Układy Cyfrowe – laboratorium

Przykład realizacji ćwiczenia nr 5 (wersja 2015)

1. Temat

Realizacja iteracyjnego algorytmu wyznaczania przybliżonej wartości pierwiastka funkcji. Metoda Newtona, zwana również metodą Newtona-Raphsona lub metodą stycznych, polega na przybliżaniu w kolejnych krokach algorytmu miejsca zerowego.

Metoda Newtona przyjmuje następujące założenia dla funkcji f:

1. W przedziale $[a, b]$ znajduje się dokładnie jeden pierwiastek.
2. Funkcja ma różne znaki na krańcach przedziału, tj. $f(a) \cdot f(b) < 0$.
3. Pierwsza i druga pochodna funkcji mają stały znak w tym przedziale.

W pierwszym kroku metody wybierany jest punkt startowy x_0 (zazwyczaj jest to wartość $a, b, 0 \text{ lub } 1$, z którego następnie wyprowadzana jest styczna w $f(x_0)$. Odcięta punktu przecięcia stycznej z osią OX jest pierwszym przybliżeniem rozwiązania (ozn. x_1). Jeśli to przybliżenie nie jest satysfakcjonujące, wówczas punkt x_1 jest wybierany jako nowy punkt startowy i wszystkie czynności są powtarzane. Proces jest kontynuowany, aż zostanie uzyskane wystarczająco dobre przybliżenie pierwiastka. Kolejne przybliżenia są dane rekurencyjnym wzorem: $x_{k+1} = x_k - \frac{f(x_k)}{f\prime(x_k)}$.

2. Zadanie

Znaleźć pierwiastek trzeciego stopnia liczby 123 z wykorzystaniem metody Newtona. Punkt startowy oraz wynik ma być zapisany 8-bitową liczbą ze znakiem. Zakładamy 10 kroków iteracji.

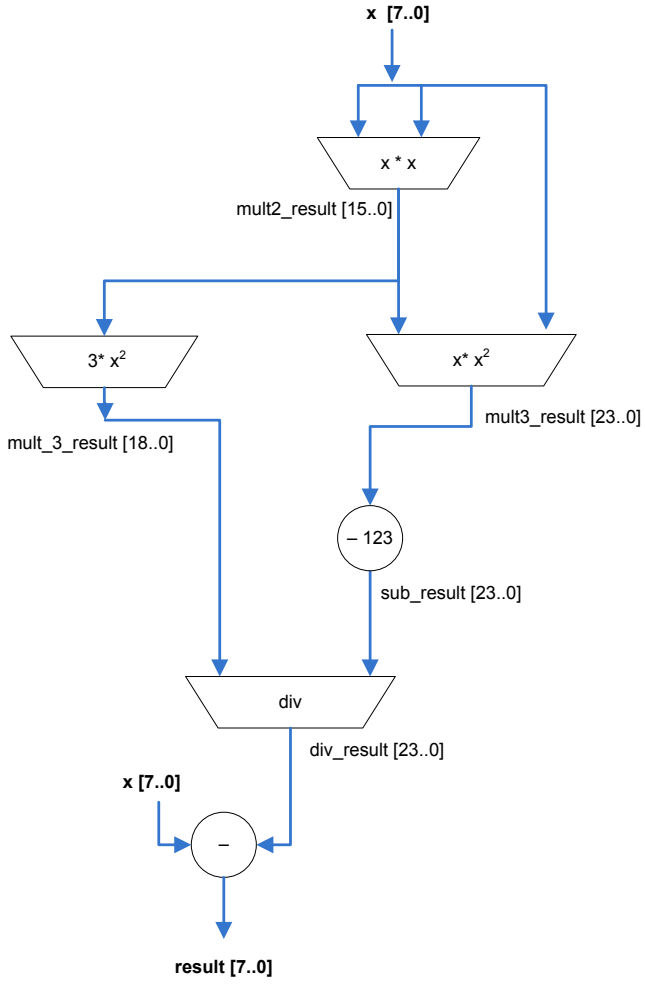
3. Przykładowe rozwiązanie

Pierwiastek 3. stopnia z liczby 123 można obliczyć znajdując miejsce zerowe następującej funkcji $f(x) = x^3 - 123$. Pochodna tej funkcji jest $f'(x) = 3x^2$. Stąd szukane równanie ma postać $x_1 = x_0 - \frac{x_0^3 - 123}{3x_0^2}$.

3.1. Funkcja $f_{next}(x)$

Należy zrealizować funkcję kombinacyjną $f_{next}(x) = x_0 - \frac{x_0^3 - 123}{3x_0^2}$ obliczającą następne przybliżenie miejsca zerowego. Z powyższego wzoru mamy do realizacji operacje mnożenia, dzielenia i odejmowania, pamiętając o kolejności wykonywanych operacji oraz o szerokościach ścieżek danych w kolejnych krokach.

