Technische Universität Darmstadt
FG Eingebettete Systeme und
ihre Anwendungen (ESA)

Prof. Dr. Andreas Koch
Holger Lange

16.05.2007

Einführung in Computer Microsystems
3. Aufgabenblatt
Sommersemester 2007

Aufgabe 1: Ringzähler

Gegeben ist folgendes Verilog-Modul, welches einen Ringzähler realisiert:

```
`timescale 1ns / 1ps
module ring_counter(COUNT, ENABLE, RESET, CLK);
    output reg [7:0] COUNT;
    input ENABLE;
    input RESET;
    input CLK;

  always @(posedge RESET or posedge CLK)
    if (RESET == 1'b1)  COUNT <= 8'b0000_0001;
    else if (ENABLE == 1'b1)  COUNT <= {COUNT[6:0], COUNT[7]};

endmodule
```

a) Simulieren und prüfen Sie alle Funktionen des Moduls mit Hilfe einer Testbench. Geben Sie den Schaltplan (Schematic) des Moduls auf Gatterebene an (Hinweis: ISE kann Ihnen dabei behilflich sein).

Lösung:

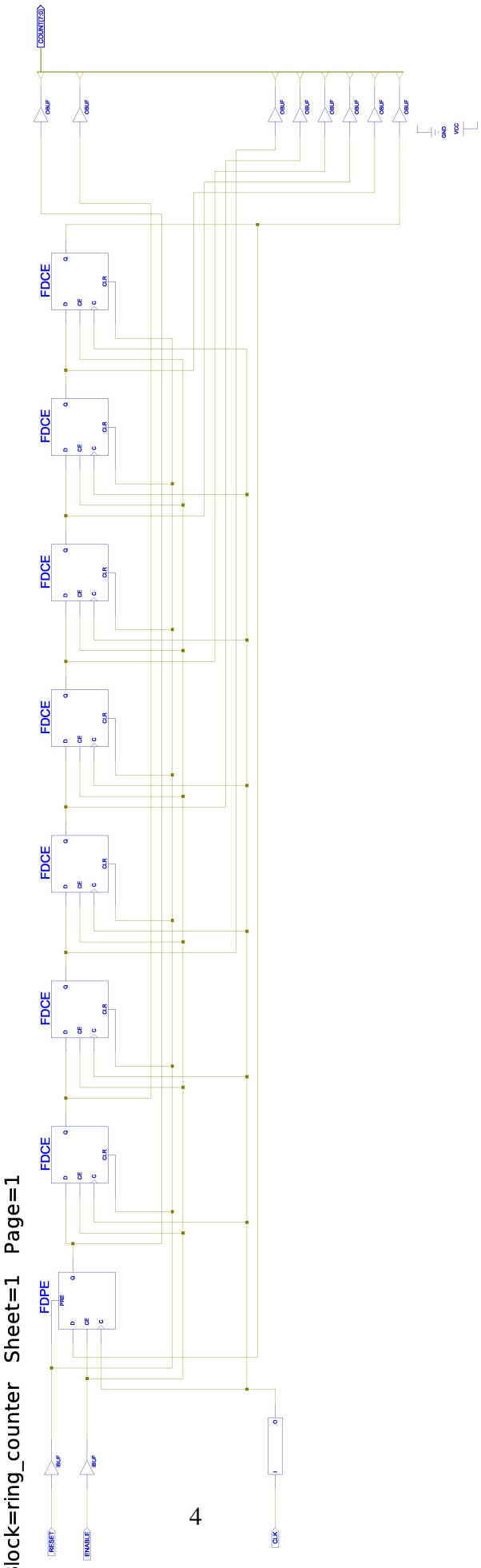
Testbench:

```
`timescale 1ns / 1ps
`define HALF_CYCLE 10

module tb_ring_counter_v;
   // Inputs
   reg ENABLE;
```

```
reg RESET;
reg CLK;

// Outputs
wire [7:0] COUNT;

// Instantiate the Unit Under Test (UUT)
ring_counter uut (
   .COUNT(COUNT),
   .ENABLE(ENABLE),
   .RESET(RESET),
   .CLK(CLK)
);

task DoClockCycle;
   begin
      #`HALF_CYCLE;
      CLK = 1;
      #`HALF_CYCLE;
      CLK = 0;
   end
endtask

task RunFor;
  input integer n;
  integer i;

  for (i = 0; i < n; i = i + 1)
     DoClockCycle;
endtask

initial begin
   // Initialize Inputs
   ENABLE = 0;
   RESET = 0;
   CLK = 0;

   // Wait 100 ns for global reset to finish
   #100;

   // Add stimulus here
   RESET = 1;
   DoClockCycle;
   RESET = 0;
   RunFor(5);
   ENABLE = 1;
   RunFor(6);
   ENABLE = 0;
   RESET = 1;
   DoClockCycle;
   RESET = 0;
   ENABLE = 1;
   while (COUNT != 'h10)
      DoClockCycle;
   ENABLE = 0;
```

```
        RunFor(5);
    end

endmodule
```

