NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (2)

Wyjście na świat, czyli obsługa GPIO i inne praktyczne porady

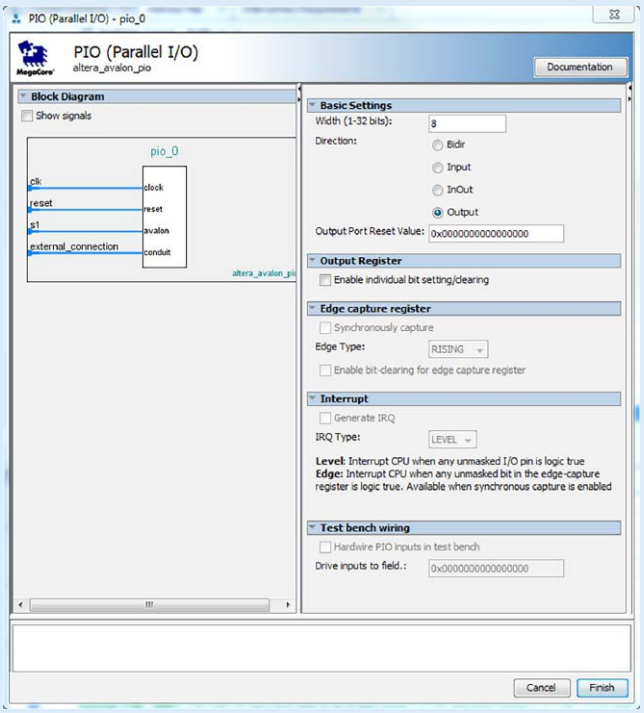
W poprzednim artykule zaimplementowaliśmy w układzie FPGA podstawowy system mikroprocesorowy oparty o rdzeń NIOS II. Cóż jednak po naszej ciężkiej pracy, skoro nasz procesor potrafi tylko i wyłącznie komunikować się ze środowiskiem programistycznym, nie potrafiąc przy tym nawet zamigać diodą... Teraz przyszedł czas, aby to zmienić!

Aby móc zrealizować zadanie, o którym wspominałem we wstępie przyda się, aby nasz procesor wyposażony został w kilka wyprowadzeń GPIO (ang. *General Purpose Input Output* – wejście/wyjście ogólnego przeznaczenia). W tym momencie, albo możemy przygotować nowy system według wskazówek z poprzedniego spotkania, albo wykorzystać przygotowany wtedy projekt. Tym razem podążymy tą drugą ścieżką, aby przy okazji nauczyć się modyfikowania projektu Qsys.

Czas na GPIO

Na początek otwieramy plik projektu Quartusa (**.qpf*). Wybieramy *File* → *Open...* i odszukujemy plik projektu systemu (**.qsys*).

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



Rysunek 1. Okno konfiguracji komponentu/modułu wejścia/wyjścia równoległego

Teraz wyszukujemy komponent *PIO (Parallel I/O)*, a następnie klikamy *Add…* Zostaje wyświetlone okno konfiguracji tego komponentu, jak na **rysunku 1**. Teraz pokrótce omówmy ustawienia tego komponentu:

