Bergische Universität Wuppertal

Elektronik Praktikum

Versuch EP9 Digitalelektronik Teil 2: Programmierbare Logikbausteine (CPLD und FPGA)

Autoren:
Henrik Jürgens
Frederik Strothmann

Tutoren:
Hans-Peter Kind
Peter Knieling
Marius Wensing

8. Januar 2015

Inhaltsverzeichnis

1 Einleitung

In diesem Versuch geht es um programmierbare Logikbausteine (CPLD, FPGA). Damit können komplexe Steuerungsaufgaben effektiv gelöst werden, da sie die Funktion einiger Logikbausteine ersetzen können, welche sont in einer eigenen Schaltung realisiert werden müssten. Das Verhalten der Logikbausteine soll und kann auf zwei Verschiedene Arten festgelegt werden. Einfache Aufgaben können mit simplem Einzeichen eines Schaltplans, in dem einfache Gatter und Speicher (Flipflops) miteinander verbunden werden, gelöst werden. Bei den meisten schwierigeren Aufgaben wird dagegen eine Hardware-Beschreibungssprache wie VHDL oder Verilog[1] verwendet.[2]

2 Hardware: Das CPLD-Board und der Programmieradapter

Die Hardwarebeschreibung kann aus der Versuchsanleitung entnommen werden.[3]

3 Anleitung CPLD-Programmierung über Schaltpläne mit Xilinx ISE

Mit dem Programm 'Xilinx ISE 10.1'(kurz ISE) soll Schritt für Schritt eine kleine Schaltung aufgebaut werden. Dazu soll zunächst die Schaltung mt Hilfe von ISE als Schaltplan eingezeichnet werden. Sobald der Schaltplan mit einem Programmieradapter vom PC in den CPLD übertragen wurde, werden einige Messungen vorgenommen. Danach wird der Schaltplan in einer neuen Datei verändert, auf den CPLD übertragen und erneut gemessen.

4 Messungen am CPLD und Änderungen der Schaltung

In diesem Versuchsabschnitt werden verschiedenen Messungen vorgenommen.

> Da es kompikationen bei der installation von ISE gab konnten wir keine eigenen Bilder für die Versuchaufbauten verwenden. Abbildung 1 ist aus der Praktikumsanleitung entnommen und wurde mit Inkscape bearbeitet, um unserem Aufbau zu entsprechen. Abbildung 3 und 6 sind von der Gruppe Pagel, Roggel und Reisch, deren Aufbau identisch zu unserem ist.

4.1 Messungen an der jetzigen Schaltung

In diesem Versuchsteil werden für den 8fach-Zähler die Frequenzen von Q0 bis Q3 mit dem Oszilloskop gemessen.

Verwendete Geräte

Es werden das CPLD-Board, Verbindungskabel und ein Oszilloskop verwendet.

[1] welche an die Programmiersprache C angelehnt ist

[2] Die Programmdatei für den CPLD wird in beiden Fällen automatisch generiert

[3] Versuchsdurchführung Seite 5 und 6 http://www.atlas.uni-wuppertal.de/~kind/ep9_14.pdf

Versuchsaufbau

Es wird die Schaltung in Abbildung 1 mit ISE aufgebaut und implementiert.

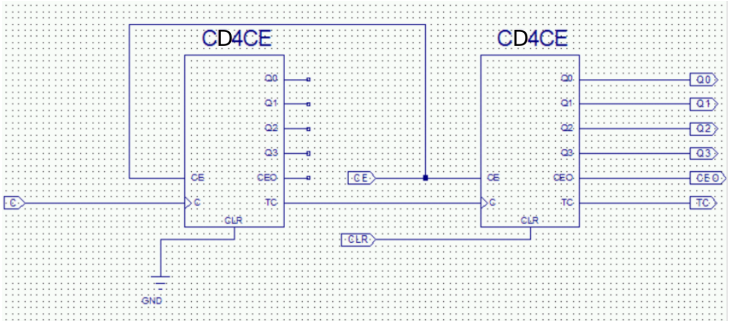

Abbildung 1: Schaltskizze des 8fach-Zählers

Dabei werden die Aus- und Eingänge nach dem folgende Schema belegt.

Marker	Pin
C	P6
CE	P2
CLR	P9
TC	P24
CEO	P22
Q0	P12
Q1	P13
Q2	P14
Q3	P18

Versuchsdurchführung

Es wird über die Mitte Pinreihe CLK ein Jumper gesetzt. Dann wird Pinreihen SV1 mit SW über ein Flachbandkabel verbunden so wie SV2 mit LED. Danach wird mit SW1 überprüft, ob die Schaltung funktioniert, mit SW8 der Zählzustand wieder zurückgesetzt und das Flachbandkabel zwischen SV2 und LED wieder entfernt. Über SV12 wird das Board an das Oszilloskop angeschlossen. Dann werden mit dem Oszillsokop die Frequenzen von Q0 bis Q3 gemessen.

Messergebnisse

Die Werte wurden direkt vom Oszillskop abgelesen.

Ausgang	Frequenz/kHz
Q0	3
Q1	1,2
Q2	0,6
Q3	0,6

Auswertung

Da in dem Aufbau ein CD4CE und kein CB4CE verwendet wurde sind die Messergebnisse nicht auswertbar. Der CD4CE zählt lediglich bis von 0 bis 9 und nicht von 0 bis 15. Das ist auch in der Aufnahme (Abbildung 2) der Ausgangsfrequenzen zu sehen. Dieser Fehler hat uns einen

großen Teil unserer Zeit gekostet, sodass wir den Versuchsteil nicht wiederholt haben. In den weiteren Versuchsteilen wird der richtige Zähler verwendet.

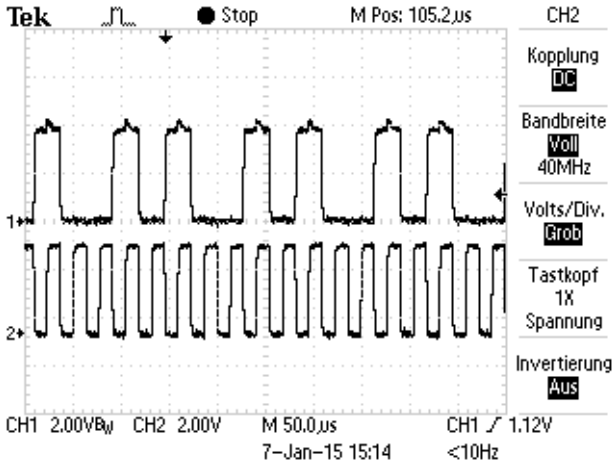

Abbildung 2: Aufnahme des Ausgangsfrequenz

Es wurde erwartet, dass immer nur Abfallende Flanke übereinander liegen.