Realizujemy kolejno: mnożenie $x * x$, mnożenie $x * x * x$ czyli potęgę 3. stopnia, odejmowanie $x^3 - 123$, mnożenie $3 * x^2$, dzielenie $(x^3 - 123)/3 * x^2$ i odejmowanie $x - (x^3 - 123)/3 * x^2$. Diagram działań przedstawia rysunek:



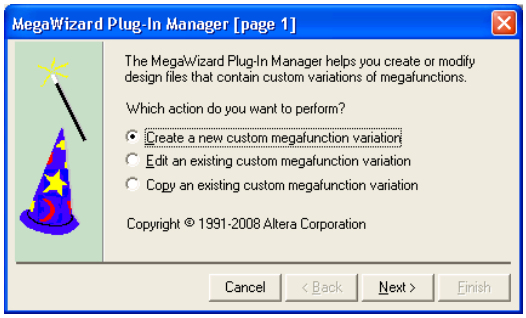
Rys. 1. Funkcja $f_{next}(x)$

W wyniku mnożenia dwóch liczb 8-bitowych otrzymany iloczyn ma długość 16-bitów. W wyniku mnożenia przez stałą +3, zwiększamy długość wyniku o 3 bity. Ponieważ założenie projektowe jest, aby wynik był 8-bitowy, należy na wyjście funkcji przekazać tylko 8 bitów z 24-bitowego wektora ilorazu.

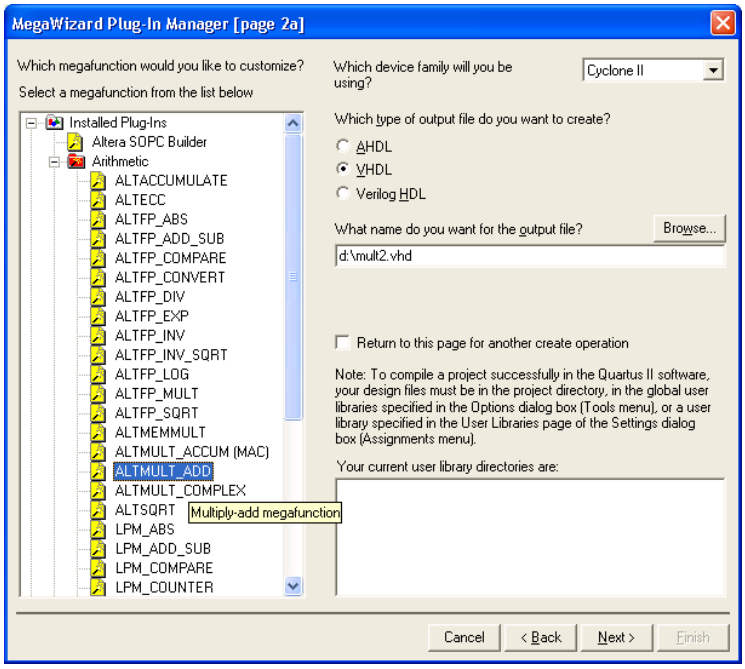
W realizacji funkcji arytmetycznych należy posłużyć się funkcjami bibliotecznymi systemu Quartus2. Założeniem projektowym jest, że funkcja $f_{next}(x)$ jest blokiem kombinacyjnym, dlatego wszystkie użyte funkcje składowe mają być blokami kombinacyjnymi.

Realizacja potęgowania x^2 z użyciem funkcji *alt_mult_add* składa się z następujących kroków:

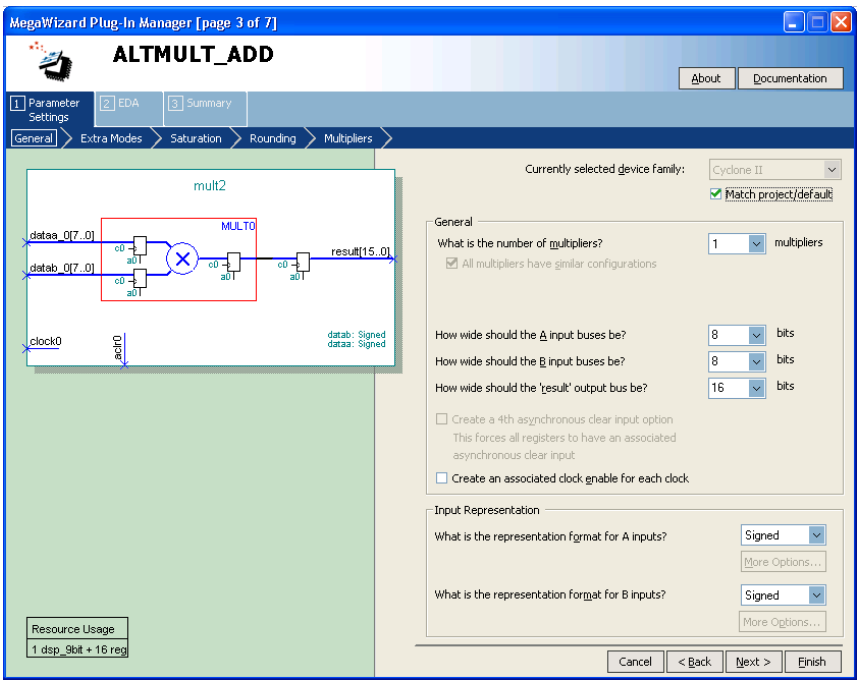
- Tworzymy nową instancję wybierając z narzędzi Quartus2 *Tools* > *Megawizard Plugin Manager*



