**Contents**

Combinational Circuit Design 2

Sequential Circuit Design 7

Finite State Machines 12

Register Transfer Methodology 20

Hierarchical Design 32

Parameterized Design 35

**Combinational Circuit Design**

**2-4 bit binary decoder**

library ieee;

use ieee.std\_logic\_1164.all;

entity decoder4 is

port(

s: in std\_logic\_vector(1 downto 0);

x: out std\_logic\_vector(3 downto 0)

);

End decoder4;

architecture cond\_arc of decoder4 is

begin

x <= ”0001” when (s=”00”) else

”0010” when (s=”01”) else

”0100” when (s=”10”) else

”1000”;

end cond\_arc;

**4-2 bit priority encoder circuit**

library ieee;

use ieee.std\_logic\_1164.all;

entity prio\_enco4 is

port(

r: in std\_logic\_vector(3 downto 0);

code: out std\_logic\_vector(1 downto 0);

active: out std\_logic

);

end prio\_enco4;

architecture cond\_arc of prio\_enco4 is

begin

code <= ”11” when (r(3)=’1’) else

”10” when (r(2)=’1’) else

”01” when (r(1)=’1’) else

”00”;

active <= r(3) or r(2) or r(1) or r(0);

end cond\_arc;

**Simple arithmetic-logic unit implemented with conditional signal assignment statement**

-- Autheur: Ioannis Fountos

-- architecture: alu with conditional signal assignment statements

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity alu is

port(

ctrl: in std\_logic\_vector(2 downto 0);

src0, src1: in std\_logic\_vector(7 downto 0);

result: out std\_logic\_vector(7 downto 0)

);

end alu;

architecture cond\_arc of alu is

signal sum, diff, inc: std\_logic\_vector(7 downto 0);

begin

inc <= std\_logic\_vector(signed(src0)+1);

sum <= std\_logic\_vector(signed(src0)+ signed(src1));

diff <= std\_logic\_vector(signed(src0)- signed(src1));

result <= inc when (ctrl(2)=’0’) else

sum when (ctrl(1 downto 0)=”00”) else

diff when (ctrl(1 downto 0)=”01” ) else

src0 and src1 when ctrl(1 downto 0)= ”10” else

src0 or src1;

end cond\_arc;

**16-bit Addition, subtraction, increment, decrement circuit**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity alu is

port(

a, b: in std\_logic\_vector(15 downto 0);

crl: in std\_logic\_vector(1 downto 0);

result: out std\_logic\_vector(15 downto 0)

);

end alu;

architecture arc1 of alu is

signal tmp: signed(15 downto 0);

begin

tmp <= signed(a) + signed(b) when ctrl=”00” else

signed(a) - signed(b) when ctrl= ”01” else

signed(a) +1 when ctrl = ”10” else

signed(a) -1 when ctrl = ”11”;

result <= std\_logic\_vector(tmp);

end arc1;

**8-bit Shift Register**

library ieee;

use ieee.std\_logic\_1164.all;

entity register\_8 is

port(

clk: in std\_logic\_vector(7 downto 0);

reset: in std\_logic\_vector(7 downto 0);

d: in std\_logic\_vector(7 downto 0);

q: out std\_logic\_vector(7 downto 0)

);

end register\_8;

architecture arc1 of register\_8 is

begin

process(clk, reset)

begin

if (reset = ’1’) then

q <=(others =>’0’);

elsif (clk’event and clk=’1’) then

q <= d;

end if;

end process;

end arc1;

**8-bit rotate left/right circuit**

--The circuit takes an 8-bit number and with two control signals it rotates the number right or left and for certain

--number of bits according to the control signals’ values.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity rotate is

port(

a: in std\_logic\_vector(7 downto 0);

left\_rait: in std\_logic\_vector(1 downto 0);

n: in std\_logic\_vector(2 downto 0);

y: out std\_logic\_vector(7 downto 0)

);

end rotate;

architecture arc1 of rotate is

signal z : unsigned(2 downto 0);

begin

left\_rait select

z <= unsigned(n) when ’0’, --rotate right

”111” – unsigned(n) when others; --rotate left

with z select

y <= a when ”000”,

a(0) & a(7 downto 1) when”001”,

a(1 downto 0) & a(7 downto 2) when”010”,

a(2 downto 0) & a(7 downto 3) when”011”,

a(3 downto 0) & a(7 downto 4) when”100”,

a(4 downto 0) & a(7 downto 5) when”101”,

a(5 downto 0) & a(7 downto 6) when”110”,

a(6 downto 0) & a(7) when others;

end arc1;

**4-bit gray-code distance calculator**

--This circuit takes as input to 4-bit gray code numbers, it converts them to binary and subtracts them.

--The difference from the subtraction is the gray distance of the initial gray code numbers.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity gray\_dist is

port(

a: in std\_logic\_vector(3 downto 0);

b: in std\_logic\_vector(3 downto 0);

ext: out std\_logic\_vector(3 downto 0)

);

end gray\_dist;

architecture arc1 of gray\_dist is

constant WIDTH: integer :=4;

signal x: std\_logic\_vector(WIDTH-1 downto 0);

signal y: std\_logic\_vector(WIDTH-1 downto 0);

signal z: std\_logic\_vector(WIDTH-1 downto 0);

begin

--Gray to binary conversion

x <= a xor (’0’ & x(WIDTH-1 downto 1));

y <= b xor (’0’ & y(WIDTH-1 downto 1));

ext <= std\_logic\_vector(unsigned(y) – unsigned(x));

end arc1;

**4-2 priority encoder**

library ieee;

use ieee.std\_logic\_1164.all;

entity enco4 is

port(

r4: in std\_logic\_vector(3 downto 0);

code: out std\_logic\_vector(1 downto 0);

act42: out std\_logic

);

end enco4;

architecture arc1 of enco4 is

begin

code <= ”11” when r4(3)=’1’ else

”10” when r4(2)=’1’else

”01” when r4(1)=’1’else

”00” ;

Act42 <= r4(3) or r4(2) or r4(1) or r4(0);

end arc1;

**16-to-4 tree priority encoder with component instantiation**

--The 4-to-2 encoder is defined right above. This encoder uses comp. instantiation of the 4-to-2 encoder from

--above.

library ieee;

use ieee.std\_logic\_1164.all;

entity enco16 is

port(

r: in std\_logic\_vector(15 downto 0);

code: out std\_logic\_vector(3 downto 0);

active: out std\_logic

);

end enco16;

architecture arc1 of enco16 is

component enco4 is

port(

r4: in std\_logic\_vector(3 downto 0);

code1: out std\_logic\_vector(1 downto 0);

act42: out std\_logic

);

end component;

signal code\_g3, code\_g2, code\_g1, code\_g0, code\_msb:

std\_logic\_vector(1 downto 0);

signal tmp: std\_logic\_vector(3 downto 0);

signal act3, act2, act1, act0: std\_logic;

begin

--four 1st stage 4-to-2 priority encoders

unit\_01: enco4

port map (r4=>r(3 downto 0), code1=>code\_g0,

act42=>act0);

unit\_02: enco4

port map (r4=>r(7 downto 4), code1=>code\_g1,

act42=>act1);

unit\_03: enco4

port map (r4=>r(11 downto 8), code1=>code\_g2,

act42=>act2);

unit\_04: enco4

port map (r4=>r(15 downto 12), code1=>code\_g3,

act42=>act3);

