Programmierbare Logik
Laborbericht

Wintersemester 2018/2019

Gruppe:

Baga, Oleksandra (Matr.-Nr. 849852)
Dupke, Marvin (Matr.-Nr. 852916)

Dozent: Prof. Dr.-Ing. Peter Gregorius

10. Oktober 2018

Inhaltsverzeichnis

1 Vorbereitung und Wiederholung

1.1 Minimierung I - DMF und KMF

Gegeben ist die Wahrheitstabelle rechts.

a) Entwickeln Sie ein Blockschaltbild zur Umsetzung der Funktion. Zu verwenden ist reine Kombinatorik! Zur Hilfestellung nutzen Sie die gegebene Wahrheitstabelle und/oder das Karnaugh-Diagramm.

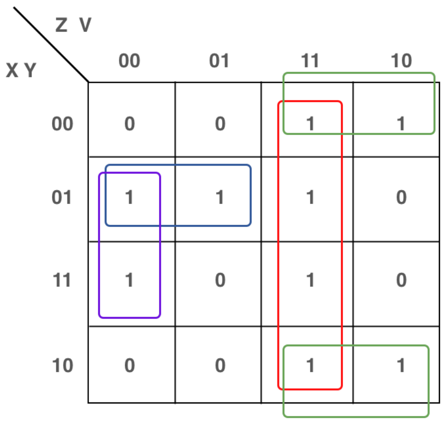
b) Geben Sie für den Ausgang f eine Boolesche Funktion an.

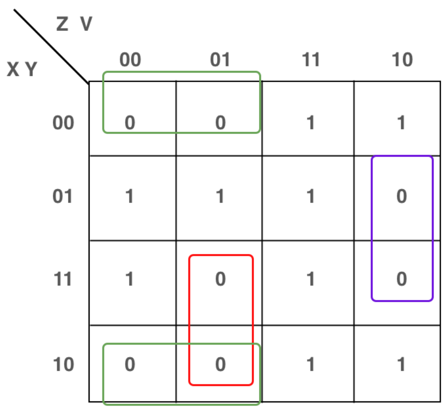
c) Entwickeln Sie in VHDL ein **Verhaltensmodell** zur Umsetzung der Funktion.

d) Entwickeln Sie eine Testumgebung für die Funktion in VHDL.

$(i)_{10}$	x	y	z	v	f
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

a:





b:

$$f_{dmf} = (y \wedge \neg z \wedge \neg v) \vee (\neg x \wedge y \wedge \neg z) \vee (z \wedge v) \vee (\neg y \wedge z) = \quad (1)$$

$$= ((y \wedge \neg z) \wedge (\neg v \vee \neg x)) \wedge (z \wedge (v \vee \neg y)) \quad (2)$$

$$f_{kmf} = (y \wedge z) \vee (\neg x \wedge z \wedge \neg v) \vee (\neg y \wedge \neg z \wedge v) \quad (3)$$

c:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY COMB_logic_4 IS
PORT( x:  IN std_logic;
                y:  IN std_logic;
                z:  IN std_logic;
                v:  IN std_logic;
                f:  OUT std_logic
        ) ;

END COMB_logic_4;

ARCHITECTURE behave OF COMB_logic_4 IS
BEGIN
                f <= (z AND NOT y) or
                          (z and v) or
                          (y and not x and not z) or
                          (not z and y and not v);

END behave;
```

d:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY CL4Testbench IS
END CL4Testbench;

ARCHITECTURE TBbehav OF CL4Testbench IS
  COMPONENT COMB_logic_4 IS
    PORT( x:  IN std_logic;
                y:  IN std_logic;
                z:  IN std_logic;
                v:  IN std_logic;
                f:  OUT std_logic
        ) ;
 END COMPONENT;

SIGNAL x_s, y_s, f_s, z_s, v_s: std_logic;

BEGIN
  CompToTest: COMB_logic_4 PORT MAP (x_s, y_s, z_s, v_s, f_s);
 v_s <=
                        '0' after 0ns,
                        '1' after 10ns,
                        '0' after 20ns,
                        '1' after 30ns,
                        '0' after 40ns,
                        '1' after 50ns,
                        '0' after 60ns,
                        '1' after 70ns,
                        '0' after 80ns,
                        '1' after 90ns,
                        '0' after 100ns,
                        '1' after 110ns,
                        '0' after 120ns,
                        '1' after 130ns,
                        '0' after 140ns,
                                    '1' after 150ns;
 z_s <=
                                    '0' after 0ns,
                                    '1' after 20ns,
                                    '0' after 40ns,
                                    '1' after 60ns,
                                    '0' after 80ns,
                                    '1' after 100ns,
                                    '0' after 120ns,
                                    '1' after 140ns;

 y_s <=
                                    '0' after 0ns,
                                    '1' after 40ns,
                                    '0' after 80ns,
                                    '1' after 120ns;

 x_s <=
                                    '0' after 0ns,
                                    '1' after 80ns;
END TBbehav;
```

2 Laboraufgaben: 1. Basiskomponenten

2.1 Einfaches Register

In Abbildung 4.1 ist ein einfaches 1-Bit-Register dargestellt. Die zusätzliche Kombinatorik erlaubt die Funktion des 1-Bit-Registers zu steuern. Der High-Aktive Ladeeingang LD gibt den Dateneingang D_0 frei. Mit der steigenden Flanke des Taktes CKL wird das Datenbit in das MS-D-FF übernommen. Der High-Aktive und asynchrone Rücksetzeingang erzeugt am Ausgang des Registers eine $Q_p = 0_b$.

