Tutor: *Oliver Keszöcze*

Matrikelnummern:
296118 Stefan Fuhrmann — Techn.Inf. Hochschule Bremen
325585 Steffen Christiansen — Techn.Inf. Hochschule Bremen
4140709 Daniel Tauritis —Inf. Uni Bremen

Übungsblatt 1

Abgabe 05.11.2015

Unsere Gruppe möchte mit bewerteten Übungszetteln und einem abschließenden Fachgespräch an der Veranstaltung teilnehmen.

Wir nutzen in den Bearbeitungen der Aufgaben einige auf StudIP verfügbare VHDL-Dateien.

Aufgabe 1 2-Bit-Zähler

$$
\begin{aligned}
S &= \{z_0, z_1, z_2, z_3\} \\
S_0 &= \{z_0\} \\
X &= \{0, 1\} \\
Y &= \{0, 1, 2, 3\}
\end{aligned}
$$

$$
\begin{aligned}
T_{0\to1} &\subseteq 1 \times z_0 \times z_1 \qquad \text{Transitionsrelation} \\
T_{1\to2} &\subseteq 1 \times z_1 \times z_2 \\
T_{2\to3} &\subseteq 1 \times z_2 \times z_3 \\
T_{3\to0} &\subseteq 1 \times z_3 \times z_0 \\
T_{0\to3} &\subseteq 0 \times z_0 \times z_3 \\
T_{1\to0} &\subseteq 0 \times z_1 \times z_0 \\
T_{2\to1} &\subseteq 0 \times z_2 \times z_1 \\
T_{3\to2} &\subseteq 0 \times z_3 \times z_2 \\
U_{z_0} &\subseteq z_0, 1 \qquad \text{Ausgaberelation} \\
U_{z_1} &\subseteq z_1, 2 \\
U_{z_2} &\subseteq z_2, 3 \\
U_{z_3} &\subseteq z_3, 4
\end{aligned}
$$

Aufgabe 2 Logische Schaltkreise mit VHDL

Der Code in dieser Abgabe wurde direkt aus unseren abgegebenen Dateien entnommen.

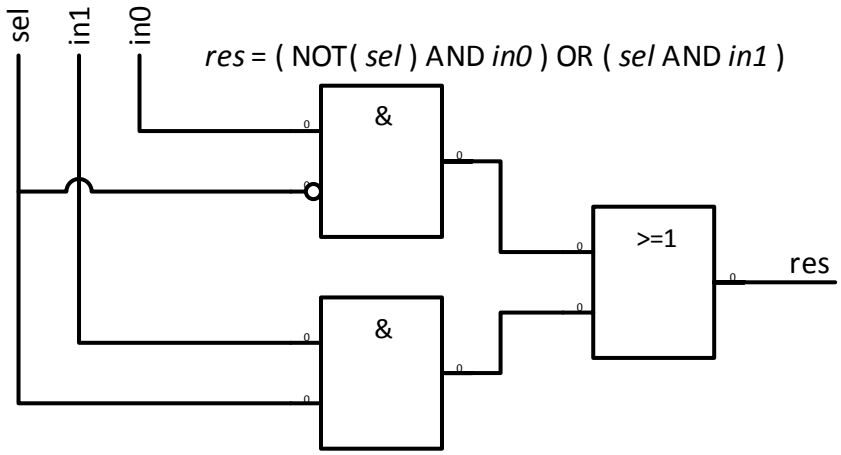
Tabelle 1: Transitionsrelation als Tabelle

	0	1
z_0	z_3	z_1
z_1	z_0	z_2
z_2	z_1	z_3
z_3	z_2	z_0

Aufgabe 2.1 2-Fach-Multiplexer (multiplexer.vhd)

Der 2-Fach-Multiplexer definiert wie folgt:

$$res = (in0 \wedge \overline{sel}) \vee (in1 \wedge sel)$$



Unsere Implementierung eines Multiplexers nutzt ein Signal `temp1`, das temporär den Ausgabewert speichert, bevor dieser unverändert an `res` ausgegeben wird. So vermeiden wir einen undefinierten Ausgabewert, da `temp1` mit dem Wert `0` initialisiert wird. Ansonsten könnte statt der Zwischenspeicherung in `temp1` auch eine direkte Ausgabe an `res` erfolgen.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity multiplexer is
5   Port (  in0 : in    STD_LOGIC;
6           in1 : in    STD_LOGIC;
7           sel : in    STD_LOGIC;
8           res : out   STD_LOGIC);
```

