

Lista 3 (VHDL) - rozwiązania

Błażej Wróbel, 250070, W11, 3. rok, informatyka

Pytanie na boku nr 1:

```
y <= a when s = '0' else b when s = '1' else 'X';
```

Powyższy kod przypisuje do y wartość:

- a gdy wartość s to '0'
- b gdy wartość s to '1'
- 'X' w pozostałych przypadkach

Pytanie na boku nr 2:

```
with s select y <= a when "00" b when "01" c when "10" "XXX" when others;
```

Powyższy kod przypisuje do y odpowiednią wartość (zdeteminowaną wartością sygnału s). Zatem:

- Jeśli s = "00", to pod y podstawia a
- Jeśli s = "01", to pod y podstawia b
- Jeśli s = "10", to pod y podstawia c
- W pozostałych przypadkach, pod y podstawia "XXX"

Zadanie 1.

Analiza kodu:

Licznik przyjmuje trzy sygnały na wejściu: sygnał rst, sygnał zegara i sygnał q. Dwa pierwsze sygnały są wartościami logicznymi, a trzeci jest wektorem bitów. Następnie opisujemy zachowanie się licznika (w ciele konstrukcji process). Jeśli sygnał reset jest równy 0, to q jest wektorem: (1, 0, 1, 0, 0, 0, 0, 0). Potem, jeśli na zegarze jest zbocze rosnące, to do wartości q dodajemy 1. Po kodzie widać, że następna wartość licznika zależy od poprzedniej (dlatego q jest **inout**).

Odpowiedź na pytanie:

Instrukcja nie będzie działać, ponieważ nie możemy dodać do wektora bitów znaku (w tym przypadku to znak 1 - dla VHDL taka operacja jest niezdefiniowana).

Zmodyfikowane czasy zadania sygnału rst znajdują się w pliku **simple_tb.vhd**

(umieszczone są tam komentarze). Aby licznik doliczył najpierw do 195, to musimy czekać przez 700 ns, ponieważ do licznika musimy dodać 35 razy jedynkę (okres zegara to 20 ns, więc czas to 35*20 ns = 700 ns). Potem musimy ustawić starą wartość licznika (tj. 160) i czekać przez 440 ns (bo teraz do licznika musimy dodać 22 razy jedynkę, więc 22*20 ns = 440 ns).

Kod:

```
LIBRARY ieee;
library std;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
use std.textio.all;

ENTITY simple_tb IS
END simple_tb;

ARCHITECTURE behavior OF simple_tb IS

    -- UUT (Unit Under Test)
    COMPONENT simple
    PORT(
        clk : IN std_logic;
        rst : IN std_logic;
        q : INOUT unsigned(7 downto 0)
    );
    END COMPONENT;

    -- input signals
    signal clk : std_logic := '0';
    signal rst : std_logic := '0';

    -- input/output signal
    signal qq : unsigned(7 downto 0);

    -- set clock period
    constant clk_period : time := 20 ns;

BEGIN
    -- instantiate UUT
    uut: simple PORT MAP (
        clk => clk,
        rst => rst,
        q => qq
    );

    -- clock management process
    -- no sensitivity list, but uses 'wait'
    clk_process:PROCESS
    BEGIN
        clk <= '0';
        WAIT FOR clk_period/2;
        clk <= '1';
        WAIT FOR clk_period/2;
    END PROCESS;

--:--- simple_tb.vhd Top L1 (VHDL)
```

```
        clk <= '1';
        WAIT FOR clk_period/2;
    END PROCESS;

    -- stimulating process
    stim_proc: PROCESS
        variable my_line : line;
    BEGIN
        -- let it run
        wait for 100 ns;
        -- apply reset
        rst <= '1';
        -- czekam 700 ns, bo musi dodac 35 razy jedynke do licznika (t = 35*clk_period)
        wait for 700 ns;
        -- and let it go
        -- wypisuje wartosc licznika na wyjście
        write(my_line, to_integer(qq));
        writeline(output, my_line);
        -- licznik znowu bedzie miec wartosc 160
        rst <= '0';
        -- czekam
        wait for 50 ns;
        -- rozpoczynam odliczanie
        rst <= '1';
        -- czekam 440 ns, bo musi teraz sie zmienic 22 razy (t = 22*clk_period)
        wait for 440 ns;
        -- wypisuje wartosc licznika
        write(my_line, to_integer(qq));
        writeline(output, my_line);
        wait;
    END PROCESS;
END;
```

Zadanie 2.

