung bis 31.01.2018
  - erwartete Bearbeitungszeit für Ü1 bis Ü5 ergänzt
  - SystemVerilog Syntax-Blatt bis Ende KW02 im Moodle verfügbar
  - Wiederholung spezifischer Fragen am 05.02.18

  $\Rightarrow$ Themen im Moodle vorschlagen

Nicht nochmal Plätzchen backen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Aktualisierung im der Back-Pipeline im Moodle verfügbar (V10)

Rückblick auf die letzte Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Historie von Hardwarebeschreibungssprachen
- SystemVerilog für kombinatorische Logik
- SystemVerilog Modulhierarchie

Harris 2013
Kap. 4.1-4.3

Wiederholungs-Bedarf
laut Moodle Abfrage und Übungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Details zu `<signal>[<range>]`
- Struktur- und Verhaltensbeschreibung
- Operatoren und Präzedenzen
- Installation / Verwendung der Simulations und Synthesetools

Wiederholung:
Bindung von Operatoren (Präzedenz)

TECHNISCHE UNIVERSITÄT DARMSTADT

`[]`	Zugriff auf Vektorelement (höchste Präzedenz)
`~, !, -, &, ^`	unäre Operatoren: NOT, Negation, Reduktion
`*, /, %`	Multiplikation, Division, Modulo
`+, -`	Addition, Subtraktion
`<<, >>, <<<, >>>`	logischer und arithmetischer Shift
`<, <=, >, >=`	Vergleich
`==, !=`	gleich, ungleich
`&, ~&`	bitweise AND, NAND
`^, ~^`	bitweise XOR, XNOR
`|, ~|`	bitweise OR, NOR
`&&`	logisches AND (Vektoren sind genau dann wahr,
`||`	logisches OR wenn wenigstens ein Bit 1 ist)
`?:`	ternärer Operator
`{}`	Konkatenation (niedrigste Präzedenz)

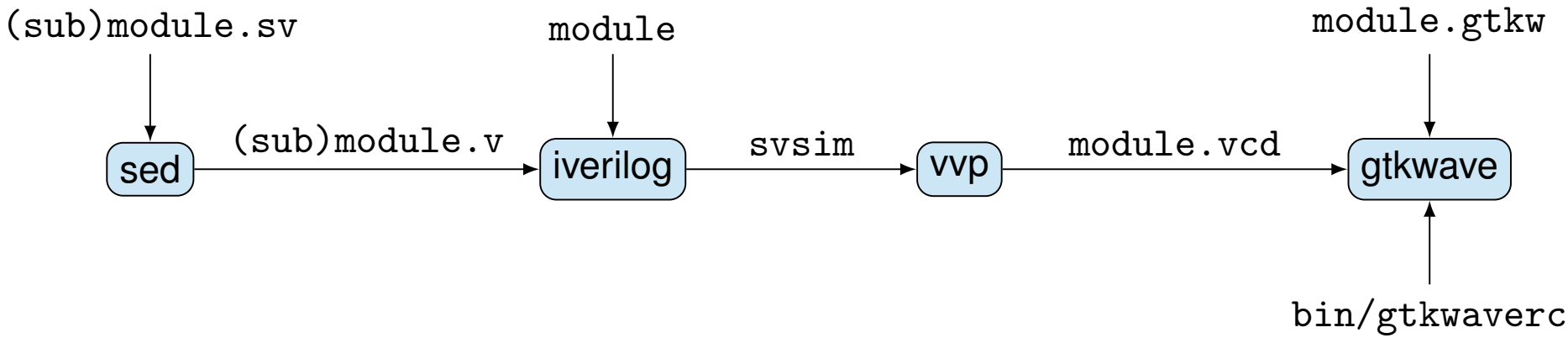
Simulations und Synthesetools

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anleitung im Markdown Format
- Icarus-Verilog Installer setzt PATH unter Windows nicht
- Skripte wurden aktualisiert
- Skripte werden aus Kommandozeile aufgerufen

Simulation von SystemVerilog
Icarus-Verilog + GTKWave

TECHNISCHE UNIVERSITÄT DARMSTADT

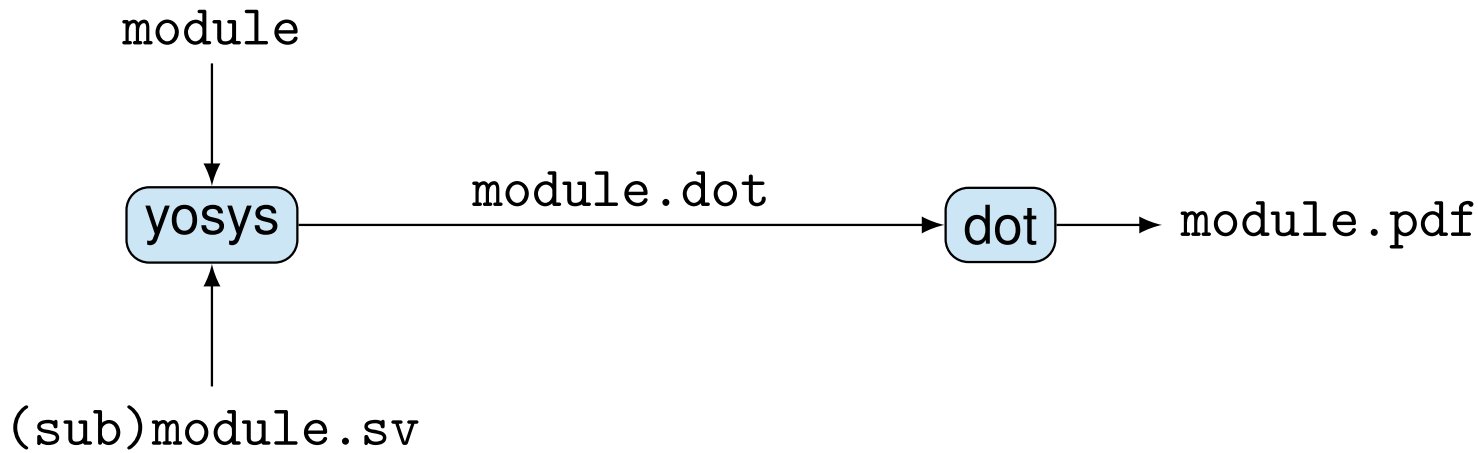



```
> cd sv/examples
> ../bin/sim.sh nand3_tb nand3_tb.sv nand3.sv inv.sv and3.sv
  VCD info: dumpfile nand3_tb.vcd opened for output.
  FINISHED nand3_tb
  GTKWave Analyzer v3.3.86 (w)1999-2017 BSI

  [0] start time.
  [8000] end time.
```