Schematic:


Block=ring_counter Sheet=1 Page=1
COUNT[7:0]
GND
VCC
FDCE
FDPE
RESET
ENABLE
CLK

b) Erweitern Sie das Modul aus a) um einen zusätzlichen Eingang **DOWN**, der die Richtung des Ringzählers umkehrt. Der Zähler zählt dann bei **DOWN**=1 von MSB (Most Significant Bit) nach LSB (Least Significant Bit), bei **DOWN**=0 wie bisher. Simulieren und prüfen Sie erneut alle Funktionen des erweiterten Moduls mit Hilfe einer angepassten Testbench. Geben Sie den Schaltplan (Schematic) des Moduls auf Gatterebene an. Was hat sich im Vergleich zu a) geändert? Markieren Sie für ein Zähler-Bit die Signalwege beider Zählrichtungen farbig im Schematic.

Lösung:

Erweitertes Modul:

```
`timescale 1ns / 1ps
module ring_counter_bidir(COUNT, ENABLE, DOWN, RESET, CLK);
    output reg [7:0] COUNT;
    input DOWN, ENABLE;
    input RESET;
    input CLK;

  always @(posedge RESET or posedge CLK)
    if (RESET == 1'b1)  COUNT <= 8'b0000_0001;
    else if (ENABLE == 1'b1)
         if (DOWN)  COUNT <= {COUNT[0], COUNT[7:1]};
         else       COUNT <= {COUNT[6:0], COUNT[7]};

endmodule
```

Testbench:

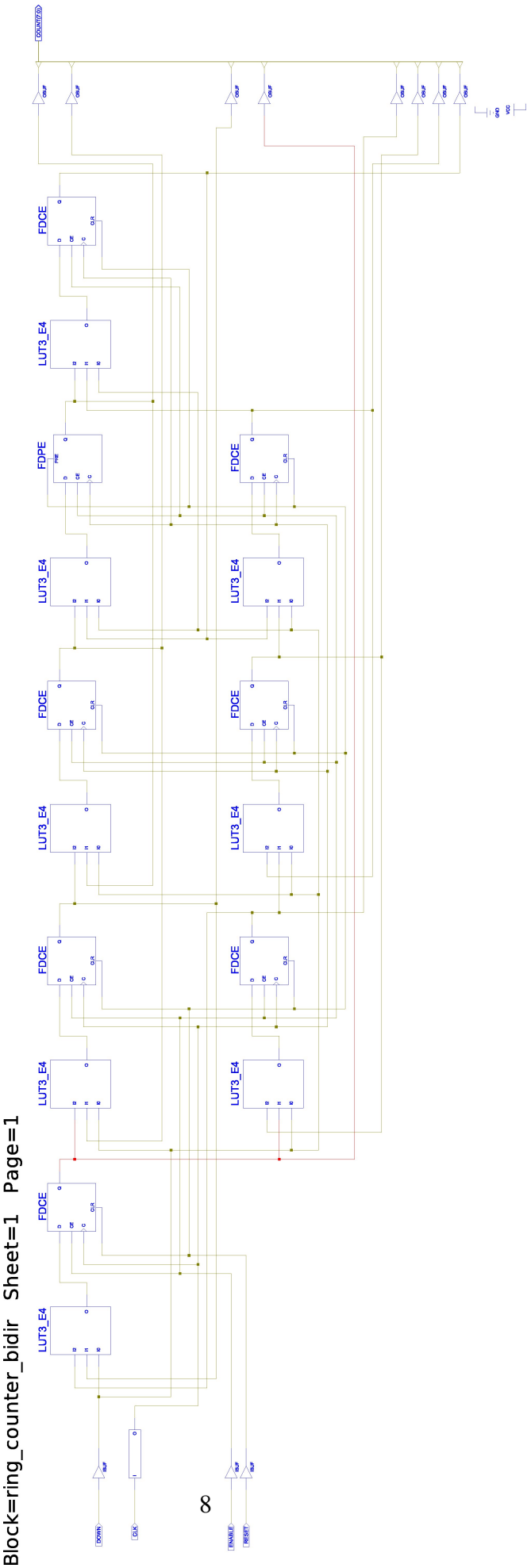
```
`timescale 1ns / 1ps
`define HALF_CYCLE 10

module tb_ring_counter_bidir_v;
   // Inputs
   reg ENABLE;
   reg DOWN;
   reg RESET;
   reg CLK;

   // Outputs
   wire [7:0] COUNT;

   // Instantiate the Unit Under Test (UUT)
   ring_counter_bidir uut (
      .COUNT(COUNT),
      .ENABLE(ENABLE),
      .DOWN(DOWN),
      .RESET(RESET),
      .CLK(CLK)
   );

   task DoClockCycle;
     begin
       #`HALF_CYCLE;
       CLK = 1;
       #`HALF_CYCLE;
       CLK = 0;
```

```
  end
endtask

task RunFor;
  input integer n;
  integer i;

  for (i = 0; i < n; i = i + 1)
    DoClockCycle;
endtask

initial begin
   // Initialize Inputs
   ENABLE = 0;
   DOWN = 0;
   RESET = 0;
   CLK = 0;

   // Wait 100 ns for global reset to finish
   #100;

   // Add stimulus here
   // Erster Testabschnitt wie bisher
   RESET = 1;
   DoClockCycle;
   RESET = 0;
   RunFor(5);
   ENABLE = 1;
   RunFor(6);
   ENABLE = 0;
   RESET = 1;
   DoClockCycle;
   RESET = 0;
   ENABLE = 1;
   while (COUNT != 'h10)
      DoClockCycle;
   ENABLE = 0;
   RunFor(5);

   // Test wiederholen mit DOWN = 1
   DOWN = 1;
   RESET = 1;
   DoClockCycle;
   RESET = 0;
   RunFor(5);
   ENABLE = 1;
   RunFor(6);
   ENABLE = 0;
   RESET = 1;
   DoClockCycle;
   RESET = 0;
   ENABLE = 1;
   while (COUNT != 'h10)
      DoClockCycle;
   ENABLE = 0;
```

