

Digitaltechnik

Wintersemester 2021/2022

11. Vorlesung












TECHNISCHE
UNIVERSITÄT
DARMSTADT





Umfrage zur letzten Woche

1. SystemVerilog Datentypen
2. SystemVerilog für kombinatorische Logik (Forts.)
3. SystemVerilog für sequentielle Logik
4. SystemVerilog für parametrisierte Module
5. SystemVerilog für Testumgebungen
6. Zusammenfassung

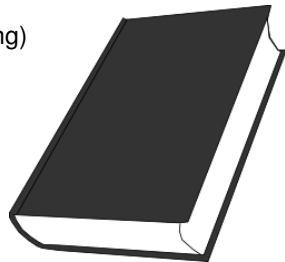
Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

Überblick der heutigen Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Auswahl wichtiger Datentypen
- ▶ SystemVerilog für kombinatorische Logik (Fortsetzung)
- ▶ SystemVerilog für sequentielle Logik
- ▶ SystemVerilog für parametrisierte Module
- ▶ SystemVerilog für Testumgebungen



Harris 2013/2016
Kap. 4.4, 4.5, 4.7 - 4.9

Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. SystemVerilog Datentypen










2. SystemVerilog für kombinatorische Logik (Forts.)

3. SystemVerilog für sequentielle Logik

4. SystemVerilog für parametrisierte Module

5. SystemVerilog für Testumgebungen

6. Zusammenfassung

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

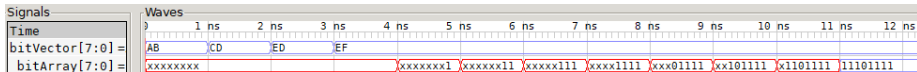
Vektoren und Arrays



TECHNISCHE
UNIVERSITÄT
DARMSTADT

comb/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     // Array-Zugriff nur elementweise möglich
12     for (int i=0; i<$size(bitArray); i++) #1 bitArray[i] = bitVector[i];
13 end
```



Operationen auf Vektoren



comb/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
```




- ▶ nicht als Ports verwendbar
- ▶ nicht mit assign verwendbar
 - ▶ kein “part select” (`assign bitArray[3:0] = 4'hF;`)
 - ▶ keine Zuweisung ganzer Arrays (`assign bitArray2 = bitArray;`)
- ▶ keine Reduktion/Konkatenation
- ▶ keine bitweisen/logischen/arithmetischen Operationen