4.2 Änderungen der Schaltung (1): 16fach-Zähler

In diesem Versuchsteil soll die Eingangsfrequenz von 50kHz auf 1Hz herabgesenkt werden. Dafür wird die selbe Schaltung wie in dem vorherigen Versuchsteil verwendet, jedoch werden vier anstatt zwei CB4CE hintereinander geschaltet.

Verwendete Geräte

Es werden das CPLD-Board, Verbindungskabel und ein Oszilloskop verwendet.

Versuchsaufbau

Es wird die Schaltung in Abbildung 3 mit ISE aufgebaut und implementiert.

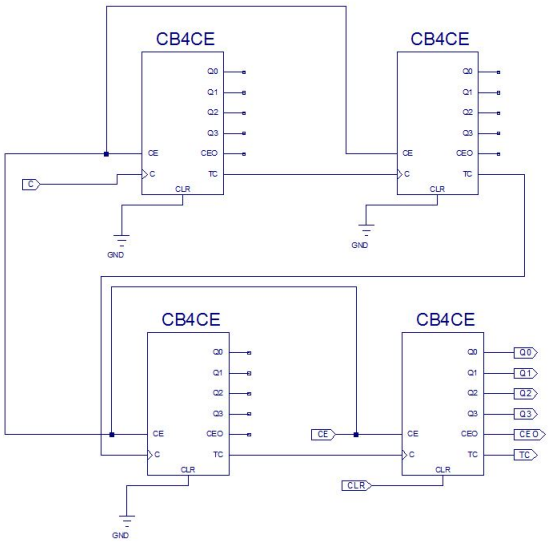

Abbildung 3: Schaltskizze des 16fach-Zählers

Dabei werden die Aus- und Eingänge nach dem folgenden Schema belegt.

Marker	Pin
C	P6
CE	P2
CLR	P9
TC	P24
CEO	P22
Q0	P12
Q1	P13
Q2	P14
Q3	P18

Versuchsdurchführung

Es werden die Pinreihen SV1 mit SW über ein Flachbandkabel verbunden. Danach wird das Board über SV12 an das Oszilloskop angeschlossen und die Periodendauer von Q0 bis Q3 mit den Cursern gemessen, da das Oszilloklop keine Frequenzen unter 10Hz messen kann.

Messergebnisse

Die Messwerte wurden direkt vom Oszillsokop abgelesen.

Ausgang	Periodendauer/ms
Q0	56
Q1	112
Q2	222
Q3	444

Tabelle 1: Messergebnisse der Periodendauern

Auswertung

Bei der Messung der Periodendauer ergaben sich auf dem Ozillsokop für die Ausgänge Q0 und Q1 der Verlauf in Abbildung 4.

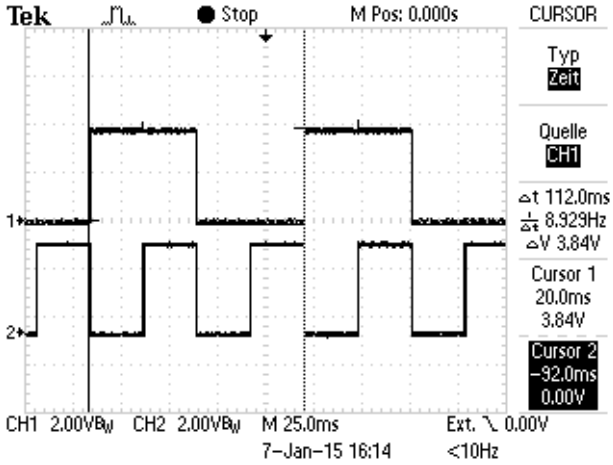

Abbildung 4: Aufnahme der Periodendauern für Q0 und Q1

Die Aufnahme auf dem Oszillsokop für Q2 und Q3 ist in Abbildung 5 zu sehen.

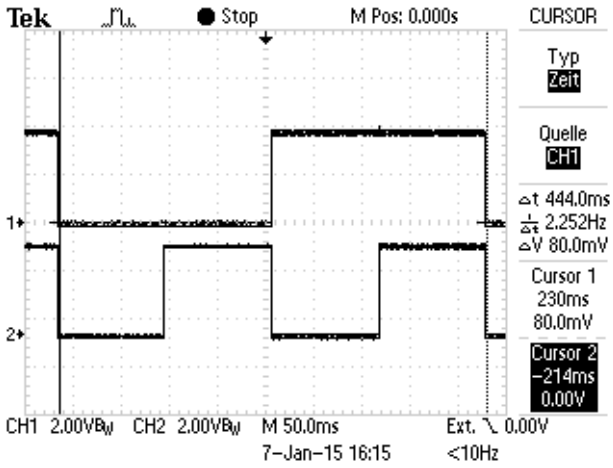

Abbildung 5: Aufnahme der Periodendauern für Q2 und Q3

Es wurde erwartet, dass die sich die Frequenz zum nächsten Ausgang halbiert, dies ist anhand von Tabelle 1 gut zu sehen. Die Frequenz und die Periodendauer hängen reziprok zusammen, daher entspricht eine Verdoppelung der Periodendauer einer Halbierung der Frequenz.

4.3 Änderungen der Schaltung (2): 3-nach-8-Dekoder

Es sollen die untersten drei Ausgänge so umprogrammiert werden, dass nacheinander die acht Leuchtdioden angehen.

Verwendete Geräte

Es werden das CPLD-Board, Verbindungskabel und ein PC verwendet.

Versuchsaufbau

Es wird die Schaltung in Abbildung 6 mit ISE aufgebaut und implementiert.

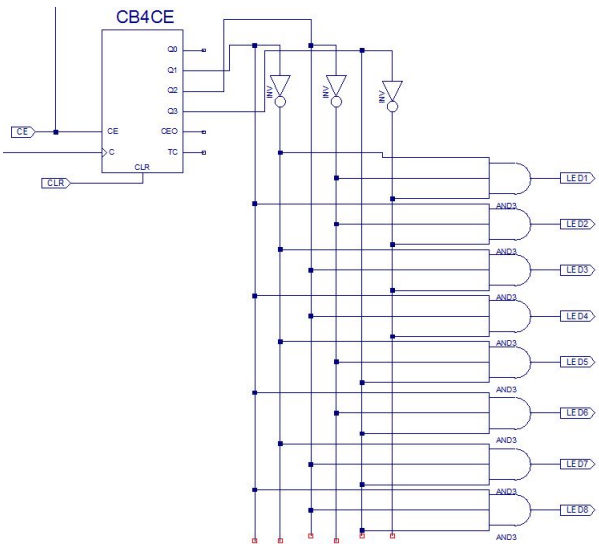

Abbildung 6: Schaltskizze des 16fach-Zählers

Dabei werden die Aus- und Eingänge nach dem folgenden Schema belegt.

