Übung zur Vorlesung Einführung in Computer Microsystems

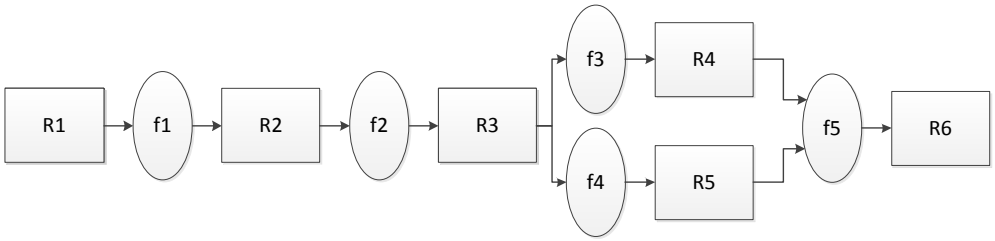
Prof. Dr. A. Koch
Thorsten Wink

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 11
Übungsblatt 2 - Lösungsvorschlag

Aufgabe 2.1 Register-Transfer-Logik

Gegeben ist folgende Pipeline in Register-Transfer-Logik:



Die kombinatorische Logik zwischen den Registern soll folgende Funktionen realisieren.

- f_1: verdoppeln
- f_2: plus 5
- f_3: quadrieren
- f_4: minus 3
- f_5: Wenn R4 gerade, dann leite R4 weiter, ansonsten R5

Aufgabe 2.1.1 Funktionen

Beschreiben Sie die Pipeline in Verilog HDL. Die Funktionen sollen in Verilog HDL als `function` realisiert werden.
Die Register R1, R2, R3, R4, R5 und R6 sind jeweils 8-Bit breit. Überlaufen der Register kann vernachlässigt werden. Das Register R1 soll mit einem Wert geladen werden können. Wenn ein Steuersignal `ldreg1` gesetzt ist, soll über den Port `in` von auSSen ein Wert in das Register R1 geladen werden. Das Ergebnis der Berechnung soll über den Port `out` nach aussen geführt werden. Weisen Sie die korrekte Funktionsweise der Pipeline durch Simulation nach.

Die Pipeline kann entsprechend dem folgenden Listing implementiert werden.

```
module pipeline_a(
        input clock,
        input ldreg1,
        input [7:0] in,
        output [7:0] out
        );

reg [7:0] r1, r2, r3, r4, r5, r6;
```

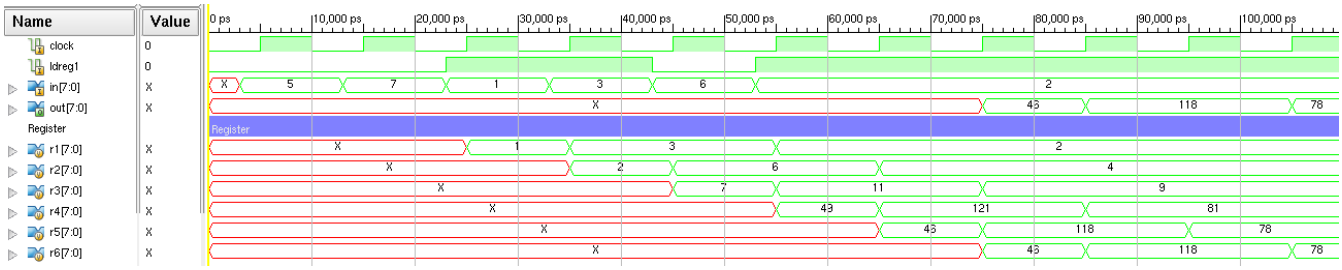
```
erste Funktion: verdoppeln
function [7:0] f1;
input [7:0] arg;
begin
f1 = arg*8'h02;

end
endfunction

// zweite Funktion: plus 5
function [7:0] f2;
input [7:0] arg;
begin
f2 = arg+8'h05;
end
endfunction

// dritte Funktion: quadrieren
function [7:0] f3;
input [7:0] arg;
begin
f3 = arg*arg;
end
endfunction

// vierte Funktion: subtrahieren
function [7:0] f4;
input [7:0] arg;
begin
f4 = arg-8'h03;
end
endfunction

// fünfte Funktion: Multiplexer
function [7:0] f5;
input [7:0] arg1, arg2;
begin
f5 = arg1[0] ? arg2 : arg1;
end
endfunction

// Die Pipeline
always @(posedge clock)
begin
if (ldreg1)
r1 <= in;
else
r1 <= r1;

r2 <= f1(r1);
r3 <= f2(r2);
r4 <= f3(r3);
r5 <= f4(r4);
```

```
r6 <= f5(r4,r5);
end

// Ausgang zuweisen
assign out = r6;
endmodule
```

