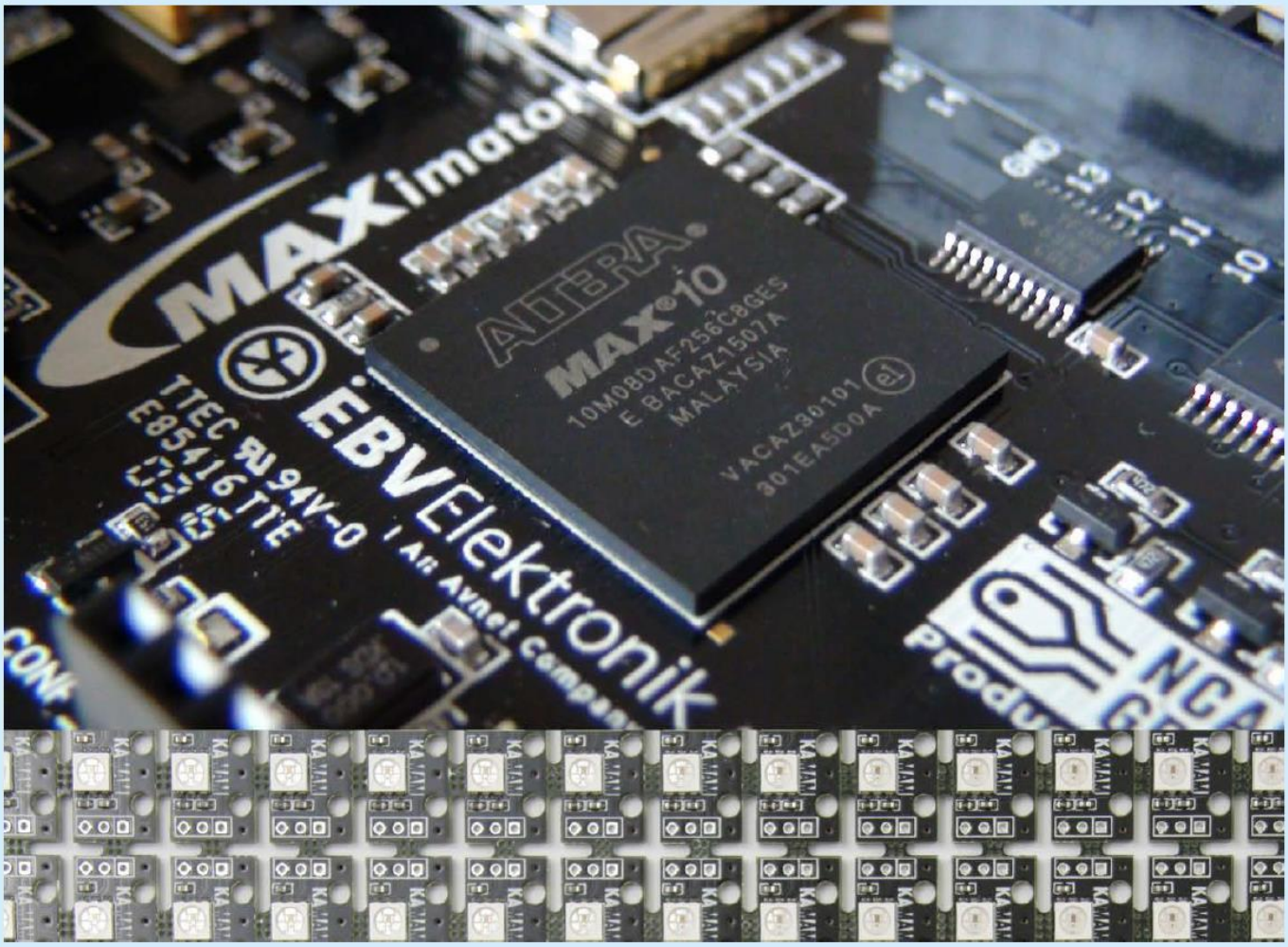


NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (11)

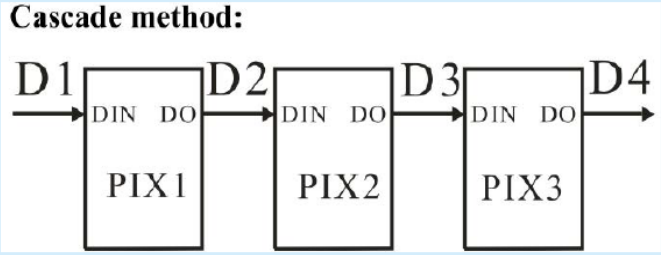
Własne moduły w systemie – WS2812 i kilka zaawansowanych tricków

Zgodnie z zapowiedzią zajmiemy się dzisiaj obsługą diod RGB WS2812, a przy okazji poznamy kolejne tajniki tworzenia własnych modułów – generowanie przerwań oraz budowę portów typu master dla magistrali Avalon.