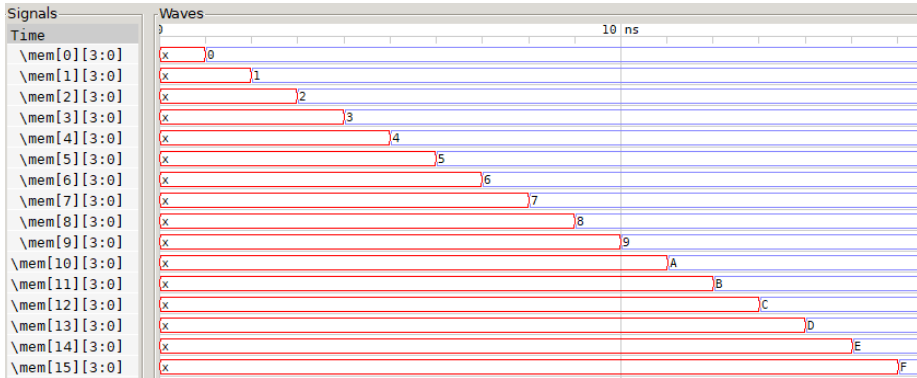
Speicher als Vektor-Array



TECHNISCHE
UNIVERSITÄT
DARMSTADT

seq/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;
```



Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. SystemVerilog Datentypen










2. SystemVerilog für kombinatorische Logik (Forts.)

3. SystemVerilog für sequentielle Logik

4. SystemVerilog für parametrisierte Module

5. SystemVerilog für Testumgebungen

6. Zusammenfassung

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

Wiederholung: Assign Statement



comb/example.sv

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

- ▶ auch *continuous assignment* genannt
- ▶ Linke Seite (LHS, “left hand side”): Variable oder Port
- ▶ Rechte Seite (RHS, “right hand side”): logischer Ausdruck
- ▶ Zuweisung, wenn der Wert von RHS sich ändert



- ▶ `always_comb <instruction>`
 - ▶ zum Zeitpunkt 0, nachdem alle `initial` und `always` Blöcke gestartet sind
 - ▶ und immer wenn der Wert von RHS sich ändert
- ▶ LHS Variablen dürfen nicht von anderen Blöcken geschrieben werden

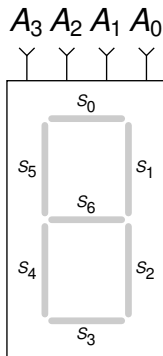
Fallunterscheidungen (case)

Beispiel: Dezimale 7-Segment Anzeige



comb/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/always_comb Blöcken verwendet werden
- ▶ für kombinatorische Logik müssen alle Eingabe-Optionen abgedeckt werden
- ▶ explizit oder per default ("alle anderen")

Fallunterscheidungen (casez)

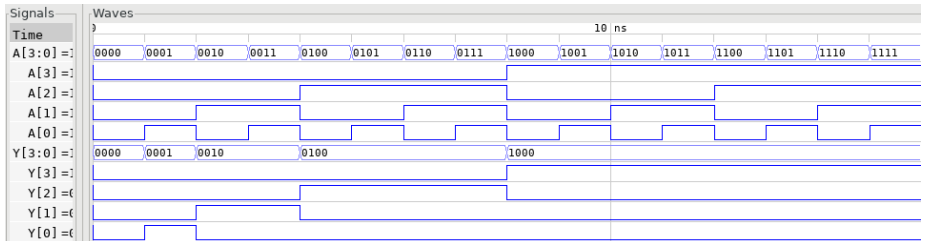
Beispiel: Prioritätsencoder



TECHNISCHE
UNIVERSITÄT
DARMSTADT

comb/priority_encoder.sv

```
1 module priority_encoder(input logic [3:0] A,  
2                       output logic [3:0] Y);  
3     always_comb casez(A) // casez erlaubt don't cares  
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
```














- ▶ werden immer ausgeführt, wenn sich ein Signal auf der rechten Seite ändert
 - ⇒ interne Zustände, die nicht (transitiv) von aktuellen Eingängen abhängen, können nicht dargestellt werden
 - ⇒ für sequentielle Logik ist anderes Sprachkonstrukt notwendig
- ▶ Reihenfolge im Quellcode nicht relevant
 - ▶ nebenläufige Signalzuweisungen (“concurrent signal assignments”)
 - ▶ *Achtung:* Blockierende Signalzuweisungen ($a = b$) *innerhalb* von Blöcken (`begin/end`) werden seriell ausgeführt.

Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. SystemVerilog Datentypen
2. SystemVerilog für kombinatorische Logik (Forts.)
3. SystemVerilog für sequentielle Logik
4. SystemVerilog für parametrisierte Module
5. SystemVerilog für Testumgebungen
6. Zusammenfassung

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

Grundkonzept von `always` Blöcken



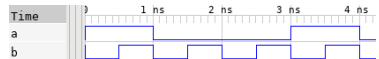
- ▶ `always <instruction>` führt eine Instruktion als Endlosschleife aus
- ▶ durch Klammerung (`begin ... end`) werden Instruktionen zusammengefasst
- ▶ alle `always` Blöcke werden parallel (nebenläufig) ausgeführt
- ▶ ohne explizite Verzögerungsangaben wird die simulierte Systemzeit durch die Ausführung nicht erhöht (nur "Deltazyklen", s. später)
- ▶ `# <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  }
```

