Akademia Górniczo – Hutnicza
im. Stanisława Staszica w Krakowie

Wydział IET

Katedra Elektroniki

AGH

Technika Cyfrowa

Wprowadzenie do laboratorium komputerowego część I

1. Wstęp

Celem wprowadzenia do laboratorium komputerowego jest zapoznanie z podstawami języka opisu sprzętu VHDL oraz środowiskiem programistycznym Vivado. W ramach pierwszej części ćwiczenia studenci zaprojektują układ pełniący funkcję *dokładnie jeden z trzech* wykonując czynności:

- utworzenie nowego projektu w środowisku Vivado
- utworzenie kodu źródłowego w języku VHDL
- symulacja poprawności działania zaprojektowanego układu
- synteza i place&route kodu
- generacja pliku *bitstreamu* – programującego układ FPGA
- zaprogramowanie dedykowanej płytki ewaluacyjnej Nexys4

W ramach drugiej części ćwiczenia studenci zaprojektują układ pełniący funkcję dodawania i odejmowania 8-bitowych liczb binarnych bez znaku oraz w kodzie uzupełnień do dwóch. Ćwiczenie pozwoli na zaznajomienie się z arytmetyką w binarnym systemie liczbowym oraz z korzystaniem z gotowego modułu.

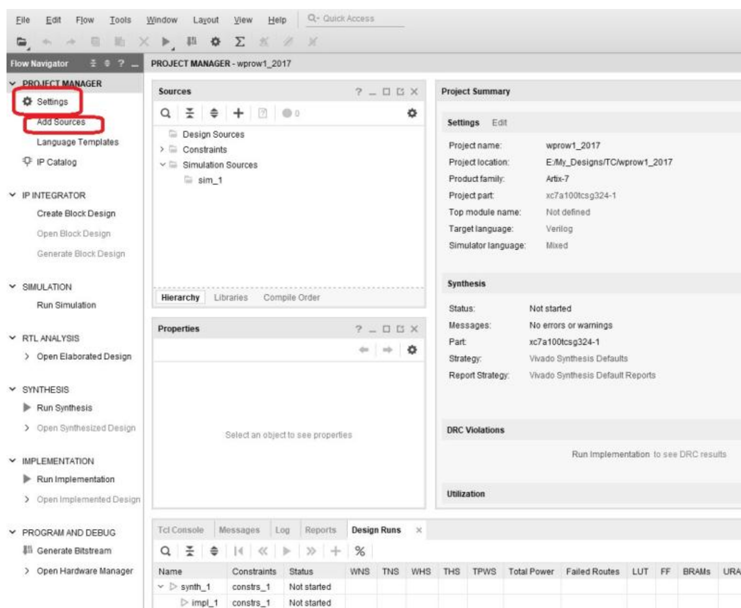
Warto podkreślić, że program Vivado w ograniczonej wersji można zainstalować na swoich komputerach w domu. W tym celu należy wejść na stronę producenta układów FPGA - firmę Xilinx: http://www.xilinx.com/support/download.html i wybrać odpowiednią wersję (2017.4 taka jest na laboratorium) oraz darmową licencję (WebPack). Aby ściągnąć plik należy wypełnić formularz rejestracyjny. Pełna instalacja Vivado wymaga ściągnięcia około 15 GB pliku. Podczas tego przedmiotu korzystamy z ograniczonych funkcji Vivado: 1) Synthesis and Place and Route, 2) Vivado Simulator oraz 3) opcjonalnie (jeśli posiadasz własną płytkę z układem FPGA) Vivado Device Programmer. Uwaga: Vivado wspiera relatywnie nowe układy FPGA (rodzinę 7 i nowsze), do programowania starszych układów FPGA należy użyć starszego programu ISE. Na laboratorium wykorzystywany jest układ Arctic7 XC7A100T-1CSG324C i płytka Digilent Nexys4.

2. Utworzenie nowego projektu w środowisku Vivado

1. Proszę uruchomić program Vivado ikoną znajdującą się na pulpicie lub poprzez Menu Start (zakładka Xilinx Design Tools).
2. W otwartym oknie klikamy opcję *Create New Project*.
3. Otworzy się okno *czarodzieja* nowego projektu, po zapoznaniu się klikamy *Next*.

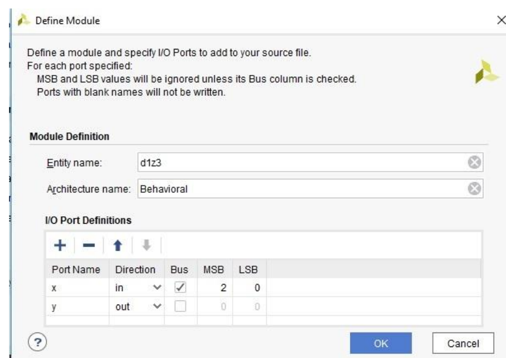
4. W następnym oknie wpisujemy nazwę projektu (np. d1z3) oraz folder w którym ma on być zapisany. Dla zachowania porządku wskazane jest, aby każdy zespół utworzył dla siebie folder w którym będzie pracować oraz aby to był podkatalog katalogu C:/My_Designs. Dobrym zwyczajem jest nieumieszczanie spacji w nazwach plików i folderów. Pozostawiamy zaznaczoną opcję *Create project subdirectory*. Klikamy *Next*.
5. W oknie zatytułowanym *Project type* wybieramy *RTL Project*. Pozostawiamy odznaczone pole *Do not specify sources at this time*. Klikamy *Next*.
6. W oknie *Default part* wybieramy zakładkę *Boards* a następnie płytkę *Nexys4*. Klikamy *Next*.
7. W kolejnym oknie pojawi się krótkie podsumowanie nowego projektu. Po zapoznaniu się klikamy *Finish*.