Anmerkung: Der Multiplexer wird niemals R4 weiterleiten, R4 ist immer ungerade.
Um die Funktionalität nachzuweisen wird folgende Testbench verwendet.

```
module testbench_a;
        // Inputs
        reg clock;
        reg ldreg1;
        reg [7:0] in;
        // Outputs
        wire [7:0] out;

        // Instantiate the Unit Under Test (UUT)
        pipeline_a uut (
                .clock(clock),
                .ldreg1(ldreg1),
                .in(in),
                .out(out)
        );

        // Simulation
        initial begin
                ldreg1 = 0;
                #3; // vor der ersten positiven Taktflanke
                in = 5;
                #10; // 1 Takt warten
                in = 7;
                #10; // 1 Takt warten
                ldreg1 = 1;             // Wert laden
                in = 1;
                #10; // 1 Takt warten
                in = 3;
                #10; // 1 Takt warten
                ldreg1 = 0;             // Keinen Wert laden
                in = 6;
                #10; // 1 Takt warten
                ldreg1 = 1;             // Wert laden
                in = 2;
        end

        // Takt
        initial begin
                clock = 1'b0;
                forever #5 clock = !clock;
        end
endmodule
```

Die Testbench setzt `in` und `ldreg1` auf verschiedene Werte damit überprüft werden kann ob das Ergebnis korrekt ist und nur bei gesetztem `ldreg1` ein Wert geladen wird. Das Simulationsergebnis ist in der nächsten Abbildung zu sehen. Man sieht wie die Zwischenergebnisse durch die Register der Pipeline “wandern”.



Aufgabe 2.1.2 asynchroner Reset

Die Pipeline aus Aufgabenteil a) soll um einen asynchronen Reset-Eingang `areset` erweitert werden, mit dem die Pipeline zurückgesetzt werden kann. Wird `areset` gesetzt, soll die Berechnung sofort gestoppt werden und solange keine neue Berechnung begonnen werden, bis `areset` wieder den Wert 0 annimmt.

Das Modul kann wie folgt geändert und erweitert werden:

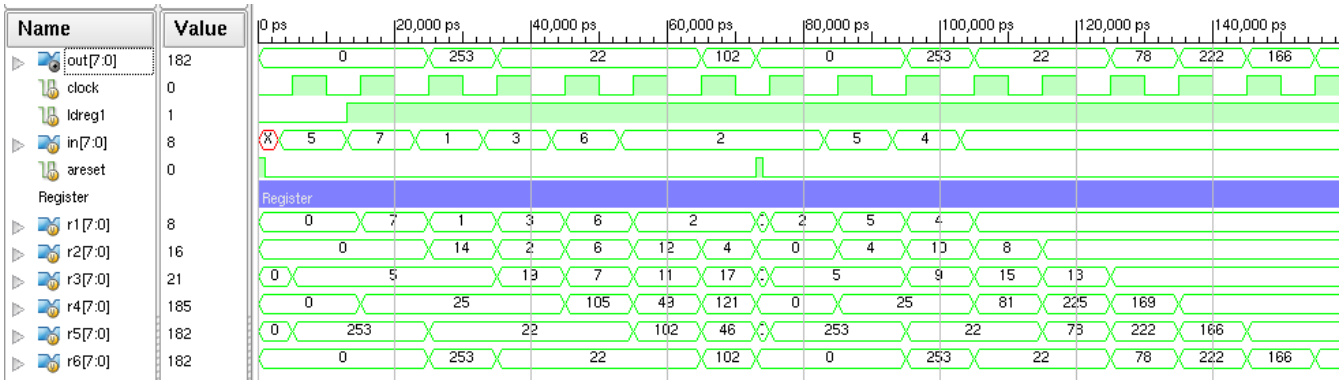
```
module pipeline_b(
         input clock,
         input areset,
         input ldreg1,
         input [7:0] in,
         output [7:0] out
         );

         ...

// Die Pipeline
always @(posedge clock or posedge areset)
if(areset) begin
        r1 <= 8'h00;
        r2 <= 8'h00;
        r3 <= 8'h00;
        r4 <= 8'h00;
        r5 <= 8'h00;
        r6 <= 8'h00;

end else begin
        if (ldreg1)
                r1 <= in;
        else
                r1 <= r1;

        r2 <= f1(r1);
        r3 <= f2(r2);
        r4 <= f3(r3);
        r5 <= f4(r4);
        r6 <= f5(r4,r5);
end

         ...
```

