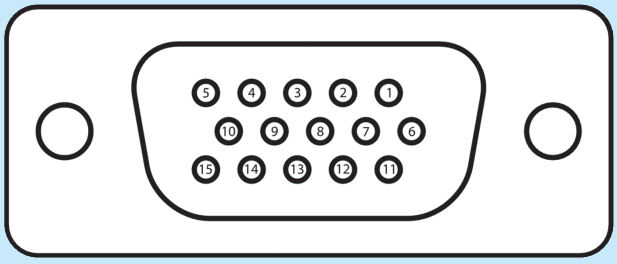
NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (13)

Czas na duuuuuże wyświetlacze!

W tej i kolejnej części zajmiemy się dwoma złączami na płytce maXimator, które z pewnością pobudzały Waszą wyobraźnię odkąd pierwszy raz otworzyliście pudełko – złączami VGA oraz HDMI. Dzięki nim możemy użyć praktycznie dowolnego ekranu komputerowego w roli wyświetlacza. W czasie naszego spotkania postaramy się rozpracować krok po kroku interfejs VGA, tak abyśmy na samym końcu mogli powiedzieć: „jakie to łatwe”!

Od czego zacząć? Najlepiej od… początku. Czyli w tym wypadku od… złącza VGA. Rozmieszczenie wyprowadzeń na tym złączu przedstawiono na **rysunku 1**, zaś w **tabeli 1** opisano funkcje poszczególnych wyprowadzeń. Sygnały ID0…ID3 używane były w starszych monitorach, w czasach kiedy wystarczało 16 różnych trybów pracy możliwych do zasygnalizowania (poprzez ustawienie odpowiedniego poziomu na tych pinach). Obecnie w monitorach znajduje się układ komunikujący się z komputerem za pomocą znanej nam już magistrali, dzięki czemu można przesłać znacznie więcej danych (teraz już wiecie skąd komputer „wie” jakiego producenta monitor podpięliście i jaka jest jego optymalna rozdzielczość). Niektórzy producenci umożliwiają sterowanie niektórymi funkcjami monitora za pomogą tego interfejsu.

Ale tymi funkcjami nie będziemy się zajmować. Nas zadowoli po prostu wyświetlenie obrazu o stałej rozdzielczości! Przyjrzyjmy się sprzętowemu połączeniu monitora VGA z układem FPGA, pokazanemu na **rysunku 2**. Jak widzimy, linie synchronizacji połączone są za pomocą rezystorów 75 Ω, monitor akceptuje na tych liniach napięcia do 5 V, a rezystory mają za zadanie zapewnienie odpowiedniej impedancji. Z kolei, linie RGB są połączone za pomocą rezystorów 270 Ω. Wartość ta nie jest przypadkowa – daje to wraz z rezystancją wejściową monitora (75 Ω) dzielnik napięcia, dzięki któremu po podaniu na wejście 3,3 V otrzymamy na wyjściu nieco powyżej 0,7 V, czyli poziomu maksymalnego nasycenia. Proste rozwiązanie, które pozwala na wykonanie połączenia z monitorem w na prawdę minimalistyczny sposób.



Rysunek 1. Układ wyprowadzeń złącza VGA od strony gniazda (żeńskiej). Źródło: Wikipedia, użytkownik: Mobius

Sposób przesyłania danych

Jak już możemy wywnioskować z lektury wcześniejszych akapitów, przesyłanie danych w interfejsie VGA odbywa się za pomocą 3 sygnałów analogowych, których napięcie odpowiada nasyceniu poszczególnych kolorów w danym pikselu. Nie mamy dostępnego sygnału zegarowego, więc dane kolejnych pikseli muszą być przesyłane w precyzyjnie określonych momentach czasu – do tego celu potrzebujemy taktowania o ściśle określonej częstotliwości – częstotliwości przesyłania pikseli.

Mimo braku sygnału zegarowego musimy zsynchronizować jakoś przesyłanie pikseli, aby monitor „wiedział”, który fragment obrazu aktualnie przesyłamy. Właśnie w tym celu są wykorzystywane linie HSYNC i VSYNC. Aby zrozumieć ich działanie, omówmy elementy budujące obraz na ekranie. Oczywiście obraz składa się z pikseli,

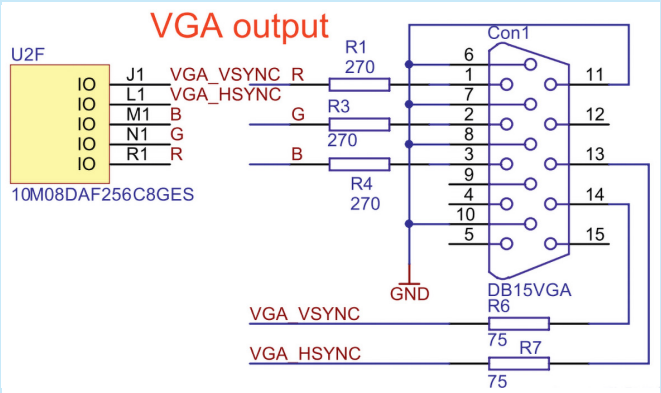
Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

