NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (15)

Parę drobiazgów, które nigdzie nie pasują, czyli… uzupełnienie

To już ostatnia część naszej wspólnej wędrówki przez świat systemu Nios II. W czasie pracy z układami FPGA i systemami mikroprocesorowymi przyda Wam się z pewnością parę dodatkowych narzędzi i umiejętności – a więc zaczynamy! I na zakończenie kolejny raz nowa wersja Quartusa – tym razem 18.1!

Czasem może okazać się konieczne wykonanie symulacji całego systemu mikroprocesorowego NIOS II wraz z oprogramowaniem w nim działającym. Na chwilę obecną jednak (Quartus 18.1) występują pewne problemy z symulacją pętli PLL umieszczonej w ekosystemie Nios II i dlatego będziemy musieli posłużyć się drobną sztuczką. Zakładam, że startujemy tym razem od systemu, w którym umieszczone są już wszystkie komponenty, a pamięć RAM jest zainicjalizowana plikiem binarnym z oprogramowaniem.

Symulacja mikroprocesora NIOS II

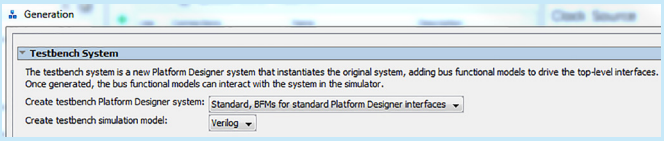
Otwieramy nasz system i dodajemy do niego komponent *Clock Bridge*. W polu *Explicit clock rate* podajemy 50000000 (50 MHz, zakładając, że takiej częstotliwości sygnał pochodził z pętli PLL). Nazwę komponentu możemy zmienić na *PLL_SIM* i umieścić go obok pętli PLL. Następnie odznaczamy obok pętli PLL zaznaczanie w kolumnie *Use*. Teraz pozostało nam połączyć sygnały tak, aby naśladować połączenia, jakie były wykonane wcześniej przez pętlę PLL (tj. na wejście *Clock Bridge* podajemy sygnał wcześniej podpięty do wejścia pętli PLL, a wyjście łączymy tam, gdzie wcześniej było dołączone wyjście z pętli PLL).

Następnie przechodzimy do *Generate → Generate Testbench System…*, a tam pozostawiamy opcje bez zmian i klikamy *Generate*. Zdziwić Was może fakt, że zostawiamy jako język syntezy wybrany język *Verilog* – to nie pomyłka ani przypadek. Otóż nasz procesor NIOS II został stworzony właśnie w języku *Verilog* i wybranie tegoż języka jako języka tej symulacji przyspieszy ją znacznie. Możecie sami potem porównać, jak szybko wykona się ta sama symulacja przy wyborze *VHDL*. I myślę, że w tym miejscu powinienem także zachęcić do nauki *Veriloga* na równi z *VHDL* – znajomość obu tych języków na pewno się przyda, a każdy z nich może służyć do osiągnięcia identycznych efektów (**rysunek 1**).

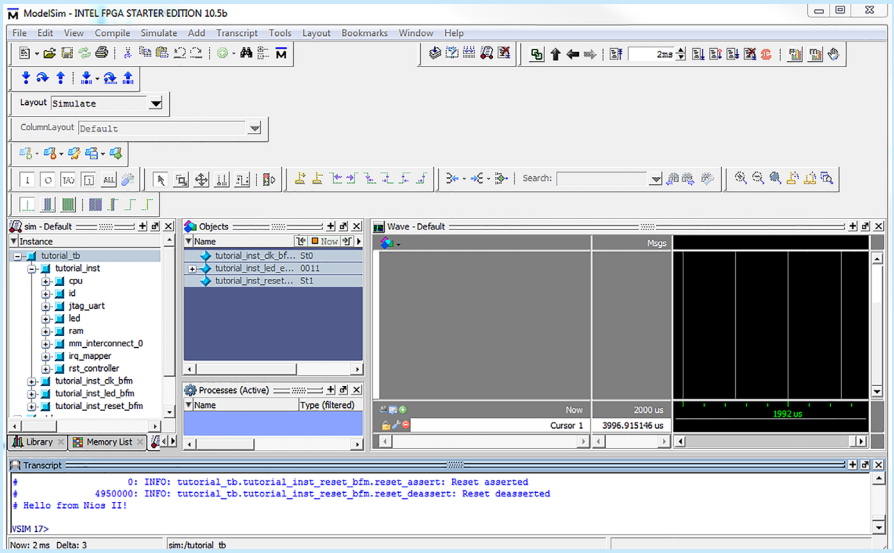
Po tym klikamy rzecz jasna *Generate*. Teraz przechodzimy do *Eclipse*, tam generujemy oczywiście BSP, kompilujemy oprogramowanie (które wysyła jedynie komunikat za pomocą JTAG UART, a potem podaje na 4 wyjścia naprzemiennie dwie kombinacje binarne). Po tym klikamy na naszym projekcie oprogramowania (bez sufiksu _bsp) i wybieramy *Run As → Nios II ModelSim*. Po tym otworzy się nowe okno programu do symulacji (w przypadku błędu, mówiącego, że nie odnaleziono pliku wykonywalnego, wskazujemy ścieżkę <katalog instalacji>\<wersja>\modelsim_ase\win32aloem), pokazane na **rysunku 2**.