In der Testbench muss `areset` ergänzt werden und der für die Simulation zuständige Initial-Block mit entsprechenden Testfällen erweitert werden.

```
module testbench_b;
```

```
... // Input, Output und UUT

// Simulation
initial begin
        areset = 0;
        ldreg1 = 0;
        areset = 1'b1;  // Pipeline am Anfang zurücksetzen
        #1; // kurz halten
        areset = 1'b0;  // Signal zurücknehmen
        #2; // vor der ersten positiven Taktflanke
        in = 5;
        #10; // 1 Takt warten
        ldreg1 = 1;     // Wert laden
        in = 7;
        #10; // 1 Takt warten
        in = 1;
        #10; // 1 Takt warten
        in = 3;
        #10; // 1 Takt warten
        in = 6;
        #10; // 1 Takt warten
        in = 2;
        #20;
        areset = 1'b1;  // Pipeline zurücksetzen
        #1; // kurz halten
        areset = 1'b0;  // Signal zurücknehmen

        #9; // rest vom Takt warten
        in = 5;
        #10; // 1 Takt warten
        in = 4;
        #10; // 1 Takt warten
        in = 8;
        // usw...
end

...
```

Die Simulation sieht dann so aus:



Aufgabe 2.1.3 Valid

Da die Pipeline einige Takte benötigt um das Ergebnis zu berechnen, liegen am Ausgang zwischenzeitlich falsche Werte an. Das Module aus Aufgabenteil b) soll deshalb um ein Signal `valid` erweitert werden. `valid` soll genau dann den Wert 1 haben, wenn der Wert an `out` ein korrektes Ergebnis eines geladenen Wertes ist.

Das folgende Listing zeigt die Änderungen für das valid-Signal

```
module pipeline_c(
         input clock,
         input areset,
         input ldreg1,
         input [7:0] in,
         output valid,
         output [7:0] out
         );

reg [7:0] r1, r2, r3, r4, r5, r6;
reg v1, v2, v3, v4, v5;

        ...

// Die Pipeline
always @(posedge clock or posedge areset)
if(areset) begin
        r1 <= 8'h00;
        r2 <= 8'h00;
        r3 <= 8'h00;
        r4 <= 8'h00;
        r5 <= 8'h00;
        r6 <= 8'h00;

        v1 <= 1'b0;
        v2 <= 1'b0;
        v3 <= 1'b0;
        v4 <= 1'b0;
        v5 <= 1'b0;
end else begin
        if (ldreg1)
                r1 <= in;
        else
                r1 <= r1;

        r2 <= f1(r1);
        r3 <= f2(r2);
        r4 <= f3(r3);
        r5 <= f4(r4);
        r6 <= f5(r4,r5);

        v1 <= ldreg1;
        v2 <= v1;
        v3 <= v2;
        v4 <= v3;
        v5 <= v4;

end
```

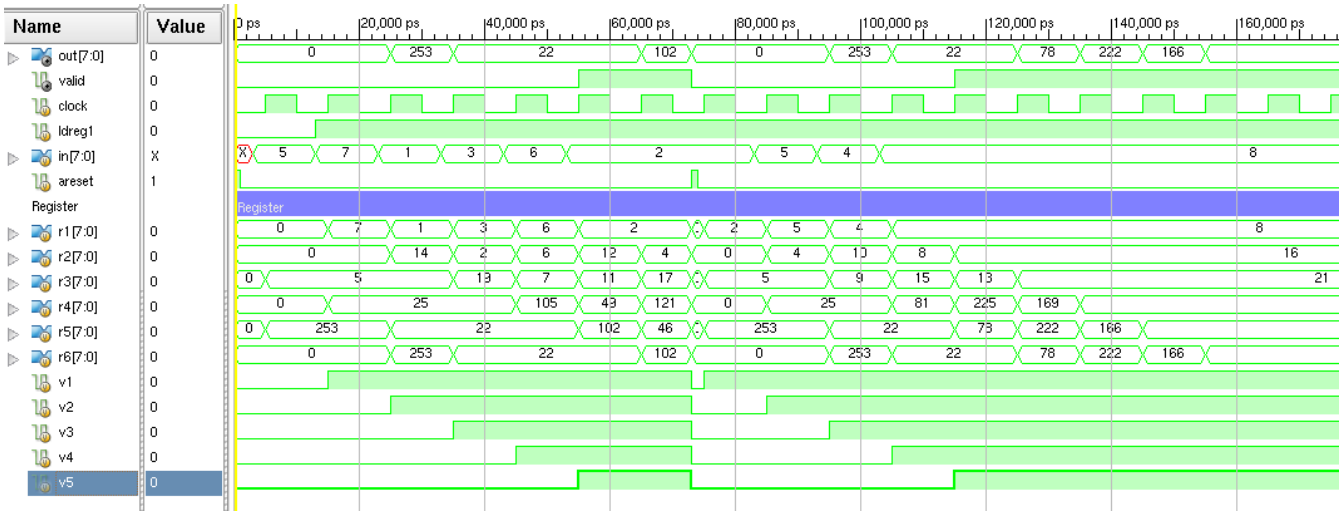
```
// Ausgang zuweisen
assign out = r6;
assign valid = v5;
endmodule
```

