



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN  
University of Applied Sciences

## Programmierbare Logik Laborbericht

Wintersemester 2018/2019

Gruppe:

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

Dozent: Prof. Dr.-Ing. Peter Gregorius

5. Dezember 2018

# Inhaltsverzeichnis

<b>1 Laboraufgaben: 1. Basiskomponenten</b>	<b>2</b>
1.1 Einfaches Register . . . . .	2
1.2 PIPO, SISO, PISO, SIPO . . . . .	6
1.3 4-Bit Universalregister . . . . .	14
1.4 Arithmetik . . . . .	18
1.5 Siebensegmentanzeige . . . . .	21
<b>2 Laboraufgaben: 2. Erkennung einer Bit-Sequenz</b>	<b>26</b>
2.1 Aufgabenstellungen A . . . . .	26
2.2 Aufgabenstellungen B . . . . .	28
2.3 Aufgabenstellungen C . . . . .	31
<b>3 Laboraufgaben: 3. Takte, Zähler und Zeitgeber</b>	<b>32</b>
3.1 BCD-Zähler und Codierung . . . . .	32
3.2 2-Digit BCD-Arithmetik . . . . .	38
3.3 3-Digit BCD-Zähler und 24h-Uhr . . . . .	38
<b>4 Booth-Algorithmus mit Datenpfad und Steuerwerk</b>	<b>39</b>
4.1 Aufgabenstellungen A . . . . .	40
4.1.1 Arithmetische Einheit . . . . .	41
4.1.2 Moore-Automaten . . . . .	42
4.1.3 VHDL Beschreibung . . . . .	42

# 1 Laboraufgaben: 1. Basiskomponenten

## 1.1 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  $Q_p = 0_b$ .

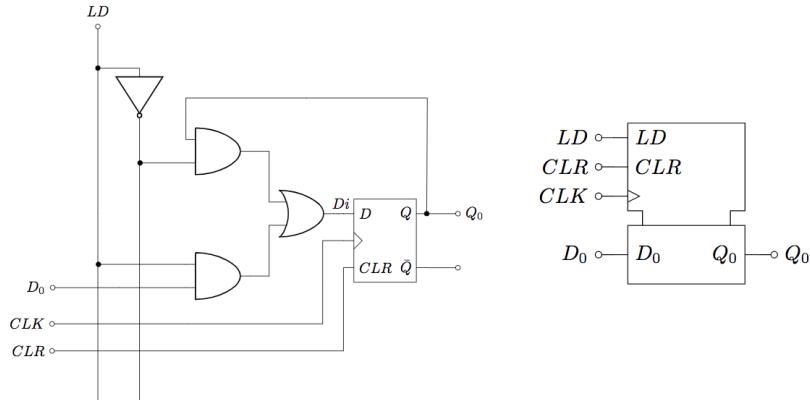


Abbildung 4.1.: Schaltbild (links) und Symbol (rechts) eines 1-Bit-Registers

- ▶ 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
  (
    ld , clr , clk      : in std_logic;
    d_0                  : in std_logic;
    q_0                  : out std_logic
  );
end one_bit_reg;

architecture behave of one_bit_reg is
  signal q_intern , q_out : std_logic := 'U';          -- intern
  signal for master slave
begin
  process ( clk , clr )
  begin
    if ( clr = '1' ) then                                -- clr ,
      async = most priority
      q_intern <= '0';
      q_out    <= '0';
    else
      if ( ld = '1' ) then
        if ( rising_edge( clk ) ) then
          q_intern <= d_0;
        elsif ( falling_edge( clk ) ) then
          q_intern <= '0';
        end if;
      end if;
    end if;
    q_out <= q_intern;
  end process;
end architecture;
```

```

        q_out      <= q_intern;
    end if;
end if;
end process;

q_0  <= q_out;
end behave;

```

Listing 1: One bit register VHDL Code

```

library ieee;
use ieee.std_logic_1164.all;

entity tb_one_bit_reg is
end tb_one_bit_reg;

-----
architecture test of tb_one_bit_reg is

component one_bit_reg

port
(
    ld , clr , clk   : in std_logic;
    d_0                 : in std_logic;
    q_0                 : out std_logic
);
end component;

signal ld , clr , clk , d_0   : std_logic;
signal q_0                  : std_logic;

begin

dut: one_bit_reg
port map
(
    ld , clr , clk , d_0, q_0
);

clk_signal : process
begin
    clk  <= '0';
    wait for 5 ns;
    clk  <= '1';
    wait for 5 ns;
end process;

clr  <= '0' after  0 ns , '1' after  2 ns ,
      '0' after 15 ns;
ld   <= '1' after  0 ns , '0' after 10 ns ,
      '1' after 35 ns , '0' after 140 ns;
d_0  <= '0' after  0 ns , '1' after  7 ns ,
      '0' after 17 ns , '1' after 26 ns ,
      '0' after 40 ns , '1' after 52 ns ,
      '0' after 59 ns , '1' after 66 ns ,
      '0' after 78 ns , '1' after 86 ns ,
      '0' after 100 ns , '1' after 110 ns ,
      '0' after 120 ns , '1' after 130 ns ,

```

```

'0' after 140 ns, '1' after 150 ns,
'0' after 160 ns, '1' after 170 ns;

end test;

```

Listing 2: One-Bit Register Testbench

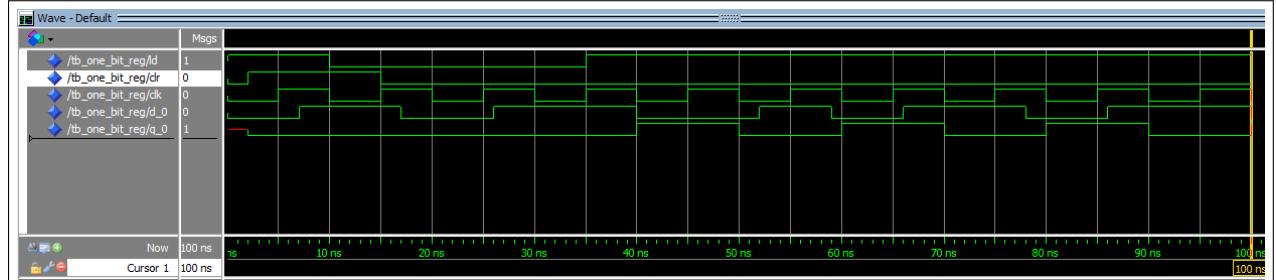


Abbildung 1: One-Bit Register Modelsim Simulation

```

library ieee;
use ieee.std_logic_1164.all;

entity n_bit_reg is
    generic
    (
        bits : natural := 8
            natural, subtype of integer, 0 to x
    );
    port
    (
        ld , clr , clk      : in std_logic;
        d               : in std_logic_vector(bits-1 downto 0);
        q               : out std_logic_vector(bits-1 downto 0)
    );
end n_bit_reg;

architecture behave of n_bit_reg is
    signal q_intern , q_out : std_logic_vector(bits-1 downto 0) := (others => 'U'); — intern signal for master slave
begin
    process ( clk , clr )
    begin
        if ( clr = '1' ) then
            async = most priority
            q_intern <= (others => '0');
            q_out     <= (others => '0');
        else
            if ( ld = '1' ) then
                if ( rising_edge( clk ) ) then
                    q_intern     <= d;
                elsif ( falling_edge( clk ) ) then
                    q_out       <= q_intern;
                end if;
            end if;
        end if;
    end process;

    q      <= q_out;
end behave;

```

Listing 3: Generisches n-Bit Register

```

library ieee;
use ieee.std_logic_1164.all;

entity tb_n_bit_reg is
end tb_n_bit_reg;

architecture test of tb_n_bit_reg is
component n_bit_reg
port
(
    ld, clr, clk      : in std_logic;
    d                 : in std_logic_vector(8-1 downto 0);
    q                 : out std_logic_vector(8-1 downto 0)
);
end component;

signal ld, clr, clk  : std_logic;
signal d             : std_logic_vector(8-1 downto 0);
signal q             : std_logic_vector(8-1 downto 0);

begin

dut: n_bit_reg
port map
(
    ld, clr, clk, d, q
);

clk_signal : process
begin
    clk  <= '0';
    wait for 5 ns;
    clk  <= '1';
    wait for 5 ns;
end process;

clr  <= '0' after  0 ns, '1' after  2 ns,
      '0' after 15 ns;
ld   <= '1' after  0 ns, '0' after 10 ns,
      '1' after 35 ns, '0' after 140 ns;
d    <= "00000000" after  0 ns, "00000001" after  7 ns,
      "00000010" after 17 ns, "00000011" after 26 ns,
      "00000100" after 40 ns, "00000101" after 52 ns,
      "00000110" after 59 ns, "00000111" after 66 ns,
      "00001000" after 78 ns, "00001001" after 86 ns,
      "00001010" after 100 ns, "00001011" after 110 ns,
      "11110000" after 120 ns, "11110001" after 130 ns,
      "11110010" after 140 ns, "11110011" after 150 ns,
      "11110100" after 160 ns, "11110101" after 170 ns;

end test;

```

Listing 4: Generisches n-Bit Register Testbench

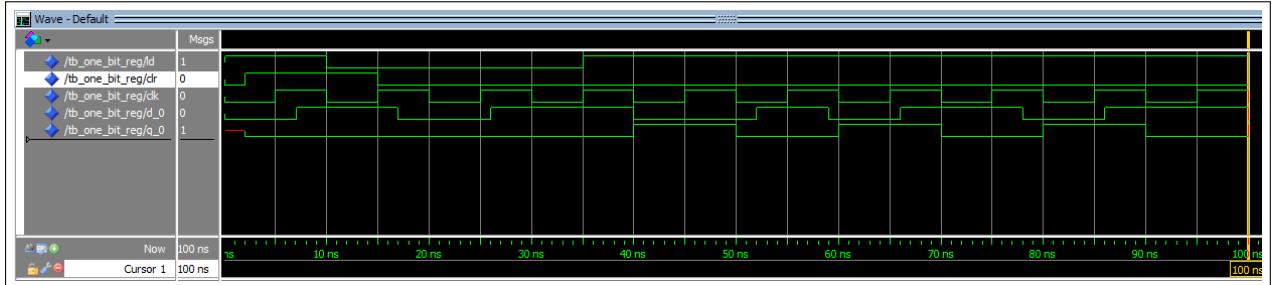


Abbildung 2: One bit Register Modelsim Simulation

## 1.2 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

signal s_qi : STD_LOGIC_VECTOR(3 downto 0) := "UUUU";
signal s_qo : STD_LOGIC_VECTOR(3 downto 0) := "UUUU";

begin
    pipo : process (clk, reset, enable) is
    begin
        if (reset = '1') then
            s_qi <= "0000";
            s_qo <= "0000";

        -- loading into internal register
        elsif (rising_edge(clk) AND enable = '1') then
            s_qi <= din;

        -- loading into internal register
        elsif (falling_edge(clk) and enable = '1') then
            s_qo <= s_qi;
    end if;
end process;
end;
```

```

        end if;
    end process pipo;
    dout <= s_qo;
end Behav;

