Digitaltechnik
Wintersemester 2017/2018
12. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Andreas Engel, Raad Bahmani **LÖSUNGSVORSCHLAG** **KW04**

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.
Die mit **„Zusatzaufgabe“** gekennzeichneten Aufgaben sind zur zusätzlichen Vertiefung für interessierte Studierende gedacht und daher nicht im Zeitumfang von 90 Minuten einkalkuliert.

Übung 12.1 Wiederholung: Rekursive Moduldefinition [25 min]

In dieser Aufgabe soll ein kombinatorischer Addierer mit einer parametrisierten Breite (Bits pro Operand) und Tiefe (Anzahl der Operanden) mit folgender Schnittstelle implementiert und getestet werden:

arith/acc.sv

```
1  module acc #(parameter      WIDTH = 4,  // Bits pro Operand
2               parameter      DEPTH = 2)  // Anzahl der Operanden
3              (input  logic [WIDTH*DEPTH        -1:0] O,
4               output logic [WIDTH+$clog2(DEPTH)-1:0] R);
```

Dabei berechnet die **$clog2** den Zweierlogarithmus einer natürlichen Zahl. Zweidimensionale Ports sind in SystemVerilog zwar prinzipiell erlaubt, werden von vielen Simulations- und Synthese-Tools aber nicht unterstützt. Daher werden die einzelnen Operanden zu einem langen Bitvektor (O) konkateniert. Dieses Konzept wird auch als *flattening* bezeichnet.

Übung 12.1.1 Implementierung

Implementieren Sie diesen Addierer mit einem möglichst kurzen kritischen Pfad. Erzeugen Sie dazu einen möglichst flachen Addierer-Baum. Für eine Tiefe von vier entspricht dies der Berechnung $R = (O_0 + O_1) + (O_2 + O_3)$. Einen solchen Baum erhält man bspw. durch eine rekursive Moduldefinition, bei der jeweils ein Addierer-Baum für die obere und die untere Operandenhälfte erzeugt, und anschließend das Ergebnis der beiden Bäume addiert wird. Beachten Sie dabei, dass die Summe zweier n Bit breiter Zahlen $n + 1$ Bit breit ist. Sie können davon ausgehen, dass `DEPTH` immer eine Zweierpotenz darstellt.

arith/acc.sv

```
6    logic [WIDTH+$clog2(DEPTH)-2:0] sh, sl;
7    assign R = sh + sl;
8
9    generate
10     // Rekursive Instantiierung
11     if (DEPTH > 2) begin
12       acc #(WIDTH, DEPTH/2) ahigh (O[ DEPTH   *WIDTH-1:(DEPTH/2)*WIDTH], sh);
13       acc #(WIDTH, DEPTH/2) alow  (O[(DEPTH/2)*WIDTH-1: 0       *WIDTH], sl);
14
15     // Rekursionsabbruch
16     end else begin
17       assign sh = O[1*WIDTH +: WIDTH];
18       assign sl = O[0*WIDTH +: WIDTH];
19     end
20   endgenerate
21
22 endmodule
```

Übung 12.1.2 Verifikation

Generieren Sie eine Testbench zur funktionalen Verifikation aller 2 bit breiten Addierer mit zwei, vier und acht Operanden. Versuchen Sie dabei, möglichst wenig redundanten Code zu verwenden.

arith/acc_tb.sv

```
`timescale 1 ns / 10 ps
module acc_tb;

  localparam WIDTH = 2;

  // Simulation starten, konfigurieren und stoppen
  initial begin
    $dumpfile("acc_tb.vcd");
    $timeformat(-9, 0, " ns", 8);
    $dumpvars;
    #(2**(WIDTH*8)+2);
    $display("FINISHED acc_tb");
    $finish;
  end

  // Generate-Schleife über alle zu testenden DEPTH Werte
  genvar n;
  generate
    for (n=1; n<=3; n=n+1) begin
      localparam DEPTH = 2**n;

      // Unit under test
      logic [WIDTH*DEPTH-1:0] o;
      logic [WIDTH+n    -1:0] r, a;
      acc  #(WIDTH, DEPTH) uut (o, r);

      // erschöpfender Test: alle Eingabekombinationen prüfen
      int i,k;
      initial begin
        #1 $display("START pass for DEPTH=%0d", DEPTH);

        for (i=0; i<2**(WIDTH*DEPTH); i++) begin
          o = i; #1;

          // erwartetes Ergebnis akkumulieren
          a = 0;
          for (k=0; k<DEPTH; k++) a += o[k*WIDTH +: WIDTH];
          if (r!=a) $display("[%0d] %t: expected %4d but got %4d",DEPTH,$time,a,r);
        end

        $display("FINISHED pass for DEPTH=%0d", DEPTH);
      end
    end
  endgenerate
endmodule
```

