Układy Cyfrowe – laboratorium

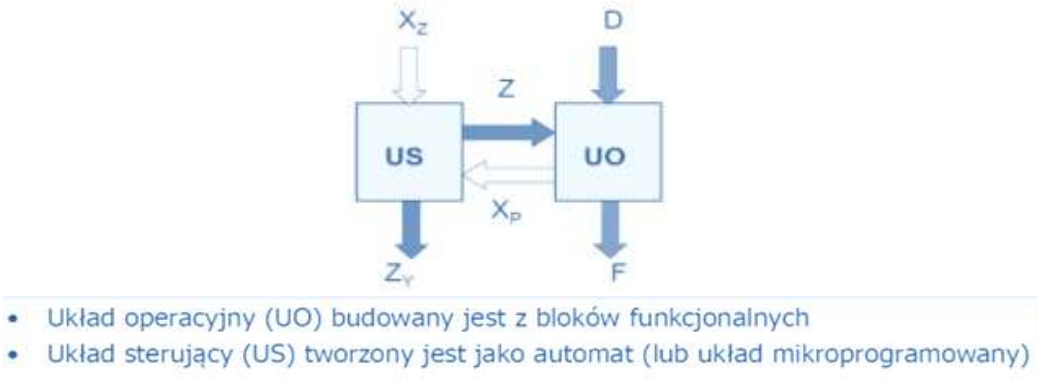
Przykład realizacji ćwiczenia nr 7

Temat:

Realizacja układu sterującego systemu cyfrowego z uwzględnieniem kodowania i dekompozycji funkcji p-w automatu dla Metody Newtona, tj. iteracyjnego algorytmu wyznaczania przybliżonej wartości pierwiastka funkcji.

Spis treści

I. Przedmiot i cel laboratorium powrót



Rys. 1. Realizacja systemu cyfrowego z podziałem na układ sterujący i operacyjny

Przedmiotem laboratorium są zaawansowane metody syntezy logicznej do realizacji systemów cyfrowych w strukturach programowalnych FPGA według modelu (rys.1) z wydzielonymi blokami – operacyjnym i sterującym. Podstawowymi narzędziami w laboratorium są: środowisko projektowe QUARTUS II, symulator ModelSim-Altera i platforma uruchomieniowa DE2-115 (z układem Cyclone IVE EP4CE115F29C7). Do specyfikacji algorytmów stosowany jest język opisu sprzętu VHDL. Głównym celem laboratorium jest zapoznanie z praktycznymi formami posługiwania się modelem UO-US przy projektowaniu systemu cyfrowego z użyciem języka VHDL ze szczególnym uwzględnieniem możliwych modyfikacji samego układu sterującego. Istotną uwagę zwraca się na różne sposoby przedstawiania algorytmów i realizacji ich funkcji za pomocą automatów. Treść poszczególnych zadań, przewidzianych w programie ćwiczenia, jest szczegółowo zilustrowana i wyjaśniona w przykładowych projektach, których specyfikacje zapisano w języku VHDL.

Przykładowe realizacje algorytmu podane są na wydrukach i przedstawiają cztery warianty implementacji części sterującej wydzielonej z systemu, który był tematem laboratorium 5 (**Realizacja algorytmu wyznaczania przybliżonej wartości pierwiastka funkcji** / Metoda Newtona, zwana również metodą Newtona-Raphsona lub metodą stycznych, polegająca na przybliżaniu w kolejnych krokach algorytmu miejsca zerowego/).

II. Zadania projektowe powrót

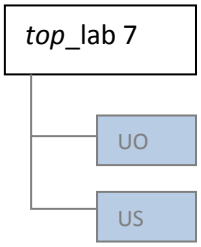
1. Punktem wyjścia jest ogólny model systemu z automatem sterującym (fsm), który jest zadany w postaci abstrakcyjnej; jest on automatycznie kodowany w trakcie kompilacji przez system QUARTUS. W dalszej części będzie modyfikowany układ sterujący systemu.
2. W tym kroku realizowany Układ Sterujący ma uwzględniać zadane kodowanie stanów automatu, z zastosowaniem różnych kodów, w tym kodów o różnych długościach słowa.
3. W kolejnej wersji zadania należy wykorzystać zakodowaną tablicę p-w automatu z pkt.1 i na jej podstawie utworzyć tablicę prawdy do schematu z rys.3. Następnie zamienić utworzony opis tablicy prawdy, wraz opisującymi ją zmiennymi we/wy, z poprzednią konstrukcją specyfikacji algorytmu z pkt.2 (tablica p-w w części - architecture asm of us is …).
4. W tym punkcie należy zastosować metodę dekompozycji tablicy prawdy z poprzedniego zadania. W tym celu należy posłużyć się programem do dekompozycji DEMAIN.

Przedstawione w dalszej części wydruki specyfikacji algorytmu stanowią podstawę do samodzielnej realizacji zadań postawionych na laboratorium 7. Szczegółowe dane będą podane na początku zajęć. W części praktycznej ćwiczenia należy uruchomić zaprojektowany i skompilowany wariant systemu przy

użyciu płyty uruchomieniowej DE2-115 wg schematu przedstawionego na rys. 5 oraz podanego opisu wyprowadzeń sygnałów (Lista wyprowadzeń we/wy).

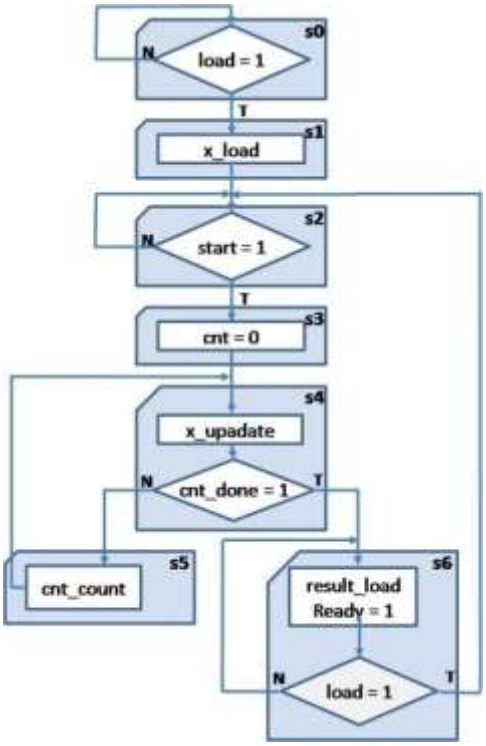
III. Przykładowe realizacje poszczególnych zadań powrót