Analiza kodu

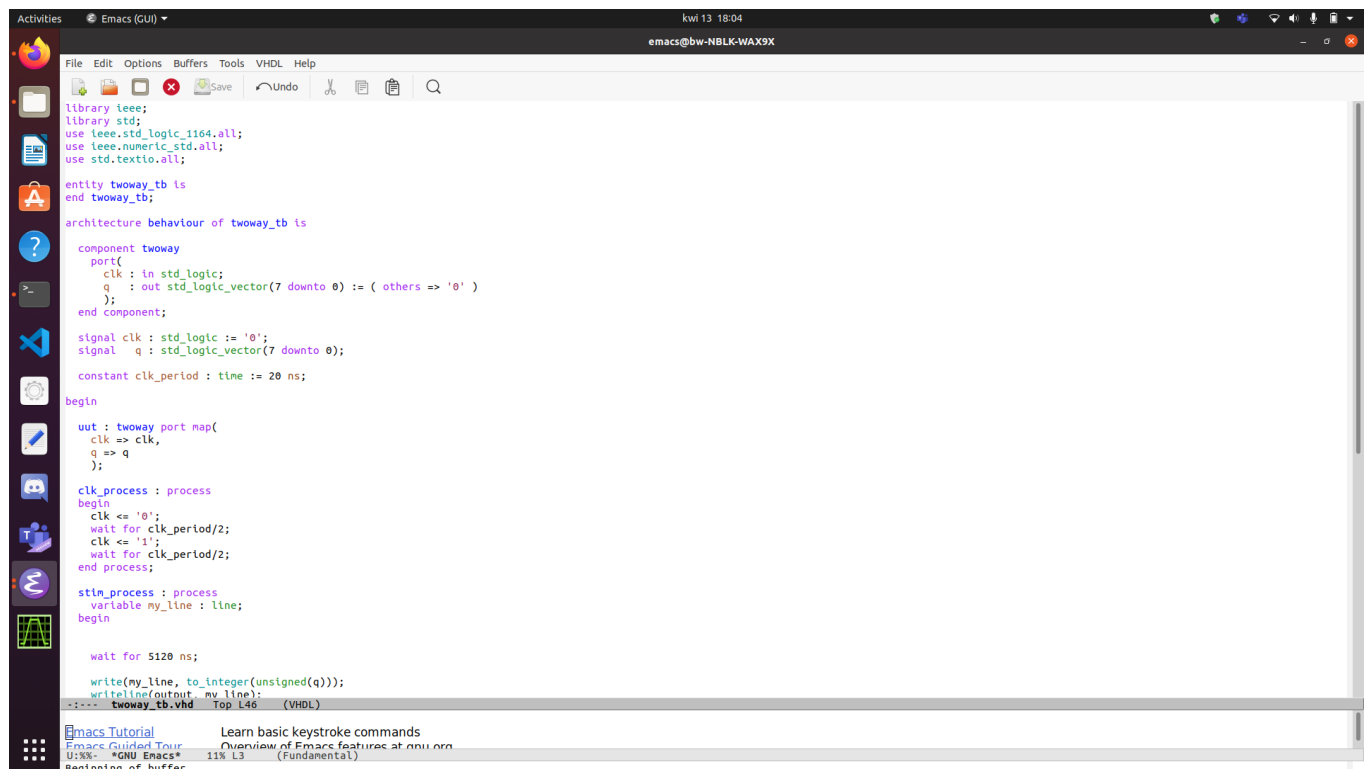
Licznik korzysta z flagi, której wartość determinuje, czy dodajemy lub odejmujemy 1 od wektora (wartość flagi to 1 => do wektora dodajemy 1 ; wartość flagi to 0 => od wektora odejmujemy 1). Jeśli wektor będzie reprezentacją dwójkową 0 (po wielokrotnym

odejmowaniu), to ustawiamy flagę na 1 (zatem później będziemy dodawać 1 do wektora). Jeśli wektor będzie reprezentacją dwójkową liczby większej niż $2^{(N\text{Bits})} - 1$, to ustawiamy flagę na 0 (więc później będziemy odejmować 1 od wektora).

Teza:

Licznik zacznie od wektora z samymi zerami. Od tego wektora odejmie 1, co da wartość $2^{(N\text{Bits})}-1$ (mamy 8 bitów, więc ta wartość będzie wynosić 255 i dodatkowo pracujemy na typie *unsigned*). Flaga *dir* ma wartość 0, więc 1 będzie odejmowana od licznika, aż ten będzie miał wartość 0 (zatem licznik odliczy od 255 do 0). Po wyzerowaniu licznika flaga *dir* zmieni wartość na 1, i będziemy dodawać do licznika 1 do momentu, kiedy wartość licznika przekroczy 255.

Kod: (znajduje się w pliku *twoway_tb.vhd*)



```
library ieee;
library std;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity twoway_tb is
end twoway_tb;

architecture behaviour of twoway_tb is

    component twoway
    port(
        clk : in std_logic;
        q : out std_logic_vector(7 downto 0) := ( others => '0' )
    );
    end component;

    signal clk : std_logic := '0';
    signal q : std_logic_vector(7 downto 0);

    constant clk_period : time := 20 ns;

begin

    uut : twoway port map(
        clk => clk,
        q => q
    );

    clk_process : process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    stin_process : process
    variable my_line : line;
    begin

        wait for 5120 ns;

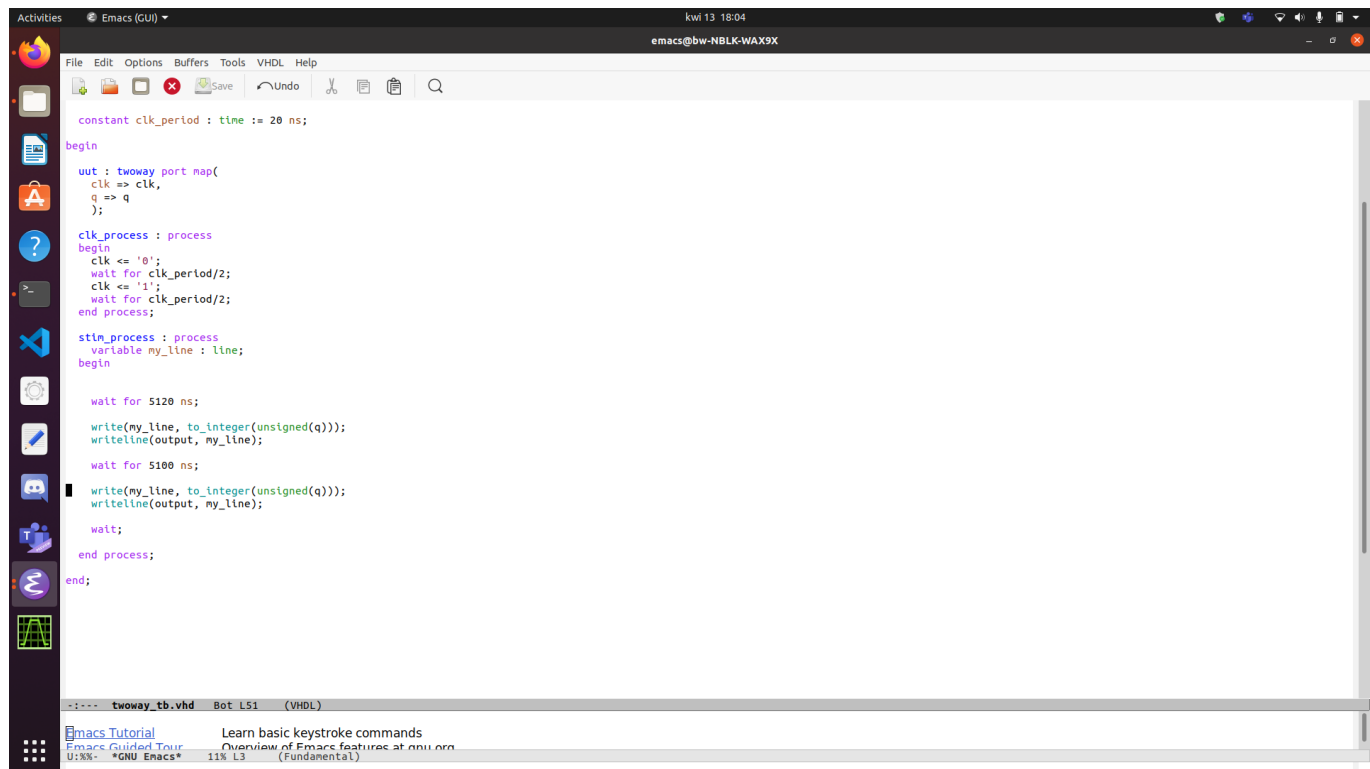
        write(my_line, to_integer(unsigned(q)));
        writeline(output, my_line);

    end process;

end architecture;
```

twoway_tb.vhd Top L46 (VHDL)

Emacs Tutorial Learn basic keystroke commands
Emacs Guided Tour Overview of Emacs features at org.org
1:38% GNU Emacs 11% L3 (Fundamental)
Beginning of buffer



Po uruchomieniu testu widać, że licznik najpierw odliczył do 0, a potem doliczył do 255 (po odpowiednim skalibrowaniu czasu).

Zadanie 3.

- *Program testowy:*
W moim programie testowym, kopiuje połowy (bajty) obecnej zawartości rejestru do dwóch tablic (*u_byte*, *l_byte*). Aby to osiągnąć korzystam z tego, że w VHDL tablice można bez problemu kopiować za pomocą instrukcji przypisania.
- *Program wygenerowany:*
Rejestr LFSR wygenerowałem za pomocą następującej instrukcji (w terminalu):

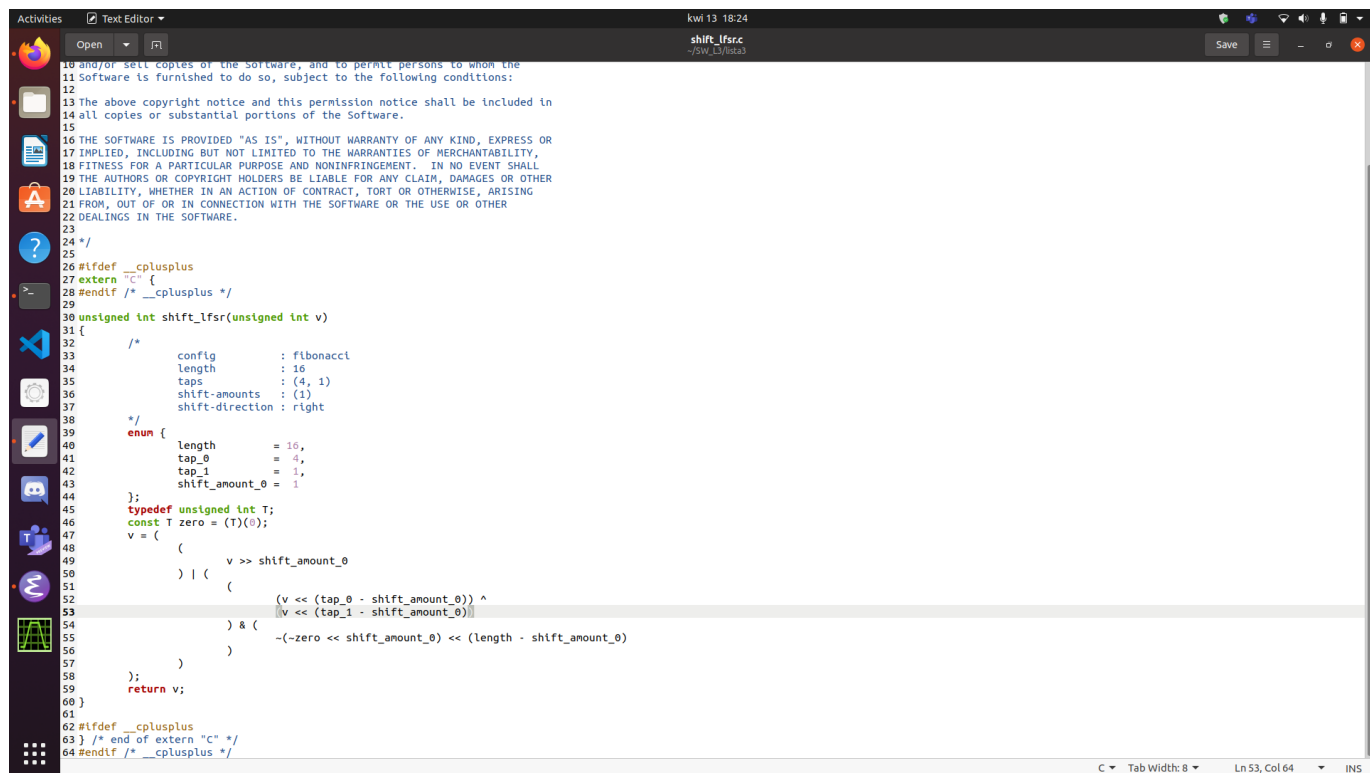
```
lfsr-generator --length=16 --taps=4,1 --shift-amounts=1 > shift_lfsr.c
```

Z dokumentacji można dowiedzieć się, że za pomocą tej konkretnej komendy, wygenerowałem następujący LFSR:

- ma długość 16 bitów
- za każdym razem przesuwamy go o jedną pozycję
- jest typu Fibonacciego (za pomocą tej komendy można również utworzyć LFSR typu Galois) - do wygenerowania nowego bitu, używamy **XOR'a** wybranych bitów znajdujących się w rejestrze
- na nowy bit, mają wpływ bit pierwszy i czwarty (oraz ostatni (tj. 15), ponieważ to jest LFSR typu Fibonacciego)

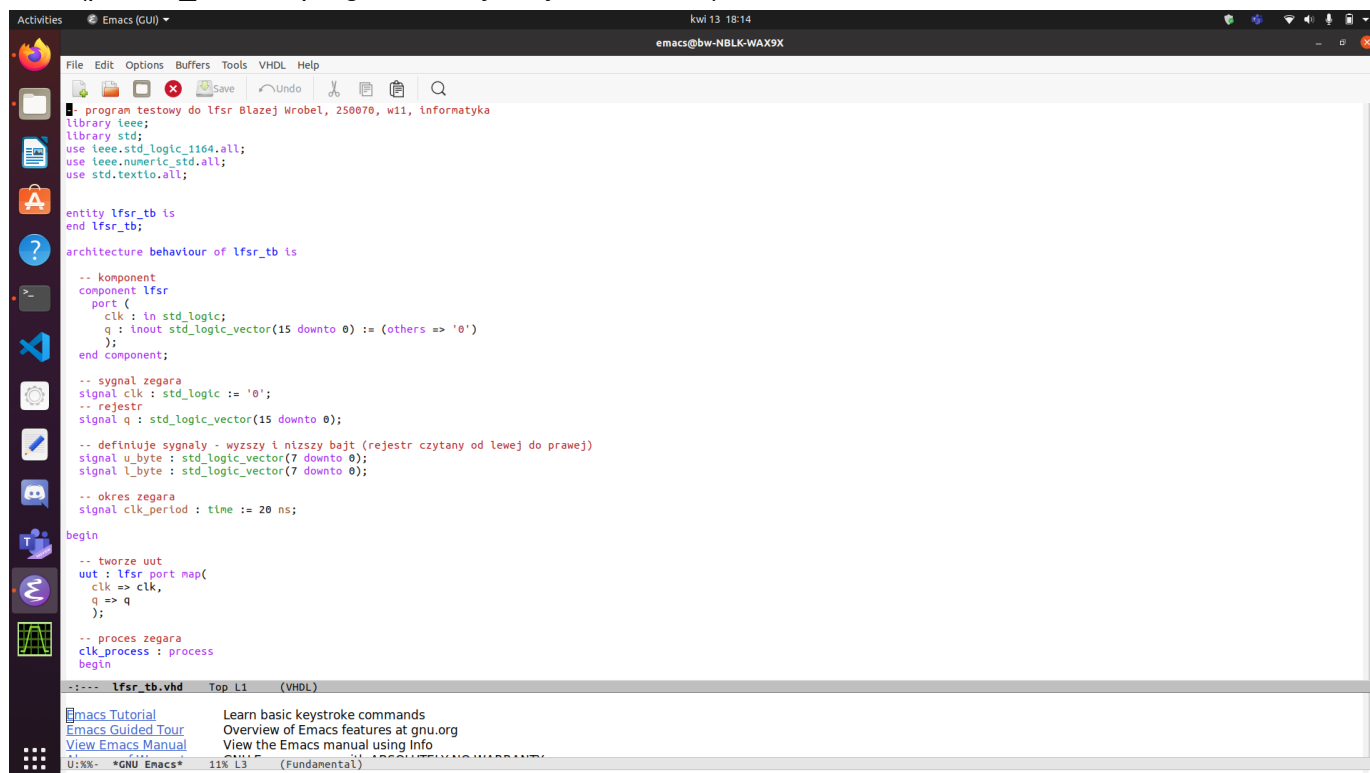
Wygenerowany kod był w języku C, więc musiałem go przetłumaczyć na model w VHDL.

Kod: (plik `shift_lfsr.c` - wygenerowany rejestr)



```
10 and/or sell copies of the Software, and to permit persons to whom the
11 Software is furnished to do so, subject to the following conditions:
12
13 The above copyright notice and this permission notice shall be included in
14 all copies or substantial portions of the Software.
15
16 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
19 THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
21 FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
22 DEALINGS IN THE SOFTWARE.
23
24 */
25
26 #ifndef __cplusplus
27 extern "C" {
28 #endif /* __cplusplus */
29
30 unsigned int shift_lfsr(unsigned int v)
31 {
32     /*
33      * config      : fibonacci
34      * length       : 16
35      * taps         : (4, 1)
36      * shift-amounts : (1)
37      * shift-direction : right
38      */
39     enum {
40         length      = 16,
41         tap_0        = 4,
42         tap_1        = 1,
43         shift_amount_0 = 1
44     };
45     typedef unsigned int T;
46     const T zero = (T)0;
47     v = (
48         (
49             v >> shift_amount_0
50         ) | (
51             (
52                 (v << (tap_0 - shift_amount_0)) ^
53                 (v << (tap_1 - shift_amount_0))
54             ) & (
55                 ~(-zero << shift_amount_0) << (length - shift_amount_0)
56             )
57         )
58     );
59     return v;
60 }
61
62 #ifdef __cplusplus
63 } /* end of extern "C" */
64 #endif /* __cplusplus */
```

Kod: (plik `lfsr_tb.vhd` - program do wydobywania bitów)



```
File Edit Options Buffers Tools VHDL Help
-- program testowy do lfsr Blazej Wrobel, 250070, w11, informatyka
library ieee;
library std;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity lfsr_tb is
end lfsr_tb;

architecture behaviour of lfsr_tb is

    -- komponent
    component lfsr
    port (
        clk : in std_logic;
        q : inout std_logic_vector(15 downto 0) := (others => '0');
    );
    end component;

    -- sygnał zegara
    signal clk : std_logic := '0';
    -- rejestr
    signal q : std_logic_vector(15 downto 0);

    -- definiuje sygnały - wyższy i niższy bajt (rejestr czytany od lewej do prawej)
    signal u_byte : std_logic_vector(7 downto 0);
    signal l_byte : std_logic_vector(7 downto 0);

    -- okres zegara
    signal clk_period : time := 20 ns;

begin

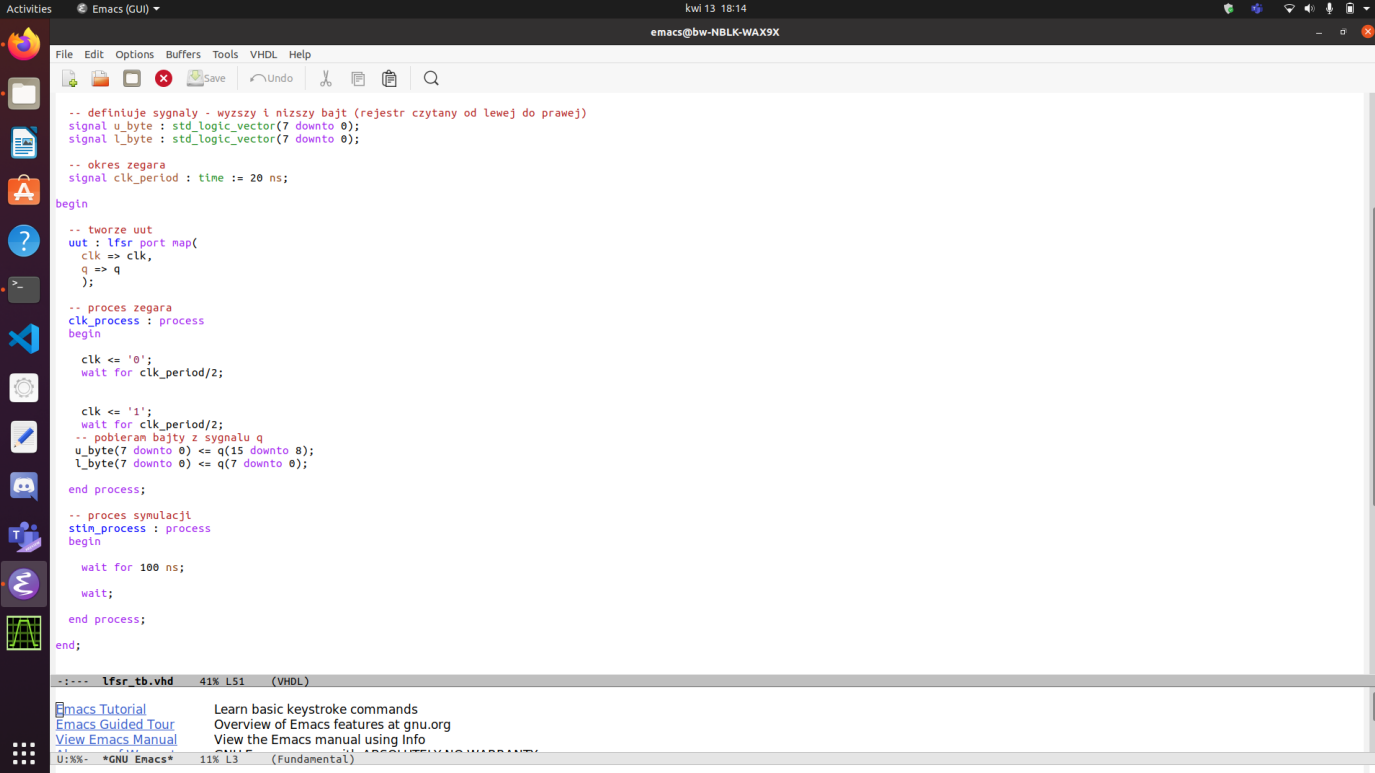
    -- tworze uut
    uut : lfsr port map(
        clk => clk,
        q => q
    );

    -- proces zegara
    clk_process : process
    begin

    end process;

end behaviour;

-- lfsr_tb.vhd Top L1 (VHDL)
```



The screenshot shows the Emacs GUI with a dark theme. The main window displays VHDL code for a shift register model. The code includes comments in Polish and defines signals, a clock period, and two processes: one for the LFSR and one for the shift register. The status bar at the bottom indicates the file is 'lfsr_tb.vhd' and the cursor is at line 41, column 151.

```
-- definiuje sygnały - wyższy i niższy bajt (rejestr czytany od lewej do prawej)
signal u_byte : std_logic_vector(7 downto 0);
signal l_byte : std_logic_vector(7 downto 0);

-- okres zegara
signal clk_period : time := 20 ns;

begin

-- tworze uut
uut : lfsr port map(
  clk => clk,
  q => q
);

-- proces zegara
clk_process : process
begin
  clk <= '0';
  wait for clk_period/2;

  clk <= '1';
  wait for clk_period/2;
  -- pobieram bajty z sygnału q
  u_byte(7 downto 0) <= q(15 downto 8);
  l_byte(7 downto 0) <= q(7 downto 0);
end process;

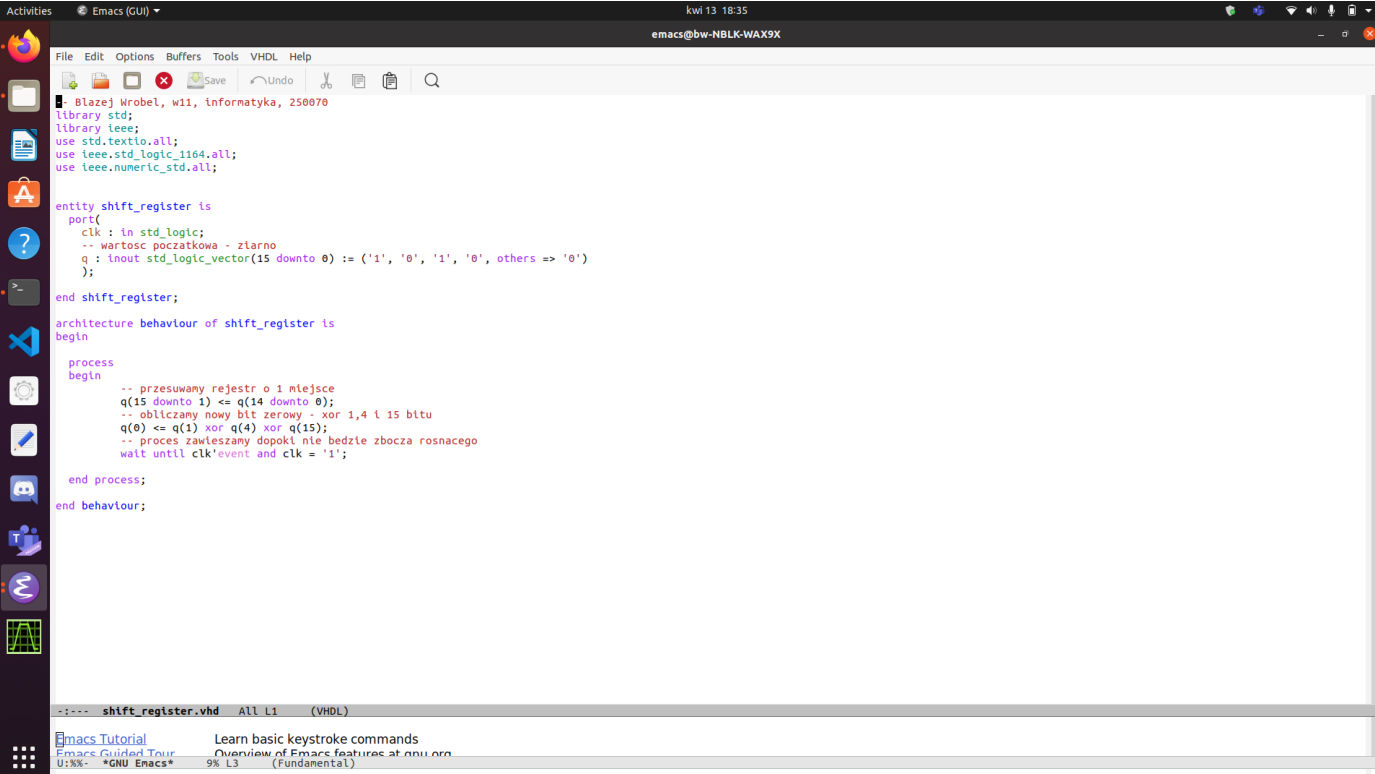
-- proces symulacji
stim_process : process
begin
  wait for 100 ns;
  wait;
end process;

end;
```

Emacs Tutorial Learn basic keystroke commands
Emacs Guided Tour Overview of Emacs features at gnu.org
View Emacs Manual View the Emacs manual using Info

U:%%~ GNU Emacs* 11% L3 (Fundamental)

Kod: (plik *shift_register.vhd* - model wygenerowanego rejestru w VHDL)



The screenshot shows the Emacs GUI with a dark theme. The main window displays VHDL code for a shift register model. The code includes comments in Polish and defines the entity, port, and architecture of the shift register. The status bar at the bottom indicates the file is 'shift_register.vhd' and the cursor is at line A11, column L1.

```
library std;
library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shift_register is
  port(
    clk : in std_logic;
    -- wartosc poczatkowa - zlaro
    q : inout std_logic_vector(15 downto 0) := ('1', '0', '1', '0', others => '0')
  );
end shift_register;

architecture behaviour of shift_register is
begin
  process
  begin
    -- przesuwamy rejestr o 1 miejsce
    q(15 downto 1) <= q(14 downto 0);
    -- obliczamy nowy bit zerowy - xor 1,4 i 15 bitu
    q(0) <= q(1) xor q(4) xor q(15);
    -- proces zawieszony dopoki nie bedzie zbocza rosnacego
    wait until clk'event and clk = '1';
  end process;
end behaviour;
```

Emacs Tutorial Learn basic keystroke commands
Emacs Guided Tour Overview of Emacs features at gnu.org
U:%%~ GNU Emacs* 9% L3 (Fundamental)

Kod: (plik *shift_register_tb.vhd* - program do wydobywania bitów z nowego modelu)

```
Blazej Wrobel, 250070, w11, informatyka
library std;
library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shift_register_tb is
end shift_register_tb;

architecture behaviour of shift_register_tb is

    -- uut
    component shift_register
        port (
            clk : in std_logic;
            q : inout std_logic_vector(15 downto 0) := (others => '0')
        );
    end component;
    -- sygnal zegara
    signal clk : std_logic := '0';
    -- rejestr
    signal q : std_logic_vector(15 downto 0);

    -- tablice na bajty (odpowlednio bajt wyzszy i nizszy)
    signal u_byte : std_logic_vector(7 downto 0);
    signal l_byte : std_logic_vector(7 downto 0);

    -- sygnal zegara
    signal clk_period : time := 20 ns;

begin
    uut : shift_register port map(
        clk => clk,
        q => q
    );

    -- proces zegara
    clk_process : process
    begin
        clk <= '0';
        wait for clk_period/2;

        clk <= '1';
        wait for clk_period/2;

    end process;

end;
```

```
        clk <= '0';
        wait for clk_period/2;

        clk <= '1';
        wait for clk_period/2;

        -- kopiujemy bajty
        u_byte(7 downto 0) <= q(15 downto 8);
        l_byte(7 downto 0) <= q(7 downto 0);

    end process;

    -- proces symulacji
    stim_process : process
    begin
        wait for 100 ns;

        wait;

    end process;

end;
```

Jak widać, obydwa rejestry są podobne do siebie - do obliczania nowego bitu, korzystają z **XOR'a** bitów znajdujących się już w rejestrze. Rejestr z dostarczonego pliku dodatkowo negował wynik **XOR'a** wybranych bitów (bit 15,14,13 i 4 tzw. *tapy*). Wygenerowany rejestr był czystym LFSR Fibonacciego - korzystał wyłącznie z **XOR'a** bitów 1,4 i 15. Za pomocą programów testowych i programu *gtkwave* możemy zobaczyć jakie pseudolosowe bajty

wygenerowały obydwa rejestry dla różnych stanów początkowych rejestrów. Dla wektora początkowego złożonego z samych 0 (tzw. ziarno), wygenerowany LFSR nie dawał żadnych ciekawych wyników (wychodziły same 0, bo **XOR** tych samych wartości zawsze daje 0), a LFSR z dostarczonego pliku, generował już konkretne bajty pseudolosowe (dla tego samego ziarna - to zasługa negacji, która pozwoliła uzyskać 1 w rejestrze). Dopiero zmiana ziarna spowodowała, że wygenerowany LFSR dawał ciekawsze wyniki. Po analizie bajtów za pomocą gtkwave można stwierdzić, że dostarczony LFSR generował bajty, których wartość w systemie dziesiętnym spełniała zależność $2x+1$ (gdzie x to wartość poprzedniego bajtu w systemie dziesiętnym). W przypadku wygenerowanego LFSR wartości miały postać $2x+1$ lub $2x$ (x jest tym samym co poprzednio) .

Wykresy generowanych bajtów w zależności od czasu:

lfsr.vcd, shift.vcd - generowane bajty pseudolosowe dla różnych ziaren.