Tabela 1. Funkcje wyprowadzeń złącza VGA

Pin	Nazwa	Pełniona funkcja
1	RED	Nasycenie koloru czerwonego (0-0,7 V, impedancja wejściowa 75 Ω)
2	GREEN	Nasycenie koloru zielonego (0-0,7V , impedancja wejściowa 75 Ω)
3	BLUE	Nasycenie koloru niebieskiego (0-0,7 V, impedancja wejściowa 75 Ω)
4	ID2/RES	Obecnie nieużywany
5	GND	Masa
6	GND-RED	Masa dla kanału czerwonego
7	GND-GREEN	Masa dla kanału zielonego
8	GND-BLUE	Masa dla kanału niebieskiego
9	KEY/PWR	Brak pinu/zasilanie 5 V
10	GND-Sync	Masa dla kanałów synchronizacji
11	ID0/RES	Obecnie nieużywany
12	ID1/SDA	Linia SDA magistrali I²C służąca do identyfikacji monitora
13	HSync	Sygnał synchronizacji poziomej (aktywny stanem niskim)
14	VSync	Sygnał synchronizacji pionowej (aktywny stanem niskim)
15	ID3/SCL	Linia SCL magistrali I²C służąca do identyfikacji monitora

a te z kolei są ułożone w linie, które tworzą wreszcie cały obraz. Sekwencja wyświetlania linii (**rysunek 3a**) składa się z następujących elementów:

1. SYNC – linia synchronizacji poziomej (HSYNC) jest ustawiana w stan niski, w tym czasie napięcie na liniach RGB musi wynosić 0 (kolor czarny).
2. BACK PORCH – czas martwy, w tym momencie nie wyświetlamy żadnych pikseli.



Rysunek 2. Schemat elektryczny interfejsu VGA w zestawie maXimator

3. DISPLAY – czas aktywny, w tym momencie zmieniamy napięcia na liniach RGB aby odwzorować kolory kolejnych pikseli w tej lini na ekranie.
4. FRONT PORCH – czas martwy, nie wyświetlamy nic.

Wszystkie czasy w linii definiowane są w ilości pikseli (czyli ilości cykli zegara taktującego nadawanie kolejnych pikseli).

Z tak wygenerowanych linii musimy zbudować cały obraz. Tutaj występują dokładnie takie same elementy jak w wypadku pojedynczej linii, z tą różnica że sterujemy sygnałem VSYNC (synchronizacja pionowa), a naszą „jednostką czasu" jest teraz cała linia obrazu (**rysunek 3b**).

Ważne, aby nie zapominać o fakcie, iż w czasie jakichkolwiek impulsów synchronizacyjnych musimy „nadawać kolor czarny". Czy to nie łatwe? Mam nadzieję, że tak i możemy zacząć… projektować nasz moduł sterowania VGA.

Generator znaków – czyli co zbudujemy

Teraz musimy odpowiedzieć sobie na pytanie – co chcemy tak naprawdę zbudować? Moja propozycja to sprzętowy generator znaków, czyli moduł umożliwiający wyświetlenie na ekranie znaków określanych w kodzie ASCII.

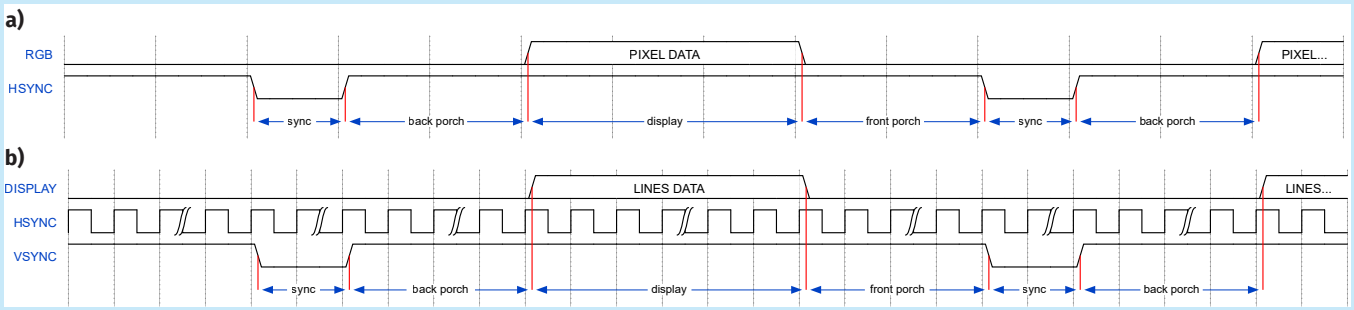
Dlaczego jednak nie wyświetlanie grafik? Odpowiedź jest prosta i ma związek z niewystarczającą ilością pamięci RAM dostępnej w naszym układzie FPGA (oraz brakiem osobnego układu pamięci). Oczywiście można wyświetlać pewne grafiki generując je „w locie" na podstawie danych, jednak także uznałem, że moduł wyświetlacza tekstowego znajdzie szersze i bardziej ogólne zastosowanie. W kwesti pamięci wybierając generowanie znaków możemy zastosować pamięć ROM przechowującą wzory znaków i „w locie" je odczytywać na podstawie pamięci przechowującej znaki do wyświetlenia w kodzie ASCII (czyli 1 bajt na 1 literę). A więc do dzieła!