```

Listing 5: PIPO Register VHDL Code

```

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 <= '0'; WAIT FOR 20 ns;
    s_enable <= '1'; WAIT FOR 180 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 ;

```

Listing 6: PIPO Register Testbench

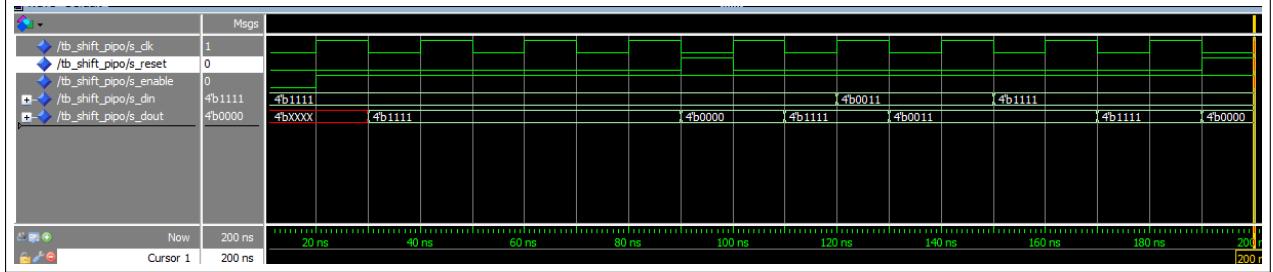


Abbildung 3: PIPO Register Modelsim Simulation

---

### Parallel-in / Serial-out, PISO

```

library ieee;
use ieee.std_logic_1164.all;

entity piso is
    port
    (
        clk, clr : in std_logic;
        d         : in std_logic_vector( 3 downto 0 );
        q_out     : out std_logic
    );
end piso;

architecture behave of piso is
    type state is (S0, S1, S2, S3);

    signal act_state, next_state : state;
    signal q_all                : std_logic_vector( 3 downto 0 );
    signal ld                   : std_logic;
begin
    process ( clk, clr ) -- Zustandsaktualisierung
    begin
        if ( clr = '1' ) then
            act_state <= S0;
            ld      <= '1';
        elsif ( falling_edge( clk ) ) then
            case next_state is
                when S0          => ld <= '1';
                when S1 | S2 | S3 => ld <= '0';
                when others       => ld <= '0';
            end case;
            act_state <= next_state;
        end if;
    end process;           -- END Zustandsaktualisierung

    process ( clr, act_state ) -- Uebergang

```

```

begin
    if ( clr = '1' ) then
        next_state <= S0;
    elsif ( clr = '0' ) then
        case act_state is
            when S0      => next_state <= S1;
            when S1      => next_state <= S2;
            when S2      => next_state <= S3;
            when S3      => next_state <= S0;
            when others => next_state <= S0;
        end case;
    end if;
end process;                                — END Uebergang

process ( act_state , ld , clr )           — Ausgang
begin
    if ( clr = '1' ) then
        q_all <= (others => '0');
    elsif ( clr = '0' ) then
        case act_state is
            when S0 =>
                if ( ld = '1' ) then
                    q_all <= q_all(2 downto 0) & '0';
                end if;
            when S1 =>
                if ( ld = '0' ) then
                    q_all <= d;
                end if;
            when S2 | S3 =>
                if ( ld = '0' ) then
                    q_all <= q_all(2 downto 0) & '0';
                end if;
        end case;
    end if;
end process;                                — END Ausgang

q_out <= q_all(3);
end behave;

```

Listing 7: PISO Register VHDL Code

```

library ieee;
use ieee.std_logic_1164.all;

entity tb_piso is
end tb_piso;



---


architecture test of tb_piso is

component piso
port
(
    clk , clr      : in std_logic;
    d              : in std_logic_vector( 3 downto 0 );
    q_out         : out std_logic
);
end component;

signal clk , clr , q_out      : std_logic;

```

```

signal d : std_logic_vector( 3 downto 0 );

begin

dut: piso
port map
(
    clk , clr , d, q_out
);

process --- clk signal
begin
    clk <= '0'; wait for 5 ns;
    clk <= '1'; wait for 5 ns;
end process;

clr <= '1' after 1 ns, '0' after 3 ns;
d <= "1010" after 0 ns, "0011" after 40 ns;

end test;

```

Listing 8: PISO Register Testbench

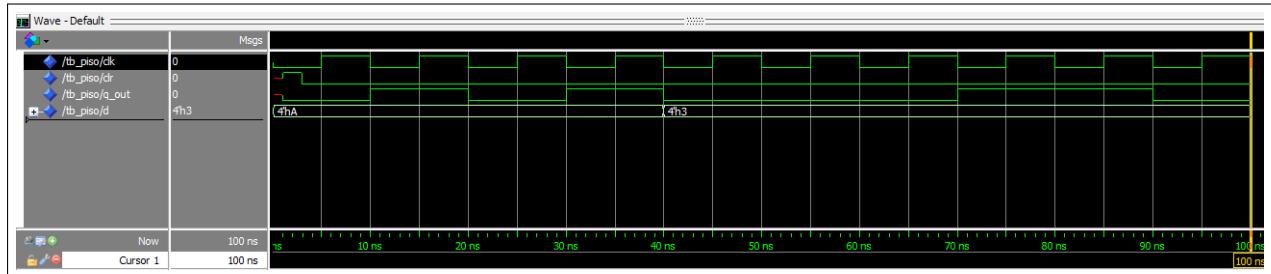


Abbildung 4: PISO Register Modelsim Simulation

```

=====
Serial-in / Parallel-out, SIPO
=====

library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE ieee.numeric_std.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;
        din : in STD_LOGIC;
        dout : out STD_LOGIC_VECTOR(3 downto 0)
    );
end shift_sipo;

architecture Behav of shift_sipo is

signal s_qo : STD_LOGIC_VECTOR(3 downto 0) := "UUUU";
signal s_q3, s_q2, s_q1, s_q0 : STD_LOGIC := 'U';

```

```

begin
    sipo : process (clk , reset) is
        variable counter : unsigned ( 1 DOWNTO 0 ) := "00";
        variable rd : std_logic := '0';
    begin
        if (reset = '1') then
            s_qo <= "0000";
            s_qo(3) <= '0';
            s_qo(2) <= '0';
            s_qo(1) <= '0';
            s_qo(0) <= '0';
            rd := '0';

        elsif (falling_edge(clk) AND rd = '0' AND reset = '0') then
            s_q0 <= din;
            s_q1 <= s_q0;
            s_q2 <= s_q1;
            s_q3 <= s_q2;

            if (counter = "11") then
                rd := '1';
            end if;

            counter := counter + 1;

        elsif (falling_edge(clk) AND rd = '1' AND reset = '0') then
            -- cleaning for a new cycle
            if (counter = "00") then
                rd := '0';
            end if;

            -- loading
            s_q0 <= din;
            s_q1 <= s_q0;
            s_q2 <= s_q1;
            s_q3 <= s_q2;

            -- output
            s_qo(3) <= s_q3;
            s_qo(2) <= s_q2;
            s_qo(1) <= s_q1;
            s_qo(0) <= s_q0;

            counter := counter + 1;
        end if;

    end process sipo;
    -- final output
    dout <= s_qo;
end Behav;

```

Listing 9: SIPO Register VHDL Code

```

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;
    din : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
  );
END COMPONENT;

-- define input stimuli signal
SIGNAL s_clk : STD_LOGIC := '0';
SIGNAL s_reset : 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,
  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 5 ns;
  s_clk <= '0'; WAIT FOR 5 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;

```

Listing 10: SIPO Register Testbench

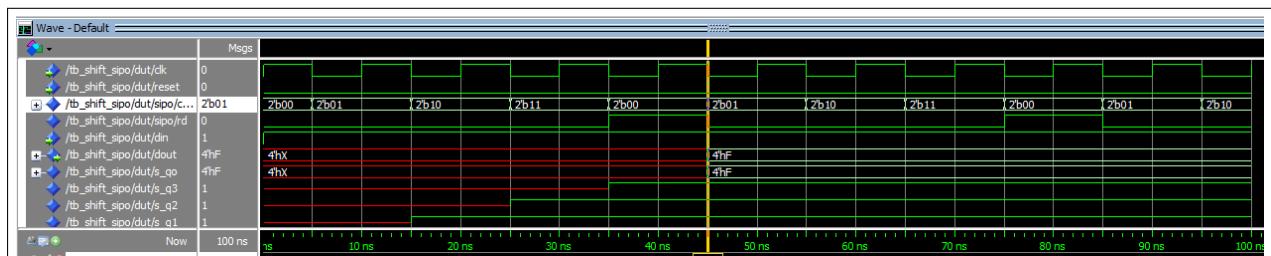


Abbildung 5: SIPO Register Modelsim Simulation

=====

**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

signal s_qi : STD_LOGIC_VECTOR(3 downto 0) := "UUUU";
signal s_qo : STD_LOGIC_VECTOR(3 downto 0) := "UUUU";

begin
    siso : process (clk, reset, enable) is
    begin
        if (reset = '1') then
            s_qi <= "0000";
            s_qo <= "0000";