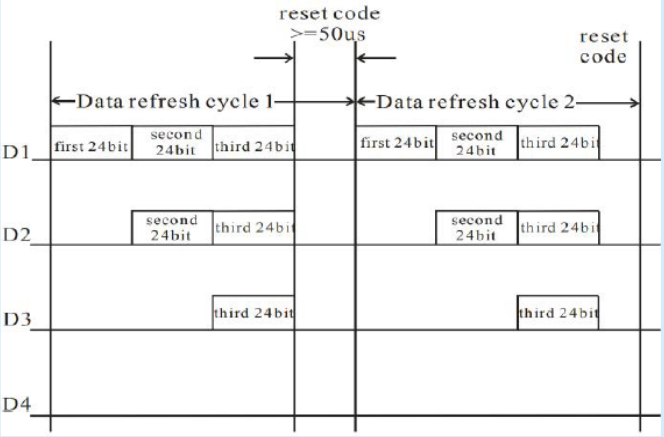
W sumie moglibyśmy korzystając z modułu PWM wykonanego parę spotkań temu uzyskać kolory – wystarczyłoby tylko podpiąć diody RGB (z odpowiednimi rezystorami) i viola! Ale na dłuższą metę to rozwiązanie dosyć problematyczne, gdyż potrzebujemy dużą ilość wyprowadzeń naszego układu do sterowania większa ilością diod. Rozwiązaniem tego problemu mogłoby być zastosowanie zewnętrznego sterownika PWM (czasem takowe są nawet wyposażone w źródła prądowe i do całego układu zdolnego do sterowania kilkunastu diod wystarczy 1 rezystor ustalający właściwy prąd), jednak możemy kupić od pewnego czasu genialne diody RGB z wbudowanym sterownikiem – WS2812.

Diody te mają jedynie 4 wyprowadzenia – dwa do zasilania oraz dwa do transmisji danych (jeden pin wejściowy i drugi wyjściowy). A transmisja danych odbywa się w bardzo ciekawy sposób – diody połączone są szeregowo i cały ich łańcuszek podpinamy do układu za pomocą jednego wyprowadzenia (**rysunek 1**). Każda z diod działa

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



Rysunek 1. Sposób łączenia diod WS2812(B). Źródło: Nota katalogowa WS2812



Rysunek 2. Oto jak diody przekazują dane między sobą. Źródło: Nota katalogowa WS2812

w ten sposób, że po otrzymaniu sygnału resetującego (jak ten i inne sygnały wyglądają powiem nieco później) odbiera pierwsze 24 bity danych przeznaczone dla niej, a wszelkie kolejne dane po prostu przekazuje na swoje wyjście, tak aby kolejna dioda mogła je odebrać – czy to nie genialne?

Teraz już wystarczy tylko wiedzieć… jak przesyłać dane i generować sygnał resetu! Ten ostatni generujemy utrzymując linię danych w stanie niskim przez czas co najmniej 50 µs. Zera i jedynki nadajemy za pomocą impulsów stanu wysokiego i niskiego o odpowiednim czasie trwania (**rysunek 3**). Czasy trwania poszczególnych fragmentów kodu dla diod WS2812 i WS2812B pokazano w **tabeli 1**.

Co jednak jest warte zauważenia, że stosując czasy średnie jesteśmy w stanie mieścić się w tolerancji dla obu modeli diodek! Dodatkowo, z dużym prawdopodobieństwem, nawet stosując parametry dla jednego z modeli, ten drugi także powinien działać.

Czas znów pomigać LED-ami

Na początek, dokładnie tak jak poprzednio, wykonamy moduł WS2812 z użyciem już przygotowanego pliku WS2812/WS2812.vhd. Tym razem od razu w pliku implementujemy możliwość definiowania ilości podpiętych diodek (i co z tym związane szerokości magistrali adresowej). Pamiętamy o tym, aby w czasie procesu tworzenia nowego komponentu zdefiniować odpowiednie parametry magistrali *Avalon*, a w szczególności te z zakładek *Timing* oraz *Pipelined Transfers*. Ostatecznie kluczowe ustawienia powinny być takie, jak pokazano