```
9  end multiplexer;
10
11 architecture multiplexer_impl of multiplexer is
12   -- Signal temp1 dient dazu, dass res von Beginn an definiert ist!
13   signal temp1 : STD_LOGIC := '0';
14
15   begin
16
17     -- Alle Eingänge werden in der Sensibilitaetsliste mit aufgenommen,
18     -- damit der Multiplexer sofort auf jede Eingangsaenderung reagiert.
19     multiplexer: process(sel, in0, in1)
20     begin
21
22     -- Wenn sel = 0 dann leite in0 auf temp1 um.
23     if (sel = '0') then
24       temp1 <= in0;
25     -- Sonst leite in1 auf temp1 um.
26     else
27       temp1 <= in1;
28     end if;
29
30         end process;
31
32   -- Ergebnis des Prozesses an die Ausgabe geben
33         res <= temp1;
34
35 end multiplexer_impl;
```

Testbench: 2-Eingaben-Multiplexer (multiplexer_tb.vhd)

In unserer Testbench definieren wir zunächst die Ports und benötigten Signale. Wir verwenden keinen Clock, da es sich um eine asynchrone Schaltung handelt.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY multiplexer_tb IS
5  END multiplexer_tb;
6
7  ARCHITECTURE behavior OF multiplexer_tb IS
8
9    COMPONENT multiplexer
10     PORT(
11       in0 : IN  std_logic;
12       in1 : IN  std_logic;
13       sel : IN  std_logic;
14       res : OUT  std_logic
15         );
16   END COMPONENT;
17
```

```
18  --Inputs
19  signal in0 : std_logic := '0';
20  signal in1 : std_logic := '0';
21  signal sel : std_logic := '0';
22
23  --Outputs
24  signal res : std_logic;
```

Wir erstellen ein Array, das alle möglichen In- und Outputkombinationen des Multiplexers enthält, entsprechend eine Wahrheitstabelle.

Diese werden in unserer Testschleife verwendet, um nacheinander alle Szenarien abzuarbeiten und das tatsächliche Ergebnis mit dem erwarteten Ergebnis in einer assert-Abfrage abzugleichen.

Mit zahlenreichen Konsolenausgaben kann der Test einfach verfolgt und ein potentieller Fehler schnell entdeckt werden.

```
26  BEGIN
27
28    -- Instantiate the Unit Under Test (UUT)
29    uut: multiplexer PORT MAP (
30          in0 => in0,
31          in1 => in1,
32          sel => sel,
33          res => res
34          );
35
36  -- Stimulus process
37  stim_proc: process
38
39  -- Es wird eine Wahrheitstabelle fuer die einzelnen Testfaelle verwendet,
40  -- um den eigentlichen Testcode simpel zu halten (eine Schleife).
41
42  -- Erstellen des Datentypen 'Wahrheitstabelle'
43  type wahrheitstabelle is record
44    -- inputs
45    in0, in1, sel : std_logic;
46    -- and exprected outputs
47    res : std_logic;
48        end record;
49
50        -- Erstellen der Wahrheitstabelle als Array
51  -- Die gewaehlte Wahrheitstabelle deckt alle moeglichen Einganzkombinationen
      ab.
52        type pattern_array is array (natural range <>) of wahrheitstabelle;
53  constant patterns : pattern_array :=
54    (('0','0','0','0'),
55    ('0','0','1','0'),
56    ('0','1','0','0'),
57    ('0','1','1','1'),
```

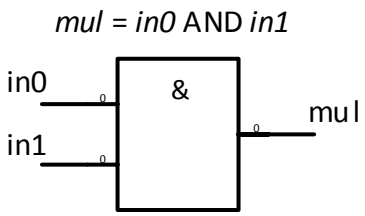
```
58      ('1','0','0','1'),
59      ('1','0','1','0'),
60      ('1','1','0','1'),
61      ('1','1','1','1')
62      );
63
64    begin
65      -- init prozesse abwarten
66      wait for 100 ns;
67
68      report "Beginn der Testbench";
69
70      -- Scheife zum durchgehen der Wahrheitstabelle
71      for i in patterns'range loop
72        report "Testschritt: " & integer'image(i);
73
74        in0 <= patterns(i).in0;
75        in1 <= patterns(i).in1;
76        sel <= patterns(i).sel;
77
78        -- sicherstellen das quantum abgelaufen ist
79        wait for 2 ns;
80
81        -- Errormeldung, wenn eine erwartete Ausgabe nicht der realen entspricht!
82        -- Abbruch wenn ein Fehler auftritt, wodurch ein solcher leichter
              entdeckt werden kann
```

Aufgabe 2.2 1-Bit-Multiplizierer (einBitMulti.vhd)

Der 1-Bit-Multiplizierer entspricht einem AND-Gate, da kein Carry-Bit erwartet wird. Dementsprechend einfach ist der eigentliche Schaltkreis.

Wie in der ersten Aufgabe nutzen wir ein temporäres und mit 0 initiiertes Signal, um keine Ungewissheit über initiale Signalwerte zu haben.

$$res = in0 \wedge in1$$



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multi is
  port (
    in0 : in STD_LOGIC;
    in1 : in STD_LOGIC;
    res : out STD_LOGIC);
end multi;

architecture multi_impl of multi is
  -- Signal temp1 dient dazu, dass res von Beginn an definiert ist!
  signal temp1 : STD_LOGIC := '0';

begin

  -- Alle Eingänge werden in der Sensibilitaetsliste mit aufgenommen,
  -- damit der Multiplexer sofort auf jede Eingangsaenderung reagiert.
  multiplier: process(in0, in1)
  begin
    -- Ein ein Bit Multiplizierer kann mit einem AND-Gatter der
    -- beiden Eingänge abgebildet werden, da kein Carry erwartet wird!
    temp1 <= in0 AND in1;
  end process;