Für jede Stufe der Pipeline wird ein Register eingeführt (`v1` – `v5`), dass das valid-Signal zu dem Wert im entsprechenden Register (`r1` – `r6`) enthält. So “wandert” das valid-Bit mit dem (Zwischen-)Ergebnis durch die Pipeline.

Die Testbench kann im Wesentlichen aus Aufgabenteil b) übernommen werden. Dort muss nur ein `wire` für das valid-Signal eingepflegt werden.

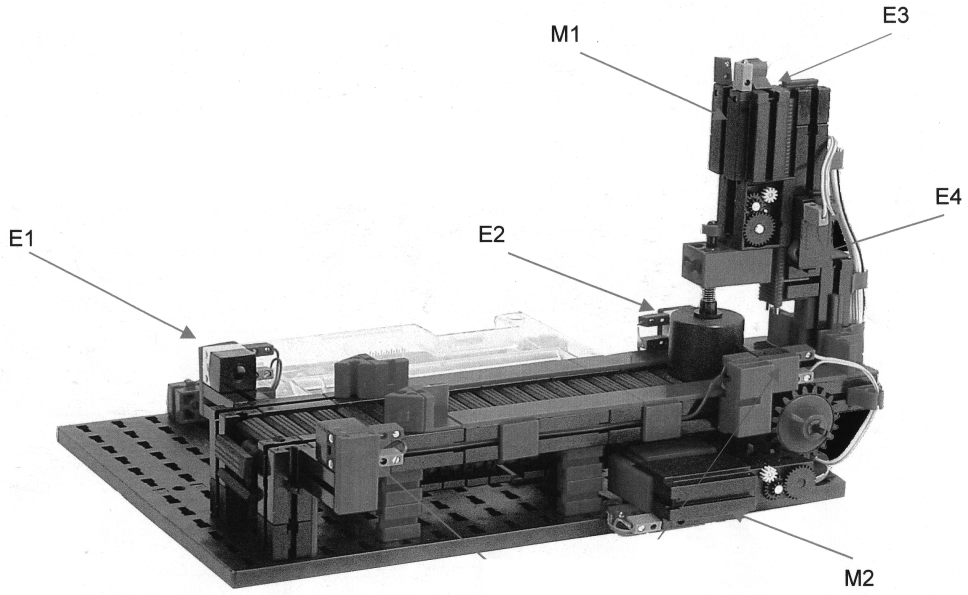
```
module testbench_c;

        ...

        // Outputs
        wire valid;

        ...

        // Instantiate the Unit Under Test (UUT)
        pipeline_c uut (
                .clock(clock),
                .areset(areset),
                .ldreg1(ldreg1),
                .in(in),
                .valid(valid),
                .out(out)
        );

        ...
```

Der folgenden Abbildung kann das Simulationsergebnis entnommen werden.



