Digitaltechnik
Wintersemester 2017/2018
11. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Andreas Engel, Raad Bahmani **LÖSUNGSVORSCHLAG** **KW03**

Die Präsenzübungen werden in Kleingruppen während der wöchentlichen Übungsstunde bearbeitet. Bei Fragen hilft Ihnen Ihr Tutor gerne weiter. Mit der angegebenen Bearbeitungszeit für die einzelnen Aufgaben können Sie Ihren Leistungsstand besser einschätzen.

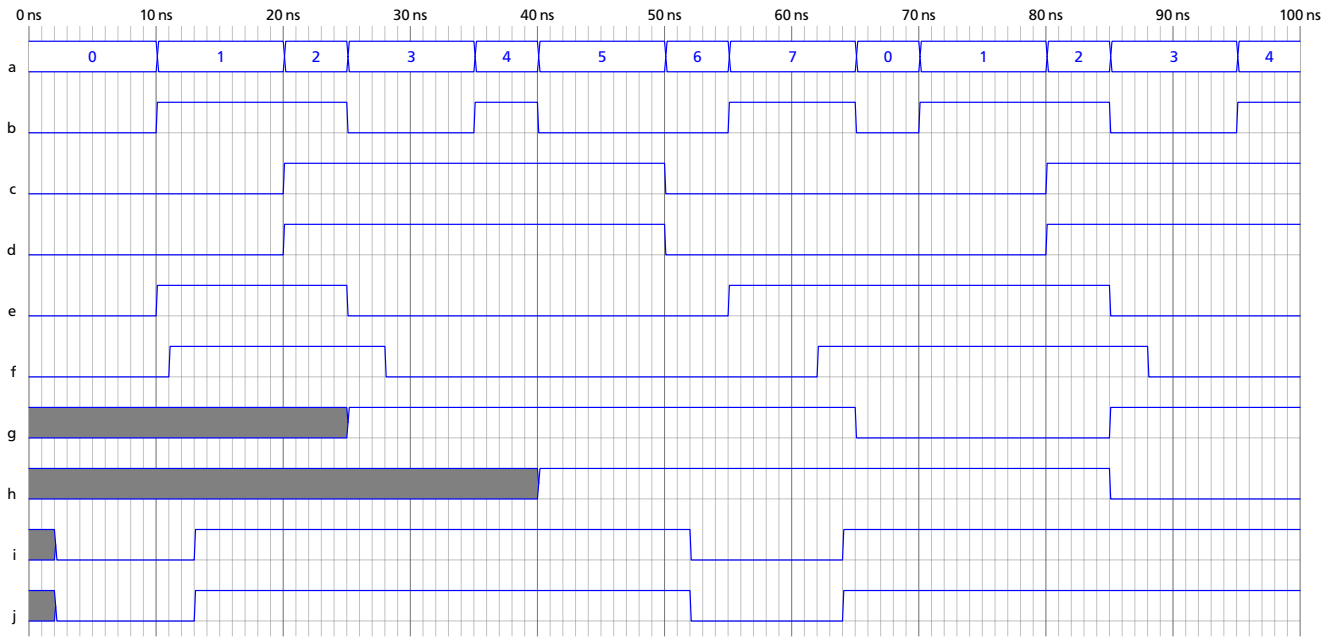
Übung 11.1 Zeitverhalten sequentieller Beschreibungen [15 min]

Ermitteln Sie das Schaltverhalten der nachfolgenden Signale für die ersten 100 ns. Bedenken Sie dabei, dass bei einfachen **`always`** Blöcken (im Gegensatz zu **`always_comb`**) die Signalinitialisierung nicht als Signaländerung interpretiert wird.

seq/timing.sv

```
1  `timescale 1 ns / 10 ps
2  module timing;
3    localparam x = 2;
4
5    logic [2:0] a = 0;
6    always begin if (!a[0]) #10; else #(3+x); a <= a+1; end
7
8    logic b, c, d, e, f, g, h, i, j;
9    assign b = ^a;
10   always               begin             c  = b;     d  = c; @(negedge a[0]); end
11   always               begin             e  = b; #a; f  = e; @(posedge a[0]); end
12   always @(negedge b) begin     g <= c; h <= g;                              end
13   always @(f|d)       begin #2; i  = e; j <= i;                              end
14 endmodule
```

Achtung: in Zeile 6 wird die Bedingung `!a[0]` für den alten Wert von a vor der letzten nicht nicht-blockierenden Zuweisung geprüft!