CPLD	SV3	LED
1	1	1
44	2	2
42	3	3
43	4	4
40	5	5
39	6	6
38	7	7
37	8	8

Versuchsdurchführung

Es wird die Schaltung wie in Abschnitt 4.3 aufgebaut und implementiert, dann wird das Verhalten der LEDs beobachtet.

Auswertung

Es war zu beobachten, dass die Schaltung über die LEDs im Binärsystem hochzählte.

Diskussion

Im ersten Versuchsteil konnten nicht die gewünschen Ergebnisse erzielt werden da ein falsches Bauteil verwendet wurde. In den anderen beiden Versuchteilen sind gute Ergebnisse erzielt worden, welche den Erwartungen entsprachen.

5 Programmierung der CPLD mit der Hardware Description Language Verilog

Da das Erstellen des Schaltplanes bei komplizierteren Schaltungen sehr mühsam wird, ist es einfacher die Schaltung mit einer HDL (Verilog) zu programmieren. Ziel dieses Versuchsteiles ist es, einen Vorzähler (12 bis 16 Bit) und einen 8-Bit-Zähler zu programmieren, welcher dann auf den 8 LEDs und den beiden Sibensegmentanzeigen angezeigt werden soll. Dafür müssen jeweils 4 Bits des Zählers in einen 8-Bit-Zustand der Siebensegmentanzeige übergeführt werden.

Verwendete Geräte

Verwendet werden ein Computer, das CPLD, ein Programmieradapter, eine 9V Spannungsquelle sowie nach Versuchsteil mehrere Flachbandkabel.

5.1 Ein neues Projekt

Diesmal soll ein neues Projekt nicht über einen Schaltplan, sondern über eine Verilog Datei definiert werden. Die genaue Vorgehensweise kann in der Versuchsbeschreibung nachgelesen werden.[4]

[4] http://www.atlas.uni-wuppertal.de/~kind/ep9_14.pdf Seite 19 und 20

5.2 Ein einfaches Verilog Programm für die 8 LEDs

Am CPLD wird die Pinreihe SV1 mit der Pinreihe SW verbunden, sowie die Pinreihe SV2 mit der Pinreihe LED. Dazu werden zwei Flachbandkabel verwendet. Es wird nun ein Vorgegebenes Programm kompiliert und dessen Funktion untersucht.

Versuchsaufbau

Es wird der Quellcode 8 aus der Versuchsanleitung übernommen, kompiliert und dessen Funktion beobachtet.

Versuchsdurchführung

Nachdem ein neues Projekt erstellt wurde (vgl. Versuchsanleitung S. 19 f.), wird der in der Versuchsanleitung angegebene Programmiercode eingefügt. Der Programmiercode ist bereits auskommentiert. Nachdem die Syntax überprüft wurde, kann das Programm kompiliert werden. Nach weiteren Arbeitsschritten (vgl. Versuchsanleitung S. 22) sollte auf den 8 LEDs das Binärmuster des hochlaufenden Zählers angezeigt werden.

Auswertung

Das Programm konnte erfolgreich kompiliert werden. Es wurde beobachtet, dass die LEDs im Binärsystem Hochzählen.

5.3 Änderung 1:Ausgabe des Zählers an die Siebensegmentanzeigen

In diesem Versuchsteil soll der Zählerstand auf der Siebensegmentanzeige sichtbar gemacht werden.

Versuchsaufbau

Die Pinreihe SV3 wird mit einem weiteren Flachbandkabel mit der Pinreihe DIS1 verbunden. Der Quellcode aus der Versuchsanleitung wurde während des Versuches konfiguriert (Code 9).

Versuchsdurchführung

Im Quellcode werden die voher auskommentierten Register 'segments1' und 'segments2' aktiviert. Am Ende des always Blocks wird ein vorgegebener Befehl hinzugefügt. Danach wird der Quellcode kompiliert und die Funktion der Siebensegmentanzeige überprüft. **Fragen**:

- Wie kann die Geschwindigkeit erhöht oder erniedrigt werden, mit der die Zählerzustände durchlaufen werden, ohne die Eingangsfrequenz (ca. 50 kHz) zu verändern?
- Das Signal CLR dient zum Rücksetzen des Zählers. Welcher Taster erzeugt es?
- Ein weiteres Signal wurde bisher gar nicht benutzt: CE (clock enable). Wie kann Sie es erreichen, daß der Zähler nur läuft, wenn CE auf 1 ist? Wie kann man den Zähler stoppen, wenn CE auf 1 ist? Welcher Taster erzeugt CE?

1. Die Geschwindigkeit, mit der die Zählerzustände durchlaufen werden, ändert sich mit der Größe der Variablen 'prescaler' (momentan 14 Bit). 15 Bit halbiert die Geschwindigkeit, 13 Bit verdoppelt sie.

2. Der Taster SW8 in der Ecke oben rechts auf dem Board ist CLR und dient zum zurücksetzen des Zählers.

3. Für CE kann eine zusätzliches If Kommando in das Programm eingefügt werden, welche den Zähler beinhaltet. Diese Funktion wurde jedoch erst im letzten Versuchsteil in den Quellcode eingebaut. CE liegt auf dem Taster SW1, und der Zähler kann, wie wir im letzten Versuchsteil sehen werden, mit CLR gestoppt werden, wenn CE auf 1 ist.

Auswertung

nachdem der Quellcode konfiguriert und kompiliert wurde, konnte beobachtet werden, dass an der 7-Segmentanzeige zuerst die 0, dann die 1 und danach jede einzelne der 8 LEDs aufleuchteten. Der If-Befehl für CE wurde erst im letzten Versuchsteil eingebaut.

5.4 Änderung 2:Strichmuster für alle 16 Zustände

In diesem Versuchsteil soll die Siebensegmentanzeige die Zahlen 0-f des Hexadezimalsystems darstellen.

Versuchsaufbau

Der Quellcode aus der Versuchsanleitung wurde während des Versuches konfiguriert (Code 10).

Versuchsdurchführung

Die Zahlen 1-f im Hexadezimalsystem sollen mit den 8 Bits des Siebensegmentdecoders dargestellt werden. Der case-Befehl weist jedem der 16 Zustände von 'decodeinput' (4 Bit) ein 8 Bit Muster, welches die LEDs am Siebensegmentdecoder aufleuchten lässt, zu. Nun sollen diese Muster im Quellcode verändert werden, sodass von 0 bis f hochgezählt wird.