W oknie programu są już ustawione najważniejsze widoki – od lewej:

- ***sim – Default*** – zawiera listę instancji wszelkich komponentów użytych w systemie – rozwijając kolejne poziomy listy, możemy podejrzeć elementy składowe danego komponentu.
- ***Objects*** – zawiera listę sygnałów obecnych w danym (podświetlonym we wcześniejszym panelu) komponencie.



Rysunek 1. Ustawienia generowania projektu symulacji NIOS II



Rysunek 2. Okno programu ModelSim zaraz po uruchomieniu symulacji z poziomu Eclipse

- ***Wave – Deafult*** – okno widoku przebiegów. Aby w czasie symulacji zarejestrować przebieg danego sygnału, musimy go przeciągnąć z panelu *Objects* do tego panelu. Pamiętajmy jednak, aby zrobić to przed symulacją – inaczej przebieg nie zostanie zarejestrowany!
- ***Transcript*** – wiersz poleceń, w którym wyświetlają się także komunikaty w czasie symulacji.

Jak już wspomniałem, jeśli chcemy jakiś przebieg zarejestrować, musimy go przeciągnąć z panelu *Objects* do panelu *Wave*. Na początek wybierzmy w panelu *sim tutorial_tb*, a następnie przeciągnijmy wszystkie trzy dostępne sygnały (można zaznaczać wiele z nich za pomocą klawiszy *CTRL* lub *SHIFT*). Aby zobaczyć trochę więcej ciekawych rzeczy postąpmy podobnie dla modułu *led* (rozwijamy kolejno *tutorial_tb*, *tutorial_inst*) i przeciągnijmy wszystkie sygnały typu *In* oraz *Out*. Dla poprawy czytelności możemy zaznaczyć wszystkie sygnały związane z modułem *led*, kliknąć następnie *PPM* i wybrać opcję *Group* i nadać odpowiednią nazwę takiej grupie sygnałów.

Tak przygotowani możemy uruchomić symulację. Do tego celu służy nam panel w górnej części okna programu (**rysunek 3**). Ikony od lewej strony to:

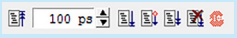
- Restart symulacji.
- Pole do wprowadzenia pożądanego czasu symulacji (my wpiszemy tam czas 2 ms).
- Uruchomienie symulacji na podany wcześniej czas.
- Wznowienie symulacji po zatrzymaniu.
- Uruchomienie symulacji, dopóki są dostępne pobudzenia symulacyjne (w naszym przypadku działałoby to w nieskończoność).
- Przerwanie symulacji.

Tak więc nie pozostało nam nic innego jak wpisać w odpowiednie pole pożądany czas 2 ms i uruchomić symulację. A po chwili zobaczymy w konsoli na dole ekranu tekst, który miał zostać wysłany przez *JATG UART*, zaś w oknie *Wave* zobaczymy przebiegi na wybranych wcześniej sygnałach. Dzięki temu możemy sprawdzić zachowanie naszego systemu, a także zobaczyć, np. jak radzą sobie nasze moduły w czasie interakcji z procesorem, obejrzeć działanie linii *byteenable* oraz zajrzeć do różnych zakamarków układu.