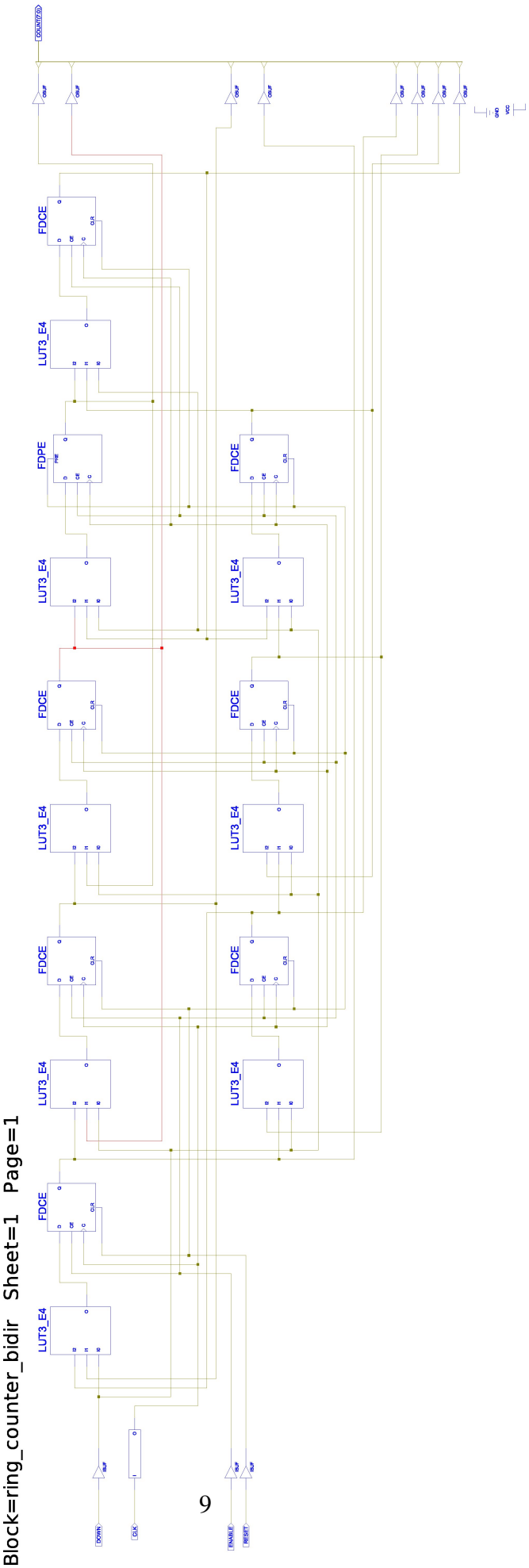
```
        RunFor(5);
    end

endmodule
```

