NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (1)

Wstęp i pierwszy projekt

W jednym z poprzednich artykułów na łamach Elektroniki Praktycznej pan Mariusz Księżak wprowadził nas w tematykę implementacji procesora NIOS II na płytce maXimator oraz zaprezentował przykładowy projekt z jego użyciem. Teraz czas, abyśmy wspólnie uporządkowali i pogłębili wiedzę na ten temat.

Po co nam procesor w układzie FPGA? Od odpowiedzi na to pytanie warto zacząć nasze rozważania, gdyż zanim zaczniemy stosować rzadziej lub częściej mikroprocesor implementowany w układzie FPGA, musimy dowiedzieć się kiedy i dlaczego to robić.

Po pierwsze warto porównać układ FPGA do zwykłego procesora (mikrokontrolera) – takie porównanie umieszczono w **tabeli 1**. Oczywiście, można by tworzyć bardzo rozbudowaną tabelę, ale myślę, że pokazana zwraca uwagę na najważniejsze kwestie i pozwala na wyciągnięcie prostego wniosku – niektóre zadania będą łatwiejsze do wykonania w układzie FPGA lub nawet będą wymagały takiego podejścia, zaś inne będą znacznie łatwiejsze do implementacji w oprogramowaniu uruchamianym na mikroprocesorze.

Posłużmy się teraz dwoma prostymi przykładami. W pierwszym z nich zadaniem niech będzie wyświetlenie na monitorze VGA jednobarwnego obrazka z rozdzielczości 1024×768 pikseli. Stosując do tego celu układ FPGA zaprojektujemy układ logiczny, taktowany z częstotliwością równą częstotliwości wyświetlania kolejnych pikseli (65 MHz), który dane z pamięci będzie wystawiał na linie sterujące wyświetlaczem oraz będzie zliczał piksele

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **95777**, pass: **53wtjyf6**

Tabela 1. Porównanie typowego procesora z zaimplementowanym w FPGA

Kategoria	Mikrokontroler/ procesor	Układ FPGA
Czym jest?	Konkretny układ logiczny przystosowany do wykonywania sekwencyjnego zadanych instrukcji (programu)	Macierz prostych komponentów cyfrowych, które można łączyć w praktycznie dowolne układy cyfrowe
Prędkość działania	Zależna od częstotliwości taktowania	Zależna od czasu propagacji w układzie
Wykonywanie zadań równocześnie	Brak, czasem pewne zadania może pełnić DMA[1]	Możliwa implementacja w pełni równoległych procesów
Łatwość implementacji złożonych algorytmów	Stosunkowo duża	Może stanowić wyzwanie, gdyż wymaga podejścia sprzętowego, a nie proceduralnego

Artykuł pisano w oparciu o najnowszą dostępną w momencie jego tworzenia wersję: Quartus 17.0.2

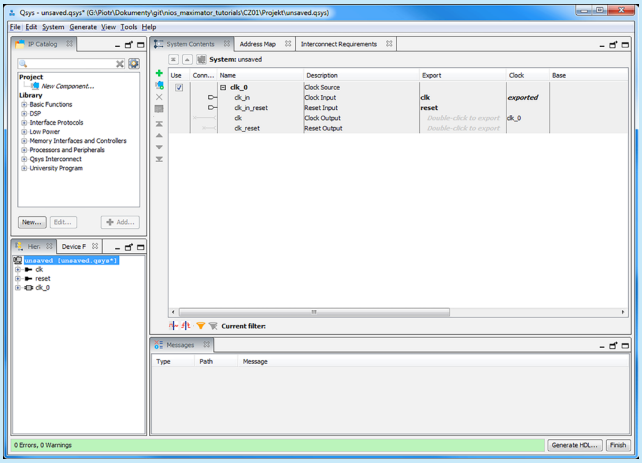
DMA – Direct Memory Access – system w niektórych mikrokontrolerach (np. ARM) i procesorach, pozwalający na kopiowanie danych (bez ich jakiejkolwiek modyfikacji) pomiędzy różnymi fragmentami przestrzeni adresowej procesora, bez użycia CPU, który może wtedy wykonywać inne zadania (np. wysyłanie danych z jakiegoś zakresu pamięci RAM na przetwornik cyfrowo-analogowy). Zakładamy także, że mówimy o procesorze jednordzeniowym.

oraz linie i generował odpowiednie sygnały synchronizacji. Może to wyglądać skomplikowanie, ale w rzeczywistości to przysłowiowa „bułka z masłem". Pamiętajmy, że układ FPGA może wykonywać przy dobrym projekcie wiele czynności na raz i w jednym takcie zegara odczytać dane pamięci, zwiększyć licznik pikseli, porównać go w celu ustalenia momentów synchronizacji… Teraz kolej, aby do tego zadania zaprzęgnąć procesor. Z grubsza policzmy, że na każdy piksel musimy: odczytać dane o pikselu z pamięci RAM, wysterować port wyjścia, zwiększyć licznik pikseli, porównać go i tak dalej. Już chyba dostrzegacie, w czym problem? Na jeden piksel procesor musi wykonać na pewno więcej niż 4 takty. Zatem musimy go taktować minimum zegarem 4×65 MHz=260 MHz. No i nie zapominajmy – procesor wtedy nie może robić nic poza wyświetlaniem tego obrazu. Wybór zatem jest tu jasny – układ FPGA zwycięża!

Zadanie drugie – obsługa skomplikowanego, wielopoziomowego menu użytkownika. FPGA? Jest mi trudno wyobrazić sobie nakład pracy na wykonanie projektu odpowiedniej struktury realizującej taką funkcję. Oczywiście, jest to możliwe, ale w procesorze można takie menu zrealizować w mig. Potrzeba do tego kilku struktur opisujących dane, parę instrukcji warunkowych, tablic i tyle. Viola! Procesor w tym zadaniu zwycięża!