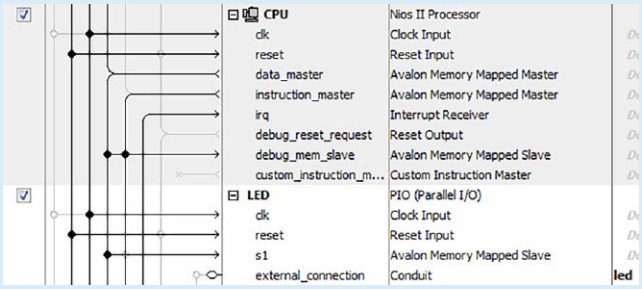
- *Width* – określamy tutaj szerokość, czyli liczbę bitów nowo tworzonego portu. Możemy utworzyć pojedynczy port o szerokości od 1 do 32 bitów.
- *Direction* – określa kierunek (konfigurację) potu. Do wyboru mamy opcje:
  - *Bidir* – port 2-kierunkowy, w którym kierunek każdego z pinów może być wybrany oddzielnie (wejściowy lub wyjściowy) z poziomu programu i może być w dowolnym jego momencie zmieniany. Pozostałe opcje nie pozwalają na kontrolę kierunku pinów z poziomu programu.
  - *Input* – wszystkie piny portu są wejściami.
  - *InOut* – tworzone są 2 porty o podanej szerokości, z których jeden jest wejściowy a drugi wyjściowy. Czyli w efekcie podając np. szerokość 4 bitów zostanie utworzony 4-bitowy port wejściowy i drugi 4-bitowy port wyjściowy.
  - *Output* – wszystkie piny portu są wejściami.
- *Output Port Reset Value* – wartość, którą przyjmą piny portu po restarcie układu. Podajemy ją w zapisie heksadecymalnym. Ustawienie takie ma szczególne znaczenie w sytuacjach, gdy chcemy mieć pewność jak zachowa się nasze urządzenie przed uruchomieniem się programu, aby np. silnik, który jest sterowany określonym pinem nie zaczął się przez chwilę kręcić.
- *Enable individual bit setting/clearing* – to ustawienie powoduje dodanie dodatkowych rejestrów, które umożliwiają szybkie ustawienie stanu wysokiego lub niskiego na danym wyjściu, bez konieczności odczytu poprzedniego stanu portu, wykonania operacji logicznych w programie i zapisu nowego stanu.
- *Synchronously capture* – powoduje dodanie do naszego portu modułu wykrywania zboczy na pinach. Informacje o tym, że wystąpiło określone zbocze dostępne są potem w odpowiednim rejestrze, bez konieczności ciągłego monitorowania pinu w programie.
  - *Edge Type* – wybieramy typ zbocza, które chcemy wykrywać (*FALLING* – opadające, *RISING* – narastające, *ANY* – narastające i opadające).
- *Enable bit-clearing for edge capture register* – umożliwia kasowanie pojedynczych bitów w rejestrze wykrywania zboczy.
- *Generate IRQ* – powoduje, że dodany zostaje moduł generowania przerwań wraz z rejestrem maskującym. Rejestr maskujący pozwala na wybranie, które piny mają generować przerwanie, a które mają być ignorowane. O przerwaniach dowiemy się nieco szerzej w kolejnych częściach naszego kursu.
- *IRQ Type* – wybieramy kiedy ma być generowane przerwanie:
  - *LEVEL* – stan wysoki na pinach, które są wybrane poprzez rejestr maskujący powoduje generowanie przerwania tak długo, jak dowolny z tych pinów pozostaje w stanie wysokim.
  - *EGDE* – korzysta z opisanego wcześniej modułu wykrywania zboczy (który musi być aktywny i w którym musimy wybrać typ zbocza). Przerwanie jest generowane tak długo, aż nie zostaną skasowane wszystkie bity w rejestrze wykrywania zboczy.
- *Test bench wiring* – pozwala na zadanie konkretnej stałej wartości pinów wejściowych w czasie symulacji

Pełną dokumentację każdego modułu zawsze możemy uzyskać klikając na *Documentation* i później korzystając z podanych tam linków do dokumentacji online. Czasem po załadowaniu strony musimy poczekać dłuższy moment, aż przeglądarka przejdzie do interesującej nas sekcji w dokumentacji – kliknięcie myszą czy poruszenie kółkiem myszy może spowodować że nie zostaniemy automatycznie przeniesieni do interesującej nas sekcji.

W naszym przykładowym projekcie chcemy sterować czterema LEDami na płytce maXimatora, zatem wybieramy port o szerokości 4 bitów pracujący jako wyjście, z wartością początkową 0xA (binarnie 1010). Zaznaczamy także opcję *Enable individual bit…*. Klikamy na *Finish*.