--2nd stage 4-to-2 priority encoder

tmp <= act3 & act2 & act1 & act0;

unit\_1: enco4

port map (r4=>tmp, code1=>code(3 downto 0),

act42=>active);

--The 2 MSBs of the code

Code(3 downto 2) <= code\_msb;

--The 2 LSBs of the code

with code\_msb select

code(1 downto 0) <= code\_g3 when ”11”,

code\_g2 when ”10”,

code\_g3 when ”01”,

code\_g0 when others;

end arc1;

**Sequential Circuit Design**

**Universal shift register-** load, shift left, shift right, rotate left, rotate right operations

library ieee;

use ieee.std\_logic\_1164.all;

entity shift\_register is

port (

clk, reset: in std\_logic;

ctrl: in std\_logic\_vector(1 downto 0);

d: in std\_logic\_vector(3 downto 0);

q: out std\_logic\_vector(3 downto 0)

);

end shift\_register;

architecture arc\_usr of shift\_register is

signal regis: std\_logic\_vector(3 downto 0);

signal nxt: std\_logic\_vector(3 downto 0);

begin

--register

process(clk, reset)

begin

if (reset=’1’) then

regis <= (others=>’0’);

elsif (clk’event and clk=’1’) then

regis <= nxt;

end if;

end process;

--next –state logic

with ctrl select

nxt <=

regis(2 downto 0) & d(0) when ”001”, --shift left

d(3) & regis(3 downto 1) when ”010”, --shift right

d when ”011”, --load

regis(2 downto 0) &regis(3) when ”100”, --rotate left

regis(0) & regis(3 downto 1) when ”101”, --rotate right

regis when others; --pause

--output logic

q <= regis;

end arc\_usr;

**Up and down free running binary counter with parameterized design of generics**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity up\_down\_counter is

generic(WIDTH: natural);

port (

clk, reset: in std\_logic;

mode: in std\_logic;

q: out std\_logic\_vector(WIDTH-1 downto 0);

);

end up\_down\_counter;

architecture arc of up\_down\_counter is

signal regis: std\_logic\_vector(WIDTH-1 downto 0);

signal nxt: std\_logic\_vector(WIDTH-1 downto 0);

begin

--register

process(clk, reset)

begin

if (reset=’1’) then

regis <= (others=>’0’);

elsif (clk’event and clk=’1’) then

regis <= nxt;

end if;

end process;

--next –state logic

nxt <= regis + 1 when mode=’1’ else

regis – 1;

--output logic

q <= std\_logic\_vector(regis);

end arc;

**Pulse width modulation circuit**

--Let k be a 4-bit signal that is interpreted as an unsigned divisor. The frequency of the new output pulse will be

--f/(k\*24) if k is not 0 and f/24 if k is 0. Moreover the w signal controls the duty cycle. The percentage of the period –when the output is active high.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity pwm is

port (

clk, reset: in std\_logic;

w: in std\_logic\_vector(3 downto 0); --signal which controls the duty cycle

k: in std\_logic(3 downto 0); --signal which controls the frequency

pwm\_pulse: out std\_logic

);

end pwm;

architecture arc1 of pwm is

signal regis: unsigned(7 downto 0);

signal nxt: unsigned(7 downto 0);

signal BufReg: std\_logic;

signal BufNxt: std\_logic;

signal z <= unsigned(k)\*16;

begin

--register & output buffer

process(clk,reset)

begin

if (reset=’1’) then

regis <= (others=>’0’);

elsif (clk’event and clk =’1’) then

regis <= nxt;

BufReg <= BufNxt;

end if;

end process;

--next state logic

if (z=’0’) then --case when k = 0 the counting will be up to 16

nxt <= regis + 1 when regis < ”1111” else

’0’;

else nxt <= regis +1 when regis < z else --case when 0<k<16. The counting will be up to k\*24

’0’;

end if;

BufNxt <=

’1’ when (regis < unsigned(w)) or (w =”0000”) else

’0’;

pwm\_pulse <= BufReg;

end arc1;

**Content addressable memory (CAM) circuit optimized with removal of the encoder and decoder parts**

--The input is a 16-bit key or destination field and the data is a 3 bit output port number. The CAM contains four

--key-data pairs similar to a stack with a pointer. There are read and write operations. To read we input a key and --if there is a match with one of the four keys stored the associated data will be routed to the output. The pointer --moves in a round robin fashion. If the CAM has full all its four key-data locations a new write operation will

--overwrite an existing key-data pair.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity key\_file is

port (

clk, reset: in std\_logic;

wr\_en: in std\_logic;

key\_in: in std\_logic\_vector(15 downto 0);

hit: out std\_logic

addr\_out: out std\_logic\_vector(1 downto 0);

);

end key\_file;

architecture arc of key\_file is

constant WORD: natural :=2;

constant BIT: natural :=16;

type reg\_file is array (2\*\*WORD-1 downto 0) of std\_logic\_vector(BIT-1 downto 0);

signal array\_reg: reg\_file;

signal array\_nxt: reg\_file

signal pointer\_reg, pointer\_nxt: unsigned(WORD-1 downto 0);

signal address: std\_logic\_vector(2 downto 0);

signal wr\_key, hit\_flag: std\_logic;

begin

--register

process(clk, reset)

begin

if (reset=’1’) then

array\_reg(3) <= (others=>’0’);

array\_reg(2) <= (others=>’0’);

array\_reg(1) <= (others=>’0’);

array\_reg(0) <= (others=>’0’);

elsif (clk’event and clk=’1’) then

array\_reg(3) <= array\_nxt(3);

array\_reg(2) <= array\_nxt(2);

array\_reg(1) <= array\_nxt(1);

array\_reg(0) <= array\_nxt(0);

end if;

end process;

--logic for register

process(array\_reg, key\_in, pointer\_reg, wr\_key)

begin

array\_nxt (3) <= array\_reg(3);

array\_nxt (2) <= array\_reg(2);

array\_nxt (1) <= array\_reg(1);

array\_nxt (0) <= array\_reg(0);

if (wr\_key=’1’) then

if pointer\_reg =”11” then

array\_nxt(3) <= key\_in;

elsif pointer\_reg =”10” then

array\_nxt(2) <= key\_in;

elsif pointer\_reg =”01” then

array\_nxt(1) <= key\_in;

elsif pointer\_reg =”00” then

array\_nxt(0) <= key\_in;

end if;

end if;

end process;