Mam nadzieję, że te przykłady pozwalają na dojście do wniosku, że do niektórych zadań lepszy będzie procesor, a do innych układ FPGA. Jeszcze inne zadania mogą być realizowane równie dobrze w procesorze jak i w układzie FPGA, jak na przykład eliminacja drgań styków.

W praktyce koncepcja łączenia procesora i logiki w układzie FPGA jest często wykorzystywana. Czasami nawet procesor nie



Rysunek 1. Okno programu Qsys zaraz po otwarciu

jest syntezowany w układzie FPGA (jak my będziemy to robili za moment), ale jest fizycznie umieszczany w strukturze układu scalonego (układy SoC, np. Cyclone V).

Poznajemy NIOS II

NIOS II to tak naprawdę przede wszystkim rdzeń procesora, który występuje w 2 podstawowych wariantach:

1. NIOS II /e – Economy – podstawowy rdzeń z rodziny NIOS, zajmuje mało zasobów sprzętowych układu FPGA, jest dostępny za darmo bez ograniczeń.
2. NIOS II /f – Fast – bardziej rozbudowany i szybszy rdzeń, zajmuje więcej zasobów sprzętowych, w wersji darmowej działa tylko przy podłączonym debuggerze lub przez godzinę po jego odłączeniu.

W wypadku naszego zestawu maXimator idealnym rozwiązaniem jest rdzeń Economy, który zajmie niewiele zasobów naszego układu, pozostawiając jeszcze sporo miejsca na implementowane przez nas funkcje.

Czy jednak sam rdzeń NIOS II zadziała? Nie! Musimy jeszcze (jak inżynierowie projektujący „normalny" procesor) dołączyć do niego pamięci (operacyjną i instrukcji), porty wejścia/wyjścia i inne potrzebne elementy. Na szczęście, do tego celu przygotowano środowisko Qsys, którego obsługi wkrótce się nauczymy.

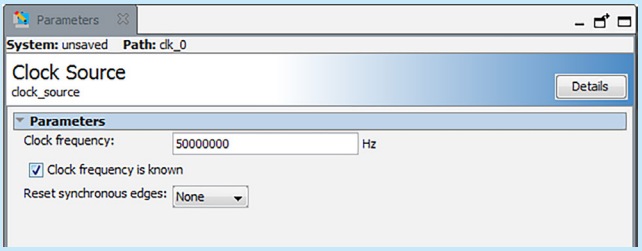
Nasz pierwszy projekt – minimalny, działający system

Zakładam, że każdy umie stworzyć pusty projekt w środowisku Quartus. Ja na potrzeby artykułu wykonałem projekt o nazwie Tutorial01.

PIERWSZE KROKI (KLIKNIĘCIA) W QSYS. Klikamy na *Tools* → *Qsys*. Zostanie wyświetlone (po chwili niezbędnej na załadowanie wszystkich komponentów) pokazane na **rysunku 1** okno naszego narzędzia. Po lewej stronie, u góry, znajduje się katalog gotowych komponentów *IP Catalog* (z którego zaraz skorzystamy). Poniżej mamy dostęp do widoku hierarchii systemy (*Hierarchy*) oraz w drugiej karcie okienko ewentualnej zmiany układu, z jakim pracujemy. W dolnej części okna, w zakładce *Messages* będą pojawiały się w czasie tworzenia naszego systemu mikroprocesorowego komunikaty o błędach i inne informacje – zawsze warto się z nimi zapoznać. W głównej części okna, w zakładce *System Contents* wyświetlany jest projekt naszego systemu. Zakładka *Address Map* będzie zawierała informacje i ustawienia związane z przestrzenią adresową, czyli pod jakimi adresami będzie widoczna dla naszego rdzenia np. pamięć RAM, a pod jakimi rejestry do sterowania portami IO. W ostatniej karcie *Interconnect Requirements* znajdują się dodatkowe ustawienia połączeń, których nie musimy zmieniać.

Teraz przyjrzyjmy się zakładce *System Contents* i jej kolumnom:

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **95777**, pass: **53wtjyf6**



Rysunek 2. Opcje źródła sygnału zegarowego

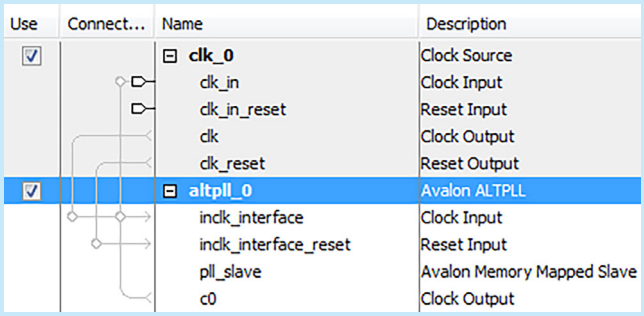
- *Use* – pozwala na włączenie lub wyłączenie danego komponentu z systemu. Może być to przydatne np. gdy chcemy usunąć jakiś jego moduł, ale tylko tymczasowo, aby później nie musieć od nowa go konfigurować – wtedy wystarczy tylko odznaczyć to pole przy jego nazwie.
- *Connections* – tu będziemy wykonywać połączenia pomiędzy komponentami.
- *Name* – zawiera nazwę modułu, którą możemy zmieniać po 2-krotnym kliknięciu lub danego portu tego modułu. Warto nazywać moduły tak, abyśmy mogli szybko zorientować się, do czego służą. Szczególnie, jeśli umieszczamy w systemie kilka modułów tego samego typu.
- *Description* – jak sama nazwa wskazuje to opis danego wiersza naszej tabelki.
- *Export* – w tej kolumnie widzimy nazwy portów jeśli są one „wyeksportowane", czyli po prostu widoczne na zewnątrz systemu mikroprocesorowego. Możemy porównać to do tego, jakie piny ma budowany przez nas procesor. Na przykład, będziemy chcieli mieć dostęp do wejścia sygnału zegarowego, ale niekoniecznie potrzebny nam będzie dostęp do magistrali danych naszego układu. Jeśli chcemy jakiś port uwidocznić należy dwukrotnie kliknąć w odpowiednim miejscu. Możemy także zmieniać nazwy, jakie będą miały nasze połączenia na zewnątrz układu. Aby cofnąć „eksportowanie" musimy kliknąć PPM na polu, i najechać na *Connections* a następnie odznaczyć zaznaczony element.
- *Clock* – w tej kolumnie widoczne są informacje o sygnale zegarowym z jakiego korzysta dany moduł.
- *Base* oraz *End* – definiują adres początkowy i końcowy w przestrzeni adresowej dla danego modułu.
- *IRQ* – w przypadku korzystania z przerwań definiujemy tu numer przerwania, z jakiego korzysta dany moduł. Omówimy to w jednej z kolejnych części tej serii artykułów.

