Zrób sobie procesor: PicoBlaze w FPGA, część 4

Pierwszy projekt

Pierwszy przykładowy projekt to będzie aplikacja typu „Witaj Świecie" – w naszym przypadku miganie diodą LED. Projekt zostanie uruchomiony na płytce ewaluacyjnej z układem FPGA z rodziny Spartan3.

Do wykonania projektu będą nam potrzebne następujące elementy:
- płytka z układem FPGA, z rodziny Spartan3, z podłączonym sygnałem zegarowym i z podłączoną diodą LED (do testów użyto zestawu ZL10PLD i ZL9PLD).
- mikrokontroler PicoBlaze opisany w języku HDL.
- program w asemblerze włączający i wyłączający diody LED.
- kompliator *kcpsm3.exe* służacy do wygenerowania modułu pamięci ROM, na podstawie przygotowanego programu w asemblerze.
- program ISE WebPACK do zaimplementowania projektu, opisanego w jednym z języków VHDL lub Verilog (jego najnowszą wersję publikujemy na CD–EP7/2008A).
- programator JTAG do układów FPGA/CPLD firmy Xilinx.

Nasz projekt ma migać diodą LED w taki sposób, żeby było to zauważalne dla ludzkiego oka. Przyjmijmy, że dioda ma mrugać z częstotliwością 1 Hz. Oznacza to, że przez pół sekundy będzie ona świecić i drugie pół sekundy będzie zgaszona. Żeby zrealizować to zadanie mając za źródło częstotliwości taktującej generator o częstotliwości 3,6864 MHz (standardowy generator na płytce ZL10PLD) należy ją podzielić w taki sposób, aby uzyskać przebieg o wymaganej częstotliwości – w naszym przypadku potrzebny jest licznik zliczający do 1843200, po jego przepełnieniu stan wyjścia sterującego diodą LED jest zmieniany na przeciwny.

Dzielnik częstotliwości taktującej wykonamy w sprzęcie, dzięki czemu PicoBlaze będzie taktowany sygnałem zegarowym o częstotliwości 576 Hz. Wiedząc że wszystkie instrukcje są wykonywane przez PicoBlaze w ciągu 2 taktów zegarowych, będziemy mogli przygotować bardzo prosty program realizujący zmianę stanu

List. 1. Opis procesu licznika spełniającego w naszym projekcie rolę dzielnika częstotliwości

```
signal clk_p      : std_logic;
signal CLK_licznik: integer range 0 to 3199;

CLK_ustaw: process (clk, rst)
begin
  if (rst = '1') then
   CLK_licznik <= 0;
   clk_p <= '0';
  elsif clk'event and clk='1' then
    if (CLK_licznik >= 3199) then
      CLK_licznik <= 0;
      clk_p <= not clk_p;
    else
      CLK_licznik <= CLK_licznik + 1;
    end if;
  end if;
end process;
```

List. 2. Program dla PicoBlaze'a zapalający i gaszący diodę LED

```
;*****************************************************************************
; Stale uzyte w programie
;*****************************************************************************

CONSTANT led_signal       , 01    ; numer diody LED
CONSTANT led_port_nr      , 00    ; numer portu, do ktorego przylaczony jest LED
CONSTANT licznik          , 2F    ; do ilu bedziemy odliczac aby zmienic stan diody LED

;*****************************************************************************
; Rejestry specjalnego przeznaczenia
;*****************************************************************************
;
; s0                ; rejestr uzyty do zliczania impulsow
; s1                ; s1 - rejestr z wartoscia portu - 1
; s2                ; s2 - rejestr z wartoscia portu - 0

;;===========================================================================
;; Nieskonczona petla.
;;===========================================================================

    __start: LOAD s1, 01                ; s1 <= 1
             LOAD s2, 00                ; s2 <= 0

      start: LOAD s0, licznik           ; ustaw licznik
             LOAD sA, sA                ; instrukcja pusta

     petla1: SUB s0, 01                 ; odejmij jeden
             COMPARE s0, 00             ; sprawdz, czy zero
             JUMP NZ, petla1            ; jesli nie skocz do petla1

             OUTPUT s1, led_port_nr     ; ustaw sygnal LED_cntrl na '1'

             LOAD s0, licznik           ; ustaw licznik

     petla2: SUB s0, 01                 ; odejmij jeden
             COMPARE s0, 00             ; sprawdz, czy zero
             JUMP NZ, petla2            ; jesli nie skocz do petla2

             OUTPUT s2, led_port_nr     ; ustaw sygnal LED_cntrl na '0'

             JUMP start                 ; skocz na poczatek programu
```