            -- shift the bits of internal register and load the single bit
        elsif (rising_edge(clk) AND enable = '1') then
            s_qi(3 downto 1) <= s_qi(2 downto 0);
            s_qi(0) <= din;

            -- output from the internal register
        elsif (falling_edge(clk) AND enable = '1') then
            s_qo <= s_qi;
        end if;
    end process siso;
    -- final output
    dout <= s_qo(3);
end Behav;

```

Listing 11: SISO Register VHDL Code

```

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 25 ns;
    s_din <= '0'; WAIT FOR 25 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 <= '0'; WAIT FOR 20 ns;
    s_enable <= '1'; WAIT FOR 180 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;

```

Listing 12: SISO Register Testbench

### 1.3 4-Bit Universalregister

```

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);

```

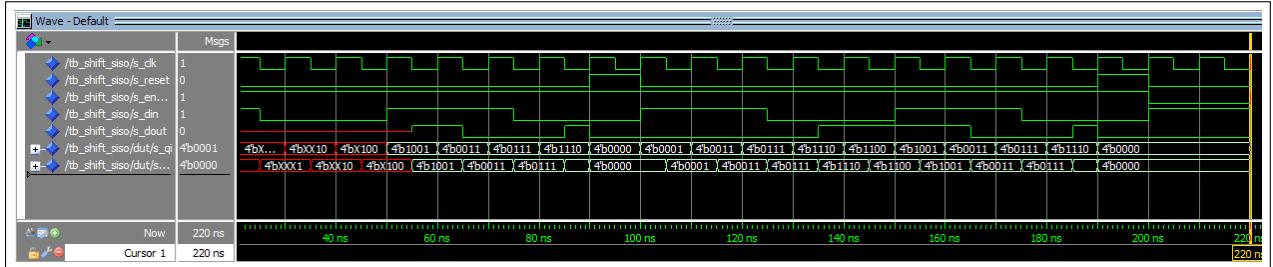


Abbildung 6: SISO Register Modelsim Simulation

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 Bauteilkomponente 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.

```

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 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;
    end case;
  END IF;
END process;

```

```

        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;

```

Listing 13: 4-Bit Universalregister VHDL Code

```

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;

```

Listing 14: 4-Bit Universalregister Testbench

## 1.4 Arithmetik

```

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

ENTITY full_carry_ripple_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_ripple_adder;

ARCHITECTURE full_carry_ripple_adder_arc OF full_carry_ripple_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;

```

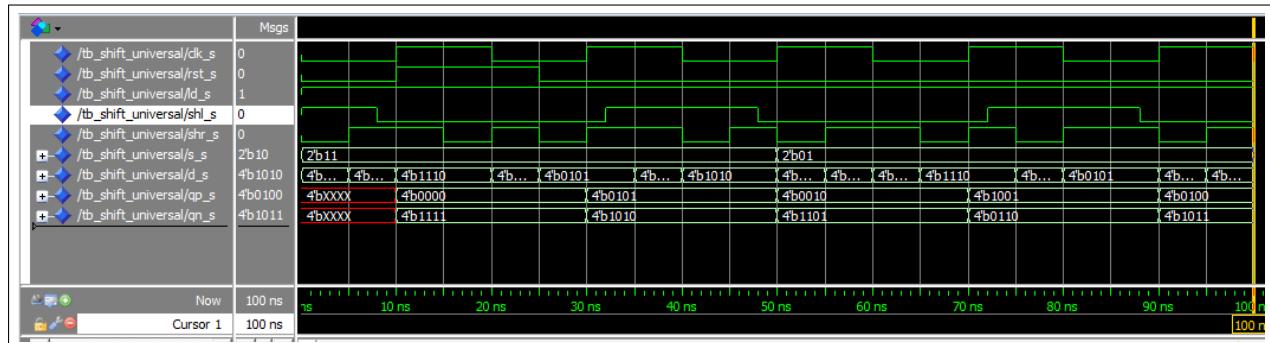
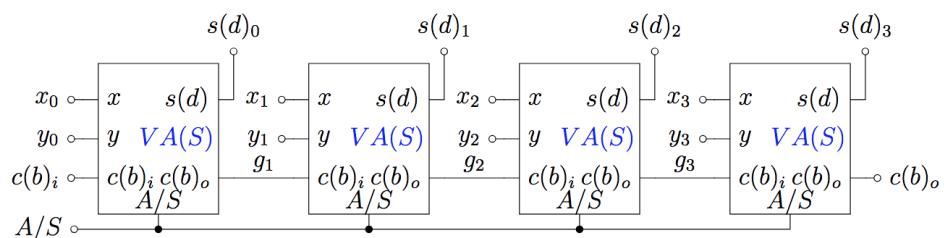


Abbildung 7: 4-Bit Register Modelsim Simulation

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



```

--                                     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;

```

Listing 15: Full Adder VHDL code

```

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;

```

Listing 16: Full Adder Testbench



Abbildung 8: Full adder Modelsim Simulation

## 1.5 Siebensegmentanzeige

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

$(i)_{10}$	Eingänge						HB=0	HB=1
	LTN	BLN	$b_3$	$b_2$	$b_1$	$b_0$	0	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	A
11	1	0	1	0	1	1	B	B
12	1	0	1	1	0	0	C	C
13	1	0	1	1	0	1	D	D
14	1	0	1	1	1	0	E	E
15	1	0	1	1	1	1	F	F
16	0	1	x	x	x	x	8	8
17	0	0	x	x	x	x	8	8

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity seven_seg is
  port
  (
    ltn           : in std_logic;                                --- enable
    decoder       : in std_logic;
    bln           : in std_logic;                                ---
    segment_test : in std_logic;
    hb            : in std_logic;                                --- if 0,
  );
end entity;
  
```

```

hexdec., if 1 displays 0 to 9
b_in : in std_logic_vector(3 downto 0); --- input
number
a,b,c,d,e,f,g : out std_logic --- output
for segments
);
end seven_seg;

architecture behave of seven_seg is
begin
process( ltn , bln , hb , b_in )
begin
if ( (ltn = '0') and (bln = '0') ) then --- not active
a <= '1'; b <= '1'; c <= '1'; d <= '1';
e <= '1'; f <= '1'; g <= '1';
elsif ( (ltn = '0') and (bln = '1') ) then --- segment test
a <= '0'; b <= '0'; c <= '0'; d <= '0';
e <= '0'; f <= '0'; g <= '0';
elsif ( (ltn = '1') and (bln = '0') ) then --- normal display
if ( (to_integer(unsigned(b_in)) < 10) or (hb = '0') ) then
--- normal when <10, or all hex
a <= ( b_in(0) and not b_in(1) and not b_in(2) and not b_in
(3))
or ( not b_in(0) and not b_in(1) and b_in(2) and not b_in
(3))
or ( b_in(0) and b_in(1) and not b_in(2) and b_in
(3))
or ( b_in(0) and not b_in(1) and b_in(2) and b_in
(3));
b <= ( b_in(0) and not b_in(1) and b_in(2) and not b_in
(3))
or ( not b_in(0) and b_in(1) and b_in(2)
)
or ( b_in(0) and b_in(1)
and b_in
(3))
or ( not b_in(0)
and b_in(2)
and b_in
(3));
c <= ( not b_in(0) and b_in(1) and not b_in(2) and not b_in
(3))
or (
)
or ( not b_in(0)
and b_in(2)
and b_in
(3));
d <= ( b_in(0) and not b_in(1) and not b_in(2) and not b_in
(3))
or ( not b_in(0) and not b_in(1) and b_in(2) and not b_in
(3))
or ( b_in(0) and b_in(1) and b_in(2)
)
or ( not b_in(0) and b_in(1) and not b_in(2) and b_in
(3));
e <= ( b_in(0) and not b_in(1) and not b_in(2)
)
or ( not b_in(1) and b_in(2) and not b_in
(3));

```

```

        or ( (3))                                and not b_in
        or ( (3)) b_in(0)
        or ( (3));                                and not b_in(2) and not b_in

        f <= ( b_in(0)                         and not b_in(2) and not b_in
                (3))
                or ( (3)) b_in(1) and not b_in(2) and not b_in
                or ( (3)) b_in(0) and b_in(1)           and not b_in
                or ( (3)) b_in(0) and not b_in(1) and b_in(2) and b_in
                (3));

        g <= ( (3)) not b_in(1) and not b_in(2) and not b_in
                or ( (3)) b_in(0) and b_in(1) and b_in(2) and not b_in
                or ( (3)) (not b_in(0) and not b_in(1) and b_in(2) and b_in
                (3));
        elsif ( (to_integer(unsigned(b_in)) > 9) and (hb = '1') ) then
            — if dec. display and greater 9: displ. off
            a <= '1'; b <= '1'; c <= '1'; d <= '1';
            e <= '1'; f <= '1'; g <= '1';
        end if;
    end if;
end process;

end behave;