Pozostałe kolumny nie są w tej chwili dla nas interesujące.

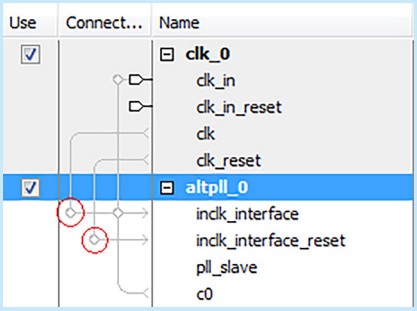
Z CZEGO ZBUDUJEMY NASZ PROCESOR? Po zapoznaniu się z terenem możemy przejść do pracy i wreszcie zbudować nasz własny system mikroprocesorowy. Zacznijmy od przyjrzenia się temu, co zostało dodane automatycznie do naszego systemu – modułowi *clk_0*. Ma on 2 wyeksportowane porty – *clk* oraz *reset*. Będą to wejścia tych sygnałów do naszego procesora (bo przecież prawie każdy procesor ma piny do podłączenia sygnału zegarowego oraz sygnału resetującego). Kolejne 2 porty, już nie widoczne na zewnątrz, to kopie odpowiednich sygnałów, które będą rozprowadzone po naszym procesorze.

Aby podejrzeć ustawienia danego modułu wystarczy 2-krotnie kliknąć w wierszu obok jego nazwy. Zróbmy tak dla modułu *clk_0*. Po prawej stronie otworzy się nowa karta *Parameters* (**rysunek 2**). Opcje tego modułu nie są rozbudowane – możemy wpisać częstotliwość wejściową (*Clock frequency*), która podamy na to wejście, ewentualnie usuwając zaznaczenie obok *Clock frequency known* poinformować Qsys, że nie znamy tej częstotliwości oraz wybrać, na którym zboczu sygnału zegarowego ma być próbkowany sygnał resetu (*Reset synchronus edge*) – u nas reset będzie asynchroniczny. Zawsze trochę pomocy i ewentualne odnośniki do zasobów online na temat danego modułu możemy znaleźć po kliknięciu na *Details*. Zasoby są oczywiście w języku angielskim, dlatego jak zawsze przy takich okazjach zachęcam do jego nauki.

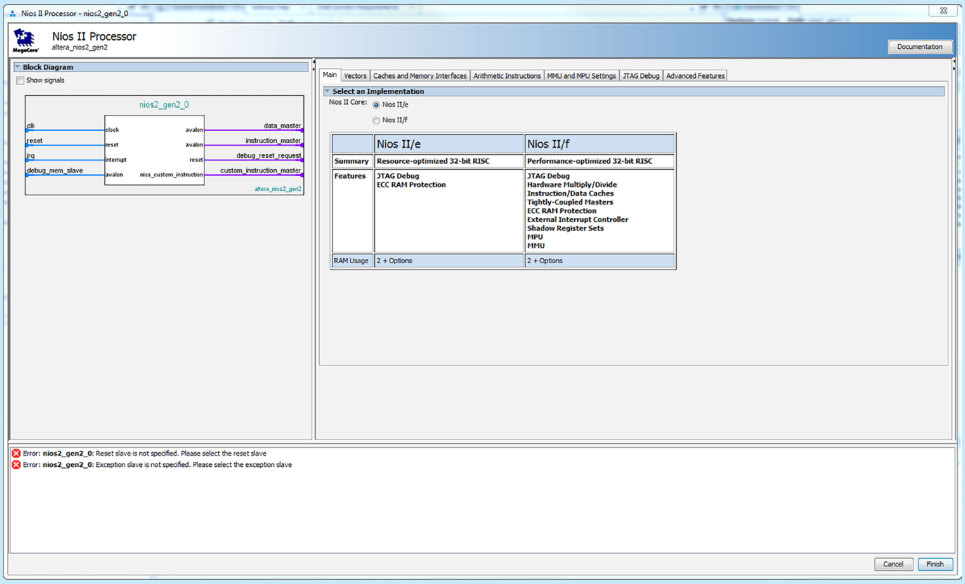
Jedyne, co na tym etapie musimy w tym miejscu zmodyfikować, to wprowadzić częstotliwość, którą będziemy taktowali nasz procesor. Będzie to 10 MHz, ponieważ taki oscylator jest zamontowany na płytce maXimator. Klikając przy otwartym okienku *Parameters*