Generator sygnałów sterujących

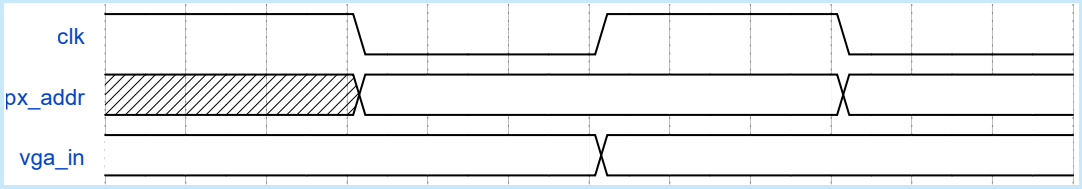
Budowę naszego systemu rozpocznijmy od najważniejszego elementu układanki, czyli modułu generującego sygnały sterujące oraz pamiętającego aktualnie wyświetlany na ekranie piksel (jego współrzędne) aby reszta naszego układu miała jakieś podstawy do przekazania danych do wyświetlania.

Nasz moduł będzie miał następujące połączenia:

```
port(
  vga_r           : out std_logic;
  vga_g           : out std_logic;
  vga_b           : out std_logic;
  vga_in          : in std_logic_vector(2 downto 0);
  vga_hs_n        : out std_logic;
  vga_vs_n        : out std_logic;
  px_addr_col     : out std_logic_vector(12 downto 0);
  px_addr_row     : out std_logic_vector(12 downto 0);
  clk             : in std_logic;
  rst_n           : in std_logic
);
```



Rysunek 3. Transmisja obrazu: a) linia obrazu przesyłana za pomocą interfejsu VGA, b) jedna ramka obrazu przesyłana za pomocą interfejsu VGA



Rysunek 4. Prawidłowe przekazywanie danych do modułu VGA

Pierwsze trzy z nich to wyjścia sygnałów VGA, kolejne to 3-bitowe wejście, którym do modułu podawać będziemy dane (kolor) kolejnego piksela, następnie zapewniamy 2 kanały synchronizacji (poziomej i pionowej). Wreszcie zapewniamy dwa wyjścia, na których nasz moduł będzie podawać adres piksela, którego dane powinniśmy podać na stosowne wejście w ciągu aktualnego cyklu zegarowego. Oczywiście całości dopełnia sygnał zegarowy (o częstotliwości równej częstotliwości nadawania pikseli) oraz sygnał resetu.

Sam moduł w zasadzie będzie szalenie prosty i zachęcam Was do analizy kodu źródłowego, w którym w zasadzie znajdziecie jedynie zliczanie pikseli, linii oraz instrukcje warunkowe włączające w odpowiednich momentach sygnały synchronizacji. Aby dane piksela podawać w odpowiednim momencie spójrzmy jeszcze na przebiegi jakie powinny być obserwowane na stosownych wyprowadzeniach w trakcie cyklu zegarowego (**rysunek 4**). Jak widzimy na zboczu opadającym zegara moduł wystawi nowy adres piksela, a my na najbliższym zboczu narastającym zegara powinniśmy ustawić na liniach *vga_in* dane odpowiadające temu pikselowi. Plik w którym opisałem w języku VHDL ten moduł to *DisplayVGA/vga_driver.vhd*. Oczywiście plik ten dodajemy w standardowy sposób do naszego projektu, a potem użyjemy go jako element modułu do systemu *Platform Designer*.