```

Listing 17: Seven segment display VHDL code

```

library ieee;
use ieee.std_logic_1164.all;

entity tb_seven_seg is
end tb_seven_seg;



---


architecture test of tb_seven_seg is

component seven_seg
port
(
    ltn : in std_logic;                                —
    enable_decoder : in std_logic;                      —
    bln : in std_logic;                                —
    segment_test : in std_logic;                        —
    hb : in std_logic;                                — if
    0, hexdec., if 1 displays 0 to 9
    b_in : in std_logic_vector(3 downto 0);           —
    input number
    a,b,c,d,e,f,g : out std_logic;                   —
    output for segments
);
end component;

signal b_in : std_logic_vector(3 downto 0);
signal ltn, bln, hb : std_logic;
signal a, b, c, d, e, f, g : std_logic;

```

```

begin

    dut: seven_seg
    port map
    (
        ltn , bln , hb , b_in , a , b , c , d , e , f , g
    );

    ltn    <= '1' after 0 ns; -- '1' after 5 ns;
    bln    <= '0' after 0 ns; -- '0' after 5 ns;
    hb     <= '1' after 0 ns; -- '1' after 5 ns;

    b_in  <= "0000" after 0 ns , "0001" after 10 ns ,
            "0010" after 20 ns , "0011" after 30 ns ,
            "0100" after 40 ns , "0101" after 50 ns ,
            "0110" after 60 ns , "0111" after 70 ns ,
            "1000" after 80 ns , "1001" after 90 ns ,
            "1010" after 100 ns , "1011" after 110 ns ,
            "1100" after 120 ns , "1101" after 130 ns ,
            "1110" after 140 ns , "1111" after 150 ns;

end test;

```

Listing 18: Seven segment display Testbench

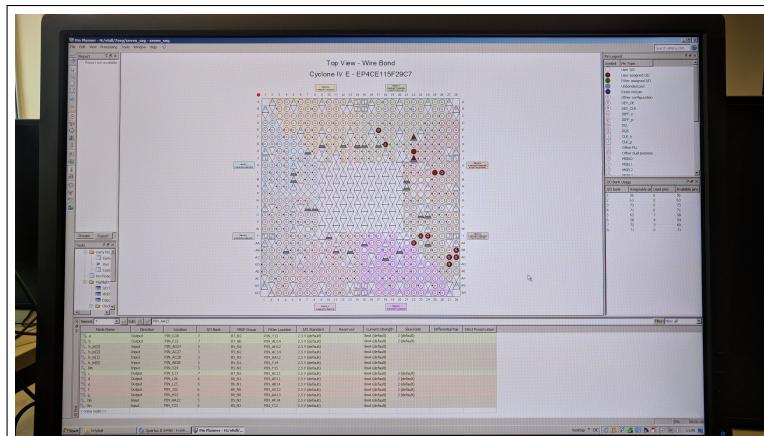


Abbildung 9: Ausführung auf dem Entwicklungsboard DE2-115

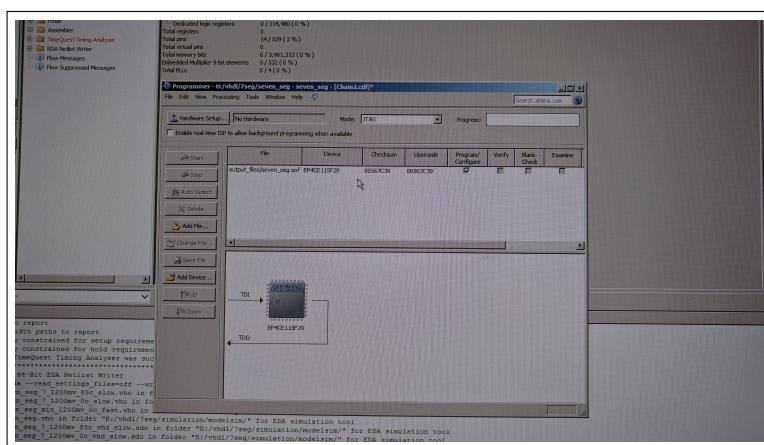


Abbildung 10: Ausführung auf dem Entwicklungsboard DE2-115n



Abbildung 11: Ausführung auf dem Entwicklungsboard DE2-115

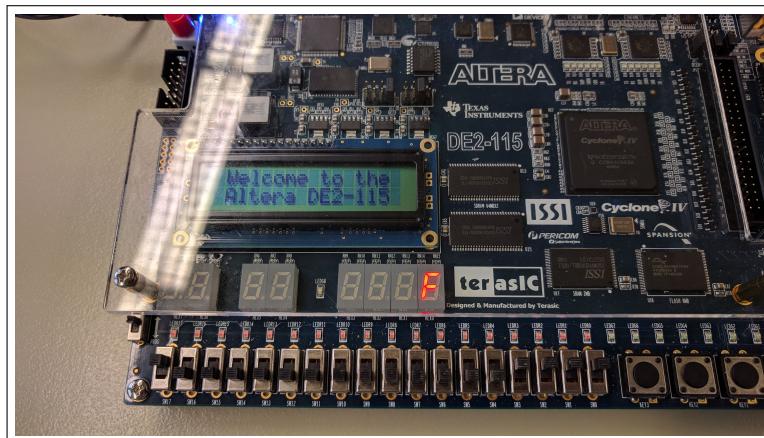


Abbildung 12: Ausführung auf dem Entwicklungsboard DE2-115



Abbildung 13: Ausführung auf dem Entwicklungsboard DE2-115

## 2 Laboraufgaben: 2. Erkennung einer Bit-Sequenz

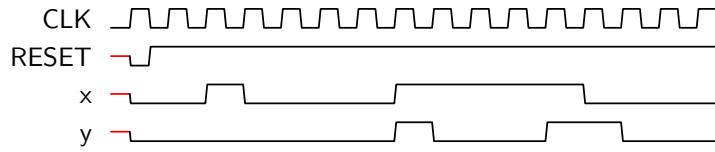


Abbildung 14: Impulsdiagramm des Zustandsautomaten zur Bit-Sequenzerkennung

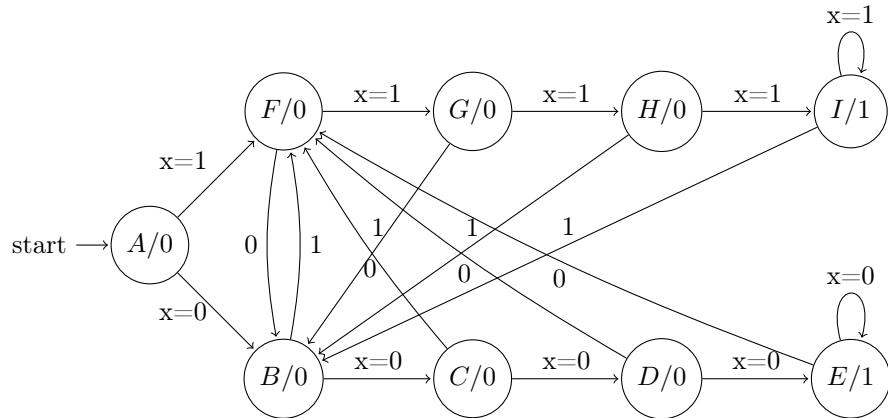


Abbildung 15: Zustandsdiagramm zur Erkennung einer gleichförmigen Bit-Sequenz

### 2.1 Aufgabenstellungen A

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY bit_sequence_a IS
PORT ( CLK, RST, X : IN std_logic;
        Y : OUT std_logic;
        Z : OUT std_logic_vector(8 downto 0));
END bit_sequence_a;

ARCHITECTURE behave OF bit_sequence_a IS

TYPE States IS (A, B, C, D, E, F, G, H, I);
SIGNAL state , nextState: States;

BEGIN

    first : PROCESS (CLK, RST, state)
    BEGIN
        — reset the state machine
        IF (RST = '0') THEN
            state <= A;
        END IF;
        — next state
        IF (rising_edge(CLK) AND RST = '1') THEN
            state <= nextState;
        END IF;
    END PROCESS first;

    process (state, X)
    begin
        case state is
            when A => if X = '0' then
                nextState <= B;
                Y <= '0';
            else
                nextState <= F;
                Y <= '1';
            end if;
            when B => if X = '0' then
                nextState <= C;
                Y <= '0';
            else
                nextState <= F;
                Y <= '1';
            end if;
            when C => if X = '0' then
                nextState <= D;
                Y <= '0';
            else
                nextState <= G;
                Y <= '1';
            end if;
            when D => if X = '0' then
                nextState <= E;
                Y <= '0';
            else
                nextState <= H;
                Y <= '1';
            end if;
            when E => if X = '0' then
                nextState <= E;
                Y <= '0';
            else
                report "Sequence detected" severity error;
                Y <= '1';
            end if;
            when F => if X = '1' then
                nextState <= G;
                Y <= '1';
            else
                nextState <= B;
                Y <= '0';
            end if;
            when G => if X = '1' then
                nextState <= H;
                Y <= '1';
            else
                nextState <= B;
                Y <= '0';
            end if;
            when H => if X = '1' then
                nextState <= I;
                Y <= '1';
            else
                nextState <= B;
                Y <= '0';
            end if;
            when I => if X = '1' then
                nextState <= I;
                Y <= '1';
            else
                report "Sequence detected" severity error;
                Y <= '0';
            end if;
        end case;
    end process;
    Z <= state;
end;

```

```

second : PROCESS ( state , X)
BEGIN
    CASE state IS
        WHEN A =>           IF ( X = '0' ) THEN nextState <= B;
                                ELSE nextState <= F;
                                END IF;
        WHEN B =>           IF ( X = '0' ) THEN nextState <= C;
                                ELSE nextState <= F;
                                END IF;
        WHEN C =>           IF ( X = '0' ) THEN nextState <= D;
                                ELSE nextState <= F;
                                END IF;
        WHEN D =>           IF ( X = '0' ) THEN nextState <= E;
                                ELSE nextState <= F;
                                END IF;
        WHEN E =>           IF ( X = '0' ) THEN nextState <= E;
                                ELSE nextState <= F;
                                END IF;