1 Pierwotna realizacja systemu UO-US – plik główny powrót

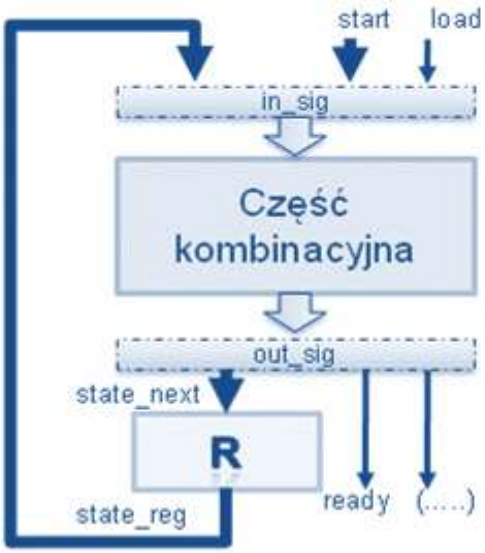



```
-- top_lab7
library ieee;
use ieee.std_logic_1164.all;

entity top_lab7 is
      port (
            rst         :  in std_logic;
            clk         :  in std_logic;
            start, load :  in std_logic;
            data        :  in std_logic_vector(9 downto 0);
            ready       :  out std_logic;
            result      :  out std_logic_vector(9 downto 0)
      );
end top_lab7;

architecture structure of top_lab7 is
      -- Liczba dla której liczony jest pierwiastek
      constant valueToFindTheRootOf : natural := 17;

      component uo
            generic(
                  number      : integer := 99
            );
            port (
                  rst         : in std_logic;
                  clk         : in std_logic;
                  cnt_reset   : in std_logic;
                  cnt_count   : in std_logic;
                  x_load      : in std_logic;
                  x_update    : in std_logic;
                  result_load : in std_logic;
                  data        : in std_logic_vector(9 downto 0);
                  cnt_done    : out std_logic;
                  result      : out std_logic_vector(9 downto 0)
            );
      end component;



      component us
            port(
                  rst         : in std_logic;
                  clk         : in std_logic;
                  start, load : in std_logic;
```

```
                cnt_done    : in std_logic;
                cnt_reset   : out std_logic;
                cnt_count   : out std_logic;
                x_load      : out std_logic;
                x_update    : out std_logic;
                result_load : out std_logic;
                ready       : out std_logic
            );
    end component;
    signal cnt_reset_wire   : std_logic;
    signal cnt_count_wire   : std_logic;
    signal x_load_wire      : std_logic;
    signal x_update_wire    : std_logic;
    signal result_load_wire : std_logic;
    signal cnt_done_wire    : std_logic;
begin
    b1 : uo
    generic map(
        number      => valueToFindTheRootOf
    )
    port map(
        rst         => rst,
        clk         => clk,
        cnt_reset   => cnt_reset_wire,
        cnt_count   => cnt_count_wire,
        x_load      => x_load_wire,
        x_update    => x_update_wire,
        result_load => result_load_wire,
        data        => data,
        cnt_done    => cnt_done_wire,
        result      => result
    );
    b2 : us port map(
        rst         => rst,
        clk         => clk,
        start       => start,
        load        => load,
        cnt_done    => cnt_done_wire,
        cnt_reset   => cnt_reset_wire,
        cnt_count   => cnt_count_wire,
        x_load      => x_load_wire,
        x_update    => x_update_wire,
        result_load => result_load_wire,
        ready       => ready
    );

end;
```



Rys. 2. Algorytm układu sterującego (bloki ASM)

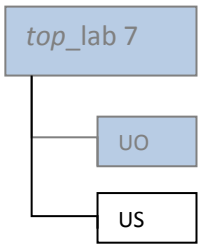


Rys.3. Schemat blokowy funkcji przejść

1 Pierwotna realizacja systemu UO-US – plik US powrót

Układ sterujący z automatem (fsm) jest zadany w postaci abstrakcyjnej; jest on automatycznie kodowany w trakcie kompilacji przez system QUARTUS.

```
-- top_lab7 [us.vhd]
library ieee;
use ieee.std_logic_1164.all;

entity us is
      port(
            rst         : in std_logic;
            clk         : in std_logic;
            start, load : in std_logic;
            cnt_done    : in std_logic;
            cnt_reset   : out std_logic;
            cnt_count   : out std_logic;
            x_load      : out std_logic;
            x_update    : out std_logic;
            result_load : out std_logic;
            ready       : out std_logic
      );
end us;
```



```
architecture asm of us is
	-- Stany automatu i sygnały rejestru stanów
	type STATE_TYPE is (s0, s1, s2, s3, s4, s5, s6);
	signal state_reg, state_next  : STATE_TYPE;

	signal ready_reg, ready_next  : std_logic;

begin
	-- Rejestr stanu automatu i sygnału ready
	process(rst, clk)
	begin
		if rst = '1' then
			state_reg   <= s0;
			ready_reg   <= '0';
		elsif rising_edge(clk) then
			state_reg   <= state_next;
			ready_reg   <= ready_next;
		end if;
	end process;

	-- Funkcja przejść-wyjść automatu
	process(state_reg, start, load, cnt_done)
	begin
		cnt_reset   <= '0';
		cnt_count   <= '0';
		x_load      <= '0';
		x_update    <= '0';
		result_load <= '0';
		ready_next  <= '0';

		case state_reg is
			when s0 =>
				if load = '1' then
					state_next <= s1;
				else
					state_next <= s0;
				end if;
			when s1 =>
				state_next  <= s2;
				x_load      <= '1';
			when s2 =>
				if start = '1' then
					state_next <= s3;
				else
					state_next <= s2;
				end if;
			when s3 =>
				state_next  <= s4;
				cnt_reset   <= '1';
```

```
                when s4 =>
                      if cnt_done = '1' then
                            state_next <= s6;
                      else
                            state_next <= s5;
                      end if;
                      x_update    <= '1';
                when s5 =>
                      state_next  <= s4;
                      cnt_count   <= '1';
                when s6 =>
                      if load = '1' then
                            state_next <= s2;
                      else
                            state_next <= s6;
                      end if;
                      ready_next  <= '1';
                      result_load <= '1';
                when others =>
                      state_next  <= s0;
          end case;
      end process;
      ready <= ready_reg;
end;
```