--replacement pointer

process(clk, reset)

begin

if (reset =’1’) then

pointer\_reg <= (others=>’0’);

elsif (clk’event and clk=’1’) then

pointer <= pointer\_nxt;

end if;

end process;

pointer\_nxt <= pointer\_reg + 1 when wr\_key=’1’ else

pointer\_reg;

--key comparison logic

process(array\_reg, key\_in)

begin

address <= (others=>’1’);

if array\_reg(3) = key\_in then

address <= ”011”;

end if;

if array\_reg(2) = key\_in then

address <= ”010”;

end if;

if array\_reg(1) = key\_in then

address <= ”001”;

end if;

if array\_reg(0) = key\_in then

address <= ”000”;

end if;

end process;

--hit

hit\_flag <= ’1’ when address /=”111”

else ’0’;

--output

hit <= hit\_flag;

addr\_out <= address when (hit\_flag =’1’) else

std\_logic\_vector(pointer\_reg);

end arc;

**4 words stack buffer or FILO circuit**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity FILO is

port (

clk, reset: in std\_logic;

push, pop: in std\_logic;

full, empty: out std\_logic\_vector(3 downto 0);

address: out std\_logic

);

end FILO;

architecture arc of FILO is

constant N: natural := 2;

signal pointer\_reg: unsigned(N downto 0);

signal pointer\_nxt: unsigned(3 downto 0);

signal full\_flag, empty\_flag: std\_logic;

begin

--register

process(clk,reset)

begin

if (reset=’1’) then

pointer\_reg <= (others=>’0’);

elsif (clk’event and clk =’1’) then

pointer\_reg <= pointer\_nxt;

end if;

end process;

--push operation next state logic

pointer\_nxt <=

pointer\_reg + 1 when push =’1’ and full\_flag =’0’ else

pointer\_reg;

full\_flag <=

’1’ when pointer = ”11” else

’0’;

--push port output

address <= std\_logic\_vector(pointer\_reg(N-1 downto 0));

full <= full\_flag;

--read pointer next state logic

pointer\_nxt <= pointer\_reg -1 when pop = ’1’ and empty flag = ’0’ else

pointer\_reg;

empty\_flag <= ’1’ when pointer =”00” else

’0’;

--pop port output

address <= std\_logic\_vector(pointer\_reg(N-1 downto 0));

empty <= empty\_flag;

end arc;

**Finite State Machines-irregular sequential circuits**

**Preample generator circuit for digital communications synchronization optimized with look ahead output buffer for prevention of any glitches and avoiding extra clock –to – output delay**

--The circuit generates the octet pattern ”10101010” for eight clock cycles.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity preample is

port (

clk, reset: in std\_logic;

start: in std\_logic;

data\_out: out std\_logic\_vector(7 downto 0);

);

end preample;

architecture arc of preample is

type state\_type is

(idle, pream1, pream2, pream3, pream4, pream5, pream6, pream7, pream8);

signal state\_reg, state\_nxt: state\_type;

signal buffer\_reg, buffer\_nxt: std\_logic\_vector(7 downto 0);

begin

--state register

process(clk,reset)

begin

if (reset=’1’) then

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

pointer\_reg <= state\_nxt;

end if;

end process;

--output buffer

process(clk,reset)

begin

if (reset=’1’) then

buffer\_reg <= (others =>’0’);

elsif (clk’event and clk =’1’) then

buffer\_reg <= buffer\_nxt;

end if;

end process;

--next-state logic

process(state\_reg, start)

begin

case state\_reg is

when idle =>

if start=’1’ then

state\_nxt <= pream1;

else state\_nxt <= idle;

when pream1 =>

state\_nxt <= pream2;

when pream2 =>

state\_nxt <= pream3;

when pream3 =>

state\_nxt <= pream4;

when pream4 =>

state\_nxt <= pream5;

when pream5 =>

state\_nxt <= pream6;

when pream6 =>

state\_nxt <= pream7;

when pream7 =>

state\_nxt <= pream8;

when pream8 =>

state\_nxt <= idle;

end case;

end process;

--look ahead output logic

process(state\_nxt)

begin

buffer\_nxt <= ”10101010”; --default value

case state\_nxt is

when idle =>

buffer\_nxt <= ”00000000”;

when pream1 =>

state\_nxt <= pream2; --for this and the next seven states buffer\_nxt has the default value.

when pream2 =>

state\_nxt <= pream3;

when pream3 =>

state\_nxt <= pream4;

when pream4 =>

state\_nxt <= pream5;

when pream5 =>

state\_nxt <= pream6;

when pream6 =>

state\_nxt <= pream7;

when pream7 =>

state\_nxt <= pream8;

when pream8 =>

state\_nxt <= idle;

end case;

end process;

--output

data\_out <= buffer\_reg;

end arc;

**Detector of the ”10101010” preample pattern**

--The circuit whenever detects the preample octet sets active high the output signal”match”. It then resets the

--register which holds the last 8 bits of the serial input signal data\_in returning to the initial count state so that to --wait for at least another 8 clock cycles as the register updates with new input bits to detect a new octet of

--”10101010”.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity detec is

port (

clk, reset: in std\_logic;

data\_in: in std\_logic;

match: out std\_logic

);

end detec;

architecture arc of detec is

type state\_type is (count, detection);

signal state\_reg, state\_nxt: state\_type;

signal regis, nxt: std\_logic\_vector(7 downto 09;

--Registers

process(clk,reset)

begin

if (reset=’1’) then

regis <= (others =>’0’);

match <= ’0’;

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

state\_reg <= state\_nxt;

regis <=nxt ;

end if;

end process;

--next-state and output logic

process(regis, state\_reg)

begin

match <= ’0’; --default value of the output signal

case state\_reg is

when count =>

nxt <= data\_in & regis(7 downto 1);

if (regis = ”10101010”) then

state\_nxt <= detection;

else

state\_nxt <= count;

end if;

when detection =>

match <= ’1’; --Set high the output signal match to signal that the octet has been --detected.

regis <= ”00000000” ; --Reset the input register so as to count again for at least 8 clock cycles --until it holds a new octet of ”10101010”.

state\_nxt <= count; --Return to the initial state count.

end case;

end process;

end arc;

**Arbiter circuit employing a timeout signal to prevent the exhaustion of resources from other systems**

--In large digital systems some resources are shared by many subsystems. An arbiter is a circuit that resolves any

--conflict and coordinates the access to the shared resource. When a subsystem needs a resource it activates the

--request signal. The arbiter grants access to the subsystem by activating the corresponding grant signal. Then the --subsystem has permission to access the resources. After the task is completed the subsystem releases the

--resources and deactivates the request signal. In this design the timeout signal when is asserted high releases

--the resources from the subsystem 0 or 1 and returns to the wait state. Access is permitted when the timeout is

--active low and there is a request.