Übung 12.2 Pipelining [20 min]

Folgende SystemVerilog Module sind auch im Moodle verfügbar und beschreiben eine kombinatorische Schaltung ($Y = \overline{(A\,B) \oplus (A\,B + C) \oplus \overline{D}}$) zwischen zwei Register-Stufen (`base`) sowie die dazugehörige Testbench für dessen funktionale Verifikation (`base_tb`):

seq/pipeline/gates.sv

```
`timescale 1 ns / 10 ps
module or_gate  #(parameter W=2) (input logic [W-1:0] A, output logic Y);
  assign #(W)   Y = |A;
endmodule

module and_gate #(parameter W=2) (input logic [W-1:0] A, output logic Y);
  assign #(W)   Y = &A;
endmodule

module xor_gate #(parameter W=2) (input logic [W-1:0] A, output logic Y);
  assign #(W+1) Y = ^A;
endmodule

module inv_gate                  (input logic         A, output logic Y);
  assign #(1)   Y = ~A;
endmodule
```

seq/pipeline/register.sv

```
`timescale 1 ns / 10 ps
module register #(parameter W      = 1,
                  parameter tsetup = 0.9,
                  parameter thold  = 0.5,
                  parameter tcq    = 0.1)
                 (input logic CLK, input logic [W-1:0] D, output logic [W-1:0] Q);

  logic [W-1:0] t;
  always @(posedge CLK) begin
           t <= D;
    #(tcq) Q <= t;
  end
endmodule
```

seq/pipeline/base.sv

```
module base (input logic CLK, A, B, C, D, output logic Y);
  logic a,b,c,d,n1,n2,n3,n4,y;
  register #(4) rin (CLK, {A,B,C,D}, {a,b,c,d});
  and_gate      g1  ({a,  b}, n1);
  or_gate       g2  ({n1, c}, n2);
  inv_gate      g3  ( d,      n3);
  xor_gate #(3) g4  ({n1, n2, n3}, n4);
  inv_gate      g5  ( n4, y);
  register      rout(CLK, y,  Y);
endmodule
```

seq/pipeline/base_tb.sv

```
`timescale 1 ns / 10 ps
module base_tb;

  logic a,b,c,d,y,clk = 0;
  always #4.9 clk = ~clk;

  base uut (clk,a,b,c,d,y);

  localparam L = 2;
  logic [L-1:0] e;
  always @(posedge clk) begin
```

```
12      e <= {e[L-2:0], ~(a&b ^ (a&b|c) ^ ~d)};
13    end
14
15    initial begin
16      $dumpfile("base_tb.vcd");
17      $timeformat(-9, 2, " ns", 10);
18      $dumpvars;
19
20      for (int i=0; i<16+L; i++) begin
21        #1 {a,b,c,d} <= i;
22        if (y!==e[L-1]) $display("%t: expected %0d but got %0d",$realtime,e[L-1],y);
23        @(posedge clk);
24      end
25
26      $display("FINISHED base_tb");
27      $finish;
28    end
29
30  endmodule
```

Übung 12.2.1 Timing-Analyse

Die Register-Parameter `tsetup`, `thold` und `tcq` beschreiben die Setup-, Hold- und Verzögerungszeit ($t_{pcq} = t_{ccq}$) des Registers in Nanosekunden. Mit welcher Frequenz kann das `base` Modul maximal getaktet werden? Welche Latenz hat das Modul.