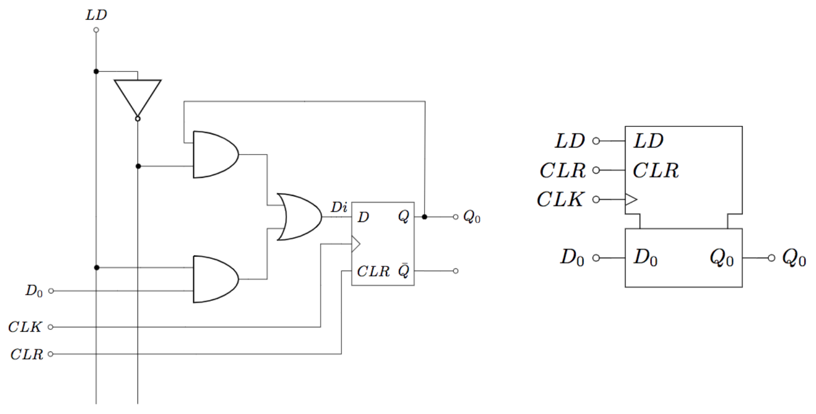


Abbildung 4.1.: Schaltbild (links) und Symbol (rechts) eines 1-Bit-Registers

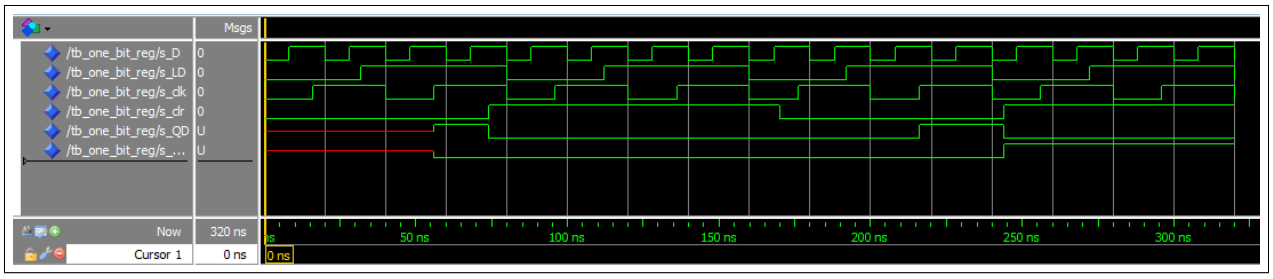
- ▶ Entwickeln Sie ein Verhaltensmodell in VHDL. Überprüfen Sie die Funktion mittels Simulation mit ModelSim.
- ▶ Entwickeln Sie auf Basis des Registers ein generische n-Bit Register.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY one_bit_reg IS
PORT (     D  : IN STD_LOGIC;
          LD  : IN STD_LOGIC;
         CLK  : IN STD_LOGIC;
          CLR  : IN STD_LOGIC;
         QD, nQD  : OUT STD_LOGIC
        );
END one_bit_reg;

-- D-FlipFlop reacts only on rising clock edge

ARCHITECTURE Behav OF one_bit_reg IS
BEGIN

-- Using rising_edge instead of clock'event to avoid unexpected trigger in
    std_logic
-- This function returns a value "TRUE" only when the present value is '1' and
     the last value is '0'.
-- If the past value is something like 'Z','U' etc. then it will return a "
    FALSE"

PROCESS (CLK, CLR, LD) -- Sensivity list
BEGIN
        IF (rising_edge(CLK) AND (LD = '1') AND (CLR = '0')) THEN
                QD <= D;
                nQD <= NOT D;
        ELSIF (CLR'event AND CLR = '1') THEN
                QD <= '0';
                nQD <= NOT D;
        END IF;
END PROCESS;
END Behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_one_bit_reg IS
END tb_one_bit_reg;

ARCHITECTURE testbench OF tb_one_bit_reg IS

-- D-FLIPFLOP
COMPONENT one_bit_reg
PORT (     D  : IN STD_LOGIC;
          LD  : In STD_LOGIC;
         CLK  : IN STD_LOGIC;
          CLR  : IN STD_LOGIC;
         QD, nQD  : OUT STD_LOGIC
);
END COMPONENT;

-- define input stimul signal
SIGNAL s_D : STD_LOGIC :='0';
SIGNAL s_LD : STD_LOGIC :='0';
SIGNAL s_clk : STD_LOGIC :='0';
SIGNAL s_clr : STD_LOGIC :='0';
SIGNAL s_QD, s_nQD : STD_LOGIC;

BEGIN

dut: ENTITY work.one_bit_reg
```

```
PORT MAP (
        D => s_D,
        LD => s_LD,
        CLK => s_clk,
        CLR => s_clr,
        QD => s_QD,
        nQD => s_nQD
);

-- common processes in the separate process
data_stimul: PROCESS
BEGIN
      s_D <= '0'; WAIT FOR 8 ns;
      s_D <= '1'; WAIT FOR 12 ns;
END PROCESS;

clock_stimul: PROCESS
BEGIN
      s_clk <= '0'; WAIT FOR 16 ns;
      s_clk <= '1'; WAIT FOR 24 ns;
END PROCESS;

enable_stimul: PROCESS
BEGIN
      s_LD <= '0'; WAIT FOR 32 ns;
      s_LD <= '1'; WAIT FOR 48 ns;
END PROCESS;

clear_stimul: PROCESS
BEGIN
      s_clr <= '0'; WAIT FOR 74 ns;
      s_clr <= '1'; WAIT FOR 96 ns;
END PROCESS;