Kompilacja i weryfikacja

Weryfikacja poprawności działania pierwotnego projektu top_lab7 (jak i pozostałych wersji projektu) jest przeprowadzana na drodze symulacji za pomocą symulatora ModelSim-Altera. Dane do symulacji: grid/time period – 200ns (okres zegara), end time - 10μs (czas symulacji).
Projekt należy zrealizować w układzie FPGA Cyclone IV EP4CE115F29C7, znajdującym się na płycie uruchomieniowej DE2-115.
Weryfikacja poprawności działania systemu polega na:
- wywołaniu symulatora ModelSim z programu Quartus:
Tools>Run EDA Simulation Tool >EDA RTL Simulation;
- uruchomieniu symulacji dla odpowiedniego pliku *top…* z biblioteki *rtl_work*;
- wykonaniu przykładowego skryptu z wektorami testowymi (podkatalog *\simulation\modelsim*).

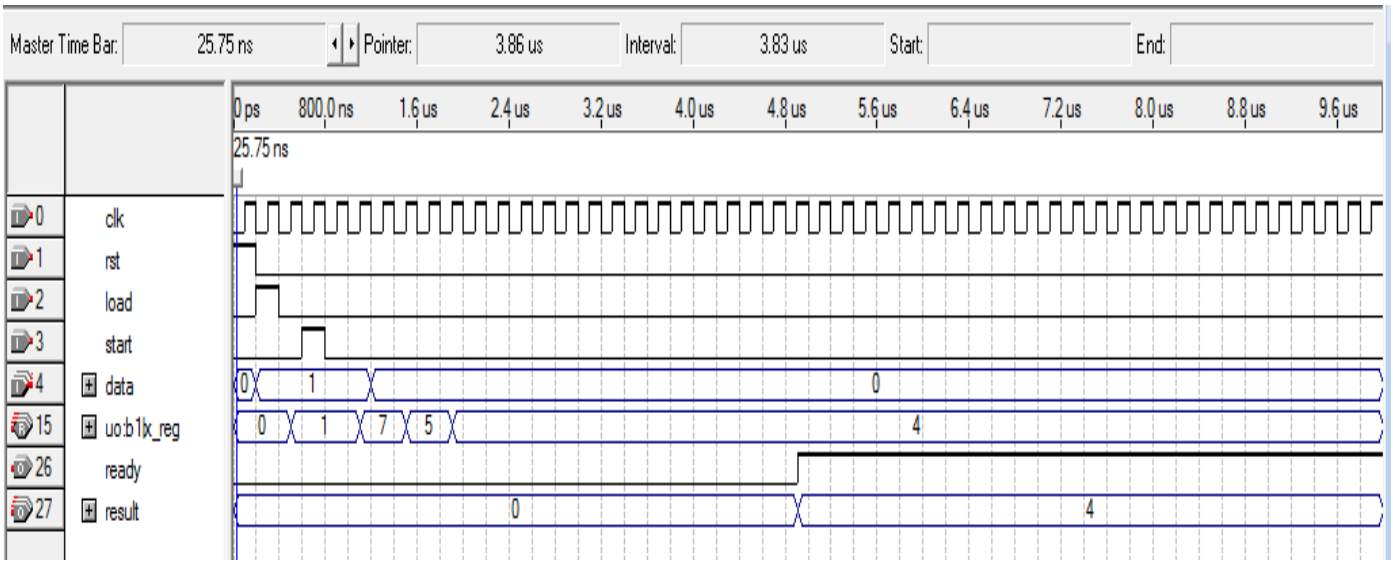
Komenda w oknie transkrypcji: VSIM> do test.do

-- test.do [zawartość skryptu symulacji]

```
restart -force -nowave
-- Lista wybranych sygnałów do symulacji
add wave \
{sim:/top_lab7/clk } \
{sim:/top_lab7/rst } \
{sim:/top_lab7/load } \
{sim:/top_lab7/start } \
{sim:/top_lab7/data } \
{sim:/top_lab7/b1/cnt_reg } \
{sim:/top_lab7/b1/x_reg } \
{sim:/top_lab7/b2/state_reg } \
{sim:/top_lab7/ready } \
{sim:/top_lab7/result }
```

```
-- Określenie przebiegu zegara i sygnałów wejściowych
force -freeze sim:/top_lab7/clk 0 0, 1 {100000 ps} -r {200 ns}
force -freeze sim:/top_lab7/rst 1 0
force -freeze sim:/top_lab7/load 0 0
force -freeze sim:/top_lab7/start 0 0
force -freeze sim:/top_lab7/data 00000000 0
run 200 ns
force -freeze sim:/top_lab7/rst 0 0
force -freeze sim:/top_lab7/load 1 0
force -freeze sim:/top_lab7/data 00000001 0
run 200 ns
force -freeze sim:/top_lab7/load 0 0
run 200 ns
force -freeze sim:/top_lab7/start 1 0
run 200 ns
force -freeze sim:/top_lab7/start 0 0
run 400 ns
force -freeze sim:/top_lab7/data 00000000 0
run 9 us
```

Widok wyniku testu przedstawia rys.4. Dodatkowo można sprawdzić poprawność wyniku wykorzystując formularz np. arkusza kalkulacyjnego jak na rys.5 (*dana* to stała valueToFindTheRootOf w top_lab7).



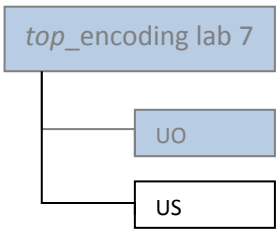
Rys.4. Wynik symulacji dla projektu top_lab7

dana=	17	17	17	17	17	17	18	17	17	17	17	17	17
x=	1	2	3	4	5	6	7	8	9	10	11	12	13
x^2=	1	4	9	16	25	36	49	64	81	100	121	144	169
x^3=	1	8	27	64	125	216	343	512	729	1000	1331	1728	2197
x^3-*dana*=	-16	-9	10	47	108	199	325	495	712	983	1314	1711	2180
x^2*3=	3	12	27	48	75	108	147	192	243	300	363	432	507
div=	-6	-1	0	0	1	1	2	2	2	3	3	3	4
sub=	7	3	3	4	4	5	5	6	7	7	8	9	9

Rys.5. Przybliżenia w kolejnych krokach miejsca zerowego wyliczania funkcji $\sqrt[3]{dana}$ według algorytmu Newtona (formularz MSExcel) [2]

2. Realizacja układu sterującego z zadanym kodowaniem powrót

Realizacja **układu sterującego z** uwzględnieniem **zadanego kodowania** stanów automatu: zastosowanie różnych kodów, w tym kodów o różnych długościach słowa.