Der kritische Pfad zwischen `rin` und `rout` verläuft durch `g1` (2 ns), `g2` (2 ns), `g4` (4 ns) und `g5` (1 ns). Zusammen mit `tcq` und `tsetup` der Register ergibt dies 10 ns. Das Modul kann daher höchstens mit 100 MHz getaktet werden. Die zwei Registerstufen erzeugen eine Latzenz von zwei Takten, oder mindestens 20 ns.

Übung 12.2.2 Testbench-Analyse

Mit welcher Frequenz wird die *unit under test* in der Testbench getaktet? Warum entdeckt die Testbench keine funktionalen Fehler, obwohl die Timing-Bedingungen der Register im `base` Module verletzt werden?

Das `clk` Signal schaltet alle 4,9 ns um, wodurch eine Taktfrequenz von $\frac{1}{9{,}8\,\text{ns}} = 102\,\text{MHz}$ erzeugt wird. Der Dateneingang an `rin` ist dadurch spätestens 0,2 ns vor der steigenden Taktflanke stabil. Dies verletzt zwar die Setup-Bedingung, die Register-Implementierung liest den Dateneingang aber auch erst zur steigenden Taktflanke, wodurch der richtige Wert übernommen wird. Insbesondere die Setup-Bedingung muss also durch zusätzliche Tests überprüft werden.

Übung 12.2.3 Überprüfen von Setup- und Hold-Bedinung

Erweitern Sie `seq/pipeline/register.sv` so, dass die Register ihre Timing-Bedingungen (t_{setup} und t_{hold}) selbst überwachen.

seq/pipeline/register.sv

```
15  real lastDevent = 0, lastCLKposedge = 0, setup, hold;
16
17  always @(D) begin
18    hold = $realtime - lastCLKposedge;
19    if (hold < thold) $display("%t@%m, D event %0t after CLK (hold violation)",
20                               $realtime,     hold);
21    lastDevent = $realtime;
22  end
23
24  always @(posedge CLK) begin
25    setup = $realtime - lastDevent;
26    if (setup < tsetup) $display("%t@%m: D event %0t before CLK (setup violation)",
27                                 $realtime,     setup);
28    lastCLKposedge = $realtime;
29  end
```

Zum Überprüfen der beiden Timing-Bedingungen werden die Zeitpunkte der Änderungen am Dateneingang und der steigenden Taktflanken mit jeweils einem **always** Block überwacht. Die Zeitpunkte der letzten Ereignisse werden in den Variablen `lastDevent` und `lastCLKposedge` gespeichert. So kann bei der nächsten steigenden Taktflanke bestimmt werden, wie lange die letzte Änderung des Dateneingangs bereits her ist, um so die Setup-Bedingung zu testen (Zeile 12). Umgekehrt kann bei jeder Änderung des Dateneingang bestimmt werden, wie lange die letzte steigende Taktflange bereits zurück liegt, um so die Hold-Bedingung zu überprüfen (Zeile 5).

Übung 12.2.4 Zusätzliche Pipeline-Stufen

Modifizieren Sie `seq/pipeline/base.sv` so, dass dieses mit 200 MHz getaktet werden kann. Die verwendeten Logikgatter sollen dabei nicht verändert werden. Modifizieren Sie auch `seq/pipeline/base_tb.sv` so, dass das schnellere Modul korrekt getestet wird. Wie wirkt sich diese Modifikation auf die Latenz der Schaltung aus.
Um eine höhere Taktrate zu erreichen, müssen mehr Pipeline-Register eingezogen werden. Das XOR3-Gatter hat die längste Verzögerungszeit und muss für die größtmögliche Taktrate alleine in einer Pipelinestufe stehen:

```
1  module fast (input logic CLK, A, B, C, D, output logic Y);
2    logic a,b,c,d,n1,n2,n3,n4,n1r,n2r,n3r,n4r,y;
3    register #(4) rin (CLK, {A,B,C,D}, {a,b,c,d});
4    and_gate      g1  ({a,  b}, n1);
5    or_gate       g2  ({n1, c}, n2);
6    inv_gate      g3  (d, n3);
7    register #(3) rp1 (CLK, {n1,n2,n3}, {n1r, n2r, n3r});
8    xor_gate #(3) g4  ({n1r, n2r, n3r}, n4);
9    register      rp2 (CLK, n4, n4r);
10   inv_gate      g5  (n4r, y);
11   register      rout(CLK, y,  Y);
12 endmodule
```