Übung 11.2 Strukturierung von sequentiellen Beschreibungen [15 min]

Wandeln Sie folgende kontrollflusslastige Beschreibung eines sequentiellen 4 bit Multiplizierers in eine äquivalente Beschreibung um, welche dessen Umsetzung als Register-Transfer-Logik besser erkennen lässt. Verfolgen Sie dafür folgende Grundregeln:

- nur ein Signal pro **`always_ff`** Block (beschreibt ein Register)
- kombinatorische Logik *vollständig* als nebenläufige Zuweisungen (beschreibt die Transfer-Logik)

arith/mul/sequential.sv

```
module mul (input  logic CLK, RST, START, input  logic [3:0] A, B,
            output logic DONE,            output logic [7:0] Y);

  logic [2:0] n;
  logic [3:0] b;
  logic [7:0] a, p;

  always_ff @(posedge CLK) begin
    if (RST) begin
      {n, a, b, p, DONE, Y} <= 0;
    end else if (START) begin
      p <= 0; a <= A; b <= B; n <= 4; DONE <= 0;
    end else if (n > 1) begin
      if (b[0]) p <= p + a;
      a <= a << 1; b <= b >> 1; n <= n-1;
    end else if (n == 1) begin
      Y <= b[0] ? p + a : p; n <= 0; DONE <= 1;
    end else begin
      {DONE, Y} <= 0;
    end
  end
endmodule
```

arith/mul/rtl.sv

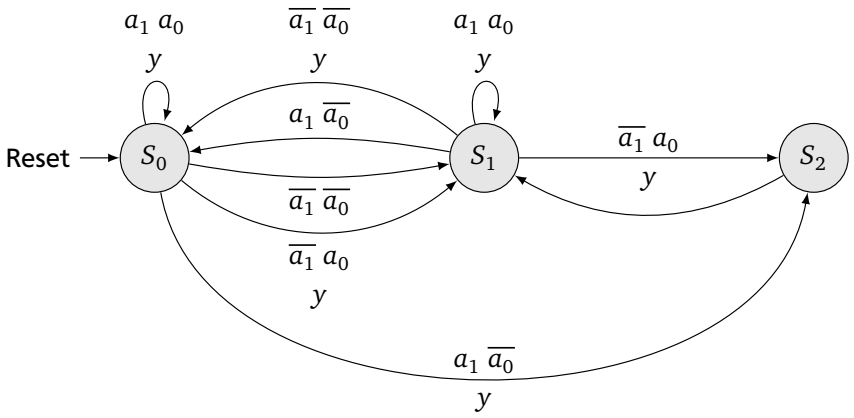
```
module mul (input  logic CLK, RST, START, input  logic [3:0] A, B,
            output logic DONE,            output logic [7:0] Y);

  logic       doneD;
  logic [2:0] n, nD;
  logic [3:0] b, bD;
  logic [7:0] a, aD, p, pa, pD, yD;

  // Register
  always_ff @(posedge CLK) n    <= nD;
  always_ff @(posedge CLK) a    <= aD;
  always_ff @(posedge CLK) b    <= bD;
  always_ff @(posedge CLK) p    <= pD;
  always_ff @(posedge CLK) DONE <= doneD;
  always_ff @(posedge CLK) Y    <= yD;

  // Transfer-Logik
  assign nD    = RST ? 0 : START ? 4 : n > 0 ? n-1 : 0;
  assign aD    = RST ? 0 : START ? A : a << 1;
  assign bD    = RST ? 0 : START ? B : b >> 1;
  assign pa    = b[0] ? p+a : p;
  assign pD    = RST || START ? 0 : pa;
  assign doneD = n == 1 ? 1  : 0;
  assign yD    = n == 1 ? pa : 0;
endmodule
```

Die kontrollflusslastige Beschreibung (alles in einem **always_ff** Block) spiegelt den zugrundeliegenden Algorithmus idR. besser wider und kommt einer Software-Implementierung nahe. Die RTL-nahe Beschreibung (möglichst viele/kleine **always_ff** Blöcke) spiegelt die parallel arbeitenden Hardware-Komponenten und die Abhängigkeiten dazwischen besser wieder und erleichtert so die Identifikation von kritischen Pfaden. In der Praxis ist ein Mittelweg zwischen beiden Extremen häufig die beste Wahl. Dabei sollten Signale möglichst nur dann in einem **always_ff** Block zusammengefasst werden, wenn sie die gleichen Kontrollbedingungen haben.