END testbench;
```



2.2 PIPO, SISO, PISO, SIPO

Entwickeln Sie als Basiskomponenten folgende Register:

- Parallel-in / Parallel-out, PIPO
- Parallel-in / Serial-out, PISO
- Serial-in / Parallel-out, SIPO
- Serial-in / Serial-out, SISO

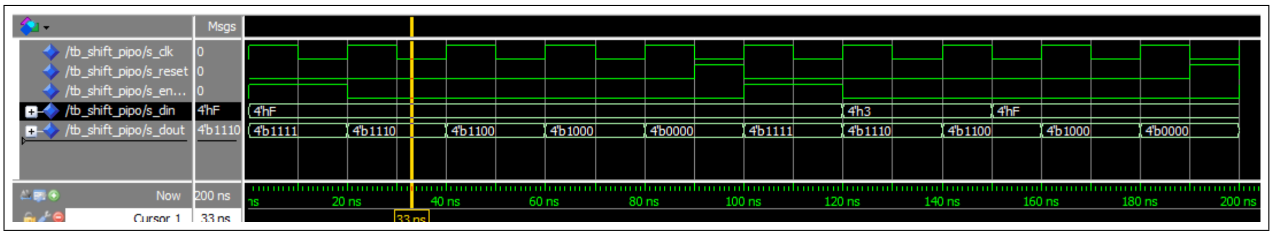
====================================

Parallel-in / Parallel-out, PIPO

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- For parallel in parallel out shift registers
-- all data bits appear on the parallel outputs immediately
--following the simultaneous entry of the data bits.

entity shift_pipo is
     port(
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : in STD_LOGIC;
          din : in STD_LOGIC_VECTOR(3 downto 0);
          dout : out STD_LOGIC_VECTOR(3 downto 0)
          );
end shift_pipo;

architecture Behav of shift_pipo is
begin
    pipo : process (clk, reset, enable) is
    begin
        if (reset = '1') then
            dout <= "0000";

        -- shifting and output all bits
        elsif (rising_edge(clk) AND enable = '0') then
          dout(3 downto 1) <= dout(2 downto 0);
          dout(0) <= '0';

        -- loading into internal register
        elsif (rising_edge(clk) and enable = '1') then
            dout <= din;
        end if;
    end process pipo;
end Behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_shift_pipo IS
END tb_shift_pipo;

ARCHITECTURE testbench OF tb_shift_pipo IS

COMPONENT shift_pipo is
     port(
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : in STD_LOGIC;
          din : in STD_LOGIC_VECTOR(3 downto 0);
          dout : out STD_LOGIC_VECTOR(3 downto 0)
          );
END COMPONENT;

-- define input stimul signal
SIGNAL s_clk : STD_LOGIC :='0';
SIGNAL s_reset : STD_LOGIC :='0';
SIGNAL s_enable : STD_LOGIC :='0';
SIGNAl s_din : STD_LOGIC_VECTOR(3 downto 0) := "0000";
SIGNAl s_dout : STD_LOGIC_VECTOR(3 downto 0) := "0000";
```

```
BEGIN

dut: ENTITY work.shift_pipo
PORT MAP (
      clk => s_clk,
      reset => s_reset,
      enable => s_enable,
      din => s_din,
      dout => s_dout
);

-- common processes in the separate process
data_stimul: PROCESS
BEGIN
     s_din <= "1111"; WAIT FOR 120 ns;
     s_din <= "0011"; WAIT FOR 30 ns;
END PROCESS;

clock_stimul: PROCESS
BEGIN
     s_clk <= '1'; WAIT FOR 10 ns;
     s_clk <= '0'; WAIT FOR 10 ns;
END PROCESS;

enable_stimul: PROCESS
BEGIN
     s_enable <= '1'; WAIT FOR 20 ns;
     s_enable <= '0'; WAIT FOR 80 ns;
END PROCESS;

reset_stimul: PROCESS
BEGIN
     s_reset <= '0'; WAIT FOR 90 ns;
     s_reset <= '1'; WAIT FOR 10 ns;
END PROCESS;

END testbench;
```



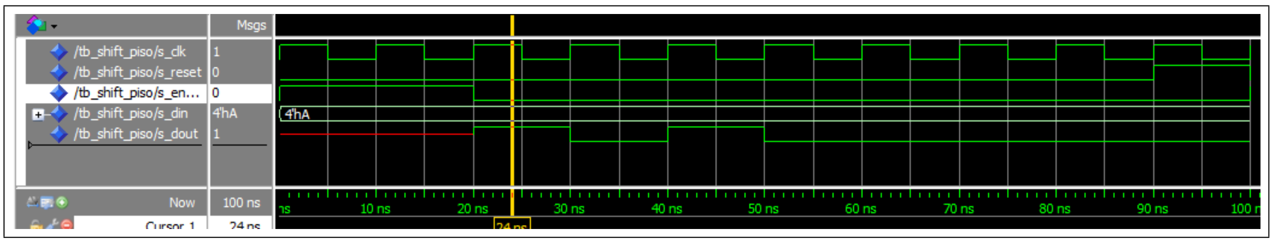
==================================

Parallel-in / Serial-out, PISO

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- For parallel in ? parallel out shift registers
-- all data bits appear on the parallel outputs immediately
--following the simultaneous entry of the data bits.

entity shift_piso is
      port(
            clk : in STD_LOGIC;
            reset : in STD_LOGIC;
```

```
          enable : in STD_LOGIC;
          din : in STD_LOGIC_VECTOR(3 downto 0);
          dout : out STD_LOGIC
          );
end shift_piso;

architecture Behav of shift_piso is

begin
    piso : process (clk, reset, enable) is
    variable s : std_logic_vector(3 downto 0) := "0000";
    begin
        if (reset = '1') then
            s := "0000";