Zwischen den Flip-Flops des Ringzählers aus a) sind nun zusätzliche Lookup-Tables (LUT) angeordnet. Verfolgt man die Signalwege, wird deutlich, dass die LUTs als Multiplexer fungieren, die zwischen den Zählrichtungen auswählen. Mit einem Doppelklick auf ein LUT (oder dem Menüpunkt **`Show LUT Content...`**) lässt sich die logische Funktion anzeigen, welche das LUT in der aktuellen Konfiguration implementiert (beliebige Funktionen aus bis zu vier Eingängen sind möglich). Der Eingang **`I0`** ist in diesem Fall der Select-Eingang des Multiplexers (angeschlossen an **`DOWN`**), welcher zwischen den Eingängen **`I1`** und **`I2`** auswählt.

Schematic mit markiertem Signalpfad count down:



Schematic mit markiertem Signalpfad count up:



Aufgabe 2: Fehlersuche

Gegeben ist folgendes Verilog-Modul, das einige Fehler enthält:

```
module faulty(A, B, Y, Z);
    input A;
    input [1:0] B;
    output reg Y;
    output [1:0] Z;

    reg A;

    always @(A)
      Z = A & B[0];

    always @(B)
      Z = B[1] & Y;

    always @(A)
      Y = B | A;

endmodule
```

a) Finden Sie die Fehler in der Verilog-Verhaltensbeschreibung. Vergleichen Sie dazu auch die Funktion des Moduls bei der Verhaltenssimulation mit der Post-Layout Simulation und den RTL-Schematics. Machen Sie Verbesserungsvorschläge und begründen Sie diese.

Lösung:

1. Zeile 7 muss **`wire`** statt **`reg`** sein, da **`A`** ein Eingang ist. Die Zeile kann auch ganz wegfallen, da Wires implizit deklariert sind, wenn nichts anderes angegeben wird.

2. Die Deklaration in Zeile 5 legt **`Z`** implizit als **`wire`** fest. Da **`Z`** aber ein Wert in einem prozeduralen Block zugewiesen wird, muss es als **`reg`** deklariert werden.

3. Dem Register **`Z`** wird im ersten *und* zweiten **`always`**-Block ein Wert zugewiesen. In der Simulation werden die Zuweisungen abhängig von den Änderungen von entweder A oder B ausgeführt. Ändern sich A und B gleichzeitig, ist die Ausführungsreihenfolge unbestimmt. Bei einer Hardware-Synthese wären für diese Funktionalität Latches und flankengesteuerte Multiplexer notwendig, diese funktionieren im Allgemeinen unzuverlässig (Hazards!) und sind auf FPGAs gar nicht vorhanden. Folglich lässt sich keine Hardware für das beschriebene Verhalten generieren.

4. Bei allen **`always`**-Blöcken ist die Sensitivity-Liste nicht komplett. Die Simulation führt die Blöcke nur dann aus, wenn sich ein Signal in der Liste ändert (exakt so wie im Code), wohingegen das Hardware-Synthesewerkzeug von ISE immer das Vorhandensein einer vollständigen Liste annimmt. Anderenfalls müssten in der Hardware Latches erzeugt werden, um die alten Werte zwischenzuspeichern, wenn sich ein nicht in der Sensitivity-Liste aufgeführter Wert ändert. Simulation und Hardware stimmen somit ungewollt *nicht* überein.

5. Z[1] ist immer 0, kann daher wegfallen, falls nicht anderweitig verwendet.

b) Wieviele Register (Flip-Flops) werden bei der Übersetzung des Moduls in Hardware erzeugt?

Lösung:

Es werden *keine* Flip-Flops erzeugt. Ob bei der Synthese ein Flip-Flop generiert wird, hängt nicht davon ab, ob in Verilog ein `reg` deklariert wurde. Es kommt auf die Art der Verwendung an: Wird einem `reg` in einem *synchronen* Block (`always @(posedge CLK) ...`) ein Wert zugewiesen, so entsteht ein Flip-Flop in Hardware. Erfolgt die Zuweisung in einem *kombinatorischen* Block (`always @(A, B, ...) ...`) ohne Clock, so wird nur kombinatorische Logik generiert. Ein `wire` hingegen erzeugt immer rein kombinatorische Logik.