Übung 11.3 Robuste Endliche Automaten [20 min]

Implementieren Sie folgende endliche Automaten in SystemVerilog. Wenn eines der Eingangsbits `1'bz` oder `1'bx` ist, soll der Automat in den Startzustand wechseln und dabei kein Ausgangsbit auf 1 setzen. Verwenden Sie den `===` Operator zum Vergleich zwischen Ausdrücken vierwertiger Logik, da ein Vergleich mit `s == 1'bx` für alle Werte von `s` immer `1'bx` ergibt und dieses als logisch falsch interpretiert wird.
Die ungültigen Eingänge fängt man am besten direkt im Zustandsregister ab, damit die next-state Logik diese Fälle nicht mehr beachten muss. Da `~1'bz===~1'bx===1'bx` (siehe V6 Folie 31), genügt dafür ein Vergleich pro Eingabebit. Für Mealy-Automaten muss der `===` Operator auch in der Ausgabelogik verwendet werden.

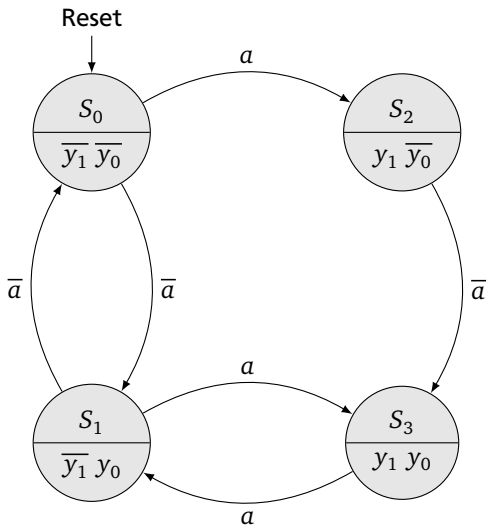
a)



fsm/robust/mealy.sv

```
module mealy(input logic CLK, RST, input logic [1:0] A, output logic Y);
  logic [1:0] state, nextstate;

  always_ff @(posedge CLK)
    state <= RST || ~A[0]===1'bx || ~A[1]===1'bx ? 0 : nextstate;

  always_comb case (state)
          0: nextstate = ~A[1] ? 1 : ~A[0] ? 2 : 0;
          1: nextstate = ~A[0] ? 0 : ~A[1] ? 2 : 1;
    default: nextstate = 1;
  endcase

  assign Y = (state==0 && (A===2'b11 || A===2'b01 || A===2'b10))
          || (state==1 && (A===2'b11 || A===2'b01 || A===2'b00));
endmodule
```

b)



fsm/robust/moore.sv

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

Übung 11.4 RCA-basierter Zähler [30 min]

Übung 11.4.1 Generischer Ripple-Carry Adder (RCA)

Implementieren Sie den RCA aus Übung 10.5.3 für eine generische Bitbreite mit dem Parameter `WIDTH`. Halb- und Volladdierer sollen aus den Übungen 10.5.1 und 10.5.2 übernommen werden.

arith/rca.sv

```
1  module rca #(parameter    WIDTH=4)
2              (input logic [WIDTH-1:0] A, B, output logic [WIDTH:0] S);
3
4    logic  [WIDTH:0] c;
5    assign c[0]     = 1'b0;
6    assign S[WIDTH] = c[WIDTH];
7
8    genvar i;
9    generate
10     for (i=0; i<WIDTH; i=i+1) full_adder fa (A[i], B[i], c[i], S[i], c[i+1]);
11   endgenerate
12
13 endmodule
```

Übung 11.4.2 Zähler

Verwenden Sie den generischen RCA zur Implementierung eines 10 bit Zählers mit folgender Schnittstelle:

arith/counter.sv

```
1  `timescale 1 ns / 10 ps
2  module counter(input  logic       CLK, // Taktsignal
3                                    RST, // synchrones Reset, high-active
4                                    INC, // Zähler erhöhen,   high-active
5                 output logic [9:0] VAL);// aktueller Zählerwert
```

Das Zählerregister soll eine Ausgabeverzögerung von $t_{ccq} = t_{pcq} = 2\,\text{ns}$ haben.