Aufgabe 2.2 Entwurf der Steuerung einer Stanzmaschine

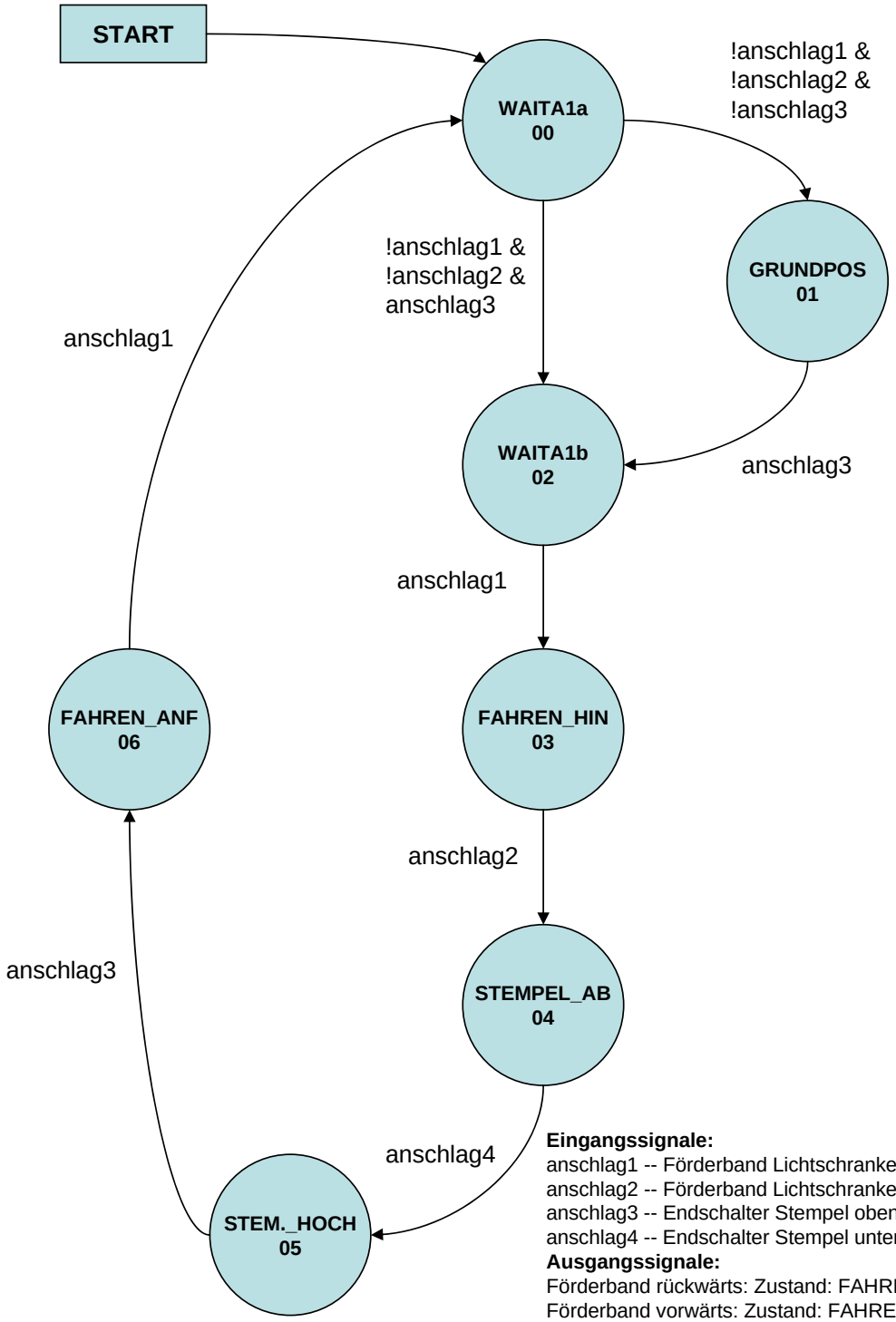
Die folgende Abbildung zeigt eine Stanzmaschine.



Die zu entwerfende Steuerung soll folgende Funktion realisieren.
Wird das zu stanzende Werkstück, z. B. von einem Gabelstapler auf das Förderband zwischen die Lichtschranke (E1) gelegt, so wird das Förderband (Motor M2) gestartet. Das Förderband läuft solange bis die hintere Lichtschranke (E2) erreicht ist. Danach kann der Stanzvorgang (Motor M1) gestartet werden. Der Ende-Schalter (E4) signalisiert, dass das Stanzen erfolgt ist. Danach muss die Stanze wieder in die Ausgangsposition gefahren werden. Das Erreichen signalisiert ein Ende-Schalter (E3). SchlieSSlich soll das Werkstück wieder zurück transportiert werden. Es kann davon ausgegangen werden, dass sich immer nur ein Werkstück auf dem Förderband befindet. Die Steuerung soll, um Unfälle zu vermeiden, beim Start eines Stanzvorgangs kontrollieren, ob sich der Stempel in seiner Grundposition befindet (E3).

Aufgabe 2.2.1 Zustandsgraph

Entwerfen Sie für die Stanzmaschine den Zustandsgraphen des Automaten (Moore) mit symbolischen Eingaben, Ausgaben und Zuständen. Zeichnen Sie den Zustandsgraphen. Kodieren Sie die Zustände des Steuerwerkes.



