Digitaltechnik
Wintersemester 2021/2022
11. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Dr.-Ing. Thomas Schneider, M.Sc. Daniel Günther, M.Sc. Amos Treiber

LÖSUNGSVORSCHLAG **KW04**

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 11.1 Pipelining – Register Transfer Logik [20 min]

Folgende Grafik zeigt eine Pipeline zur Berechnung der Funktion $(x^2+5)\cdot 2-8$:

Mittels eines Eingangsignals **set** wird signalisiert, dass mit einer steigenden Taktflanke ein neuer Wert in das erste Register der Pipeline geladen werden soll. Der in R5 gespeicherte Wert entspricht dem Ergebnis der Funktion.

a) Setzen Sie die Pipeline in SystemVerilog um. Erweitern Sie dafür den unten gegebenen Quelltext für das Pipeline-Modul und entwerfen Sie zusätzliche Module zum Berechnen der Funktion jeder Pipeline-Stufe.

Das Verwenden der arithmetischen Operatoren von SystemVerilog ist in den funktionalen Zusatzmodulen erlaubt. Die Setup/Hold-Zeiten der Register sowie Überläufe können vernachlässigt werden.

seq/pipeline/pipeline.sv

```
1  module pipeline (input logic clock, set, input logic [7:0] in,
2                   output logic [7:0] out);
3
4    // Register zum puffern der Werte zwischen den Pipeline-Stufen
5    logic [7:0] register1, register2, register3, register4, register5;
6
7    // Ausgänge der einzelnen Rechenoperationen
8    logic [7:0] out1, out2 , out3 , out4;
9
10   // Rechnungen innerhalb der einzelnen Pipeline-Stufen
11   stufe1 st1(register1, out1);
12   stufe2 st2(register2, out2);
13   stufe3 st3(register3, out3);
14   stufe4 st4(register4, out4);
15
16   // Setzen des Eingangs der Pipeline
17   always_ff @(posedge clock)
18     if(set)
19       register1 <= in;
20     else
21       register1 <= register1;
22
23   // Fortschreiten der Pipeline
24   always_ff @(posedge clock)
```

```
  begin
    register2 <= out1;
    register3 <= out2;
    register4 <= out3;
    register5 <= out4;
  end

  // Ausgang
  assign out = register5;

endmodule

// Quadrieren
module stufe1 (input [7:0] in, output [7:0] out);
  assign out = in * in;
endmodule

// Addition von 5
module stufe2 (input [7:0] in, output [7:0] out);
  assign out = in + 8'd5;
endmodule

// Multiplikation mit 2
module stufe3 (input [7:0] in, output [7:0] out);
  assign out = in * 2;
endmodule

// Substraktion von 8
module stufe4 (input [7:0] in, output [7:0] out);
  assign out = in - 8'd8;
endmodule
```

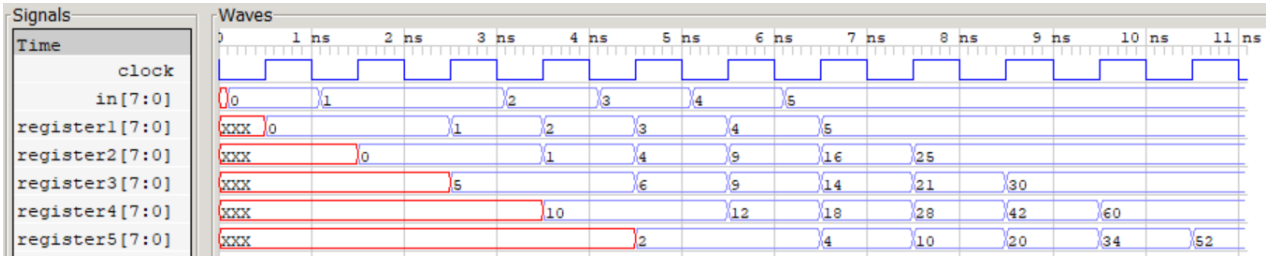
b) Erstellen Sie eine Testbench, um die Funktion der Pipeline anhand von einigen Werten per Simulation zu testen.

seq/pipeline/pipeline_tb.sv

```
`timescale 1 ns / 10 ps

module pipeline_tb;
  logic clock = 0;
  logic set = 0;
  logic [7:0] in, out;

  always  #(0.5) clock <= ~clock;

  // Das zu testende Modul (Unit-Under-Test (UUT))
  pipeline uut(clock, set, in, out);

  initial begin
    $dumpfile("pipeline_tb.vcd");
    $timeformat(-9, 5, " ns", 10);
    $dumpvars;

      #0.1;
          in = 0;
          set = 1;
      #1; // 1 Takt warten
          set = 0; // => Wert sollte diesen Takt nicht übernommen werden
          in = 1;
          #1; // 1 Takt warten
```

```
25            set = 1;
26            #1; // 1 Takt warten
27            in = 2;
28            #1; // 1 Takt warten
29            in = 3;
30            #1; // 1 Takt warten
31            in = 4;
32            #1; // 1 Takt warten
33            in = 5;
34            #5; // 5 Takte warten
35
36      $display("FINISHED pipeline_tb");
37      $finish;
38    end
39
40  endmodule
```

Der Takt schaltet alle 0,5 ns um, was eine steigende Taktflanke alle 1 ns zur Folge hat. Anhand des folgenden Timing-Diagramms erkennt man gut, wie die Werte vom Eingang der jeweiligen Pipeline-Stufe nach jedem Takt durch die Pipeline weitergereicht werden:



c) Nehmen Sie nun an, dass die t_{pcq}- sowie t_{setup}-Zeit der Register bei 0,5 ns liegt und die kombinatorischen Schaltungen für die Rechnungen zwischen den Registern kritische Pfade mit folgenden Verzögerungen haben: x^2: 0,6 ns, +5: 0,8 ns, ×2: 0,5 ns, −8: 1 ns. Mit welcher Taktfrequenz lässt sich die Pipeline maximal betreiben? Welche Frequenz wäre möglich, wenn man auf Register 2 und 4 verzichtet?

Der längste kritische Pfad zwischen zwei Registern benötigt 1 ns. Berücksichtigt man noch $2 \cdot 0{,}5\,\text{ns}$ aufgrund von t_{pcq} und t_{setup}, so ergibt dies eine minimale Taktperiodendauer von 2 ns. Folglich lässt sich die Schaltung mit maximal $\frac{1}{2\,\text{ns}} = 500\,\text{MHz}$ betreiben.

Ohne die Register 2 und 4 würde der längste kritische Pfad zwischen zwei Registern $0{,}5\,\text{ns} + 1\,\text{ns} = 1{,}5\,\text{ns}$ benötigen. Zuzüglich t_{pcq} und t_{setup} ist eine Taktfrequenz von $\frac{1}{2{,}5\,\text{ns}} = 400\,\text{MHz}$ erlaubt.

d) Welcher Vor- und welcher Nachteil ergibt sich hauptsächlich aus der Verwendung von vielen Pipeline-Stufen?

Viele Pipeline-Stufen erlauben oft starke Erhöhungen der Taktfrequenz und steigern somit den Durchsatz der Schaltung. Allerdings steigern viele Pipeline-Stufen fast immer die Latenz, also die Zeit von der Eingabe eines Werts in die Pipeline bis zur Ausgabe des korrespondierenden Ergebnisses.

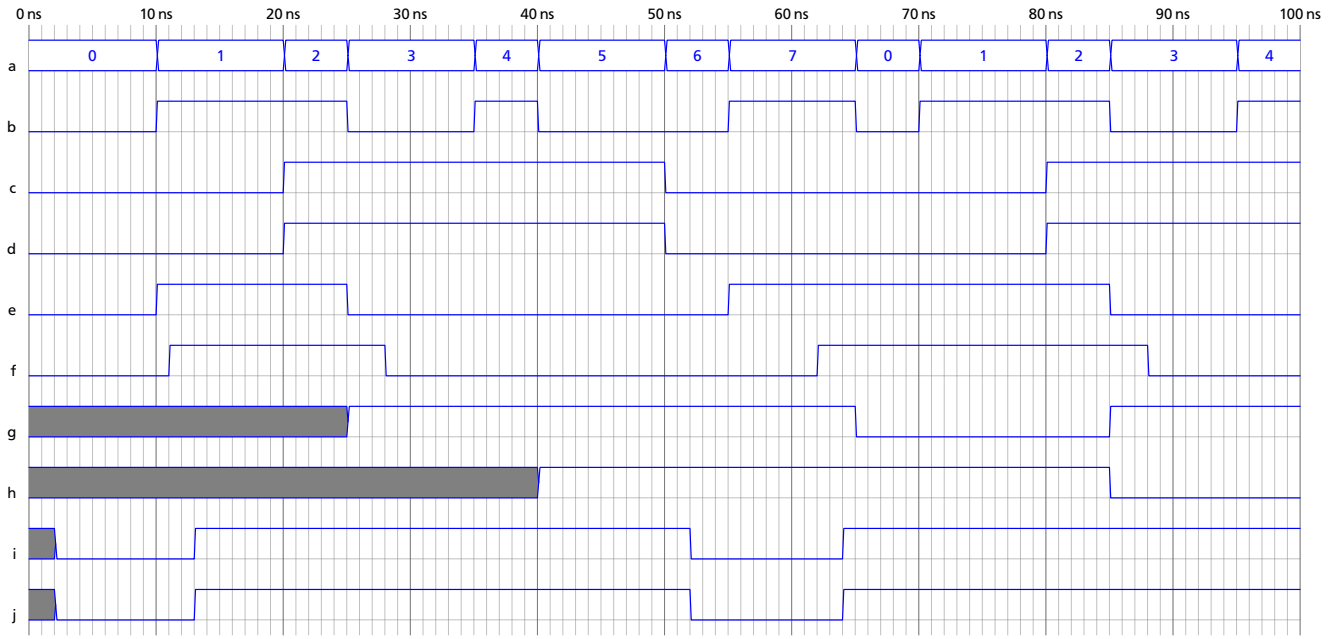
Übung 11.2 Zeitverhalten sequentieller Beschreibungen [15 min]

Simulieren Sie das Verhalten der nachfolgenden Signale für die ersten 100 ns. Bedenken Sie, 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-blockierenden Zuweisung geprüft!



Übung 11.3 Parametrisierte Moduldefinition [15 min]

In dieser Aufgabe soll ein XOR Gatter mit einer variablen Anzahl an Eingängen realisiert werden.

a) Implementieren Sie ein funktionales Modul (Verhaltensbeschreibung) zu folgender Schnittstelle:

comb/xor/Parameter_Xor_Funct.sv

```
1  module xor_functional #(parameter SIZE = 2)
2                         (input  logic [SIZE-1 : 0] I,
3                          output logic              O);
4     assign O = ^I;
5  endmodule
```

b) Realisieren Sie ein äquivalentes strukturelles Modul (Strukturbeschreibung) zu nachfolgender Schnittstelle. Nutzen Sie dafür eine **for** Schleife um eine variable Anzahl an Zuweisungen zu generieren.

comb/xor/Parameter_Xor_Struct.sv

```
module xor_structural #(parameter SIZE = 2)
                       (input  logic [SIZE-1 : 0] I,
                        output logic              O);

    logic [SIZE-2 : 0] B;
    assign O = B[SIZE-2];
    genvar i;
    generate
        for (i = 0; i < SIZE-1; i = i+1) begin
            if(i == 0)
                assign B[0] = I[0] ^ I[1];
            else
                assign B[i] = I[i+1] ^ B[i-1];
        end
    endgenerate

endmodule
```

c) Entwickeln Sie eine selbsttestende Testbench für beide Module mit Größe 4.

comb/xor/Parameter_Xor_Tb.sv

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

  logic clk = 0;
  always #1 clk <= ~clk;

  logic rst = 1;
  initial @(posedge clk) rst <= 0;

  logic [3:0] a;
  logic       yF, yS;

  xor_functional #(4) uut_f (a, yF);
  xor_structural #(4) uut_s (a, yS);

  initial begin
    $dumpfile("param_xor_tb.vcd");
    $timeformat(-9, 0, " ns", 8);
    $dumpvars;

    for (int i = 0; i < 16; i = i + 1) begin

        a <= i;
        @(posedge clk);

        if(yF != ^a)
          $display("%t: Functional Unit wrong. Expected %d but was %d for %d",
            $time, ^a, yF, a);

        if(yS != ^a)
          $display("%t: Structural Unit wrong. Expected %d but was %d for %d",
            $time, ^a, yS, a);

```

```
34      end
35
36      $display("FINISHED param_xor_tb");
37      $finish;
38    end
39  endmodule
```