  -- Ergebnis des Prozesses sofort an die Ausgabe geben
  res <= temp1;

end multi_impl;
```

Testbench: 1-Bit-Multiplizierer (einBitMulti_tb.vhd)

Auch in diesem Testbench verzichten wir auf einen Clock und überprüfen die Korrektheit der Entität mit einer Wahrheitstabelle. Diese fällt aufgrund der Einfachheit der Entität bedeutend einfacher aus.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY multi_tb IS
END multi_tb;

ARCHITECTURE behavior OF multi_tb IS

  -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT multi
    PORT(
         in0 : IN  std_logic;
```

```
14          in1 : IN  std_logic;
15          res : OUT  std_logic
16         );
17     END COMPONENT;
18
19    --Inputs
20    signal in0 : std_logic := '0';
21    signal in1 : std_logic := '0';
22
23    --Outputs
24    signal res : std_logic;
25
26 BEGIN
27
28   -- Instantiate the Unit Under Test (UUT)
29    uut: multi PORT MAP (
30           in0 => in0,
31           in1 => in1,
32           res => res
33         );
34
35    -- Stimulus process
36    stim_proc: process
37
38   -- Es wird eine Wahrheitstabelle fuer die einzelnen Testfaelle verwendet,
39   -- um den eigentlichen Testcode simpel zu halten (eine Schleife).
40
41   -- Erstellen des Datentypen 'Wahrheitstabelle'
42   type wahrheitstabelle is record
43   -- inputs
44   in0, in1 : std_logic;
45   -- and exprected outputs
46   res : std_logic;
47   end record;
48
49   -- Erstellen der Wahrheitstabelle als Array
50   type pattern_array is array (natural range <>) of wahrheitstabelle;
51   constant patterns : pattern_array :=
52   (('0','0','0'),
53   ('0','1','0'),
54   ('1','0','0'),
55   ('1','1','1'));
56
57   begin
58     -- init prozesse abwarten
59     wait for 100 ns;
60     report "Beginn der Testbench";
61
62     -- Scheife zum durchgehen der Wahrheitstabelle
```

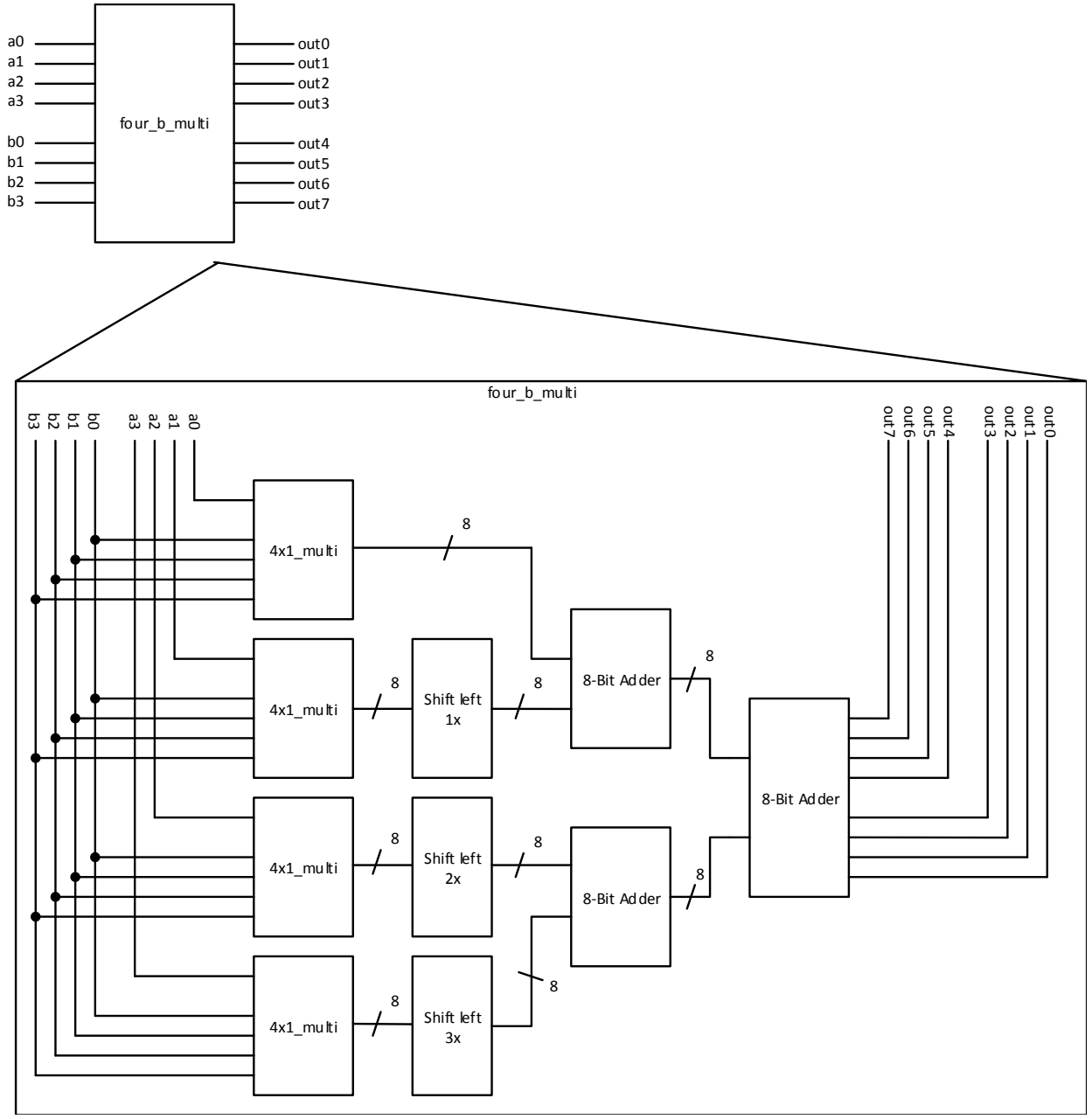
```
63      for i in patterns'range loop
64        report "Testschritt: " & integer'image(i);
65
66        in0 <= patterns(i).in0;
67        in1 <= patterns(i).in1;
68
69        wait for 2 ns;
70
71        -- Errormeldung, wenn eine erwartete Ausgabe nicht der realen entspricht!
72        -- Abbruch wenn ein Fehler auftritt, wodurch ein solcher leichter
              entdeckt werden kann
73        assert (res = patterns(i).res) report "Falscher Wert am Ausgang!"
              ↪ severity error;
74
75      end loop;
76
77      -- Ausgabe damit erkennbar wird, dass der Test erfolgreich durchlief
78      report "Ende der Testbench" severity failure;
79      wait;
80    end process;
81
82  END;
```