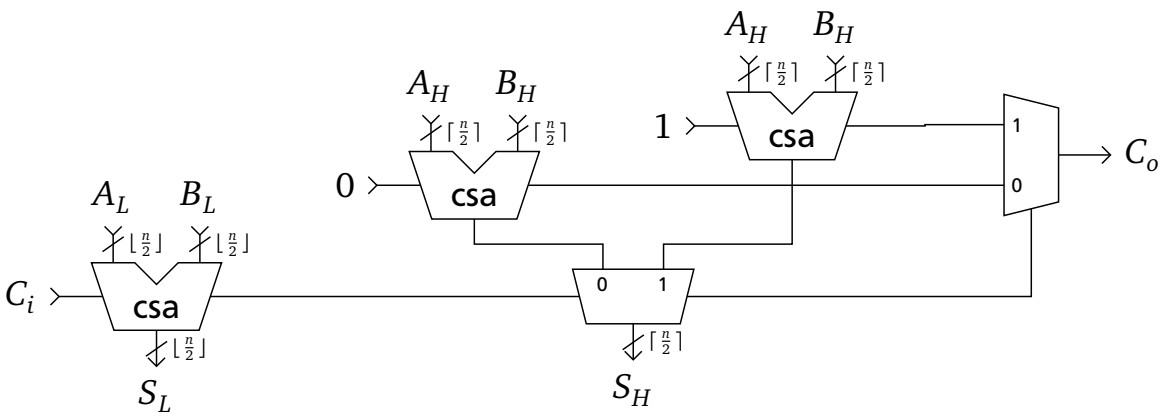
Der resultierende kritische Pfad ist $t_{\text{pcq}} + t_{\text{pd,XOR3}} + t_{\text{setup}} = 5\,\text{ns}$, was eine Taktrate von 200 MHz ermöglicht. Eine weitere Pipeline-Stufe zwischen OR2- und AND2-Gatter würde den kritischen Pfad hingegen nicht weiter verkürzen. Zum Anpassen der Testbench muss

- in Zeile 5 die Toggle-Periode aus 2,5 ns gesetzt werden
- in Zeile 7 das `fast` Modul instantiiert werden
- in Zeile 9 die `uut` Latenz auf 4 Takte erhöht werden

Die Verdopplung der Latenz auf 4 Takte wird durch die Halbierung der Pediodendauer kompensiert. Die absolute Latenz bleibt daher unverändert bei 20 ns.

Übung 12.3 Conditional Sum Adder (CSA) [25 min]

Ein Nachteil des Ripple-Carry-Adders ist dessen lineare Übertragskette vom LSB bis zum MSB, wodurch der kritische Pfad linear mit der Bitbreite ansteigt. Ein n-Bit CSA bricht diese Übertragskette auf, indem für die oberen $\lceil \frac{n}{2} \rceil$ Eingabebits sowohl die einfache Summe ($A_H + B_H$), als auch dessen Inkrement ($A_H + B_H + 1$) *gleichzeitig* berechnet werden. Sobald der Übertrag des unteren Halbworts ($A_L + B_L + C_i$) verfügbar ist, muss nur noch das korrekte Ergebnis (Summe und Übertrag) aus den beiden Berechnungen für das obere Halbwort ausgewählt werden:



Übung 12.3.1 Rekursive Implementierung

Implementieren Sie den CSA in SystemVerilog als rekursives Modul mit Übertragsein- und ausgang:

arith/csa.sv

```
2  module csa #(parameter     WIDTH=4)
3              (input logic  [WIDTH-1:0] A, B, input  logic CI,
4               output logic [WIDTH-1:0] S,    output logic CO);
```

