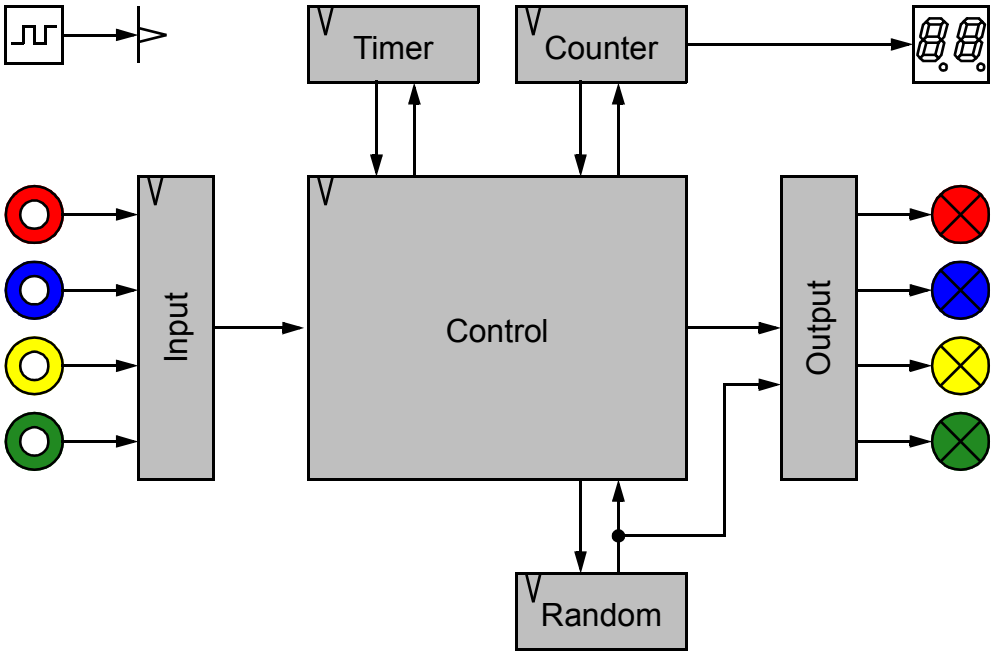
**Universität Stuttgart**
**Institut für Kommunikationsnetze und Rechnersysteme**
**Prof. Dr.-Ing. A. Kirstädter**

# Entwurf digitaler Systeme

## Übung 2 Implementierung des Spiels *Senso* in VHDL

In dieser Übung werden Sie das aus dem Kapitel 1.3 des Vorlesungsskripts bekannte Spiel *Senso* in VHDL implementieren. Folgende Abbildung zeigt die Grundstruktur der Hardware-Implementierung des Spiels.



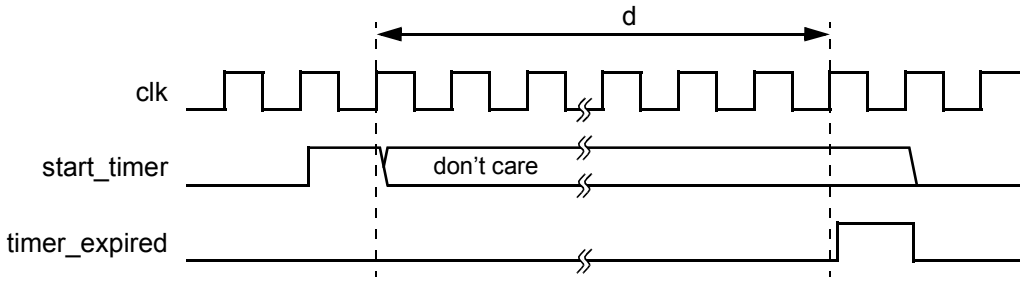

In den Aufgaben dieser Übung werden Sie die einzelnen Komponenten des Spiels *Senso* in VHDL implementieren und testen. Legen Sie dazu ein *HDL Designer*-Projekt an, um darin alle Komponenten für das Spiel *Senso* zu verwalten.

## Aufgabe 1 Zeitgeber

Im Spiel *Senso* wird zur Steuerung der Ablaufgeschwindigkeit beim Vorspielen einer Sequenz ein Zeitgeber benötigt. Zur Steigerung des Schwierigkeitsgrades in höheren Levels soll diese Ablaufgeschwindigkeit sukzessive erhöht werden können. In dieser Aufgabe werden Sie einen Zeitgeber in Form einer Komponente *timer* realisieren, deren Entity nachfolgend gegeben ist.

```
entity timer is
  port( clk, res_n:    in  std_logic;
        start_timer:   in  std_logic;
        dec_duration:  in  std_logic;
        res_duration:  in  std_logic;
        timer_expired: out std_logic);
end entity timer;
```

Das Eingangssignal *start_timer* aktiviert den Zeitgeber. Nachdem eine gewisse Zeitdauer *d* abgelaufen ist, setzt der Zeitgeber das Ausgangssignal *timer_expired* für die Dauer genau einer Taktperiode. Dies erfolgt unabhängig davon, ob das Eingangssignal *start_timer* weiterhin anliegt oder nicht. Folgende Abbildung zeigt ein Signal-Zeit-Diagramm für die Komponente *timer*.




Zunächst soll die Komponente *timer* mit einer konstanten Zeitdauer *d* implementiert werden. Das heißt, dass die Eingänge *dec_duration* und *res_duration* zunächst keine Funktion haben.

**Frage 1** Implementieren Sie die Komponente *timer* mit konstanter Zeitdauer $d = 500\,\text{ms}$. Die Frequenz des Taktsignals *clk* betrage 50 MHz.