List. 3. Proces obsługujący wyjście sterujące pracą diody LED

```
LED_ustaw: process (clk_p, rst)
begin
  if (rst = '1') then
    LED_cntrl <= '0';
  elsif clk_p'event and clk_p='1' then
    if ((port_id="00000000") and
                  (write_strobe='1')) then
      LED_cntrl <= out_port(0);
    end if;
  end if;
end process;
```

diody LED dokładnie co pół sekundy. W opisie implementacji naszego projektu trzeba dodać licznik spełniający rolę dzielnika częstotliwości (**list. 1**).

Proces ma dwa sygnały sterujące i dwa sygnały są przez niego ustawiane. Sygnał *clk_p* jest sygnałem wyjściowym, który zmienia się co 3200 impulsów sygnału synchronizującego *clk*. Aby poprawnie odliczyć 3200 impulsów potrzebujemy dodatkowego sygnału, w tym przypadku typu *integer*. Sygnał ten do celów symulacji i syntezy został ograniczony do maksymalnej wartości, jaką może przyjąć licznik. Taki proces po syntezie powinien być reprezentowany jako synchroniczny licznik z 12 przerzutnikami i częścią logiczną dekodującą wartość 3199 dziesiętnie.

A teraz wytłumaczymy skąd się wzięła wartość 3200. Jak już wcześniej wspomniano, układ będziemy uruchamiać na płytce ewaluacyjnej z generatorem o częstotliwości 3,6864 MHz. Chcemy uzyskać częstotliwość 1 Hz, sterowaną przez program PicoBlaze, który bez dodatkowych procedur jest w stanie wykonywać operacje arytmetyczne w zakresie ośmiu bitów. Nasz licznik generuje sygnał *clk_p*, który zmienia się z częstotliwością 1152 Hz (3,6864 MHz/3200). Oznacza to, że sygnał *clk_p* ma częstotliwość 576 Hz. Wszystkie instrukcje PicoBlaze zajmują dokładnie dwa takty sygnału zegarowego, czyli podłączając sygnał *clk_p* jako sygnał zegarowy mikrokontrolera, będzie on wykonywał 288 instrukcji w ciągu jednej sekundy. Aby zapewnić miganie diody LED z dokładnością 1 Hz należy więc wykonać 144 instrukcje i zapalić diodę LED. Następnie wykonać kolejne 144 instrukcje i zgasić diodę LED. Aby to osiągnąć potrzebujemy albo napisać program, który będzie mial 143 instrukcje puste i jedną ustawiającą wyjście diody LED, albo pętlę, która wykona odpowiednio 143 instrukcje puste. Cały program realizujący to zadanie przedstawiono na **list. 2**.

Zwróćmy uwagę na główną pętlę programu: od nazwy *start* do miejsca powrotu *JUMP start*. Wewnątrz tej pętli znajduje się:

– 6 zwykłych instrukcji;
– 2 pętle odliczające do wartości 47, które zawierają 3 zwykłe instrukcje.