seq/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

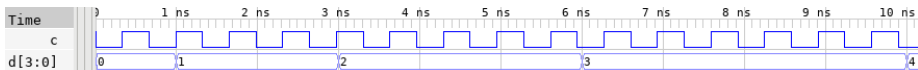


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ ``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

seq/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
```





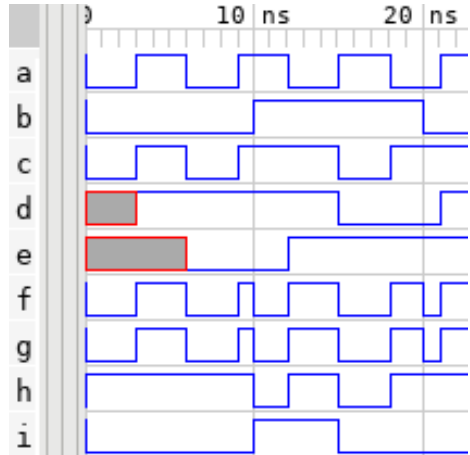
- ▶ @ <expr> wartet auf Änderung von kombinatorischem Ausdruck <expr>
- ▶ @(posedge <expr>) wartet auf steigende Flanke von <expr>
($0 \rightarrow 1, x \rightarrow 1, z \rightarrow 1, 0 \rightarrow z, 0 \rightarrow x$)
- ▶ @(negedge <expr>) wartet auf fallende Flanke von <expr>
($1 \rightarrow 0, x \rightarrow 0, z \rightarrow 0, 1 \rightarrow z, 1 \rightarrow x$)
- ▶ @(<event1> or <event2>) 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



seq/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;
```





- ▶ blockierende Zuweisungen: `<signal> = <expr>;`
 - ▶ `<expr>` wird ausgewertet und an `<signal>` zugewiesen, *bevor* nächste Zuweisung behandelt wird
 - ⇒ 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, sondern nur vorgemerkt
 - ▶ Erst bei Fortschreiten der Systemzeit (# oder @) erfolgt Zuweisung an `<signal>` (Abarbeitung der Vormerkungen als “Deltazyklen”)
 - ⇒ nicht-blockierende Zuweisungen werden nebenläufig (parallel) abgehandelt

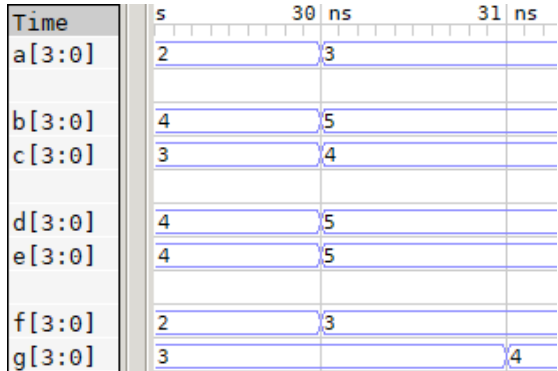
Zuweisungssequenzen in always Blöcken



TECHNISCHE
UNIVERSITÄT
DARMSTADT

seq/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
```





- ▶ `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 Speicherelementen

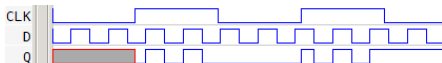
always Blöcke für Latches und Flip-Flops



TECHNISCHE
UNIVERSITÄT
DARMSTADT

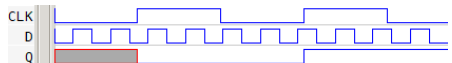
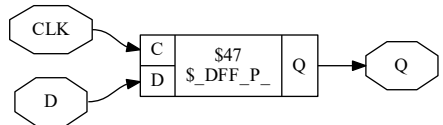
seq/latch.sv

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



seq/dff.sv

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



Modellierung von Speicherelementen

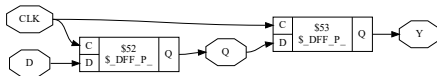
Blockierende `always_ff` Blöcke



TECHNISCHE
UNIVERSITÄT
DARMSTADT

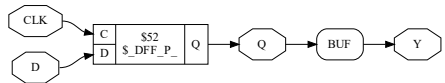
seq/non_blocking_ff.sv