**Frage 2** Testen Sie Ihre Implementierung in einer Simulation.

Nun soll die Komponente *timer* erweitert werden, sodass sie die Funktion der Eingänge *dec_duration* und *res_duration* implementiert. Das Eingangssignal *dec_duration* verkürzt die Zeitdauer *d*. Mit dem Eingangssignal *res_duration* kann die Zeitdauer *d* auf einen vordefinierten Initialwert zurückgesetzt werden.

**Frage 3** Erweitern Sie Ihre Komponente *timer* entsprechend.

**Frage 4** Testen Sie Ihre erweiterte Komponente *timer* in einer Simulation.

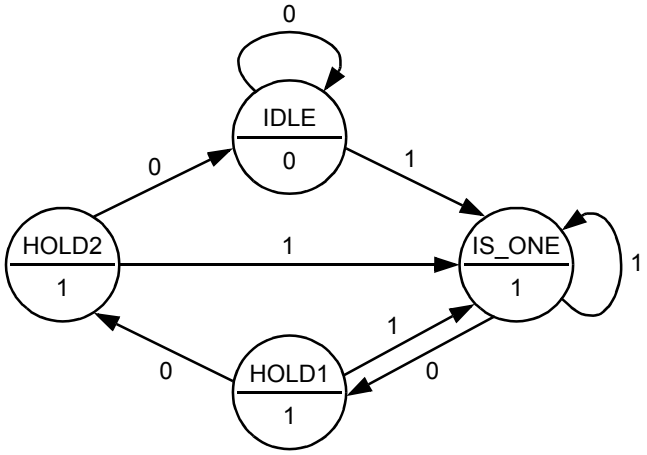
**Frage 5** Synthetisieren Sie Ihre Komponente *timer* mit Hilfe des Synthesewerkzeuges *Quartus* und untersuchen Sie das Syntheseergebnis.
Setzten Sie dazu im *HDL Designer* die Design Root auf den Timer und führen Sie den *Quartus Synthesis Flow* aus. Starten Sie dann das Synthesewerkzeug in einer Konsole durch Eingabe des Befehls *quartus*. Wählen Sie bei den *Device Settings* einen *Cyclone V* mit dem Namen *5CGXFC5C6F27C7* aus.

# 1 Tipps und Tricks zu Aufgabe 2

## 1.1 Implementierung eines Moore-Automaten in VHDL

Eine schaltungstechnische Realisierung eines Moore-Automaten kann grundsätzlich aus drei Teilen aufgebaut werden. Einem Zustandsspeicher, einer Zustandsübergangskombinatorik und einer Ausgangskombinatorik. Der Zustandsspeicher hält den aktuellen Zustand. Die Zustandsübergangskombinatorik ermittelt in Abhängigkeit vom aktuellen Zustand und den Werten der Eingangssignale den Folgezustand. Die Ausgangskombinatorik leitet vom aktuellen Zustand die Werte der Ausgangssignale ab.

Als Beispiel wird hier die Implementierung eines Moore-Automaten mit folgendem Zustands-Übergangsdiagramm dargestellt.




Eine Implementierung dieses Automaten in Form einer VHDL-Architecture *behav* zu einer Komponente *fsm* mit Eingangsport *x* und Ausgagnsport *y* ist folgend angegeben.

```
architecture behav of fsm is
  type state_t is (IDLE, IS_ONE, HOLD1, HOLD2);
  signal current_state, next_state: state_t;
begin

  state: process(clk, res_n) is
  begin
    if res_n = '0' then
      current_state <= IDLE;
    else
      if clk'event and clk = '1' then
        current_state <= next_state;
      end if;
    end if;
  end process state;

  output: process(current_state) is
  begin
    case current_state is
    when IDLE =>
      y <= '0';
    when IS_ONE | HOLD1 | HOLD2 =>
      y <= '1';
    end case;
  end process output;
```

```
  state_transition: process(current_state, x) is
  begin
    case current_state is
    when IDLE =>
      if x = '1' then
        next_state <= IS_ONE;
      else
        next_state <= IDLE;
      end if;
    when IS_ONE =>
      if x = '1' then
        next_state <= IS_ONE;
      else
        next_state <= HOLD1;
      end if;
    when HOLD1 =>
      if x = '1' then
        next_state <= IS_ONE;
      else
        next_state <= HOLD2;
      end if;
    when HOLD2 =>
      if x = '1' then
        next_state <= IS_ONE;
      else
        next_state <= IDLE;
      end if;
    end case;
  end process state_transition;

end architecture behav;
```

Die drei Prozesse *state*, *output* und *state_transition* kommunizieren über die Signale *current_state* und *next_state*. Der Prozess *state* realisiert den Zustandsspeicher des Automaten. Der Prozess *output* realisiert die Ausgangskombinatorik des Automaten. Der Prozess *state_transition* realisiert die Zustandsübergangskombinatorik des Automaten.

Für die Signale *current_state* und *next_state* wurde ein Aufzählungstyp *state_t* deklariert, dessen Wertemenge die Namen der vier Zustände des Automaten umfasst. Mit Hilfe dieses Aufzählungstyps wird im VHDL-Modell des Zustandsautomaten von der Codierung der Zustände abstrahiert. Dadurch wird das VHDL-Modell besser lesbar und die Codierung der Zustände bleibt ein Freiheitsgrad bei der Synthese der Schaltung.