Tabela 1. Czasy trwania poszczególnych fragmentów kodu dla diod WS2812 i WS2812B

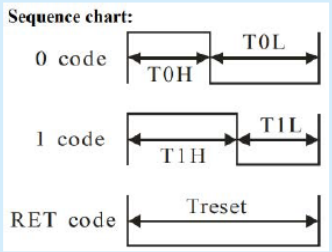
Parametr	WS2812 [µs]	WS2812B [µs]	Średnia [µs]	Tolerancja [ns]
T0H	0,35	0,35	0,35	±150
T0L	0,80	0,90	0,85	±150
T1H	0,70	0,90	0,80	±150
T1L	0,60	0,35	0,475	±150
Treset	>50	>50		

Zasilanie modułów z WS2812(B)

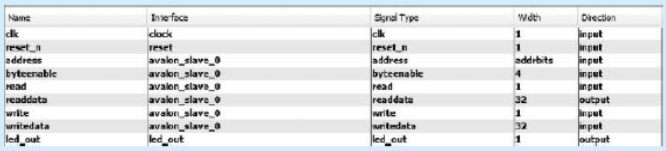
Uwaga! O ile 2 diody WS2812 możemy spokojnie zasilić ze złącza USB komputera, o tyle już 40 sztuk znacznie przekracza jego możliwości – przy wszystkich diodach włączonych mogą pobierać nawet prawie 2 A! W takiej sytuacji koniecznie należy na module odciąć zasilanie 5 V od złącza Arduino i zasilić go z zewnętrznego zasilacza 5 V o wydajności prądowej minimum 2 A. Oczywiście musimy bezwzględnie pamiętać o połączeniu masy zasilacza z masą maXimatora (i komputera). Co ważne – nie zakładajcie nigdy, że na raz włączycie tylko kilka diodek, albo nigdy nie użyjecie pełnej jasności – bo wystarczy jeden błąd w kodzie i wszystko zaświeci. A potem zgaśnie…

na **rysunkach 4…6**. Po tych operacjach na liście powinien pojawić się nasz nowy moduł, gotowy do użycia. Dodajemy go do systemu, podając liczbę diod równą 2 oraz ilość bitów adresu wynoszącą 3. Następnie uzupełniamy konieczne połączenia (**rysunek 7**).

Jeśli mamy moduł *NeoPixel Shield* możemy zdefiniować ilość diod na 40 a liczbę bitów



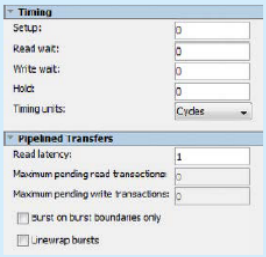
Rysunek 3. Przebiegi czasowe dla transmisji "0", "1" oraz sygnału resetu (RET). Źródło: Nota katalogowa WS2812



Name	Interface	Signal Type	Width	Direction
clk	clock	clk	1	input
reset_n	reset	reset_n	1	input
address	avalon_slave_0	address	addrbits	input
byteenable	avalon_slave_0	byteenable	4	input
read	avalon_slave_0	read	1	input
readdata	avalon_slave_0	readdata	32	output
write	avalon_slave_0	write	1	input
writedata	avalon_slave_0	writedata	32	input
led_out	led_out	led_out	1	output

Rysunek 4. Widok panelu Signals podczas dodawania komponentu WS2812

adresu na 6. W tej chwili wykonujemy *Generate HDL…* a potem *Analysis & Synthesis*. Teraz jesteśmy gotowi na zdefiniowanie wyprowadzeń w *Pin Planner* – oczywiście w zależności od tego czy skorzystamy z naszego ekspandera, czy modułu z 40-stoma diodami musimy zdefiniować inne wyprowadzenie (w pierwszym wypadku jest to *C15*, w drugim np. *G16* (**rysunek 8**), choć zawsze musimy to sprawdzić z dokumentacją płytki, którą trzymamy w ręce).



Rysunek 5. Kluczowe parametry interfejsu Avalon slave dla komponentu WS2812

Teraz możemy już skompilować projekt, a w *Eclipse* wygenerować projekt oprogramowania. Tam, tradycyjnie przy korzystaniu z modułów sprzętowych nie mamy zbyt wiele do zrobienia – właściwie jedynie wyliczyć i ustawić czasy do generowania przebiegów (**listing 1**) i już możemy ustawiać „kolorki" naszych diodek.