Eingangssignale:
anschlag1 -- Förderband Lichtschranke Start
anschlag2 -- Förderband Lichtschranke Stempel
anschlag3 -- Endschalter Stempel oben
anschlag4 -- Endschalter Stempel unten
Ausgangssignale:
Förderband rückwärts: Zustand: FAHREN_ANF
Förderband vorwärts: Zustand: FAHREN_HIN
Stempel hoch: STEMPEL_HOCH | GRUNDPOS
Stempel runter: Zustand: STEMPEL_AB

Aufgabe 2.2.2 Implementierung

Implementieren Sie den Moore-Automaten in Verilog HDL. Weisen Sie die korrekte Funktionsweise des Automaten durch Simulation nach. Wieviele Flip-Flops werden zur Realisierung Ihres Schaltwerks benötigt? Die Kodierung der Zustände ist Ihnen überlassen.

Die Implementierung als expliziter Automat ist im folgenden Listing zu sehen.

```
module stanze(
  input wire clk,
  input wire reset,
  input wire anschlag1, anschlag2, anschlag3, anschlag4,
  output wire m1_ab,
  output wire m1_hoch,
  output wire m2_hin,
  output wire m2_zur
  );

parameter
S_0 = 4'b0000,
S_1 = 4'b0001,
S_2 = 4'b0010,
S_3 = 4'b0011,
S_4 = 4'b0100,
S_5 = 4'b0101,
S_6 = 4'b0110;


reg [3:0] state, next_state;

// Realisierung der Ausgaben

assign m1_ab = (state == S_4);
assign m1_hoch = ((state == S_5) || (state == S_1));
assign m2_hin = (state == S_3);
assign m2_zur = (state == S_6);


// Realisierung der Transitionen
always@(state, anschlag1, anschlag2, anschlag3, anschlag4)
begin
case (state)
S_0 : if (!anschlag1 & !anschlag2 & !anschlag3)
      begin
                next_state = S_1;
               end
      else if (!anschlag1 & !anschlag2 & anschlag3)
               begin
                next_state = S_2;
               end
               else next_state = S_0;
S_1 : if (anschlag3)
      begin
                next_state = S_2;
               end
               else next_state = S_1;
S_2 : if (anschlag1)
```

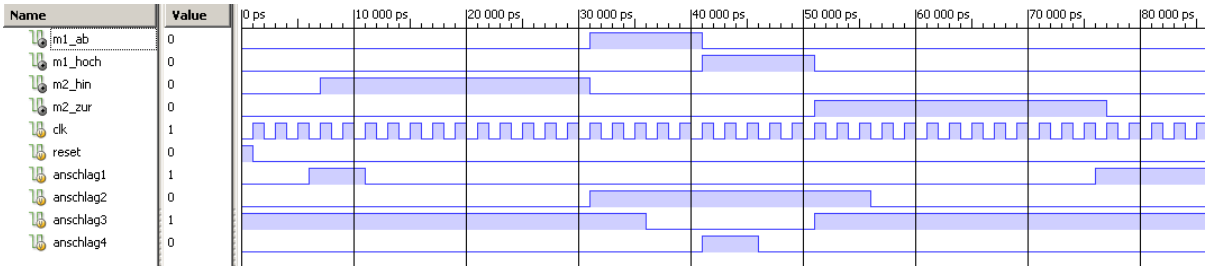
```
      begin
                next_state = S_3;
               end
               else next_state = S_2;
S_3 : if (anschlag2)
      begin
                next_state = S_4;
               end
               else next_state = S_3;
S_4 : if (anschlag4)
      begin
                next_state = S_5;
               end
               else next_state = S_4;
S_5 : if (anschlag3)
      begin
                next_state = S_6;
               end
               else next_state = S_5;
S_6 : if (anschlag1)
      begin
                next_state = S_0;
               end
               else next_state = S_6;
default: begin next_state = S_0; end
endcase
end

always@(posedge clk or posedge reset)
begin
 if (reset == 1) state <= S_0;
 else
 begin
  state <= next_state;
 end
end

endmodule
```

Als Stimuli wird folgendes Verilog HDL Programm verwendet.