Synthese von SystemVerilog
Yosys + GraphViz

TECHNISCHE UNIVERSITÄT DARMSTADT




```
1 > cd sv/examples
2 > ../bin/synth.sh nand3 nand3.sv inv.sv and3.sv
3   nand3.dot
4   nand3.pdf
```

Überblick der heutigen Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehr SystemVerilog für kombinatorische Logik
- SystemVerilog für sequentielle Logik
- SystemVerilog für Zustandsautomaten
- SystemVerilog für parametrisierte Module
- SystemVerilog für Testumgebungen

Harris 2013
Kap. 4.4-4.8
Seite 190 - 218

Mehr SystemVerilog für kombinatorische Logik

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswahl wichtiger Datentypen

TECHNISCHE UNIVERSITÄT DARMSTADT

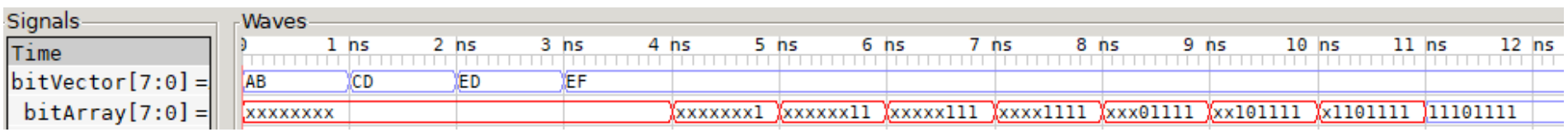
- `bit` = `{1'b0, 1'b1}` (zweiwertige Logik)
- `logic` = `{1'b0, 1'b1, 1'bx, 1'bz}` (vierwertige Logik)
- `int` = `{-2**31, ..., 2**31-1}` = `bit signed [31:0]`
- `integer` = `{-2**31, ..., 2**31-1}` = `logic signed [31:0]`
- `enum` = Aufzählung symbolischer Werte (bspw. für endliche Automaten)
- `time, real, typedef, struct, ...`

Vektoren und Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

vecarr.sv

```
1   // Deklaration
2   logic [7:0] bitVector = 8'hAB; // 8 bit Vektor   [MSB:LSB]
3   logic       bitArray  [0:7];   // 8 bit Array  [first:last]
4
5   // Zugriffe / Modifikation
6   initial begin
7     #1 bitVector      = 8'hCD;   // alle Vektorbits überschreiben
8     #1 bitVector[5]   = 1'b1;    // Vektorbits einzeln überschreiben
9     #1 bitVector[3:0] = 4'hF;    // Vektorbereich überschreiben
10
11    // Arrays-Zugriff nur elementweise möglich
12    for (int i=0; i<$size(bitArray); i++) #1 bitArray[i] = bitVector[i];
13  end
```



Vektoren-Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

vecop.sv

```
1  module vecop(input  logic [3:0] A, input  logic [3:0] B,
2               output logic    U, V, output logic [3:0] W,
3               output logic [1:0] X, output logic [5:0] Y,
4               output logic [7:0] Z);
5
6    // Reduktion
7    assign U = & A;         // U =  A[0] & A[1] & A[2] & A[3]
8
9    // logische Verknüpfung
10   assign V = A && B;      // V = (A[0] | A[1] | A[2] | A[3])
11                           //   & (B[0] | B[1] | B[2] | B[3])
12
13   // bitweise Verknüpfung
14   assign W = A &  B;      // W[0] = (A[0] & B[0]), W[1] = (A[1] & B[1])
15                           // W[2] = (A[2] & B[2]), W[3] = (A[3] & B[3])
16
17   // Konkatenation
18   assign {X,Y} = {A,B}; // X = A[3:2], Y[5:4] = A[1:0], Y[3:0] = B
19
20   // (unsigned) Arithmetik
21   assign Z = A * B;
22
23 endmodule
```

Einschränkungen von Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

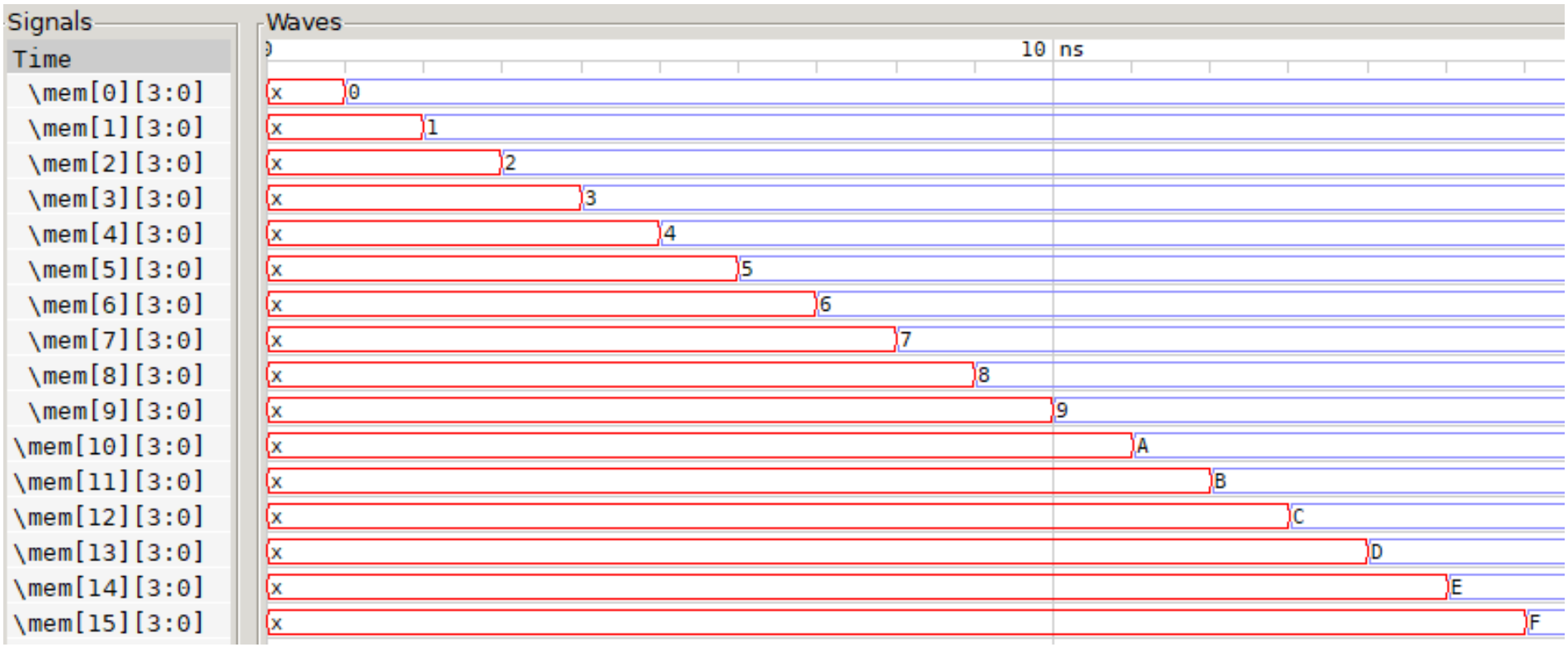
- nicht als Ports verwendbar
- kein „part select“, bspw. `assign bitArray[3:0] = 4'hF;`
- keine Zuweisung ganzer Arrays, bspw. `assign bitArray2 = bitArray;`
- keine Initialisierung bei der Deklaration
- keine Reduktion / Konkatenation
- keine bitweisen / logischen / arithmetischen Operationen