Rysunek 6. Sygnały i ich funkcje w module WS2812

Ale zaraz, zaraz! Jak właściwie działa nasz moduł napisany w VHDL? Tym razem odpowiadając na to pytanie nie zamieszczę kodu źródłowego (który jest dosyć długi, choć w istocie prosty), ale posłużę się graficzną interpretacją maszyny stanu, jaka realizuje generowanie impulsów sterujących (**rysunek 9**). Przejście do każdego z następnych stanów następuje rzecz jasna po upłynięciu odpowiedniego czasu, czego nie zaznaczałem na diagramie, aby nie zaciemniać obrazu.

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

Mając nadzieję, że jeden obraz wyraża więcej niż tysiąc słów, na tym skończę omawianie tego modułu. Jednakże warto zwrócić uwagę na jedną, szalenie ważną rzecz – mianowicie zużycie zasobów. O ile dla modułu dla 2 diodek nie powinniśmy mieć większego problemu, o tyle moduł dla 40 diodek zajmuje… no właśnie ile zasobów?

Listing 1. Ustawienie czasów do generowania przebiegów

```
IOWR_32DIRECT(WS2812_0_BASE,  0, (NIOS2_CPU_FREQ * 350LL)   / 1000000000LL);
IOWR_32DIRECT(WS2812_0_BASE,  4, NIOS2_CPU_FREQ * 900LL    / 1000000000LL);
IOWR_32DIRECT(WS2812_0_BASE,  8, NIOS2_CPU_FREQ * 900LL    / 1000000000LL);
IOWR_32DIRECT(WS2812_0_BASE, 12, NIOS2_CPU_FREQ * 350LL    / 1000000000LL);
IOWR_32DIRECT(WS2812_0_BASE, 16, NIOS2_CPU_FREQ * 60000LL / 1000000000LL);
IOWR_32DIRECT(WS2812_0_BASE, 20, 0x00FF00);
IOWR_32DIRECT(WS2812_0_BASE, 24, 0x0000FF);
```

Zużycie zasobów – czas na RAM

Aby to sprawdzić przechodzimy do *Compilation Report* (jeśli nie jest widoczna taka zakładka to otwieramy ją z menu *Processing*) i tam przechodzimy do *Fitter* ➔ *Resource Section* ➔ *Resource Utilization by Entity*. Okaże się, że moduł do obsługi 40 diod zajmuje ponad 2000 komórek logicznych (z 8000 dostępnych w naszym układzie) – dosyć nieporęczne… Jeśli porównamy to z modułem obsługującym 2 diodki (około 500 komórek) to można uznać, że marnujemy bardzo dużo zasobów układu tylko na pamiętanie koloru diod. Ale czy musimy to robić w komórkach logicznych? Odpowiedź brzmi: nie – lepiej wykorzystać do tego dedykowane komórki RAM w układzie FPGA (w tym modelu są to komórki M9K), czyli po prostu użyć pamięci RAM.

Na realizację takiego pomysłu mamy 2 sposoby. Pierwszy z nich to wstawienie pamięci ram wewnątrz naszego modułu – jest to jednak rozwiązanie wymagające większej ilości zabiegów w celu łatwej zmiany ilości dostępnych komórek. Drugie rozwiązanie to wykorzystanie 2-portowej pamięci RAM wstawianej z poziomu *Platform Designer* oraz wyposażenie naszego modułu w port *Avalon Memory Mapped Master*, za pomocą którego będzie on zdolny do odczytu zawartości pamięci. Dzięki temu projektując układ będziemy w stanie szybko dobrać stosowny rozmiar pamięci.

Projektujemy i używamy modułu z portem *Avalon Master*

Zacznijmy zatem od utworzenia nowego pliku (*WS2812_RAM/WS2812_RAM.vhd*) i zacznijmy od umieszczenia w nim kopii starego modułu. Na początek musimy zmienić (w 3 miejscach) nazwę modułu na nową (*WS2812_RAM*). Teraz już zostało nam przeprowadzić właściwe modyfikacje:

Usunąć parametr definiujący liczbę bitów adresu (teraz w module adresować będziemy jedynie zawsze stałą ilość bitów kontrolnych) i szerokość szyny adresowej ustawić na 3 bity.

Usunąć tablicę zawierającą rejestry danych dla kolejnych diod i zastąpić ja prostym sygnałem o szerokości 24 bitów. Następnie we wszystkich miejscach usunąć indeksowanie elementu tablicy, pozostawiając jedynie indeksowanie bitów.

