Dally, Harting and Aamodt: Digital Design Using VHDL

Kapitel 8

8.1 Vi tecknar sanningstabellen, *Figur 8.1a* och skriver VHDL-kod

x_2	x_1	x_0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Figur 8.1a

```
-----------------
-- ex8_1.vhdl --
-----------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ex8_1 IS
   PORT (x:IN STD_LOGIC_VECTOR(2 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END ex8_1;

ARCHITECTURE arch_ex8_1 OF ex8_1 IS
BEGIN
   y(0)<=NOT(x(2)) AND NOT(x(1)) AND NOT(x(0));
   y(1)<=NOT(x(2)) AND NOT(x(1)) AND x(0);
   y(2)<=NOT(x(2)) AND x(1) AND NOT(x(0));
   y(3)<=NOT(x(2)) AND x(1) AND x(0);
   y(4)<=x(2) AND NOT(x(1)) AND NOT(x(0));
   y(5)<=x(2) AND NOT(x(1)) AND x(0);
   y(6)<=x(2) AND x(1) AND NOT(x(0));
   y(7)<=x(2) AND x(1) AND x(0);
END arch_ex8_1;
```

8.1 forts.

som vi simulerar med en do-fil

```
---------------
-- ex8_1.do --
---------------

restart -f -nowave
view signals wave
add wave x -radix unsigned x
add wave -binary y
force x(0) 0 0ns, 1 50ns -repeat 100ns
force x(1) 0 0ns, 1 100ns -repeat 200ns
force x(2) 0 0ns, 1 200ns -repeat 400ns
run 400ns
```

8.4 Vi skall använda en 2->4 dekoder och en 3->8 dekoder för att bygga en 5->32 dekoder. Vi kan använda 2->4 dekodern för att dela in de 32 utgångarna i fyra block medan vi använder 3->8 dekodern för att välja utgång inom blocket.
Vi börjar med att skapa en generisk dekoder

```
------------------
-- decoder.vhdl --
------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY decoder IS
   GENERIC(n_in:NATURAL:=2);
   PORT (x:IN STD_LOGIC_VECTOR(n_in-1 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(2**n_in-1 DOWNTO 0));
END decoder;

ARCHITECTURE arch_decoder OF decoder IS
   SIGNAL one:UNSIGNED(2**n_in-1 DOWNTO 0);
BEGIN
   one <= TO_UNSIGNED(1,2**n_in);
   y <=
STD_LOGIC_VECTOR(SHIFT_LEFT(one,TO_INTEGER(UNSIGNED(x))));
END arch_decoder;
```

8.4 forts.

Notera hur antalet bitar in (n_in) används för att ange antalet bitar ut. Vi simulerar dekodern

```
-------------
-- decoder --
-------------

restart -f -nowave
view signals wave
add wave x y
add wave -binary x y
force x(0) 0 0ns, 1 50ns -repeat 100ns
force x(1) 0 0ns, 1 100ns -repeat 200ns
run 200ns
```

och vi sedan använder dekodern som komponent i lösningen av exemplet

```
-----------------
-- ex8_4.vhdl --
-----------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ex8_4 IS
   PORT (x:IN STD_LOGIC_VECTOR(4 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END ex8_4;

ARCHITECTURE arch_ex8_4 OF ex8_4 IS
   COMPONENT decoder IS
      GENERIC(n_in:NATURAL:=2);
      PORT (x:IN STD_LOGIC_VECTOR(n_in-1 DOWNTO 0);
            y:OUT STD_LOGIC_VECTOR(2**n_in-1 DOWNTO 0));
   END COMPONENT decoder;
   SIGNAL bit_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL block_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   decoder3_8:
   COMPONENT decoder
      GENERIC MAP(n_in => 3)
      PORT MAP(x=>x(2 DOWNTO 0),
               y=>bit_signal);
```

8.4 forts.

```
   decoder2_4:
      COMPONENT decoder
         GENERIC MAP(n_in => 2)
         PORT MAP(x=>x(4 DOWNTO 3),
                  y=>block_signal);
   y(31 DOWNTO 24)<=bit_SIGNAL AND (7 DOWNTO 0 =>
                                          block_signal(3));
   y(23 DOWNTO 16)<=bit_SIGNAL AND (7 DOWNTO 0 =>
                                          block_signal(2));
   y(15 DOWNTO 8)<=bit_SIGNAL AND (7 DOWNTO 0 =>
                                          block_signal(1));
   y(7 DOWNTO 0)<=bit_SIGNAL AND (7 DOWNTO 0 =>
                                          block_signal(0));
END arch_ex8_4;
```