## Aufgabe 2 Tasterschnittstelle

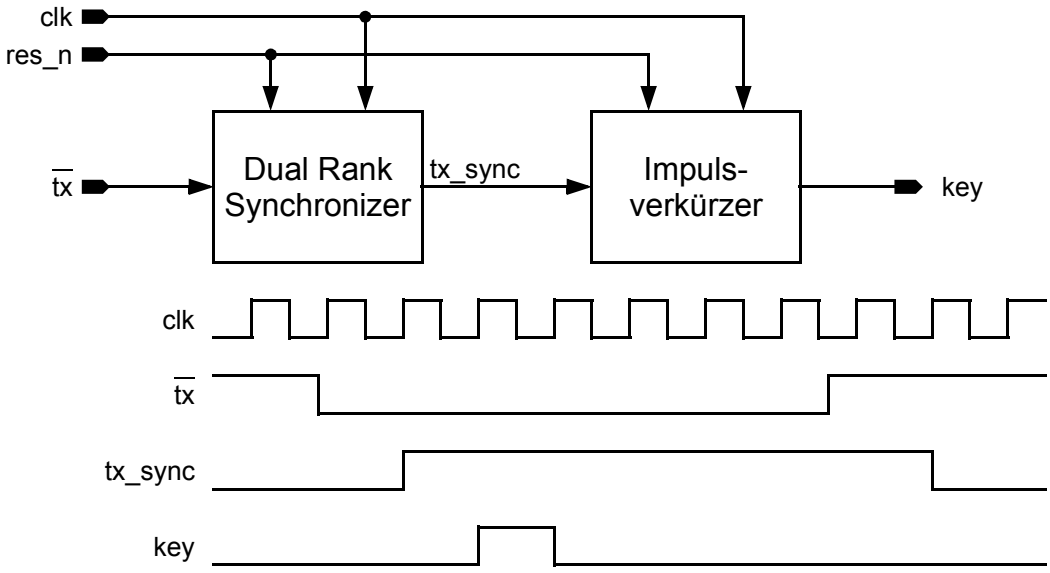
Für die Steuerung des Spiels *Senso* ist es erforderlich, die Eingangssignale der vier Eingabetaster auszuwerten. Bei der Auswertung der vier Tastersignale müssen zwei Probleme gelöst werden:

- Die Tastersignale können sich unabhängig vom Systemtakt *clk* jederzeit ändern. Es handelt sich also um asynchrone Signale bezogen auf den Takt *clk*. Diese dürfen nicht direkt als Eingangssignale eines synchronen sequenziellen Netzwerks verwendet werden, da sonst Setup- und Hold-Zeiten der Flipflops verletzt werden können.
- Ein Tastendruck soll unabhängig von seiner Dauer sicher genau einmal erkannt werden.

Störeffekte durch mechanisches Prellen der Taster werden auf dem FPGA-Entwicklungsboard *Cyclone V GX Starter Kit* durch eine Entprellschaltung kompensiert. Es muss also keine Schaltung zur Entprellung der Tastersignale im FPGA realisiert werden.

In dieser Aufgabe entwickeln Sie eine Komponente *button.* Diese soll das vom Taster erhaltene Eingangssignal mit dem Taktsignal *clk* synchronisieren und den Impuls des Tastersignals auf genau eine Taktperiode verkürzen.

Folgende Abbildung zeigt die Komponente *button* mit den beiden Stufen zur Synchronisation und zur Impulsverkürzung.

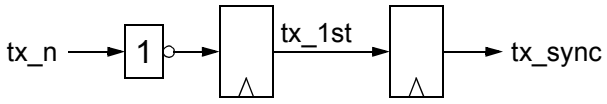



Die Entity der Komponente *button* ist nachfolgend gegeben.

```
entity button is
  port( clk, res_n: in  std_logic;
        tx_n:       in  std_logic;
        key:        out std_logic);
end entity button;
```

Die Synchronisation des asynchronen Tastersignals kann durch ein einfaches Synchronisations-Flipflop erreicht werden. Es ist jedoch im Sinne der MTBF (Mean Time Between Failure) sicherer, wenn Sie einen Dual Rank Synchronizer verwenden (siehe "Designrichtlinien"-Kapitel der Vorlesung). Ein Dual Rank Synchronizer besteht aus zwei hintereinandergeschalteten

Synchronisations-Flipflops. Folgende Abbildung zeigt den Aufbau des Dual Rank Synchronizers für die Komponente *button*.




Nachfolgend sind vier Varianten zur Verhaltensbeschreibung dieses Dual Rank Synchronizers in der Architecture der Komponente *button* angegeben.

a)
```
architecture behav of button is
  signal tx_1st, tx_sync: std_logic;
begin
  drs: process(clk, res_n) is
  begin
    if res_n = '0' then
      tx_1st  <= '0';
      tx_sync <= '0';
    else
      if clk'event and clk = '1' then
        tx_1st  <= not tx_n;
        tx_sync <= tx_1st;
      end if;
    end if;
  end process drs;
end architecture behav;
```

b)
```
architecture behav of button is
  signal tx_1st, tx_sync: std_logic;
begin
  drs: process(clk, res_n) is
  begin
    if res_n = '0' then
      tx_1st  <= '0';
      tx_sync <= '0';
    else
      if clk'event and clk = '1' then
        tx_sync <= tx_1st;
        tx_1st  <= not tx_n;
      end if;
    end if;
  end process drs;
end architecture behav;
```

c)
```
architecture behav of button is
  signal tx_sync: std_logic;
begin
  drs: process(clk, res_n) is
    variable tx_1st: std_logic;
  begin
    if res_n = '0' then
      tx_1st  := '0';
      tx_sync <= '0';
    else
      if clk'event and clk = '1' then
        tx_1st  := not tx_n;
        tx_sync <= tx_1st;
      end if;
    end if;
  end process drs;
end architecture behav;
```

d)
```
architecture behav of button is
  signal tx_sync: std_logic;
begin
  drs: process(clk, res_n) is
    variable tx_1st: std_logic;
  begin
    if res_n = '0' then
      tx_1st  := '0';
      tx_sync <= '0';
    else
      if clk'event and clk = '1' then
        tx_sync <= tx_1st;
        tx_1st  := not tx_n;
      end if;
    end if;
  end process drs;
end architecture behav;
```

**Frage 1** Welche der vier gegebenen Varianten beschreiben das Verhalten des Dual Rank Synchronizers der Komponente *button* tatsächlich? Falls es mehrere Varianten sind, welche davon würden Sie bevorzugen und warum?