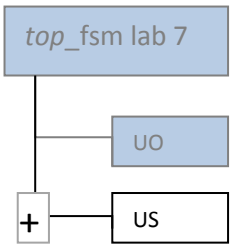

```
-- 2_top_encoding_lab7 [us.vhd]
library ieee;
use ieee.std_logic_1164.all;

entity us is
      port(
            rst               : in std_logic;
            clk               : in std_logic;
            start, load       : in std_logic;
            cnt_done          : in std_logic;
            cnt_reset         : out std_logic;
            cnt_count         : out std_logic;
            x_load            : out std_logic;
            x_update          : out std_logic;
            result_load       : out std_logic;
            ready             : out std_logic
      );
end us;

architecture asm of us is

      -- Stany automatu i sygnały rejestru stanów
      constant s0 : std_logic_vector( 4 downto 0) := "00000";
      constant s1 : std_logic_vector( 4 downto 0) := "00110";
      constant s2 : std_logic_vector( 4 downto 0) := "01100";
      constant s3 : std_logic_vector( 4 downto 0) := "11000";
      constant s4 : std_logic_vector( 4 downto 0) := "10001";
      constant s5 : std_logic_vector( 4 downto 0) := "00011";
      constant s6 : std_logic_vector( 4 downto 0) := "01010";
      -- type STATE_TYPE is (s0, s1, s2, s3, s4, s5, s6);
      signal state_reg, state_next  : std_logic_vector( 4 downto 0);
      signal ready_reg, ready_next  : std_logic;

begin
      -- Rejestr stanu automatu i sygnału ready
      process(rst, clk)
      begin
            if rst = '1' then
                  state_reg   <= s0;
                  ready_reg   <= '0';
            elsif rising_edge(clk) then
                  state_reg   <= state_next;
                  ready_reg   <= ready_next;
            end if;
      end process;

      -- Funkcja przejść-wyjść automatu
      process(state_reg, start, load, cnt_done)
```

```
	begin
		cnt_reset   <= '0';
		cnt_count   <= '0';
		x_load      <= '0';
		x_update    <= '0';
		result_load <= '0';
		ready_next  <= '0';

		case state_reg is
			when s0 =>
				if load = '1' then
					state_next <= s1;
				else
					state_next <= s0;
				end if;
			when s1 =>
				state_next  <= s2;
				x_load      <= '1';
			when s2 =>
				if start = '1' then
					state_next <= s3;
				else
					state_next <= s2;
				end if;
			when s3 =>
				state_next  <= s4;
				cnt_reset   <= '1';
			when s4 =>
				if cnt_done = '1' then
					state_next <= s6;
				else
					state_next <= s5;
				end if;
				x_update    <= '1';

			when s5 =>
				state_next  <= s4;
				cnt_count   <= '1';
			when s6 =>
				if load = '1' then
					state_next <= s2;
				else
					state_next <= s6;
				end if;
				ready_next  <= '1';
				result_load <= '1';
			when others =>
				state_next <= s0;
		end case;
	end process;
	ready <= ready_reg;
end;
```

3. Konstrukcja tablicy prawdy na podstawie zakodowanej tablicy p-w powrót

Zadanie polega na **utworzeniu tablicy prawdy** do schematu z rys.3 na podstawie zakodowanej tablicy p-w automatu. Utworzoną tablicę prawdy (z opisującymi ją zmiennymi we/wy) należy podstawić w miejsce poprzedniej konstrukcji algorytmu z pkt 2 (tablica p-w w części - architecture asm of us is …).

3.1 Realizacja Układu Sterującego z tablicą prawdy powrót




```
-- 3_top_fsm_tab_lab7 [us.vhd]
library ieee;
use ieee.std_logic_1164.all;

entity us is
      port(
            rst               : in std_logic;
            clk               : in std_logic;
            start, load       : in std_logic;
            cnt_done          : in std_logic;
            cnt_reset         : out std_logic;
            cnt_count         : out std_logic;
            x_load            : out std_logic;
            x_update          : out std_logic;
            result_load       : out std_logic;
            ready             : out std_logic
      );
end us;

architecture asm of us is
      -- Stany automatu i sygnały rejestru stanów
      -- type STATE_TYPE is (s0, s1, s2, s3, s4, s5, s6);
      signal state_reg, state_next  : std_logic_vector( 4 downto 0);
      signal ready_reg, ready_next  : std_logic;

      component fsm_table is
            port(
                  inputs      : in std_logic_vector( 7 downto 0);
                    --- start, load, cnt_done, q[4..0]
                  outputs     : out std_logic_vector( 10 downto 0)
                  --- q[4..0], cnt_reset, cnt_count, x_load, x_update, result_load,
                  ready;
            );
      end component;
      signal fsm_tab_in       : std_logic_vector( 7 downto 0);
      signal fsm_tab_out      : std_logic_vector( 10 downto 0);
begin
      -- Rejestr stanu automatu i sygnału ready
      process(rst, clk)
      begin
            if rst = '1' then
                  state_reg   <= (others => '0');
                  ready_reg   <= '0';
            elsif rising_edge(clk) then
```

```
                state_reg   <= state_next;
                ready_reg   <= ready_next;
          end if;
    end process;

    fsm_tab_in  <= (start & load & cnt_done & state_reg);
    ready_next  <= fsm_tab_out(0);
    result_load <= fsm_tab_out(1);
    x_update    <= fsm_tab_out(2);
    x_load      <= fsm_tab_out(3);
    cnt_count   <= fsm_tab_out(4);
    cnt_reset   <= fsm_tab_out(5);
    state_next  <= fsm_tab_out(10 downto 6);
    --    (state_next, cnt_reset, cnt_count, x_load, x_update, result_load,
    ready_next) <= fsm_tab_out;

    table : fsm_table
          port map (
                inputs      => fsm_tab_in,
                outputs     => fsm_tab_out
          );

    ready <= ready_reg;
end;
```

3.2 Realizacja tablicy prawdy powrót

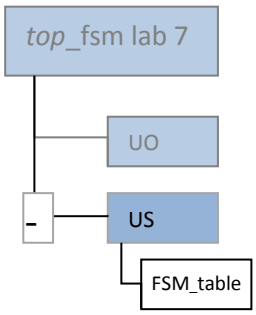
Zasada tworzenia tablicy prawdy (tabela 1) na podstawie funkcji przejść i wyjść (kodowania stanów, sygnałów wejściowych i sygnałów wyjściowych).