Auswertung

Die Strichmuster konnten erfolgreich konfiguriert werden, der Zähler zählt von 0 bis f.

5.5 Änderung 3: Aus dem Binärzähler wird ein Dezimalzähler

In diesem Versuchsteil soll die Siebensegmentanzeige nach der Zahl '9' wieder bei '0' anfangen zu zählen.

Versuchsaufbau

Der Quellcode aus der Versuchsanleitung wurde während des Versuches konfiguriert (Code 11).

Versuchsdurchführung

In die always-Schleife wird wie in der Versuchsbeschreibung beschrieben ein If-Befehl, welcher 'counter[3:0]' bei 'counter[3:0] == 10' innerhalb dieser auf 0 setzt, eingefügt, sodass nach der '9' wieder die '0' am Siebensegmentdecoder angezeigt wird.

Auswertung

Der If-Befehl wurde an der passenden Stelle im Quellcode eingefügt und sorgt dafür, dass nur noch von 0 bis 9 hochgezählt wird. Die Funktion des Quellcodes wurde nach dem programmieren des CPLD an Siebensegmentdecoder beobachtet.

5.6 Änderung 4: Ein zweistelliger Zähler

Es soll nun ein zweistelliger Zähler implementiert werden, welcher von '00' bis '99' hochzählt. Daneben wird auch die Funktion des CE Tasters, welche im zweiten Versuchsteil angesprochen wurde, implementiert.

Versuchsaufbau

Die Pinreihe SV4 wird mit einem Flachbandkabel mit der Pinreihe DIS2 verbunden. Der Quellcode aus der Versuchsanleitung wurde während des Versuches konfiguriert (Code 12 und Code 13).

Versuchsdurchführung

Zuerst wird ein zweites 4 Bit Register counter2 hinzugefügt. Danach wird die always-Schleife verändert. Falls CLR gedrückt wird, soll ebenfalls counter2 (neben dem Prescaler und counter) auf null gesetzt werden.
Frage:

- Wenn Sie den CLR-Taster drücken, springt das Siebensegmentdisplay nicht sofort auf Null. Warum? Wie können Sie das ändern?

1. Damit das Siebensegmentdisplay auf null springt muss dahinter 'segments1' und 'segments2' mit der segment7-Funktion sowie den Registern 'conter[3:0]' für 'segments1' und 'counter2[3:0]' für 'segments2' das entsprechende 8 Bitmuster zugewiesen werden. Dies wird erst im letzten Teil des Quellcodes eingefügt, sodass CLR davor zum Anhalten des Zählers benutzt werden kann.

Nun soll der Zähler nur laufen, falls CE gedrückt gehalten wird. Dafür wird anstatt 'else begin ... end' der Befehle 'else if(CE) begin ... end' implementiert. In dem Befehl 'if(counter[3:0] == 10) begin ... end' muss jetzt zusätzlich 'counter2' um 1 erhöht werden. Damit counter2 nicht von 0 bis f, sondern von 0 bis 9 hochzählt, wird ein weiterer If-Befehl ergänzt, welcher die gleiche Struktur hat, wie der für counter, sodass an der Siebensegmentanzeige nach '99' wieder '00' angezeigt wird. Zum Schluss soll neben 'segments1' auch 'segments2' das richtige 8 Bitmuster zugewiesen werden. Also wird neben 'segments1' auch 'segments2' mit der Funktion segment7 das passende 8 Bitmuster zugewiesen. (In diesem Fall das von counter2) Falls es zu Fehlermeldungen wegen den Ressourcen des CPLD kam, wurden die Größen der Register angepasst.

Auswertung

Der Quellcode konnte nach kurzer Zeit angepasst werden, sodass auf der Siebensegmentanzeige von '00' bis '99' hochgezählt wird. Da zu diesem Zeitpunkt die Anzeige mit CLR lediglich angehalten wurde, sollte der Quellcode danach so verändert werden, dass die Anzeige auf '00' springt, sobald CLR gedrückt wird. Dafür wurde am Ende des if(CLR)-Befehls 'segments1' und 'segments2' das entsprechende 8 Bitmuster zugewiesen.

Diskussion

Verilog konnte in diesem Versuch ohne große Schwierigkeiten verwendet werden. Die einzelnen Konfigurationsschritte konnten alle umgesetzt werden und nach dem programmieren des CPLD an den leuchtenden LEDs der LED-Anzeige und der bzw. den beiden Anzeigen der Siebensegmentdecoder überprüft werden. Es stellte sich je nach Aufgabenteil das erwartete Ergebnis ein.

6 Fazit

Bis auf einen kleinen Fehler am Anfang des Versuches, der uns sehr viel Zeit gekostet hat, ist der Versuch gut verlaufen. Am Ende war leider keine Zeit mehr für die Bearbeitung der Zusatzaufgabe über.

7 Anhang

Im Anhang befinden sich die jeweils verwendeten Verilogdateien und die ufc Datei, die die Pinbelegung angibt.

```
#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "C"       LOC = "P6"   ;
NET "CE"      LOC = "P2"   ;
NET "CLR"     LOC = "P9"   ;
NET "segments1<0>"  LOC = "P1"   ;
NET "segments1<1>"  LOC = "P44"  ;
NET "segments1<2>"  LOC = "P42"  ;
NET "segments1<3>"  LOC = "P43"  ;
NET "segments1<4>"  LOC = "P40"  ;
NET "segments1<5>"  LOC = "P39"  ;
NET "segments1<6>"  LOC = "P38"  ;
NET "segments1<7>"  LOC = "P37"  ;
NET "segments2<0>"  LOC = "P35"  ;
NET "segments2<1>"  LOC = "P34"  ;
NET "segments2<2>"  LOC = "P33"  ;
NET "segments2<3>"  LOC = "P29"  ;
NET "segments2<4>"  LOC = "P28"  ;
NET "segments2<5>"  LOC = "P27"  ;
NET "segments2<6>"  LOC = "P26"  ;
NET "segments2<7>"  LOC = "P25"  ;
NET "LED_OUT<0>"    LOC = "P12"  ;
NET "LED_OUT<1>"    LOC = "P13"  ;
NET "LED_OUT<2>"    LOC = "P14"  ;
NET "LED_OUT<3>"    LOC = "P18"  ;
NET "LED_OUT<4>"    LOC = "P19"  ;
NET "LED_OUT<5>"    LOC = "P20"  ;
NET "LED_OUT<6>"    LOC = "P22"  ;
NET "LED_OUT<7>"    LOC = "P24"  ;

#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE
```