Der Impulsverkürzer der Komponente *button* soll als Moore-Automat realisiert werden. Seine Aufgabe ist es, einen beliebig langen Eingangsimpuls auf die Dauer genau einer Taktperiode zu verkürzen.

**Frage 2** Zeichnen Sie einen Moore-Graphen für den Automat zur Impulsverkürzung.

**Frage 3** Ergänzen Sie die Architecture der Komponente *button* um eine Implementierung des Automaten zur Impulsverkürzung.

**Frage 4** Testen Sie die Komponente *button* in einer Simulation.

Eine Komponente *input* soll die Auswertung der Eingangsignale aller vier Eingabetaster zusammenfassen und geeignete Signale für die Hauptsteuerung erzeugen. Die Entity dieser Komponente ist nachfolgend gegeben.

```
entity input is
  port( clk, res_n: in  std_logic;
        key_in_n:   in  std_logic_vector(3 downto 0);
        key_valid:  out std_logic;
        key_color:  out std_logic_vector(1 downto 0));
end entity input;
```

Für die vier low-aktiven Tastersignale ist der Eingangsport *key_in_n* vorgesehen. Wird eine der Tasten gedrückt, soll dies von der Komponente *input* durch die Signale *key_valid* und *key_color* angezeigt werden. Für jeden Tastendruck wird *key_valid* für genau eine Taktperiode aktiv gesetzt. Gleichzeitig wird am Ausgangsport *key_color* die Farbe der Taste ausgegeben.

**Frage 5** Implementieren Sie die Komponente *input*. Verwenden Sie dazu vier Instanzen der Komponente *button* und leiten Sie aus deren Ausgangssignalen die Signale für die Ausgangsports *key_valid* und *key_color* der Komponente *input* ab.

# 2 Tipps und Tricks zu Aufgabe 3

## 2.1 Typwandlung zwischen *std_logic_vector* und *integer*

Für die Wandlung zwischen *integer* und *std_logic_vector* benötigen sie die Typen *unsigned* bzw. *signed* aus dem Package *ieee.numeric_std*. Diese können Sie mit folgenden Anweisungen einbinden.

```
library ieee; use ieee.numeric_std.all;
```

Einen *integer i* können Sie wie folgt in einen *std_logic_vector v* der Länge *l* konvertieren.

```
v := std_logic_vector(to_unsigned(i, l));
```

Einen *std_logic_vector v* können Sie wie folgt in einen *integer i* konvertieren.

```
i := to_integer(unsigned(v));
```

## 2.2 For-Schleife in VHDL

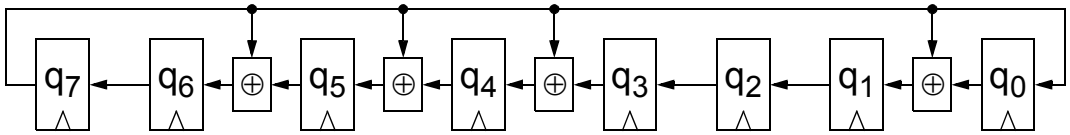
Folgender nicht synthetisierbarer Prozess weist jeweils zur steigenden Taktflanke einem vier Bit breiten *std_logic_vector* Signal *four_bit* nacheinander alle möglichen Werte zu.

```
process is
begin
  for i in 0 to 15 loop
    four_bit <= std_logic_vector(to_unsigned(i, 4));
    wait on clk until clk = '1';
  end loop;
end process;
```

Die for-Schleife deklariert eine Iterator-Variable und einen diskreten Wertebereich, über den iteriert wird. Der Typ der Iterator-Variable leitet sich dabei vom Wertebereich ab.

## Aufgabe 3 Komponente *Random*

Die Komponente Random dient dazu, eine zufällige Sequenz zu erzeugen, die vom Spiel vorgespielt wird und dann vom Spieler nachzuspielen ist. Nachfolgende Abbildung zeigt eine Möglichkeit für eine schaltungstechnische Realisierung eines Pseudozufallszahlengenerators in Form eines rückgekoppelten Schieberegisters.



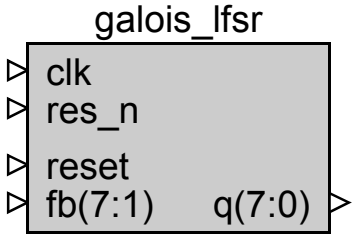

Die acht Flipflops $q_0$ bis $q_7$ repräsentieren eine 8 Bit breite Binärzahl $q$, wobei $q_0$ das niederwertigste Bit und $q_7$ das höchstwertigste Bit darstellt. Das Rückkopplungsmuster realisiert eine Zustandsübergangsfunktion $f$ mit $q^{n+1} = f(q^n)$.

**Frage 1** Zu einem Zeitpunkt $n$ gelte $q^n = 250$. Geben Sie $q^{n+1}$, $q^{n+2}$ und $q^{n+3}$ an. Wie wird sich die Schaltung im weiteren zeitlichen Verlauf verhalten?