```
1 module non_blocking_ff(  
2     input logic CLK,D,  
3     output logic Q,Y);  
4  
5     always_ff @(posedge CLK) begin  
6         Q <= D; // non-blocking  
7         Y <= Q; // non-blocking  
8     end  
9 endmodule
```



seq/blocking_ff.sv

```
1 module blocking_ff(  
2     input logic CLK,D,  
3     output logic Q,Y);  
4  
5     always_ff @(posedge CLK) begin  
6         Q = D; // blocking  
7         Y = Q; // blocking  
8     end  
9 endmodule
```



- **Empfehlung:** Verwenden Sie *keine* blockierenden Zuweisungen in `always_ff` und `always_latch` Blöcken!

Spezialisierte `always` Blöcke für Speicherelemente



- ▶ `always_latch` `<instruction>`

- ▶ für Schaltungen mit Latches
 - ▶ entspricht `always_comb` `<instruction>`
 - ▶ *Achtung:* Latches werden in synchronen Schaltungen kaum benutzt
- ⇒ i.d.R. durch Fehler in der HDL-Beschreibung verursacht

- ▶ `always_ff` `<instruction>`

- ▶ für Schaltungen mit Flip-Flops
- ▶ entspricht `always` `<instruction>`
- ▶ vergleichbare Verbesserungen wie bei `always_comb`

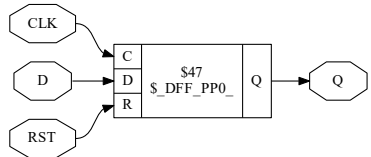
⇒ Synthese-Tools erkennen Absicht des Designers besser und können bei ungeeigneter HDL-Beschreibung warnen

Rücksetzbare Flip-Flops



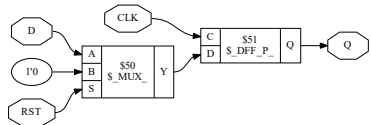
seq/dfar.sv

```
1 // asynchron rücksetzbar
2 module dfar (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
```



seq/dfrr.sv

```
1 // synchron rücksetzbar
2 module dfrr (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
```

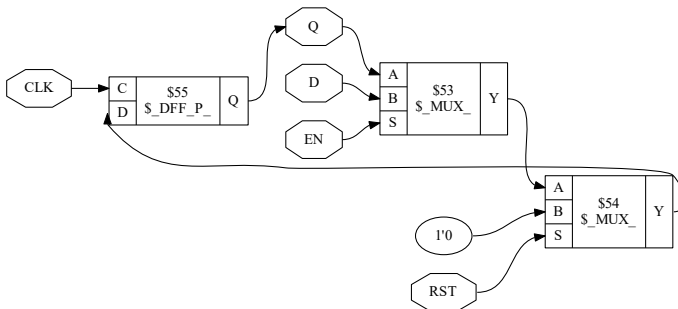


Flip-Flop mit Taktfreigabe



seq/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-blockierenden Zuweisungen (`<=`)
 - ▶ *möglichst* nur ein/wenige Zustände pro `always_ff` 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












Pause & Umfrage bis hier

Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. SystemVerilog Datentypen
2. SystemVerilog für kombinatorische Logik (Forts.)
3. SystemVerilog für sequentielle Logik
4. SystemVerilog für parametrisierte Module
5. SystemVerilog für Testumgebungen
6. Zusammenfassung

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

- ▶ 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, ...

comb/mux2xW.sv

```
1 module mux2xW
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
```

comb/mux2xW_tb.sv

```
1 module mux2xW_testbench;
2
3   localparam W=4;
4
5   logic [W-1:0] a=4,b=3,y;
6   logic s;
7
8   mux2xW #(W) uut(a,b,s,y);
9
10 endmodule
```

Beispiel: Register



seq/register.sv