---


        WHEN F =>           IF X = '1' THEN nextState <= G;
                                ELSE nextState <= B;
                                END IF;
        WHEN G =>           IF X = '1' THEN nextState <= H;
                                ELSE nextState <= B;
                                END IF;
        WHEN H =>           IF X = '1' THEN nextState <= I;
                                ELSE nextState <= B;
                                END IF;
        WHEN I =>           IF X = '1' THEN nextState <= I;
                                ELSE nextState <= B;
                                END IF;
        WHEN OTHERS => nextState <= A;

    END CASE;
END PROCESS second;

third : PROCESS ( state )
BEGIN
    CASE state IS
        WHEN A =>           Z <= "000000001"; Y <= '0';
        WHEN B =>           Z <= "000000010"; Y <= '0';
        WHEN C =>           Z <= "000000100"; Y <= '0';
        WHEN D =>           Z <= "000001000"; Y <= '0';
        WHEN E =>           Z <= "000010000"; Y <= '1';
        WHEN F =>           Z <= "000100000"; Y <= '0';
        WHEN G =>           Z <= "001000000"; Y <= '0';
        WHEN H =>           Z <= "010000000"; Y <= '0';
        WHEN I =>           Z <= "100000000"; Y <= '1';
        WHEN OTHERS => Z <= "000000001"; Y <= '0';

    END CASE;
END PROCESS third;

END behave;

```

Listing 19: State Machine VHDL Code

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

```

```

ENTITY bit_sequence_a_tb IS
END bit_sequence_a_tb;

ARCHITECTURE behavior OF bit_sequence_a_tb IS

COMPONENT bit_sequence_a
PORT ( CLK, RST, X : IN std_logic;
       Y : OUT std_logic;
       Z : OUT std_logic_vector(8 downto 0));
END COMPONENT;

SIGNAL s_clk : std_logic := '0';
SIGNAL s_rst, s_x, s_y : std_logic;
SIGNAL s_z : std_logic_vector(8 downto 0);

BEGIN

dut : ENTITY work.bit_sequence_a
PORT MAP (
    CLK => s_clk,
    RST => s_rst,
    X => s_x,
    Y => s_y,
    Z => s_z
);

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

reset_stimul: PROCESS
BEGIN
    s_rst <= '0'; WAIT FOR 10 ns;
    s_rst <= '1'; WAIT FOR 100 ns;
END PROCESS;

data_stimul: PROCESS
BEGIN
    s_x <= '0'; WAIT FOR 20 ns;
    s_x <= '1'; WAIT FOR 80 ns;
    s_x <= '0'; WAIT FOR 50 ns;
    s_x <= '1'; WAIT FOR 10 ns;
    s_x <= '0'; WAIT FOR 20 ns;
END PROCESS;

END behavior;

```

Listing 20: State Machine Testbench

## 2.2 Aufgabenstellungen B

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY bit_sequence_b IS
PORT ( CLK, RST, X : IN std_logic;
       Y : OUT std_logic;
       Z : OUT std_logic_vector(8 downto 0));

```

```

END bit_sequence_b;

ARCHITECTURE behave OF bit_sequence_b IS
TYPE States IS (A, B, C, D, E, F, G, H, I);
SIGNAL state, nextState: States;

BEGIN
    first : PROCESS (CLK, RST, state)
    BEGIN
        — reset the state machine
        IF (RST = '0') THEN
            state <= A;
        END IF;
        — next state
        IF (rising_edge(CLK) AND RST = '1') THEN
            state <= nextState;
        END IF;
    END PROCESS first;

    second : PROCESS (state, X)
    BEGIN
        CASE state IS
            WHEN A =>           IF (X = '0') THEN nextState <= B;
                                ELSE nextState <= F;
                                END IF;
            WHEN B =>           IF (X = '0') THEN nextState <= C;
                                ELSE nextState <= F;
                                END IF;
            WHEN C =>           IF (X = '0') THEN nextState <= D;
                                ELSE nextState <= F;
                                END IF;
            WHEN D =>           IF (X = '0') THEN nextState <= E;
                                ELSE nextState <= F;
                                END IF;
            WHEN E =>           IF (X = '0') THEN nextState <= E;
                                ELSE nextState <= F;
                                END IF;
            —
            WHEN F =>           IF X = '1' THEN nextState <= G;
                                ELSE nextState <= B;
                                END IF;
            WHEN G =>           IF X = '1' THEN nextState <= H;
                                ELSE nextState <= B;
        END CASE;
    END PROCESS second;

```

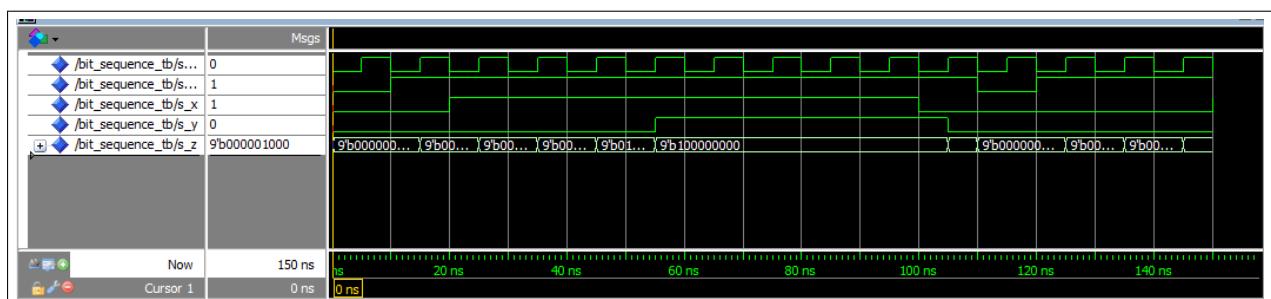


Abbildung 16: State Machine Modelsim Simulation

```

                END IF;
WHEN H =>      IF X = '1' THEN nextState <= I;
                  ELSE nextState <= B;
                  END IF;
WHEN I =>      IF X = '1' THEN nextState <= I;
                  ELSE nextState <= B;
                  END IF;
WHEN OTHERS => nextState <= A;

END CASE;
END PROCESS second;

third : PROCESS (state)
BEGIN

CASE state IS
  WHEN A => Z <= "000000000"; Y <= '0';
  WHEN B => Z <= "000000011"; Y <= '0';
  WHEN C => Z <= "000000101"; Y <= '0';
  WHEN D => Z <= "000001001"; Y <= '0';
  WHEN E => Z <= "000010001"; Y <= '1';
  WHEN F => Z <= "000100001"; Y <= '0';
  WHEN G => Z <= "001000001"; Y <= '0';
  WHEN H => Z <= "010000001"; Y <= '0';
  WHEN I => Z <= "100000001"; Y <= '1';
  WHEN OTHERS => Z <= "000000000"; Y <= '0';
END CASE;
END PROCESS third;

END behave;

```

Listing 21: State Machine VHDL Code

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bit_sequence_b_tb IS
END bit_sequence_b_tb;

ARCHITECTURE behavior OF bit_sequence_b_tb IS

COMPONENT bit_sequence_b
PORT ( CLK, RST, X : IN std_logic;
       Y : OUT std_logic;
       Z : OUT std_logic_vector(8 downto 0));
END COMPONENT;

SIGNAL s_clk : std_logic := '0';
SIGNAL s_rst, s_x, s_y : std_logic;
SIGNAL s_z : std_logic_vector(8 downto 0);

BEGIN

dut : ENTITY work.bit_sequence_b
PORT MAP (
            CLK => s_clk,
            RST => s_rst,
            X => s_x,
            Y => s_y,
            Z => s_z

```

```

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

reset_stimul: PROCESS
BEGIN
    s_rst <= '0'; WAIT FOR 10 ns;
    s_rst <= '1'; WAIT FOR 100 ns;
END PROCESS;

data_stimul: PROCESS
BEGIN
    s_x <= '0'; WAIT FOR 20 ns;
    s_x <= '1'; WAIT FOR 80 ns;
    s_x <= '0'; WAIT FOR 50 ns;
    s_x <= '1'; WAIT FOR 10 ns;
    s_x <= '0'; WAIT FOR 20 ns;
END PROCESS;

```

END behavior;

Listing 22: State Machine Testbench

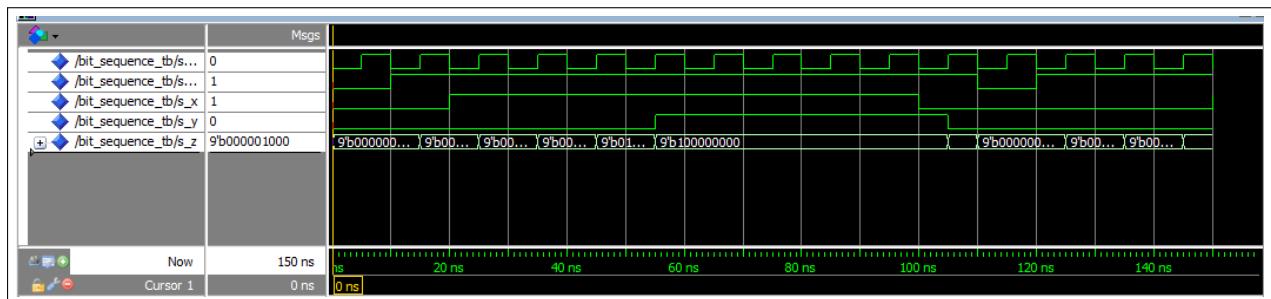


Abbildung 17: State Machine Modelsim Simulation

### 2.3 Aufgabenstellungen C

Die Aufgabenstellungen A und B wurden mit der CASE Anweisung innerhalb eines PROCESS-Blocks geschrieben, da es uns schon bekannt ist, dass diese Lösung effizienter ist und wurde uns während der Vorlesung Digital Technik empfohlen.