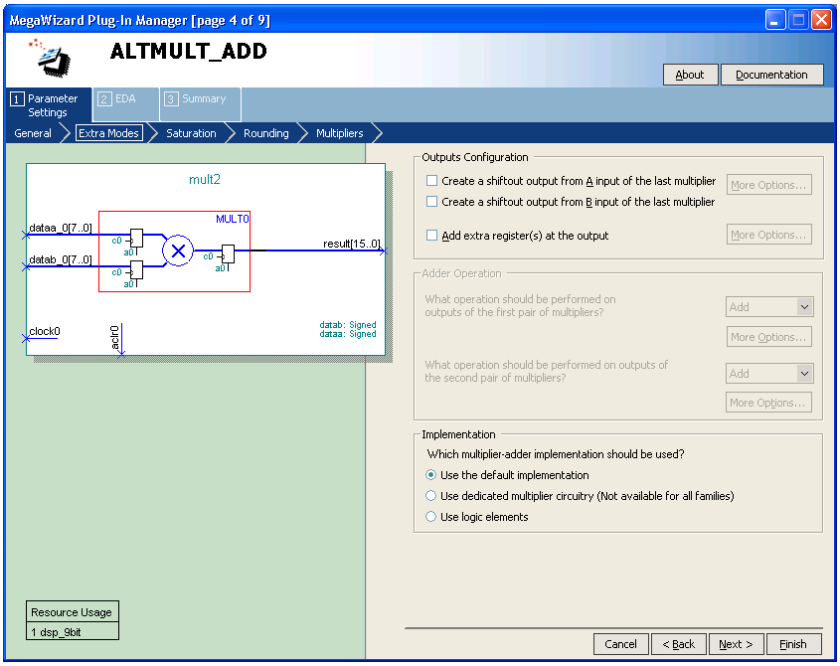
- Wybieramy biblioteczną funkcję mnożenia *altmult_add*, podajemy nazwę instancji *mult2.vhd* (można użyć funkcji *lpm_mult*, UWAGA! nie można użyć trybu potęgowania w funkcji LPM_MULT ponieważ źle symuluje w ModelSim „*Multiply dataa input by itself (squaring operation))*”)



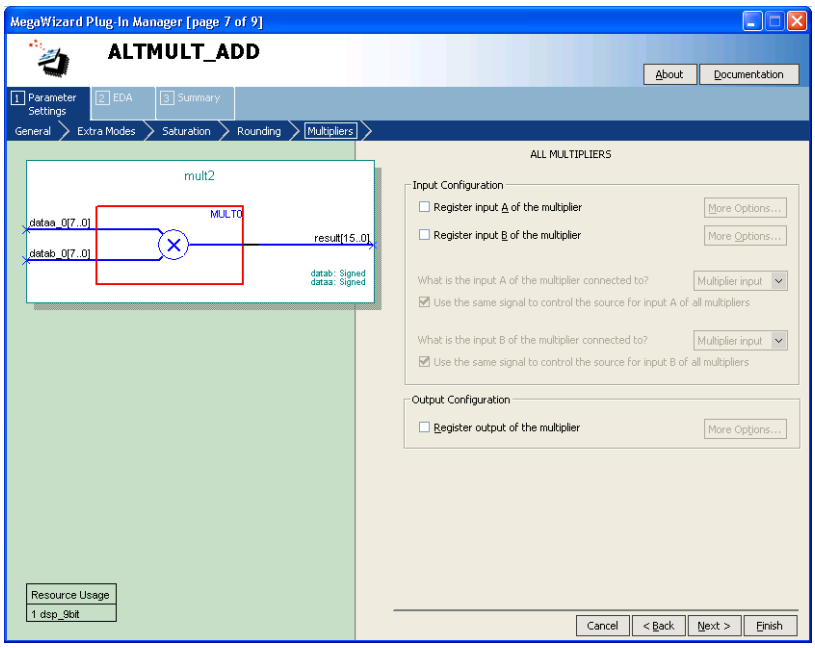
- Ustawiamy szerokość sygnałów wejściowych na 8 bitów i reprezentację liczb ze znakiem *signed*, sygnał wyjściowy został automatycznie ustawiony na 16 bitów