Aufgabe 2.3 4-Bit-Multiplizierer

Der 4-Bit-Multiplizierer gestaltet sich etwas komplexer in der Lösung als die vorherigen Aufgaben:

Unser Lösungsansatz ist, den Multiplikator mit den einzelnen Stellen (Bits) des Faktors zu multiplizieren (von rechts nach links). So erhalten wir immer den ursprünglichen Wert des Multiplikators oder 0 als Ergebnis.

Nach jeder Multiplikation die nächste Multiplikation um ein Bit nach links zu verschieben, sodass immer größere Zahlen entstehen. Anschließend werden alle Zwischenergebnisse in einem 8-Bit-Addierer zusammengerechnet und wir erhalten den multiplizierten Wert.


a0
a1
a2
a3
b0
b1
b2
b3
four_b_multi
out0
out1
out2
out3
out4
out5
out6
out7
four_b_multi
b3
b2
b1
b0
a3
a2
a1
a0
out7
out6
out5
out4
out3
out2
out1
out0
4x1_multi
4x1_multi
4x1_multi
4x1_multi
Shift left 1x
Shift left 2x
Shift left 3x
8-Bit Adder
8-Bit Adder
8-Bit Adder
8

In dem folgenden Beispiel wird der prinzipielle Ablauf unserer Multiplikationsberechnung beschrieben:

$$\begin{aligned}
\text{Faktor} &= 1011 \\
\text{Multiplikator} &= 1101 \\
\\
\text{1. Stelle d. Faktors} \cdot \text{Multiplikator} &= 1 \cdot 1101 \\
&= 1101 \\
\\
\text{2. Stelle d. Faktors} \cdot \text{Multiplikator} &= 1 \cdot 1101 \\
&= 1101 \\
[\ldots] \text{ mit Bitshift} &= 11010 \\
\\
\text{3. Stelle d. Faktors} \cdot \text{Multiplikator} &= 0 \cdot 1101 \\
&= 0 \\
[\ldots] \text{ mit Bitshift} &= 0 \\
\\
\text{4. Stelle d. Faktors} \cdot \text{Multiplikator} &= 1 \cdot 1101 \\
&= 1101 \\
[\ldots] \text{ mit Bitshift} &= 1101000
\end{aligned}$$

$$\begin{array}{r}
1011 \\
\times 1101 \\
\hline
1101 \\
+11010 \\
+000000 \\
+1101000 \\
\hline
{\scriptstyle 1} \\
+1111111 \\
\hline
10001111
\end{array}$$

Die Zwischenergebnisse dieser Rechnung werden jeweils in den Signalen `tmp0` bis `tmp6` gespeichert.

Wir nutzen einen 8-Bit-Addierer, den wir in einer zusätzlichen Datei namens `eight_bit_adder.vhd` gespeichert haben, um die Zwischenergebnisse zu einem Gesamtergebnis zu addieren.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity four_b_multi is
5   port (
```

```
6       a0 : in STD_LOGIC;
7       a1 : in STD_LOGIC;
8       a2 : in STD_LOGIC;
9       a3 : in STD_LOGIC;
10      b0 : in STD_LOGIC;
11      b1 : in STD_LOGIC;
12      b2 : in STD_LOGIC;
13      b3 : in STD_LOGIC;
14      out0 : out STD_LOGIC;
15      out1 : out STD_LOGIC;
16      out2 : out STD_LOGIC;
17      out3 : out STD_LOGIC;
18      out4 : out STD_LOGIC;
19      out5 : out STD_LOGIC;
20      out6 : out STD_LOGIC;
21      out7 : out STD_LOGIC
22      );
23  end four_b_multi;
24
25  architecture four_b_multi_impl of four_b_multi is
26    component eight_bit_adder
27      Port (
28        a0 : in  STD_LOGIC;
29        a1 : in  STD_LOGIC;
30        a2 : in  STD_LOGIC;
31        a3 : in  STD_LOGIC;
32        a4 : in  STD_LOGIC;
33        a5 : in  STD_LOGIC;
34        a6 : in  STD_LOGIC;
35        a7 : in  STD_LOGIC;
36        b0 : in  STD_LOGIC;
37        b1 : in  STD_LOGIC;
38        b2 : in  STD_LOGIC;
39        b3 : in  STD_LOGIC;
40        b4 : in  STD_LOGIC;
41        b5 : in  STD_LOGIC;
42        b6 : in  STD_LOGIC;
43        b7 : in  STD_LOGIC;
44        s0 : out  STD_LOGIC;
45        s1 : out  STD_LOGIC;
46        s2 : out  STD_LOGIC;
47        s3 : out  STD_LOGIC;
48        s4 : out  STD_LOGIC;
49        s5 : out  STD_LOGIC;
50        s6 : out  STD_LOGIC;
51        s7 : out  STD_LOGIC;
52        c_out : out  STD_LOGIC);
53    end component;
54
```