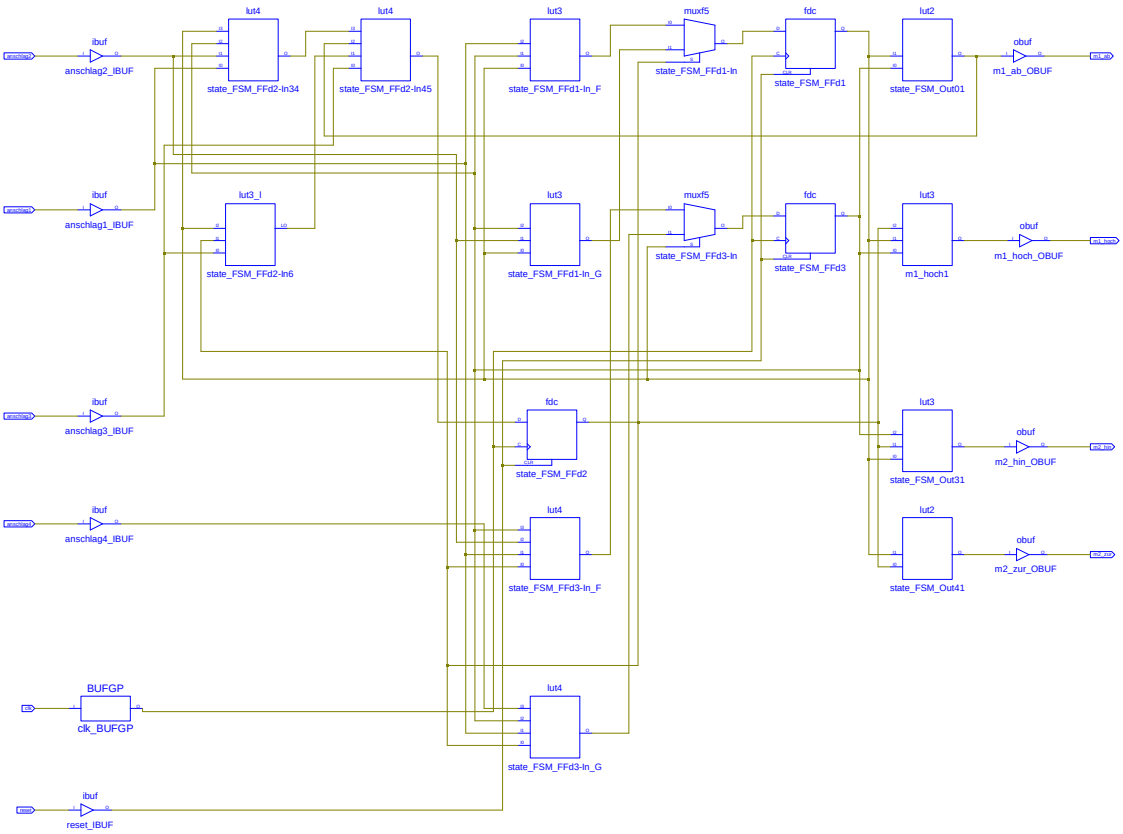
```
‘timescale 1ns / 1ns
module testbed;

        // Inputs
        reg clk;
        reg reset;
        reg anschlag1;
        reg anschlag2;
        reg anschlag3;
        reg anschlag4;

        // Outputs
        wire m1_ab;
        wire m1_hoch;
        wire m2_hin;
        wire m2_zur;
```

```
        // Instantiate the Unit Under Test (UUT)
        stanze uut (
                .clk(clk),
                .reset(reset),
                .anschlag1(anschlag1),
                .anschlag2(anschlag2),
                .anschlag3(anschlag3),
                .anschlag4(anschlag4),
                .m1_ab(m1_ab),
                .m1_hoch(m1_hoch),
                .m2_hin(m2_hin),
                .m2_zur(m2_zur)
        );

        initial begin
                // Initialize Inputs
                reset = 1;
                anschlag1 = 0;
                anschlag2 = 0;
                anschlag3 = 1;
                anschlag4 = 0;
                #1;
                reset = 0;
                #5;
                anschlag1 = 1;
                #5;
                anschlag1 = 0;
                #20;
                anschlag2 = 1;
                #5;
                anschlag3 = 0;
                #5;
                anschlag4 = 1;
                #5;
                anschlag4 = 0;
                #5;
                anschlag3 = 1;
                #5;
                anschlag2 = 0;
                #20;
                anschlag1 = 1;
        end
   // Takterzeugung
   initial begin
                clk = 1'b0;
                forever #1 clk = !clk;
        end
endmodule
```

Die Simulation zeigt, dass das Steuerwerk der Spezifikation genügt.