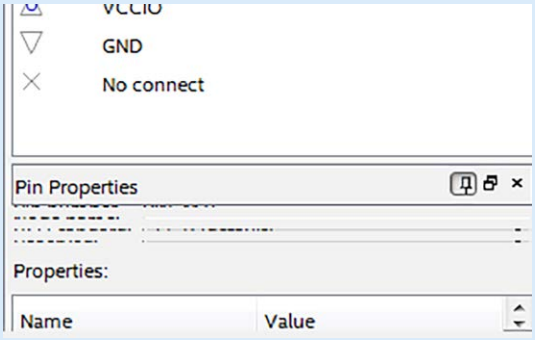
Następnie musimy wykonać połączenia odpowiednich portów naszego modułu. Łączymy *clk* z tym samym sygnałem, co zegar taktujący rdzeń. Podobnie postępujemy z sygnałem *reset*. Port *s1* łączymy z *data_master* rdzenia. Pozostało nam jeszcze dokonać „eksportowania" *external_connection*. W tym celu klikamy dwukrotnie na *Double-click to export* w odpowiednim wierszu i podajemy nazwę, pod jaką nasz port ma być widoczny na zewnątrz procesora. Warto też zmienić nazwę modułu na bardziej znaczącą – ja w obu miejscach wpisałem *LED*. Połączenia powinny wyglądać, jak na **rysunku 2**. Moduły możemy przenosić w górę lub w dół na naszym „schemacie" za pomocą niebieskich strzałek w lewej części okna *System Contents*.

Po zakończeniu tej czynności wybieramy *System* → *Assign Base Addresses*, a następnie klikamy na *Finish*. W oknie *Save System*



Rysunek 2. Podłączenie modułu PIO z systemem

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



Rysunek 3. Lokalizacja okna Pin/Node Properties

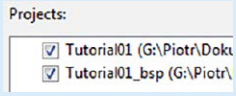
Completed wybieramy *Close*, a w kolejnym odpowiadamy twierdząco na pytanie o to, czy wygenerować system po modyfikacjach. Następnie klikamy w kolejnych (już znajomych oknach) *Generate* a potem *Finish*. Zostanie pokazane również nam znane okno informujące o konieczności dołączenia wygenerowanego pliku do projektu, jednak jest on już dodany do projektu, więc wystarczy zamknąć wyświetlone okno.

Czego jeszcze brakuje nam od strony sprzętowej? Rzecz jasna – przypisania utworzonego przez nas przed chwilą portu do konkretnych wyprowadzeń układu FPGA. Wzorem poprzedniego projektu najpierw uruchamiamy proces *Analysis & Synthesis*, a po jego zakończeniu uruchamiamy *Assignments → Pin Planner*. Tam powinniśmy zobaczyć nowe piny o nazwach *led_export[…]*. Przypisujemy im lokalizacje zgodne z projektem płytki, czyli *M16, N16, P16, R16*. Następnie musimy ponownie zmienić standard napięciowy wyjścia na *3.3-V LVTTL*. Aby jednak nie robić tego dla każdego pinu z osobna, możemy nieco przyśpieszyć ten proces. Na początku wybieramy klikamy prawym przyciskiem myszy na jednym z interesujących nas pinów i zaznaczany *Node Properties*. Niestety, to okno często zostaje wyświetlone „zwinięte" pod *Pin Legend*, po prawej stronie. Czasem także różny jest tytuł tego okna (*Node* albo *Pin Properties* – **rysunek 3**). Musimy rozciągnąć okienko, po czym (z wciśniętym Ctrl lub Shift) możemy zaznaczyć wszystkie interesujące nas piny i ustawić w niedawno otwartym okienku wspomniany wcześniej standard wyprowadzeń.

Zamykamy *Pin Planner'a* i uruchamiamy *Compile Design*, po czym wgrywamy konfigurację do układu. Właśnie wyposażyliśmy nasz procesor w 4 piny GPIO przyłączone do diod LED na naszej płytce, z których 2 powinny świecić, a dwie pozostawać zgaszone.