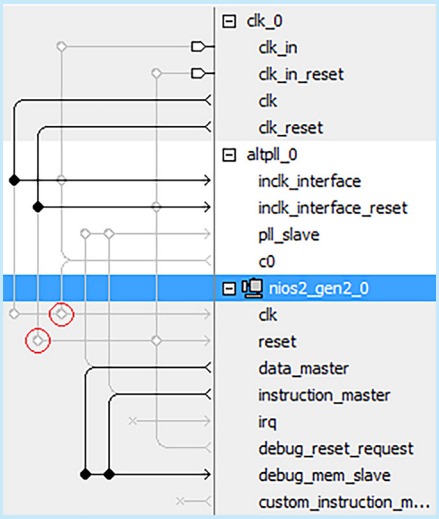
Rysunek 3. Widok systemu po dodaniu modułu pętli PLL



Rysunek 4. Zaznaczone miejsca wykonania połączeń



Rysunek 5 Okno konfiguracji procesora Nios II



Rysunek 6. Nowe połączenia do wykonania po dodaniu procesora

na pola poszczególnych portów modułu możemy czasem zobaczyć dodatkowe i interesujące informacje czy ustawienia.

Czy 10MHz to nie za mało? Możliwe, że tak, ale nic nie stoi na przeszkodzie, żeby do układu wstawić mnożnik częstotliwości i uzyskać praktycznie dowolną częstotliwość do taktowania procesora. Dokładnie tak, jak w bardziej rozbudowanych procesorach, np. ARM. Dlaczego jednak nie zamontujemy sobie od razu oscylatora o pożądanej częstotliwości? Odpowiedź jest prosta – im wyższa częstotliwość, tym większe problemy. Musielibyśmy dobierać odpowiednio parametry ścieżek, aby np. nie stanowiły one filtrów przy wyższych częstotliwościach. Znacznie wygodniej jest doprowadzić do układu niższą częstotliwość i powielić ją wewnętrznie zostawiając najtrudniejszą pracę inżynierom, którzy zaprojektowali dla nas „czarną kostkę".

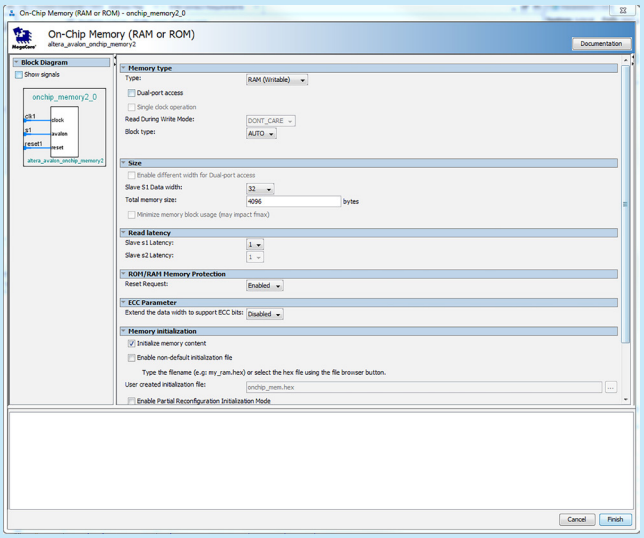
Aby dodać do układu pętlę PLL w okienku wyszukiwania *IP Catalog* wpisujemy *ALTPLL* i następnie dwukrotnie klikamy na *Avalon ALTPLL*. Po chwili pojawia się okno konfiguracji pętli PLL. Omawiając budowę oscyloskopu w poprzednich częściach tłumaczyłem krok-po-kroku co i jak, a tu konfiguracja przebiega w sposób identyczny. Wpisujemy jako częstotliwość wejściową 10 MHz, w kolejnym ekranie odznaczamy opcję tworzenia wejścia *assert* oraz wyjścia *locked*, w następnych ekranach nic nie zmieniamy, aż do momentu ustawienia zegara *c0*. Wybieramy opcję wprowadzania częstotliwości i wpisujemy w okienko częstotliwości oczekiwanej 50 MHz. Możemy kliknąć 2-krotnie *Finish*. Jeśli w przyszłości potrzebowalibyśmy dodatkowych zegarów o wyższych częstotliwościach – musielibyśmy je skonfigurować jako kolejne wyjścia pętli PLL (c1..4).

W widoku systemu pojawił się moduł pętli PLL oraz możliwość wykonania połączeń pomiędzy wejściem zegara a nowo dodaną pętlą PLL (**rysunek 3**). Następnie klikamy na punktach zaznaczonych na **rysunku 4** czerwonymi kółkami. Dzięki temu wykonujemy połączenia między wyjściem sygnału zegarowego *Clock Source*, a wejściem zegara dla pętli PLL oraz łączymy sygnał resetujący. Jeśli istniałaby potrzeba usunięcia połączenia wystarczy ponownie kliknąć na „kropkę". Jednocześnie warto obserwować pole *Messages* u dołu ekranu. Zawsze, gdy brakuje jakichś połączeń znajdziemy tam stosowny komunikat z wyjaśnieniem. Warto tez zapisać nasz system, np. pod nazwą tutorial. Polecam ponadto utworzenie osobnego katalogu w folderze naszego projektu i dopiero tam zapisanie naszego systemu.