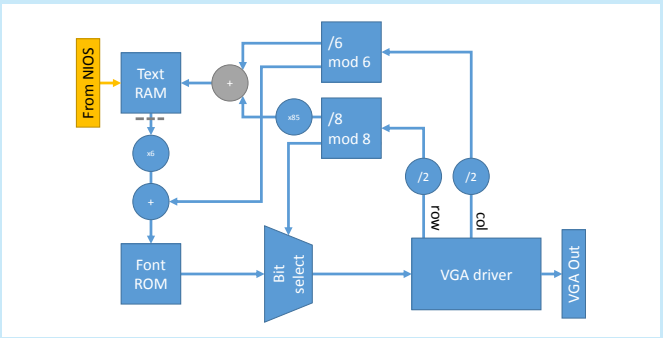
Dobry plan to podstawa!

Aby skutecznie poradzić sobie z resztą zadania, jakim jest wyświetlanie tekstu na ekranie z interfejsem VGA, musimy mieć dobry plan i przyjąć na tym etapie pewne założenia.

Jeśli chodzi o założenia to przyjmijmy następujące:

1. Rozdzielczość generowanego obrazu: 1024×768 (przy 60 Hz odświeżania da to nam częstotliwość zadawania pikseli równą 65 MHz – całkiem dobra i możliwa do uzyskania).
2. Rozmiar „naszego” piksela: 2×2 (po prostu sztucznie zmniejszymy rozdzielczość, dzięki czemu nasz tekst będzie nieco większy – jednocześnie podana wyżej rozdzielczość jest obsługiwana przez większość monitorów, a mniejsze rozdzielczości już nie zawsze – stąd rozwiążemy tym 2 potencjalne problemy).
3. Rozmiar znaku: 6×8 (podobnie jak w popularnym HD44780, dzięki temu sumarycznie otrzymamy matrycę mogącą pomieścić 48 wierszy po 85 znaków każdy).
4. Pamięć RAM przechowująca tekst w formie kodu ASCII oraz pamięć ROM przechowująca wzory znaków.

Teraz, gdy już mamy zapisany pomysł na papierze, czas na analizę jak go zrealizować technicznie. Na **rysunku 5** przedstawiam uproszczony schemat blokowy rozwiązania, które zaproponowałem i które zaraz omówię.



Rysunek 5. Uproszczony schemat blokowy prezentowanego rozwiązania generatora znaków z obsługą wyjścia VGA

Omawianie tegoż rozwiązania zacznijmy od… końca tym razem, czyli od wyjścia VGA. Jest ono sterowane przez driver VGA, który już przygotowaliśmy. Udostępnia on nam adres wiersza i kolumny, w którym znajduje się piksel, który wyświetlany będzie w kolejnym cyklu, oraz oczekuje podania na wejście koloru tegoż piksela.

Numer wiersza oraz kolumny dzielimy na początku przez 2, aby otrzymać przeskalowanie rozmiaru „naszego” piksela. Dzięki temu 4 sąsiednie piksele będą zawsze miały tę samą barwę. Następnie dokonujemy dzielenia z resztą tych wartości odpowiednio przez 8 i 6. Iloraz z tych działań stanowi współrzędne znaku, zaś reszty z tych działań stanowią współrzędne piksela we wzorze danego znaku.

Na podstawie współrzędnych znaku, którego piksele aktualnie są wyświetlane, ustalamy adres w pamięci RAM przechowującej tekst do wyświetlenia poprzez sumowanie numeru kolumny oraz numeru wiersza pomnożonego przez 85, czyli liczba znaków w jednym wierszu.

Na wyjściu tak zaadresowanej pamięci RAM dostępny będzie zatem kod ASCII znaku, który mamy wyświetlić. Wzory znaków (przypomnijmy sobie: o rozmiarach 6×8 pikseli) są przechowywane w pamięci ROM w taki sposób, że każde 6 kolejnych 8 bitowych słów stanowi jeden znak. Kolejne słowa stanowią kolejne kolumny naszego znaku, zaś kolejne bity w danym słowie oznaczać będą kolejne wiesze pikseli.

Aby prawidłowo zaadresować tę pamięć na jej wejście podamy kod ASCII znaku pomnożony 6-cio krotnie (bo 6 bajtów zajmuje wzór jednego znaku) z dodaną resztą z dzielenia kolumny (patrz wyżej). Dane wyjściowe podamy na układ, który w zależności od danego wiersza w znaku (reszta z dzielenia numeru linii obrazu) wybierze odpowiedni bit z tegoż słowa. Ten właśnie bit zostanie podany na wejście drivera VGA. I tu nasza pętla się zamyka.