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity arbiter is

port (

clk, reset: in std\_logic;

timeout: in std\_logic;

r: in std\_logic\_vector(1 downto 0); --request signals

g: out std\_logic\_vector(1 downto 0) --grant signals

);

end arbiter;

architecture arc1 of arbiter is

type state\_type is

(wait1, wait0, grant1, grant0);

signal state\_reg, state\_nxt: state\_type;

begin

--state register

process(clk,reset)

begin

if (reset=’1’) then

state\_reg <= wait1;

elsif (clk’event and clk =’1’) then

pointer\_reg <= state\_nxt;

end if;

end process;

--next-state and output logic

process(state\_reg, r, timeout)

begin

g <= ”00”; --default values

case state\_reg is

when wait1 =>

if( r(1) =’1’ and timeout =’0’) then

state\_nxt <= grant1;

elsif (r(0) =’1’ and timeout =’0’) then

state\_nxt <=grant0;

else

state\_nxt <= wait1;

end if;

when wait0 =>

if( r(0) =’1’ and timeout =’0’) then

state\_nxt <= grant0;

elsif (r(1) =’1’ and timeout =’0’) then

state\_nxt <=grant1;

else

state\_nxt <= wait0;

end if;

g(1) <= ’1’;

when grant0 =>

if( r(0) =’1’ and timeout =’0’) then

state\_nxt <= grant0;

else

state\_nxt <= wait1;

end if;

g(0) <= ’1’;

end case;

end process;

end arc1;

**Manchester decoder. The circuit transforms Manchester-coded data back to regular binary data stream.**

--The Manchester encoder produces a 01 sequence (two clock cycles) for a regular binary input of 0 and a 10

--sequence (again two clock cycles) for a 1 bit regular binary input. When the valid signal is asserted the decoder

--accepts valid transitions so if a 01 transition is arriving at the input the decoder produces a 0 at state s0 and

-- then waits for a clock cycle in the wait state. If a 10 is arriving the decoder jumps into the the s1 state producing

--a 1 and waiting for one clock cycle more in the wait state since two bit-clock cycles for the encoder correspond

--to one binary output for the decoder.

use ieee.std\_logic\_1164.all;

entity decod is

port (

clk, reset: in std\_logic;

valid: in std\_logic;

d: in std\_logic\_vector(1 downto 0);

data: out std\_logic

);

end decod;

architecture arc of decod is

type state\_type is

(idle, s0, s1, wait);

signal state\_reg, state\_nxt: state\_type;

begin

--state register

process(clk,reset)

begin

if (reset=’1’) then

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

pointer\_reg <= state\_nxt;

end if;

end process;

--next-state and output logic

process(state\_reg, valid, d)

begin

case state\_reg is

when idle =>

if valid =’0’ then

state\_nxt <= idle;

else

if d=’0’then

state\_nxt <=s0;

else

state\_nxt <= s1;

end if;

end if;

when s0 =>

if valid=’1’ then

state\_nxt <=wait;

else

state\_nxt <= idle;

end if;

end if;

when s1 =>

if valid=’1’ then

state\_nxt <=wait;

else

state\_nxt <= idle;

end if;

end if;

when wait =>

state\_nxt <=idle;

end case;

end process;

--output logic

data <= ’1’ when state\_reg =s1 else

’0’;

end arc;

**NRZI-Non-return to-zero invert-to ones encoder**

--The output of the NRZI encoder is ’0’ if the current input value is different from the previous input value and is a -- ’1’ otherwise. The design employs two registers for storing the two serial input bits and one register for the

--output.

use ieee.std\_logic\_1164.all;

entity nrzi\_enco is

port (

clk, reset: in std\_logic;

d: in std\_logic;

q: in std\_logic\_vector(1 downto 0)

);

end nrzi\_enco;

architecture arc of nrzi is

signal d0\_reg, d1\_reg,d0\_nxt, d1\_nxt: std\_logic;

signal q\_reg, q\_nx: std\_logic;

signal type state\_type is (idle, s0, s1);

signal sta\_re, sta\_nx: state\_type;

--registers

process(clk,reset)

begin

if (reset=’1’) then

sta\_re <= idle;

d0\_reg <= ’0’;

d1\_reg <= ’0’;

q\_reg <= ’0’;

elsif (clk’event and clk =’1’) then

sta\_re <= sta\_nx;

d0\_reg <= d0\_nx;

d1\_reg <= d1\_nx;

q\_reg <= q\_nx;

end if;

end process;

--next-state

process(sta\_re, d0\_reg, d1\_reg)

begin

--Default values

d0\_nx <= d; --The input bit goes to the first register.

d1\_nx <= d0\_reg; --The output bit of the first register is the input for the second.

case sta\_re is

when idle =>

if d1\_reg /= d0\_reg’0’ then

sta\_re <= s0;

else

sta\_re <= s1;

end if;

when s0 =>

if d1\_reg /= d0\_reg then --condition to remain in s0 state

sta\_re <=s0;

else

sta\_re <= s1;

end if;

q\_nx <=0; --The output is zero

when s1 =>

if d1\_reg /= d0\_reg then

sta\_re <=s0;

else

sta\_re <= s1;

end if;

q\_nx <= 1; --The output is one. The last two input bits are equal.

end case;

end process;

--Output

q <= q\_reg; --NRZI coded output

end arc;

**Register Transfer Methodology**

**Repetitive- subtraction division algorithm**

--y is the dividend and d is the divisor. The algorithm obtains the quotient q and the remainder r by subtracting d

--from y repeatedly until y is smaller than d.

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity divition is

port (

y, d: in std\_logic\_vector(7 down to 0);

q\_out, r\_out: out std\_logic\_vector( 7 downto 0)

);

end division;

architecture arc of division is

signal y\_reg, d\_reg: unsigned(7 downto 0);

signal q\_reg, r\_reg: unsigned(7 downto 0);

signal y\_nxt, d\_nxt: unsigned(7 downto 0);

signal q\_nxt, r\_nxt: unsigned(7 downto 0);

type state is

(idle, divide);

signal state\_reg, state\_nxt: state;

--state register

process(clk,reset)

begin

if (reset=’1’) then

y\_reg <= (others=>’0’);

d\_reg <= (others=>’0’);

q\_reg <= (others=>’0’);

r\_reg <= (others=>’0’);

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

y\_reg <= y\_nxt;

d\_reg <= d\_nxt;

q\_reg <= q\_nxt;

r\_reg <= r\_nxt;

state\_reg <= state\_nxt;

end if;

end process;

--next-state logic

process(state\_reg, y\_reg, d\_reg, q\_reg, r\_reg, y, d)

begin

state\_nxt <= state\_reg;

y\_nxt <= y\_reg;

r\_nxt <= r\_reg;

d\_nxt <= d\_reg;

q\_nxt<= q\_reg;

case state\_reg is

q\_nxt <= (others=>’0’);

r\_nxt <= (others=>’0’);

when idle =>

y\_nxt <= unsigned(y);

d\_nxt <= unsigned(d);

state\_nxt <= divide;

when divide =>

if d\_nxt =’0’ then --prevent division by zero

state\_nxt <= idle;

elsif (y\_nxt =’0’) or (y\_nxt > d\_nxt) then

q\_nxt <= ’0’;

r\_nxt <= ’0’;

state\_nxt <= idle;

elsif (y\_nxt > d\_nxt) then

y\_nxt <= y\_nxt – d\_nxt;

q\_nxt <= q\_nxt + 1;

r\_nxt <= y\_nxt;

state\_nxt <= divide;

end if;

end case;

end process;

q <= std\_logic\_vector(q\_reg);

r <= std\_logic\_vector(r\_reg);

end arc;