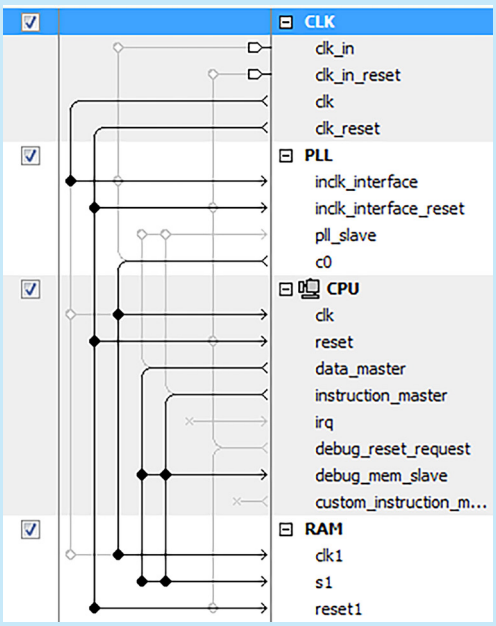
Teraz przyszła kolej na dodanie kluczowego komponentu naszego systemu – procesora NIOS II. Wyszukujemy go w *IP Catalog* a następnie wybieramy *Nios II Procesor*. Zostanie wyświetlone okno konfiguracji naszego procesora, jak na **rysunku 5**. Jak wspominałem, wybierzemy prostszy i darmowy rdzeń Nios II/e, po czym kilkamy na *Finish*.

W widoku systemu zostanie pokazany nasz nowo dodany procesor – musimy teraz wykonać odpowiednie połączenia jego portów – a jest już trochę więcej możliwości niż poprzednio (**rysunek 6**). Na razie łączymy tylko sygnał taktujący rdzeń z wyjściem pętli PLL oraz sygnał resetu. Co dalej? W naszym systemie brakuje jeszcze pamięci.

Zwykle w mikrokontrolerach znajdują się 2 osobne pamięci – RAM oraz pamięć programu. U nas moglibyśmy ten schemat powielić, jednak możemy wybrać metodę prostszą i bardziej elastyczną – zastosować jedną pamięć RAM. Dlaczego ta metoda jest bardziej elastyczna? Ponieważ na etapie kompilacji programu pamięć ta zostanie wirtualnie podzielona na część w której będzie przechowywany program oraz na część dostępną jako faktyczna pamięć RAM. Ma to szczególne znaczenie w układach takich jak nasz, gdzie dysponujemy stosunkowo ograniczoną ilością pamięci operacyjnej. Czy to jednak oznacza, że nasz procesor po odłączeniu zasilania straci swój program? Na szczęście nie, o czym opowiem jednak nieco później.



Rysunek 7. Konfiguracja RAM



Rysunek 8. Widok połączeń po dodaniu pamięci RAM

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **95777**, pass: **53wtjyf6**

Wyszukujemy w *IP Catalog* RAM, a następnie wybieramy *On-Chip Memory (RAM or ROM)* – **rysunek 7**.

Na chwile obecną musimy dokonać tutaj 2 zmian (Rys. 7.). Po pierwsze zwiększamy rozmiar pamięci (*Total memory size*) do 32768 oraz odznaczamy opcję *Initialize memory content*. Tę drugą czynność wykonujemy „tymczasowo" – to właśnie ta opcja pozwoli nam na zachowanie programu pomimo odłączenia zasilania.

Teraz znów kolej na połączenia – sygnał zegarowy i reset łączymy tak jak w procesorze, zaś port *s1* łączymy jednocześnie z portami *data_master* oraz *instruction_master* procesora. Jak nietrudno się domyśleć pierwszy port jest odpowiedzialny za transmisję danych (czyli połączenie z pamięcią RAM i innymi komponentami), zaś drugi – przesyła instrukcje (nasz program) do wykonania przez rdzeń. Przy okazji warto zmienić też nazwy komponentów na bardziej przyjazne. Efekt koniecznych działań pokazano na **rysunku 8**.

8 Errors	
tutorial.CPU	Reset slave is not specified. Please select the reset slave
tutorial.CPU	Exception slave is not specified. Please select the exception slave
tutorial.CPU.data_master	ID.control_slave (0x0..0x7) overlaps RAM.s1 (0x0..0x7fff)
tutorial.CPU.data_master	JTAG_UART.avalon_jtag_slave (0x0..0x7) overlaps ID.control_slave (0x0..0x7)
tutorial.CPU.data_master	RAM.s1 (0x0..0x7fff) overlaps JTAG_UART.avalon_jtag_slave (0x0..0x7)
tutorial.CPU.data_master	CPU.debug_mem_slave (0x800..0xfff) overlaps RAM.s1 (0x0..0x7fff)
tutorial.CPU.instruction_master	ID.control_slave (0x0..0x7) overlaps RAM.s1 (0x0..0x7fff)
tutorial.CPU.instruction_master	JTAG_UART.avalon_jtag_slave (0x0..0x7) overlaps ID.control_slave (0x0..0x7)
1 Warning	
tutorial.PLL	PLL.pll_slave must be connected to an Avalon-MM master
3 Info Messages	
tutorial.sysid_qsys_0	System ID is not assigned automatically. Edit the System ID parameter to provide a unique ID
tutorial.sysid_qsys_0	Time stamp will be automatically updated when this component is generated.
tutorial.jtag_uart_0	JTAG UART IP input clock need to be at least double (2x) the operating frequency of JTAG TCK on board

Rysunek 9. Błędy na obecnym etapie tworzenia systemu

Teoretycznie, mamy już wszystkie wymagane komponenty, jednak dodamy jeszcze dwa. Nie są one niezbędne, ale znacząco ułatwią nam pracę, gdy zaczniemy pisać pierwszy program na nasz procesor.