```
55
56    -- Zusammenfassen der einzelnen Eingangsleitungen für b(0-3) zu einem
          Bitvektor
57    -- Vector wurde verwendet um das Programm uebersichtlicher zu halten und den
58    -- Zugriff auf b0 - b3 zu vereinfachen
59    signal b: STD_LOGIC_VECTOR(3 downto 0);
60    -- Zur Zwischenspeicherung der carry bits
61    signal zero: STD_LOGIC_VECTOR(2 downto 0);
62
63    -- Hilfssignale um das die Ergebnisse der 1 bit multi. und des Shiften zu
          sichern
64    -- Es wurden acht bit verwendet um das Programm uebersichtlicher zu halten.
65    signal tmp0 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
66    signal tmp1 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
67    signal tmp2 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
68    signal tmp3 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
69
70    -- Temporaere Variablen für das aufsummieren der einzelnen Multiplikationen
71    -- Es wurden acht bit verwendet um das Programm uebersichtlicher zu halten.
72    signal tmp4 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
73    signal tmp5 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
74    signal tmp6 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
75
76  begin
77    -- Eingangsleitungen b0,b1,b2,b3 in den Eingangsbitvektor b schreiben.
78    b(0) <= b0;
79    b(1) <= b1;
80    b(2) <= b2;
81    b(3) <= b3;
82    --Zuweisungen für zerodefinitionen!
83    zero(0) <= '0';
84    zero(1) <= '0';
85    zero(2) <= '0';
86
87    multiply : process (b, a0, a1, a2, a3)
88    begin
89      if(a0 = '1') then
90        tmp0 <= '0' & '0' & '0' & '0' & b;
91      else
92        tmp0 <= "00000000";
93      end if;
94
95      if(a1 = '1') then
96        tmp1 <= '0' & '0' & '0' & b & '0';
97      else
98        tmp1 <= "00000000";
99      end if;
100
101     if(a2 = '1') then
102       tmp2 <= '0' & '0' & b & '0' & '0';
```