**Repetitive-addition multiplier** [O(2n) complexity]

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity multiplier is

port (

clk, reset: in std\_logic;

start: in std\_logic;

a\_in, b\_in: in std\_logic\_vector(7 downto 0);

ready: out std\_logic;

r: out std\_logic\_vector(15 downto 0)

);

end multiplier;

architecture arc of multiplier is

constant WIDTH: integer:=8;

type state\_type is (idle, ab0, load, op);

signal state\_reg, state\_nxt: state\_type;

signal a\_reg, a\_nxt: unsigned(WIDTH-1 downto 0);

signal n\_reg, n\_nxt: unsigned(WIDTH-1 downto 0);

signal r\_reg, r\_nxt: unsigned(2\*WIDTH-1 downto 0);

signal adder\_out: unsigned(2\*WIDTH-1 downto 0);

signal sub\_out: unsigned(WIDTH-1 downto 0);

begin

--control path: state register

process(clk,reset)

begin

if (reset=’1’) then

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

state\_reg <= state\_nxt;

end if;

end process;

--control path: combinational logic

process(start, state\_reg, a\_in, b\_in, n\_nxt)

begin

ready <=’0’;

case state\_reg is

when idle =>

if start=’1’ then

if (a\_in=”00000000” or b\_in=”00000000”) then

state\_nxt <= ab0;

else

state\_nxt <= load;

end if;

else

state\_nxt <= idle;

end if;

ready =’1’;

when ab0 =>

state\_nxt <= idle;

when load =>

state\_nxt <= op;

when op =>

if (n\_nxt=”00000000”) then

if start=’1’ then

if (a\_in=”00000000” or b\_in=”00000000”) then

state\_nxt <= ab0;

else

state\_nxt <= load;

end if;

else

state\_nxt <= idle;

ready =’1’;

end if;

else state\_nxt <= op;

end if;

end case;

end process;

--data path: data register

process(clk,reset)

begin

if (reset=’1’) then

a\_reg <= (others=>’0’);

n\_reg <= (others=>’0’);

r\_reg <= (others=>’0’);

elsif (clk’event and clk =’1’) then

a\_reg <= a\_nxt;

n\_reg <= n\_nxt;

r\_reg <= r\_nxt;

end if;

end process;

--data path: combinational circuit

process(state\_reg, a\_reg, n\_reg, r\_reg, a\_in,b\_in)

begin

--default value

a\_nxt <= a\_reg;

n\_nxt <= n\_reg;

r\_nxt <= r\_reg;

case state\_reg is

when idle =>

when ab0 =>

a\_nxt <= unsigned(a\_in);

n\_nxt <= unsigned(b\_in);

r\_nxt <= (others=>’0’);

when load =>

a\_nxt <= unsigned(a\_in);

n\_nxt <= unsigned(b\_in);

r\_nxt <= (others=>’0’);

when op =>

n\_nxt <= n\_reg - 1;

r\_nxt <= (”00000000” & a\_reg) + r\_reg;

end case;

end process;

--data path output

r <= std\_logic\_vector(r\_reg);

end arc;

**Add and shift sequential multiplier** [O(n) operational complexity]

--In this design the add and shift states are merged into one state since the RT operations are independent. Thus

--for each iteration one clock cycle is required. Furthermore when a is added to the partial products, only the

--eight leftmost bits are involved in the operation and the remaining trailing bits are kept unchanged. Instead of

--using a 16-bit adder, one may use a 9-bit adder (8-bits for addition and one for the carry). We can shift the

--partial product to the right one position in each iteration and thus the eight current leftmost bits are always

--connected to the input of the adder. This approach reduces the width of the a register by half and discards the

--shifter for the a operand from the previous design (right above).

-- The circuit adds the upper half of the p register and the a register and then combines the output of the adder

-- with the original half of the p register to form the new partial product. The p register is then shifted right one

-- bit. Since the addition and shifting are merged in one state, there is no register between the adder and shifter

--and thus the two operations are performed in the same clock cycle.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity multiplier is

port (

clk, reset: in std\_logic;

start: in std\_logic;

a\_in, b\_in: in std\_logic\_vector(7 downto 0);

ready: out std\_logic;

r: out std\_logic\_vector(15 downto 0)

);

end multiplier;

architecture arc of multiplier is

constant WIDTH: integer:=8;

constant C\_WIDTH: integer:=4; --width of the counter

constant C\_INIT: unsigned(C\_WIDTH-1 downto 0):=”1000”;

type state\_type is (idle, add\_shft);

signal state\_reg, state\_nxt: state\_type;

signal a\_reg, a\_nxt: unsigned(WIDTH-1 downto 0);

signal n\_reg, n\_nxt: unsigned(C\_WIDTH-1 downto 0);

signal p\_reg, p\_nxt: unsigned(2\*WIDTH downto 0);

--alias for the upper and lower parts of the p\_reg partial products register

alias pu\_nxt: unsigned(WIDTH downto 0) is

p\_nxt(2\*WIDTH downto WIDTH);

alias pu\_reg: unsigned(WIDTH downto 0) is

p\_reg(2\*WIDTH downto WIDTH);

alias pu\_nxt: unsigned(WIDTH-1 downto 0) is

p\_reg(WIDTH-1 downto 0);

signal adder\_out: unsigned(2\*WIDTH-1 downto 0);

signal sub\_out: unsigned(WIDTH-1 downto 0);

begin

-- data and state registers

process(clk,reset)

begin

if (reset=’1’) then

state\_reg <= idle;

a\_reg <= (others=>’0’);

n\_reg <= (others=>’0’);

p\_reg <= (others=>’0’);

elsif (clk’event and clk =’1’) then

state\_reg <= state\_nxt;

a\_reg <= a\_nxt;

n\_reg <= n\_nxt;

r\_reg <= r\_nxt;

end if;

end process;

--next state logic

process(start, state\_reg, a\_reg, n\_reg, p\_reg,

a\_in, b\_in,n\_nxt, p\_nxt)

begin

a\_nxt <= a\_reg;

n\_nxt <= n\_reg;

p\_nxt <= p\_reg;

ready <= ’0’;

case state\_reg is

when idle =>

if start = ’1’ then

p\_nxt <= ”00000000” & unsigned(b\_in);

a\_nxt <= unsigned(a\_in);

n\_nxt <= C\_INIT;

state\_nxt <= add\_shft;

else

state\_nxt <= idle;

end if;

ready <= ’1’;

when add\_shft =>

n\_nxt <= n\_reg -1 ;