**Frage 2** Wie verhält sich die Schaltung, wenn $q^n = 0$ gilt?

Die Zustandsübergangsfunktion $f$ ordnet jedem Zustand $q^n$ einen eindeutigen Folgezustand $q^{n+1}$ zu. Aufgrund der Struktur des rückgekoppelten Schieberegisters lässt sich jedem Zustand $q^n$ auch ein eindeutiger Vorgänger $q^{n-1}$ zuordnen. Daraus folgt, dass das Zustandsübergangsdiagramm des rückgekoppelten Schieberegisters ausschließlich aus disjunkten Zyklen besteht.

Mit Hilfe einer Simulation soll nun untersucht werden, ob es Rückkopplungsmuster gibt, bei denen ein rückgekoppeltes Schieberegister einen maximal langen Zyklus aufweist, in dem alle von 0 verschiedenen Zahlen enthalten sind. Dazu werde zunächst eine VHDL-Komponente *galois_lfsr* implementiert. Diese beinhaltet ein rückgekoppeltes Schieberegister, dessen Rückkopplungsmuster durch einen Eingangsport *fb* definiert werden kann. Folgende Abbildung zeigt die Schnittstelle der Komponente *galois_lfsr*.




Das i-te Element des Eingangs *fb* gibt an, ob sich in dem rückgekoppelten Schieberegister vor dem Eingang des Flipflops $q_i$ ein Exklusiv-Oder befindet (*fb*(i) = '1') oder nicht (*fb*(i) = '0'). Das Rückkopplungsmuster der oben angegebenen Schaltung entspricht also dem Wert "0111001" am Eingang *fb*. Bei *reset* handelt es sich um einen synchronen Rücksetzeingang. Im Gegensatz zum asynchronen Rücksetzeingang *res_n* wird *reset* synchron zum Takt ausgewertet und ist high-aktiv.

**Frage 3** Implementieren Sie die oben angegebene Schnittstelle in Form einer VHDL-Entity *galois_lfsr*.

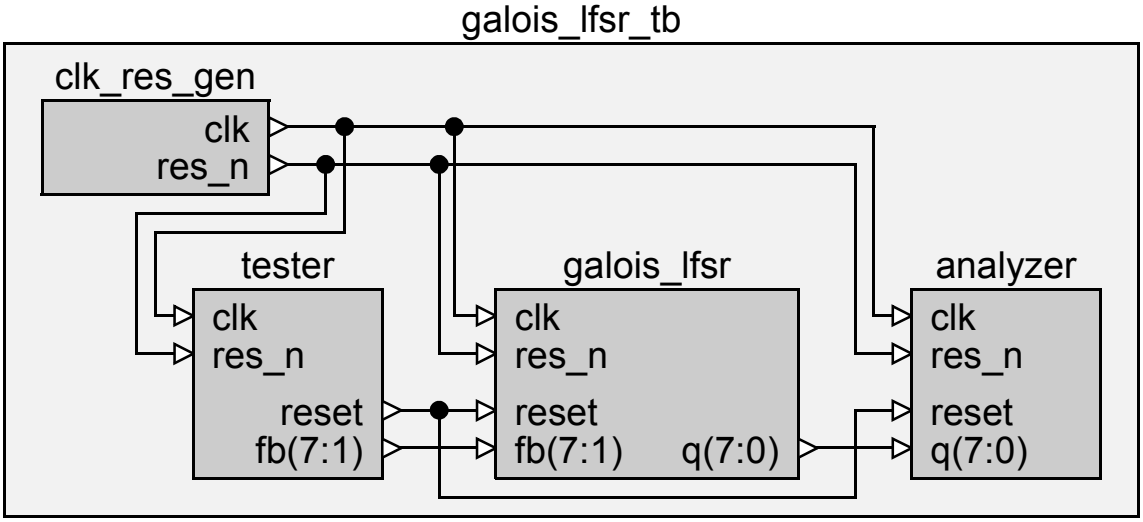
**Frage 4** Implementieren Sie das rückgekoppelte Schieberegister in Form einer VHDL-Architecture *behav* zur Entity *galois_lfsr*. Nachfolgenden Auszug aus einer

VHDL-Verhaltensbeschreibung können Sie dabei verwenden, um das rückgekoppelte Schieberegister zu modellieren.

```
if q_int(7) = '1' then
  q_int := (q_int(6 downto 0) xor fb) & q_int(7);
else
  q_int := q_int(6 downto 0) & q_int(7);
end if;
```

**Frage 5** Lässt sich das rückgekoppelte Schieberegister auch mit Hilfe einer for-Schleife synthetisierbar beschreiben? Geben Sie gegebenenfalls eine alternative Beschreibung zu Frage 4 an.

Für die Untersuchung des rückgekoppelten Schieberegisters mit Hilfe einer Simulation muss eine geeignete Testbench erstellt werden. Die nachfolgende Abbildung zeigt den Aufbau einer solchen Testbench.




Neben der zu untersuchenden Komponente *galois_lfsr* und einem Generator *clk_res_gen* zur Erzeugung des Takt- und Rücksetzsignals enthält die Testbench eine Komponente *tester* und eine Komponente *analyzer*. Die Komponente *tester* dient dazu, an die Eingänge *reset* und *fb* von *galois_lfsr* geeignete Signalverläufe anzulegen. Die Komponente *analyzer* dient dazu, die Zykluslänge zu ermitteln.