Rysunek 7. Połączenia z modułem WS2812

ws2812_led_out	Output	PIN_C15		3.3-V LVTTL
ws2812_neo_led_out	Output	PIN_G16		3.3-V LVTTL

Rysunek 8. Wyprowadzenia dla modułu maXimator expander oraz NeoPixel Shield

W naszym projekcie musimy zadbać o to, aby nazwy modułów, nazwy instancji (w szczególności kolumna *Name* w *Platform Designer*) nie dublowały się. Nie możemy zatem jednocześnie mieć modułu (*entity*) o nazwie *Diodak*, a następnie dodać go w *Platform Designer* i nadać mu nazwę (*Name*) *Diodak*. Spowoduje to błąd na etapie kompilacji projektu.

Listing 2. Dodanie sygnałów magistrali Avalon Master

```
--avalon memory-mapped master
m_address    : out std_logic_vector(16 downto 0);
m_byteenable : out std_logic_vector(3 downto 0);
m_read       : out std_logic;
m_readdata   : in std_logic_vector(31 downto 0);
m_write      : out std_logic;
m_writedata  : out std_logic_vector(31 downto 0);
```

Listing 3. Zapis/odczyt danych do/z pamięci

```
-- będziemy zawsze prowadzić odczyt całego słowa 32-bitowego
-- z pamięci - ustawiamy linie
m_write <= '0';
m_writedata <= (others => '0');
m_read <= '1';
m_byteenable <= "1111";
-- nasze dane będą zawsze ostatnio pobranymi danymi z pamięci,
-- z pominięciem najstarszych 8 bitów (aby ułatwić adresowanie)
data <= m_readdata(23 downto 0);
```

```
signal data                 : std_logic_vector(23 downto 0);
```

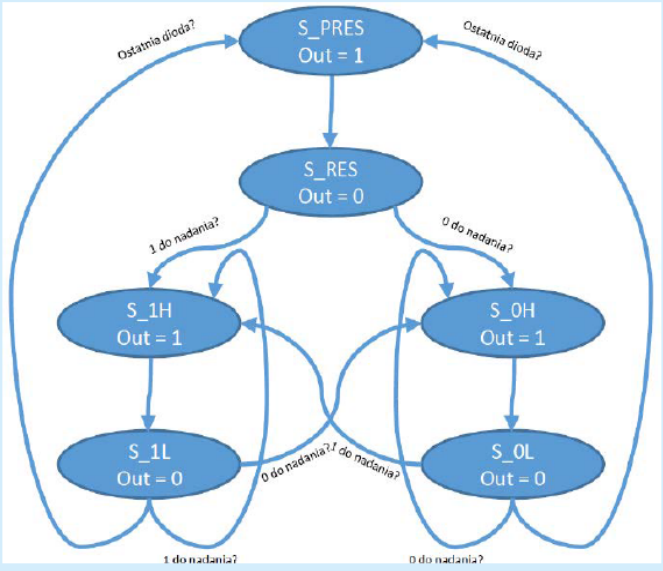
Dodać sygnały portu magistrali *Avalon Memory Mapped Master*. Są to w rzeczy samej takie same sygnały jak używane w układzie *slave* z tą różnicą, że mają przeciwne kierunki (**listing 2**).

Statycznie (poza *process*) ustawiamy linie tak, aby pamięć przy każdym cyklu była odczytywana (nie sterujemy linią *m_read*) oraz łączymy linię danych magistrali z wewnętrzną linią danych diody – przy okazji pomijamy najstarsze 8 bitów. Robimy tak z powodu łatwości adresowania danych w taki sposób – dzięki temu dla każdej diody wszystkie 24 bity znajdują się w obrębie jednego 32-bitowego słowa i proces zarówno odczytu jak i zapisu danych do tak zorganizowanej pamięci jest łatwy (**listing 3**) – oczywiście tracimy 8 bitów danych na jedną diodę, jednak jest to poświęcenie na które jesteśmy gotowi.

Na koniec musimy zadbać o to, aby adres na magistrali *Master* był zatrzaskiwany na odpowiednim zboczu, czyli narastającym:

```
m_address <= std_logic_vector(to_unsigned(actLed,m_address'length));
```

Tak przygotowany plik dodajemy, analogicznie jak poprzednio, jako plik źródłowy nowego modułu w *Platform Designer*. Tym razem do zrobienia mamy nieco więcej, gdyż musimy skonfigurować

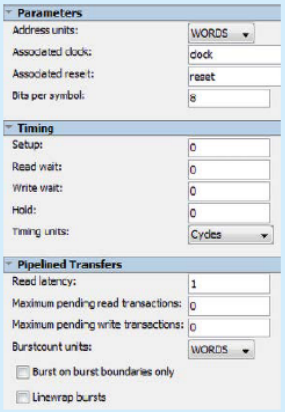


Rysunek 9. Uproszczony diagram stanów układu sterującego diodami

interfejs *Master*. W zakładce *Signals & Interfaces* dodajemy teraz dodatkowo interfejs *Avalon Memory Mapped Master* i do niego przeciągamy wszystkie sygnały z prefiksem *m_*. Następnie musimy ręcznie wybrać typ (*Signal Type*) dla każdego z nich – oczywiście jest to proste zadanie bo typy zgodne są z nazwami sygnałów. Następnie dla całego interfejsu wybieramy sygnał reset.