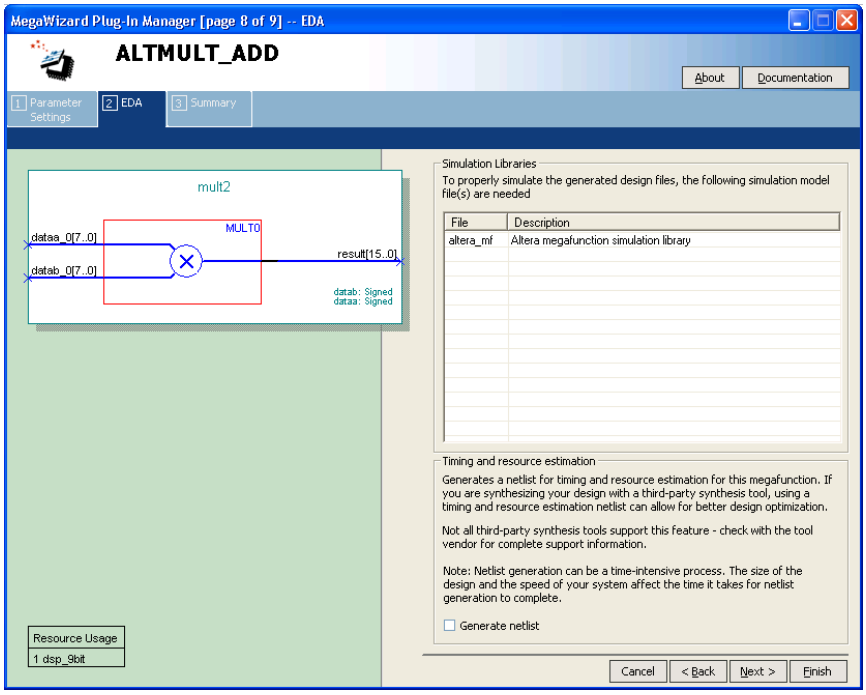
- Wyłączamy dodatkowe rejestry wyjściowe



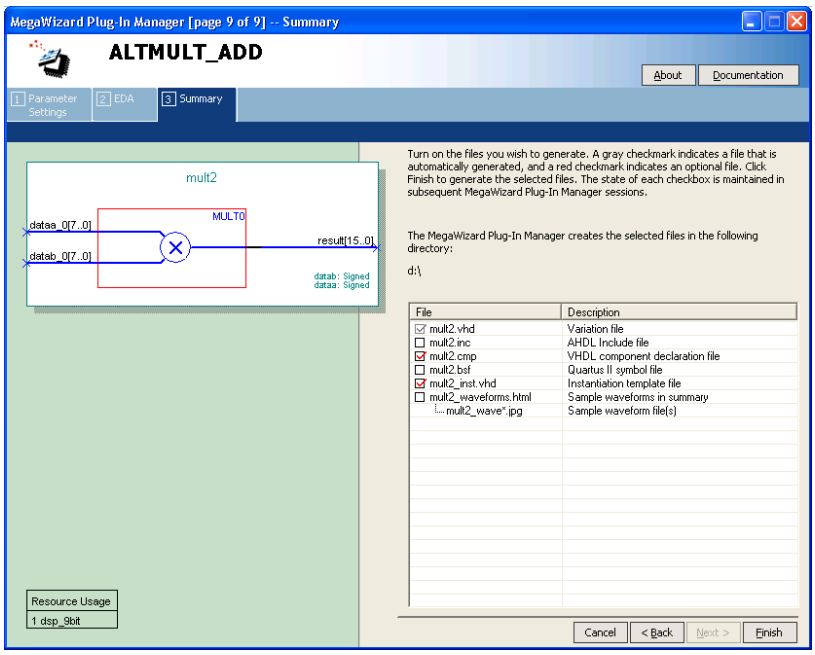
- Strona 5. i 6. bez zmian
- Wyłączamy rejestry na wejściu i wyjściu bloku mnożącego



- Informacje na temat bibliotek, w pliku *fnext.hvd* należy dodać bibliotekę *altera_mf*



- Zaznaczamy pola do wygenerowania plików z szablonem instancji *mult2_inst.vhd* oraz z komponentem użytej funkcji *mult2.cmp*, pliki te posłużą do skopiowania zawartości w odpowiednich sekcjach



Analogicznie tworzymy pozostałe bloki arytmetyczne.