Musimy jednak zwrócić tu uwagę na ważną kwestię, a mianowicie opóźnienia w układzie oraz to na jakich zboczach pamięci zatrzaskują dane. I tak, wiemy, że driver VGA ustawia adresy na zboczu opadającym, zakładając, że układy kombinacyjne poradzą sobie czasowo, to na kolejnym zboczu narastającym pamięć RAM tekstu na wyjściu poda nam kod znaku. Tu pierwsza lampka ostrzegawcza – w wypadku bezpośredniego połączenia z pamięcią ROM zmieniamy adres na wejściu pamięci na zboczu na którym pamięć tenże zatrzaskuje – narażamy się na wyścig (ang. *race*) i problemy. Zakładając jednak, że z tym poradzimy sobie, to pamięć ROM znaków poda nam prawidłowe słowo na wyjściu dopiero przy kolejnym zboczu narastającym… Mamy więc tutaj opóźnienie o 1 cykl zegara względem tego, czego oczekuje moduł VGA…

Pierwszy problem (zagrożenie wyścigiem) rozwiążemy dodając (w miejscu oznaczonym szarą przerywaną linią) rejestr zatrzaskujący dane na zboczu opadającym. Drugi problem, z opóźnieniem o jeden cykl zegarowy, rozwiążemy zwiększając w określonej sytuacji wynik dodawania (oznaczonego na szaro) o 1. Zauważając, że wartość tej sumy zmienia się co 6 kolumn zwiększenie sumy o 1 musimy wykonać tylko dla najwyższej wartości kolumny, tak aby zatrzaśnięty kod znaku na wyjściu pamięci korespondował potem (bez opóźnienia) z adresem piksela we wzorze znaku. Może to wydawać się nieco zagmatwane w tym momencie, ale jeśli przeanalizujecie na spokojnie – takt po takcie zegara – działanie tego układu to z pewnością zrozumiecie celowość takiego rozwiązania.

Czas wcielić w życie nasz plan

Teraz już po tym długim, ale potrzebnym wstępie możemy przystąpić do realizacji naszego projektu. Pytanie na jakie musimy sobie odpowiedzieć to sposób realizacji operacji matematycznych. Wydawać by się mogło, że najprościej zapisać je po prostu w formie odpowiednich wyrażeń. Niestety w praktyce takie podejście doprowadza do nieoptymalnego wykorzystania zasobów i pogorszenia czasów propagacji, które w tak szybkim rozwiązaniu mają znaczenie. Dlatego też do realizacji większości funkcji matematycznych posłużymy się przygotowanymi przez producenta blokami funkcjonalnymi.

Na sam początek ustawmy nasz projekt tak, abyśmy mogli zainicjalizować zawartość pamięci ROM. W tym celu przechodzimy do: *Assignments* → *Device* …, następnie *Device and Pin Options*… i w zakładce *Configuration* zmieniamy tryb konfiguracji na *Single Uncompressed Image with Memory Initialization*.

Następnie utworzymy pamięci. Rzecz jasna pamięć ROM będzie musiała zostać zainicjalizowana. Do tego celu na początek stworzymy pusty plik inicjalizacyjny: *File* → *New* → *Memory Initialization File*. Podajemy rozmiar jako 768 słów 8 bitowych. Następnie dokonujemy jakiejś zmiany w pliku i zapisujemy go w folderze *DisplayVGA* jako *FONT.mif*.

W oknie *Quartus* w zakładce *IP Catalog* (nie w *Platform Designer*!!!, jeśli nie jest widoczny *View* → *Utility Windows* → *IP Catalog*) wyszukujemy ROM i następnie wybieramy pamięć ROM 1-portową. Jako język syntezy tradycyjnie tu (i praktycznie zawsze) wybieramy VHDL oraz wskazujemy lokalizacje pliku wewnątrz folderu *DisplayVGA* i nadajemy nazwę *fontROM*. Następnie określamy rozmiar pamięci jako 768 słów 8-bitowych, oraz w kolejnej zakładce odznaczamy opcję *Which ports should be registred?: 'q' optput port*. W zakładce *Mem Init* wskazujemy przygotowanych przed chwilą plik *.mif* (musimy w oknie *Browse*… ręcznie zmienić typ poszukiwanego pliku na pożądany format). Następnie klikamy do skutku *Finish*, a potem odpowiadamy twierdząco na pytanie o dodanie wygenerowanych plików do projektu.