Sygnały wejściowe `start, load, cnt_done, q[4..0]`

Sygnały wyjściowe `q[4..0], cnt_reset, cnt_count, x_load, x_update, result_load, ready`

Tabela 1

S	start	load	cnt_done	q[4..0]	=>	S'	q'[4..0]	cnt_reset	cnt_count	x_load	x_update	result_load	ready
S0	-	0	-	00000	=>	S0	00000	0	0	0	0	0	0
	-	1	-		=>	S1	00110	0	0	0	0	0	0
S1	-	-	-	00110	=>	S2	01100	0	0	1	0	0	0
S2	0	-	-	01100	=>	S2	01100	0	0	0	0	0	0
	1	-	-		=>	S3	11000	0	0	0	0	0	0
S3	-	-	-	11000	=>	S4	10001	1	0	0	0	0	0
S4	-	-	1	10001	=>	S6	01010	0	0	0	1	0	0
	-	-	0		=>	S5	00011	0	0	0	1	0	0
S5	-	-	-	00011	=>	S4	10001	0	1	0	0	0	0
S6	-	0	-	01010	=>	S6	01010	0	0	0	0	1	1
	-	1	-			S2	01100	0	0	0	0	1	1




```
-- 3_top_fsm_tab_lab7 [fsm_table.vhd]
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fsm_table is
      port(
            inputs      : in std_logic_vector( 7 downto 0);
            --- start, load, cnt_done, q[4..0]
            outputs     : out std_logic_vector( 10 downto 0)
            --- q[4..0], cnt_reset, cnt_count, x_load, x_update, result_load, ready;

      );
end fsm_table;
architecture table of fsm_table is
begin
      outputs <=
            --s0
            "00000000000" when std_match(inputs, "-0-00000") else
            "00110000000" when std_match(inputs, "-1-00000") else
            --s1
            "01100001000" when std_match(inputs, "---00110") else
            --s2
            "01100000000" when std_match(inputs, "0--01100") else
            "11000000000" when std_match(inputs, "1--01100") else
            --s3
            "10001100000" when std_match(inputs, "---11000") else
            --s4
            "01010000100" when std_match(inputs, "--110001") else
            "00011000100" when std_match(inputs, "--010001") else
            --s5
            "10001010000" when std_match(inputs, "---00011") else
            --s6
            "01010000011" when std_match(inputs, "-0-01010") else
            "01100000011" when std_match(inputs, "-1-01010") else
            ----
            "-----------";
end;
```

4. Dekompozycja tablicy prawdy powrót

Realizacja automatu Układu Sterującego ze **zdekomponowaną tablicą prawdy**. W tym celu należy zapisać tablicę prawdy z pkt.2 w standardzie ESPRESSO (Berkeley'owskim) i zdekomponować przy użyciu programu DEMAIN (instrukcja obsługi i program w materiałach do laboratorium UCYF). W kolejnym uzyskany wynik należy przekonwertować na język VHDL (program DmainToVHD). Następnie podmienić 'component table_dec' w miejsce pierwotnego modułu 'component fsm_table'.

4.1 Konwersja tablicy prawdy .vhdl -> .pla / standard ESPRESSO powrót

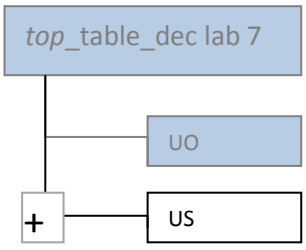
```
--s0
"00000000000" when std_match(inputs, "-0-00000") else
"00110000000" when std_match(inputs, "-1-00000") else
--s1
"01100001000" when std_match(inputs, "---00110") else
--s2
"01100000000" when std_match(inputs, "0--01100") else
"11000000000" when std_match(inputs, "1--01100") else
--s3
"10001100000" when std_match(inputs, "---11000") else
--s4
"01010000100" when std_match(inputs, "--110001") else
"00011000100" when std_match(inputs, "--010001") else
--s5
"10001010000" when std_match(inputs, "---00011") else
--s6
"01010000011" when std_match(inputs, "-0-01010") else
"01100000011" when std_match(inputs, "-1-01010") else
----
"-----------";
```

```
.type fr
.i 8
.o 11
.ilb i7 i6 i5 i4 i3 i2 i1 i0
.ob o10 o9 o8 o7 o6 o5 o4 o3 o2 o1 o0
-0-00000  00000000000
-1-00000  00110000000
---00110  01100001000
0--01100  01100000000
1--01100  11000000000
---11000  10001100000
--110001  01010000100
--010001  00011000100
---00011  10001010000
-0-01010  01010000011
-1-01010  01100000011
.end
```

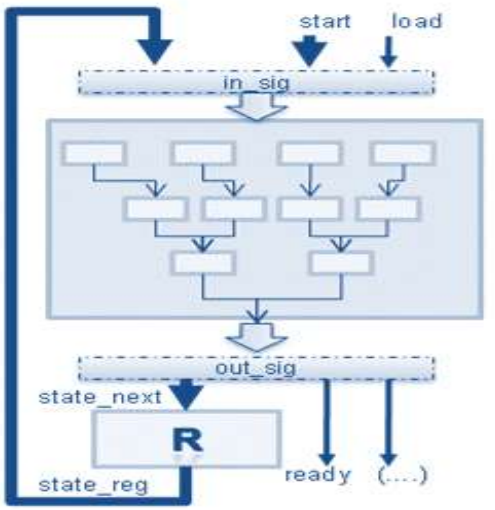
4.2 Plik US z zadeklarowanym komponentem po dekompozycji powrót

Schemat blokowy funkcji przejść po dekompozycji został przedstawiony na rys.4.

```
-- 4_top_tab_dec_lab7   [us.vhd]
library ieee;
use ieee.std_logic_1164.all;

entity us is
      port(
            rst         : in std_logic;
            clk         : in std_logic;
            start, load : in std_logic;
            cnt_done    : in std_logic;
            cnt_reset   : out std_logic;
            cnt_count   : out std_logic;
            x_load      : out std_logic;
            x_update    : out std_logic;
            result_load : out std_logic;
            ready       : out std_logic
      );
end us;
```



```
architecture asm of us is
	-- Stany automatu FSM i sygnaly rejestru stanow
	-- type STATE_TYPE is (s0, s1, s2, s3, s4, s5, s6);
	signal state_reg, state_next  : std_logic_vector( 4 downto 0);
	signal ready_reg, ready_next  : std_logic;

	component table_dec is
		port(
			i : in std_logic_vector( 7 downto 0);
	-- start, load, cnt_done, q[4..0]
			o	: out std_logic_vector( 10 downto 0)
	-- q[4..0], cnt_reset, cnt_count, x_load, x_update, result_load, ready;
		);
	end component;
		signal tab_in  : std_logic_vector( 7 downto 0);
	signal tab_out : std_logic_vector( 10 downto 0);
begin

	-- Rejestr stanu FSM i sygnalu ready
	process(rst, clk)
	begin
		if rst = '1' then
			state_reg   <= (others => '0');
			ready_reg   <= '0';
		elsif rising_edge(clk) then
			state_reg   <= state_next;
			ready_reg   <= ready_next;
		end if;
	end process;


	tab_in      <= (start & load & cnt_done & state_reg);
	ready_next  <= tab_out(0);
	result_load <= tab_out(1);
	x_update    <= tab_out(2);
	x_load      <= tab_out(3);
	cnt_count   <= tab_out(4);
	cnt_reset   <= tab_out(5);
	state_next  <= tab_out(10 downto 6);
	--(state_next, cnt_reset, cnt_count, x_load, x_update, result_load, ready_next)
	<= fsm_tab_out;

	table : table_dec
		port map (
			i	=> tab_in,
			o	=> tab_out
		);
	ready <= ready_reg;
end;
```



Rys.6. Schemat blokowy funkcji przejść po dekompozycji

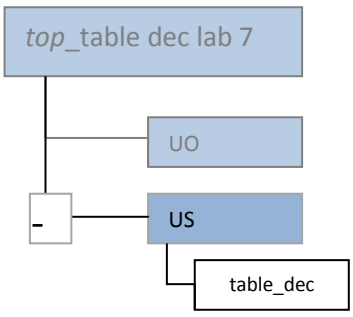
4.3 Dekompozycja tablicy prawdy powrót

Dekompozycję tablicy prawdy należy zrealizować przy użyciu programu Demain. Następnie należy dokonać konwersji wyników na język VHDL programem demainToVHD (konieczne pliki wejściowy x.pla i wynikowy x.out).

UWAGA: W celu uzyskania poprawnej symulacji należy w pliku table_dec.vhd dodać LIBRARY altera_mf.

Komponent wygenerowany z dekompozera Demain po konwersji na język VHD z dodaną biblioteką altera_mf.

```
-- 4_top_ tab_ dec [table_dec.vhd]
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY table_dec IS
   PORT (
       i       : IN STD_LOGIC_VECTOR (7 downto  0);
       o       : OUT STD_LOGIC_VECTOR (10 downto  0)
     );
END table_dec;
ARCHITECTURE tab_arch OF table_dec IS
    COMPONENT LCELL
       PORT(
            a_in  : IN STD_LOGIC;
            a_out : OUT STD_LOGIC
       );
    END COMPONENT;
    SIGNAL g0_0, g1_0    : STD_LOGIC;
    SIGNAL block1_in     : STD_LOGIC_VECTOR(1 TO 4);
    SIGNAL block2_in     : STD_LOGIC_VECTOR(1 TO 4);
```



```
 SIGNAL block3_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block4_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block5_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block6_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block7_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block8_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block9_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block10_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block11_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block12_in     : STD_LOGIC_VECTOR(1 TO 4);
 SIGNAL block13_in     : STD_LOGIC_VECTOR(1 TO 4);

 SIGNAL block1_out     : STD_LOGIC;
 SIGNAL block2_out     : STD_LOGIC;
 SIGNAL block3_out     : STD_LOGIC;
 SIGNAL block4_out     : STD_LOGIC;
 SIGNAL block5_out     : STD_LOGIC;
 SIGNAL block6_out     : STD_LOGIC;
 SIGNAL block7_out     : STD_LOGIC;
 SIGNAL block8_out     : STD_LOGIC;
 SIGNAL block9_out     : STD_LOGIC;
 SIGNAL block10_out     : STD_LOGIC;
 SIGNAL block11_out     : STD_LOGIC;
 SIGNAL block12_out     : STD_LOGIC;
 SIGNAL block13_out     : STD_LOGIC;

BEGIN
    block1_in <= i(7) & i(5) & i(4) & i(3);
    lc1: LCELL PORT MAP (a_in => block1_out, a_out => g0_0);
    block1: PROCESS (block1_in)
        BEGIN
           CASE block1_in IS
              WHEN "0000" => block1_out <= '0';
              WHEN "0001" => block1_out <= '0';
              WHEN "0010" => block1_out <= '0';
              WHEN "0011" => block1_out <= '1';
              WHEN "0100" => block1_out <= '0';
              WHEN "0101" => block1_out <= '0';
              WHEN "0110" => block1_out <= '1';
              WHEN "0111" => block1_out <= '1';
              WHEN "1000" => block1_out <= '0';
              WHEN "1001" => block1_out <= '1';
              WHEN "1010" => block1_out <= '0';
              WHEN "1011" => block1_out <= '1';
              WHEN "1100" => block1_out <= '0';
              WHEN "1101" => block1_out <= '1';
              WHEN "1110" => block1_out <= '1';
              WHEN "1111" => block1_out <= '1';
              WHEN OTHERS => block1_out <= '0';
           END CASE;
    END PROCESS block1;
```

```
    block2_in <= i(2) & i(1) & i(0) & g0_0;
    lc2: LCELL PORT MAP (a_in => block2_out, a_out => o(10) );
    block2: PROCESS (block2_in)
        BEGIN
           CASE block2_in IS
              WHEN "0000" => block2_out <= '0';
              WHEN "0001" => block2_out <= '1';
              WHEN "0010" => block2_out <= '0';
              WHEN "0011" => block2_out <= '0';
              WHEN "0100" => block2_out <= '0';
              WHEN "0101" => block2_out <= '0';
              WHEN "0110" => block2_out <= '1';
              WHEN "1000" => block2_out <= '0';
              WHEN "1001" => block2_out <= '1';
              WHEN "1100" => block2_out <= '0';
              WHEN OTHERS => block2_out <= '0';
           END CASE;
    END PROCESS block2;
```