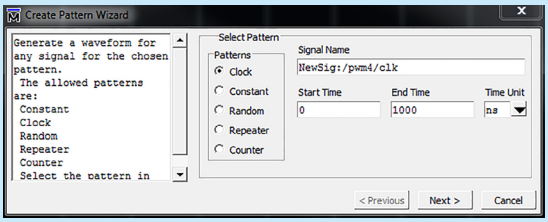
Symulacja trochę prościej

Wiemy już, jak dokonać symulacji całego systemu z procesorem NIOS II, jednak nie zawsze musimy wytaczać aż taką armatę – do symulacji pojedynczego modułu wystarczy, a nawet lepiej będzie dokonać mniejszej symulacji, tylko interesującej nas części. Na warsztat weźmy przygotowany kiedyś moduł do generowania 4 sygnałów PWM. Dodajemy jego plik źródłowy do naszego projektu.

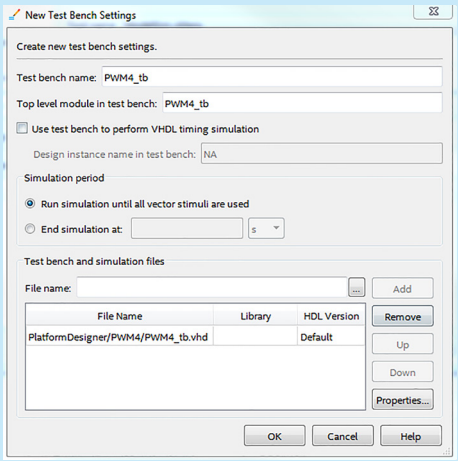
1. Jeśli dany moduł obecnie nie jest wykorzystywany w systemie, to na chwilę musimy ustawić go jako *Top Level* – po wygenerowaniu testbenchu należy wtedy wrócić do ustawienia właściwego modułu jako głównego.
2. Uruchamiamy *Analysis & Synthesis*.
3. Następnie uruchamiamy *Tools* → *Run Simulation Tool* → *RTL Simulation*.
4. Wybieramy *Simulation Language* na *VHDL*.
5. Jeśli symulator nie chce się uruchomić, wyświetlając błąd informujący o nieodnalezieniu pliku wykonywalnego, wchodzimy w *Tools* → *Options…* i tam w zakładce *General* → *EDA Tool options* w polu *ModelSim-Altera* wskazujemy ścieżkę dostępu do symulatora: <katalog instalacji>\<wersja>\modelsim_ase\win32aloem.
6. Następnie, w zależności od punktu pierwszego, nasz moduł znajdziemy w panelu *Library* albo w grupie *work*, albo w grupie o nazwie takiej, jak nazwa systemu nadana w *Platform Designer* i w takim miejscu, w jakim moduł ten jest w rzeczywistości podłączony.
7. Gdy go odnajdziemy, klikamy na jego nazwie PPM, a następnie wybieramy *Create Wave*. Teraz po prawej stronie pojawią się sygnały, będące wejściami i wyjściami naszego modułu. Kliknijmy PPM na */pwm4/clk* i wybierzmy *Edit* → *Wave Editor* → *Create/Modify Waveform*.
8. Zobaczymy nowe okno (**rysunek 4**), w którym możemy wybrać różne typy pobudzeń. Zostawiamy zaznaczoną opcję *Clock* i pozostałe parametry. Klikamy *Next*.
9. W kolejnym okienku podajemy wartość początkową na 0, okres na 20 ns (50 MHz) i wypełnienie na 50%.
10. Następnie klikamy na *File* → *Export* → *Waveform*. Wybieramy *VHDL testbench* oraz wskazujemy nazwę pliku i zapisujemy nasz wygenerowany testbench.
11. Otwieramy wygenerowany plik w *Quartus* i dopisujemy w nim pobudzanie pozostałych sygnałów modułu. Plik ten możemy dodać do projektu, jeśli chcemy mieć do niego łatwy dostęp. Warto powiedzieć o tym, w którym momencie symulacja się skończy – nastąpi to albo po zadanym czasie (pamiętacie wprowadzanie czasu przy symulacji całego procesora?), albo po „wyczerpaniu się" wektorów testowych – czyli w momencie kiedy wszystkie *process* pliku testowego dotrą do instrukcji *wait;*.
12. Zmodyfikujmy także proces generowania sygnału zegarowego, aby generował 1000 okresów sygnału – abyśmy zobaczyli kawałek działania naszego układu.
13. Na tym etapie możemy w projekcie przywrócić już właściwy moduł *Top Level*.
14. Dodatkowo, jeśli z jakichś powodów w czasie generowania nadane zostaną komponentom nazwy z ukośnikami oraz dopiskiem .vhd – usuńmy wszelkie ukośniki oraz wystąpienia .vhd – korekta kosmetyczna. Zapamiętajmy także, jak się nazywa główny moduł w naszym *testbench* – u mnie jest to *PWM4_tb*.
15. Następnie wchodzimy w *Assignments* → *Settings…* a tam w zakładce *EDA Tool Settings* → *Simulation* w panelu *NativeLink settings* wybieramy *Compile test bench*, a następnie klikamy na *Test Benches…* a potem na *New...*
16. W nowo otwartym oknie (**rysunek 5**) wpisujemy nazwę *testbench'a*, a poniżej nazwę głównego modułu w tymże pliku.