**Frage 6** Erstellen Sie eine Komponente *analyzer* (Entity und Architecture). Verwenden Sie in der Architecture den nachfolgend angegebenen Prozess.

```
process is
  variable cycle_length_counter: integer;
  variable start_value: std_logic_vector(7 downto 0);
begin
  wait until res_n = '1';
  loop
    wait on clk until clk = '1' and reset = '1';
    cycle_length_counter := 1;
    wait on clk until clk = '1' and reset = '0';
    start_value := q;
    wait on clk until clk = '1';
    while q /= start_value loop
      cycle_length_counter := cycle_length_counter + 1;
      wait on clk until clk = '1';
    end loop;
    report "cycle length: " & integer'image(cycle_length_counter);
```

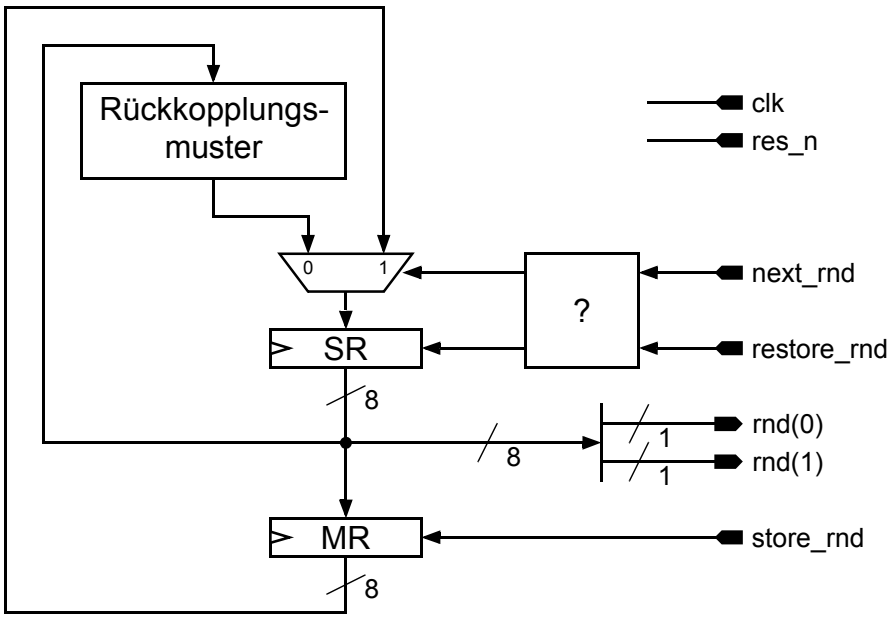
```
  end loop;
end process;
```

**Frage 7** Erstellen Sie eine Komponente *tester* (Entity und Architecture). Implementieren Sie die Architecture in Form einer Verhaltensbeschreibung. Das darin realisierte Testszenario soll alle möglichen Rückkopplungsmuster untersuchen. Dazu wird das jeweilige Rückkopplungsmuster am Ausgang *fb* angelegt und der Ausgang *reset* für die Dauer einer Taktperiode aktiviert. Danach wird die maximal mögliche Zykluslänge abgewartet, bevor das nächste Rückkopplungsmuster angelegt wird.

**Frage 8** Erstellen Sie die Testbench *galois_lfsr_tb* in Form einer Strukturbeschreibung.

**Frage 9** Simulieren Sie die Testbench und ermitteln Sie mit Hilfe der Simulation mögliche Rückkopplungsmuster mit maximaler Zykluslänge.

Für das Spiel *Senso* ist es erforderlich, dass dieselbe pseudozufällige Sequenz reproduziert werden kann. Dies können Sie dadurch erreichen, dass Sie einen Startwert eines Pseudozufallszahlengenerators (Seed) in einem Register sichern und bei Bedarf in das Register des Generators zurückschreiben. Die Struktur einer entsprechenden Schaltung ist in der folgenden Abbildung angegeben.




Im Register *SR* liegt die aktuelle Pseudozufallszahl. Mit dem Signal *store_rnd* wird der Wert aus *SR* in das Register *MR* übernommen. Wird das Signal *restore_rnd* aktiviert, so wird der Wert aus *MR* wieder in *SR* zurückgeschrieben. Anderenfalls wird für *next_rnd* = '1' die nächste Pseudozufallszahl erzeugt.

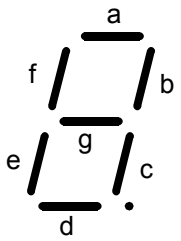
Die „?"-Schaltung leitet aus den Signalen *next_rnd* und *restore_rnd* das Steuersignal für den Multiplexer und das Freigabesignal für das Register *SR* ab. Aus den 8 Bit des Registers *SR* werden zwei beliebige Bits herausgegriffen, um die Ausgangssignale *rnd*(0) und *rnd*(1) zu treiben.

**Frage 10** Setzen Sie die in der Abbildung angegebene Schaltung in eine VHDL-Komponente *random* um. Wählen Sie dabei ein Rückkopplungsmuster mit maximaler Zykluslänge.

# 3 Tipps und Tricks zu Aufgabe 4

## 3.1 Siebensegmentanzeigen

Üblicherweise werden die Segmente einer Siebensegmentanzeige mit Buchstaben benannt. Nachfolgende Abbildung zeigt die übliche Bennenung der Segmente einer Siebensegmentanzeige.




Auf dem im Rahmen dieser Übung als Hardwareplattform verwendeten FPGA-Entwicklungsboard *Cyclone V GX Starter Kit* befinden sich vier Siebensegmentanzeigen. Die Segmente dieser Anzeigen werden low-aktiv angesteuert. Das heißt ein Segment leuchtet dann auf, wenn der entsprechende Ausgang des FPGA den Wert ’0’ aufweist.