Notera hur vi via GENERIC-argumentet kan använda samma komponent för att instantiera två dekodrar av olika storlek.
Vi kan lika gärna kunnat dela in de 32 utgångarna i åtta block med 3->8 dekodern och sedan använda 2->4 dekodern för att välja utgång inom blocket

```
-------------------
-- ex8_4_v2.vhdl --
-------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ex8_4_v2 IS
   PORT (x:IN STD_LOGIC_VECTOR(4 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END ex8_4_v2;

ARCHITECTURE arch_ex8_4_v2 OF ex8_4_v2 IS
   COMPONENT decoder IS
      GENERIC(n_in:NATURAL:=2);
      PORT (x:IN STD_LOGIC_VECTOR(n_in-1 DOWNTO 0);
            y:OUT STD_LOGIC_VECTOR(2**n_in-1 DOWNTO 0));
   END COMPONENT decoder;
   SIGNAL bit_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL block_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
```

8.4 forts.

```
BEGIN
   decoder3_8:
   COMPONENT decoder
      GENERIC MAP(n_in => 2)
      PORT MAP(x=>x(1 DOWNTO 0),
               y=>bit_signal);
   decoder2_4:
      COMPONENT decoder
         GENERIC MAP(n_in => 3)
         PORT MAP(x=>x(4 DOWNTO 2),
                  y=>block_signal);
   y(31 DOWNTO 28)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(7));
   y(27 DOWNTO 24)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(6));
   y(23 DOWNTO 20)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(5));
   y(19 DOWNTO 16)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(4));
   y(15 DOWNTO 12)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(3));
   y(11 DOWNTO 8)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(2));
   y(7 DOWNTO 4)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(1));
   y(3 DOWNTO 0)<=bit_SIGNAL AND (3 DOWNTO 0 =>
                                       block_signal(0));
END arch_ex8_4_v2;
```

Vi kan simulera med samma do-fil som I den första lösningen.

8.5 Vi skall två stycken 3->8 dekodrar för att bygga en 6->64 dekoder. Vi kan använda samma modell som i *Exempel 8.4* och använda 3->8 dekodern för att dela in de 64 utgångarna i fyra block medan vi använder den andra 3->8 dekodern för att välja utgång inom blocket

8.5 forts.

```
----------------
-- ex8_5.vhdl --
----------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;


ENTITY ex8_5 IS
   PORT (x:IN STD_LOGIC_VECTOR(5 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(63 DOWNTO 0));
END ex8_5;

ARCHITECTURE arch_ex8_5 OF ex8_5 IS
   COMPONENT decoder IS
      GENERIC(n_in:NATURAL:=2);
      PORT (x:IN STD_LOGIC_VECTOR(n_in-1 DOWNTO 0);
            y:OUT STD_LOGIC_VECTOR(2**n_in-1 DOWNTO 0));
   END COMPONENT decoder;
   SIGNAL bit_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL block_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   decoder3_8_0:
   COMPONENT decoder
      GENERIC MAP(n_in => 3)
      PORT MAP(x=>x(2 DOWNTO 0),
               y=>bit_signal);
   decoder3_8_1:
      COMPONENT decoder
         GENERIC MAP(n_in => 3)
         PORT MAP(x=>x(5 DOWNTO 3),
               y=>block_signal);
   y(63 DOWNTO 56) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(7));
   y(55 DOWNTO 48) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(6));
   y(47 DOWNTO 40) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(5));
   y(39 DOWNTO 32) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(4));
   y(31 DOWNTO 24) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(3));
```

8.5 forts.

```
   y(23 DOWNTO 16) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(2));
   y(15 DOWNTO 8) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(1));
   y(7 DOWNTO 0) <= bit_SIGNAL AND (7 DOWNTO 0 =>
block_signal(0));

END arch_ex8_5;
```