Rysunek 3. Ikony na panelu kontroli symulacji

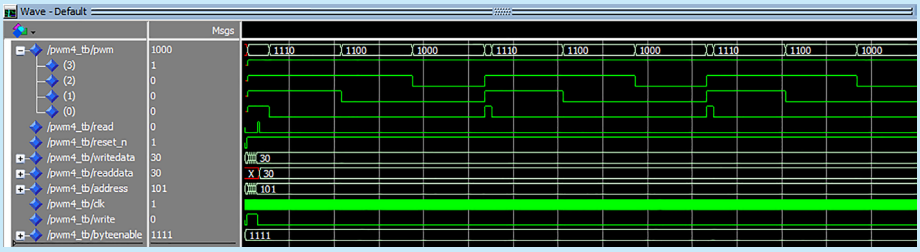


Rysunek 4. Okno definiowania pobudzenia w ModelSim



Rysunek 5. Prawidłowo skonfigurowane okno dodawania nowego pliku testowego

Na dole wskazujemy i dodajemy odpowiedni plik. Możemy tu także wybrać, kiedy dana symulacja ma się zakończyć – czy po wyczerpaniu wszystkich wektorów testowych (opcja domyślna, pozostawiamy ją zaznaczoną), czy też ma się zakończyć po określonym czasie. Klikamy *OK* w kolejnych oknach, aż do kliknięcia *OK* w oknie ustawień.



Rysunek 6. Wynik symulacji modułu PWM

17. Teraz już tylko uruchamiamy, jak na początku *Tools* → *Run Simulation Tool* → *RTL Simulation*, a po chwili powinniśmy zobaczyć okno *ModelSim* z już przeprowadzoną symulacją – możemy obserwować, jak wygląda zapis do naszego modułu, a następnie jak wyglądają generowane sygnały PWM (aby je lepiej zobaczyć, trzeba rozwinąć 4-bitowy wektor *pwm*).

Dodatkowo, jeśli chcemy podejrzeć sygnały wewnętrzne naszego modułu – możemy zresetować symulację, przeciągnąć do odpowiedniego widoku sygnały wyłuskane ze środka naszego *DUT* (*Device Under Test*) i ponownie wykonać symulację. Można także zmienić format wyświetlania liczb, z binarnego na liczbowy (u nas *Unsigned*), poprzez kliknięcie na dany sygnał i wybranie odpowiedniej opcji *Radix* (**rysunek 6**). Dzięki temu możemy przetestować zachowanie naszego modułu lub dowolnej innej logiki bez wykonywania czasochłonnej syntezy, implementacji. Właśnie tak w rzeczywistości projektuje się układy opisywane w językach opisu sprzętu – oczywiście potem wykorzystuje się dodatkowe narzędzia i procedury, m.in. inny typ symulacji, w której symulujemy układ z uwzględnieniem tego, jak zostanie on zsyntezowany i jakie wygeneruje to opóźnienia (tzw. *Gate Level Simulation*).

A może NIOS będzie tylko komponentem?

Moim zdaniem, najłatwiej (a potem tworzy to najbardziej czytelny obraz) jest połączyć dodatkowe komponenty do systemu *NIOS II* z wykorzystaniem narzędzi *Platform Designera*, tak jak to robiliśmy z modułem buforów dla magistrali I²C, jednak nie jest to jedyna możliwość – możemy sprawić, aby cały system mikroprocesorowy był komponentem w większym projekcie.