Kolej na miganie – piszemy pierwszy program

Teraz przyszła kolej na napisanie pierwszego programu obsługującego właśnie dodany przez nas port PIO. W tym celu, jak poprzednio, uruchamiamy środowisko *NIOS II Software Build Tools for Eclipse*. Nasz projekt z poprzednich zajęć powinien być już czytany do środowiska. Jeśli jednak tak nie jest musimy go zaimportować. Wybieramy *File → Import…*, z folderu *General* wybieramy *Existing Project into Workspace*. Następnie, w *Search root directory* wybieramy folder, który zawiera podfoldery z nazwą projektu oraz nazwą projektu z sufiksem *_bsp*. W polu *Projects* powinny pojawić się 2 pozycje, z zaznaczonymi polami wyboru, jak na **rysunku 4**. Klikamy *Finish* i po chwili w naszym środowisku mamy oba projekty. Jeśli prostokąty wyboru są „wyszarzone" oznacza to, że nazwy projektu się dublują (patrz porada poniżej).



Rysunek 4. Projekty gotowe do zaimportowania do środowiska

Jeśli decydujemy się kopiować projekty to należy pamiętać, że w obrębie danego *workspace* nie mogą znajdować się 2 projekty o takich samych nazwach. Rozwiązaniem tego jest albo utworzenie innego *workspace*, albo usunięcie projektów, z którymi nie chcemy pracować (poprzez kliknięcie na nich PPM i wybranie *Delete*). Uważajmy jednak, aby opcja *Delete project contents on disk …* nie była zaznaczona, gdyż w przeciwnym razie nieodwracalnie skasujemy pliki projektu i nie będzie możliwe ich ponowne zaimportowanie. Jeśli wszystko zrobimy poprawnie i w przyszłości będziemy chcieli wrócić do jakiegoś projektu o dublującej się nazwie – wystarczy wykonać import, jak pokazano powyżej. Równie dobrze moglibyśmy teraz wykonać nowy projekt, jednak kontynuując/importując ten, nad którym pracowaliśmy

Należy zachować ostrożność w momencie modyfikowania systemu – plik z programem nie aktualizuje się samoczynnie (za każdym razem trzeba wywołać *mem_init_generate*) oraz nie jest wykonywane sprawdzenie, czy został on skompilowany pod prawidłowy system. Najlepiej używać opcji dołączania programu do konfiguracji układu FPGA tylko i wyłącznie w sytuacji, gdy chcemy finalnie zaprogramować układ FPGA, a nie w fazie projektowania.

```
/*
 * LED configuration
 *
 */

#define ALT_MODULE_CLASS_LED altera_avalon_pio
#define LED_BASE 0x11000
#define LED_BIT_CLEARING_EDGE_REGISTER 0
#define LED_BIT_MODIFYING_OUTPUT_REGISTER 1
#define LED_CAPTURE 0
#define LED_DATA_WIDTH 4
#define LED_DO_TEST_BENCH_WIRING 0
#define LED_DRIVEN_SIM_VALUE 0
#define LED_EDGE_TYPE "NONE"
#define LED_FREQ 50000000
#define LED_HAS_IN 0
#define LED_HAS_OUT 1
#define LED_HAS_TRI 0
#define LED_IRQ -1
#define LED_IRQ_INTERRUPT_CONTROLLER_ID -1
#define LED_IRQ_TYPE "NONE"
#define LED_NAME "/dev/LED"
#define LED_RESET_VALUE 10
#define LED_SPAN 32
#define LED_TYPE "altera_avalon_pio"
```