I teraz najważniejsze – musimy ustawić parametry tak, jak dla modułu pamięci RAM (możemy podejrzeć!). Ostatecznie parametry oraz lista w zakładce *Signals & Interfaces* powinna wyglądać tak, jak pokazano na **rysunkach 10** i **11** . Szczególnie ważne jest, aby zmienić *Address units* na *WORDS* – inaczej adresowanie będzie realizowane bajt-po-bajcie i nie będziemy mogli skorzystać z szybkiej możliwości odczytania danych dla danej diody z naszej pamięci RAM.

Rysunek 10. Lista sygnałów modułu WS2812_RAM



Rysunek 11. Ustawienia interfejsu avalon_master

Na początek dodajmy do naszego systemu 2-portową pamięć RAM, w tym celu w *IP Catalog* wyszukujemy *On-Chip Memory (RAM or ROM)* i dodajemy taki komponent. Potem ustawiamy ważne parametry (**rysunek 12**):

Zaznaczamy opcję *Dual-port access* oraz *Single clock operation*. Dzięki temu pamięć będzie miała 2 porty dostępowe pracujące przy tym samym zegarze – dokładnie to czego nam trzeba

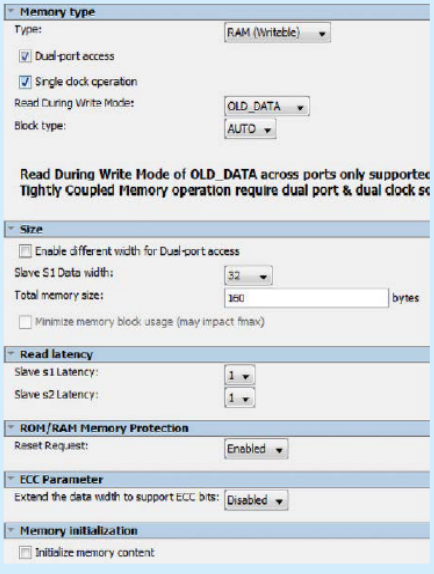
Wybieramy dodatkowo *Read During Write Mode: OLD_DATA*. Zagwarantuje to, że na wyjściu pamięci nie pojawią się żadne śmieci w chwili gdy procesor będzie prowadził zapis, a nasz moduł odczyt pod tym samym adresem

Podajemy rozmiar pamięci: 160 bajtów (dla 40 diodek) – pamiętajmy, że z powodu tego iż jeden bajt „tracimy" na każdej diodzie to de-facto potrzebujemy 4 bajtów na każdą diodę

Odznaczamy opcję *Initialize memory content*

Teraz podłączamy pamięć do systemu – zegar i reset łączymy w sposób wiadomy, port *s1* łączymy z *data_master* CPU, zaś port *s2* pozostawiamy na razie niepołączony.

Dodajemy wreszcie nowostworzony moduł *WS2812_RAM*, podając jako parametr liczbę diod wynoszącą 40.



Rysunek 12. Ustawienia pamięci RAM 2-portowej

Rysunek 13. Kluczowe połączenia nowego modułu i przeznaczonej dla niego 2-portowej pamięci RAM

	Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers	Memory Bits	M9Ks
12	|tutorial_ws2812_1:ws2812_1|	2091 (0)	1235 (0)	0	0
7	|tutorial_LED_RAM:led_ram|	1 (1)	0 (0)	1280	2
12	|WS2812_RAM:we2812_ram_0|	514 (514)	281 (281)	0	0

Rysunek 14. Porównanie wykorzystania zasobów przez oryginalny moduł sterujący, oraz przez moduł sterujący z zewnętrzną pamięcią

Zegar, reset i *avalon_slave_0* łączymy w oczywisty sposób z procesorem, *led_out* – eksportujemy pod taką samą nazwą jak w poprzedniej wersji modułu (wcześniej rzecz jasna usuwając stary, pożerający zasoby moduł; dzięki takiej samej nazwie nie będziemy musieli nic modyfikować w *Pin Planner*). Na koniec *avalon_master* łączymy z portem *s2* pamięci RAM.