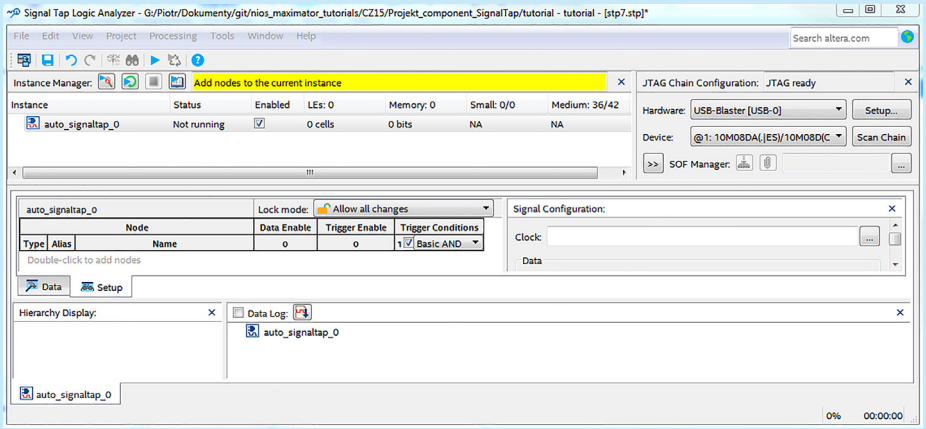
1. Tworzymy nowy plik w naszym projekcie i nadajemy mu praktycznie dowolną nazwę – ja wybrałem, nieprzypadkowo, *top.vhd*. W nim tworzymy *entity* z portami, jakie chcemy, aby miał nasz układ – ja wybrałem dodatkowo nazwy takie same jak obecne w systemie wygenerowanym z *Platform Designer*, dzięki temu nie musimy modyfikować przypisania pinów układu. Dodatkowo utworzyłem dodatkowy sygnał, aby zademonstrować, że nie robimy całej tej operacji sztuka dla sztuki.
2. Uruchamiamy *Platform Designer* i w nim wybieramy *Generate* → *Show Instantiation Template…*, a naszym oczom okaże się okienko pokazujące szablony. W odpowiednim miejscu wybieramy nasz ulubiony język (*VHDL*) i za pomocą przycisku kopiujemy zawartość okna do schowka.
3. Wklejamy odpowiednie fragmenty tych szablonów do naszego pliku *top.vhd* a następnie dokonujemy koniecznych połączeń, przy okazji negując stan linii sterujących diodami LED.
4. Na koniec ustawiamy właśnie przygotowany plik jako *Top Level* i możemy wykonać syntezę.

I tym sposobem możemy uczynić procesor NIOS elementem większego systemu, gdzie mikrokontroler będzie jedynie wspomagał pewne funkcje.

A jeśli jednak trzeba sprawdzić coś na żywo?

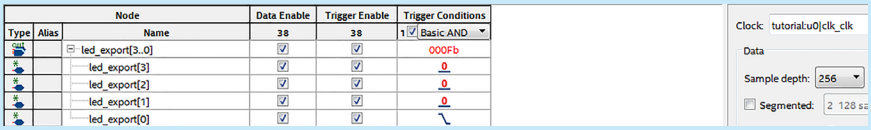
Oczywiście symulacje, nawet te prowadzone po syntezie (*Gate Level*), czasem nie dają możliwości sprawdzenia w 100% zachowania układu i część rzeczy jesteśmy zmuszenia sprawdzić w realnym świecie – np. podłączenie jakiejś pamięci czy układu peryferyjnego lub urządzeń wejściowych. Oczywiście większość takich elementów możemy z mniejszą lub większą precyzją zasymulować, jednak czasem problem może stwarzać choćby opóźnienie jakiegoś sygnału wynikające z błędu na PCB. Wtedy nieodzowne staje się użycie analizatora stanów logicznych. Rzecz jasna czasem będziemy musieli wyposażyć się w sprzętowy analizator stanów logicznych, jednakże mamy dostęp także do programowego analizatora stanów logicznych w naszym układzie FPGA – wystarczy, korzystając z odpowiedniej funkcji *Quartusa*, dodać do naszego projektu logikę analizatora stanów logicznych, a następnie połączyć się z nim za pomocą interfejsu JTAG. Co lepsze – będziemy mieć możliwość zobaczenia nie tylko tego, co dzieje się na wyprowadzeniach układu, ale także tego, co dzieje się w środku!