### 3 Laboraufgaben: 3. Takte, Zähler und Zeitgeber

#### 3.1 BCD-Zähler und Codierung

In Abbildung 18 ist das vereinfachte Systemschaltbild zur Aufgabenstellung dargestellt. Der BCD-Counter kann über die Schalter auf dem Entwicklungsbord initialisiert werden (Startwerteingabe). Der Zählerstand soll sowohl auf einer Siebensegmentanzeige dargestellt als auch auf dem LCD-Display (siehe Abbildung 19) angezeigt werden.

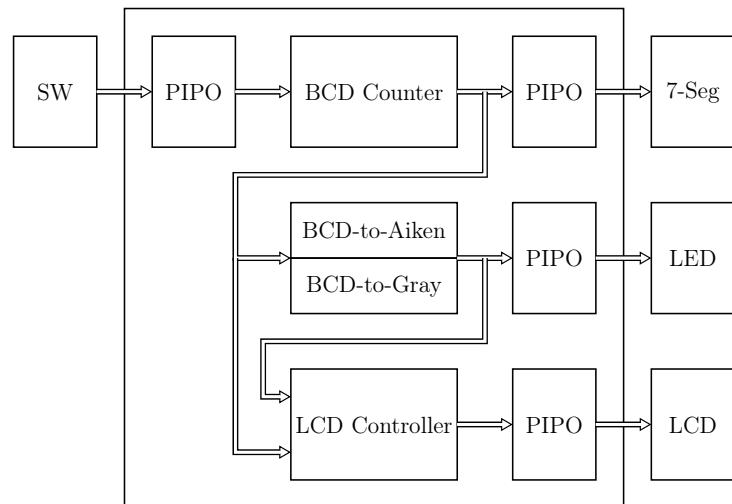


Abbildung 18: L3 Systemschaltbild zur Aufgabenstellung



Abbildung 19: L3 Anzeige auf dem LC Display

$i_{10}$	BCD	Gray	Aiken
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	1011
6	0110	0101	1100
7	0111	0100	1101
8	1000	1100	1110
9	1001	1101	1111

Tabelle 1: Codierung von BCD, Gray, Aiken

```

library ieee;
use ieee.std_logic_1164.all;

entity bcd_counter is
  port
  
```

```

(
    clk , ld      : in std_logic;
    Clock Signal, Load Signal for specific start value
    clr          : in std_logic;
    Clear Signal
    ld_4bit      : in std_logic_vector(3 downto 0);
    4 bit input load start value
    out_4bit     : out std_logic_vector(3 downto 0)
    4 bit output
);
end bcd_counter;

architecture behave of bcd_counter is
type state is ( s0 , s1 , s2 , s3 , s4 , s5 , s6 , s7 , s8 , s9 );
signal act_state , next_state  : state;
signal out_temp                : std_logic_vector(3 downto 0);
begin
process( clk , clr , ld ) _____
    Zustandsaktualisierung
begin
    if ( rising_edge(clk) ) then
        synchron reset
        if ( clr = '1' ) then
            s0
            act_state <= s0;
        elsif ( ld = '1' ) then
            load: evaluate the input
            case ld_4bit is
                when "0000" => act_state <= s0;
                when "0001" => act_state <= s1;
                when "0010" => act_state <= s2;
                when "0011" => act_state <= s3;
                when "0100" => act_state <= s4;
                when "0101" => act_state <= s5;
                when "0110" => act_state <= s6;
                when "0111" => act_state <= s7;
                when "1000" => act_state <= s8;
                when "1001" => act_state <= s9;
                when others => act_state <= s0;
            end case;
        elsif ( ld = '0' ) then
            case:
                if ( clr = '0' ) then
                    act_state <= next_state;
                end if;
            end if;
        end if;
    end process; _____
    Zustandsaktualisierung
END

process( act_state ) _____
    Ausgang
begin
    case act_state is
        when s0 => out_temp <= "0000"; next_state <= s1;
        when s1 => out_temp <= "0001"; next_state <= s2;
        when s2 => out_temp <= "0010"; next_state <= s3;
        when s3 => out_temp <= "0011"; next_state <= s4;
        when s4 => out_temp <= "0100"; next_state <= s5;
        when s5 => out_temp <= "0101"; next_state <= s6;
    end case;
end process;

```

```

        when      s6 => out_temp <= "0110"; next_state <= s7;
        when      s7 => out_temp <= "0111"; next_state <= s8;
        when      s8 => out_temp <= "1000"; next_state <= s9;
        when      s9 => out_temp <= "1001"; next_state <= s0;
        when others => out_temp <= "0000"; next_state <= s0;
    end case;
end process; ————— END Ausgang

out_4bit <= out_temp;
end behave;

```

Listing 23: BCD Counter VHDL Code

```

library ieee;
use ieee.std_logic_1164.all;

entity tb_bcd_counter is
end tb_bcd_counter;

————

architecture test of tb_bcd_counter is

component bcd_counter
port
(
    clk , ld      : in std_logic; ——
    Clock Signal, Load Signal for specific start value ——
    clr          : in std_logic; ——
    Clear Signal ——
    ld_4bit       : in std_logic_vector(3 downto 0); ——
    4 bit input load start value ——
    out_4bit      : out std_logic_vector(3 downto 0); ——
    4 bit output ——
);
end component;

signal clk , ld , clr   : std_logic;
signal ld_4bit         : std_logic_vector(3 downto 0) := "UUUU";
signal out_4bit         : std_logic_vector(3 downto 0) := "UUUU";

begin

dut: bcd_counter
port map
(
    clk , ld , clr , ld_4bit , out_4bit
);

process ————— Clock
    Stimuli
begin
    clk <= '0';
    wait for 5 ns;
    clk <= '1';
    wait for 5 ns;
end process; ————— END
    Clock Stimuli

    clr      <= '1' after 0 ns , '0' after 15 ns;

```

```

ld      <= '1' after 0 ns, '0' after 25 ns;
ld_4bit <= "0011" after 0 ns;

end test;

```

Listing 24: BCD Counter Testbench

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bcd_to_aiken is
    port
    (
        bcd_in      : in std_logic_vector(3 downto 0);
        aiken_out   : out std_logic_vector(3 downto 0)
    );
end bcd_to_aiken;

architecture behave of bcd_to_aiken is
begin
    process( bcd_in )
    begin
        if ( unsigned(bcd_in) < 5 ) then
            — 1st half from 0000 to 0100
            aiken_out <= bcd_in;
        elsif ( (unsigned(bcd_in) > 4) and (unsigned(bcd_in) < 10) ) then
            — 2nd half from 1011 to 1111
            aiken_out <= std_logic_vector((unsigned(bcd_in) + 6) );
        else
            — if bcd > 9 => ERROR
            aiken_out <= "UUUU";
        end if;
    end process;
end behave;

```

Listing 25: BCD to Aiken VHDL Code

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bcd_to_gray is
    port
    (
        bcd_in      : in std_logic_vector(3 downto 0);
        gray_out    : out std_logic_vector(3 downto 0)
    );
end bcd_to_gray;

architecture behave of bcd_to_gray is
begin
    process( bcd_in )
    begin
        if ( unsigned(bcd_in) < 10 ) then
            — valid BCD range
            gray_out(3) <= bcd_in(3);
            gray_out(2) <= bcd_in(3) or bcd_in(2);
            — KV Diagram via
    end if;
end process;

```

```

        gray_out(1) <= bcd_in(2) xor bcd_in(1);
        gray_out(0) <= bcd_in(1) xor bcd_in(0);
    else
        > 9 => ERROR
        gray_out <= "UUUU";
    end if;
end process;
end behave;

```

Listing 26: BCD to Gray Testbench

```

library ieee;
use ieee.std_logic_1164.all;

entity codierer is
    port
    (
        clk , clr , ld      :  in std_logic;
        input                  :  in std_logic_vector(3 downto 0);
        output                 :  out std_logic_vector(6 downto 0)
    );
end codierer;

architecture behave of codierer is
    Component List
    component pipo
        port
        (
            clk , clr :  in std_logic;
            -- clock and clear
            en       :  in std_logic;
            -- enable
            d        :  in std_logic_vector( 3 downto 0 );
            -- data input
            q        :  out std_logic_vector( 3 downto 0 );
            -- data output
        );
    end component;

    component bcd_counter
        port
        (
            clk , ld      :  in std_logic;
            -- Clock Signal, Load Signal for specific start value
            clr      :  in std_logic;
            -- Clear Signal
            ld_4bit   :  in std_logic_vector(3 downto 0);
            -- 4 bit input load start value
            out_4bit   :  out std_logic_vector(3 downto 0);
            -- 4 bit output
        );
    end component;

    component seven_seg
        port
        (
            ltn       :  in std_logic;
            -- enable decoder
            bln      :  in std_logic;
            -- segment test
        );
    end component;

```

```

        hb      : in std_logic;           — if
        0, hexdec., if 1 displays 0 to 9
        b_in   : in std_logic_vector(3 downto 0);    —
            input number
        a,b,c,d,e,f,g : out std_logic           —
            output for segments
    );
end component;

component bcd_to_aiken
port
(
    bcd_in      : in std_logic_vector(3 downto 0);
    aiken_out   : out std_logic_vector(3 downto 0)
);
end component;

component bcd_to_gray
port
(
    bcd_in      : in std_logic_vector(3 downto 0);
    gray_out    : out std_logic_vector(3 downto 0)
);
end component;