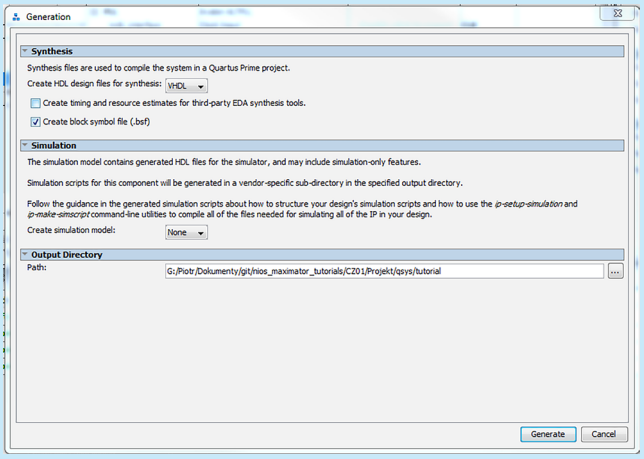
Po pierwsze, dodajemy *System ID Peripherial*. W jego konfiguracji możemy wpisać dowolną wartość identyfikatora systemu. Po co taki komponent? Odpowiedź jest banalnie prosta – pomoże on debuggerowi sprawdzić, czy komunikuje się z właściwym procesorem podobnie jak programator sprawdza model podłączonego procesora podczas wgrywania wsadu np. do AVR czy STM32. Co więcej sprawdzać będzie on także datę syntezy naszego procesora. A wszystko to po to, aby mieć pewność, że program, który wgrywamy został skompilowany z właściwym kompletem *BSP* (*Board Support Package*), czyli w uproszczeniu mówiąc zestawem plików definiujących, pod jakimi adresami znajdują się jakie komponenty naszego systemu oraz jak są one skonfigurowane. Nie trudno wyobrazić sobie sytuację, w której wgrywamy do naszego procesora program, który dane zamiast wysłać na port szeregowy, kieruje w bliżej nieokreślone miejsce, bo zapomnieliśmy zaktualizować plików *BSP*.

Łączymy zegar i reset jak poprzednio, zaś port *control_slave* łączymy tylko z *data_master* naszego CPU.

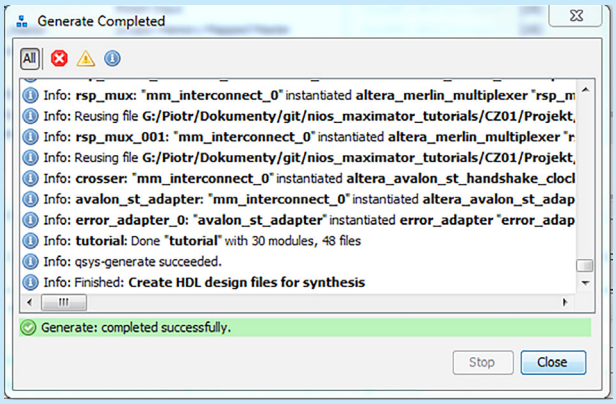
Drugim elementem dodatkowym będzie *JTAG UART* – dodajemy go z domyślnymi ustawieniami. Reset i zegar łączymy w znany już nam sposób. *avalon_jtag_slave* łączymy z *data_master* naszego CPU. Dodatkowo, łączymy porty *irq* nowo dodanego modułu i rdzenia. Te ostatnie odpowiedzialne są za system przerwań, o którym opowiem w innej części tego kursu. Moduł ten pozwoli nam na komunikację między procesorem i komputerem przez debugger JTAG, bez konieczności używania dodatkowych połączeń interfejsu UART.

Uff, sporo pracy za nami, ale z okienka *Messages* zerkają na nas czerwone błędy… I to aż 8! Teraz przyjrzyjmy się im i spróbujmy po kolei przeanalizować wszystkie komunikaty i usnąć błędy, które są tutaj bardzo dobrze opisane (**rysunek 9**).

Może zacznijmy analizę od przyjaznych, zielonych informacji. Pierwsze dwie odnoszą się do dodanego przez nas komponentu służącego identyfikacji procesora przez debugger. Ostatnia dotyczy wymagań sygnału zegarowego taktującego moduł *JTAG UART* względem zegara debuggera.



Rysunek 10. Zmiana języka na VHDL

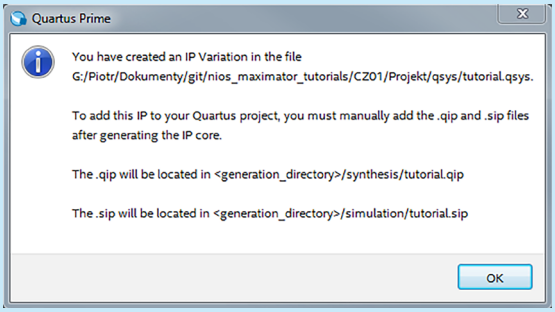


Rysunek 11. Komunikat o poprawnym wygenerowaniu systemu

Wyżej widzimy jedno samotne ostrzeżenie, które mówi o tym, że port sterujący pętlą PLL nie został połączony z CPU. Wykonajmy zatem jego połączenie z *data_master*. Mimo, że nie będziemy korzystać z tego modułu z poziomu procesora, warto dla porządku wykonać to połączenie.

Teraz zostały wyświetlone jeszcze dodatkowe dwa błędy. Osiem ostatnich błędów tyczy się *„overlap"*, czyli nakładania się przestrzeni adresowych modułów. Jeśli w naszym edytorze zobaczymy na kolumny *Base* oraz *End*, to bez zaskoczenia to potwierdzimy – wszystkie moduły mają ten sam adres bazowy. To samo zobaczymy w zakładce *Address Map*. Zatem kalkulator w rękę i poprawiamy? Na szczęście *Qsys* oferuje nam pomoc – z menu *System* wybieramy *Assign Base Addresses*.