Realizację bloku funkcji $f_{next}(x)$ przedstawia wydruk *fnext.vhd*:

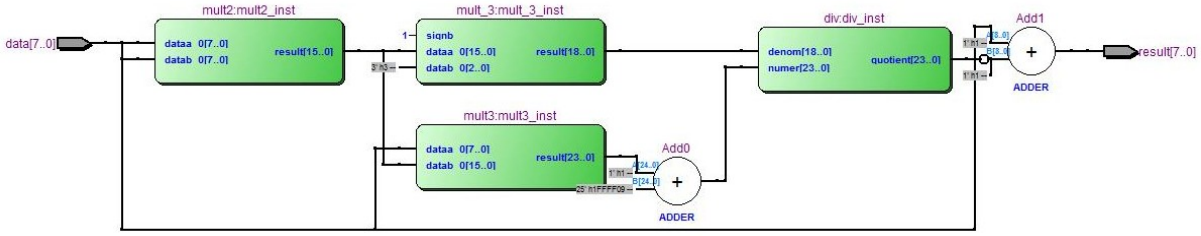
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Altera Megafunctions
library altera_mf;
use altera_mf.altera_mf_components.all;
library lpm;
use lpm.all;

entity fnext is
	port
	(
		data   : in  std_logic_vector(7 downto 0);
		result : out std_logic_vector(7 downto 0)
	);
end fnext;

architecture arch_fnext of fnext is

	constant sqrt3      : integer := 123;

	-- sygnaly blokow operacyjnych
	-- wymagane do polaczenia
	signal mult2_result : std_logic_vector(15 downto 0);
	signal mult3_result : std_logic_vector(23 downto 0);
	signal mult_3_result : std_logic_vector(18 downto 0);
	signal div_result : std_logic_vector(23 downto 0);
	signal sub_result : std_logic_vector(23 downto 0);
	-- funkcje zewnetrzne
	component mult2
		PORT
		(
			dataa_0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
```

```
                datab_0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
                result : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
        );
    end component;
    component mult3
        PORT
        (
                dataa_0 : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
                datab_0 : IN STD_LOGIC_VECTOR (15 DOWNTO 0) :=  (OTHERS => '0');
                result : OUT STD_LOGIC_VECTOR (23 DOWNTO 0)
        );
    end component;
    component mult_3
        PORT
        (
                dataa_0 : IN STD_LOGIC_VECTOR (15 DOWNTO 0) :=  (OTHERS => '0');
                datab_0 : IN STD_LOGIC_VECTOR (2 DOWNTO 0) :=  (OTHERS => '0');
                signb : IN STD_LOGIC  := '0';
                result : OUT STD_LOGIC_VECTOR (18 DOWNTO 0)
        );
    end component;
    component div
        PORT
        (
                denom : IN STD_LOGIC_VECTOR (18 DOWNTO 0);
                numer : IN STD_LOGIC_VECTOR (23 DOWNTO 0);
                quotient : OUT STD_LOGIC_VECTOR (23 DOWNTO 0);
                remain : OUT STD_LOGIC_VECTOR (18 DOWNTO 0)
        );
    end component;

begin
    mult2_inst : mult2 PORT MAP (
        dataa_0 => data,
        datab_0 => data,
        result => mult2_result
    );

    mult3_inst : mult3 PORT MAP (
        dataa_0 => data,
        datab_0 => mult2_result,
        result => mult3_result
    );

    mult_3_inst : mult_3 PORT MAP (
        dataa_0 => mult2_result,
        datab_0 => "011",
        signb => '1', -- 1 dla signed, 0 dla unsigned
        result => mult_3_result
    );

    sub_result <= std_logic_vector(signed(mult3_result) - to_signed(sqrt3, 24));

    div_inst : div PORT MAP (
        denom => mult_3_result,
        numer => sub_result,
        quotient => div_result --,
        --remain      => remain_sig
    );

    result <= std_logic_vector(signed(data) - signed(div_result(7 downto 0)));