```

---

END

#### Component List

```

signal input_pipo_output      : std_logic_vector(3 downto 0);
signal bcd_output   : std_logic_vector(3 downto 0);
signal to_seven_seg          : std_logic_vector(3 downto 0);
signal aiken_to_pipo         : std_logic_vector(3 downto 0);
signal to_aiken_led          : std_logic_vector(3 downto 0);
signal gray_to_pipo          : std_logic_vector(3 downto 0);
signal to_gray_led           : std_logic_vector(3 downto 0);
begin
    input_pipo  : pipo
        port map ( clk , '0', '1', input, input_pipo_output );
    bcd       : bcd_counter
        port map ( clk , ld , clr , input_pipo_output, bcd_output );
    in_sevSeg : pipo
        port map ( clk , '0', '1', bcd_output, to_seven_seg );
    sevSeg    : seven_seg
        port map ( '1', '0', '1', to_seven_seg,
                    output(0), output(1), output(2), output(3),
                    output(4), output(5), output(6) );           — low
                                                active
    bcd_aiken : bcd_to_aiken
        port map ( bcd_output, aiken_to_pipo );
    bcd_gray  : bcd_to_gray
        port map ( bcd_output, gray_to_pipo );
    aiken_pipo: pipo
        port map ( clk , '0', '1', aiken_to_pipo, to_aiken_led );
    gray_pipo : pipo
        port map ( clk , '0', '1', gray_to_pipo, to_gray_led );
end behave;

```

Listing 27: Codierer VHDL Code

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity tb_codierer is
end tb_codierer;

architecture test of tb_codierer is

component codierer
port
(
    clk , clr , ld      : in std_logic;
    input                 : in std_logic_vector(3 downto 0);
    output                : out std_logic_vector(6 downto 0)
);
end component;

signal clk , clr , ld : std_logic;
signal input   : std_logic_vector(3 downto 0);
signal output  : std_logic_vector(6 downto 0);

begin

dut: codierer
port map
(
    clk , clr , ld , input , output
);

process
begin
    clk <= '0'; wait for 5 ns;
    clk <= '1'; wait for 5 ns;
end process;
clr <= '1' after 0 ns , '0' after 4 ns;
ld  <= '1' after 0 ns , '0' after 17 ns;
input <= "0011" after 0 ns;

end test;

```

Listing 28: Codierer Testbench

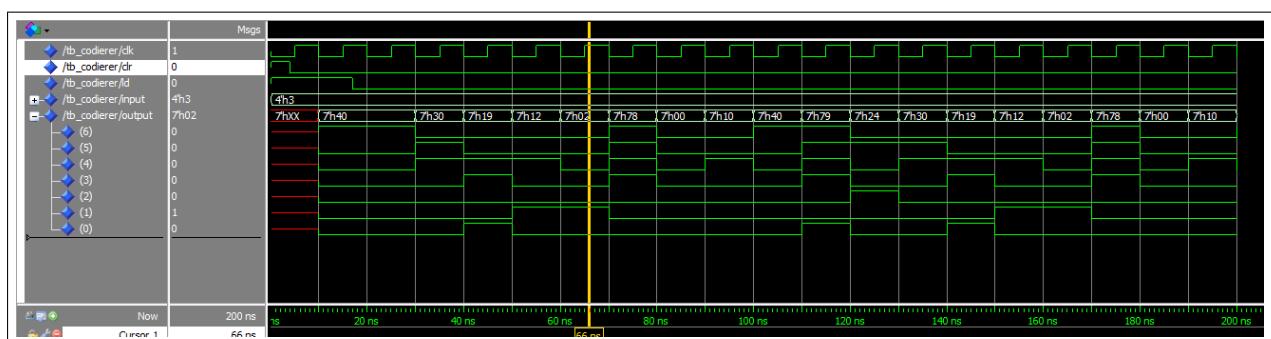


Abbildung 20: State Machine Modelsim Simulation

### 3.2 2-Digit BCD-Arithmetik

### 3.3 3-Digit BCD-Zähler und 24h-Uhr

## 4 Booth-Algorithmus mit Datenpfad und Steuerwerk

Das Flußdiagramm nach Abbildung 21 beschreibt die vorzeichenrichtige Multiplikation von Zahlen im Zweierkomplement durch den sog. Booth-Algorithmus.

Der Multiplikand X wird in das Register M geladen, der Multiplikator Y in das Register P. Das Register A dient als Akkumulator. Das Ergebnis bildet sich im Akkumulator und im Register P.

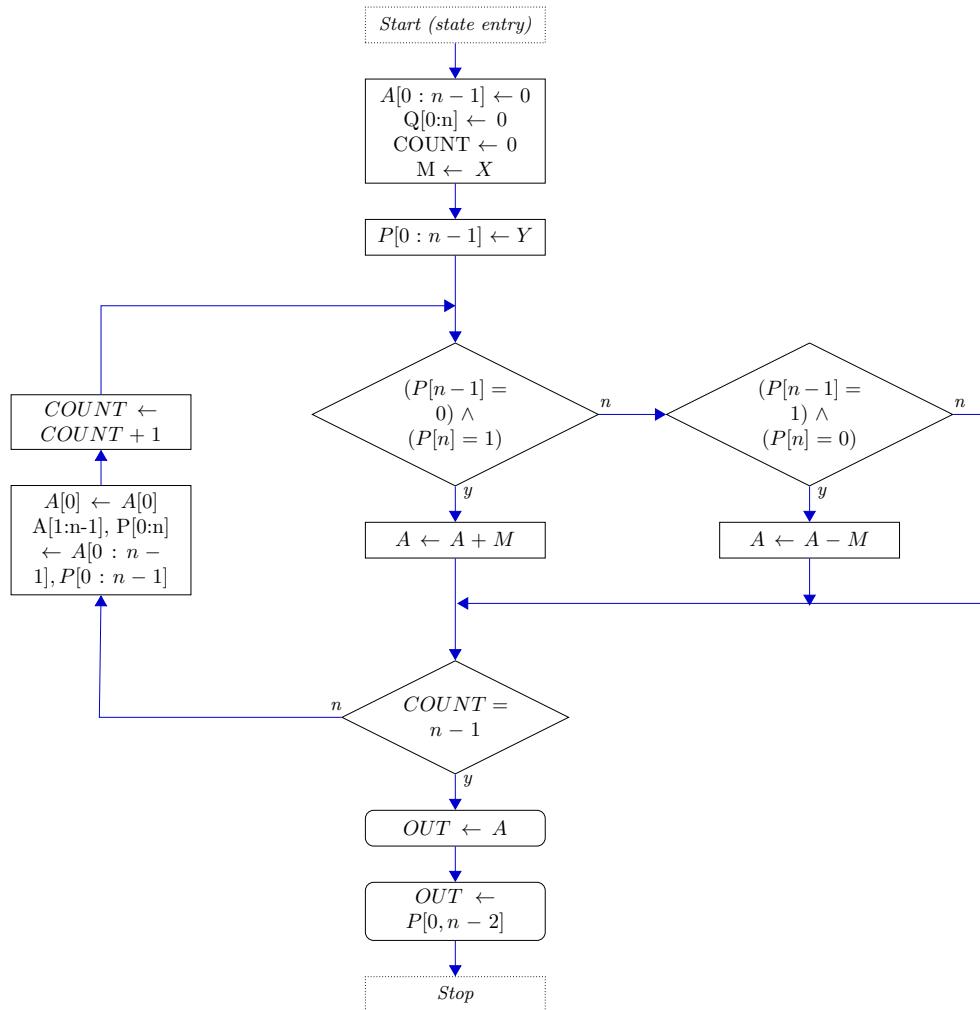


Abbildung 21: L4 ASM-Chart Booth-Algorithmus

## 4.1 Aufgabenstellungen A

Anhand des Booth-Algorithmus wird eine Multiplikation (bzw. Division) zweier vorzeichenbehafteten Zahlen in  $n$  Schritte durchgeführt, wobei  $n$  der Anzahl der als Ergebnis darzustellenden Bits entspricht.

Dieser Algorithm ist besonders bei einer längeren Rechengröße geeignet, da die Anzahl partieller Produkte reduziert werden kann.

Im Folgenden werden die handschriftlichen Berechnungen von drei Multiplikationen gezeigt.

Bitkombination	Operation
01	Addition
10	Subtraktion
00 / 11	-

$$P_{1,10} = 2 \cdot 2 \Rightarrow P_{1,2} = 0010 \cdot 0010$$

$$M = 0010$$

Schritt	A	P Register	Platzhalter	Durchgeführte Operation
Init.	0000	0010	0	
1	0000	0001	0	Check Bits: 00 Schift
2	1110 1111	0001 0000	0 1	Check Bits: 10 $A \leftarrow A - M$ Schift
3	0001 0000	0000 1000	1 0	Check Bits: 01 $A \leftarrow A + M$ Schift
4	0000	0100	0	Check Bits: 00 Schift

$$\text{Ergebnis: } P_{1,2_{A,P}} = 00000100 \Rightarrow P_{1,10_{A,P}} = 4$$

$$P_{2,10} = 6 \cdot 7 \Rightarrow P_{2,2} = 0110 \cdot 0111$$

$$M = 0110$$

Schritt	A	P Register	Platzhalter	Durchgeführte Operation
Init.	0000	0111	0	
1	1010 1101	0111 0011	0 1	Check Bits: 10 $A \leftarrow A - M$ Schift
2	1110	1001	1	Check Bits: 11 Schift
3	1111	0100	1	Check Bits: 11 Schift
4	0101 0010	0100 1010	1 0	Check Bits: 01 $A \leftarrow A + M$ Schift

$$\text{Ergebnis: } P_{2,2_{A,P}} = 00101010 \Rightarrow P_{2,10_{A,P}} = 42$$

$$P_{3,10} = 6 \cdot (-7) \Rightarrow P_{3,2} = 0110 \cdot 1001$$

$$M = 0110$$

Schritt	A	P Register	Platzhalter	Durchgeführte Operation
Init.	0000	1001	0	
1	1010	1001	0	Check Bits: 10 $A \leftarrow A - M$
	1101	0100	1	Schift
2	0011	0100	1	Check Bits: 01 $A \leftarrow A + M$
	0001	1010	0	Schift
3	0000	1101	0	Check Bits: 00 Schift
	1010	1101	0	Check Bits: 10 $A \leftarrow A - M$
4	1101	0110	1	Schift

$$\text{Ergebnis: } P_{3,2_{A,P}} = 11010110 \Rightarrow P_{3,10_{A,P}} = -42$$

#### 4.1.1 Arithmetische Einheit

Abbildung 22 zeigt eine aus Datenpfad und Steuerwerk bestehende arithmetische Einheit, die das Produkt von zwei 4-Bit Zweirkomplement-Zahlen nach dem Booth-Algorithmus berechnet.

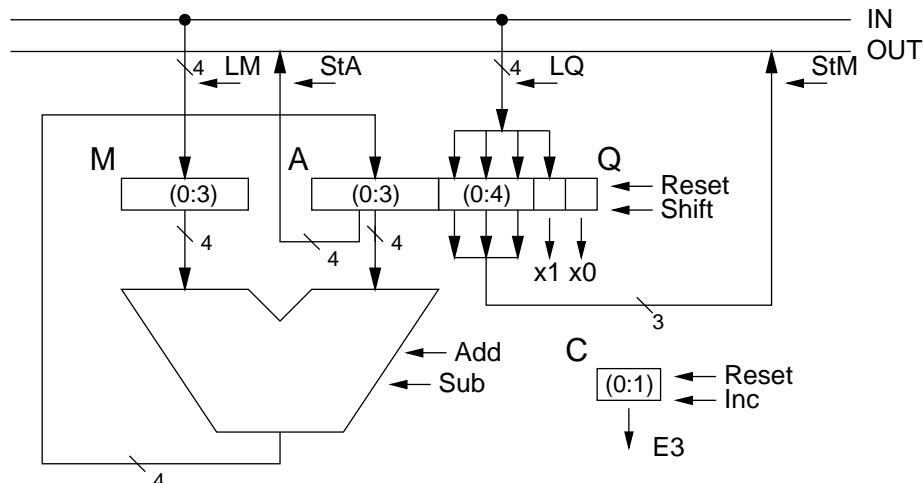


Abbildung 22: L4 Arithmetische Einheit für den Booth-Algorithmus

Die Funktionalität der Arithmetischen Einheit, bereits um INBUS und OUTBUS erweitert, ist wie Folgt:

Bezeichnung	Beschreibung
Reset	Zurücksetzung in den Anfangszustand (alle Registerinhalte auf Null)
LM	Übernahme der Daten aus dem INBUS in das M-Register
LQ	Übernahme der Daten aus dem INBUS in das Q-Register
Add	Addition von M und A, die Summe wird in A geschrieben
Sub	Subtraktion von M und A, die Differenz wird in A geschrieben
Shift	A und Q werden gemeinsam nach rechts geschoben, Bit 0 wird dupliziert
Inc	Der Zähler wird um eins erhöht
StA	Der Inhalt von A wird an den OUTBUS übergeben
StQ	Der Inhalt von Q wird an den OUTBUS übergeben
E3	Der Zähler ist gleich drei
x1	Bit n-1 aus Q
x0	Bit n aus Q

#### 4.1.2 Moore-Automaten

Abbildung 23 zeigt einen dem Flussdiagramm (Abb. 21) entsprechenden Moore-Automaten.

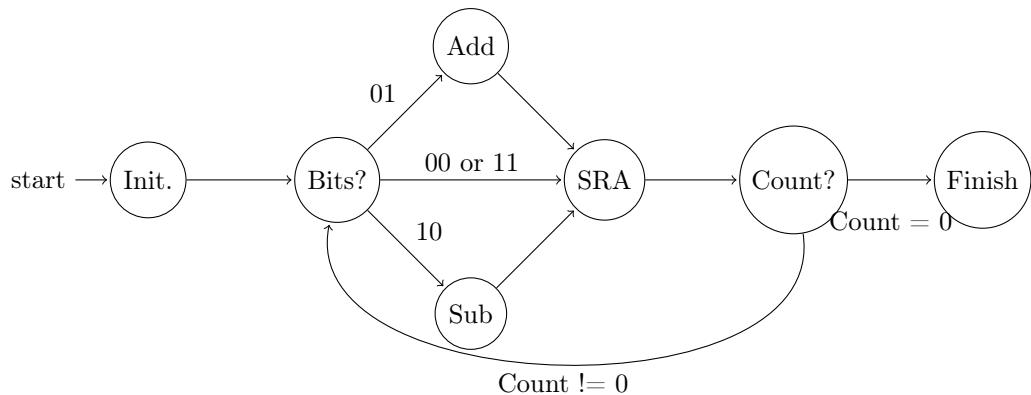


Abbildung 23: Moore-Automaten

#### 4.1.3 VHDL Beschreibung

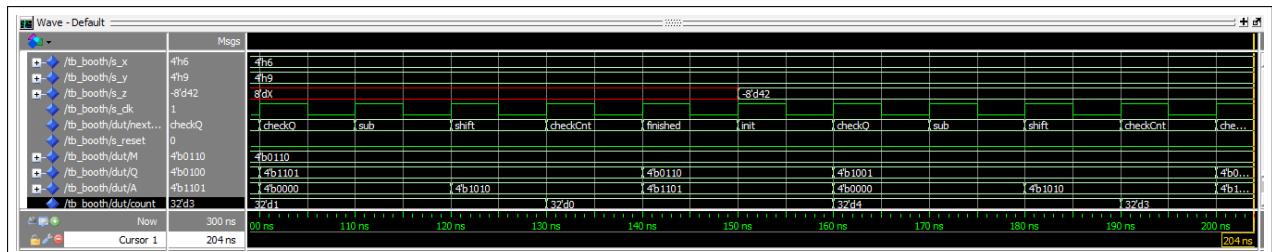


Abbildung 24: State Machine Modelsim Simulation

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity booth is
  port(
    x, y : in std_logic_vector(3 downto 0);
    z     : out std_logic_vector(7 downto 0);
    clk, reset : in std_logic
  );
end booth;

architecture behavior of booth is
type statetype is (init, checkQ, add, sub, shift, checkCnt, finished);
signal state, next_state: statetype := init;
signal Qadditional : std_logic;
signal M, Q           : std_logic_vector(3 downto 0);
signal A             : std_logic_vector(3 downto 0);
signal count         : integer;
signal hand          : std_logic;
begin

--Zustandsaktualisierung
  pr1: process(clk, reset)
  begin
    if reset = '1' then
      state <= init;
    end if;
    case state is
      when init =>
        ...
      when checkQ =>
        ...
      when add =>
        ...
      when sub =>
        ...
      when shift =>
        ...
      when checkCnt =>
        ...
      when finished =>
        ...
    end case;
  end process;
  ...
end architecture;
  