```
103     else
104       tmp2 <= "00000000";
105     end if;
106
107     if(a3 = '1') then
108       tmp3 <= '0' & b & '0' & '0' & '0';
109     else
110       tmp3 <= "00000000";
111     end if;
112   end process;
113
114   -- Addition der zuvor bestimmten Zwischenergebnisse (Ergebnis in tmp6)
115   -- Verwendung von 8-bit Addierern um das Shiften einfacher zu handhaben
116   eba0: eight_bit_adder PORT MAP(
117       tmp0(0),tmp0(1),tmp0(2),tmp0(3),tmp0(4),tmp0(5),tmp0(6),tmp0(7),
118       tmp1(0),tmp1(1),tmp1(2),tmp1(3),tmp1(4),tmp1(5),tmp1(6),tmp1(7),
119       tmp4(0),tmp4(1),tmp4(2),tmp4(3),tmp4(4),tmp4(5),tmp4(6),tmp4(7),
120       zero(0));
121   eba1: eight_bit_adder PORT MAP(
122       tmp2(0),tmp2(1),tmp2(2),tmp2(3),tmp2(4),tmp2(5),tmp2(6),tmp2(7),
123       tmp3(0),tmp3(1),tmp3(2),tmp3(3),tmp3(4),tmp3(5),tmp3(6),tmp3(7),
124       tmp5(0),tmp5(1),tmp5(2),tmp5(3),tmp5(4),tmp5(5),tmp5(6),tmp5(7),
125       zero(1));
126   eba2: eight_bit_adder PORT MAP(
127       tmp4(0),tmp4(1),tmp4(2),tmp4(3),tmp4(4),tmp4(5),tmp4(6),tmp4(7),
128       tmp5(0),tmp5(1),tmp5(2),tmp5(3),tmp5(4),tmp5(5),tmp5(6),tmp5(7),
129       tmp6(0),tmp6(1),tmp6(2),tmp6(3),tmp6(4),tmp6(5),tmp6(6),tmp6(7),
130       zero(2));
131
132   -- Das Ergebnis tmp6 wird an den Ausgang out(0-7) gelegt!
133   out0 <= tmp6(0);
134   out1 <= tmp6(1);
135   out2 <= tmp6(2);
136   out3 <= tmp6(3);
137   out4 <= tmp6(4);
138   out5 <= tmp6(5);
139   out6 <= tmp6(6);
140   out7 <= tmp6(7);
141
142
143 end four_b_multi_impl;
```

Für Aufgabe 3: Vier-Bit-Addierer (eight_bit_adder.vhd)

Den Volladdierer und die Testbench des Volladdierers haben wir aus den auf StudIP hochgeladenen Dateien entnommen.

Der Acht-Bit-Addierer gestaltet sich als vergleichsweise unkompliziert. Wir führen die

einzelnen Additionen mit acht Volladdierern aus, die das Carry-Bit (`tmp0` bis `tmp7`) des jeweils vorherigen Volladdierers nutzen, um die nächste Stelle des Ergebnisses zu berechnen. Jeder Volladdierer errechnet so das Ergebnis einer einzelnen Stelle.

Der Volladdierer ist in `full_adder.vhd` näher beschrieben.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity eight_bit_adder is
5    Port ( a0 : in  STD_LOGIC;
6      a1 : in  STD_LOGIC;
7      a2 : in  STD_LOGIC;
8      a3 : in  STD_LOGIC;
9      a4 : in  STD_LOGIC;
10     a5 : in  STD_LOGIC;
11     a6 : in  STD_LOGIC;
12     a7 : in  STD_LOGIC;
13     b0 : in  STD_LOGIC;
14     b1 : in  STD_LOGIC;
15     b2 : in  STD_LOGIC;
16     b3 : in  STD_LOGIC;
17     b4 : in  STD_LOGIC;
18     b5 : in  STD_LOGIC;
19     b6 : in  STD_LOGIC;
20     b7 : in  STD_LOGIC;
21     s0 : out  STD_LOGIC;
22     s1 : out  STD_LOGIC;
23     s2 : out  STD_LOGIC;
24     s3 : out  STD_LOGIC;
25     s4 : out  STD_LOGIC;
26     s5 : out  STD_LOGIC;
27     s6 : out  STD_LOGIC;
28     s7 : out  STD_LOGIC;
29     c_out : out  STD_LOGIC);
30 end eight_bit_adder;
31
32 -- Der 8-Bit Adder verwendet den im studip bereitgestellten full_adder
33
34 architecture Behavioral of eight_bit_adder is
35   component full_adder
36       Port ( a : in  STD_LOGIC;
37            b : in  STD_LOGIC;
38            c_in : in  STD_LOGIC;
39            s : out  STD_LOGIC;
40            c_out : out  STD_LOGIC);
41   end component;
42
43   signal zero : STD_LOGIC := '0';
44   signal tmp0,tmp1,tmp2,tmp3,tmp4,tmp5,tmp6 : STD_LOGIC;
```

```
45 begin
46
47   va0: full_adder PORT MAP(a => a0, b => b0, c_in => zero,s => s0,
48         c_out => tmp0);
49   va1: full_adder PORT MAP(a1,b1,tmp0,s1,tmp1);
50   va2: full_adder PORT MAP(a2,b2,tmp1,s2,tmp2);
51   va3: full_adder PORT MAP(a3,b3,tmp2,s3,tmp3);
52   va4: full_adder PORT MAP(a4,b4,tmp3,s4,tmp4);
53   va5: full_adder PORT MAP(a5,b5,tmp4,s5,tmp5);
54   va6: full_adder PORT MAP(a6,b6,tmp5,s6,tmp6);
55   va7: full_adder PORT MAP(a7,b7,tmp6,s7,c_out);
56
57 end Behavioral;
```

Für Aufgabe 3: Volladdierer (full_adder.vhd)

Der Volladdierer operiert auf folgende Weise:

`a` und `b` sind die zu addierenden Zahlen, `c_in` ist das Input-Carry, `c_out` das Output-Carry und `s` das Ergebnis der Addition.

$$s = a \veebar b \veebar c\_in$$
$$c = (a \vee b) \wedge (a \vee c\_in) \vee (b \wedge c\_in))$$

```
1  --Den Volladdierer und die Testbench des Volladdierers haben wir aus den auf
       StudIP hochgeladenen Dateien entnommen.
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity full_adder is
6      Port ( a : in  STD_LOGIC;
7             b : in  STD_LOGIC;
8             c_in : in  STD_LOGIC;
9             s : out  STD_LOGIC;
10            c_out : out  STD_LOGIC);
11 end full_adder;
12
13 architecture Behavioral of full\_adder is
14
15 begin
16         s <= a xor b xor c_in;
17         c_out <= (a and b) or (a and c_in) or (b and c_in);
18 end Behavioral;
```

Aufgabe 3 Geisterfigur (ghost.vhd)

Wir nutzen 2 Inputs, den Bewegungssensor `move` und den Clock, um das Ergebnis der Outputs zu bestimmen.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity ghost is
5    port (
6    -- Die Uhr, jede Sekunde gibt es eine steigende Flanke
7    clk : in STD_LOGIC;
8    -- Der Bewegunsmelder
9    move : in STD_LOGIC;
10   -- Die Lampen fuer die Augen
11   eyes : out STD_LOGIC;
12   -- Der Lautsprecher fuer das Gerassel der Ketten
13   chains : out STD_LOGIC);
14 end ghost;
```

Erneut nutzen wir Signale, damit die Werte zu Beginn der Simulation eindeutig sind. Dabei werden `speaker` für `chains` und `bulb` für `eyes` in jedem Clock-Cycle als Wert zugewiesen.