Jeszcze tylko 2 błędy, odnoszą się one do faktu, że nie podaliśmy naszemu procesorowi miejsca w pamięci, z którego ma



Rysunek 12. Komunikat informujący o wygenerowaniu plików systemu

altera_reserved_tck	Input				3.3-V LVTTL
altera_reserved_tdi	Input				3.3-V LVTTL
altera_reserved_tdo	Output				3.3-V LVTTL
altera_reserved_tms	Input				3.3-V LVTTL
clk_clk	Input	PIN_L3	2	B2_N0	3.3-V LVTTL
reset_reset_n	Input	PIN_B10	8	B8_N0	3.3-V LVTTL

Rysunek 13. Prawidłowe ustawienie lokalizacji pinów. Wyprowadzenia związane z interfejsem JTAG zostaną ustawione automatycznie przez Quartus'a

rozpoczynać program, oraz do którego ma skoczyć w wypadku zgłoszenia wyjątku. Klikamy zatem na CPU w edytorze (jeśli nie jest widoczne okienko *Parameters* musimy kliknąć 2-krotnie), przechodzimy do zakładki *Vectors* i wybieramy w obu przypadkach *RAM.s1*.

Wreszcie nasz system jest bezbłędny i możemy kliknąć na *Generate HDL…* (**rysunek 10**). Język syntezy zmieniamy z *Verilog* na *VHDL* i klikamy *Generate*. W okienku, które zostanie wyświetlone (i powinno poinformować o bezbłędnym zakończeniu operacji) klikamy *Close*. Zostaje wyświetlone kolejne okno, w którym po dłuższej chwili powinien być pokazany komunikat *Generate: completed successfully* (**rysunek 11**). Teraz nie pozostało nic innego jak kliknąć *Finish*. Po tej operacji będzie wyświetlony komunikat, jak na **rysunku** 12.

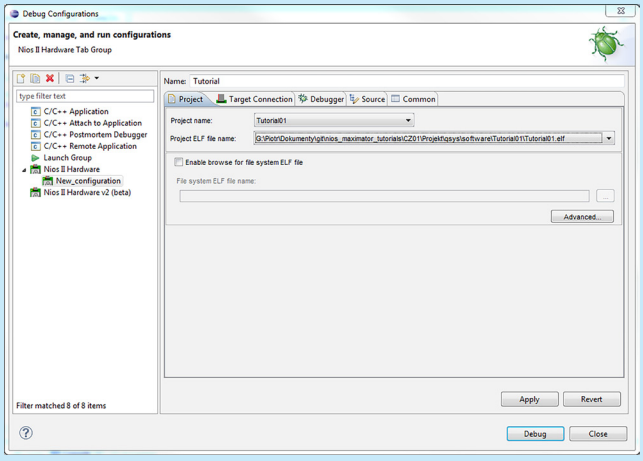
Komunikat w jednoznaczny sposób mówi nam, że należy ręcznie dodać wygenerowane pliki do projektu. Klikamy *Projekt → Add/Remove Files in Project…*, klikamy na … obok okienka *File name* i lokalizujemy plik *.qip* wskazany przez komunikat z rys. 12. Teraz musimy powiadomić syntezer, że procesor, którego opis znajduje się w pliku *tutorial.qip* jest głównym obiektem w syntezie (*Top-level entity*). W *Project Navigator* wybieramy widok *Files*, klikamy na pliku *tutorial.qip* prawym klawiszem i wybieramy *Set as Top-Level Entity*.

Teraz możemy kliknąć *Analysis & Synthesis*. Po paru chwilach projekt zostanie zsyntezowany i będziemy mogli przypisać jego wyprowadzenia do fizycznych pinów układu. W *Pin Planner* ustawiamy lokalizację clk_clk oraz reset_reset_n odpowiednio na piny L3 (oscylator 10 MHz na płytce) oraz B10 (przycisk DEV_CLRn na płytce). Ustawiamy standard wszystkich pinów (w tym 4 odnoszących się do interfejsu JTAG) na 3.3-V LVTTL (**rysunek 13**). Uruchamiamy *Compile Design*, a po zakończonym procesie – zaprogramować nasz układ (na razie plikiem *.sof*).

Piszemy pierwszy program

W tej chwili w naszym układzie FPGA zaczął (nie) działać pierwszy samodzielnie zaprojektowany system mikroprocesorowy. Czas go w pełni ożywić pisząc pierwszy program.

TWORZYMY PROJEKT I GENERUJEMY BSP. Zaczynamy bez przydługich wstępów od uruchomienia środowiska programistycznego: *Tools → Nios II Software Build Tools for Eclipse*. Na początek zostanie pokazane okno wyboru środowiska pracy – klikamy OK. Jeśli w przyszłości będziecie dużo pracować, możecie utworzyć kilka środowisk pracy (*Workspace*). Podobnie po aktualizacji Quartusa



Rysunek 14. Konfiguracja debuggera

do nowszej wersji możecie wybrać środowisko, którego używaliście w wersji starszej – domyślnie każda nowsza wersja wskazuje na inny folder, oznaczony jej numerem.

Wybieramy *File → Nios II Application and BSP form Template*. W kolejnym oknie wskazujemy jako plik *SOPC Information…* plik *tutorial.sopcinfo* z folderu, w którym zapisaliśmy nasz system za pomocą *Qsys*. Po chwili program przeanalizuje definicję naszego systemu. Wpisujemy nazwę projektu, np. Tutorial01. Jako *Project template* wybieramy *Hello World Small*.

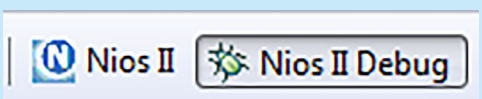
Dzięki temu zostaną pokazane de facto dwa projekty – jeden zwykły, a drugi z sufiksem _bsp. Ten pierwszy to projekt, w którym tworzymy nasz program, zaś drugi to pakiet BSP, o którym wcześniej wspominałem. Zawiera on wygenerowane na podstawie pliku *.sopcinfo* definicje adresów, obecnych w systemie komponentów (plik *system.h*) oraz drivery do tych komponentów. Klikamy PPM na nazwie projektu BSP i wybieramy *Nios II → Generate BSP*. Czynność tę należy powtórzyć po każdej zmianie systemu dokonanej w Qsys – powoduje to aktualizację zawartości BSP względem pliku *.sopcinfo*.