        -- shifting and output the msb
        elsif (rising_edge(clk) and enable = '0') then
            dout <= s(3);
            s := s(2 downto 0) & '0';

        -- loading into internal register
        elsif (rising_edge(clk) and enable = '1') then
           s := din;
        end if;
    end process piso;
end Behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_shift_piso IS
END tb_shift_piso;

ARCHITECTURE testbench OF tb_shift_piso IS

COMPONENT shift_piso is
      port(
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : in STD_LOGIC;
          din : in STD_LOGIC_VECTOR(3 downto 0);
          dout : out STD_LOGIC
          );
END COMPONENT;

-- define input stimul signal
SIGNAL s_clk : STD_LOGIC :='0';
SIGNAL s_reset : STD_LOGIC :='0';
SIGNAL s_enable : STD_LOGIC :='0';
SIGNAl s_din : STD_LOGIC_VECTOR(3 downto 0) := "0000";
SIGNAl s_dout : STD_LOGIC := '0';

BEGIN

dut: ENTITY work.shift_piso
PORT MAP (
      clk => s_clk,
      reset => s_reset,
      enable => s_enable,
      din => s_din,
      dout => s_dout
);

-- common processes in the separate process
data_stimul: PROCESS
BEGIN
     s_din <= "1010"; WAIT FOR 200 ns;
END PROCESS;

clock_stimul: PROCESS
BEGIN
     s_clk <= '1'; WAIT FOR 5 ns;
     s_clk <= '0'; WAIT FOR 5 ns;
END PROCESS;

enable_stimul: PROCESS
BEGIN
     s_enable <= '1'; WAIT FOR 20 ns;
     s_enable <= '0'; WAIT FOR 80 ns;
END PROCESS;

reset_stimul: PROCESS
BEGIN
     s_reset <= '0'; WAIT FOR 90 ns;
     s_reset <= '1'; WAIT FOR 10 ns;
END PROCESS;

END testbench;
```



==================================

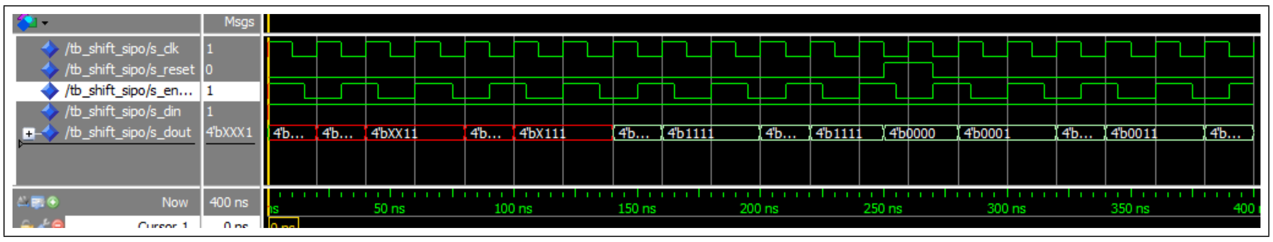
Serial-in / Parallel-out, SIPO

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- Serial-in to Parallel-out (SIPO) - the register is loaded with serial
   data,
-- one bit at a time, with the stored data being available at the output in
   parallel form.

entity shift_sipo is
     port(
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : in STD_LOGIC;
          din : in STD_LOGIC;
          dout : out STD_LOGIC_VECTOR(3 downto 0)
          );
end shift_sipo;

architecture Behav of shift_sipo is

begin
```

```
    sipo : process (clk, reset, enable) is
    begin
        if (reset = '1') then
            dout <= "0000";

        -- shifting and output
        elsif (rising_edge(clk) AND enable = '0') then
            dout(3 downto 1) <= dout(2 downto 0);
            dout(0) <= '0';

        -- load the single bit as lsb
         elsif (rising_edge(clk) AND enable = '1') then
            dout(0) <= din;
        end if;
    end process sipo;

end Behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_shift_sipo IS
END tb_shift_sipo;

ARCHITECTURE testbench OF tb_shift_sipo IS

COMPONENT shift_sipo is
      port(
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           enable : in STD_LOGIC;
           din : in STD_LOGIC;
           dout : out STD_LOGIC_VECTOR(3 downto 0)
           );
END COMPONENT;

-- define input stimul signal
SIGNAL s_clk : STD_LOGIC :='0';
SIGNAL s_reset : STD_LOGIC :='0';
SIGNAL s_enable : STD_LOGIC :='0';
SIGNAl s_din : STD_LOGIC :='0';
SIGNAl s_dout : STD_LOGIC_VECTOR(3 downto 0) := "0000";

BEGIN

dut: ENTITY work.shift_sipo
PORT MAP (
      clk => s_clk,
      reset => s_reset,
      enable => s_enable,
      din => s_din,
      dout => s_dout
);

-- common processes in the separate process
data_stimul: PROCESS
BEGIN
    s_din <= '1'; WAIT FOR 100 ns;
END PROCESS;

clock_stimul: PROCESS
```

```
BEGIN
     s_clk <= '1'; WAIT FOR 10 ns;
     s_clk <= '0'; WAIT FOR 10 ns;
END PROCESS;

enable_stimul: PROCESS
BEGIN
     s_enable <= '1'; WAIT FOR 15 ns;
     s_enable <= '0'; WAIT FOR 15 ns;
END PROCESS;

reset_stimul: PROCESS
BEGIN
     s_reset <= '0'; WAIT FOR 250 ns;
     s_reset <= '1'; WAIT FOR 20 ns;
END PROCESS;

END testbench;
```