Speicher als Vektor-Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

memory.sv

```
1   //     Breite     Tiefe
2   logic [3:0] mem [0:15];  // 16 Worte zu je 4 bit
3
4   initial for (int i=0; i<$size(mem); i++) #1 mem[i] = i;
```

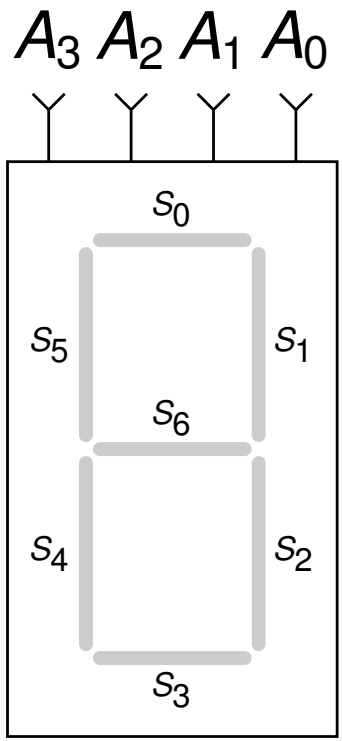


Fallunterscheidungen (case)
Siebensegment-Anzeige

TECHNISCHE UNIVERSITÄT DARMSTADT

sevenseg.sv

```
1  module sevenseg (input  logic [3:0] A,
2                   output logic [6:0] S);
3    always_comb case (A)
4              0: S = 7'b011_1111;
5              1: S = 7'b000_0110;
6              2: S = 7'b101_1011;
7              3: S = 7'b100_1111;
8              4: S = 7'b110_0110;
9              5: S = 7'b110_1101;
10             6: S = 7'b111_1101;
11             7: S = 7'b000_0111;
12             8: S = 7'b111_1111;
13             9: S = 7'b110_1111;
14       default: S = 7'b000_0000;
15   endcase
16 endmodule
```



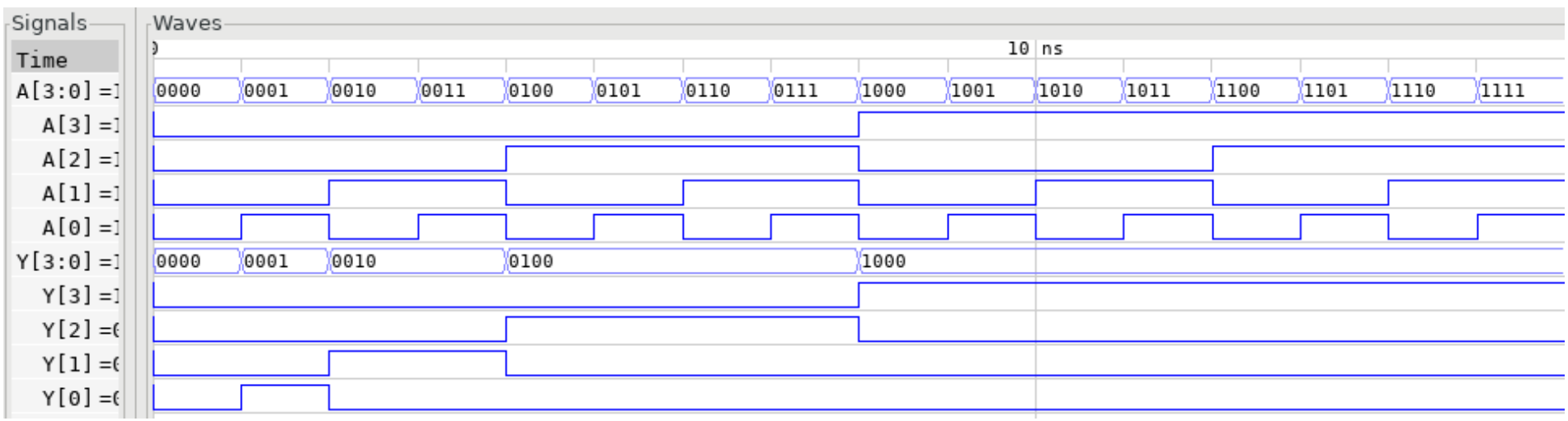
- `case` darf nur in `always` Blöcken verwendet werden
- für kombinatorische Logik müssen alle Eingabe-Optionen abgedeckt werden
- explizit oder per `default` („alle anderen“)

Fallunterscheidungen (casez)
Prioritätsencoder

TECHNISCHE UNIVERSITÄT DARMSTADT

priority_encoder.sv

```
1  module priority_encoder(input  logic [3:0] A,
2                          output logic [3:0] Y);
3    always_comb casez(A)
4      4'b1???: Y = 4'b1000;  // ? = don't care
5      4'b01??: Y = 4'b0100;
6      4'b001?: Y = 4'b0010;
7      4'b0001: Y = 4'b0001;
8      default: Y = 4'b0000;
9    endcase
10 endmodule
```



Eigenschaften von `assign` und `always_comb`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Reihenfolge im Quellcode nicht relevant
  - „nebenläufige Signalzuweisungen“ (concurrent signal assignments)
  - *Achtung:* das gilt nicht (immer) für Signalzuweisungen *innerhalb* von `always_comb` Blöcken
- werden immer ausgeführt, wenn sich ein Signal auf der rechten Seite ändert
  - $\Rightarrow$ interne Zustände, die nicht (transitiv) von aktuellen Eingängen abhängen, können nicht dargestellt werden
  - $\Rightarrow$ für sequentielle Logik ist anderes Sprachkonstrukt notwendig

SystemVerilog für sequentielle Logik

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundkonzept von `always` Blöcken