1. Dodajemy do projektu nowy plik, a w oknie wyboru jego typu wskazujemy *Verification/Debugging Files* → *Signal Tap Logic Analyzer File*. Wyświetli się nam okno konfiguracji naszego analizatora.
2. Następnie musimy wybrać sygnał zegarowy. W tym celu klikamy w panelu *Signal Configuration* na ikonę… obok pola *Clock*. Rozciągamy okno, które się pojawi (**rysunek 7**).
3. Klikamy w nim na ikonę podwójnej strzałki w dół, a następnie wybieramy z listy *Filter* opcję *Signal Tap: pre-synthesis*. Spowoduje że wyświetlą nam się nazwy sygnałów takie, jak te obecne w plikach źródłowych, a nie zmienione później przez oprogramowanie.
4. Klikamy na *List* i w oknie *Matching Nodes* wybieramy *tutorial:u0* → *clk_clk*. Klikamy go dwukrotnie, po czym klikamy na *OK*. Ustawiliśmy tym samym sygnał zegarowy, przy którego narastających zboczach pobierane będą kolejne próbki danych cyfrowych.
5. Poniżej mamy możliwość ustalenia, jaka liczba próbek będzie pobrana – ustawmy ją na 256.
6. W panelu po lewej stronie klikamy na białym polu (na którym jest napis *Double-click…*). Ponownie pojawia się znane



Rysunek 7. Początek konfiguracji Signal Tap

nam okno. Tym razem wybierzemy *led_export*. Możemy także wybrać inne interesujące sygnały (np. *write_n* oraz *writedata* z modułu *tutorial_LED:led*).



Rysunek 8. Prawidłowe ustawienia analizatora dla naszego projektu

Rysunek 9. Przebiegi obserwowane na wyjściach sterujących diodami LED oraz na wybranych sygnałach magistrali Avalon-MM

7. Teraz ustawiamy wyzwalanie – w kolumnie *Trigger Conditions*, po rozwinięciu sygnałów *led_export* (klikając PPM) wybieramy dla bitów 3, 2 oraz 1 wartość *Low*, natomiast dla bitu zerowego wybieramy *Falling*. Ustawienia powinny teraz wyglądać w sposób pokazany na **rysunku 8**.
8. Zapisujemy plik pod wybraną nazwą i odpowiadamy twierdząco na pytanie o to, czy włączyć ten analizator do projektu. Potem klikamy na niebieską strzałkę w górnym pasku okna analizatora – uruchomi to kompilację naszego projektu wraz z dodanym analizatorem Możemy także kliknąć na ikonę obok ikony zapisywania pliku – *Attach Window* – spowoduje to, że okienko analizatora stanie się zakładką w oknie głównym Quartusa – podobnie możemy zrobić także z niektórymi innymi oknami, jednak czasem opcja ta jest dostępna w różnych miejscach i nieco ukryta.
9. Po zakończeniu kompilacji programujemy nasz układ – możemy to zrobić, korzystając z panelu w prawym górnym rogu okna analizatora – wskazujemy tam plik *.sof*, a potem za pomocą ikony ze strzałką i układem dokonujemy programowania.
10. Następnie klikamy na linijkę z wpisem *auto_signaltap_0* – to nasz analizator, którego nazwę możemy zmienić, włączyć lub wyłączyć. Ponadto możemy dodać tutaj wiele różnych analizatorów. Obok widzimy także, ile zasobów układu FPGA zajmuje nasz analizator – nic nie mamy za darmo…
11. Powyżej klikamy na ikonę *Run Analysis* (obok napisu *Instance Manager*) i gotowe – nasz analizator właśnie zebrał dane z wybranych przez nas linii. Musimy jednak zwrócić uwagę, że niektóre sygnały wewnątrz procesora mogą być próbkowane zbyt rzadko (np. *write_n, writedata*) – próbkujemy z częstotliwością sygnału zegarowego sygnały, które z taką samą częstotliwością się zmieniają.
12. Aby temu zaradzić, musimy zwiększyć częstotliwość próbkowania przynajmniej 2-krotnie. W tym celu w *Platform Designer* musimy dodać do naszej pętli PLL kolejne wyjście, podające sygnał zegarowy 100 MHz, którego nigdzie nie podepniemy. Następnie przeprowadzamy *Analysis & Synthesis*.
13. W zakładce *Setup* analizatora zmieniamy źródło sygnału zegarowego na *tutorial:u0* → *tutorial_PLL:pll* → *c1* (o ile na wyjściu c1 generujemy sygnał zegarowy 100 MHz). Po tym kompilujemy projekt.
14. Po zaprogramowaniu układu i uruchomieniu analizatora powinniśmy zobaczyć przebieg na wyjściu naszego układu (**rysunek 9**), a także sekwencje zapisu na magistrali *Avalon*. Warto zwrócić uwagę, że znaczące są tu tylko 4 najmniej znaczące bity magistrali *writedata* oraz że same na wyjściu są finalnie zanegowane w tym projekcie (chyba że negację tę usuniemy).
15. Aby wyłączyć dodawanie do projektu analizatora stanów, albo możemy odznaczyć zaznaczenie w kolumnie *Enabled* w *Instance Manager* (za pomocą takiego rozwiązania możemy mieć także kilka różnych analizatorów, które szybko możemy przełączać), albo w menu *Assignments* → *Settings* → *Signal tap Logic Analyzer* odznaczyć opcję *Enable Signal Tap…*, co wyłączy wszystkie analizatory.