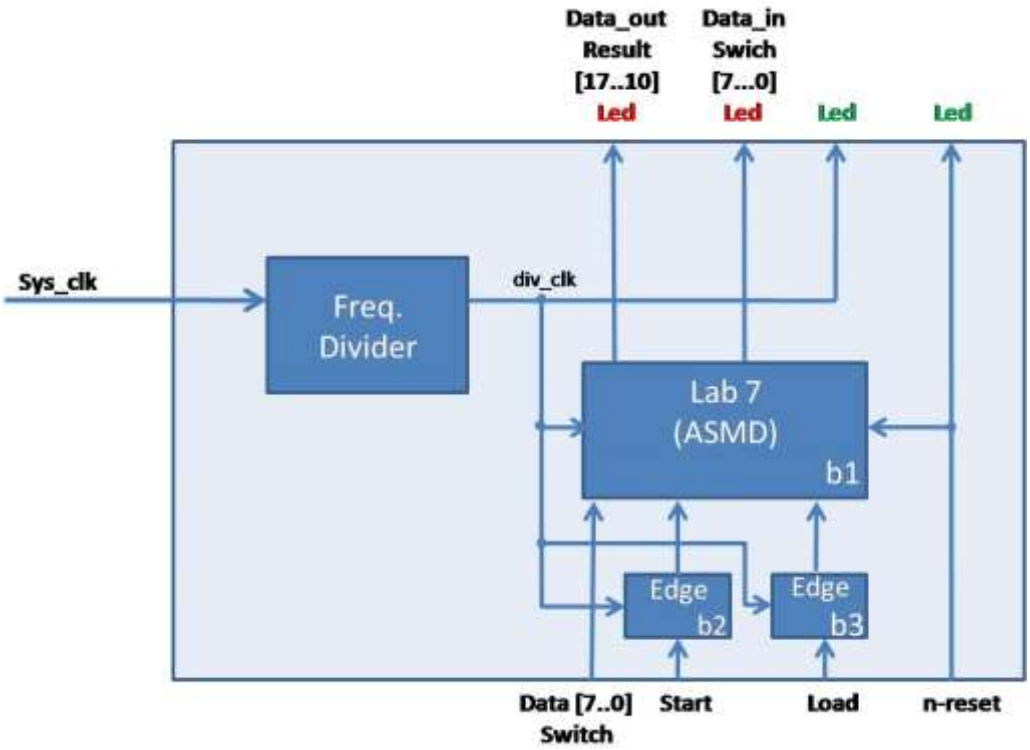
• • •

```
    block13_in <= i(2) & i(1) & i(0) & g1_0;
    lc13: LCELL PORT MAP (a_in => block13_out, a_out => o(7) );
    block13: PROCESS (block13_in)
        BEGIN
           CASE block13_in IS
              WHEN "0000" => block13_out <= '1';
              WHEN "0001" => block13_out <= '0';
              WHEN "0010" => block13_out <= '1';
              WHEN "0011" => block13_out <= '1';
              WHEN "0100" => block13_out <= '1';
              WHEN "0101" => block13_out <= '0';
              WHEN "0110" => block13_out <= '0';
              WHEN "0111" => block13_out <= '0';
              WHEN "1000" => block13_out <= '0';
              WHEN "1001" => block13_out <= '0';
              WHEN "1100" => block13_out <= '0';
              WHEN OTHERS => block13_out <= '0';
           END CASE;
    END PROCESS block13;
END tab_arch;
```

IV Realizacja na zestawie uruchomieniowym DE2-115 powrót

1. Schemat implementacji sprzętowej powrót

Każdy z etapów 1-4 w punkcie III laboratorium należy zrealizować w postaci fizycznej wykorzystując zestaw uruchomieniowy DE2-115 (rys.5) i zweryfikować poprawność działania. Projekt jest realizowany na płytkach uruchomieniowych DE2-115 z układem FPGA Cyclone IV EP4CE115.



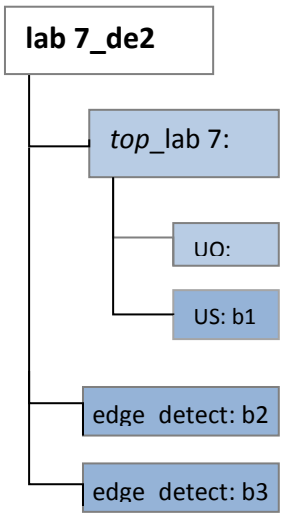
Rys.7. Implementacja systemu na platformie DE2

Przyjęto założenia dotyczące sterowania układu:

- Wejście danych *data*: przełączniki *sw7..sw0*
- Wejście sterujące *start*: przycisk *key3*
- Wejście sterujące *load*: przycisk *key2*
- Wejście sygnału *reset*: przycisk *key0*
- Wyjście danych wyniku *result*: diody *ledr17..ledr10*
- Wyjście pokazujące status danych wejściowych *data*: diody *ledr7..ledr0*
- Wyjście pokazujące status sygnału *ready*: dioda *ledg7*
- Wyjście pokazujące status sygnału *reset*: dioda *ledg0*
- Wyjście pokazujące status sygnału *div_clk*: dioda *ledg1*

Należy zauważyć, że przyciski *key* w stanie zwolnionym mają wartość logiczną 1, natomiast po wciśnięciu mają wartość logiczną 0 (patrz: dokumentacja DE2-115). Moment przyciśnięcia jest sygnalizowany wygenerowaniem pojedynczego sygnału na wyjściu bloku *edge_detect*. Sygnał *sys_clk* został przypisany do sygnału zegarowego o częstotliwości 50 MHz, który następnie jest dzielony a uzyskany sygnał *div_clk* steruje pozostałymi blokami układu.