Rysunek 5. Fragment pliku system.h zawierający definicje związane z dodanym modułem PIO

```
#define IOADDR_ALTERA_AVALON_PIO_DATA(base)
#define IORD_ALTERA_AVALON_PIO_DATA(base)
#define IOWR_ALTERA_AVALON_PIO_DATA(base, data)

#define IOADDR_ALTERA_AVALON_PIO_DIRECTION(base)
#define IORD_ALTERA_AVALON_PIO_DIRECTION(base)
#define IOWR_ALTERA_AVALON_PIO_DIRECTION(base, data)

#define IOADDR_ALTERA_AVALON_PIO_IRQ_MASK(base)
#define IORD_ALTERA_AVALON_PIO_IRQ_MASK(base)
#define IOWR_ALTERA_AVALON_PIO_IRQ_MASK(base, data)

#define IOADDR_ALTERA_AVALON_PIO_EDGE_CAP(base)
#define IORD_ALTERA_AVALON_PIO_EDGE_CAP(base)
#define IOWR_ALTERA_AVALON_PIO_EDGE_CAP(base, data)


#define IOADDR_ALTERA_AVALON_PIO_SET_BIT(base)
#define IORD_ALTERA_AVALON_PIO_SET_BITS(base)
#define IOWR_ALTERA_AVALON_PIO_SET_BITS(base, data)

#define IOADDR_ALTERA_AVALON_PIO_CLEAR_BITS(base)
#define IORD_ALTERA_AVALON_PIO_CLEAR_BITS(base)
#define IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(base, data)
```

Rysunek 6. Makra ułatwiające sterowanie modułem PIO

```
#include "system.h"
#include "sys/alt_stdio.h"
#include "sys/alt_sys_wrappers.h"
#include "altera_avalon_pio_regs.h"

int main()
{
  alt_putstr("Hello from Nios II!\n");

  int i = 0;

  /* Event loop never exits. */
  while (1){
      ALT_USLEEP(1000000);
      IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, ~i);
      i++;
  }

  return 0;
}
```

Rysunek 7. Przykładowy kod testowy

chcę pokazać, jak należy postępować w wypadku zmiany strony sprzętowej procesora.

Na początku zaczynamy od kliknięcia PPM na projekt z końcówką *_bsp* (u mnie *Tutorial01_bsp*) i wybrania *Nios II → Generate BSP*. Spowoduje to aktualizację pakietu BSP, która uwzględni dokonane przez nas zmiany. Możemy podejrzeć plik *system.h* z projektu BSP i znaleźć tam ważne informacje o nowo dodanym module (**rysunek 5**). Widzimy tu m. in. informacje o adresie bazowym, o tym, jakie funkcje włączyliśmy podczas konfiguracji lub jaki jest stan początkowy tego portu.

W projekcie BSP w folderze *drivers/inc/* znajdziemy plik *altera_avalon_pio_regs.h*. Zawiera on definicje makr, które ułatwią nam komunikację z modułem (**rysunek 6**). Makra parami pozwalają na odczyt i zapis odpowiednich rejestrów, o których pisałem wcześniej.

Teraz pora na napisanie kilku linijek programu – pokazano je na **rysunku 7**. Pierwszy plik nagłówkowy jest nam już dobrze znany, drugi definiuje funkcje odpowiedzialne za standardowy strumień wejścia/wyjścia (czyli u nas funkcję, która pozwala na wysyłanie znaków przez debugger). Kolejny plik nagłówkowy zapewnia nam dostęp m.in. do funkcji generującej opóźnienie. Zawartość ostatniego z plików nagłówkowych także znamy. W programie deklarujemy zmienną *i*, którą za każdym obiegiem pętli inkrementujemy. Dodatkowo jej zanegowaną wartość wyświetlamy na naszych diodach. Dlaczego jednak zanegowaną? Powód tej negacji tkwi w sposobie zasilania LEDów – są one zaświecane wtedy, gdy wyjście układu FPGA jest wyzerowane.

Zanim wgramy program do naszego procesora muszę jeszcze przekazać jedną ważną uwagę, dotyczącą funkcji *ALT_USLEEP*. Po pierwsze, jest ona dosyć niedokładna – sekunda wcale nie jest sekundą. Po drugie – generowanie opóźnień w ten sposób w programie jest naganną praktyką. W skrócie – poza celami demonstracyjnymi, jeśli w programie znajdzie się tego typu funkcja, oznacza to, że program jest napisany źle. Zwykle do odmierzania czasu używa się timerów, ale o tym opowiem nieco później.