Vi simulerar med en do-fil som är likadan som i *Exempel 8.4* bortsett från att insignalen har en bit till och vi får köra dubbelt så lång simuleringstid

```
--------------
-- ex8_5.do --
--------------

restart -f -nowave
view signals wave
add wave x -radix unsigned x
add wave -binary y
force x(0) 0 0ns, 1 50ns -repeat 100ns
force x(1) 0 0ns, 1 100ns -repeat 200ns
force x(2) 0 0ns, 1 200ns -repeat 400ns
force x(3) 0 0ns, 1 400ns -repeat 800ns
force x(4) 0 0ns, 1 800ns -repeat 1600ns
force x(5) 0 0ns, 1 800ns -repeat 3200ns
run 3200ns
```

8.6 Vi skall lösa samma uppgift som i *Exempel 8.5* men med hjälp av 2->4 dekodrar. Det kräver att vi delar upp utsignalerna i fyra semiblock om fyra utgångar som vi i sin tur delar upp i fyra block. Vi får

```
----------------
-- ex8_6.vhdl --
----------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ex8_6 IS
   PORT (x:IN STD_LOGIC_VECTOR(5 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(63 DOWNTO 0));
END ex8_6;
```

8.6 forts.

```
ARCHITECTURE arch_ex8_6 OF ex8_6 IS
   COMPONENT decoder IS
      GENERIC(n_in:NATURAL:=2);
      PORT (x:IN STD_LOGIC_VECTOR(n_in-1 DOWNTO 0);
            y:OUT STD_LOGIC_VECTOR(2**n_in-1 DOWNTO 0));
   END COMPONENT decoder;
   SIGNAL bit_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL semi_block_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL block_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   decoder2_4_0:
   COMPONENT decoder
      GENERIC MAP(n_in => 2)
      PORT MAP(x=>x(1 DOWNTO 0),
               y=>bit_signal);
   decoder2_4_1:
      COMPONENT decoder
         GENERIC MAP(n_in => 2)
         PORT MAP(x=>x(3 DOWNTO 2),
                  y=>semi_block_signal);
   decoder2_4_2:
      COMPONENT decoder
         GENERIC MAP(n_in => 2)
         PORT MAP(x=>x(5 DOWNTO 4),
                  y=>block_signal);

   y(63 DOWNTO 60)<=bit_SIGNAL AND
                    (3 DOWNTO 0=>semi_block_signal(3)) AND
                    (3 DOWNTO 0=>block_signal(3));
   y(59 DOWNTO 56)<=bit_SIGNAL AND
                    (3 DOWNTO 0=>semi_block_signal(2)) AND
                    (3 DOWNTO 0=>block_signal(3));
   y(55 DOWNTO 52)<=bit_SIGNAL AND
                    (3 DOWNTO 0=>semi_block_signal(1)) AND
                    (3 DOWNTO 0=>block_signal(3));
   y(51 DOWNTO 48)<=bit_SIGNAL AND
                    (3 DOWNTO 0=>semi_block_signal(0)) AND
                     3 DOWNTO 0=>block_signal(3));
   y(47 DOWNTO 44)<=bit_SIGNAL AND
                    (3 DOWNTO 0=>semi_block_signal(2)) AND
                    (3 DOWNTO 0=>block_signal(3));
```

8.6 forts.

```
    y(43 DOWNTO 40)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(2)) AND
                     (3 DOWNTO 0=>block_signal(2));
    y(39 DOWNTO 36)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(2)) AND
                     (3 DOWNTO 0=>block_signal(1));
    y(35 DOWNTO 32)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(2)) AND
                     (3 DOWNTO 0=>block_signal(0));
    y(31 DOWNTO 28)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(1)) AND
                     (3 DOWNTO 0=>block_signal(3));

    y(27 DOWNTO 24)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(1)) AND
                     (3 DOWNTO 0=>block_signal(2));
    y(23 DOWNTO 20)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(1)) AND
                     (3 DOWNTO 0=>block_signal(1));
    y(19 DOWNTO 16)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(0)) AND
                       (3 DOWNTO 0=>block_signal(1));
    y(15 DOWNTO 12)<=bit_SIGNAL AND
                     (3 DOWNTO 0=>semi_block_signal(3)) AND
                       (3 DOWNTO 0=>lock_signal(0));
    y(11 DOWNTO 8)<=bit_SIGNAL AND
                    (3 DOWNTO 0=>semi_block_signal(2)) AND
                    (3 DOWNTO 0=>block_signal(0));
    y(7 DOWNTO 4)<=bit_SIGNAL AND
                   (3 DOWNTO 0=>semi_block_signal(1)) AND
                   (3 DOWNTO 0=>block_signal(0));
    y(3 DOWNTO 0)<=bit_SIGNAL AND
                   (3 DOWNTO 0=>semi_block_signal(0)) AND
                   (3 DOWNTO 0=>block_signal(0));
END arch_ex8_6;
```