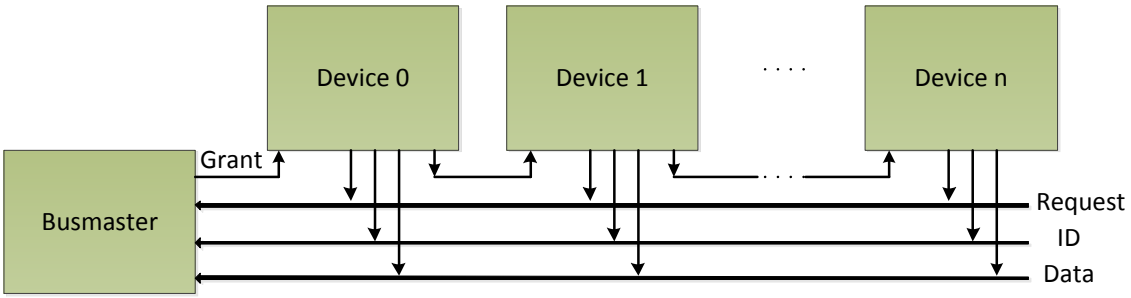
Das Ergebnis der Synthese ist im folgenden Bild zu sehen. Das Register *next_state* wird bei der Synthese zu kombinatorischer Logik (Vollständigkeit). Bei unvollständiger Beschreibung (z. B. weglassen von `else next_state = S_0`) ergibt sich ein Latch für das Register *next_state*. Zur Realisierung des Schaltwerks werden drei Flip-Flops benötigt.



Diese Hausaufgaben müssen bis 20.5.11, 18:00 über das Moodle-System abgegeben werden.

Hausaufgabe 2.1 Daisy Chain (5 Punkte)

Das Daisy-Chain Busprotokoll wird verwendet, um mehreren Sendern den Zugriff auf ein gemeinsames Medium zu ermöglichen. Die Steuerung erfolgt einerseits durch einen Busmaster, der die Daten empfängt und über das `Grant`-Signal mitteilt, wenn er bereit für neue Daten ist. Auf der anderen Seite sind die Sender, die über ein `Request`-Signal ihren Wunsch, Daten zu versenden, mitteilen.



a) Implementieren Sie folgende Module:

- `Busmaster`: Dieses Modul empfängt die Daten der Devices und erteilt das `Grant`-Signal, wenn es bereit ist Daten zu empfangen. Es soll die folgende Schnittstelle haben:

```
module busmaster(
  input  wire        clk,
  input  wire        reset,
  input  wire        request,
  input  wire [31:0] datain,
  input  wire [3:0]  id,
  output wire        grant
)
```

- `Device`: Dieses Modul implementiert einen Sender. Die Sendedaten selbst sind in dieser Aufgabe nicht relevant. Die Schnittstelle:

```
module device #(
  parameter device_id = 0
)(
  input  wire        clk,
  input  wire        reset,
  input  wire        grant,
  output wire [31:0] dataout,
  output wire [3:0]  id,
  output wire        next_grant,
  output wire        request
)
```

- `Testbench`: Im Testmodul soll 1 Busmaster und 4 Devices instanziiert werden. Desweiteren sollen Testdaten angelegt werden, um die korrekte Implementierung zu dokumentieren.

b) Welche Probleme können bei dieser Art von Busarbitrierung auftreten? Welche Lösungen sind als Abhilfe denkbar?

Hausaufgabe 2.2 CRC-Algorithmus (5 Punkte)

Fehlerkorrektur-Algorithmen werden z.B. verwendet, um Fehler bei der Datenübertragung oder Speicherung zu erkennen (und falls möglich zu korrigieren). Ein relativ einfaches Verfahren ist das CRC-Verfahren. Hier bei wird auf der Senderseite aus den Sendedaten eine Checksumme berechnet und an die Nutzdaten angehängt. Auf der Empfängerseite wird dann überprüft, ob die Checksumme die selbe ist. Falls ja, ist (wahrscheinlich) kein Fehler bei der Übertragung passiert, falls doch wird ein Fehler gemeldet. Weitere Erläuterungen finden sich leicht durch Suche im Internet.
Für das CRC-Verfahren muss einige sowohl auf der Sende- als auch auf der Empfängerseite das verwendete Polynom gleich sein.
Wir verwenden das Polynom $x^5 + x^3 + 1$. Implementieren Sie ein Modul, das sowohl für die Generierung als auch für die Überprüfung verwendet werden kann. Das Modul soll die folgende Schnittstelle haben:

```
module crc(
  input  wire       clk,       //Takt
  input  wire       reset,     //Reset
  input  wire       datain,    //Eingang der Daten, werden seriell angelegt
  input  wire       enable,    //= 1, wenn gültige Daten am Eingang liegen
  input  wire       check,     //0 = generiere CRC, 1 = checke CRC
  output wire [4:0] dataout,   //Ausgang der Daten im Generierungsmodus
  output wire       error      //= 1, falls ein Fehler im Checkmodus festgestellt wird
  )
```

Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Weitere Infos unter www.informatik.tu-darmstadt.de/plagiarism