Digitaltechnik
Wintersemester 2021/2022
12. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Dr.-Ing. Thomas Schneider, M.Sc. Daniel Günther, M.Sc. Amos Treiber
Tablet Version **KW05**

Bitte bearbeiten Sie die Übungsblätter bereits im Voraus, sodass Sie Ihre Lösungen zusammen mit Ihren Kommilitonen und Tutoren während der wöchentlichen Übungsstunde diskutieren können.
Mit der angegebenen Bearbeitungszeit für die einzelnen Aufgaben können Sie Ihren Leistungsstand besser einschätzen.

Übung 12.1 2 bit Addierer [20 min]

In dieser Aufgabe implementieren Sie einen 2 bit Addierer auf verschiedene Arten.
Bei einem 2 bit Addierer handelt es sich um eine kombinatorische Schaltung mit folgender Schnittstelle:

- Inputs:
  - 2 bit breite Zahlen A und B ($A := a_1 a_0$, $B := b_1 b_0$)
- Outputs:
  - 2 bit breite Summe S von A und B ($S := s_1 s_0$)
  - Übertrag C

a) Implementieren Sie den 2 bit Addierer mit Basisgattern:
   1. Stellen Sie die Wahrheitstabelle für den 2 bit Addierer auf.

2. Nutzen Sie Karnaugh Diagramme, um die minimierte DNF für s_0, s_1 und C zu ermitteln.

3. Implementieren Sie die Schaltung als zweistufige Logik.

4. Modellieren Sie die zweistufige Logik als Verhaltensbeschreibung in SystemVerilog. Verwenden Sie für Basisgatter die entsprechenden Operatoren (&, |, ∼).

b) Implementieren Sie den 2 bit Addierer mit Hilfe von Halb- und Volladdierern.

1. Ein Halbaddierer ist eine kombinatorische Schaltung zur Addition von zwei 1 bit Eingängen (A und B). Das 2 bit Ergebnis wird auf die beiden Signale S und C aufgeteilt. Implementieren Sie einen Halbaddierer als Verhaltensbeschreibung in SystemVerilog.

2. Ein Volladdierer ist eine kombinatorische Schaltung zur Addition von drei 1 bit Eingängen (A, B und C_{in}). Das 2 bit Ergebnis wird auf die beiden Signale S und C_{out} aufgeteilt. Implementieren Sie einen Volladdierer in SystemVerilog. Verwenden Sie dafür zwei Halbaddierer.

3. Implementieren Sie den 2 bit Addierer als Strukturbeschreibung in SystemVerilog. Verwenden Sie dafür einen Halb- und einen Volladdierer.

c) Implementieren Sie den 2 bit Addierer als Verhaltensbeschreibung in SystemVerilog.

Übung 12.2 Pipelining – Timing-Bedingungen [20 min]

Folgender SystemVerilog Code beschreibt die kombinatorische Schaltung $Y = A + (\overline{A \oplus D})\ C + \overline{B}$ zwischen zwei Registern im Modul `base`, sowie die dazugehörige funktionale Verifikation in der Testbench `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_ff @(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});
  xor_gate      g1  ({a,d},     n1);
  inv_gate      g2  ( n1,       n2);
  and_gate      g3  ({n2,c},    n3);
  inv_gate      g4  ( b,        n4);
  or_gate  #(3) g5  ({a,n3,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 #4.8 clk = ~clk;

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

```
8
9   localparam L = 2;
10  logic [L-1:0] e;
11
12  always @(posedge clk) begin
13    e <= {e[L-2:0], a | ~(a^d)&c | ~b};
14  end
15
16  initial begin
17    $dumpfile("base_tb.vcd");
18    $timeformat(-9, 2, " ns", 10);
19    $dumpvars;
20
21    for (int i=0; i<16+L; i++) begin
22      #1 {a,b,c,d} <= i;
23      if (y!==e[L-1]) $display("%t: expected %0d but got %0d",$realtime,e[L-1],y);
24      @(posedge clk);
25    end
26
27    $display("FINISHED base_tb");
28    $finish;
29  end
30
31 endmodule
```

Übung 12.2.1 Timing-Analyse

Die Parameter der Registerimplementierung (`tsetup`, `thold` und `tcq`) beschreiben die Setup-, Hold- und Verzögerungszeiten (mit $t_{pcq} = t_{ccq}$) in Nanosekunden. Mit welcher Frequenz kann das `base` Modul theoretisch maximal getaktet werden ohne die Timing-Bedingungen zu verletzen? Welche Latenz hat das Modul?

Übung 12.2.2 Testbench-Analyse

Mit welcher Frequenz wird die Schaltung in der Testbench getaktet? Warum entdeckt die Testbench keine funktionalen Fehler, obwohl die theoretischen Timing-Bedingungen der Register im `base` Modul verletzt werden?

Übung 12.2.3 Überprüfen von Setup- und Hold-Bedingung

Erweitern Sie die Registerimplementierung so, dass die Timing-Bedingungen t_{setup} und t_{hold} automatisch überprüft werden. Dazu bietet es sich an, die Zeitpunkte der Änderungen am Dateneingang und der steigenden Taktflanken in entsprechenden **`always`** Blöcken zu kontrollieren. Bei einer Verletzung der Bedingungen soll eine entsprechende Meldung ausgegeben werden.

Übung 12.2.4 Zusätzliche Pipeline-Stufen

Modifizieren Sie das base Modul durch Einführen zusätzlicher Pipeline-Stufen so, dass es mit Taktperiodendauer 4,2 ns betrieben werden kann, was einer Taktfrequenz von 238 MHz entspricht. Die verwendeten Logikgatter sollen dabei nicht verändert werden. Modifizieren Sie auch die Testbench so, dass das schnellere Modul korrekt getestet wird. Wie wirkt sich diese Modifikation auf die Latenz der Schaltung aus?

Übung 12.3 Strukturierung sequentieller 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* mittels nebenläufiger Zuweisungen realisieren (beschreibt die Transfer-Logik)

arith/mul/sequential.sv

```
module mul4x4 (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
```

Übung 12.4 Barrel-Shifter [15 min]

In der Vorlesung wurden sogenannte Barrel-Shifter behandelt. Wir betrachten ergänzend die “Rotate Left” Variante (umlaufender Linksshift). Anders als die “Arithmetic” Variante aus der Vorlesung lässt der rotierende Shifter nach links “raus geschobene” Bits nicht fallen, sondern fügt diese am anderen Ende des zu verschiebenden Wortes wieder ein.

a) Erstellen Sie eine Verhaltensbeschreibung in SystemVerilog. Der Parameter *SIZE* in der gegebenen Schnittstelle bestimmt die Anzahl an Steuersignalen. (Hinweis: Die minimale Lösung besteht aus einer Zeile.)

comb/barrel/Barrel_Funct.sv

```
1 module functional #(parameter SIZE=2)
2                     (input  logic [SIZE-1:0]      S,
3                      input  logic [(2**SIZE)-1:0] I,
4                      output logic [(2**SIZE)-1:0] O);
```

b) Erstellen Sie eine Strukturbeschreibung in SystemVerilog basierend auf Multiplexern. Anstelle von Multiplexermodulen soll der ternäre Operator zur Instantiierung verwendet werden.

comb/barrel/Barrel_Struct.sv

```
1 module structural #(parameter SIZE=2)
2                     (input  logic [SIZE-1:0]      S,
3                      input  logic [(2**SIZE)-1:0] I,
4                      output logic [(2**SIZE)-1:0] O);
```

c) Erstellen Sie eine Testbench (muss nicht selbstprüfend sein), die beide Implementierungen für einen von Ihnen gewählten Parameter *SIZE* mittels eines aussagekräftigen Bitmusters für alle möglichen Schiebewerte *S* testet.