TECHNISCHE UNIVERSITÄT DARMSTADT

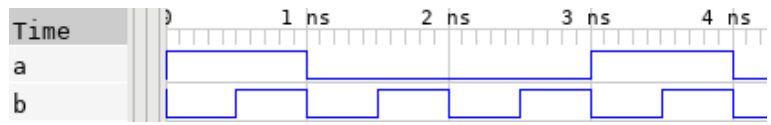
- `always <instruction>` führt eine Instruktion als Endlosschleife aus
- durch Klammerung (`begin end`) werden Instruktionen zusammengefasst
- alle `always` Blöcke werden parallel (nebenläufig) ausgeführt
- ohne explizite Verzögerungsangaben wird die simulierte Systemzeit (abgesehen von „Deltazyklen“) durch die Ausführung nicht erhöht
- `# <tval>` verzögert die Ausführung *des umgebenden* `always` blocks

Delay.java

```
1 boolean a;
2 while (true) {
3   a = true;
4   Thread.sleep(1);
5   a = false;
6   Thread.sleep(2);
7 }
```

delay.sv

```
1  logic a;
2  always begin
3    a = 1;
4    #1;
5    a = 0;
6    #2;
7  end
8
9  logic b=0;
10 always #0.5 b=!b;
```



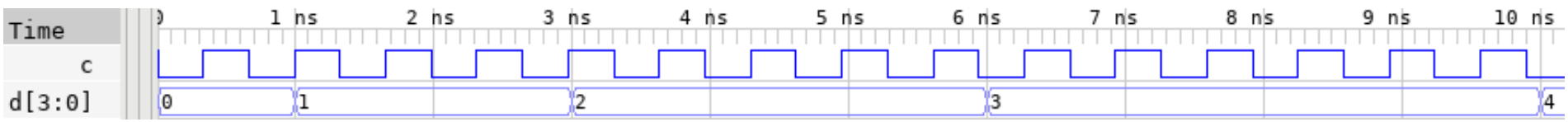
Interpretation von Verzögerungszeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

- `‘timescale <base> / <precision>` vor Modul spezifiziert
  - Zeitbasis (`<base>`), mit der die Verzögerungsangabe (`<tval>`) multipliziert wird
  - Genauigkeit (`<precision>`), auf welche die Verzögerungszeit gerundet wird
- für `<tval>` kann arithmetischer Ausdruck verwendet werden, der auch von variablen Signalen abhängig sein darf

delay.sv

```
1  ‘timescale 1 ns / 10 ps
2
3  module delay;
4    logic c=0;
5    always #(1/3.0) c=!c; // 0.33 ns
6
7    logic [3:0] d=0;
8    always #(d+1) d=d+1;
9  endmodule
```



Warten auf Ereignisse

TECHNISCHE UNIVERSITÄT DARMSTADT

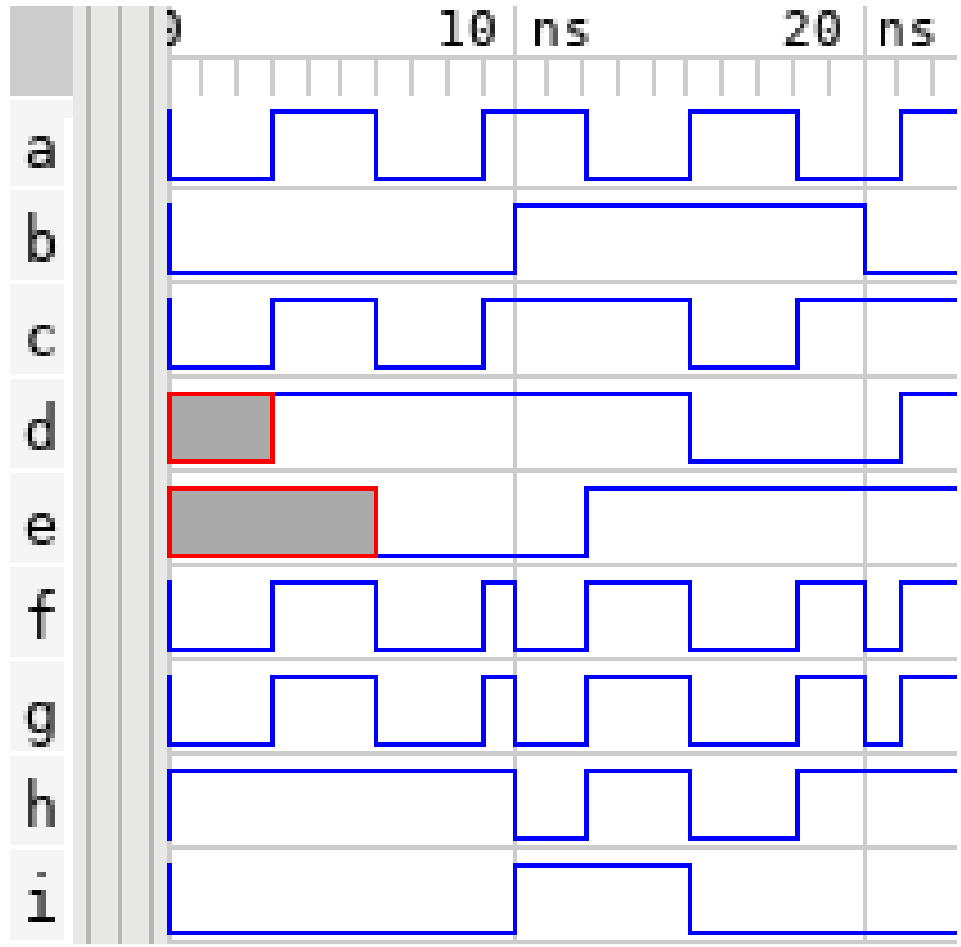
- `@ <expr>` wartet auf Änderung von kombinatorischem Ausdruck `<expr>`
- `@(posedge <expr>)` wartet auf steigende Flanke von `<expr>`
  $(0 \to 1, x \to 1, z \to 1, 0 \to z, 0 \to x)$
- `@(negedge <expr>)` wartet auf fallende Flanke von `<expr>`
  $(1 \to 0, x \to 0, z \to 0, 1 \to z, 1 \to x)$
- `@(<event> or <event>)` wartet auf Eintreten eines der aufgelisteten Ereignisse
  - `or` kann auch durch `,` ersetzt werden
  - wird auch als *Sensitivitätsliste* bezeichnet
- `@*` wartet auf Änderung eines der im `always` Block gelesen Signale
- Warte-Statements können an beliebiger Stelle im `always` Block stehen

Warten auf Ereignisse

TECHNISCHE UNIVERSITÄT DARMSTADT

events.sv

```
1  logic  a=0,b=0;
2  always #3              a=!a;
3  always #10             b=!b;
4
5  logic  c,d,e,f,g;
6  always @a              c=a^b;
7  always @(posedge a)    d=a^b;
8  always @(negedge a)    e=a^b;
9  always @(a,b)          f=a^b;
10 always @*              g=a^b;
11
12 logic  h=0,i=0;
13 always @(a&b)             h=!h;
14 always @(posedge a&b)     i=!i;
```



Zuweisungssequenzen in `always` Blöcken

TECHNISCHE UNIVERSITÄT DARMSTADT

- blockierende Zuweisungen: `<signal> = <expr>;`
  - `<expr>` wird ausgewertet und an `<signal>` zugewiesen, *bevor* nächste Zuweisung behandelt wird

  $\Rightarrow$ blockierende Zuweisungen werden in gegebener Reihenfolge (sequentiell) abgehandelt

- Nicht-blockierende Zuweisungen: `<signal> <= <expr>;`
  - `<expr>` aller nicht-blockierenden Zuweisungen in einer Sequenz werden ausgewertet, aber *noch nicht* an `<signal>` *zugewiesen*
  - Zuweisung an `<signal>` erfolgt erst bei Fortschreiten der Systemzeit (`#` oder `@`)

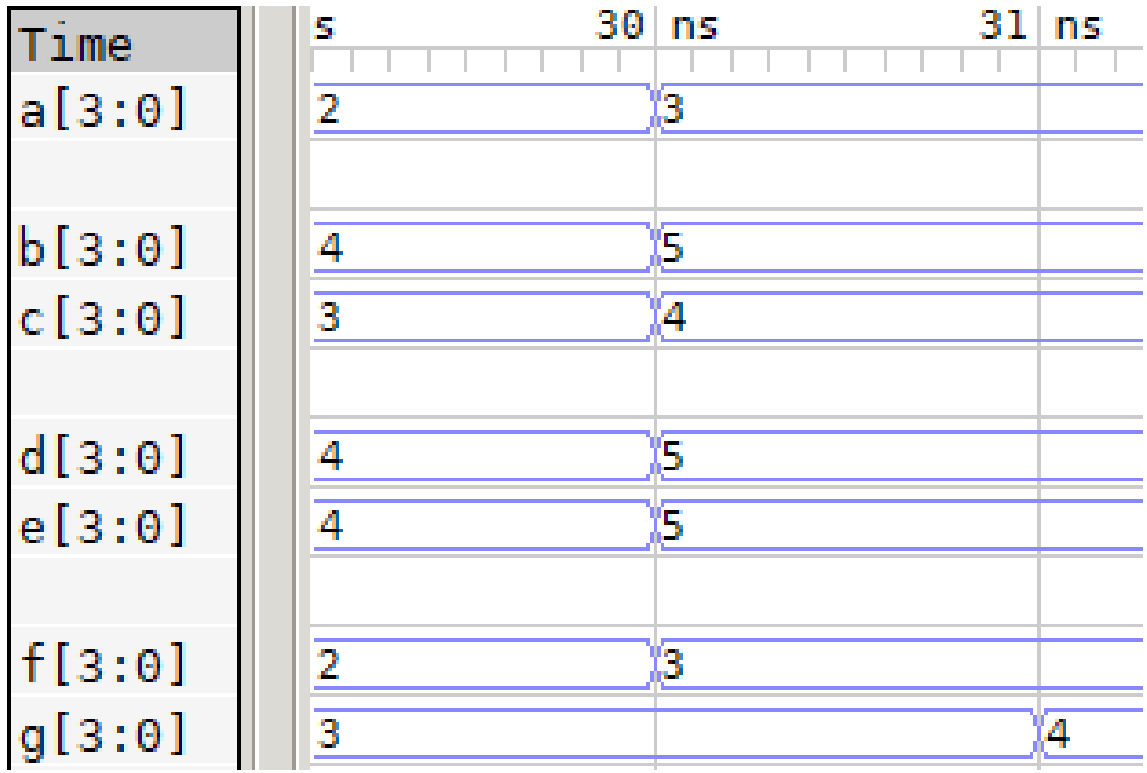
  $\Rightarrow$ nicht-blockierende Zuweisungen werden nebenläufig (parallel) abgehandelt

Zuweisungssequenzen in `always` Blöcken

TECHNISCHE UNIVERSITÄT DARMSTADT

non_blocking.sv

```
1   logic [3:0] a = 0;
2   always #10 a++;
3
4   logic [3:0] b,c,d,
5               e,f,g;
6   always @a begin
7     b <= a+2;
8     c <= b;
9
10    d  = a+2;
11    e  = d;
12
13    f  = c;
14    #1;
15    g  = c;
16  end
```



Einmalige und kombinatorische Ausführung

TECHNISCHE UNIVERSITÄT DARMSTADT

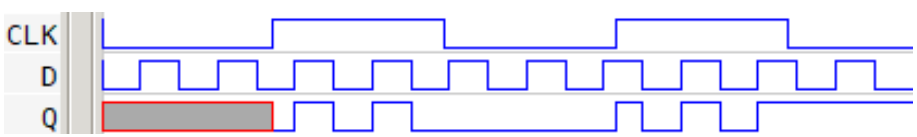
- `initial <instruction>`
  - entspricht `always begin <instruction> @(0); end`
  - ⇒ für Initialisierung in der *Simulation* verwenden
- `always_comb <instruction>`
  - verbessert `always @* <instruction>`
  - einmalige Ausführung zu Beginn der Simulation, auch wenn sich Eingabesignale noch nicht geändert haben
  - Fehlermeldung, wenn selbes Signal aus verschiedenen `always_comb` Blöcken geschrieben werden soll
  - ⇒ für (komplexe) kombinatorische Logik (`for, if else, case, casez`) verwenden
  - *Achtung:* Icarus-Verilog unterstützt `always_comb` (noch) nicht
  - ⇒ wird durch `always @*` ersetzt

Modellierung von Speicherelemente
`always` Blöcke für Latches und Flip-Flops

TECHNISCHE UNIVERSITÄT DARMSTADT

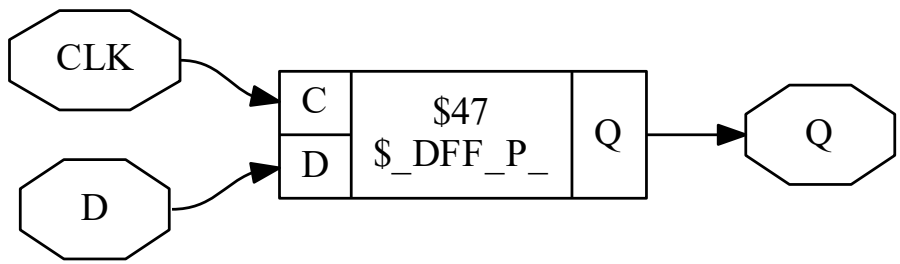
latch.sv

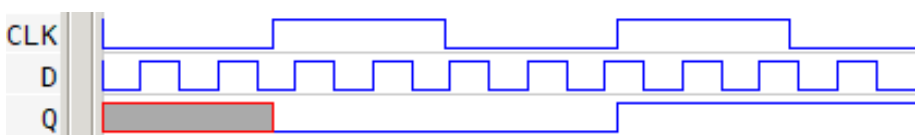
```
1 module latch (input  logic CLK,D,
2                output logic Q);
3
4   always_comb if (CLK) Q <= D;
5
6 endmodule
```



dff.sv

```
1 module dff (input  logic CLK,D,
2              output logic Q);
3
4   always @(posedge CLK) Q <= D;
5
6 endmodule
```