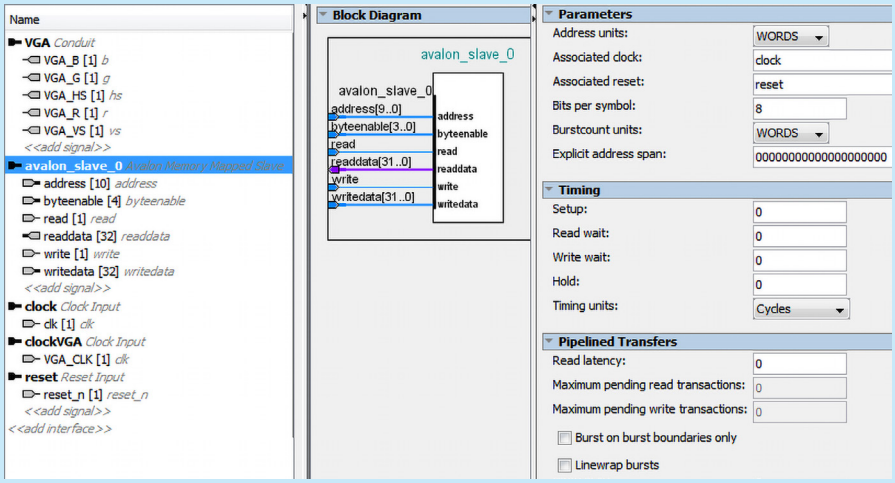
Kolejno wyszukujemy w ten sam sposób 2 portową pamięć RAM, plik z jej implementacją nazywamy *charRAM.vhd* i umieszczamy w identycznej lokalizacji jak poprzednio. W pierwszym oknie nic nie zmieniamy, w drugim ustawiamy rozmiar na 4080 słów 7-bitowych. W kolejnej zakładce wybieramy pierwsza z opcji *Dual clock* tworząca osobne zegary dla zapisu i odczytu. W czwartej zakładce odznaczamy opcję objęcia rejestrem *Read output port(s)*. Następnie klikamy *Finish/Yes*.

Następnie potrzebujemy dwóch dedykowanych układów mnożących, które pomnożą odpowiednie wartości 6-cio i 85-cio krotnie. W tym celu wyszukujemy *LPM_MULT* w *IP Catalog*. Pliki nazwijmy odpowiednio *multiplier6* / *85* i zapiszmy w takiej lokacji jak poprzednio.

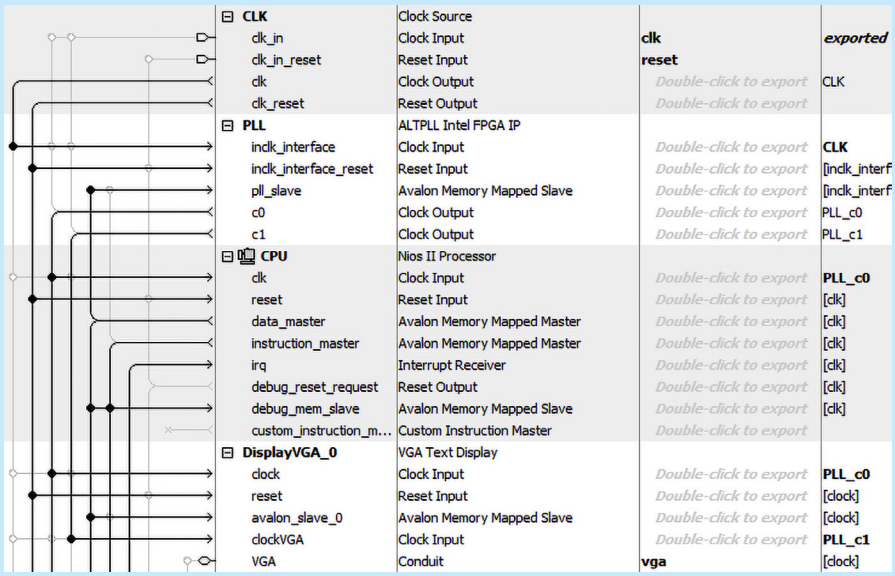
Dla mnożarki x6 ustawiamy szerokości bitów na 7 i 3 (odpowiednio dla portu *dataa* i …*b*), zaś dla x85 ustawiamy je odpowiednio na 9 i 8. W zakładce *General2* ustawiamy na stałe wartość *datab* na właściwą wartość. Po tym w obu przypadkach, tak jak poprzednio, kończymy generowanie modułów.

Brakuje nam jeszcze układu wykonującego dzielenie z resztą. W tym celu wyszukujemy komponent *LPM_DIVIDE*. W jego ustawieniach jedynie zmieniamy szerokości dzielnej (*numerator*) i dzielnika (*denominator*) na odpowiednio 12 i 3. Dzielnik ten wykorzystamy do dzielenia z resztą przez 6, zaś dzielenie z resztą przez 8 (jako potęgę dwójki) zastąpimy zwyczajnie odpowiednim połączeniem sygnałów z odpowiednim przesunięciem bitowym.