Ein 1 bit CSA entspricht damit gerade einem Volladdierer. Beachten Sie, dass `WIDTH` nicht immer ohne Rest durch zwei teilbar ist. Halb- und Volladdierer sollen aus den Übungen 10.5.1 und 10.5.2 übernommen werden. Die Verzögerungszeit der Multiplexer soll 4 ns betragen.

```
6    generate
7      if (WIDTH > 1) begin        // Rekursion fortsetzen
8        localparam WL = WIDTH/2;  // floor(WIDTH/2)
9        localparam WH = WIDTH-WL; // ceil (WIDTH/2)
10       logic [WH-1:0] sh0,sh1;
11       logic      cl, ch0,ch1;
12       csa #(WL) csal  (A[ 0 +: WL], B[ 0 +: WL], CI,   S[0 +: WL], cl);
13       csa #(WH) csah0 (A[WL +: WH], B[WL +: WH], 1'b0, sh0,        ch0);
14       csa #(WH) csah1 (A[WL +: WH], B[WL +: WH], 1'b1, sh1,        ch1);
15       assign #4 {CO,S[WL +: WH]} = cl ? {ch1,sh1} : {ch0,sh0}; // verzögerter MUX
16
17     end else full_adder fa (A, B, CI, S, CO); // Rekursionsende
18   endgenerate
19 endmodule
```

Übung 12.3.2 Modul-Kapselung

Verpacken Sie Ihren CSA in ein Modul mit der allgemeinen Addierer-Schnittstelle:

```
21 module add #(parameter    WIDTH=4)
22             (input logic [WIDTH-1:0] A, B, output logic [WIDTH:0] S);

24   csa #(WIDTH) inst (A, B, 1'b0, S[WIDTH-1:0], S[WIDTH]);
25 endmodule
```

Übung 12.3.3 Verifikation

Schreiben Sie eine Testbench, die einen CSA mit einer per **`localparam`** konfigurierbaren Bitbreite erschöpfend funktional validiert. Bestimmen Sie dabei auch die maximale Verzögerungszeit des CSA und vergleichen Sie diese mit der Verzögerung der entsprechenden RCA Implementierung. Ergänzen Sie dazu folgende Tabelle:

WIDTH	2	4	6	8	10
$t_{pd,RCA}$	10 ns	20 ns	30 ns	40 ns	50 ns
$t_{pd,CSA}$	12 ns	16 ns	20 ns	20 ns	24 ns

arith/add_tb.sv

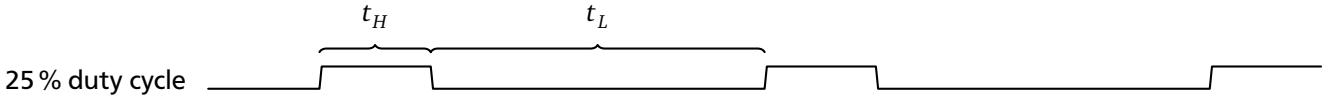
```
1  `timescale 1 ns / 10 ps
2  module add_tb;
3
4    localparam W = 8;
5
6    logic [W-1:0] a,b;
7    logic [W  :0] s;
8
9    add #(W) uut(a,b,s);
10
11   real inputEvent;  // letzte Änderung an a oder b
```

```
12    real delay;       // Verzögerungszeit
13    real maxDelay = 0;// maximale Verzögerungszeit
14
15    always @s delay = $realtime-inputEvent;
16
17    initial begin
18      $dumpfile("add_tb.vcd");
19      $timeformat(-9, 0, " ns", 8);
20      $dumpvars;
21
22      // erschöpfender Test: alle Eingabekombinationen prüfen
23      for (int i=0; i<(1<<2*W); i++) begin
24        {a,b} = i;
25        inputEvent = $realtime;
26        #(10*W); // warten, bis s sicher stabil ist
27        if (s != a+b) $display("%t: %0d+%0d=%0d but got %0d", $time, a, b, a+b, s);
28        if (delay > maxDelay) maxDelay = delay;
29      end
30
31      $display("FINISHED add_tb for W=%0d, max delay=%f", W, maxDelay);
32      $finish;
33    end
34  endmodule
```

Durch die einheitliche Addierer-Schnittstelle kann die Testbench mit minimalen Änderungen (Anpassung des Modulnamens in Zeile 9) für RCA und CSA verwendet werden. Die Verzögerungszeit des CSA steigt nicht linear mit der Bitbreite an, aber erst ab einer bestimmten Bitbreite ist der CSA schneller als der RCA. Für die hier verwendeten Gatterverzögerungen sind dies 4 bit.