Spezialisierte `always` Blöcke für Speicherelemente

TECHNISCHE UNIVERSITÄT DARMSTADT

- `always_latch <instruction>`
  - entspricht `always_comb <instruction>`
  - *Achtung:* Latches werden in synchronen Schaltungen kaum benutzt

  $\Rightarrow$ idR. durch Fehler in der HDL-Beschreibung verursacht

- `always_ff <instruction>`
  - entspricht `always <instruction>`
  - vergleichbare Verbesserungen wie bei `always_comb`

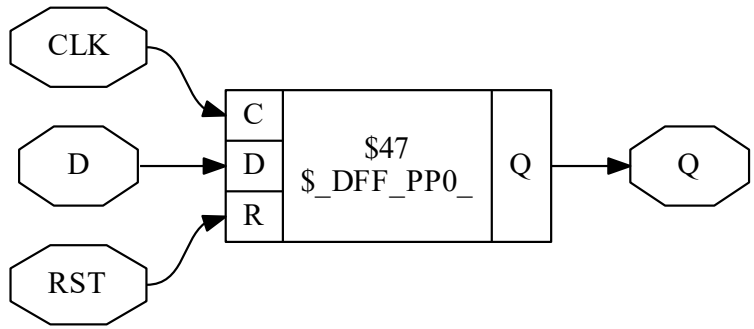
$\Rightarrow$ Synthese-Tools erkennen Absicht des Designers besser und können bei ungeeigneter HDL-Beschreibung warnen

Rücksetzbare Flip-Flops

TECHNISCHE UNIVERSITÄT DARMSTADT

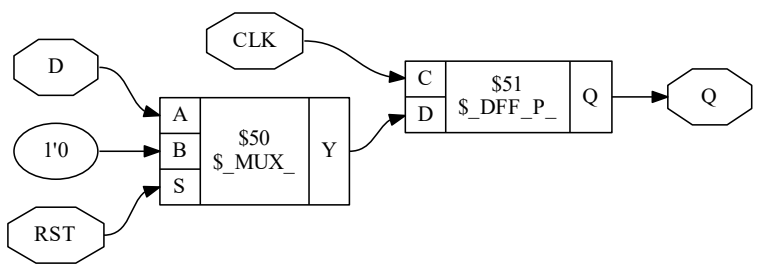
dffar.sv

```
1  // asynchron rücksetzbar
2  module dffar (input  logic CLK,RST,D,
3                output logic Q);
4
5    always_ff @(posedge CLK, posedge RST)
6      if (RST) Q <= 0;
7      else     Q <= D;
8
9  endmodule
```



dffr.sv

```
1  // synchron rücksetzbar
2  module dffr (input  logic CLK,RST,D,
3               output logic Q);
4
5    always_ff @(posedge CLK)
6      if (RST) Q <= 0;
7      else     Q <= D;
8
9  endmodule
```

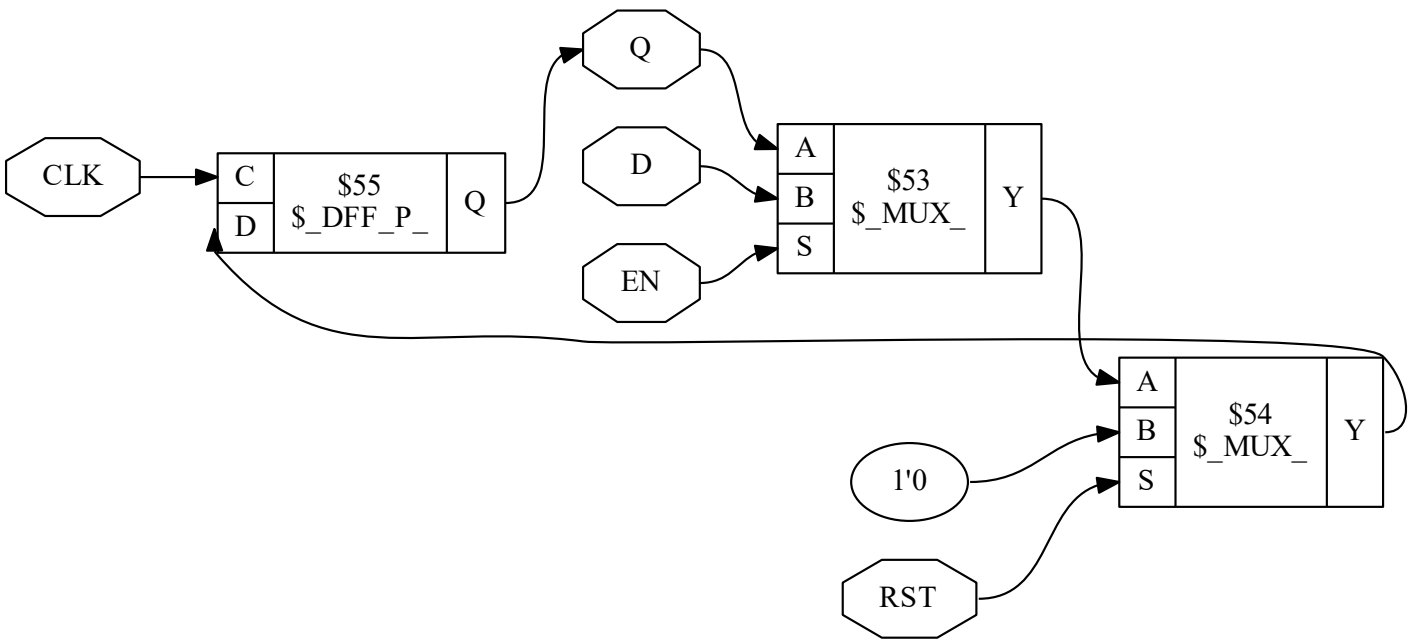


Flip-Flop mit Taktfreigabe

TECHNISCHE UNIVERSITÄT DARMSTADT

dffe.sv

```
1  module dffe (input logic CLK,RST,EN,D, output logic Q);
2
3    always_ff @(posedge CLK)
4      if      (RST) Q <= 0;
5      else if (EN) Q <= D;
6
7  endmodule
```