Msgs
/tb_shift_sipo/s_clk 1
/tb_shift_sipo/s_reset 0
/tb_shift_sipo/s_en... 1
/tb_shift_sipo/s_din 1
/tb_shift_sipo/s_dout 4'bXXX1
4'b... 4'b... 4'bXX11 4'b... 4'bX111 4'b... 4'b1111 4'b... 4'b1111 4'b0000 4'b0001 4'b... 4'b0011 4'b...
Now 400 ns
Cursor 1 0 ns
ns 50 ns 100 ns 150 ns 200 ns 250 ns 300 ns 350 ns 400
0 ns

==================================

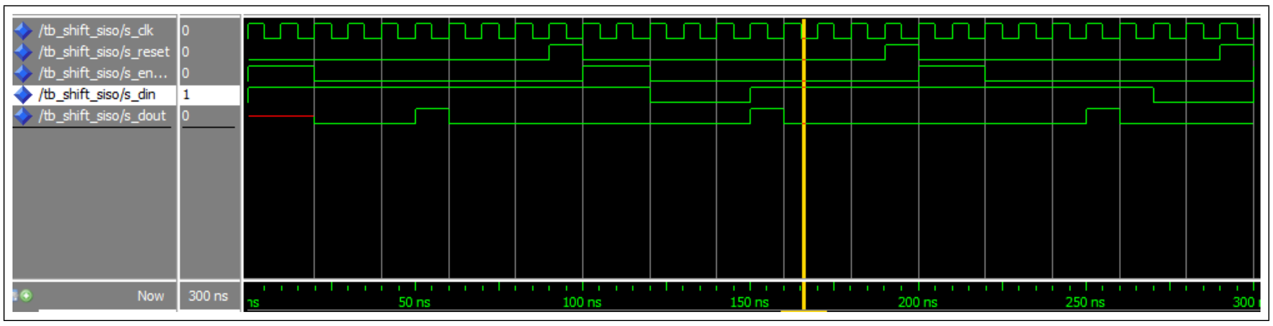
Serial-in / Serial-out, SISO

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

-- These are the simplest kind of shift registers.
-- The data string is presented at 'Data In', and is shifted right one
-- stage each time 'Data Advance' is brought high.
-- At each advance, the bit on the far left (i.e. 'Data In')
-- is shifted into the first flip-flop's output

entity shift_siso is
     port(
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : in STD_LOGIC;
          din : in STD_LOGIC;
          dout : out STD_LOGIC
          );
end shift_siso;

architecture Behav of shift_siso is

begin
     siso : process (clk, reset, enable) is

     variable s : std_logic_vector(3 downto 0) := "0000";

     begin
          if (reset = '1') then
               s := "0000";
```

```
        -- shift the bits of internal register and output the single bit
        elsif (rising_edge(clk) AND enable = '0') then
            dout <= s(3);
            s := s(2 downto 0) & '0';

        -- load the single bit as lsb in internal register
        elsif (rising_edge(clk) AND enable = '1') then
            s(0) := din;
        end if;
    end process siso;
end Behav;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_shift_siso IS
END tb_shift_siso;

ARCHITECTURE testbench OF tb_shift_siso IS

COMPONENT shift_siso is
     port(
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : in STD_LOGIC;
          din : in STD_LOGIC;
          dout : out STD_LOGIC
          );
END COMPONENT;

-- define input stimul signal
SIGNAL s_clk : STD_LOGIC :='0';
SIGNAL s_reset : STD_LOGIC :='0';
SIGNAL s_enable : STD_LOGIC :='0';
SIGNAl s_din : STD_LOGIC :='0';
SIGNAl s_dout : STD_LOGIC :='0';

BEGIN

dut: ENTITY work.shift_siso
PORT MAP (
     clk => s_clk,
     reset => s_reset,
     enable => s_enable,
     din => s_din,
     dout => s_dout
);

-- common processes in the separate process
data_stimul: PROCESS
BEGIN
    s_din <= '1'; WAIT FOR 120 ns;
    s_din <= '0'; WAIT FOR 30 ns;
END PROCESS;

clock_stimul: PROCESS
BEGIN
    s_clk <= '1'; WAIT FOR 5 ns;
    s_clk <= '0'; WAIT FOR 5 ns;
END PROCESS;
```

```
enable_stimul: PROCESS
BEGIN
    s_enable <= '1'; WAIT FOR 20 ns;
    s_enable <= '0'; WAIT FOR 80 ns;
END PROCESS;

reset_stimul: PROCESS
BEGIN
    s_reset <= '0'; WAIT FOR 90 ns;
    s_reset <= '1'; WAIT FOR 10 ns;
END PROCESS;