Aby zobaczyć program, otwieramy plik *hello_world_small.c*. Możemy zmienić wcześniej nazwę tego pliku na *main.c*, za pomocą opcji po kliknięciu na niego PPM. Program ten wyśle na standardowe wyjście, którym „z automatu" stał się nasz JTAG UART trochę tekstu, po czym zakończy działanie. Teraz klikamy PPM na naszym podstawowym projekcie (*Tutorial01*) i wybieramy *Build Project*.

Zaczynamy debuggowanie

Aby wgrać nasz program do układu i jednocześnie zacząć jego debuggowanie musimy najpierw utworzyć konfigurację. W tym celu wybieramy *Run → Debug Configurations…* i klikamy dwukrotnie na *Nios II Hardware* po lewej stronie okna (**rysunek 14**). Następnie z listy *Project name* wybieramy nasz projekt. Możemy także zmienić nazwę konfiguracji.

W zakładce *Target Connection* klikamy po prawej stronie *Refresh Connections*. Jeśli system nie zwróci błędów, znaczy, że wszystko przebiegło OK. Klikamy następnie *Apply*. Moglibyśmy teraz kliknąć na *Debug*, ale zatrzymajmy się w tym miejscu i przypomnijmy o komponencie *System ID*, który dodaliśmy wcześniej do projektu. Jeśli klikniemy w obecnie otwartym oknie *System ID Properties…*, to zobaczymy parametry, które są przechowywane przez wspomniany komponent i jego adres bazowy.



Rysunek 15. Przyciski wyboru widoku

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **95777**, pass: **53wtjyf6**

Rysunek 16. Przyciski kontroli pracy programu – od lewej: uruchomienie, wstrzymanie, zakończenie sesji debuggowania

Nieco niżej widzimy też pola, które pozwalają nam wgrać „na siłę" program, pomimo niezgodności parametrów identyfikacyjnych (pole *System ID checks*). Klikamy na *Debug*. W zależności od ustawień komputera może być konieczne zezwolenie oprogramowaniu na dostęp do sieci. W oknie, które pojawi się (z zapytaniem, czy chcemy przełączyć się na widok debuggowania), odpowiadamy twierdząco. Między widokami zawsze możemy przełączać się za pomocą przycisków w prawym górnym rogu okna – pokazano je na **rysunku 15**.

Jeśli wszystko przebiegło bez błędów (a tak być powinno, jeśli podążaliście według wskazówek) program został wgrany do procesora (do naszej uwspólnionej pamięci operacyjnej i programu). Program został także zatrzymany na pierwszej instrukcji w funkcji main. Aby go uruchomić klikamy na strzałkę w pasku narzędziowym, pokazaną na **rysunku 16**.

Teraz pora na omówienie tego widoku. U dołu widoczna jest *Nios II Console* – czyli terminal naszego JTAG UART! – powinien wyświetlać się tam komunikat wysłany przez nasz pierwszy program: *Hello form Nios II!* Powyżej znajduje się podgląd programu (gdzie możemy zobaczyć, gdzie wylądował nasz program jeśli wciśniemy przycisk wstrzymania pracy programu). Jeszcze wyżej widzimy podgląd stanu debuggera, zaś okna po jego prawej stronie służą do podglądu zmiennych, pamięci, rejestrów oraz kontroli breakpointów. Pamiętajmy, aby zawsze po zakończonej pracy kliknąć na przycisk zakończenia sesji debuggowania – inaczej, jeśli będziemy chcieli rozpocząć kolejną sesję program poinformuje nas o błędach. Po przyciśnięciu tego przycisku możemy wrócić do normalnego widoku.

Podsumowanie

Po bardzo długich bojach z systemem Nios II dotarliśmy wspólnie do końca tej części szkolenia. W jego czasie zapoznaliśmy się z systemem Nios II oraz środowiskiem Qsys. Następnie stworzyliśmy pierwszy działający system mikroprocesorowy i uruchomiliśmy na nim program.

Na pewno pojawią się pytania – dlaczego tyle czasu poświęciliśmy na omawianie krok po kroku tych wszystkich działań? Odpowiedź jest prosta – chcę, aby każdy zrozumiał dlaczego klikamy tak a nie inaczej i wybieramy taką a nie inną opcję. Ponadto utworzony dzisiaj system będę traktować jako referencyjny – w kolejnych „spotkaniach" będę odnosił się do tego systemu i prosił o stworzenie dokładnie takiego układu, lub będę jedynie wymieniał różnicę (np. w rozmiarze pamięci itp.) i dodatkowe elementy, które będziemy poznawać.

Na początku te wszystkie połączenia i ilość informacji zawarta w tym artykule może przytłaczać, ale polecam powtórzenie wszystkich czynności (ze zrozumieniem!) kilka razy – ćwiczenie czyni mistrza i w końcu będziecie budowali takie systemy dokładnie rozumiejąc czemu podłączam do pamięci RAM port (magistralę) danych i instrukcji, a do komponentu portu szeregowego tylko magistralę danych.

Zadanie domowe po tej „lekcji" to – zrozumienie wszystkiego, co dziś się wydarzyło i nabycie umiejętności samodzielnego powtórzenia tych czynności oraz powtórzenie/zdobycie wiadomości nt. debuggowania programów z wykorzystaniem środowiska Eclipse.

Pozdrawiam i życzę pomyślnej pracy z Nios II.

Piotr Rzeszut