Uruchamiamy *Assign Base Addresses* po czym jeden z adresów musimy zmodyfikować! Mianowicie adres bazowy dwuportowej pamięci RAM na porcie *s2* – ustawmy go na 0 – inaczej nasz moduł, który zbudowaliśmy tak, aby adresowanie przebiegało od zera, nie będzie w stanie dostać się do przeznaczonych dla niego danych. Ostatecznie połączenia oraz adresy powinny wyglądać jak na **rysunku 13**.

Teraz możemy wygenerować projekt, dokonać kompilacji i zanim zabierzemy się za *BSP* w *Eclipse* sprawdzić jaka jest teraz „objętość" modułu (**rysunek 14**).

Jak widzimy zmniejszyliśmy zajętość komórek logicznych prawie 4-ro krotnie! I wykorzystaliśmy leżący dotąd odłogiem RAM i komórki M9K. Teraz nie pozostaje już nic innego jak przetestować nasze rozwiązanie ze zmodyfikowanym oprogramowaniem – w zasadzie musimy tylko zmienić miejsce, gdzie zapisywane są dane o kolorze – zamiast do modułu będziemy je zapisywać do specjalnej pamięci RAM, którą utworzyliśmy, począwszy od zerowego adresu w tejże pamięci. Proste i optymalnie wykorzystujące miejsce w FPGA!

Oczywiście w razie konieczności trzeba by dalej przeprowadzać optymalizacje (do czego zachęcam) i np. widząc, że wartości, które są zapisywane w niektórych rejestrach ustalających czasy przebiegów są mniejsze od 255 – zmniejszyć szerokość tych rejestrów do 8 bitów.

Przerwania – od podszewki

Nasz moduł sterujący diodami działa już całkiem sprawnie. Jednak dociekliwsze osoby zauważą pewien problem – mianowicie skąd niby wiemy, że moduł daną ramkę już wysłał i zapalił diody zgodnie z naszym życzeniem? O ile jeszcze w wypadku niewielkiej ilości diodek to nie problem, o tyle gdybyśmy zdecydowali się na profesjonalne wykorzystanie takiego układu to czas wysłania danych do naszego łańcuszka mógłby mieć istotne znaczenie.

Dlatego też dodajmy do naszego modułu 2 modyfikacje. Po pierwsze wyjście ze stanu S_PRES będzie teraz aktywowane poprzez zapis jedynki do odpowiedniego rejestru – dzięki temu będziemy mieć kontrolę nad tym, kiedy wysyłanie danych zostanie rozpoczęte. Po tym moduł oczywiście sam zresetuje ten bit, aby nadawanie nie zostało włączone kolejny raz. Dodatkowo dodamy flagę, która będzie informowała o tym, czy wysyłanie danych zostało już zakończone, a także dodamy możliwość generowania w tym momencie przerwania.

O ile pierwsze z tych zadań wymaga w zasadzie tylko odrobiny wiedzy z VHDL, o tyle nad drugim z nich warto się nieco pochylić. Aby zasygnalizować przerwanie moduł *slave* musi posiadać linię wyjściową, na której będzie podawał stan wysoki dotąd, dokąd będzie chciał zgłaszać przerwanie. Stan tej linii może być zmieniany tylko na zboczu narastającym, o co tez musimy zadbać. Ponadto warto dodać rejestr, który umożliwi nam wyłączenie generowania tego przerwania.

Listing 4. Modyfikacja programu - użycie funkcji zapisu WS2812

```
volatile uint8_t WS2812Done = 0;
void WS2812Interrupt(void* context)
{
    //kasowanie flagi przerwania
    IOWR_32DIRECT(WS2812_INT_0_BASE, WS2812_STATUS_REG, 0);
    WS2812Done = 1;
}

void WS2812UpdateWaitInt(void)
{
    IOWR_32DIRECT(WS2812_INT_0_BASE, WS2812_CONFIG_REG, WS2812_CONFIG_INT | WS2812_CONFIG_START);
    while(WS2812Done != 1);
    WS2812Done = 0;
    ALT_USLEEP(5000);
}
```

Listing 5. Użycie rejestru przerwań (bez przerwań)

```
void WS2812UpdateWaitPoll(void)
{
    IOWR_32DIRECT(WS2812_INT_0_BASE, WS2812_CONFIG_REG, WS2812_CONFIG_START);
    while(IORD_32DIRECT(WS2812_INT_0_BASE, WS2812_STATUS_REG) != 1);
    IOWR_32DIRECT(WS2812_INT_0_BASE, WS2812_STATUS_REG, 0);
    ALT_USLEEP(5000);
}
```