END testbench;
```



2.3 4-Bit Universalregister

Für viele Funktionen werden Universalregister benötigt. In der Tabelle unten sind die Anschlüsse für den Funktionsblock gelistet.

Pin	Beschreibung	Anmerkung
S_0, S_1	Mode Control Input	High active
P_0, P_3	Parallel Data Input	
SHR	Serial Shift Right Data Input	High active
SHL	Serial Shift Left Data Input	High active
CLK	Clock	
RST	Reset Signal	High active
$Q_0 - Q_3$	Parallel Output	

Die Funktion des 4-Bit Universalregisters ist mit der Wahrheitstabelle definiert.

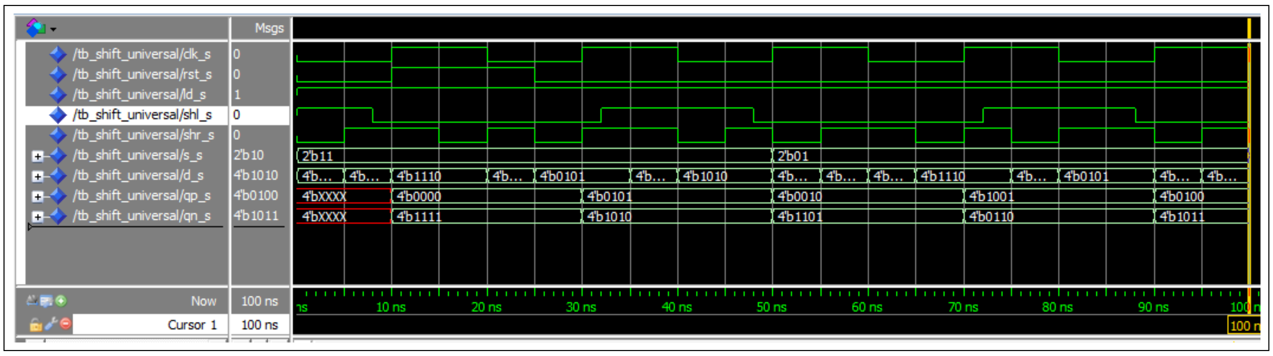
- Beschreiben Sie die Funktionen eines generischen n-Bit Universalregisters als Basiskomponente für das universelle Register. Entwickeln Sie ein Blockschaltbild.
- Geben Sie die Impulsdiagramme für alle Betriebsarten des Universalregisters an.
- Schreiben Sie ein VHDL-Code zur Umsetzung der Funktion. Entwickeln Sie eine Testumgebung und verifizieren Sie die Funktion mit ModelSim.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY shift_universal IS
GENERIC (N : integer := 4);
PORT(    CLK, RST, LD:  IN std_logic; -- Control Inputs
```

```vhdl
        SHL, SHR:  IN std_logic; -- Direction of serial data shift
        D:  IN std_logic_vector(N-1 downto 0); -- Parallel Data Input
        S:  IN std_logic_vector(1 downto 0 ); -- Mode Control Input
        Qp, Qn: OUT std_logic_vector(N-1 downto 0)); -- Parallel Output
END shift_universal;

ARCHITECTURE behav OF shift_universal IS

SIGNAL Qo :std_logic_vector(N-1 downto 0) := (others => 'U');

BEGIN
shift_universal: PROCESS (D, CLK, RST, LD, SHL, SHR, S) IS
BEGIN
    IF (RST='1') THEN --clear register
        Qo <= ( others => '0' );
    ELSIF ((LD='1') and (rising_edge(CLK) ) ) THEN --LD is active to set D on
        rising edge
                        case S is
                                when "10" => -- shift left in
                                        Qo(N-1 downto 1) <= Qo(N-2 downto 0);
                                        Qo(0) <= SHL;
                                when "01" => -- shift right in
                                        Qo(N-2 downto 0) <= Qo(N-1 downto 1);
                                        Qo(N-1) <= SHR;
                                when "11" => -- load parallel
                                        Qo <= D;
                                when "00" => -- hold
                                when others =>
                                        Qo <= Qo;
                        end case;
    ELSE
            Qo <= Qo;
    END IF;
END PROCESS shift_universal;

Qp <= Qo;
Qn <= NOT Qo;

END behav;
```

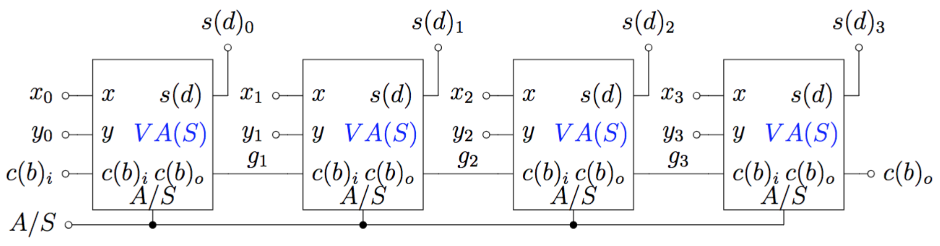
```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY tb_shift_universal IS
END tb_shift_universal;

ARCHITECTURE testbench OF tb_shift_universal IS

SIGNAL superN : integer := 4;
COMPONENT shift_universal IS
generic (N : integer := superN);
PORT(   CLK, RST, LD:  IN std_logic; -- Control Inputs
        SHL, SHR:  IN std_logic; -- Direction of serial data shift
        D:  IN std_logic_vector(N-1 downto 0); -- Parallel Data Input
        S:  IN std_logic_vector(1 downto 0 ); -- Mode Control Input
        Qp, Qn: OUT std_logic_vector(N-1 downto 0)); -- Parallel Output
END COMPONENT;

signal N: integer := superN;
SIGNAL clk_s, rst_s, ld_s : std_logic := 'U';
SIGNAL shl_s, shr_s: std_logic := 'U';

SIGNAL s_s : std_logic_vector(1 downto 0) := (others => 'U' );

SIGNAL d_s  : std_logic_vector(N-1 downto 0) := (others => 'U' );
SIGNAL qp_s : std_logic_vector(N-1 downto 0) := (others => 'U' );
SIGNAL qn_s : std_logic_vector(N-1 downto 0) := (others => 'U' );

BEGIN
dut: ENTITY work.shift_universal
PORT MAP (
          CLK => clk_s,
          RST => rst_s,
           LD => ld_s,
           SHL => shl_s,
           SHR => shr_s,
            D => d_s,
            S => s_s,
           Qp => qp_s,
           Qn => qn_s );

-- common processes in the separate process
reset_stimul: PROCESS
        BEGIN
        rst_s <= '0'; WAIT FOR 10 ns;
        rst_s <= '1'; WAIT FOR 15 ns;
        rst_s <= '0'; WAIT FOR 500 ns;
END PROCESS;

clock_stimul: PROCESS
        BEGIN
        clk_s <= '0'; WAIT FOR 10 ns;
        clk_s <= '1'; WAIT FOR 10 ns;
END PROCESS;

enable_stimul: PROCESS
        BEGIN
        ld_s <= '1'; WAIT FOR 10 ns;
END PROCESS;

left_stimul: PROCESS
        BEGIN
        shl_s <= '1'; WAIT FOR 8 ns;
        shl_s <= '0'; WAIT FOR 8 ns;
        shl_s <= '0'; WAIT FOR 8 ns;
        shl_s <= '0'; WAIT FOR 8 ns;
        shl_s <= '1'; WAIT FOR 8 ns;
END PROCESS;

right_stimul: PROCESS
        BEGIN
        shr_s <= '0'; WAIT FOR 5 ns;
        shr_s <= '1'; WAIT FOR 5 ns;
        shr_s <= '1'; WAIT FOR 5 ns;
        shr_s <= '0'; WAIT FOR 5 ns;
        shr_s <= '1'; WAIT FOR 5 ns;
END PROCESS;

mode_stimul: PROCESS
        BEGIN
```

```
            s_s <= "11"; WAIT FOR 50 ns; -- parallel in
            s_s <= "01"; WAIT FOR 50 ns; -- shift right
            s_s <= "10"; WAIT FOR 50 ns; -- shift left
            s_s <= "00"; WAIT FOR 50 ns; -- hold
END PROCESS;

 data_stimul: PROCESS
            BEGIN
            d_s <= "0101"; WAIT FOR 5 ns;
            d_s <= "1000"; WAIT FOR 5 ns;
            d_s <= "1110"; WAIT FOR 10 ns;
            d_s <= "0010"; WAIT FOR 5 ns;
            d_s <= "0101"; WAIT FOR 10 ns;
            d_s <= "1110"; WAIT FOR 5 ns;
            d_s <= "1010"; WAIT FOR 10 ns;
            d_s <= "0011"; WAIT FOR 5 ns;
END PROCESS;

END testbench;
```