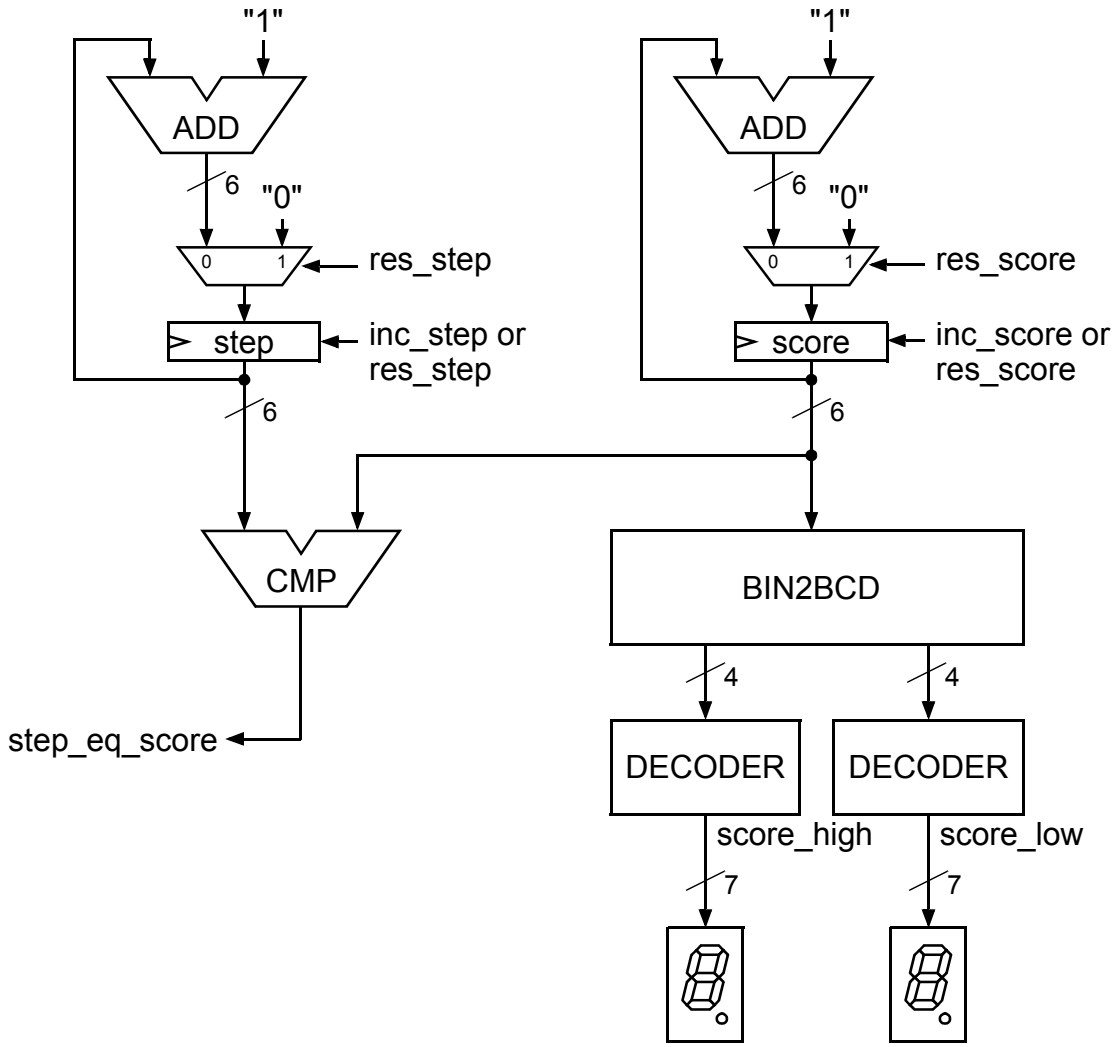
## Aufgabe 4 Zählereinheit

Zur Realisierung des Spiels *Senso* werden zwei Zähler benötigt.

- Ein Zähler *score*, der die aktuelle Sequenzlänge angibt.
- Ein Zähler *step*, der die Nummer des aktuellen Schritts innerhalb einer Sequenz angibt.

Beim Vorspielen und beim Nachspielen einer Sequenz zählt der Zähler *step* die Schritte innerhalb der Sequenz. Dazu wird er zum Beginn der Sequenz zurückgesetzt. Eine Sequenz ist beendet, wenn der Zähler *step* und der Zähler *score* denselben Wert aufweisen. Innerhalb einer Sequenz gilt also: $0 \leq step \leq score$. Da der Wert des Zählers *score* den Punktestand repräsentiert, soll er auf einer zweistelligen Siebensegmentanzeige ausgegeben werden.

Folgende Abbildung zeigt den Aufbau der Zählereinheit auf Register-Transfer-Ebene. Das Signal *res_step* soll den Zähler *step* synchron zurücksetzen. Ist während einer steigenden Taktflanke *inc_step* aktiv, so soll der Zähler *step* um eins erhöht werden. Entsprechendes gilt für den Zähler *score* und die Signale *res_score* und *inc_score*. Das Signal *step_eq_score* werde genau dann aktiv, wenn die Zähler *step* und *score* denselben Wert aufweisen. Die Komponente BIN2BCD wandelt eine 6 Bit breite Binärzahl in zwei BCD-Ziffern (BCD: Binary Coded Decimal). Die Komponente DECODER wandelt eine BCD-Ziffer in die Ansteuersignale einer Siebensegmentanzeige.