arith/counter.sv

```
7   logic  [9:0] tmp;
8   logic [10:0] inc;
9   rca  #(10) add (VAL, 10'd1, inc);
10
11  always_ff @(posedge CLK) begin
12       tmp <= RST ? 0 : INC ? inc : VAL;
13    #2 VAL <= tmp;
14  end
15
16 endmodule
```

Beim Einbinden des Addierers (Zeile 8) muss der passende Parameter für die Bitbreite der Eingänge angegeben werden. Der `INC` Eingang wird als Taktfreigabe (enable) der Zählerregister verwendet (Zeile 11). Der inkrementierte Wert wird also immer berechnet, aber nur bei `INC==1` gespeichert. Der neue Zählerwert wird zur steigenden Taktflanke zunächst in `tmp` gespeichert (Zeile 11). Erst nach Ablauf der Ausgabeverzögerung wird der neue Zählerwert am Registerausgang `VAL` sichtbar (Zeile 12).

Übung 11.4.3 Testbench

Implementieren Sie eine selbstüberprüfende Testbench für den Zähler aus Übung 11.4.2. Diese soll einen 20 MHz Takt an den Zähler anlegen und das funktionale Verhalten des Zählers überprüfen.

arith/counter_tb.sv

```
1  `default_nettype none
2  `timescale 1 ns / 10 ps
3
4  module counter_tb;
5
6    // toggle clock after half period
7    logic rst,inc,clk=0;
8    always #(0.5/0.02) clk = ~clk;
9
10   logic [9:0] val;
11   counter uut(clk,rst,inc,val);
12
13   initial begin
14     $dumpfile("counter_tb.vcd");
15     $timeformat(-9, 0, " ns", 8);
16     $dumpvars;
17
18     // test reset
19     inc <= 0;
20     rst <= 1;
21     @(posedge clk);
22     #3 if (val != 0) $display("%t: expected %0d but got %0d", $time, 0, val);
23
24     // count to overflow
25     inc <= 1;
26     rst <= 0;
27     for (int i=1; i<(1<<$size(val)); i++) begin
28       @(posedge clk);
29       #3 if (val != i) $display("%t: expected %0d but got %0d", $time, i, val);
30     end
31
32     // test overflow
33     for (int i=0; i<10; i++) begin
34       @(posedge clk);
35       #3 if (val != i) $display("%t: expected %0d but got %0d", $time, i, val);
```

```
36      end
37
38      // test inc inactive
39      inc <= 0;
40      for (int i=0; i<5; i++) begin
41        @(posedge clk);
42        #3 if (val != 9) $display("%t: expected %0d but got %0d", $time, 9 , val);
43      end
44
45      // test reset
46      rst <= 1;
47      inc <= 1;
48      @(posedge clk);
49      #3 if (val != 0) $display("%t: expected %0d but got %0d", $time, 0, val);
50
51      $display("FINISHED counter_tb");
52      $finish;
53    end
54
55  endmodule
```

20 MHz = 0,02 GHz enspricht einer Taktperiode von $\frac{1\,\text{ns}}{0.02}$. Die Verzögerung zwischen dem Toggeln des Taktsignals muss die Hälfte der Taktperiode betragen (Zeile 8). Im **`initial`** Block werden verschiedene Kombinationen der Steuersignale (`INC` und `RST`) erzeugt. Nach der nächsten steigenden Taktflange und einer zusätzlichen Verzögerung (größer als die Ausgabeverzögerung des Zählerregisters), wird der Wert am Zählerausgang überprüft. Bei bestimmten Zählerständen ist die RCA-Verzögerung (kritischer Pfad) größer als die Taktperiode. Dies führt zu funktionalen Fehlern, die von der Testbench erkannt werden sollten:

```
1  VCD info: dumpfile counter_tb.vcd opened for output.
2  25628 ns: expected 512 but got 0
3  25678 ns: expected 513 but got 1
4  25728 ns: expected 514 but got 2
5  25778 ns: expected 515 but got 3
6  25828 ns: expected 516 but got 4
7  ...
8  51028 ns: expected 1020 but got 508
9  51078 ns: expected 1021 but got 509
10 51128 ns: expected 1022 but got 510
11 51178 ns: expected 1023 but got 511
12 FINISHED counter_tb
13 WM Destroy
```