--do addition if the multiplier bit is ’1’

If (p\_reg(0) =’1’) then

pu\_nxt <= pu\_reg + (’0’ & a\_reg);

else

pu\_nxt <= pu\_reg;

end if;

--shift

p\_nxt <= ’0’ & pu\_nxt &

pl\_reg(WIDTH-1 downto 1);

if (n\_nxt /= ”0000”) then

state\_nxt <= add\_shft;

else

state\_nxt <= idle;

end if;

end case;

end process;

r <= std\_logic \_vector(p\_reg(2\*WIDTH -1 downto 0);

end arc;

**Leading -zero counting circuit**

--The circuit chekcs one bit of the input at a time and increments accordingly. Counting stops when a ’1’ is

--encountered.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity zeros is

port (

clk, reset: in std\_logic;

d: in std\_logic; --serial input

q: out std\_logic\_vector(7 downto 0); --counting output

);

end zeros;

architecture arc1 of zeros is

type state\_type is (idle, count);

signal state\_reg, state\_nxt: state\_type;

signal n\_reg, n\_nxt: unsigned(7 downto 0);

begin

--register

process(clk,reset)

begin

if (reset=’1’) then

n\_reg <= (others=>’0’);

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

n\_reg <= n\_nxt;

state\_reg <= state\_nxt;

end if;

end process;

--next state logic

process(state\_reg, d\_reg)

begin

--default value

n\_nxt <= n\_reg;

case state\_reg is

when idle =>

if d =’1’ then

state\_reg <= count;

else

state\_reg <= idle;

end if;

when count =>

n\_nxt <= n\_reg + 1;

if d\_reg =’1’ then

state\_reg <= count;

else

state\_reg <= idle;

end if;

end case;

end process;

--output

q <= std\_logic\_vecotr(n\_reg);

end arc1;

**Greatest common divisor**

--The gcd returns the greatest common divisor of two positive integer, a and b. The main algorithm is based on

-- the equation:

-- gcd(a,b) = a if a=b

-- gcd(a-b,b) if a>b

-- gcd(a,b-a) if a<b

--The following simplification rules are employed to optimize the design from O(2n) to O(n) computation time.

-- gcd(a,b) = a if a=b

-- 2gcd(a/2,b/2) if a/= b and a, b are even

-- gcd(a,b/2) if a/= b, a is odd and b is even

-- gcd(a/2,b) if a/= b and a is even while b is odd

-- gcd(a-b,b) if a>b and a, b are odd

-- gcd(a,b-a) if a<b and a, b are odd

--For the computation of 2gcd(a/2,b/2) the factor 2 can be ignored initially. Using a register n we keep track of the

--number of occurences in which both operands are even. The final GCD value can be restored by multiplying the

--initial result by 2n which corresponds to shifting the initial result left n positions.

--In the swap state, the LSBs of the a and b operands are checked. The register is shifted right one position if it is

--even. Then the n register is incremented if both operands are even. If a and b are odd they are compared,

--swapped and subtracted the one from the other. An extra state res is added to restore the final GCD value.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity gcd is

port (

clk, reset: in std\_logic;

start: in std\_logic;

a\_in, b\_in: in std\_logic\_vector(7 downto 0);

ready: out std\_logic;

r: out std\_logic\_vector(7 downto 0)

);

end gcd;

architecture arc1 of gcd is

type state\_type is (idle, swap, res);

signal state\_reg, state\_nxt: state\_type;

signal a\_reg, a\_nxt, b\_reg, b\_nxt: unsigned(7 downto 0);

signal n\_reg, n\_nxt: unsigned(2 downto 0);

begin

--state & data registers

process(clk, reset)

begin

if (reset=’1’) then

state\_reg <= idle;

a\_reg <= (others=>’0’);

b\_reg <= (others=>’0’);

n\_reg <= (others=>’0’);

elsif (clk’event and clk =’1’) then

state\_reg <= state\_nxt;

a\_reg <= a\_nxt;

b\_reg <= n\_nxt;

n\_reg <= n\_nxt;

end if;

end process;

--next state logic, data path functional units

process(start, state\_reg, a\_reg, n\_reg, b\_reg,

a\_in, b\_in,n\_nxt)

begin

a\_nxt <= a\_reg;

b\_nxt <= b\_reg;

n\_nxt <= n\_reg;

case state\_reg is

when idle =>

if start = ’1’ then

a\_nxt <= unsigned(a\_in);

b\_nxt <= unsigned(b\_in);

n\_nxt <= (others => ’0’);

state\_nxt <= swap;

else

state\_nxt <= idle;

end if;

when swap =>

if a\_reg = b\_reg then

if n\_reg = 0 then

state\_nxt <= idle;

else

state\_nxt <= res;

end if;

else

if (a\_reg(0) = ’0’) then –condition that checks if a is even

a\_nxt <= ’0’& a\_reg(7 downto 1);

if b\_reg(0)=’0’ then -- if both a and b are even

b\_nxt <= ’0’& b\_reg(7 downto 1);

n\_nxt <= n\_reg + 1;

end if;

state\_nxt <= swap;

else –a\_reg is odd

if b\_reg(0)=’0’ then -- b\_reg is even

b\_nxt <= ’0’& b\_reg(7 downto 1);

state\_nxt <= swap;

else -- again if both operands are even

if a\_reg < b\_reg then

a\_nxt <= b\_reg;

b\_nxt <= a\_reg;

end if;

a\_nxt <= a\_reg – b\_reg;

state\_nxt <= swap;

end if;

end if;

end if;

when res =>

a\_nxt <= a\_reg(6 downto 0) & ’0’;

n\_nxt <= n\_reg – 1;

if n\_nxt = 0 then

state\_nxt <= idle;

else

state\_nxt <= res;

end if;

end case;

end process;

--output

ready <= ’1’ when state\_reg = idle else ’0’;

r <= std\_logic\_vector(a\_reg);

end arc1;

**Universal Asynchronous Receiver Transmitter (UART)**

--The universal asynchronous receiver and transmitter is a system that sends bytes of data through a serial line.

–When it is idle the serial line is ’1’. The transmission starts with a start bit a ’0’, followed by eight data bits and

-- ended with a stop bit which is ’1’. The receiver uses oversampling to ensure the data bits are recovered

--properly. This design incorporates selection among four baud rates with the baud\_sel signal. Also the design

-- with the parity\_sel signal supports odd parity, even parity or no parity by activating the respective output

--signals as odd\_parity and even\_parity. The incoming stream is recovered as follows.

-- When the incoming line becomes ’0’ the sampling pulse counter starts counting.

--When the counter reaches 7, clear it to 0 and restart. At this point, the incoming signal reaches about a half of

--the start bit.

--When the counter reaches 15, clear it to 0 and restart. At this point, the incoming signal progresses for one bit

--and reaches the middle of the first data bit. The data in the serial line should be retrieved and shifted into a

--register.