```
1 module register #(parameter WIDTH=8)
2     (input  logic CLK, RST,
3      input  logic [WIDTH-1:0] D,
4      output logic [WIDTH-1:0] Q);
5
6     always_ff @(posedge CLK) Q <= RST ? 0 : D;
7
8 endmodule
```

- Anzahl von Submodulen hängt oft von Parameter ab

seq/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
```

Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. SystemVerilog Datentypen
2. SystemVerilog für kombinatorische Logik (Forts.)
3. SystemVerilog für sequentielle Logik
4. SystemVerilog für parametrisierte Module
5. SystemVerilog für Testumgebungen
6. Zusammenfassung

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen



- ▶ 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: Testdaten an uut anlegen und Ausgaben anzeigen
 - ▶ selbstprüfend: Ausgaben zusätzlich auf Korrektheit prüfen
 - ▶ selbstprüfend mit Testvektoren: variable Testdaten (bspw. aus Datei lesen), s. Kapitel 4.9 in Harris 2013/2016

Beispiel



comb/simple.sv

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

comb/simple_tb.sv

```
1  module simple_tb;
2      logic a, b, c, y;
3      simple uut(a, b, c, y);
4
5      initial begin
6          // dump changes of all variables to this file
7          $dumpfile("simple_tb.vcd");
8          $dumpvars;
9
10         a = 0; b = 0; c = 0; #10;
11             c = 1; #10;
12         b = 1; c = 0; #10;
13             c = 1; #10;
14
15         $display("FINISHED simple_tb"); // Text ausgeben
16         $finish; // beendet Simulation
17     end
```



comb/simple_tb2.sv

```
1  module simple_tb2;
2      logic a, b, c, y;
3      simple uut(a, b, c, y);
4
5      initial begin
6          $dumpfile("simple_tb2.vcd");
7          $dumpvars;
8          // === testet auf logische Gleichheit (0,1,X,Z)
9          a=0;b=0;c=0; #10; assert(y==1) else $error("000 failed.");
10             c=1; #10; assert(y==0) else $error("001 failed.");
11             b=1;c=0; #10; assert(y==0) else $error("010 failed.");
12             c=1; #10; assert(y==0) else $error("011 failed.");
13
14             $display("FINISHED simple_tb2");
15             $finish;
16     end
17 endmodule
```


Testumgebungen (testbenches)



- ▶ Modul ohne Ports
- ▶ Stimuli erzeugen (Takt, Reset, Eingabedaten)
- ▶ “unit under test” instantiieren
- ▶ Ausgabedaten und Timing verifizieren
 - ▶ erschöpfend oder zufällig
 - ▶ Grenzfälle abdecken
- ▶ wird nicht synthetisiert
- ▶ speziell für Icarus-Verilog
 - ▶ VCD-Datei öffnen
 - ▶ beobachtete Signale konfigurieren
 - ▶ Simulation beenden

```
1 `timescale 1 ns / 10 ps
2 module sub_tb;
3     logic clk = 0, reset = 1;
4     always #(0.5/10)      clk <= ~clk;
5     initial @(posedge clk) reset <= 0;
6
7     logic [3:0] a,b,y;
8     sub uut(clk,reset,a,b,y);
9
10    initial begin
11        $dumpfile("sub_tb.vcd");
12        $dumpvars;
13
14        for (int i=0; i<256; i++) begin
15            {a,b} <= i;
16            @(posedge clk);
17            if (y != a-b) $display("error");
18        end
19
20        $display("FINISHED sub_tb");
21        $finish;
22    end
23 endmodule
```

