Digitaltechnik
Wintersemester 2017/2018
12. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Andreas Engel, Raad Bahmani **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.

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.

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;

```

```
7   base uut (clk,a,b,c,d,y);
8
9   localparam L = 2;
10  logic [L-1:0] e;
11  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.

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?

Übung 12.2.3 Überprüfen von Setup- und Hold-Bedinung

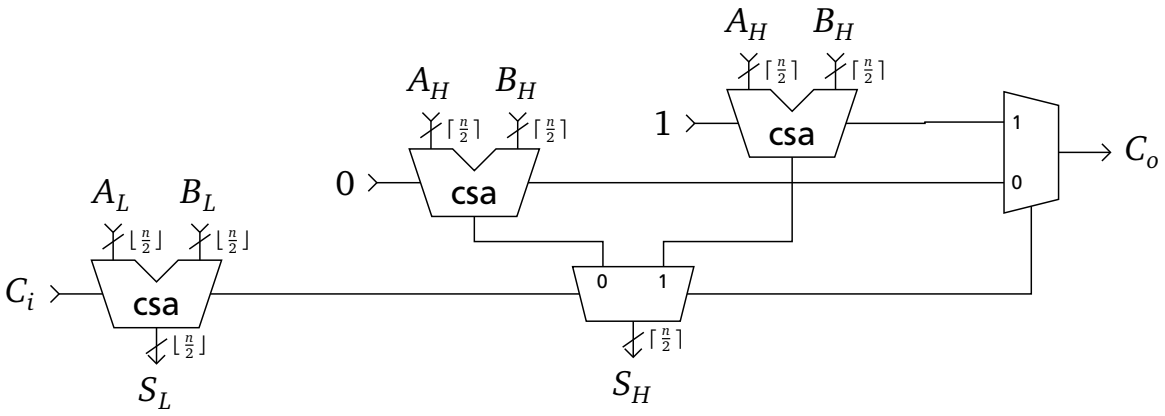
Erweitern Sie `seq/pipeline/register.sv` so, dass die Register ihre Timing-Bedingungen (t_{setup} und t_{hold}) selbst überwachen.

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.

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.

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);
```

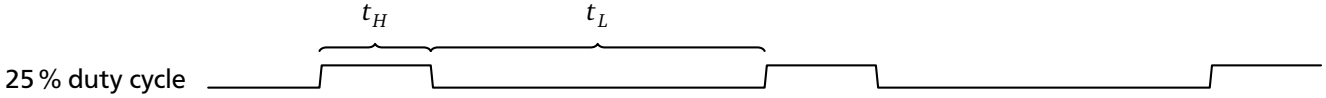
Übung 12.3.3 Verifikation

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
$t_{pd,RCA}$					
$t_{pd,CSA}$					

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.

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.