--Repeat step 3 seven times to retrieve the remaining seven data bits.

--Repeat step 3 one more time but without shifting. The incoming signal should reach the middle of the stop bit at

--this point, and its value should be ’1’.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity uart\_receiver is

port(

clk, reset: in std\_logic;

rx: in std\_logic;

baud\_sel in std\_logic\_vector( 1downto 0);

parity\_sel in std\_logic\_vector(1 downto 0);

ready: out std\_logic;

odd\_parity out std\_logic;

even\_parity out std\_logic;

pout: out std\_logic\_vector(7 downto 0)

);

end uart\_receiver;

architecture arc of uart\_receiver is

type state\_type is (idle, start, data, stop);

signal state\_reg, state\_nxt: state\_type;

signal clk16\_reg, clk16\_nxt: unsigned(5 downto 0);

signal s\_reg, s\_nxt: unsigned(3 downto 0);

signal n\_reg, n\_nxt: unsigned(2 downto 0);

signal b\_reg, b\_nxt: std\_logic\_vector(7 downto 0);

signal s\_pulse: std\_logic;

signal parity\_sum: unsigned(2 downto 0);

variable DVSR: integer;

begin

--

process(clk, reset)

begin

if reset =’1’ then

clk16\_reg <= (others=>’0’);

elsif clk’event and clk=’1’ then

clk16\_reg <= clk16\_nxt;

end if;

end process;

--next state logic

--baud rate adjustment. Baud rates: 1200, 2400, 4800, 9600

process(baud\_sel)

begin

with baud\_sel select

DVSR := 52 when ”00”,

26 when ”01”,

13 when ”10”,

7 when ”11”;

clk16\_nxt <= (others =>’0’) when clk16\_reg=(DVSR-1) else

clk16\_reg + 1;

s\_pulse <= ’1’ when clk16\_reg= 0 else ’0’;

end process;

--FSMD state & data registers

process(clk, reset)

begin

if reset = ’1’ then

state\_reg <= idle;

s\_reg <= (others=>’0’);

n\_reg <= (others=>’0’);

b\_reg <= (others=>’0’);

elsif (clk’event and clk =’1’) then

state\_reg <= state\_nxt;

s\_reg <= s\_nxt;

n\_reg <= n\_nxt;

b\_reg <= b\_nxt;

end if ;

end process;

--next state logic & data path functional units

process(state\_reg, s\_reg, n\_reg, b\_reg, s\_pulse, rx)

begin

s\_nxt <= s\_reg;

n\_nxt <= n\_reg;

b\_nxt <= b\_reg;

ready <= ’0’;

case state\_reg is

when idle =>

if rx=’0’ then

state\_nxt <= start;

else

state\_nxt <= idle;

end if;

ready <= ’1’;

when start =>

if s\_pulse = ’0’ then

state\_nxt <= start;

else

if s\_reg = 7 then

state\_nxt <= data;

s\_nxt <= (others = >’0’);

else

state\_nxt <= start;

s\_nxt <= s\_reg + 1;

end if;

end if;

when data =>

if s\_pulse= ’0’ then

state\_nxt <= data;

else

if s\_reg = 15 then

s\_nxt <= others=>’0’;

b\_nxt <= rx & b\_reg(7 downto 1);

if n\_reg= 7 then

state\_nxt <= stop;

n\_nxt <= others=>’0’;

else

state\_nxt <= data;

n\_nxt <= n\_reg + 1;

end if;

end if;

when stop =>

if s\_pulse = ’0’ then

state\_nxt <= stop;

else

if s\_reg = 15 then

state\_nxt <= idle;

state\_nxt <= (other=>’0’);

else

state\_nxt <= stop;

s\_nxt <= s\_reg + 1;

end if;

end if;

end case;

end process;

process(parity\_sel, parity\_sum)

parity\_sum <= ’0’;

for k in 1 to 8 loop

parity\_sum <= parity\_sum + b\_reg(0);

b\_reg <= ’0’ & b\_reg( 7 downto 1);

end loop;

if parity\_sel = ”10” then --Let parity\_sel be ”10” for selecting odd parity.

if parity\_sum(0) = ’1’ then -- Then test for odd parity.

odd\_parity <= ’1’;

end if;

elsif parity\_sel = ”01” then --Let parity\_sel be ”01” for selecting even parity.

if parity\_sum(0) = ’0’ then --Then test the condition for even parity.

even\_parity <= ’1’;

end if;

else

odd\_parity <= ’0’;

even\_parity <= ’0’;

end if;

end process;

pout <= b\_reg;

end arc;

**HIERARCHICAL DESIGN**

**Decade counter** [mod-10]

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity dec\_counter is

port (

clk, reset: in std\_logic;

en: in std\_logic;

q: out std\_logic\_vector(3 downto 0);

pulse: out std\_logic

);

end dec\_counter;

architecture arc1 of dec\_counter is

signal regis: unsigned(3 downto 0);

signal nxt: unsigned(3 downto 0);

constant TEN: integer:= 10;

begin

--register

process(clk,reset)

begin

if (reset=’1’) then

regis <= (others=>’0’);

elsif (clk’event and clk =’1’) then

regis <= nxt;

end if;

end process;

--next state logic

process(en, regis)

begin

nxt <= regis;

if (en=’1’) then

if regis=(TEN-1) then

nxt <= (others =>’0’);

else

nxt <= regis +1;

end if;

end if;

end process;

--output logic

q <= std\_logic\_vector(regis);

pulse <= ’1’ when regis =(TEN-1’) else

’0’;

end arc1;

**Parameterized mod-n counter using component instantiation of the decade counter (right above) and generics**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity thousand\_counter is

port (

clk, reset: in std\_logic;

en: in std\_logic;

q\_ten, q\_ one, q\_hundred: out std\_logic\_vector(3 downto 0);

p1000: out std\_logic

);

end thousand\_counter;

architecture arc1 of thousand\_counter is

component dec\_counter -- Component declaration

generic ( --Generic declaration

N: natural;

WIDTH: natural

);

port(

clk, reset: in std\_logic;

en: in std\_logic;

q : out std\_logic\_vector(WIDTH-1 downto 0);

pulse: out std\_logic

);

end component;

signal p\_one, p\_ten, p\_hundred: std\_logic;

begin --Components instantiation

ones: dec\_counter -- Counter for the ones

generic map (N=>10, WIDTH=>4)

port map (clk=>clk, reset=>reset, en=>en,

pulse=>p\_one, q=>q\_one);

tens: dec\_counter -- Counter for the tens

generic map (N=>10, WIDTH=>4)

port map (clk=>clk, reset=>reset, en=>p\_one,

pulse=>p\_ten, q=>q\_ten);

hundreds: dec\_counter --Counter for the hundreds

generic map (N=>10, WIDTH=>4)

port map (clk=>clk, reset=>reset, en=>p\_ten,

pulse=>p\_hundred, q=>q\_hundred);

p1000 <= p\_one and p\_ten and p\_hundred;

end arc1;