Msgs
/tb_shift_universal/clk_s 0
/tb_shift_universal/rst_s 0
/tb_shift_universal/ld_s 1
/tb_shift_universal/shl_s 0
/tb_shift_universal/shr_s 0
/tb_shift_universal/s_s 2'b10
/tb_shift_universal/d_s 4'b1010
/tb_shift_universal/qp_s 4'b0100
/tb_shift_universal/qn_s 4'b1011
2'b11 2'b01
4'b1110 4'b0101 4'b1010 4'b1110 4'b0101
4'bXXXX 4'b0000 4'b0101 4'b0010 4'b1001 4'b0100
4'bXXXX 4'b1111 4'b1010 4'b1101 4'b0110 4'b1011
Now 100 ns
Cursor 1 100 ns
10 ns 20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns 100 ns

2.4 Arithmetik

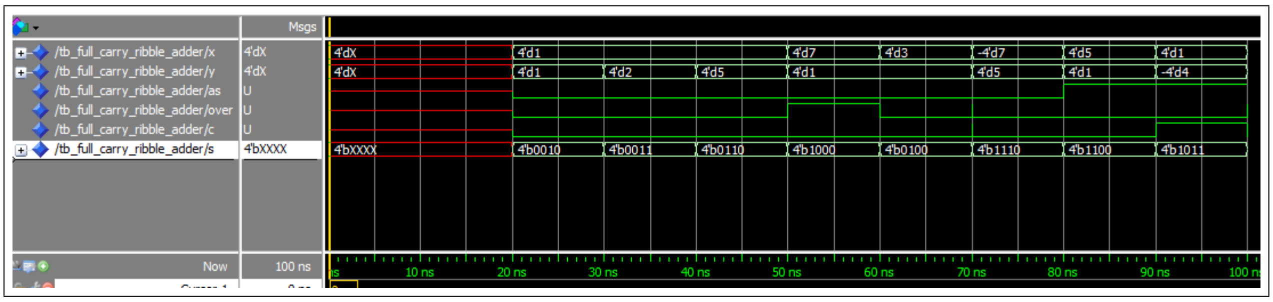
Abbildung 4.2 zeigt das Schaltbild und das Symbol eines umschaltbaren Volladdierer/Vollsubtrahierers.



```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY full_carry_ribble_adder IS
PORT(   x:  IN std_logic_vector(3 downto 0);
        y:  IN std_logic_vector(3 downto 0);
            as:  IN std_logic;
    over:  OUT std_logic;
            c:  OUT std_logic;
        s:  OUT std_logic_vector(3 downto 0 ));
END full_carry_ribble_adder;

ARCHITECTURE full_carry_ribble_adder_arc OF full_carry_ribble_adder IS

Component full_adder
PORT( x, y, ci:  IN std_logic;
          c, s:  OUT std_logic);
End Component;

signal carries : std_logic_vector(3 downto 0) := (others => 'U' );
signal temp : std_logic_vector(3 downto 0) := (others => 'U' );

BEGIN

temp(0) <= x(0) xor as;
temp(1) <= x(1) xor as;
temp(2) <= x(2) xor as;
temp(3) <= x(3) xor as;

--                            x:IN   ,y:IN,     ci:IN    ,   c:OUT    ,s:OUT
VA0 : full_adder PORT MAP (temp(0),y(0),      as     , carries(0),s(0) );
VA1 : full_adder PORT MAP (temp(1),y(1), carries(0), carries(1),s(1) );
VA2 : full_adder PORT MAP (temp(2),y(2), carries(1), carries(2),s(2) );
VA3 : full_adder PORT MAP (temp(3),y(3), carries(2), carries(3),s(3) );

over <= carries(3) xor carries(2);
c <= carries(3);

END full_carry_ribble_adder_arc;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_full_carry_ribble_adder Is
END tb_full_carry_ribble_adder;

ARCHITECTURE behavior OF tb_full_carry_ribble_adder Is

COMPONENT full_carry_ribble_adder
PORT(   x:  IN std_logic_vector(3 downto 0);
        y:  IN std_logic_vector(3 downto 0);
       as:  IN std_logic;
     over:  OUT std_logic;
        c:  OUT std_logic;
        s:  OUT std_logic_vector(3 downto 0));
END COMPONENT;

signal x : std_logic_vector(3 downto 0) := (others => 'U');
signal y : std_logic_vector(3 downto 0) := (others => 'U');

signal as: std_logic := 'U';
signal over: std_logic := 'U';
signal c : std_logic := 'U';

signal s : std_logic_vector(3 downto 0) := (others => 'U');

BEGIN

 dut: full_carry_ribble_adder PORT MAP (
 x => x,
 y => y,
 as => as,
 over => over,
 c => c,
 s => s
 );

-- stimulus process
stim_proc: process
BEGIN
wait for 20 ns;
-- add
x <= "0001";
y <= "0001";
as <= '0';
wait for 10 ns;

x <= "0001";
y <= "0010";
as <= '0';
wait for 10 ns;

x <= "0001";
y <= "0101";
as <= '0';
wait for 10 ns;

x <= "0111";
y <= "0001";
as <= '0';
wait for 10 ns;

x <= "0011";
y <= "0001";
as <= '0';
wait for 10 ns;

x <= "1001";
y <= "0101";
as <= '0';
wait for 10 ns;

-- sub

x <= "0101";
y <= "0001";
as <= '1';
wait for 10 ns;

x <= "0001";
y <= "1100";
as <= '1';
wait for 10 ns;

x <= "1100";
y <= "0010";
as <= '1';
wait for 10 ns;

x <= "0001";
y <= "0101";
```