2. Realizacja projektu lab7_de2 powrót

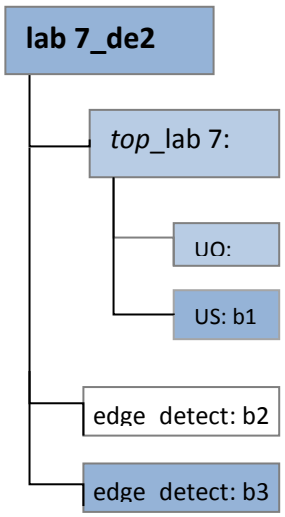



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lab7_de2 is
      generic(
            d : integer := 24
            -- dlugosc podzielnika czestotltiwosci
            );
      port
      (
            sys_clk, n_reset  : in  std_logic;
            start, load       : in  std_logic;
            data              : in  std_logic_vector(9 downto 0);

            data_out    : out std_logic_vector(9 downto 0);
            result      : out std_logic_vector(9 downto 0);
            ready       : out std_logic;
            div_clk_out : out std_logic;
            reset_out   : out std_logic
      );
end lab7_de2;

architecture arch_lab7 of lab7_de2 is
      signal cnt_reg, cnt_next  : unsigned(d-1 downto 0);
      signal div_clk            : std_logic;
      signal start_tick         : std_logic;
      signal load_tick          : std_logic;
component top_lab7 is
      port
      (
            clk, rst    : in  std_logic;
            start, load : in  std_logic;
            data        : in  std_logic_vector(9 downto 0);
            result      : out std_logic_vector(9 downto 0);
            ready       : out std_logic
      );
end component;


component edge_detect is
      port(
            clk, reset  : in std_logic;
            level       : in std_logic;
            tick        : out std_logic
      );
end component;
```

```
begin
	-- licznik
	process (sys_clk, n_reset)
	begin
		if n_reset = '0' then
			cnt_reg <=(others => '0');
		elsif rising_edge(sys_clk) then
			cnt_reg <= cnt_next;
		end if;
	end process;
		process(cnt_reg)
	begin
		cnt_next <= cnt_reg + 1;
	end process;
		-- dzielnik czestotliwosci
	div_clk <= cnt_reg(d-1);

	-- obserwacja danych na diodach led
	data_out    <= data;
	reset_out   <= n_reset;
	div_clk_out <= div_clk;

b1: top_lab7   -- project z laboratorium 7
	port map (clk => div_clk, rst => not(n_reset), start => start_tick, load =>
load_tick, data => data,
		result => result, ready => ready);
b2: edge_detect port map(clk => div_clk, reset => not(n_reset), level => not(start),
tick => start_tick);
b3: edge_detect port map(clk => div_clk, reset => not(n_reset), level => not(load),
tick => load_tick);

end arch_lab7;
```

3. Realizacja detektora zbocza sygnału powrót

```
-- z ksiazki:"Fpga prototyping by VHDL examples" Pong P.Chu
library ieee;
use ieee.std_logic_1164.all;
entity edge_detect is
	port(
		clk, reset : in std_logic;
		level : in std_logic;
		tick : out std_logic
	);
end edge_detect;

architecture edge_detect_arch of edge_detect is
	type state_type is (zero, edge, one);
	signal state_reg, state_next : state_type;
begin
	process (clk, reset)
	begin
		if reset = '1' then
			state_reg <= zero;
		elsif rising_edge(clk) then
			state_reg <= state_next;
```



```
            end if;
      end process;

      process(state_reg, level)
      begin
            state_next <= state_reg;
            tick <= '0';
            case state_reg is
                  when zero =>
                        if level='1' then
                              state_next <= edge;
                        end if;
                  when edge =>
                        tick <= '1';
                        if level = '1' then
                              state_next <= one;
                        else
                              state_next <= zero;
                        end if;
                  when one =>
                        if level='0' then
                              state_next <= zero;
                        end if;
            end case;
      end process;
end edge_detect_arch;
```

4. Lista przypisań sygnałów do nóżek układu (patrz: dokumentacja DE2) powrót

W celu umożliwienia poprawnej realizacji sprzętowej projektu należy w końcowej kompilacji przypisać odpowiednie końcówki układu FPGA sygnałom we/wyj. Należy w tym celu użyć w Quartus II narzędzia *Assignments>Pin Planner (*plik *top_lab7.qsf)*. Poniżej lista przypisań (dokumentacja DE2-115).

```
set_location_assignment PIN_AB28 -to data[0]
set_location_assignment PIN_AC28 -to data[1]
set_location_assignment PIN_AC27 -to data[2]
set_location_assignment PIN_AD27 -to data[3]
set_location_assignment PIN_AB27 -to data[4]
set_location_assignment PIN_AC26 -to data[5]
set_location_assignment PIN_AD26 -to data[6]
set_location_assignment PIN_AB26 -to data[7]
set_location_assignment PIN_N21 -to load
set_location_assignment PIN_G21 -to ready
set_location_assignment PIN_J15 -to result[0]
set_location_assignment PIN_H16 -to result[1]
set_location_assignment PIN_J16 -to result[2]
set_location_assignment PIN_H17 -to result[3]
set_location_assignment PIN_F15 -to result[4]
set_location_assignment PIN_G15 -to result[5]
set_location_assignment PIN_G16 -to result[6]
set_location_assignment PIN_H15 -to result[7]
set_location_assignment PIN_R24 -to start
set_location_assignment PIN_Y2 -to sys_clk
set_location_assignment PIN_E21 -to reset_out
set_location_assignment PIN_E22 -to div_clk_out
set_location_assignment PIN_M23 -to n_reset
set_location_assignment PIN_G19 -to data_out[0]
set_location_assignment PIN_F19 -to data_out[1]
set_location_assignment PIN_E19 -to data_out[2]
```

```
set_location_assignment PIN_F21 -to data_out[3]
set_location_assignment PIN_F18 -to data_out[4]
set_location_assignment PIN_E18 -to data_out[5]
set_location_assignment PIN_J19 -to data_out[6]
set_location_assignment PIN_H19 -to data_out[7]
```

Literatura i materiały pomocnicze powrót

1. Dokument PDF: „DE2-115 User Manual”
2. Materiały do UCYF laboratorium 7
3. Plansze do wykładu UCYF
4. Instrukcja obsługi programu Demain i DemainToVHD,
   T. Łuba, D. Ojrzeńska-Wójter: *Układy logiczne w zadaniach*, OWPW, 2011
5. Literatura podana na wykładzie, ze szczególnym uwzględnieniem rozdz. 6 książki:
   T. Łuba, pr. zbiorowa: *Programowalne układy przetwarzania sygnałów i informacji*, WKŁ

Opracowanie wewnętrzne ZCB IT, PW, styczeń 2016;
mgr inż. Danuta Ojrzeńska-Wójter