```
16 architecture ghost_impl of ghost is
17
18 -- Lokale Variablen, die abgefragt werden koennen. Die geschalteten Ausgaenge
       ergeben zusammen einen Zustand,
19 -- dieser Zustand kann anhand der folgenden Variablen abgefragt werden!
20 signal speaker : std_logic := '0';
21 signal bulb : std_logic := '0';
22 -- Die Counter, die nach aktivieren von Move und deaktivieren von Move anfangen
       zu zaehlen.
23 signal counterON : Integer := 0;
24 signal counterOFF : Integer := 0;
```

Nachdem der Bewegungssensor überprüft wurde, werden abhängig davon, ob Bewegung festgestellt wurde, verschiedene Signale hochgezählt, um die Tätigkeiten der Figur in korrekter Zeit und Reihenfolge ablaufen zu lassen. In Zeile 35-47 wird der Code bei Bewegung beschrieben und in Zeile 49-63 wird beschrieben, was passiert, wenn keine Bewegung festgestellt wurde.

Es wird jeweils `counterON` bzw. `counterOFF` genutzt, um den zeitlichen Ablauf zu definieren. So wird beispielsweise `bulb` auf 0 gesetzt, wenn `counterOff` 30 erreicht.

```
26 begin
27
28 Zustandsuebergangsschaltnetz: process(clk)
29 begin
30 -- Wir reagieren nur auf die steigende Taktflanke
```

```
31 if rising_edge(clk) then
32
33         if(move = '1') then -- Der Bewegungsmelder detektiert eine Bewegung!
34                 counterON <= counterON + 1;
35                 -- Wenn der Bewegungsmelder aktiviert ist und sowohl speaker
                       als auch bulb deaktiviert sind
36                 -- muss der Counter auf 1 gesetzt werden und der speaker soll
                       aktiviert werden.
37                 if(speaker = '0' AND bulb = '0') then
38                         counterON <= 1;
39                         speaker <= '1';
40                 end if;
41
42                 -- Sobald der speaker aktiviert ist soll es laut
                       aufgabenstellung 3 sekunden dauern, bis die bulb aktiviert
                       wird.
43                 if(speaker = '1' AND bulb = '0') then
44                         if(counterON = 3) then
45                                 bulb <= '1';
46                         end if;
47                 end if;
48         end if;
49
50         if(move = '0') then-- Es kann keine Bewegung mehr festgestellt werden!
51                 counterOFF <= counterOFF + 1;
52                 -- Wenn der speaker und die bulb noch angeschaltet sind, soll
                       nach 10 Sekunden der speaker augeschaltet werden.
53                 if(speaker = '1' AND bulb = '1') then
54                         if (counterOFF = 10) then
55                                 speaker <= '0';
56                         end if;
57                 end if;
58                 -- Wenn der Speaker bereits ausgeschaltet ist, dauert es
                       weitere 20 Sekunden, bis die bulb deaktiviert wird!
59                 if(speaker = '0' AND bulb = '1') then
60                         if(counterOFF = 30) then
61                                 bulb <= '0';
62                         end if;
63                 end if;
64                 -- Wenn keine Bewegung detektiert wird und beide Ausgaenge
                       deaktiviert sind, wird der counter dauerhaft resetet!
65                 if(speaker = '0' AND bulb = '0') then
66                         counterOFF <= 0;
67                 end if;
68         end if;
69 end if;
70
71 end process;
```

`speaker` und `bulb` müssen natürlich ihren Wert an die Outputs übergeben, die sie stellvertreten.

```
73 Ausgabeschaltnetz: process(bulb, speaker)
74 begin
75         chains <= speaker;
76         eyes <= bulb;
77 end process;
78
79 end ghost_impl;
```

Testbench: ghost_tb.vhd

Unsere Testbench überprüft, ob `ghost` beim üblichen Betriebsablauf die Outputs korrekt setzt.

Indem vor und nach dem Auslösen des Bewegungsmelders zu verschiedenen Zeitpunkten mir einer Assert-Abfrage überprüft wird, ob die Outputs zu genau diesem Clock-Cycle wie erwartet sind, können wir den gesamten Ablauf überprüfen.

Der erste und letzte Schritt unseres Tests überprüft, ob die Outputs 0 ausgeben. So können wir ein unerwartetes oder verspätetes Verhalten der Figur ausschließen.

Mit zahlreichen `report`-Befehlen bleibt der Tests gut verfolgbar, um mögliche Fehlerquellen zu entdecken.