end arch_fnext;
```

Kompilacja i weryfikacja

Projekt jest realizowany na płytkach uruchomieniowych DE2-115 z układem fpga CycloneIVE EP4CE115F29C7, dla którego należy ustawić projekt (ustawiamy wersje na *Revision fnext*).

Schemat blokowy (rys. 2) można uzyskać za pomocą narzędzia *Tools>Netlist Viewers>RTL Viewer*, który obrazuje diagram z rysunku 1.



Rys. 2. Schemat blokowy funkcji $f_{next}(x)$ po kompilacji

Weryfikacja poprawności działania bloku $f_{next}(x)$ jest przeprowadzona na drodze symulacji, wywołując ModelSim z programu Quartus: *Tools>Run EDA Simulation Tool >EDA RTL Simulation*. Uruchamiamy symulację dla *fnext* z biblioteki *rtl_work* dwukrotnie na nią klikając lub wykonując komendę:

ModelSim> vsim rtl_work.fnext

Wektory testowe można podać wykonując przykładowy skrypt *test_fnext.do*, który został utworzony (lub skopiowany) w podkatalogu *\simulation\modelsim* i wykonany za pomocą komendy:

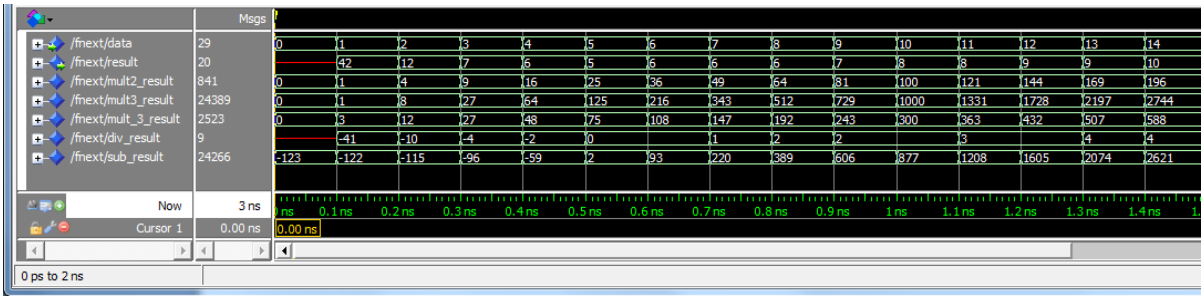
VSIM> do test_fnext.do

Skrypt testu przedstawia poniższy wydruk:

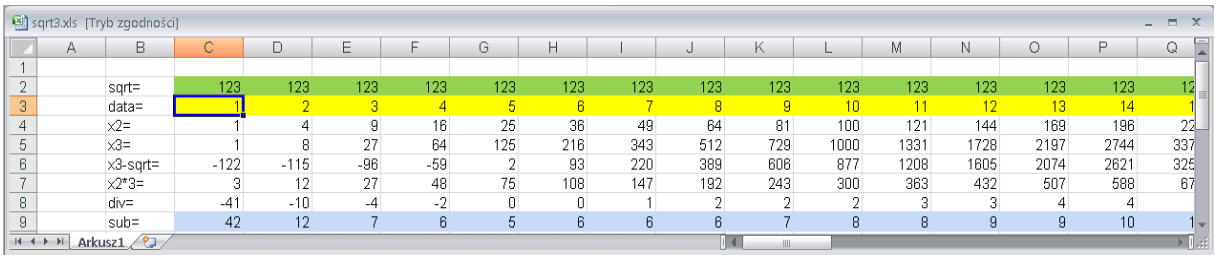
```
restart -nowave
add wave -radix decimal *
force data 10#0 0
run 100
force data 10#1 0
run 100
force data 10#2 0
run 100
force data 10#3 0
run 100
force data 10#4 0
run 100
force data 10#5 0
run 100
force data 10#6 0
run 100
force data 10#7 0
run 100
force data 10#8 0
```