Wynika z tego, że główna pętla programu PicoBlaze wykona dokładnie 288 instrukcji w jednym przebiegu:

6 + 2x(47x3) = 288

Instrukcje ustawiające wyjściowy sygnał *LED_cntrl* są wykonywane dokładnie co 144 instrukcje. Dzięki

List. 4. Opis VHDL projektu na PicoBlaze, migającego diodą LED

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pr_1 is Port
( clk : in  STD_LOGIC;
  rst : in  STD_LOGIC;
  LED_cntrl : out  STD_LOGIC);
end pr_1;

architecture Behavioral of pr_1 is
-- deklaracja KCPSM3
component kcpsm3 Port
( address : out std_logic_vector(9 downto 0);
  instruction : in std_logic_vector(17 downto
                                          0);
  port_id : out std_logic_vector(7 downto 0);
  write_strobe : out std_logic;
  out_port : out std_logic_vector(7 downto 0);
  read_strobe : out std_logic;
  in_port : in std_logic_vector(7 downto 0);
  interrupt : in std_logic;
  interrupt_ack : out std_logic;
  reset : in std_logic;
  clk : in std_logic);
end component;
-- deklaracja pamieci programu ROM
component prog_1 Port
( address : in std_logic_vector(9 downto 0);
  instruction : out std_logic_vector(17 downto
                                          0);
  proc_reset : out std_logic;
  clk : in std_logic);
end component;
-- Sygnaly uzyte do polaczenia KCPSM3 z pamiecia progra-
mu ROM
signal address : std_logic_vector(9 downto 0);
signal instruction : std_logic_vector(17 downto
                                          0);
signal port_id : std_logic_vector(7 downto 0);
signal out_port : std_logic_vector(7 downto 0);
signal in_port : std_logic_vector(7 downto 0);
signal write_strobe    : std_logic;
signal read_strobe     : std_logic;
signal interrupt       : std_logic;
signal interrupt_ack   : std_logic;
signal proc_reset      : std_logic;
signal main_reset      : std_logic;

-- Sygnaly dodatkowe
signal clk_p       : std_logic;
signal CLK_licznik : integer range 0 to 3199;

begin
-- KCPSM3 and the program memory
  processor: kcpsm3
    port map(      address => address,
               instruction => instruction,
                   port_id => port_id,
              write_strobe => write_strobe,
                  out_port => out_port,
               read_strobe => read_strobe,
                   in_port => in_port,
                 interrupt => interrupt,
             interrupt_ack => interrupt_ack,
                     reset => main_reset,
                       clk => clk_p);

  program_rom: prog_1
    port map(      address => address,
               instruction => instruction,
                proc_reset => proc_reset,
                       clk => clk_p);

  interrupt <= interrupt_ack;
  main_reset <= rst;

  LED_ustaw: process (clk_p, rst)
  begin
    if (rst = '1') then
      LED_cntrl <= '0';
    elsif clk_p'event and clk_p='1' then
      if ((port_id="00000000") and
                    (write_strobe='1')) then
        LED_cntrl <= out_port(0);
      end if;
    end if;
  end process;

  CLK_ustaw: process (clk, rst)
  begin
    if (rst = '1') then
      CLK_licznik <= 0;
      clk_p <= '0';
    elsif clk'event and clk='1' then
      if (CLK_licznik >= 3199) then
        CLK_licznik <= 0;
        clk_p <= not clk_p;
      else
        CLK_licznik <= CLK_licznik + 1;
      end if;
    end if;
  end process;
end Behavioral;
```

temu dioda LED będzie migać z częstotliwością 1 Hz. Do głównej części opisu mikrokontrolera należy jeszcze dodać element ustawiający sygnał wyjściowy *LED_cntrl*. Zadanie to opisano za pomocą procesu pokazanego na **list. 3**.

W przypadku wystąpienia sygnału zerującego, *LED_cntrl* przyjmuje wartość '0', która zostanie ustawiona, na podstawie wartości portu wyjściowego (dokładnie jego pierwszego sygnału) w momencie gdy, numer portu jest równy „00000000" i wystąpiła instrukcja zapisu do portu (sygnał *write_strobe* został uaktywniony wartością '1'). Odpowiada to instrukcji asemblerowej PicoBlaze *OUTPUT sX, 00*, gdzie sX ma wartość na pierwszym bicie '0', żeby zgasić diodę lub '1' – żeby ją zapalić.

Na **list. 4** pokazano pełny opis głównej części projektu mikrokontrolera, realizującego miganie diodą LED.

Symulacja

Rozwiązania oparte o PicoBlaze możemy weryfikować na kilka sposobów. Jeden z nich opiera się o symulację kodu asemblerowego. Drugim rodzajem symulacji jest wykorzystanie symulatorów języków HDL. Do tego celu są tworzone specjalne opisy, zwane testbenchami. Są to fragmenty opisujące zachowanie się układów zdefiniowanych językami opisu sprzętu. Aby przesymulować nasz układ mrugający diodą wystarczy, że testbench wyzeruje na początku mikrokontroler i dostarczy taktujący sygnał zegarowy. Następnie powinniśmy na wyjściu *LED_cntrl* obserwować jego zmiany co 0,5 sekundy.

List. 5.