Dodajemy nasz moduł na podstawie pliku WS2812_RAM_INT.vhd dokładnie tak jak poprzednio. Modyfikacje względem poprzedniej wersji są kosmetyczne: dodany zostanie nowy port wyjściowy, trzy proste 1-bitowe rejestry (pogrupowane pod dwoma adresami – jeden rejestr będzie obsługiwał samą sygnalizację przerwania i kasowanie przerwania, zaś drugi – rozpoczynanie wysyłania danych do diod oraz włączanie i wyłączanie przerwania), wraz z ich odczytem i zapisem (tu znów trochę marnuję adresy, ale i tak mamy do wykorzystania ich aż 8). Na zboczu narastającym zaś ustawiać linię wyjściową sygnalizacji przerwania, ale tylko wtedy, jeśli użytkownik ustawi odpowiedni bit na to zezwalający. Tu gorąco zachęcam do analizy tego (i wszystkich) kodów VHDL, a także do… optymalizacji! Korzystając z linii *byteenable* możemy wszystkie rejestry odpowiadające za czasy ograniczyć do 8-bitów zgrupować jako jeden 32-bitowy rejestr, następnie czas resetu zaimplementować jako rejestr 16-bitowy i do niego dołączyć rejestr konfiguracji (8-bit) oraz rejestr statusu (8-bit). Całą przestrzeń zatem powinniście dać radę zmniejszyć do… jednego bitu adresującego! Ale wróćmy do naszego, na razie nieoptymalnego, zadania.

W momencie dodawania interfejsów mamy dostępny dodatkowy sygnał – sygnał przerwania. Musimy teraz dodać nowy interfejs: *<<add interface>>* i wybieramy *Interrupt Sender*. Do tego interfejsu przeciągamy sygnał *irq*, a następnie wskazujemy jako *Signal Type* również *irq*. W ustawieniach interfejsu *Interrupt Sender* wybieramy jako *Associated addressable interface* nazwę naszego interfejsu *Avalon-MM Slave*, czyli jeśli nic nie zmienialiśmy będzie to: *avalon_slave_0*. resztę opcji pozostawiamy bez zmian. Po ukończeniu procesu usuwamy z naszego projektu stary blok sterownika diod i w jego miejsce wstawiamy i konfigurujemy identycznie naszą nową zabawkę. Dodatkowo teraz musimy podłączyć do procesora sygnał przerwania. Jednak zanim klikniecie *Assign Base Addresses* ulatwmy sobie nieco życie i spowodujemy aby adres pamięci ram na porcie *s2* (połączonym do naszego modułu) nie zmieniał się. W tym celu klikamy na małą kłódkę obok tego adresu – powinna ona zmienić kolor na czarny i tym samym zablokować automatycznym narzędziom możliwość zmiany tego parametru (**rysunek 15**).

Rysunek 15. Blokowanie adresów bazowych modułów

Po uzupełnieniu połączeń w zasadzie zostało nam tylko nieco zmodyfikować program, aby korzystał z nowych funkcji, które zaimplementowaliśmy (**listing 4**).

Kod chyba nie wymaga bardziej obszernego komentarza, może oprócz tego, że wykorzystujemy tu przerwanie w bardzo sztuczny sposób, stosujemy instrukcje opóźniające i nie robimy nic pożytecznego poza zmianą kolorów w bliżej nieokreślony sposób. No ale przecież chodzi głównie o demonstrację zastosowania przerwań!

Oczywiście w tak prostej sytuacji możemy też skorzystać z rejestru sygnalizującego przerwanie… zupełnie bez przerwania, np. jak pokazano na **listingu 5**.

Podsumowanie

W tej części wspólnie poznaliśmy kolejne ważne i przydatne elementy interfejsu *Avalon-MM*, czyli budowę własnych portów typu *master* oraz metodę wykonywania własnych modułów z obsługą przerwań. W ramach bonusu w materiałach znajdziecie już zoptymalizowany przeze mnie moduł, który teraz zajmuje jedynie nieco ponad 230 komórek logicznych, jednak postarajcie się nie zaglądać do pliku źródłowego i samemu „powalczyć" z wyzwaniem. W czasie kolejnego spotkania zajmiemy się obsługą kart pamięci SD oraz jedną ważną kwestią – metodami przenoszenia już przygotowanych modułów pomiędzy projektami. Powodzenia z zadaniami optymalizacyjnymi!

Piotr Rzeszut, AGH