Som vi kan simulera med samma do-fil som i *Exempel 8.5*

8.21 Vi har sanningstabellen i *Figur 8.21*. Vi har fyra inbitar som leder till ett minne med 16 adresser och på varje minnesposition lagrar vi en bit som indikerar om det aktuella invärdet är ett primtal.
Vi skriver VHDL-kod

Address	*y*
0000	0
0001	1
0010	1
0011	1
0100	0
0101	1
0110	0
0111	1
1000	0
1001	0
1010	0
1011	1
1100	0
1101	1
1110	0
1111	0

Figur 8.21 Primtal

```
-----------------
-- ex8_21.vhdl --
-----------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ex8_21 IS
   PORT (address:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         y:OUT STD_LOGIC);
END ex8_21;

ARCHITECTURE arch_ex8_21 OF ex8_21 IS
   TYPE ROM_TABLE IS ARRAY (0 to 15) OF STD_LOGIC;
   CONSTANT ROM: ROM_TABLE :=
             ROM_TABLE('0','1','1','1','0','1','0',’1’,
                       '0','0','0','1','0','1','0','0');
BEGIN
   y<= ROM(TO_INTEGER(UNSIGNED(address)));
END arch_ex8_21;
```

Vi simulerar med en do-fil

```
-- ex8_21.do --
---------------

restart -f -nowave
view signals wave
add wave address -radix unsigned address
add wave y
force address(0) 0 0ns, 1 50ns -repeat 100ns
force address(1) 0 0ns, 1 100ns -repeat 200ns
```

8.21 forts.

```
force address(2) 0 0ns, 1 200ns -repeat 400ns
force address(3) 0 0ns, 1 400ns -repeat 800ns
run 800ns
```

8.22 Vi har en hex-siffra i fyra inbitar och sju segmentsignaler ut. Vi har sanningstabellen i *Figur 8.22*. Vi behöver ett minne med 16 adresser och på varje minnesposition lagrar vi sju bitar för segmenten. Vi skriver VHDL-kod

Input				Segment						
x_3	x_2	x_1	x_0	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	1	0	0	1	1	1

Figur 8.12 Sanningstabell för 7-segmentdisplay

```
----------------
-- ex8_22.vhdl --
----------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ex8_22 IS
   PORT (hex_digit:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         segments:OUT STD_LOGIC_VECTOR(0 TO 6));
END ex8_22;
```

8.22 forts.

```
ARCHITECTURE arch_ex8_22 OF ex8_22 IS
   TYPE ROM_TABLE IS ARRAY (0 to 15) OF
                                STD_LOGIC_VECTOR(0 TO 6);
   CONSTANT ROM: ROM_TABLE := ROM_TABLE'("1111110",
                                         "0110000",
                                         "1101101",
                                         "1111001",
                                         "0110011",
                                         "1011011",
                                         "1011111",
                                         "1110000",
                                         "1111111",
                                         "1110011",
                                         "1110111",
                                         "0011111",
                                         "1001110",
                                         "0111101",
                                         "1001111",
                                         "1100111");
BEGIN
   segments<= ROM(TO_INTEGER(UNSIGNED(hex_digit)));
END arch_ex8_22;
```

Och simulerar med en do-fil

```
---------------
-- ex8_22.do --
---------------

restart -f -nowave
view signals wave
add wave hex_digit
add wave segments -radix binary segments
force hex_digit(0) 0 0ns, 1 50ns -repeat 100ns
force hex_digit(1) 0 0ns, 1 100ns -repeat 200ns
force hex_digit(2) 0 0ns, 1 200ns -repeat 400ns
force hex_digit(3) 0 0ns, 1 400ns -repeat 800ns
run 800ns
```