**Counter that counts from m to n and then wraps around by the use of generics.**

--The circuit counts from M to N and then wraps around and starts counting again from M.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity m\_n\_counter is

generic(

M: natural;

N: natural

);

port (

clk, reset: in std\_logic;

en: in std\_logic;

q: out std\_logic\_vector(3 downto 0)

);

end dec\_counter;

architecture arc1 of m\_n\_counter is

signal regis: unsigned(3 downto 0);

signal nxt: unsigned(3 downto 0);

begin

--register

process(clk,reset)

begin

if (reset=’1’) then

regis <= M;

elsif (clk’event and clk =’1’) then

regis <= nxt;

end if;

end process;

--next state logic

process(en, regis)

begin

nxt <= regis;

if (en=’1’) then

if regis=N-1 then

nxt <= M;

else

nxt <= regis +1;

end if;

end if;

end process;

--output logic

q <= std\_logic\_vector(regis);

end arc1;

**Subprogram-function f(n) =2n**

--This function just implements the formula 2n.

function power2(n: integer) return integer is

variable result: integer;

begin

result: = 2;

if n=’0’ then

result = 1;

elsif n <’0’ then

for k in 1 to -n loop

result := result\*2;

end loop;

result := 1/result;

else

for k in 1 to n loop

result := result\*2;

end loop;

end if;

return result;

end power2;

**Parameterized Design**

**1-bit incrementor**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_arith.all;

use ieee.std\_logic\_unsigned.all;

entity inc\_1\_bit is

generic(

M: natural;

N: natural

);

port (

a,cin: in std\_logic;

s, cout: out std\_logic

);

end inc\_1\_bit;

architecture arc1 of inc\_1\_bit is

begin

process(a, cin)

begin

s <= a xor cin;

cout <= a and cin;

end process;

end arc1;

**Parameterized N-bit incrementor by the use of generic and component instantiation of a 1-bit incrementor cell inside a for generate statement**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_arith.all;

use ieee.std\_logic\_unsigned.all;

entity incre is

generic( WIDTH: natural);

port (

a,cin: in std\_logic;

s, cout: out std\_logic

);

end inc\_1\_bit;

architecture arc1 of incre is

component inc\_1\_bit

port(

a,cin: in std\_logic;

s, cout: out std\_logic

) ;

end component;

signal a\_WID, cin\_WID: std\_logic\_vector(WIDTH-1 downto 0);

signal s\_WID, cout\_WID: std\_logic\_vector(WIDTH-1 downto 0);

begin

inc\_1\_bit\_gen:

for i in (WIDTH-1) downto 0 generate

incrementors: inc\_1\_bit

port map(a=>a\_WID(i) , cin =>cin\_WID(i) , s=>s\_WID(i) , cout =>cout\_WID(i) );

end generate;

end arc1;

**Parameterized memory controller-FSM. BUF generic parameter chooses to include the look-ahead buffer or exclude it for the output signal with the use of conditional generate statement**

--Memory controller specification

--The controller is between a processor and a memory chip, interpreting commands from the processor and then

--generating control sequence accordingly. mem, burst and rw are input signals from the processor. Initially the

--FSM is in the idle state and waits for the mem signal from the processor. When mem is asserted then the FSM

--examines the rw signal. If that is one it jumps to the read1 state else it jumps into the write state. If the FSM is in

--the read1 state it examines the burst signal which is used for special read operation. If burst is 1 the FSM will go

--through read2, read3 and read4 in the next clock cycles and it will return on the idle state. Alternatively if burst

--is 0 it will return from the read1 to the idle state.

--The BUF generic parameter is used to implement an additional look-ahead output buffer if BUF is 1.Or not

--implement the buffer if BUF is 0. The purpose of the look-ahead buffer is to remove any glitches at the output

--and avoid additional clock delays due to the inserted register by clever routing of the output signals.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity mem\_controller is

generic( BUF: natural);

port(

clk, reset: in std\_logic;

mem, rw, burst: in std\_logic;

oe, we, we\_me: out std\_logic

);

end mem\_controller;

architecture arc1 of mem\_controller is

type state\_type is

(idle, read1, read2, read3, read4, write);

signal oe\_nxt, we\_nxt, oe\_buf, we\_buf, state\_reg, state\_nxt: state\_type;

begin

--state register

begin

process(clk, reset)

if (reset=’1’) then

state\_reg <= idle;

elsif (clk’event and clk =’1’) then

state\_reg <= state\_nxt;

end if;

end process;

--Output buffer in case BUF is 1

Out\_Buf\_gen:

If BUF =1 generate

process(clk, reset)

begin

if( reset =’1’) then

oe\_buf <= ’0’;

we\_buf <= ’0’;

elsif (clk’event and clk =’1’) then

oe\_buf <= oe\_nxt;

we\_buf <= we\_nxt;

end if;

end process;

end generate;

--next state logic

process(state\_reg, mem, rw, busrt)

begin

case state\_reg is

when idle =>

if mem=’1’ then

if (rw =’1’) then

state\_nxt <= read1;

else

state\_nxt <= write;

end if;

else

state\_nxt <= idle;

end if;

when write =>

state\_nxt <= idle;

when read1 =>

if (burst =’1’) then

state\_nxt <= read2;

else

state\_nxt <= idle;

end if;

when read2 =>

state\_nxt <= read3;

when read3 =>

state\_nxt <= read4;

when read4 =>

state\_nxt <= idle;

when others =>

state\_nxt <= idle;

end case;

end process;

--Moore output logic in case BUF is 0

No\_Look\_Ahead:

If BUF /=1 generate

process(state\_reg) --For the output without the look-ahead buffer controller the process of the

begin -- Moore output logic has **state\_reg** as argument

we <= ’0’ ;

oe <= ’0’;

case state\_reg is

when idle =>

when write =>

we <= ’1’;

when read1 =>

oe <= ’1’;

when read2 =>

oe <= ’1’;

when read3 =>

oe <= ’1’;

when read4 =>

oe <= ’1’;

when others =>

end case;

end process;

--Mealy output logic

we\_me <= ’1’ when ((state\_reg=idle) and (mem=’1’) and

(rw=’0’)) else

’0’;

end generate;

--look-ahead output logic if BUF parameter is 1

Look\_Ahead\_Gen:

If BUF=1 generate

process(state\_nxt) --For the output with the look-ahead buffer controller the process of the

begin -- look-ahead output logic has **state\_nxt** as argument

we\_nxt <= ’0’ ; --default values

oe\_nxt <= ’0’;

case state\_nxt is

when idle =>

when write =>

we\_nxt <= ’1’;

when read1 =>

oe\_nxt <= ’1’;

when read2 =>

oe\_nxt <= ’1’;

when read3 =>

oe\_nxt <= ’1’;

when read4 =>

oe\_nxt <= ’1’;

end case;

end process;

--output

we <= we\_buf;

oe <= oe\_buf;

end generate;

end arc1;