Abbildung 7: Datei zur Festlegung der Pinbelegung

```
timescale 1ns / 1ps // Diese Zeitskala (Schrittweite, Aufloesung)
// ist nur bei Simulationen wichtig
//////////////////////////////////////////////////////////////////////////////////
// Company: // Engineer: VERSCHIEDENE KOMMENTARE
// Create Date: 12:22:32 01/08/2010
// ...
//////////////////////////////////////////////////////////////////////////////////
module testXC9572( // Jedes Verilog-Programm ist ein Modul mit einem Namen
    C, CE, CLR, // und in runden Klammern muessen
    segments1, segments2, LED_OUT // alle Ein-/Ausgangssignale genannt werden
    );

input C;                        // Signale als EINgaenge definieren
input CE; input CLR;            // C = Clock, CLR = Clear, jeweils 1 Signal
output [7:0] segments1;         // Signale als AUSgaenge definieren
output [7:0] segments2;         // hier haben wir z.B. Signalgruppen von
output [7:0] LED_OUT;           // 8 Bits, daher die Angabe [7:0]

// reg [7:0] segments1;         // Das ist erstmal noch auskommentiert,
// reg [7:0] segments2;         // weil wir es erst spaeter brauchen.
  reg [14:0] prescaler;         // reg = Register, das ist praktisch ein D-Flipflop.
  reg [7:0]  counter;           // Hier haben wir Register mit 15 bzw. 8 Bit.

assign LED_OUT = counter;       // Der Inhalt des 8-Bit-Zaehlers "counter" wird
                                // mit assign direkt den
                                // 8 Bits von LED_OUT zugeordnet.

                                    // Unser Zaehler soll mit Signal C getaktet
always @ (posedge C or posedge CLR) // und mit CLR zurueckgesetzt werden.
    begin                           // (posedge = ansteigende Flanke)
        if(CLR)                     // Wenn also CLR auf 1 ist ...
            begin
                prescaler = 0;      // dann setzte counter und prescaler auf 0
                counter = 0;
            end
        else                        // andernfalls (also wenn CLR = 0) ...
            begin
                prescaler = prescaler + 1;  // ... erhoehe den prescaler um 1
                if(prescaler == 0)          // und wenn dieser auf 0 ist (das passiert
                    begin                   // einmal in 2 hoch 15 Takten!) dann ...
                        counter = counter + 1; // ... erhoehe den counter um 1
                    end
            end
end


// ----  FUNCTION: 7-segment-decoder ------------------------------------
function [7:0] segment7;            // Eine Funktion mit dem Namen und
                                    // 8 Bit grossen Ausgabewert segment7

    input [3:0] decodeinput;        // Was an die Funktion uebergeben wird,
                                    // wird innerhalb der Funktion unter dem
                                    // Namen decodeinput gefuehrt (4 Bit).

    case (decodeinput)              // Fuer den Fall, dass decodeinput ...
        4'h0 : segment7 = 8'b00111111; // ... den Wert 0 hat: segment7 = Muster "0"
                          // Dgfedcba <<< Stellen fuer Dezimalpunkt (D) und 7 Segmente
                          // als 8-Bit-Binaerzahl (daher 8'b)
                          // decodeinput schreiben wir als 4-Bit-Hexadezimalzahl, also 0-9,a-f
        4'h1 : segment7 = 8'b00000110; // das Strichmuster "1"
        4'h2 : segment7 = 8'b00000001; // diese Strichmuster muessen Sie noch aendern
        4'h3 : segment7 = 8'b00000010;
        4'h4 : segment7 = 8'b00000100;
        4'h5 : segment7 = 8'b00001000;
        4'h6 : segment7 = 8'b00010000;
        4'h7 : segment7 = 8'b00100000;
        4'h8 : segment7 = 8'b01000000;
        4'h9 : segment7 = 8'b10000000;
        4'ha : segment7 = 8'b01000000;
        4'hb : segment7 = 8'b00100000;
        4'hc : segment7 = 8'b00010000;
        4'hd : segment7 = 8'b00001000;
        4'he : segment7 = 8'b00000100;
        4'hf : segment7 = 8'b00000010;
    endcase                         // Ende der case-Anweisung
endfunction                         // Ende der Funktion

endmodule
```

Abbildung 8: Quellcode für die 1. Aufgabe

```
`timescale 1ns / 1ps // Diese Zeitskala (Schrittweite, Aufloesung)
// ist nur bei Simulationen wichtig
//////////////////////////////////////////////////////////////////////////////////
// Company: // Engineer: VERSCHIEDENE KOMMENTARE
// Create Date: 12:22:32 01/08/2010
// ...
//////////////////////////////////////////////////////////////////////////////////
module testXC9572( // Jedes Verilog-Programm ist ein Modul mit einem Namen
    C, CE, CLR, // und in runden Klammern muessen
    segments1, segments2, LED_OUT // alle Ein-/Ausgangssignale genannt werden
    );

input C;                        // Signale als EINgaenge definieren
input CE; input CLR;            // C = Clock, CLR = Clear, jeweils 1 Signal
output [7:0] segments1;         // Signale als AUSgaenge definieren
output [7:0] segments2;         // hier haben wir z.B. Signalgruppen von
output [7:0] LED_OUT;           // 8 Bits, daher die Angabe [7:0]

  reg [7:0] segments1;        // Das ist erstmal noch auskommentiert,
  reg [7:0] segments2;        // weil wir es erst spaeter brauchen.
  reg [14:0] prescaler;       // reg = Register, das ist praktisch ein D-Flipflop.
  reg [7:0]  counter;         // Hier haben wir Register mit 15 bzw. 8 Bit.

assign LED_OUT = counter;     // Der Inhalt des 8-Bit-Zaehlers "counter" wird
                              // mit assign direkt den
                              // 8 Bits von LED_OUT zugeordnet.

                                  // Unser Zaehler soll mit Signal C getaktet
always @ (posedge C or posedge CLR)  // und mit CLR zurueckgesetzt werden.
   begin                          // (posedge = ansteigende Flanke)
      if(CLR)                     // Wenn also CLR auf 1 ist ...
         begin
            prescaler = 0;        // dann setzte counter und prescaler auf 0
            counter = 0;
         end
      else                        // andernfalls (also wenn CLR = 0) ...
         begin
            prescaler = prescaler + 1;  // ... erhoehe den prescaler um 1
            if(prescaler == 0)    // und wenn dieser auf 0 ist (das passiert
               begin              // einmal in 2 hoch 15 Takten!) dann ...
                  counter = counter + 1; // ... erhoehe den counter um 1
               end
         segments1 = segment7(counter[3:0]);
              end
end


// ----- FUNCTION: 7-segment-decoder ------------------------------------
function [7:0] segment7;          // Eine Funktion mit dem Namen und
                                  // 8 Bit grossen Ausgabewert segment7

   input [3:0] decodeinput;       // Was an die Funktion uebergeben wird,
                                  // wird innerhalb der Funktion unter dem
                                  // Namen decodeinput gefuehrt (4 Bit).

   case (decodeinput)             // Fuer den Fall, dass decodeinput ...
      4'h0 : segment7 = 8'b00111111; // ... den Wert 0 hat: segment7 = Muster "0"
                     // Dgfedcba <<< Stellen fuer Dezimalpunkt (D) und 7 Segmente
                     // als 8-Bit-Binaerzahl (daher 8'b)
                     // decodeinput schreiben wir als 4-Bit-Hexadezimalzahl, also 0-9,a-f
      4'h1 : segment7 = 8'b00000110; // das Strichmuster "1"
      4'h2 : segment7 = 8'b00000001; // diese Strichmuster muessen Sie noch aendern
      4'h3 : segment7 = 8'b00000010;
      4'h4 : segment7 = 8'b00000100;
      4'h5 : segment7 = 8'b00001000;
      4'h6 : segment7 = 8'b00010000;
      4'h7 : segment7 = 8'b00100000;
      4'h8 : segment7 = 8'b01000000;
      4'h9 : segment7 = 8'b10000000;
      4'ha : segment7 = 8'b01000000;
      4'hb : segment7 = 8'b00100000;
      4'hc : segment7 = 8'b00010000;
      4'hd : segment7 = 8'b00001000;
      4'he : segment7 = 8'b00000100;
      4'hf : segment7 = 8'b00000010;
   endcase                           // Ende der case-Anweisung
endfunction                          // Ende der Funktion

endmodule
```

Abbildung 9: Quellcode für die 2. Aufgabe

```
`timescale 1ns / 1ps // Diese Zeitskala (Schrittweite, Aufloesung)
// ist nur bei Simulationen wichtig
//////////////////////////////////////////////////////////////////////////////////
// Company: // Engineer: VERSCHIEDENE KOMMENTARE
// Create Date: 12:22:32 01/08/2010
// ...
//////////////////////////////////////////////////////////////////////////////////
module testXC9572( // Jedes Verilog-Programm ist ein Modul mit einem Namen
    C, CE, CLR, // und in runden Klammern muessen
    segments1, segments2, LED_OUT // alle Ein-/Ausgangssignale genannt werden
    );

input C;                        // Signale als EINgaenge definieren
input CE; input CLR;            // C = Clock, CLR = Clear, jeweils 1 Signal
output [7:0] segments1;         // Signale als AUSgaenge definieren
output [7:0] segments2;         // hier haben wir z.B. Signalgruppen von
output [7:0] LED_OUT;           // 8 Bits, daher die Angabe [7:0]

  reg [7:0] segments1;         // Das ist erstmal noch auskommentiert,
  reg [7:0] segments2;         // weil wir es erst spaeter brauchen.
  reg [14:0] prescaler;         // reg = Register, das ist praktisch ein D-Flipflop.
  reg [7:0]  counter;           // Hier haben wir Register mit 15 bzw. 8 Bit.

assign LED_OUT = counter;       // Der Inhalt des 8-Bit-Zaehlers "counter" wird
                                // mit assign direkt den
                                // 8 Bits von LED_OUT zugeordnet.

                                    // Unser Zaehler soll mit Signal C getaktet
always @ (posedge C or posedge CLR) // und mit CLR zurueckgesetzt werden.
   begin                            // (posedge = ansteigende Flanke)
      if(CLR)                       // Wenn also CLR auf 1 ist ...
         begin
            prescaler = 0;          // dann setzte counter und prescaler auf 0
            counter = 0;
         end
      else                          // andernfalls (also wenn CLR = 0) ...
         begin
            prescaler = prescaler + 1;  // ... erhoehe den prescaler um 1
            if(prescaler == 0)          // und wenn dieser auf 0 ist (das passiert
               begin                    // einmal in 2 hoch 15 Takten!) dann ...
                  counter = counter + 1; // ... erhoehe den counter um 1
               end
         segments1 = segment7(counter[3:0]);
              end
end


// ----- FUNCTION: 7-segment-decoder ------------------------------------
function [7:0] segment7;            // Eine Funktion mit dem Namen und
                                    // 8 Bit grossen Ausgabewert segment7

   input [3:0] decodeinput;         // Was an die Funktion uebergeben wird,
                                    // wird innerhalb der Funktion unter dem
                                    // Namen decodeinput gefuehrt (4 Bit).

   case (decodeinput)               // Fuer den Fall, dass decodeinput ...
      4'h0 : segment7 = 8'b00111111; // ... den Wert 0 hat: segment7 = Muster "0"
                     // Dgfedcba <<< Stellen fuer Dezimalpunkt (D) und 7 Segmente
                     // als 8-Bit-Binaerzahl (daher 8'b)
                     // decodeinput schreiben wir als 4-Bit-Hexadezimalzahl, also 0-9,a-f
      4'h1 : segment7 = 8'b00000110; // das Strichmuster "1"
      4'h2 : segment7 = 8'b01011011; // diese Strichmuster muessen Sie noch aendern
      4'h3 : segment7 = 8'b01001111;
      4'h4 : segment7 = 8'b01100110;
      4'h5 : segment7 = 8'b01101101;
      4'h6 : segment7 = 8'b11111101;
      4'h7 : segment7 = 8'b00000111;
      4'h8 : segment7 = 8'b01111111;
      4'h9 : segment7 = 8'b11101111;
      4'ha : segment7 = 8'b01110111;
      4'hb : segment7 = 8'b01111111;
      4'hc : segment7 = 8'b00111001;
      4'hd : segment7 = 8'b00111111;
      4'he : segment7 = 8'b01111001;
      4'hf : segment7 = 8'b01110001;
   endcase                          // Ende der case-Anweisung
endfunction                         // Ende der Funktion

endmodule
```