```

```

        elsif (rising_edge(clk)) then
            state <= next_state;
        end if;
    end process pr1;
--Folgezustandsberechnung
pr2: process(state, Q, Qadditional, count)
begin
    --next_state <= state;
    case state is
        when init =>
            next_state <= checkQ;
        when checkQ =>
            if ((Q(0) = '1' and Qadditional = '1') or (Q(0) = '0' and Qadditional = '0')) then
                next_state <= shift;
            elsif(Q(0) = '0' and Qadditional = '1') then
                next_state <= add;
            elsif(Q(0) = '1' and Qadditional = '0') then
                next_state <= sub;
            end if;
        when add =>
            next_state <= shift;
        when sub =>
            next_state <= shift;
        when shift =>
            next_state <= checkCnt;
        when checkCnt =>
            if(count = 0) then
                next_state <= finished;
            else
                next_state <= checkQ;
            end if;
        when others =>
            next_state <= init;
    end case;
end process pr2;
--Ausgangsberechnung
pr3: process(state, x, y)
begin
    case state is
        when init =>
            A <= "0000";
            M <= x;
            Q <= y;
            Qadditional <= '0';
            count <= 4;
        when checkQ =>
            -- nur next_state
        when add =>
            A <= std_logic_vector(signed(A) + signed(M));
        when sub =>
            A <= std_logic_vector(signed(A) - signed(M));
        when shift =>
            count <= count-1;
            hand <= A(0);
            Qadditional <= Q(0); --shift right 1 bit
        when checkCnt =>
            A <= A(3) & A(3 downto 1); --arithmetic shift
                                         right 1 bit
            Q <= hand & Q(3 downto 1); --arithmetic shift
                                         right 1 bit

```

```

        when finished =>
            z <= A & Q;
        when others =>
            --nothing
    end case;
end process pr3;
end behavior;

```

Listing 29: Booth Algorithm VHDL Code

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_booth is
end tb_booth;

architecture testbench of tb_booth is
    component booth is
        port(
            x, y : in std_logic_vector(3 downto 0);
            z     : out std_logic_vector(7 downto 0);
            clk, reset : in std_logic
        );
    end component;

    signal s_x : std_logic_vector(3 downto 0);
    signal s_y : std_logic_vector(3 downto 0);
    signal s_z : std_logic_vector(7 downto 0);
    signal s_clk : std_logic := '0';
    signal s_reset : std_logic := '0';

begin
dut: ENTITY work.booth
PORT MAP (
    x => s_x,
    y => s_y,
    z => s_z,
    clk => s_clk,
    reset => s_reset
);

-- common processes in the separate process
-- test: 6*(-7) = -42 (1101 )
data_stimul_X: PROCESS
BEGIN
    s_x <= "0110"; WAIT FOR 100 ns;
END PROCESS;

data_stimul_Y: PROCESS
BEGIN
    s_y <= "1001"; WAIT FOR 100 ns;
END PROCESS;

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

```

```
END PROCESS;

reset_stimul: PROCESS
BEGIN
    s_reset <= '0'; WAIT FOR 280 ns;
    s_reset <= '1'; WAIT FOR 20 ns;
END PROCESS;

end testbench;
```

Listing 30: Booth Algorithm Testbench