Teraz, gdy mamy już wszystkie klocki systemu (a pozostałe, prostsze, realizujemy już bezpośrednio w języku VHDL, jak np. dodawanie), czas napisać główny plik. Ten także znajdziecie w plikach dołączonych do tej części kursu, a w nim sporo komentarzy. Gdy już zapiszemy tenże plik (u mnie nazwa to *DisplayVGA.vhd*) czas uruchomić *Platform Designer* i rozpocząć dodawanie nowego komponentu (**rysunek 6**). Po podaniu nazwy przechodzimy do okienka wskazywania plików i tam dodajemy wszystkie pliki *.vhd* z folderu *DisplayVGA*. Upewniamy się, że plik *DisplayVGA.vhd* jest oznaczony jako *top-level*. Za pomocą odpowiednich przycisków kopiujemy ten zestaw plików do dwóch poniższych list symulacyjnych. Następnie rozpoczynamy



Rysunek 6. Ustawienia portów modułu wyświetlacza VGA



Rysunek 7. Prawidłowo wykonane połączenia modułu VGA

Edycja parametrów modułów *IP Core* dodanych poza *Platform Designer*.
Aby edytować moduły dodane do systemu poza *Platform Designer* musimy w *Project Navigator* wybrać widok *IP Components*. Tam znajdziemy listę wszystkich modułów użytych w projekcie i klikając na każdym z nich dwukrotnie możemy przejść do edycji parametrów.

analizę plików, a po jej zakończeniu ustawiamy odpowiednio sygnały i parametry interfejsu *Avalon*. Wszystkie prawidłowe ustawienia przedstawiam na **rysunku 6**. Parametry odczytu ustawiam na 0, gdyż i tak w naszym module nie będziemy mieć możliwości odczytu danych (pamięć 2-portowa jaką zastosowaliśmy ma 1 port do zapisu a drugi do odczytu).

vga_b	Output	PIN_M1
vga_g	Output	PIN_N1
vga_hs	Output	PIN_L1
vga_r	Output	PIN_R1
vga_vs	Output	PIN_J1

Rysunek 8. Prawidłowo przypisane wyprowadzenia dla interfejsu VGA

Następnie moduł zapisujemy i dodajemy do naszego systemu. Zanim zaczniemy go podłączać musimy wejść w ustawienia pętli PLL i w zakładce *Output Clocks/clk c1* włączamy ten zegar i nadajemy mu częstotliwość 65 MHz – to będzie nasz zegar synchronizujący nadawanie pikseli. Następnie w tradycyjny sposób łączymy wszystkie linie, z tym, że zegar *clock* łączymy z wyjściem *c0* pętli PLL, czyli 50 MHz, zaś *clockVGA* z wyjściem *c1*, czyli 65 MHz (**rysunek 7**). Na koniec eksportujemy wyjście jako *vga*, zapisujemy i generujemy system oraz przeprowadzamy *Analysis & Synthesis*. Następnie w *Pin Planner* ustawiamy standard wszystkich nowych wyprowadzeń na *3.3-V LVTTL* i ustawiamy odpowiednie wyprowadzenia (**rysunek 8**). Kompilujemy projekt i tradycyjnie tworzymy potem projekt oprogramowania w *Eclipse*, ale…

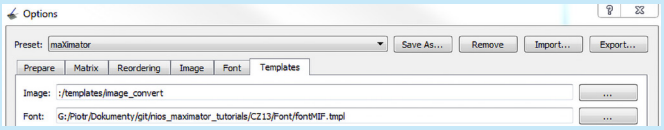
...jeszcze czcionka!

Jak pamiętacie pewnie w pewnym momencie stworzyliśmy pusty plik, w którym mieliśmy zdefiniować czcionkę. W plikach do tej części kursu znajdziecie gotowy plik z czcionką przygotowaną na podstawie czcionki znanej nam z wyświetlaczy HD44780. Jeśli jednak chcielibyście zdefiniować własną czcionkę, czy do tej dodać np. nasze „ogonki i kreseczki" to jestem niejako zobligowany opowiedzieć jak wygenerowałem plik *FONT.mif* z tajemniczymi cyframi, które potem staną się literkami. Posłużyłem się do tego programem *LCD Image Converter*, który jest darmowym oprogramowaniem. O samym tym programie i jego możliwościach można by napisać spory artykuł, więc tu opowiem o nim w telegraficznym skrócie.