- ▶ `$display(<format>, <values>*)`;
- ▶ ähnlich `printf` in C und Java
- ▶ wichtige Platzhalter:
 - ▶ `%d %b %h` für dezimal, binär, hexadezimal
 - ▶ `%m` für Modulname (implizites Argument), bspw. `add_tb.uut`
 - ▶ `%t` für Zeit (mit Einheit)
- ▶ `$timeformat(-9, 1, "ns", 8)`; zum Einstellen des Zeitformats
 - ▶ Skalierung auf 10^{-9}
 - ▶ eine Nachkommastelle
 - ▶ Einheiten-Suffix
 - ▶ Anzahl der anzuzeigenden Zeichen
- ▶ bspw.: `$display("%t%m: y = %d", $time, y)`;
erzeugt: 3.0 ns@add_tb: y = 5

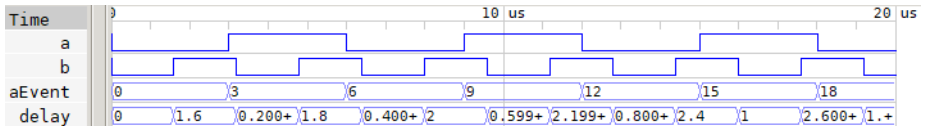
Auslesen der Simulationszeit



- ▶ \$time - aktuelle Systemzeit als ganze Zahl (int)
- ▶ \$realtime - aktuelle Systemzeit als rationale Zahl (real)
- ▶ Anwendungsbeispiel: Zeitspanne zwischen zwei Signalfanken bestimmen

seq/deltat.sv

```
1 `timescale 1 us / 10 ns
2 module deltat;
3     logic a=0;    always #3    a <= ~a;
4     logic b=0;    always #1.6  b <= ~b;
5
6     real aEvent;  always @a    aEvent <= $realtime;
7     real delay;   always @b    delay <= $realtime - aEvent;
8 endmodule
```



- ▶ Erstellen effizienter Testpläne ist nicht trivial
 - ▶ Abdeckung maximieren (gezielt vs. zufällig)
 - ▶ Wiederverwendbarkeit maximieren
 - ▶ Überlappung minimieren
 - ▶ Multi-Domänen Cosimulation von Hardware und
 - ▶ Software
 - ▶ Event-basierten Kommunikationsprotokollen
 - ▶ kontinuierlichen physikalischen Prozessen
 - ▶ Testgetriebene Entwicklung (TDD)
- ⇒ SystemVerilog bringt hier viele Verbesserungen
- ▶ file IO
 - ▶ assertions, implications
 - ▶ (constrained) random
 - ▶ Klassen, Vererbung, Schnittstellen
 - ▶ Direct Programming Interface (DPI) zu C, C++, SystemC, etc.












Chris Spear:
SystemVerilog
for Verification
(Springer)

Agenda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. SystemVerilog Datentypen
2. SystemVerilog für kombinatorische Logik (Forts.)
3. SystemVerilog für sequentielle Logik
4. SystemVerilog für parametrisierte Module
5. SystemVerilog für Testumgebungen
6. Zusammenfassung

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen



- ▶ SystemVerilog Datentypen
- ▶ SystemVerilog für kombinatorische Logik (Fortsetzung)
- ▶ SystemVerilog für sequentielle Logik
- ▶ SystemVerilog für parametrisierte Module
- ▶ SystemVerilog für Testumgebungen

- ▶ Nächste Vorlesung behandelt
 - ▶ SystemVerilog für Zustandsautomaten
 - ▶ SystemVerilog Abschluss und Ausblick
 - ▶ Sequentielle Grundelemente
 - ▶ Speicherfelder



► Ziel / Nutzen

- mittel-/langfristige Verbesserung der Lehre
 - Diskussionsgrundlage für Kontrollgremien des FB 20
 - Bewertungsgrundlage für Vergabe vom “Preis für gute Lehre” des FB 20
- ⇒ kommt Studierenden und Lehrenden zugute

► Ablauf

- anonymisierte Online-Fragebögen
 - Link und persönliche TANs in Moodle verfügbar
 - Vorlesung und Übung werden getrennt evaluiert
- ⇒ Online-Fragebögen mit zwei unterschiedlichen TANs öffnen
- bitte bis **30.01.2022** ausfüllen!