Übung 12.4 Pulsweitenmodulation (PWM) - **Zusatzaufgabe** [15 min]

Bei einem periodischen Rechtecksignal wechseln sich high-Phase (t_H) und low-Phase (t_L) ab. Die Summe beider Phasen entspricht der Periodendauer des Signals und das Verhältnis aus high-Phase zur Periodendauer ($\frac{t_H}{t_H+t_L}$) wird als Tastgrad („duty cycle“) bezeichnet. In dieser Aufgabe wird ein PWM Modul mit einem zur Laufzeit (nach der Synthese) konfigurierbaren Tastgrad entwickelt und validiert.



Übung 12.4.1 Implementierung

Beschreiben Sie ein SystemVerilog PWM Modul mit folgender Schnittstelle:

seq/pwm.sv

```
1  module pwm #(parameter WIDTH=4)
2              (input logic CLK, RST, input logic [WIDTH-1:0] DC, output logic Y);
```

Neben dem Takt- und dem Reset-Signal wird der dritte Eingang (`DC`) verwendet, um den Tastgrad des Ausgangs einzustellen. Dabei soll am Ausgang `Y` ein periodisches Signal mit dem Tastgrad $\frac{\texttt{DC}}{2^{\texttt{WIDTH}}}$ erzeugt werden. Die Periodendauer des Ausgangs soll unabhänig von `DC` sein und $2^{\texttt{WIDTH}}$ Takten entsprechen.

seq/pwm.sv

```
4    logic [WIDTH-1:0] cnt;
5    always @(posedge CLK) cnt <= RST ? 0 : cnt+1;
6
7    assign Y = cnt < DC;
8  endmodule
```

Der Zähler `cnt` beginnt nach jedem Überlauf erneut bei Null und erzeugt so ein periodisches Verhalten mit einer Periodendauer von $2^{\texttt{WIDTH}}$ Takten unabhängig von `DC`. Durch den Vergleich des Zählerwerts mit `DC` (Zeile 7) wird `Y` für die ersten `DC` Takte auf 1, und für die restlichen $2^{\texttt{WIDTH}} - \texttt{DC}$ Takte auf 0 gesetzt. Dies entspricht dem geforderten Tastgrad.

Übung 12.4.2 Verifikation

Implementieren Sie eine Testbench zur funktionalen Verifikation des PWM Moduls mit 256 konfigurierbaren Tastgraden. Das Modul soll mit 1 MHz getaktet werden. Verifizieren Sie *alle* konfigurierbaren Tastgrade.

seq/pwm_tb.sv

```
`timescale 1 us / 10 ns
module pwm_tb;

  logic y, rst=1, clk=0;
  always  #0.5            clk = ~clk;
  initial @(posedge clk) rst <= 0;

  localparam W = 8;
  logic [W-1:0] dc;
  pwm #(W) uut (clk,rst,dc,y);

  real r,f,m,e;
  initial begin
    $dumpfile("pwm_tb.vcd");
    $timeformat(-6, 2, " us", 10);
    $dumpvars;

    // Spezialbehandlung für DC=0, weil keine y events generiert werden
    dc = 0;
    for (int i=0; i<2**W; i++) begin
      if (y) $display("%t: y high unexpected", $realtime);
      @(posedge clk);
    end

    // alle anderen Tastgrade können über die y events ausgemessen werden
    for (int i=1; i<2**W; i++) begin
      dc = i;
      @(posedge y);
      r = $realtime;
      @(negedge y);
      f = $realtime;
      @(posedge y);
      e = dc/2.0**W;            // erwarteter duty cycle
      m = (f-r)/($realtime-r);  // gemessener duty cycle
      if (e != m) $display("%t: expected %f but got %f", $realtime, e, m);
    end

    $display("FINISHED pwm_tb");
    $finish;
  end

endmodule
```