Aby skorzystać z przygotowanej przeze mnie konfiguracji na początku po otworzeniu programu wchodzimy w *Options* → *Conversion…* i tam za pomocą menu *Import…* importujemy dostarczony (w folderze *Font*) plik *maXimatorCFG.xml*. Zawiera on wszelkie ustawienia konieczne do prawidłowego wygenerowania czcionki, przede wszystkim dotyczących kolejności bajtów i bitów we wzorze danego znaku, a także sposób zapisu liczb. Następnie z listy rozwijanej Preset wybieramy *maXimator*. Później w zakładce *Templates* w polu *Font* wskazujemy także dostarczony plik *fontMIF.tmpl*. Plik ten zawiera wzorzec pliku wymaganego przez środowisko *Quartus* do zainicjalizowania pamięci (**rysunek 9**). Teraz możemy otworzyć naszą czcionkę: *File* → *Open* i wskazujemy plik *maXimator.xml* (**rysunek 10**).

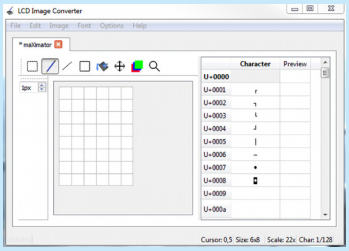
Na liście po prawej stronie wybieramy znak, jaki chcemy edytować, zaś po lewej stronie widzimy podgląd wszystkich pikseli znaku i dokonujemy jego edycji. Korzystając z narzędzia wybranego na powyższym obrazku za pomocą lewego przycisku myszy „zapalamy" piksel, a za pomocą prawego – kasujemy. Po modyfikacji / zaprojektowaniu znaków z menu *File* wybieramy *Convert* i zapisujemy plik jako *FONT.mif* – musimy ręcznie w okienku usnąć dopisane przez program *.h*, gdyż program został pierwotnie przemyślany pod kątem tworzenia plików z czcionkami dla oprogramowania przygotowywanego w językach C, C++ lub pokrewnych… Ciekawe czy Autor (któremu składam serdeczne podziękowania za wyjątkowo udane oprogramowanie) przypuszczał, że jego program będzie miał kontakt z układami FPGA!

Teraz wystarczy podmienić wcześniej przygotowany pusty plik z czcionką na właśnie wygenerowany. No i teraz możemy wykonać ostateczną syntezę naszego projektu i zaprogramować układ FPGA.



Rysunek 9. Okno konfiguracji programu LCD Image Converter z prawidłowo ustawionym wzorcem pliku wyjściowego z czcionką

Szczypta oprogramowania – wersja demo



Rysunek 10. Widok okna głównego programu LCD Image Converter z wczytaną czcionką maXimator

Jeśli przypuszczacie, że nasze oprogramowanie będzie ekstremalnie proste… to macie rację! Znów 99% pracy wykonuje za nas dedykowana logika w układzie FPGA. W standardowy sposób tworzymy projekt oprogramowania oraz *BSP* i możemy podziwiać prostotę obsługi naszego ekranu:

```
void VGAPutChar(uint32_t base, uint8_t col, uint8_t
row, char c)
{
    uint32_t position = row * 85 + col;
    if(position > 0xFEF) return;
    IOWR_8DIRECT(base, position, c);
}

void VGAPutCharX(uint32_t base, uint32_t position,
char c){
    if(position > 0xFEF) return;
    IOWR_8DIRECT(base, position, c);
}

void VGAPutString(uint32_t base, uint8_t col, uint8_t
row, char* s){
    uint32_t position = row * 85 + col;
    do {
        VGAPutCharX(base, position, *s);
        position++;
    } while(*(s++) != 0);
}
```

Czyli w zasadzie zapis do pamięci RAM naszego kontrolera! Na tym etapie chyba pozostaje już zostawić każdemu możliwość samodzielnej zabawy z systemem. Rzecz jasna wyświetlać możemy tez w pewien sposób obrazki (ASCII ART), lub zdefiniować własne znaki w czcionce, które mogą posłużyć tworzeniu nieco bardziej skomplikowanych struktur.

Podsumowanie

W czasie naszego spotkania, z drobną pomocą, zbudowaliśmy działający układ generujący obraz przesyłany w standardzie VGA, a także poznaliśmy pewne sztuczki, które pomogły na dokonanie tego z bardzo ograniczoną ilością pamięci RAM.

Czy coś jest jeszcze do zrobienia? Ależ oczywiście! Choćby można nasz moduł rozszerzyć o możliwość definiowania koloru znaku oraz koloru tła (możemy to robić dla każdego znaku, albo większymi blokami, jak np. wierszem czy jego fragmentem). Względnie pamięć ROM możemy zastąpić także 2-portową pamięcią RAM umożliwiając modyfikację wzorów znaków z poziomu oprogramowania. Ograniczają nas tylko dostępne zasoby!

A już następnym razem zajmiemy się osiągnięciem identycznego efektu, ale z wykorzystaniem interfejsu HDMI – stay tuned!

Piotr Rzeszut, AGH