Teraz, ponieważ zmieniliśmy strukturę procesora (a co za tym idzie zmieniły się parametry identyfikacyjne systemu), musimy wejść do *Run → Debug Configurations*, tam w zakładce *Target Connection* kliknąć na *Refresh Connections* (oczywiście na liście po lewej stronie powinna być podświetlona konfiguracja

> Programowanie plikiem **.pof* należy wykonywać także wyłącznie w momencie, gdy chcemy, żeby nasza aplikacja działała w pełni autonomicznie. Pamięć FLASH ma ograniczoną ilość cykli programowania oraz proces ten przebiega znacznie wolniej, niż w przypadku pamięci RAM i plików **.sof*.

Tutorial). Klikamy na *Appy* a potem *Close*. Teraz zapisujemy wszystkie zmiany w plikach (*File → Save All*), kompilujemy nasz projekt (*Project → Build All*) i na koniec, jak poprzednio, rozpoczynamy debuggowanie i uruchamiamy program za pomocą zielonej strzałki.

Diody na płytce powinny zacząć bardzo powoli migać, realizując zliczanie w kodzie binarnym. Sukces! Po zakończonej obserwacji nie zapomnijmy zatrzymać debuggera za pomocą czerwonego kwadratu.

Jeszcze wejście

Teraz dla wprawy wprowadźmy jeszcze do naszego systemu dodatek – 3 bitowy port wejściowy, który posłuży nam do odczytywania danych z przycisków (możemy podpiąć je przewodami do układu lub skorzystać z nakładki maXimator Expander).

Otwórzmy *Qsys*, a w nim nasz projekt. Dodajemy do niego komponent *PIO*, z szerokością 3 bitów oraz funkcją wejścia. Resztę parametrów pozostawiamy bez zmian, wykonujemy połączenia analogicznie jak w wypadku modułu *LED*. Zmieńmy nazwę modułu np. na *SW*, oraz pod taką samą nazwą dokonajmy „eksportowana" *external_connection*. Na sam koniec *System → Assign Base Addresses*.

Identycznie, jak poprzednio zamykamy *Qsys* jednocześnie wykonując generowanie systemu. Uruchamiamy *Analysis & Synthesis* (jeśli program zapyta nas, czy chcemy powtórzyć ten proces, odpowiadamy twierdząco – przecież dokonaliśmy modyfikacji), po czym w *Pin Planner* musimy przypisać do wyprowadzeń *sw_export[…]* odpowiednie piny układu. Niech do najmniej znaczącego bitu będzie podpięty będzie przycisk *RES* (pin *R15*), do kolejnego bitu *L* (*B16*) a do najwyższego bitu *R* (*B15*). Nie zapomnijmy o zmianie standardu na *3.3-V LVTTL*. Po zakończeniu operacji nowo dodane linie powinny wyglądać jak na **rysunku 8**.

Teraz uruchamiamy *Compile Design*. W czasie oczekiwania na jej zakończenie możemy przejść do środowiska Eclipse i wykonać generowanie BSP (mam nadzieję, że pamiętacie, gdzie trzeba kliknąć). Jeśli wszystko wykonaliśmy prawidłowo w pliku *system.h* powinny zostać wyświetlone wpisy odpowiedzialne za nowo dodany moduł SW.

W pliku *main.c* dokonujemy prostej modyfikacji (**rysunek 9**) – zmniejszamy czas oczekiwania oraz dodajemy funkcję odczytującą stan przycisków i w zależności od niego zwiększająca, zmniejszającą lub resetującą wartość naszego licznika. Mam nadzieję, że analiza tego programu w ramach ćwiczenia nie sprawi wielkich trudności.

Teraz czas na zapisanie zmian i skompilowanie programu. Po niej, ponieważ znów zmodyfikowaliśmy nasz procesor, musimy wejść do stawień debuggera i wykonać *Refresh Connections*. Ooops… Coś nie działa tak jak powinno (**rysunek 10**)?