```
run 100
force data 10#9 0
run 100
force data 10#10 0
run 100
# … i tak dalej dla kolejnych wektorow testowych…
```

Przykładowy widok wyniku testu:

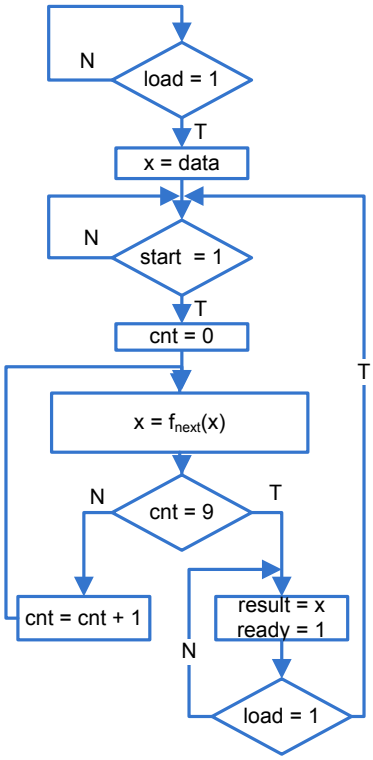


Weryfikację poprawności działania funkcji $f_{next}(x)$ można także dokonać z wykorzystaniem formularza np. MSExcel:



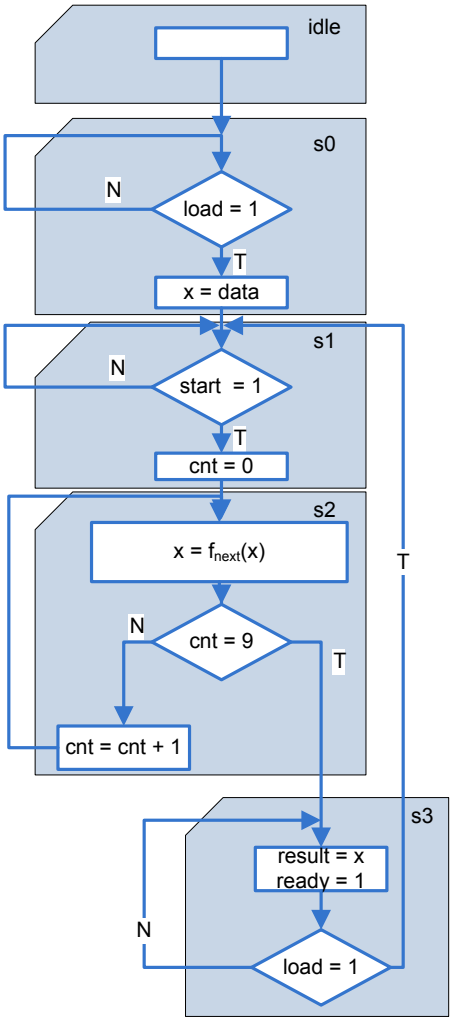
3.2. Realizacja bloku ASMD

Diagram algorytmu przedstawiony jest na rysunku:



Rys. 3. Diagram algorytmu realizujący metodę Newtona

Wykorzystują metodę ASMD dzielimy algorytm na bloki ASM (rys. 4).



Rys. 4. Algorytm podzielony na bloki ASM

Przykładowa realizacja algorytmu podana jest na wydruku:

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

entity lab5 is
	port
	(
		clk, reset   : in  std_logic;
		start, load  : in  std_logic;
		data   : in  std_logic_vector(7 downto 0);

		result : out std_logic_vector(7 downto 0);
		ready  : out std_logic
	);
end lab5;

architecture arch_lab5 of lab5 is
	type state_type is (idle, s0, s1, s2, s3);
	signal state_reg, state_next : state_type;

	signal cnt_reg, cnt_next : unsigned(3 downto 0);
	signal in_res_reg, in_res_next : signed(7 downto 0);
	signal acc_reg, acc_next : signed(7 downto 0);
	signal ready_reg, ready_next : std_logic;
```

```
	signal fnext_result : std_logic_vector(7 downto 0);

	component fnext
		port
		(
			data   : in  std_logic_vector(7 downto 0);
			result : out std_logic_vector(7 downto 0)
		);
	end component;
begin
	-- automat
	process (clk, reset)
	begin
		if reset = '1' then
			state_reg <= idle;
		elsif rising_edge(clk) then
			state_reg <= state_next;
		end if;
	end process;

	-- licznik
	process (clk, reset)
	begin
		if reset = '1' then
			cnt_reg <=(others => '0');
		elsif rising_edge(clk) then
			cnt_reg <= cnt_next;
		end if;
	end process;

	-- akumulator i rejestry
	process (clk, reset)
	begin
		if reset = '1' then
			acc_reg <=(others => '0');
		elsif rising_edge(clk) then
			acc_reg <= acc_next;
		end if;
	end process;

	-- rejestr wejsciowy
	process (clk, reset)
	begin
		if reset = '1' then
			in_res_reg <= (others => '0');
		elsif rising_edge(clk) then
			in_res_reg <= in_res_next;
		end if;
	end process;

	-- flaga
	process (clk, reset)
	begin
		if reset = '1' then
			ready_reg <= '0';
		elsif rising_edge(clk) then
			ready_reg <= ready_next;
		end if;
	end process;

	-- tablica przejsc automatu
	process(state_reg, load, start, cnt_reg)
	begin
		state_next <= state_reg; --domyslny
```

```
		case state_reg is
			when idle =>
				state_next <= s0;
			when s0 => -- ustawienie danych
				if load = '1' then
					state_next <= s1;
				end if;
			when s1 => -- załadowanie i start obliczeń
				if start = '1' then
					state_next <= s2;
				end if;
			when s2 =>  -- obliczenia
				if cnt_reg = "1001" then -- liczba iteracji
					state_next <= s3;
				end if;
			when s3 => -- prezentacja wyniku, załadowanie nowych danych
				if load = '1' then
					state_next <= s1;
				end if;
			when others =>
				state_next <= idle;
		end case;
	end process;

	-- sygnaly wyjsciowe automatu Moore'a
	process(state_reg, acc_reg, cnt_reg, in_res_reg, start, load, data,
ready_reg, fnext_result)
	begin
		acc_next <= acc_reg; -- wartosci
		cnt_next <= cnt_reg; -- domyslne
		in_res_next <= in_res_reg;
		ready_next <= '0';
		case state_reg is
			when idle =>
			when s0 => -- ustawienie danych
				if load = '1' then
					in_res_next <= signed(data);
				end if;
			when s1 => -- załadowanie i start obliczeń
				if start = '1' then
					acc_next <= in_res_reg;
					cnt_next <= (others => '0');
				end if;
			when s2 => -- obliczenia
				cnt_next <= cnt_reg + 1;
				acc_next <= signed(fnext_result);
				if cnt_reg = "1001" then -- liczba iteracji
					ready_next <= '1';
				end if;
			when s3 => -- prezentacja wyniku, załadowanie nowych danych
				ready_next <= ready_reg;
				if load = '1' then
					in_res_next <= signed(data);
					ready_next <= '0';
				end if;
			when others =>
		end case;
	end process;

	-- sygnaly wyjsciowe ukladu
	ready <= ready_reg;
	result <= std_logic_vector(acc_reg);

	b1: fnext port map
```

```
            (
                  data => std_logic_vector(acc_reg),
                  result => fnext_result
            );
end arch_lab5;
```

Kompilacja i weryfikacja

Projekt należy stworzyć dla układu CycloneIVE EP4CE115F29C7 (ustawiamy wersje na *Revision lab5*).