Podsumowanie… całego kursu

To już zakończenie tego cyklu artykułów, czas więc na drobne podsumowanie cyklu 15 spotkań z systemem NIOS II. Rozpoczęliśmy od budowy podstawowego systemu mikroprocesorowego, który nawet nie miał wyprowadzeń do zamigania diodami. Później sukcesywnie uczyliśmy się wzbogacać jego możliwości o porty wejścia/wyjścia, timery (wraz z obsługą przerwań), porty komunikacyjne (UART, SPI, I_2C) po przetwornik ADC. Potem poświęciliśmy trochę czasu na budowę prostych, dedykowanych modułów, które realizowały czysto sprzętowo różne zadania – od generowania sygnałów PWM po obsługę diod WS2812. Na koniec udało nam się opanować generowanie sygnału wideo i przesyłanie go za pomocą interfejsów VGA oraz HDMI. Wreszcie teraz uzupełniliśmy wiedzę o pewne techniki i narzędzia ułatwiające pracę z systemami i układami programowalnymi.

Zawsze w takich momentach przychodzi czas na pytanie: co dalej? Właściwie kurs ten można potraktować jak naukę alfabetu, a potem pisania słów – teraz Wasza kolej na napisanie korzystając z tej wiedzy całych opowiadań, a także poszerzenie posiadanych informacji. Polecam szczegółową naukę języków opisu sprzętu: VHDL i Verilog, a następnie realizację wszelkich projektów, jakie przyjdą Wam do głowy – od budowy kostki LED po zaprojektowanie prostego generatora grafiki 3D z interfejsem HDMI – słowem dobrze wykorzystajcie zasoby tego lub każdego innego zestawu wyposażonego w układ FPGA. Powodzenia i czekam, aż kiedyś opublikujecie własne projekty, wykorzystujące ekosystem NIOS II!

Podziękowania

Na zakończenie warto wspomnieć, że cała ta seria artykułów nie powstałaby, gdyby nie niektóre osoby, której pojawiły się na mojej trasie podróży przez świat elektroniki (którą rozpocząłem dzięki moim Rodzicom).

I tak chciałbym podziękować:

- Dominikowi Bieczyńskiemu, po rozmowach z którym w 2013 roku kupiłem swój pierwszy zestaw z układem FPGA.
- Doktorom: Ernestowi Jamro, Pawłowi Rajdzie i Jerzemu Kasperkowi, którzy w czasie studiów pomagali mi rozwijać moje elektroniczne zainteresowania związane z układami FPGA i dzięki wsparciu których powstał mój pierwszy projekt na zestawie maXimator.
- Panu Piotrowi Zbysińskiemu, bez którego pomocy ta seria by nie powstała.
- Jakubowi Tyburskiemu, który sprawdzał, czy kody VHDL, które piszę, są merytorycznie prawidłowe.
- Nie mogę zapomnieć o doktorze Łukaszu Krzaku oraz Piotrze Chodorowskim, którzy zgodzili się na wykorzystanie fragmentów ich kodów w tych publikacjach.

Dziękuję serdecznie wszystkim tu wymienionym, a także wielu innym osobom, które doprowadziły mnie do tego miejsca.

Piotr Rzeszut, AGH

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl