Digitaltechnik
Wintersemester 2021/2022
12. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

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

LÖSUNGSVORSCHLAG **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:

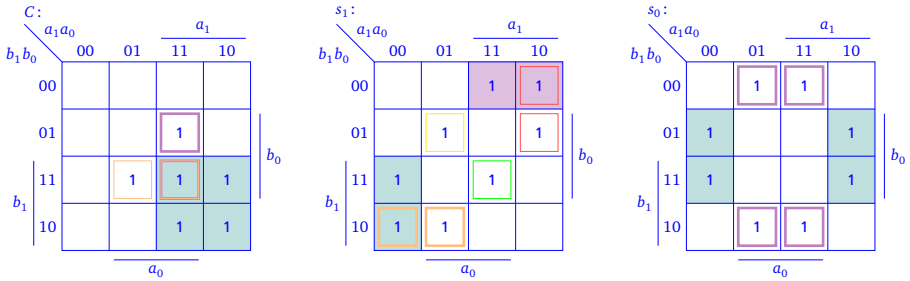
- Inputs:
  - 2 bit breite Zahlen A und B ($A := a_1a_0$, $B := b_1b_0$)
- Outputs:
  - 2 bit breite Summe S von A und B ($S := s_1s_0$)
  - Übertrag C

a) Implementieren Sie den 2 bit Addierer mit Basisgattern:

1. Stellen Sie die Wahrheitstabelle für den 2 bit Addierer auf.

a_1	a_0	b_1	b_0	C	s_1	s_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

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

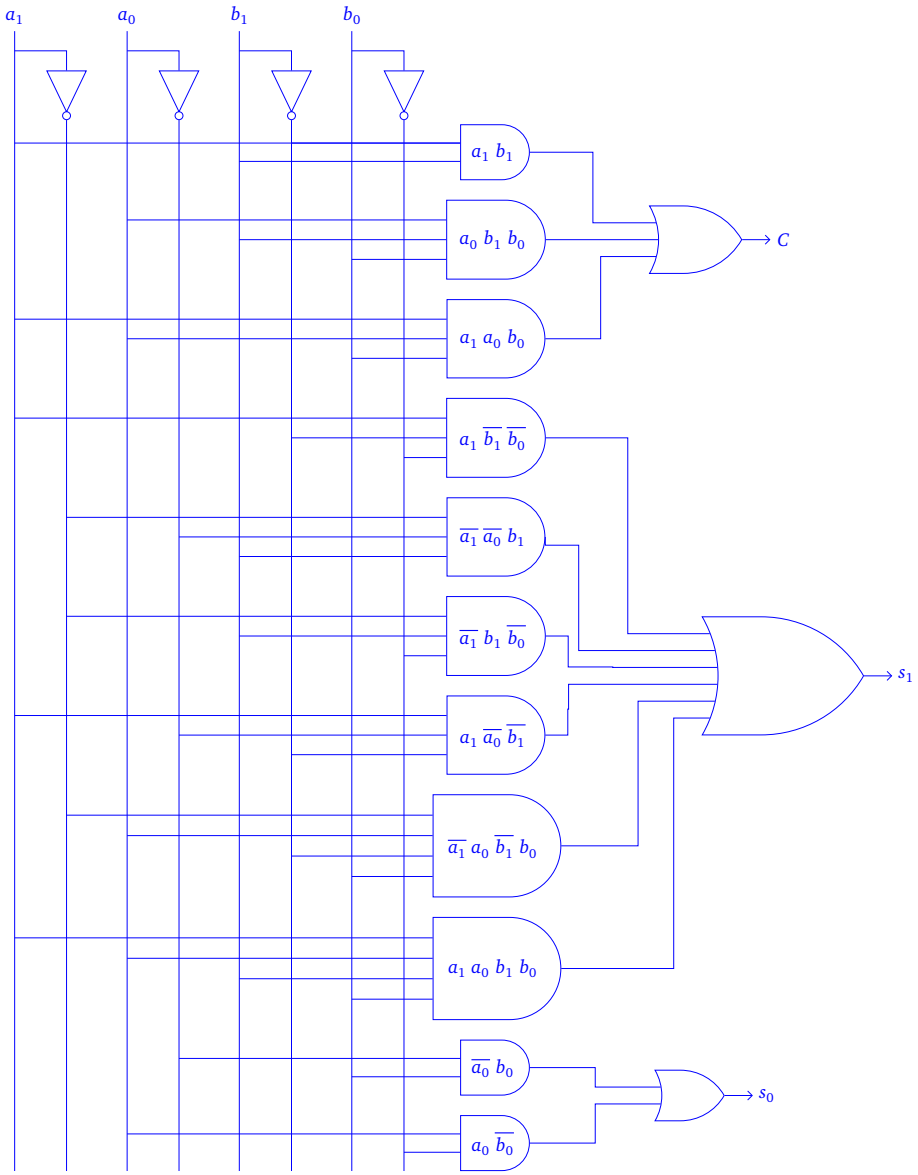


$$C = a_1\ b_1 + a_0\ b_1\ b_0 + a_1\ a_0\ b_0$$

$$s_1 = a_1\ \overline{b_1}\ \overline{b_0} + \overline{a_1}\ \overline{a_0}\ b_1 + \overline{a_1}\ b_1\ \overline{b_0} + a_1\ \overline{a_0}\ \overline{b_1} + \overline{a_1}\ a_0\ \overline{b_1}\ b_0 + a_1\ a_0\ b_1\ b_0$$

$$s_0 = \overline{a_0}\ b_0 + a_0\ \overline{b_0}$$

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 (&, |, ~).

arith/adder/zweistufig2BitAddierer.sv

```
1  module zweistufig2BitAddierer (
2    input  logic [1:0] A,
3    input  logic [1:0] B,
4    output logic C,
5    output logic [1:0] S);
6
7    assign S[0] =  ( ~A[0] & B[0] ) | (A[0] & ~B[0] );
8
9    assign S[1] =  ( A[1]          & ~B[1] & ~B[0] ) |
10                  (~A[1] & ~A[0] &  B[1]          ) |
```

```
                  (~A[1]          &  B[1] & ~B[0] ) |
                  ( A[1] & ~A[0] & ~B[1]          ) |
                  (~A[1] &  A[0] & ~B[1] &  B[0] ) |
                  ( A[1] &  A[0] &  B[1] &  B[0] );

  assign C    =   (A[1] & B[1]        ) |
                  (A[0] & B[1] & B[0]) |
                  (A[1] & A[0] & B[0]);

endmodule
```

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.

arith/adder/Halbaddierer.sv

```
module Halbaddierer (
  input  logic A, B,
  output logic C, S);

  assign S =  A ^ B;
  assign C =  A & B;

endmodule
```

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.

arith/adder/Volladdierer.sv

```
module Volladdierer (
  input  logic CIN,  A, B,
  output logic COUT, S);

  logic ha1_c, ha1_s, ha2_c;

  Halbaddierer ha1 (.A(A),     .B(B),   .C(ha1_c), .S(ha1_s) );
  Halbaddierer ha2 (.A(ha1_s), .B(CIN), .C(ha2_c), .S(S)     );

  assign COUT = ha1_c || ha2_c;

endmodule
```

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

arith/adder/struktur2BitAddierer.sv

```
module struktur2BitAddierer (
  input logic [1:0] A, B,
  output logic C,
  output logic [1:0] S);

  logic ha_c;

  Halbaddierer ha (           .A(A[0]), .B(B[0]), .C(ha_c), .S(S[0]) );
  Volladdierer fa (.CIN(ha_c), .A(A[1]), .B(B[1]), .COUT(C), .S(S[1]) );

endmodule
```

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

arith/adder/verhalten2BitAddierer.sv

```
module verhalten2BitAddierer (
  input  logic [1:0] A,
  input  logic [1:0] B,
  output logic C,
  output logic [1:0] S);

  assign {C, S} =  A + B;

endmodule
```

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);

  localparam L = 2;
  logic [L-1:0] e;

  always @(posedge clk) begin
    e <= {e[L-2:0], a | ~(a^d)&c | ~b};
  end

  initial begin
    $dumpfile("base_tb.vcd");
    $timeformat(-9, 2, " ns", 10);
    $dumpvars;

    for (int i=0; i<16+L; i++) begin
      #1 {a,b,c,d} <= i;
      if (y!==e[L-1]) $display("%t: expected %0d but got %0d",$realtime,e[L-1],y);
      @(posedge clk);
    end

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

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?

Der kritische Pfad zwischen den Registern `rin` und `rout` verläuft durch die Gatter `g1` (3 ns), `g2` (1 ns), `g3` (2 ns) und `g5` (3 ns). Zusammen mit `tcq` und `tsetup` der Register ergibt dies 10 ns. Das Modul kann daher höchstens mit 100 MHz getaktet werden. Da das Ergebnis nach einem Takt an der finalen Registerstufe anliegt, ergibt sich eine Latenz von 10 ns.

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?

Das `clk` Signal schaltet alle 4,8 ns um, wodurch alle 9,6 ns eine steigende Taktflanke und somit eine Taktfrequenz von 104 MHz erzeugt wird. Der Eingang des zweiten Registers `rout` ist dadurch erst 0,5 ns vor der steigenden Taktflanke stabil und nicht wie durch `tsetup` gefordert 0,9 ns. Dies verletzt zwar die Setup-Bedingung, die Register-Implementierung liest den Dateneingang aber 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-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.

seq/pipeline/register.sv

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

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 prüfen (Zeile 27). Umgekehrt kann bei jeder Änderung des Dateneingangs bestimmt werden, wie lange die letzte steigende Taktflanke bereits zurück liegt, um so die Hold-Bedingung zu prüfen (Zeile 20).

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?

Aufgrund der Taktfrequenzvorgabe darf der kritische Pfad in jeder Pipeline-Stufe maximal eine Ausbreitungsverzögerung von $4{,}2\,\text{ns} - t_{\text{pcq}} - t_{\text{setup}} = 3{,}2\,\text{ns}$ aufweisen.

seq/pipeline/fast.sv

```
1  module fast (input logic CLK, A, B, C, D, output logic Y);
2    logic a,b,c,d,n1,n2,n3,n4,n5,n6,n7,n1r,n3r,n4r,n6r,n7r,y;
3    register #(4) rin (CLK, {A,B,C,D}, {a,b,c,d});
4    xor_gate      g1  ({a,d},      n1);
5    inv_gate      g2  ( b,         n2);
6    or_gate       g3  ({a,n2},     n3);
7    and_gate      g4  ({c,c},      n4); // Verzögerung/Buffer für Hold-Bedingung rp1
8    register #(3) rp1 (CLK, {n1,n3,n4}, {n1r,n3r,n4r});
9    inv_gate      g5  ( n1r,       n5);
10   and_gate      g6  ({n5,n4r},   n6);
11   and_gate      g7  ({n3r,n3r},  n7); // Verzögerung/Buffer für Hold-Bedingung rp2
12   register #(2) rp2 (CLK, {n6,n7}, {n6r, n7r});
13   or_gate       g8  ({n6r,n7r},  y );
14   register      rout(CLK, y, Y);
15 endmodule
```

Zum Anpassen der Testbench muss die `clk` Toggle-Periode auf 2,1 ns gesetzt (Zeile 5), das `fast` Modul instantiiert (Zeile 7) und die Anzahl der Pipeline-Stufen auf 4 erhöht werden (Zeile 9).

Da nun 3 Takte mit einer Periodendauer von 4,2 ns benötigt werden, erhöht sich die Latenz auf 12,6 ns.

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
```

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

Eine kontrollflusslastige Beschreibung (sämtliche Logik in einem **always_ff** Block) spiegelt den zugrundeliegenden Algorithmus meist besser wider und kommt einer Software-Implementierung nahe. Die RTL-nahe Beschreibung (möglichst viele aber dafür kleine **always_ff** Blöcke) spiegelt die parallel arbeitenden Hardware-Komponenten und die Abhängigkeiten dazwischen besser wider 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 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

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

    assign O = {I, I} >> ((2**SIZE) - S);
endmodule
```

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

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

    logic [(2**SIZE)-1:0] B [SIZE:0];
    assign B[0] = I;
    assign O = B[SIZE];

    genvar i;
    generate

        for (i = 0; i < SIZE; i = i + 1)
            assign B[i+1] = S[i] ? { B[i][(2**SIZE)-(2**i)-1:0],
                                     B[i][(2**SIZE)-1:(2**SIZE)-(2**i)] }
                                 : B[i];

    endgenerate
endmodule
```

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.

comb/barrel/Barrel_Tb.sv

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

    logic CLK = 0;
    logic [15:0] I, OF, OS;
    logic [3:0]  S;

    functional #(4) uut_functional(S, I, OF);
    structural #(4) uut_structural(S, I, OS);

    always #(0.5/0.02) CLK <= ~CLK;

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

        I <= 16'b0100110011101111;
        S <= 0;

        for (int i = 0; i < 16; i++) begin
            @(posedge CLK);
            S <= S + 1;
        end

        $finish;
    end

endmodule
```