```
CLK_gen: process
begin
  if END_SIM = FALSE then
    clk <= '0';
    wait for 136 ns;
  else
    wait;
  end if;
  if END_SIM = FALSE then
    clk <= '1';
    wait for 136 ns;
  else
    wait;
  end if;
end process;
```

List. 6.

```
STIMULUS: process
begin
  rst <= '1';
    wait for 50 ns;
  rst <= '0';
    wait for 2000 ms;
  END_SIM <= TRUE;
  wait;
end process;

library ieee;
use ieee.std_logic_1164.all;

entity pr_1_tb is
end pr_1_tb;
```

Opis działania testbencha

W części testbencha opisującej zachowanie testowanego układu należy utworzyć proces, który dostarczy sygnału zegarowego o oczekiwanej częstotliwości. Następnie wykorzystując instrukcje czasowe steruje się zmianą pozostałych wejść testowanego układu, natomiast wyjścia można sprawdzać wizualnie na przebiegach lub porównywać je z wartościami oczekiwanymi. W naszym przypadku powinniśmy dostarczyć sygnał zegarowy o częstotliwości 3,6864 MHz, za co odpowiada proces pokazany na **list. 5**.

Niestety, w językach HDL nie możemy podać częstotliwości zmiany sygnału, a jedynie czas kolejnych jego zmian. Ponieważ 1/3,6864 MHz nie dzieli się całkowicie, więc trzeba użyć najbliższej wartości w danych jednostkach. Większość symulatorów języków VHDL poprawnie obsługuje jednostki *ns*. Mniejsze jednostki *fs* nie są zawsze obsługiwane. Podając wartość 136 ns na każdą zmianę sygnału *clk* uzyskujemy częstotliwość bliską 3,6764 MHz. Do zastosowań symulacji ta wartość jest jak najbardziej wystarczająca. Następnie opisujemy zachowanie się symulacji dla sygnału *rst*. Dodatkowy sygnał *END_SIM* (**list. 6**) pozwala nam kontrolować czas trwania symulacji. Kompletny testbench pokazano na **list. 7**.

Marcin Nowakowski

List. 7.

```
architecture TB_ARCHITECTURE of pr_1_tb is
  component pr_1
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           LED_cntrl : out  STD_LOGIC);
  end component;

  -- Sygnaly potrzebne do symulacji
  signal clk : std_logic;
  signal rst : std_logic;
  signal LED_cntrl : std_logic;

  -- Sygnal do kontroli czasu trwania symulacji
  signal END_SIM: BOOLEAN:=FALSE;

begin

  UUT : pr_1
  port map (
      clk => clk,
      rst => rst,
      LED_cntrl => LED_cntrl
    );

  STIMULUS: process
  begin
    rst <= '1';
      wait for 50 ns;
    rst <= '0';
      wait for 2000 ms;
    END_SIM <= TRUE;
    wait;
  end process;

  CLOCK_Ref_clk : process
  begin
    if END_SIM = FALSE then
      clk <= '0';
      wait for 136 ns;
    else
      wait;
    end if;
    if END_SIM = FALSE then
      clk <= '1';
      wait for 136 ns;
    else
      wait;
    end if;
  end process;

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_pr_1 of pr_1_tb is
 for TB_ARCHITECTURE
    for UUT : pr_1
      use entity work.pr_1(Behavioral);
    end for;
 end for;
end TESTBENCH_FOR_pr_1;
```