Wręcz przeciwnie – system pokazuje nam informację, wynikającą z faktu, że… nie wgraliśmy nowego systemu do układu FPGA – próbujemy wgrać nowy program do starego procesora, który przecież nie miał żadnych portów wejściowych. Na szczęście,

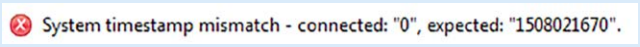
sw_export[2]	Input	PIN_B15	6	B6_N0	3.3-V LVTTL
sw_export[1]	Input	PIN_B16	6	B6_N0	3.3-V LVTTL
sw_export[0]	Input	PIN_R15	5	B5_N0	3.3-V LVTTL

Rysunek 8. Prawidłowe ustawienia pinów wejściowych

```
while (1){
    ALT_USLEEP(100000);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, ~i);

    unsigned int sw = IORD_ALTERA_AVALON_PIO_DATA(SW_BASE);
    if(!(sw & (1<<0))){
        i = 0;
    }else if(!(sw & (1<<1))){
        i++;
    }else if(!(sw & (1<<2))){
        i--;
    }
}
```

Rysunek 9. Modyfikacje programu po dodaniu modułu obsługującego przyciski



Rysunek 10. Błąd podczas próby połączenia z procesorem

system uratował nas przed tą katastrofą. Szybko naprawiamy błąd wgrywając konfigurację układu FPGA i potem rozpoczynając procedurę uruchomienia debuggowania od początku.

Nasz program napisany jest znów w sposób demonstracyjny i nie reaguje na przyciśnięcia przycisku, ale za każdym obiegiem pętli sprawdza, czy trzymamy przycisk. To znów nie jest dobra praktyka (podobnie jak opóźnienie), ale w tej sytuacji stosuję ją abyśmy szybko mogli sprawdzić efekty naszej pracy, bez zbędnego wysiłku programistycznego.

Refresh Connectins jest wymagane tylko wtedy, gdy zmieniamy projekt systemu w Qsys. Jeśli nie zmieniamy systemu, a jedynie zajmujemy się oprogramowaniem po prostu

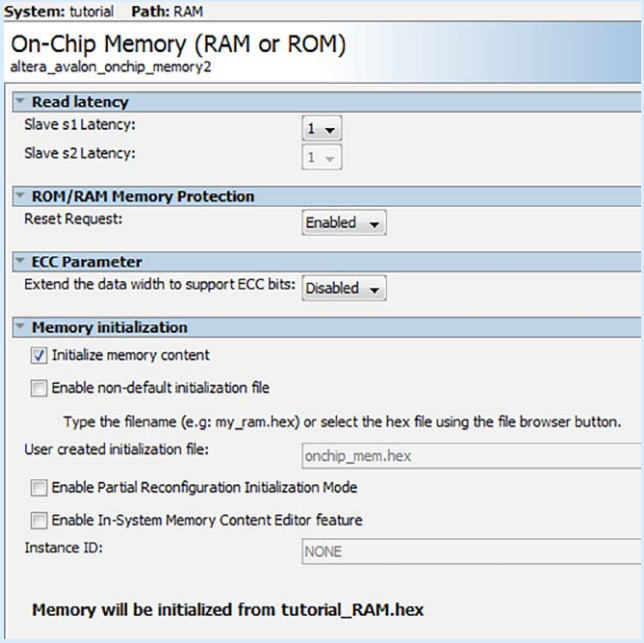
Programie, nie uciekaj, czyli jak sprawić, aby nasz program pozostał w układzie na dłużej

Nasze dotychczasowe wysiłki wydają się być ekstremalnie nietrwałe – wystarczy odłączyć zasilanie i tracimy całą naszą pracę. Aby temu zapobiec musimy po pierwsze zdać sobie sprawę z tego, że nasz projekt teraz składa się z 2 części – konfiguracji układu FPGA, zawierającej opis naszego procesora oraz oprogramowania działającego w procesorze. O ile, aby trwale skonfigurować układ FPGA wystarczy za pomocą programatora wgrać plik **.pof* zamiast **.sof* to, aby program także stał się trwały, musimy dołączyć go do plików konfiguracyjnych procesora.

Na początek, jeszcze w *Eclipse*, klikamy PPM na nasz projekt (bez *_bsp*) i wybieramy *Make Targets* → *Build…*. Wybieramy *mem_init_generate* i klikamy na *Build*. Spowoduje to wygenerowanie pliku z naszym programem, który zaraz będziemy wykorzystywać dalej.