Allgemeine Regeln für Signalzuweisungen (synchrone sequentielle Logik)

TECHNISCHE UNIVERSITÄT DARMSTADT

- interne Zustände
  - innerhalb von `always_ff @(posedge CLK)`
  - mit nicht-blockierende Zuweisungen
  - *möglicht* nur ein/wenige Zustände pro `always` block
- einfache kombinatorische Logik durch nebenläufige Zuweisungen (`assign`)
- komplexere kombinatorische Logik:
  - innerhalb von `always_comb`
  - mit blockierenden Zuweisungen
- ein Signal darf nicht
  - von mehreren nebenläufigen Prozessen (`assign` oder `always`) beschrieben werden
  - innerhalb eines `always` Blocks mit blockierenden und nicht-blockierenden Zuweisungen beschrieben werden

SystemVerilog für Zustandsautomaten

TECHNISCHE UNIVERSITÄT DARMSTADT

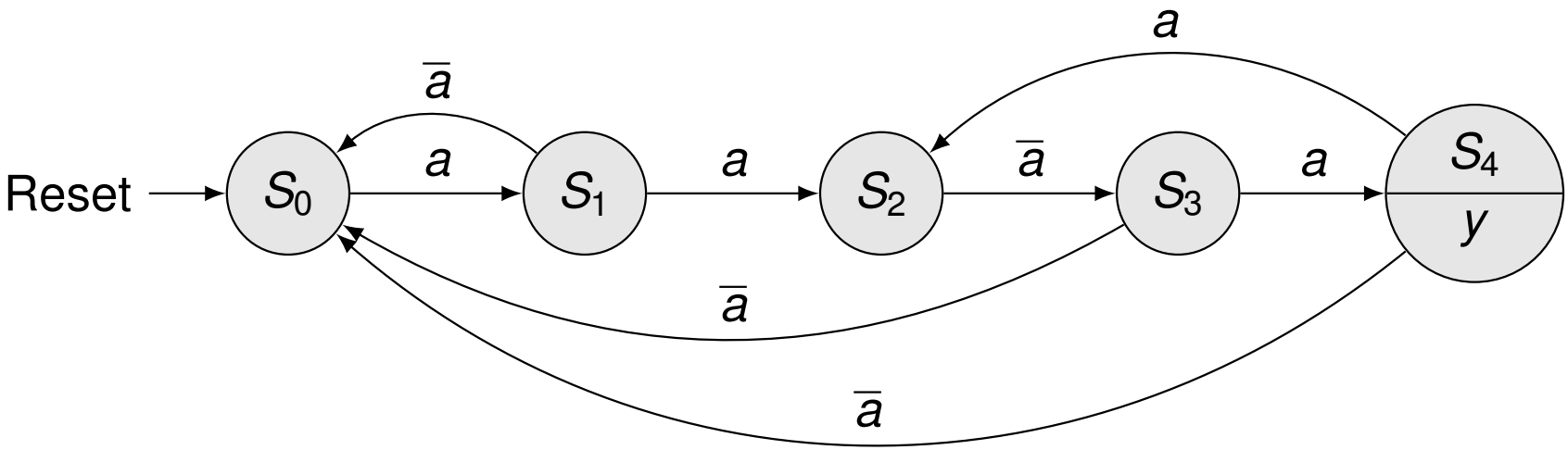
Grundidee für FSM-Modellierung

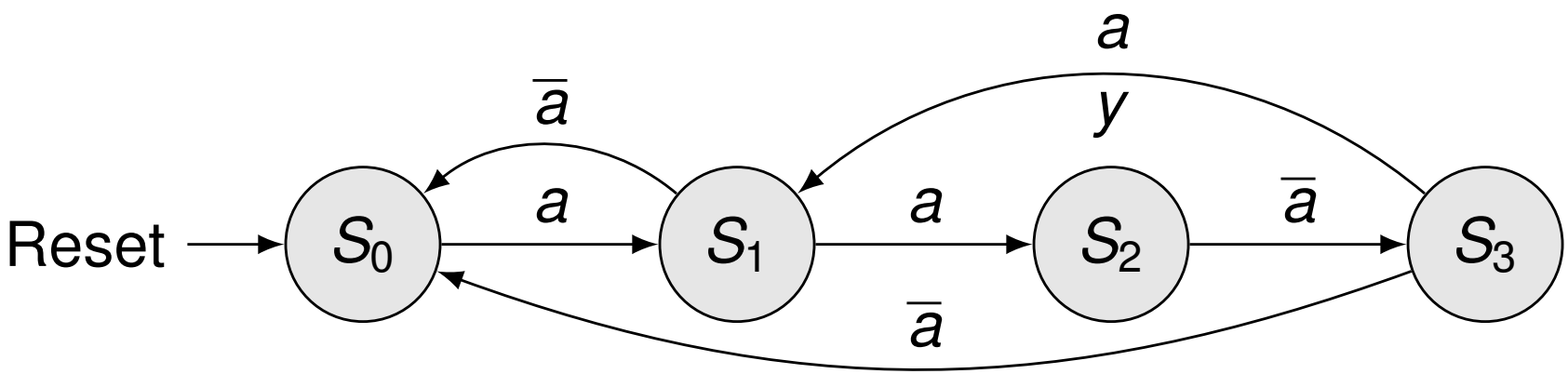
TECHNISCHE UNIVERSITÄT DARMSTADT

- Logikvektor oder `enum` für Zustände
- rücksetzbares Flip-Flop als Zustandsspeicher
- kombinatorische next-state Logik per `case` in `always_comb`
- kombinatorische Ausgabe-Logik per nebenläufiger Zuweisungen