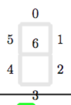
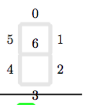
```
as <= '1';
wait for 10 ns;

x <= "0110";
y <= "1111";
as <= '1';
wait for 10 ns;

end process;

END;
```



```
                                Msgs
/tb_full_carry_ribble_adder/x    4'dX    4'dX   4'd1                          4'd7    4'd3    -4'd7   4'd5    4'd1
/tb_full_carry_ribble_adder/y    4'dX    4'dX   4'd1    4'd2    4'd5          4'd1            4'd5    4'd1    -4'd4
/tb_full_carry_ribble_adder/as   U
/tb_full_carry_ribble_adder/over U
/tb_full_carry_ribble_adder/c    U
/tb_full_carry_ribble_adder/s    4'bXXXX 4'bXXXX 4'b0010 4'b0011 4'b0110 4'b1000 4'b0100 4'b1110 4'b1100 4'b1011
Now                              100 ns
ns   10 ns   20 ns   30 ns   40 ns   50 ns   60 ns   70 ns   80 ns   90 ns   100 ns
```

2.5 Siebensegmentanzeige

Die Siebensegmentanzeige eignet sich auch zur Darstellung von Sedezimal-Code oderBCD-Codes. Zu entwerfen ist ein umschaltbarer Decoder gemäß der unten gegebenen Wahrheitstabelle

							HB=0	HB=1
	Eingänge							
$(i)_{10}$	LTN	BLN	b_3	b_2	b_1	b_0		
0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1	1
2	1	0	0	0	1	0	2	2
3	1	0	0	0	1	1	3	3
4	1	0	0	1	0	0	4	4
5	1	0	0	1	0	1	5	5
6	1	0	0	1	1	0	6	6
7	1	0	0	1	1	1	7	7
8	1	0	1	0	0	0	8	8
9	1	0	1	0	0	1	9	9
10	1	0	1	0	1	0	A	
11	1	0	1	0	1	1	b	
12	1	0	1	1	0	0	C	
13	1	0	1	1	0	1	d	
14	1	0	1	1	1	0	E	
15	1	0	1	1	1	1	F	
16	0	1	x	x	x	x	8	8
17	0	0	x	x	x	x		

```
LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.all;

ENTITY Segdis IS
PORT( B0, B1, B2, B3:  IN std_logic;
          A, B, C, D, E, F, G:  OUT std_logic);
END Segdis;

ARCHITECTURE Segdis_behavior OF Segdis IS
BEGIN
    A <= not ( B0 OR B2 OR (B1 AND B3) OR (NOT B1 AND NOT B3) );
    B <= not ( (NOT B1) OR (NOT B2 AND NOT B3) OR (B2 AND B3) );
    C <= not ( B1 OR NOT B2 OR B3 );
    D <= not ( (NOT B1 AND NOT B3) OR (B2 AND NOT B3) OR (B1 AND NOT B2 AND B3)
          OR (NOT B1 AND B2) OR B0 );
    E <= not ( (NOT B1 AND NOT B3) OR (B2 AND NOT B3) );
    F <= not ( B0 OR (NOT B2 AND NOT B3) OR (B1 AND NOT B2) OR (B1  AND NOT B3)
         );
    G <= not ( B0 OR (B1 AND NOT B2) OR (NOT B1 AND B2) OR (B2  AND NOT B3));
END Segdis_behavior;
```