In dieser Aufgabe implementieren und testen Sie die Zählereinheit *counter*. Die Entity der Zählereinheit *counter* ist nachfolgend gegeben.

```
entity counter is
  port( clk, res_n:    in  std_logic;
        res_step:      in  std_logic;
        inc_step:      in  std_logic;
        res_score:     in  std_logic;
        inc_score:     in  std_logic;
        step_eq_score: out std_logic;
        score_low:     out std_logic_vector(6 downto 0);
        score_high:    out std_logic_vector(6 downto 0));
end entity counter;
```

**Frage 1** Nutzen Sie das VHDL-Typsystem um eine bessere Schnittstelle für die Komponente Counter zu definieren, insbesondere für die Signale *score_low* und *score_high*.

**Frage 2** Implementieren Sie die Zählereinheit. Überlegen Sie sich dazu eine geeignete Modularisierung der Einheit. Wägen Sie ab, ob Sie die Komponente mit Hilfe von Verhaltens- oder Strukturbeschreibungen realisieren möchten.

**Frage 3** Testen Sie Ihre Implementierung der Zählereinheit in einer Simulation.

## Aufgabe 5 Ausgabeeinheit

Das Spiel *Senso* weist vier LEDs auf. Mit Hilfe dieser LEDs wird die Sequenz vorgespielt, die von der Spielerin oder dem Spieler nachgespielt werden soll. Die Ansteuerung der vier LEDs erfolgt durch die Ausgabeeinheit *output*, deren Entity nachfolgend gegeben ist.

```
entity output is
  port( led_on, all_on: in  std_logic;
        color:          in  std_logic_vector(1 downto 0);
        leds:           out std_logic_vector(3 downto 0));
end entity output;
```

Die Komponente *output* wird über die Eingangssignale *led_on*, *all_on* und *color* angesteuert. Ist das Eingangssignal *led_on* aktiv, soll die LED mit der vom Eingangssignal *color* angegebenen Farbe aufleuchten. Ist das Eingangssignal *all_on* aktiv, sollen alle Leuchtdioden unabhängig von *led_on* und *color* aufleuchten.

**Frage 1** Warum besitzt die Komponente *output* keine Eingangsports für ein Taktsignal *clk* und ein asynchrones Rücksetztsignal *res_n*?

**Frage 2** Geben Sie eine Schaltung für die Komponente *output* an, die ausschließlich aus Gattern besteht.

**Frage 3** Implementieren Sie die Komponente *output* in Form einer abstrakten Verhaltensbeschreibung.

## Aufgabe 6 Hauptsteuerung

Die Hauptsteuerung (*Control* in der Abbildung am Beginn der Übung) besteht aus einem Automaten, dessen Funktionalität nachfolgend beschrieben ist.

1. Nach einem asynchronen Reset verharrt der Automat solange in einem Initialzustand, bis ein beliebiger Taster gedrückt wird. Während sich der Automat im Initialzustand befindet, erzeugt die Pseudozufallszahleneinheit *random* fortlaufend Pseudozufallszahlen. Außerdem werden die Zähler der Zählereinheit zurückgesetzt.
2. Sobald ein Taster betätigt wird, wird der Pseudozufallszahlengenerator angehalten und die aktuelle Pseudozufallszahl als Startwert gespeichert. Auf diese Weise wird ein zufälliger Startwert bestimmt, der von der Zeit bis zum Spielstart durch die Spielerin oder den Spieler abhängt.
3. Der Startwert wird wieder in den Pseudozufallszahlengenerator geladen und der Schrittzähler zurückgesetzt.
4. Die Sequenz wird vorgespielt. Der Schrittzähler der Zählereinheit dient zur Steuerung der "Schleife".
5. Nach dem Vorspielen der Sequenz wird der Startwert wieder in den Pseudozufallszahlengenerator geladen und der Schrittzähler zurückgesetzt.
6. Die Hauptsteuerung wartet auf die Betätigung eines Tasters. Nach jeder Betätigung wird der Wert von *key_color* auf Richtigkeit überprüft.
   - Hat die Spielerin oder der Spieler den falschen Taster betätigt, wird durch Aufleuchten aller LEDs signalisiert, dass das Spiel verloren ist und mit Schritt 1 fortgefahren.
   - Hat die Spielerin oder der Spieler den richtigen Taster betätigt, so wird
     - der Wert des Schrittzählers inkrementiert und mit Schritt 6 fortgefahren, falls die Sequenz noch nicht zu Ende ist,
     - sonst der Wert des Punktezählers inkrementiert und mit Schritt 3 fortgefahren.

**Frage 1** Die Hauptsteuerung soll als Moore-Automat realisiert werden. Zeichnen Sie das Zustandsübergangsdiagramm des Automaten. Geben Sie für jeden Zustand den Wert aller Ausgangssignale an.

> **Hinweis** Beachten Sie, dass die Signale *next_rn*, *inc_step*, *inc_score* und *dec_duration* für eine auszulösende Aktion jeweils nur eine Taktperiode lang aktiv sein dürfen. Anderenfalls wird die Aktion mehrfach ausgelöst. Daraus folgt, dass die Zustände, in denen diese Signale aktiv sind, **unbedingt** verlassen werden müssen.

**Frage 2** Setzen Sie den von Ihnen entworfenen Moore-Automaten in eine VHDL-Implementierung um.

**Frage 3** Setzen Sie alle Komponenten des Spiels *Senso* zusammen.

**Frage 4** Testen Sie das vollständige Spiel in einer Simulation.

**Frage 5** Synthetisieren Sie das Spiel und testen Sie es mit einem FPGA-Entwicklungsboard.