Moore- und Mealy-Automat für 1101 Mustererkennung

TECHNISCHE UNIVERSITÄT DARMSTADT



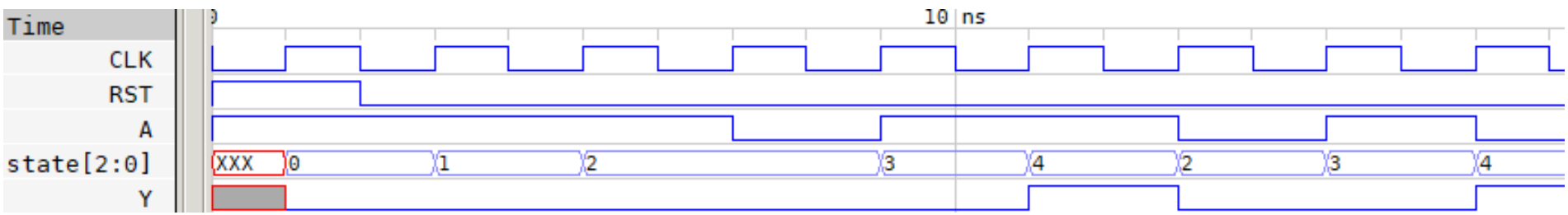


Moore FSM für 1101 Mustererkennung

TECHNISCHE UNIVERSITÄT DARMSTADT

pattern/moore.sv

```
1  module moore (input logic CLK, RST, A, output logic Y);
2
3    logic [2:0] state, nextstate;
4    always_ff @(posedge CLK) state <= RST ? 0 : nextstate;
5
6    // next state logic
7    always_comb case (state)
8      0:       nextstate = A ? 1 : 0;
9      1:       nextstate = A ? 2 : 0;
10     2:       nextstate = A ? 2 : 3;
11     3:       nextstate = A ? 4 : 0;
12     default: nextstate = A ? 2 : 4;
13   endcase
14
15   // output logic
16   assign Y = state == 4;
17 endmodule
```

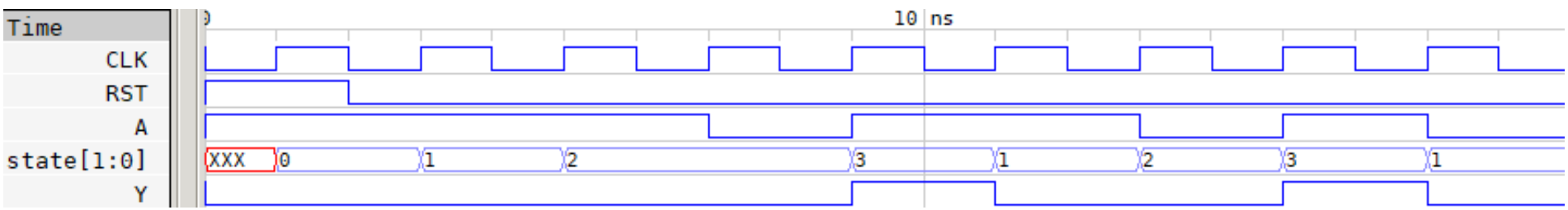


Mealy FSM für 1101 Mustererkennung

TECHNISCHE UNIVERSITÄT DARMSTADT

pattern/mealy.sv

```
1  module mealy (input logic CLK, RST, A, output logic Y);
2
3    logic [1:0] state, nextstate;
4    always_ff @(posedge CLK) state <= RST ? 0 : nextstate;
5
6    // next state logic
7    always_comb case (state)
8      0:       nextstate = A ? 1 : 0;
9      1:       nextstate = A ? 2 : 0;
10     2:       nextstate = A ? 2 : 3;
11     default: nextstate = A ? 1 : 0;
12   endcase
13
14   // output logic
15   assign Y = state == 3 && A;
16 endmodule
```



SystemVerilog für parametrisierte Module

TECHNISCHE UNIVERSITÄT DARMSTADT

Parametrisierte Module

TECHNISCHE UNIVERSITÄT DARMSTADT

- neben Ein- und Ausgaben kann Modulschnittstelle auch `parameter` definieren
- parametrisierte Eigenschaften werden bei Instanziierung durch konkrete Werte ersetzt
  - zur Laufzeit nicht änderbar
  - vergleichbar mit C-Präprozessor oder Java-Generics
- typische Parameter: Port-Breite, Speichertiefe, ...

mux.sv

```
1  module mux
2   #(parameter WIDTH=8)
3    (input  logic [WIDTH-1:0] A,B,
4     input  logic S,
5     output logic [WIDTH-1:0] Y);
6
7    assign Y = S ? A : B;
8
9  endmodule
```

mux_tb.sv

```
1  module mux_tb;
2
3    localparam W=4;
4
5    logic [W-1:0] a=4,b=3,y;
6    logic s;
7
8    mux #(W) uut (a,b,s,y);
9
10 endmodule
```

Iterative / Optionale Instantiierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anzahl von Submodulen hängt oft von Parameter ab

⇒ generische Instanziierung mit `if else` und `for` notwendig

shift_reg.sv

```
1  module shift_reg #(parameter WIDTH=8,
2                     parameter DEPTH=32)
3                    (input  logic CLK, RST,
4                     input  logic [WIDTH-1:0] D,
5                     output logic [WIDTH-1:0] Q);
6
7    logic [WIDTH-1:0] c [0:DEPTH];
8    assign c[0] = D;
9    assign Q    = c[DEPTH];
10
11   genvar i; // für Schleife im generate-Block
12   generate  // für SystemVerilog optional
13     for (i=0; i<DEPTH; i=i+1) begin
14       register #(WIDTH) r (.CLK(CLK), .RST(RST), .D(c[i]), .Q(c[i+1]));
15     end
16   endgenerate
17 endmodule
```

SystemVerilog für Testumgebungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Testumgebungen

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 werden 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

silly/function.sv

```
1 module sillyfunction(input  logic a, b, c,
2                       output logic y);
3
4   assign y = ~b & ~c | a & ~b;
5
6 endmodule
```

Einfacher Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

silly/tb.sv

```
1  module tb;
2    logic a, b, c, y;
3    sillyfunction uut(a, b, c, y);
4
5    initial begin
6      $dumpfile("tb.vcd"); // iverilog spezifisch
7      $dumpvars;
8
9      a = 0; b = 0; c = 0; #10;
10                   c = 1; #10;
11           b = 1; c = 0; #10;
12                   c = 1; #10;
13
14     $display("FINISHED␣tb");
15     $finish;
16   end
17 endmodule
```

Selbstprüfender Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

silly/tb2.sv

```
1  module tb2;
2    logic a, b, c, y;
3    sillyfunction uut(a, b, c, y);
4
5    initial begin
6      $dumpfile("tb2.vcd"); // iverilog spezifisch
7      $dumpvars;
8
9      a = 0; b = 0; c = 0; #10; if (y!=1) $display("000␣failed.");
10                   c = 1; #10; if (y!=0) $display("001␣failed.");
11             b = 1; c = 0; #10; if (y!=0) $display("010␣failed.");
12                   c = 1; #10; if (y!=0) $display("011␣failed.");
13
14     $display("FINISHED␣tb2");
15     $finish;
16   end
17 endmodule
```

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfassung und Ausblick

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehr SystemVerilog für kombinatorische Logik
- SystemVerilog für sequentielle Logik
- SystemVerilog für Zustandsautomaten
- SystemVerilog für parametrisierte Module
- SystemVerilog für Testumgebungen

- Nächste Vorlesung behandelt
  - Mehr zu Testumgebungen
  - Arithmetische Grundschaltungen