Abbildung 10: Quellcode für die 3. Aufgabe

```
`timescale 1ns / 1ps // Diese Zeitskala (Schrittweite, Aufloesung)
// ist nur bei Simulationen wichtig
//////////////////////////////////////////////////////////////////////////////////
// Company: // Engineer: VERSCHIEDENE KOMMENTARE
// Create Date: 12:22:32 01/08/2010
// ...
//////////////////////////////////////////////////////////////////////////////////
module testXC9572( // Jedes Verilog-Programm ist ein Modul mit einem Namen
    C, CE, CLR, // und in runden Klammern muessen
    segments1, segments2, LED_OUT // alle Ein-/Ausgangssignale genannt werden
    );

input C;                        // Signale als EINgaenge definieren
input CE; input CLR;            // C = Clock, CLR = Clear, jeweils 1 Signal
output [7:0] segments1;         // Signale als AUSgaenge definieren
output [7:0] segments2;         // hier haben wir z.B. Signalgruppen von
output [7:0] LED_OUT;           // 8 Bits, daher die Angabe [7:0]

  reg [7:0] segments1;         // Das ist erstmal noch auskommentiert,
  reg [7:0] segments2;         // weil wir es erst spaeter brauchen.
  reg [14:0] prescaler;         // reg = Register, das ist praktisch ein D-Flipflop.
  reg [7:0]  counter;           // Hier haben wir Register mit 15 bzw. 8 Bit.

assign LED_OUT = counter;       // Der Inhalt des 8-Bit-Zaehlers "counter" wird
                                // mit assign direkt den
                                // 8 Bits von LED_OUT zugeordnet.

                                    // Unser Zaehler soll mit Signal C getaktet
always @ (posedge C or posedge CLR) // und mit CLR zurueckgesetzt werden.
   begin                            // (posedge = ansteigende Flanke)
      if(CLR)                       // Wenn also CLR auf 1 ist ...
         begin
            prescaler = 0;          // dann setzte counter und prescaler auf 0
            counter = 0;
         end
      else                          // andernfalls (also wenn CLR = 0) ...
         begin
            prescaler = prescaler + 1;  // ... erhoehe den prescaler um 1
            if(prescaler == 0)          // und wenn dieser auf 0 ist (das passiert
               begin                    // einmal in 2 hoch 15 Takten!) dann ...
                  counter = counter + 1; // ... erhoehe den counter um 1
                                              if(counter[3:0] == 10)
                                                    begin
                                                          counter[3:0] = 0;
                                                    end
               end
         segments1 = segment7(counter[3:0]);
             end
end


// ----  FUNCTION: 7-segment-decoder ------------------------------------
function [7:0] segment7;            // Eine Funktion mit dem Namen und
                                    // 8 Bit grossen Ausgabewert segment7

   input [3:0] decodeinput;         // Was an die Funktion uebergeben wird,
                                    // wird innerhalb der Funktion unter dem
                                    // Namen decodeinput gefuehrt (4 Bit).

   case (decodeinput)               // Fuer den Fall, dass decodeinput ...
      4'h0 : segment7 = 8'b00111111; // ... den Wert 0 hat: segment7 = Muster "0"
                     // Dgfedcba <<< Stellen fuer Dezimalpunkt (D) und 7 Segmente
                     // als 8-Bit-Binaerzahl (daher 8'b)
                     // decodeinput schreiben wir als 4-Bit-Hexadezimalzahl, also 0-9,a-f
      4'h1 : segment7 = 8'b00000110; // das Strichmuster "1"
      4'h2 : segment7 = 8'b01011011; // diese Strichmuster muessen Sie noch aendern
      4'h3 : segment7 = 8'b01001111;
      4'h4 : segment7 = 8'b01100110;
      4'h5 : segment7 = 8'b01101101;
      4'h6 : segment7 = 8'b11111101;
      4'h7 : segment7 = 8'b00000111;
      4'h8 : segment7 = 8'b01111111;
      4'h9 : segment7 = 8'b11101111;
      4'ha : segment7 = 8'b01110111;
      4'hb : segment7 = 8'b01111111;
      4'hc : segment7 = 8'b00111001;
      4'hd : segment7 = 8'b00111111;
      4'he : segment7 = 8'b01111001;
      4'hf : segment7 = 8'b01110001;
   endcase                          // Ende der case-Anweisung
endfunction                         // Ende der Funktion

endmodule
```

Abbildung 11: Quellcode für die 4. Aufgabe

```
`timescale 1ns / 1ps // Diese Zeitskala (Schrittweite, Aufloesung)
// ist nur bei Simulationen wichtig
//////////////////////////////////////////////////////////////////////////////////
// Company: // Engineer: VERSCHIEDENE KOMMENTARE
// Create Date: 12:22:32 01/08/2010
// ...
//////////////////////////////////////////////////////////////////////////////////
module testXC9572( // Jedes Verilog-Programm ist ein Modul mit einem Namen
   C, CE, CLR, // und in runden Klammern muessen
   segments1, segments2, LED_OUT // alle Ein-/Ausgangssignale genannt werden
   );

input C;                    // Signale als EINgaenge definieren
input CE; input CLR;        // C = Clock, CLR = Clear, jeweils 1 Signal
output [7:0] segments1;     // Signale als AUSgaenge definieren
output [7:0] segments2;     // hier haben wir z.B. Signalgruppen von
output [7:0] LED_OUT;       // 8 Bits, daher die Angabe [7:0]

  reg [7:0] segments1;     // Das ist erstmal noch auskommentiert,
  reg [7:0] segments2;     // weil wir es erst spaeter brauchen.
  reg [14:0] prescaler;     // reg = Register, das ist praktisch ein D-Flipflop.
  reg [3:0]  counter;       // Hier haben wir Register mit 15 bzw. 8 Bit.
  reg [3:0] counter2;

assign LED_OUT = counter;   // Der Inhalt des 8-Bit-Zaehlers "counter" wird
                            // mit assign direkt den
                            // 8 Bits von LED_OUT zugeordnet.

                                  // Unser Zaehler soll mit Signal C getaktet
always @ (posedge C or posedge CLR)  // und mit CLR zurueckgesetzt werden.
   begin                          // (posedge = ansteigende Flanke)
      if(CLR)                     // Wenn also CLR auf 1 ist ...
         begin
            prescaler = 0;        // dann setzte counter und prescaler auf 0
            counter = 0;
                              counter2 = 0;
         end
      else                        // andernfalls (also wenn CLR = 0) ...

                              if (CE)
                                    begin
                                          prescaler = prescaler +1;
                                          if(prescaler == 0)       // und wenn dieser auf 0 ist (das passiert
                                                begin
// einmal in 2 hoch 15 Takten!) dann ...
                                                      counter = counter + 1; // ... erhoehe den counter um 1
                                                      if(counter[3:0] == 10)
                                                            begin
                                                                  counter[3:0] = 0;
                                                                  counter2[3:0] = counter2 + 1;
                                                            end
                                                      if(counter2[3:0] == 10)
                                                            begin
                                                                  counter2[3:0] = 0;
                                                            end
                                                end
                                    segments1 = segment7(counter[3:0]);
                                    segments2 = segment7(counter2[3:0]);
                                    end

       end


// ----  FUNCTION: 7-segment-decoder ------------------------------------
function [7:0] segment7;          // Eine Funktion mit dem Namen und
                                  // 8 Bit grossen Ausgabewert segment7

   input [3:0] decodeinput;       // Was an die Funktion uebergeben wird,
                                  // wird innerhalb der Funktion unter dem
                                  // Namen decodeinput gefuehrt (4 Bit).

   case (decodeinput)             // Fuer den Fall, dass decodeinput ...
      4'h0 : segment7 = 8'b00111111; // ... den Wert 0 hat: segment7 = Muster "0"
                     // Dgfedcba <<< Stellen fuer Dezimalpunkt (D) und 7 Segmente
                     // als 8-Bit-Binaerzahl (daher 8'b)
                     // decodeinput schreiben wir als 4-Bit-Hexadezimalzahl, also 0-9,a-f
      4'h1 : segment7 = 8'b00000110; // das Strichmuster "1"
      4'h2 : segment7 = 8'b01011011; // diese Strichmuster muessen Sie noch aendern
      4'h3 : segment7 = 8'b01001111;
      4'h4 : segment7 = 8'b01100110;
      4'h5 : segment7 = 8'b01101101;
      4'h6 : segment7 = 8'b11111101;
      4'h7 : segment7 = 8'b00000111;
      4'h8 : segment7 = 8'b01111111;
      4'h9 : segment7 = 8'b11101111;
      4'ha : segment7 = 8'b01110111;
      4'hb : segment7 = 8'b01111111;
      4'hc : segment7 = 8'b00111001;
      4'hd : segment7 = 8'b00111111;
      4'he : segment7 = 8'b01111001;
      4'hf : segment7 = 8'b01110001;
   endcase                        // Ende der case-Anweisung
endfunction                       // Ende der Funktion

endmodule
```