Przechodzimy do *Quartus'a* i wybieramy *Assignments* → *Device*. Tam klikamy na *Device and Pin Options…*, w nowo otwartym oknie wybieramy kategorię *Configuration* i w polu *Configuration mode* wskazujemy *Single Uncompressed Image with Memory Initialization…*. Dwukrotnie klikamy w *OK*. Dzięki tej zmianie umożliwimy inicjalizację pamięci RAM po podłączeniu zasilania wskazaną przez nas zawartością. Z tą możliwością otwieramy *Qsys*, a w nim nasz projekt. Klikamy na nasza pamięć RAM w widoku systemu, a następnie zaznaczamy opcję *Initialize memory content* (**rysunek 11**).

Znów zamykamy Qsys po drodze generując nową wersję systemu. W *Eclipse* uruchamiamy *Generate BSP*, a następnie dokładnie jak poprzednio generujemy plik z naszym programem (*mem_init_generate*). Wracamy do *Quartusa* i wybieramy *Project* → *Add/Remove files in Project…*, wskazujemy na plik *meminit.qip*, który powinien znajdować się w folderze z projektem naszego oprogramowania, w podfolderze *mem_init*. Spowoduje to dodanie do ustawień projektu ścieżki, w której znajduje się plik z programem, dzięki czemu oprogramowanie będzie mogło zostać dołączone do projektu.



Rysunek 11. Wybór opcji inicjalizacji pamięci naszego procesora

Na koniec rozpoczynamy *Compile Design*. Po zakończeniu całej procedury możemy wgrać do naszego układu FPGA plik **.sof* lub **.pof*. Pierwszy z nich spowoduje wgranie konfiguracji układu FPGA wraz z naszym programem sterującym diodami do pamięci RAM układu FPGA – program zacznie od razu działać po zakończeniu programowania, bez konieczności używania debuggera. Drugi plik powoduje wgranie konfiguracji układu FPGA wraz z naszym programem do pamięci Flash, dzięki czemu nasz mikroprocesor wraz z programem będą na stałe zapisane i będą uruchamiać się po podłączeniu zasilania.

Biorąc pod uwagę powyższe uwagi, warto znów otworzyć nasz projekt *Qsys* (można to zrobić także będąc w widoku *Hierarchy* w *Project Navigator* – wystarczy kliknąć PPM na najbardziej nadrzędnym elemencie i wybrać *Qsys*) i usunąć zaznaczenie z pola dotyczącego inicjalizacji pamięci RAM.

Zadania domowe

Im dalej w las tym więcej drzew, zatem teraz czeka Was nieco więcej zadań niż ostatnio:

1. Należy zapoznać się z funkcją ustawiania i kasowania poszczególnych bitów. Poeksperymentujcie z makrami z pliku *altera_avalon_pio_regs.h*: *IOWR_ALTERA_AVALON_PIO_CLEAR_BITS oraz IOWR_ALTERA_AVALON_PIO_SET_BITS*. Pierwsze z nich powoduje ustawienie na „0" wszystkich tych bitów, dla których w argumencie funkcji znajduje się „1", zaś druga funkcja ustawia te bity na „1".
2. Należy do naszego systemu dodać 2 kolejne porty PIO, które będą portami wyjściowymi:
   - 4-bitowy, do sterowania wyborem wyświetlacza na płytce maXimator Expander (DS1..4),
   - 8-bitowy, do sterowania segmentami wyświetlacza na płytce ekspandera (A..G oraz DP).
3. Należy przypomnieć sobie zasadę multipleksowania wyświetlaczy 7-segmentowych (można korzystając z posiadanych już informacji spróbować napisać prosty program, sterujący tym wyświetlaczem).

Jak pewnie niektórzy się już domyślają, w kolejnej części zmierzymy się ze sterowaniem programowym tego wyświetlacza oraz tematyką timerów i przerwań.

Piotr Rzeszut, AGH