Uruchamiamy symulację dla *lab5* z biblioteki *rtl_work* dwukrotnie na nią klikając lub wykonując komendę:

ModelSim> vsim rtl_work.lab5

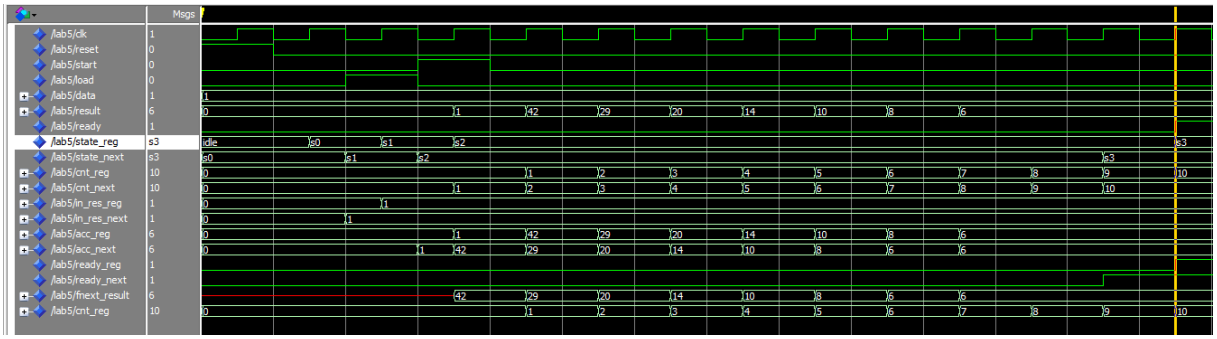
Wektory testowe można podać wykonując przykładowy skrypt *test_lab5.do*, który został utworzony (lub skopiowany) w podkatalogu *\simulation\modelsim* i wykonany za pomocą komendy:

VSIM> do test_lab5.do

Skrypt testu przedstawia poniższy wydruk:

```
restart -nowave
add wave -radix decimal *
add wave -radix unsigned cnt_reg
add wave -radix unsigned cnt_next
force clk 0 0, 1 {50 ps} -r 100
force reset 1 0, 0 100
force load 0 0, 1 200, 0 300
force start 0 0, 1 300, 0 400
force data 10#1 0
run 1400
```

Przykładowy widok wyniku testu:



3.3. Realizacja na zestawie uruchomieniowym 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*

Projekt należy stworzyć dla układu CycloneIVE EP4CE115F29C7 (ustawiamy wersje na *Revision top_lab5*).

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.

Realizacja projektu pokazana jest na wydruku:

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

entity top_lab5 is
	generic(
		d : integer := 25 -- dlugosc podzielnika czestotltiwosci
		);
	port
	(
		sys_clk, n_reset    : in  std_logic;
		start, load  : in  std_logic;
		data   : in  std_logic_vector(7 downto 0);

		data_out : out std_logic_vector(7 downto 0);
		result : out std_logic_vector(7 downto 0);
		ready  : out std_logic;
		div_clk_out : out std_logic;
		reset_out : out std_logic
	);
end top_lab5;

architecture arch_lab5 of top_lab5 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 lab5 is
	port
	(
		clk, reset   : in  std_logic;
		start, load  : in  std_logic;
		data   : in  std_logic_vector(7 downto 0);
		result : out std_logic_vector(7 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: lab5
	port map (clk => div_clk, reset => 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_lab5;
```

Realizacja detektora zbocza sygnału podana jest na wydruku:

```
-- 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;
```

Po kompilacji należy rozlokować sygnały na płytce za pomocą narzędzia *Assignments>Pin Planner*. Lista przypisań sygnałów do nóżek układu (patrz: dokumentacja DE2-115) z pliku *top_lab5.qsf*:

```
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:

1. Plansze do wykładu UCYF
2. Literatura podana na wykładzie, ze szczególnym uwzględnieniem rozdz. 6 książki „Programowalne układy przetwarzania sygnałów i informacji”
3. Dokument PDF: „DE2-115 User Manual”

Opracowanie wewnętrzne ZCB IT, PW, grudzień 2015:
dr inż. P. Tomaszewicz