Abbildung 12: Quellcode für die 5. Aufgabe

```
`timescale 1ns / 1ps // Diese Zeitskala (Schrittweite, Aufloesung)
// ist nur bei Simulationen wichtig
//////////////////////////////////////////////////////////////////////////////////
// Company: // Engineer: VERSCHIEDENE KOMMENTARE
// Create Date: 12:22:32 01/08/2010
// ...
//////////////////////////////////////////////////////////////////////////////////
module testXC9572( // Jedes Verilog-Programm ist ein Modul mit einem Namen
    C, CE, CLR, // und in runden Klammern muessen
    segments1, segments2, LED_OUT // alle Ein-/Ausgangssignale genannt werden
    );

input C;                        // Signale als EINgaenge definieren
input CE; input CLR;            // C = Clock, CLR = Clear, jeweils 1 Signal
output [7:0] segments1;         // Signale als AUSgaenge definieren
output [7:0] segments2;         // hier haben wir z.B. Signalgruppen von
output [7:0] LED_OUT;           // 8 Bits, daher die Angabe [7:0]

  reg [7:0] segments1;        // Das ist erstmal noch auskommentiert,
  reg [7:0] segments2;        // weil wir es erst spaeter brauchen.
  reg [14:0] prescaler;       // reg = Register, das ist praktisch ein D-Flipflop.
  reg [3:0]  counter;         // Hier haben wir Register mit 15 bzw. 8 Bit.
  reg [3:0] counter2;

assign LED_OUT = counter;     // Der Inhalt des 8-Bit-Zaehlers "counter" wird
                              // mit assign direkt den
                              // 8 Bits von LED_OUT zugeordnet.

                                    // Unser Zaehler soll mit Signal C getaktet
always @ (posedge C or posedge CLR) // und mit CLR zurueckgesetzt werden.
    begin                           // (posedge = ansteigende Flanke)
        if(CLR)                     // Wenn also CLR auf 1 ist ...
            begin
                prescaler = 0;      // dann setzte counter und prescaler auf 0
                counter = 0;
                                counter2 = 0;
                                segments1 = segment7(counter[3:0]);
                                segments2 = segment7(counter2[3:0]);
            end
        else                        // andernfalls (also wenn CLR = 0) ...

                                if (CE)
                                        begin
                                                prescaler = prescaler +1;
                                        if(prescaler == 0)      // und wenn dieser auf 0 ist (das passiert
                                                begin
// einmal in 2 hoch 15 Takten!) dann ...
                                                        counter = counter + 1; // ... erhoehe den counter um 1
                                                        if(counter[3:0] == 10)
                                                                begin
                                                                        counter[3:0] = 0;
                                                                        counter2[3:0] = counter2 + 1;
                                                                end
                                                        if(counter2[3:0] == 10)
                                                                begin
                                                                        counter2[3:0] = 0;
                                                                end
                                                end
                                        segments1 = segment7(counter[3:0]);
                                        segments2 = segment7(counter2[3:0]);
                                        end

        end


// ----  FUNCTION: 7-segment-decoder ------------------------------------
function [7:0] segment7;            // Eine Funktion mit dem Namen und
                                    // 8 Bit grossen Ausgabewert segment7

    input [3:0] decodeinput;        // Was an die Funktion uebergeben wird,
                                    // wird innerhalb der Funktion unter dem
                                    // Namen decodeinput gefuehrt (4 Bit).

    case (decodeinput)              // Fuer den Fall, dass decodeinput ...
        4'h0 : segment7 = 8'b00111111; // ... den Wert 0 hat: segment7 = Muster "0"
                                // Dgfedcba <<< Stellen fuer Dezimalpunkt (D) und 7 Segmente
                                // als 8-Bit-Binaerzahl (daher 8'b)
                                // decodeinput schreiben wir als 4-Bit-Hexadezimalzahl, also 0-9,a-f
        4'h1 : segment7 = 8'b00000110; // das Strichmuster "1"
        4'h2 : segment7 = 8'b01011011; // diese Strichmuster muessen Sie noch aendern
        4'h3 : segment7 = 8'b01001111;
        4'h4 : segment7 = 8'b01100110;
        4'h5 : segment7 = 8'b01101101;
        4'h6 : segment7 = 8'b11111101;
        4'h7 : segment7 = 8'b00000111;
        4'h8 : segment7 = 8'b01111111;
        4'h9 : segment7 = 8'b11101111;
        4'ha : segment7 = 8'b01110111;
        4'hb : segment7 = 8'b01111111;
        4'hc : segment7 = 8'b00111001;
        4'hd : segment7 = 8'b00111111;
        4'he : segment7 = 8'b01111001;
        4'hf : segment7 = 8'b01110001;
    endcase                         // Ende der case-Anweisung
endfunction                         // Ende der Funktion

endmodule
```

Abbildung 13: Quellcode für die 5. Aufgabe