```
1  --Das Testziel dieser Testbench ist ein Ablauf wie in der Aufgabenstellung
       beschrieben.
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4
5  ENTITY ghost_tb IS
6  END ghost_tb;
7
8  ARCHITECTURE behavior OF ghost_tb IS
9
10     -- Component Declaration for the Unit Under Test (UUT)
11
12     COMPONENT ghost
13     PORT(
14          clk : IN  std_logic;
15          move : IN  std_logic;
16          eyes : OUT  std_logic;
17          chains : OUT  std_logic
18         );
19     END COMPONENT;
20
21
22    --Inputs
23    signal clk : std_logic := '0';
24    signal move : std_logic := '0';
25
```

```
26          --Outputs
27     signal eyes : std_logic;
28     signal chains : std_logic;
29
30     -- Clock period definitions
31     constant clk_period : time := 1000 ms;
32
33  BEGIN
34
35          -- Instantiate the Unit Under Test (UUT)
36     uut: ghost PORT MAP (
37            clk => clk,
38            move => move,
39            eyes => eyes,
40            chains => chains
41          );
42
43     -- Clock process definitions
44     clk_process :process
45     begin
46          -- Die Taktflanke wird immer zu Beginn des Taktzyklus gegeben und nicht
                erst zur häflte des Taktzyklusses.
47          -- Damit lassen sich beim testen die Zeiten besser ablesen!
48                  clk <= '1';
49                  wait for clk_period/2;
50                  clk <= '0';
51                  wait for clk_period/2;
52
53     end process;
54
55     -- Stimulus process
56     stim_proc: process
57     begin
58                  report "Beginn der Testbench!";
59                  -- Testen wie sich die Schaltung nach dem einschalten erhält!
60                  report "Testschritt 1 nichtstun";
61        wait for clk_period*1;
62                  assert chains = '0' report "Kettengeraschel, soll aber nicht"
                        ↪ severity warning;
63                  assert eyes = '0' report "Augen Leuchten, sollen aber nicht"
                        ↪ severity warning;
64
65                  -- Testen, was passiert wenn eine Bewegung erkannt wird! Die
                        Ketten sollen aktiviert werden!
66                  report "Testschritt 2 aktivieren des Bewegungsmelders";
67                  move <= '1';
68                  wait for clk_period*1; -- Zeit, bis die Schaltung auf das
                        geänderte Move reagieren kann!
69                  assert chains = '1' report "Kein Kettengeraschel, soll aber"
                        ↪ severity warning;
```

```
70                assert eyes = '0' report "Augen␣Leuchten,␣sollen␣aber␣nicht"
                      ↪ severity warning;
71
72                -- Testen ob die Augen nach 3 Sekunden aktviert wurden!
73                report "Testschritt␣3␣Bewegungsmelder␣3␣Sekunden␣aktiv";
74                wait for clk_period*3;
75                assert chains = '1' report "Kein␣Kettengeraschel,␣soll␣aber"
                      ↪ severity warning;
76                assert eyes = '1' report "Augen␣Leuchten␣nicht,␣sollen␣aber"
                      ↪ severity warning;
77
78                -- Es wird getetet, ob es Änderungen gibt solange sich move
                      nicht ändert!
79                report "Testschritt␣4␣nichtstun";
80            wait for clk_period*10;
81                assert chains = '1' report "Kein␣Kettengeraschel,␣soll␣aber"
                      ↪ severity warning;
82                assert eyes = '1' report "Augen␣Leuchten␣nicht,␣sollen␣aber"
                      ↪ severity warning;
83
84                -- Der Bewegungsmelder detektiert keine Bewegung mehr, die
                      Ketten und die Augen sollen aber weiter aktiv bleiben
85                report "Testschritt␣5␣Bewegungsmelder␣nicht␣mehr␣aktiv";
86                move <= '0';
87            wait for clk_period*1; -- Zeit, bis die Schaltung auf das geänderte Move
                  reagieren kann!
88                assert chains = '1' report "Kein␣Kettengeraschel,␣soll␣aber"
                      ↪ severity warning;
89                assert eyes = '1' report "Augen␣Leuchten␣nicht,␣sollen␣aber"
                      ↪ severity warning;
90
91                -- Nach 10 Sekunden sollen die Ketten deaktiviert werden!
92                report "Testschritt␣6␣Bewegungsmelder␣10s␣nicht␣mehr␣aktiv";
93            wait for clk_period*10;
94                assert chains = '0' report "Kein␣Kettengeraschel,␣soll␣aber"
                      ↪ severity warning;
95                assert eyes = '1' report "Augen␣Leuchten␣nicht,␣sollen␣aber"
                      ↪ severity warning;
96
97                -- 29 Sekunden, nachdem keine Bewegungen mehr vorhanden sind,
                      sollen immer noch die Augen leuchten!
98                report "Testschritt␣7␣Bewegungsmelder␣29s␣nicht␣mehr␣aktiv";
99            wait for clk_period*19;
100               assert chains = '0' report "Kein␣Kettengeraschel,␣soll␣aber"
                      ↪ severity warning;
101               assert eyes = '0' report "Augen␣Leuchten␣nicht,␣sollen␣aber"
                      ↪ severity warning;
102
103               -- Nach 31 Sekunden sollen dann auch endlich die Augen nicht
                      mehr leuchten
104               report "Testschritt␣8␣Bewegungsmelder␣31s␣nicht␣mehr␣aktiv";
```

```
105       wait for clk_period*2;
106                 assert chains = '0' report "Kein␣Kettengeraschel,␣soll␣aber"
                        ↪ severity warning;
107                 assert eyes = '0' report "Augen␣Leuchten␣nicht,␣sollen␣aber"
                        ↪ severity warning;
108
109                 report "Testbench␣Beendet" severity failure;
110
111       wait;
112    end process;
113
114 END;
```