W konsekwencji mamy otwarty nowy pusty projekt Vivado co pokazuje poniższy obrazek:



1) W oknie *Project Manager* (po lewej stronie) klikamy *Settings* i zmieniamy *Target language* na *VHDL* i klikamy OK.

2) Następnie stworzymy nowy plik źródłowy (VHDL – plik d1z3.vhd) w naszym projekcie. Plik ten będzie zawierał funkcjonalność naszego układu. Aby stworzyć plik: klikamy *Add Sources* (zob. powyższy rysunek) w oknie *Project Managera* co spowoduje, że otworzy się nam okno, w którym wybieramy *Add or create design sources* i klikamy *Next*. Następnie klikany *Create file*, i wpisujemy nazwę pliku *d1z3* oraz klikamy OK. Następnie wybieramy *Finish*. Pojawia nam się okno, w którym dodajemy porty (klikając symbol +) zgodnie z poniższym rysunkiem: dodajemy port wejściowy *x*(2 *downto* 0) jako magistralę (bus) oraz port wyjściowy *y*. Klikamy przycisk OK.



Warto zauważyć, że ten sam efekt w dodawaniu portów możemy osiągnąć edytując ręcznie plik tekstowy d1z3.vhd:

```
entity d1z3 is
   Port ( x : in STD_LOGIC_VECTOR (2 downto 0);
         y : out STD_LOGIC);
end d1z3;
```

3) Następnie dodamy plik z rozszerzeniem *.xdc*. Jest on plikiem konfiguracyjnym o szerokim zastosowaniu, my będziemy używać go do przypisania portów na płytce ewaluacyjnej. Plik *Nexys4_Master.xdc* znajduje się w katalogu c:/TC/tutorial1. Jeśli nie ma takiego katalogu to proszę go utworzyć oraz ściągnąć plik *materiały* znajdujący się obok niniejszej instrukcji (ze strony http://www.fpga.agh.edu.pl/tc) i rozpakować go do tego katalogu. Plik *.xdc* z zawartością dla płytki Nexys4 można również pobrać ze producenta płytki.
   Plik dodajemy do projektu klikając *Add files / Add or Create Constraines / Add Files* i wybierając plik *Nexys4_Master.xdc*.

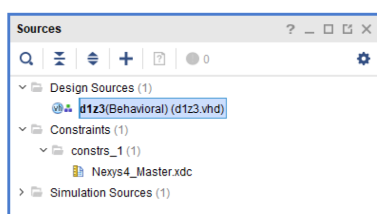
3. Kod źródłowy w języku VHDL

x_0	x_1	x_2	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Funkcja *dokładnie 1 z N* zwraca wartość '1' wtedy, i tylko wtedy gdy dokładnie jeden bit argumentu *X* posiada wartość '1'. Układ zaprogramowany tą funkcją jest układem kombinacyjnym, ponieważ stan jego wyjścia nie zależy od poprzedniego stanu wyjścia, a jedynie od aktualnego stanu wejść. Projektujemy funkcję *dokładnie jeden z trzech*, więc nasz układ będzie posiadał 3 wejścia (nazwijmy je x_0, x_1, x_2) w kodzie VHDL to wektor 3-bitowy: x(2 downto 0) oraz jedno wyjście (nazwijmy je *y* – jeden bit). Tablica prawdy dla naszej funkcji przedstawia się następująco:

Edycja pliku

Proszę kliknąć dwukrotnie na nazwie pliku *d1z3.vhd* znajdujący się w *folderze Design Sources* drzewa naszego projektu (co pokazuje poniższy rysunek).



Z prawej strony zostanie otwarta nowa zakładka, a w niej zawartość pliku. Program Vivado wygenerował szkielet naszego kodu:

```
library IEEE;  -- deklarację bibliotek – biblioteki deklarujemy zawsze na początku kodu.
use IEEE.STD_LOGIC_1164.ALL; -- Biblioteka std_logic_1164 jest podstawową biblioteką, która umożliwia
                             -- korzystanie z logiki std_logic, która oprócz stanów '0' i '1' posiada dodatkowe
                             -- stany, np. 'X' – nieokreślony, 'U' – niezainicjalizowany (np. brak resetu).

entity d1z3 is       --  Tutaj są zdeklarowane porty (połączenia wejściowe i wyjściowe) naszego układu
                     -- zostały utworzone przez kreator nowego pliku.
```

```
  Port ( x : in STD_LOGIC_VECTOR (2 downto 0); -- Sygnał x jest portem wejściowym, wektorem (magistralą)
                                              -- składającym się z trzech bitów (indeksy od 2 do 0).
        y : out STD_LOGIC);  -- sygnał wyjściowy pojedynczy bit
end d1z3;

architecture Behavioral of d1z3 is        – to tutaj umieścimy kod stanowiący funkcjonalność naszego układu.
            -- miejsce deklaracji sygnałów wewnętrznych
begin
 -- nasza logika: y<= ……………………..
end Behavioral;
```

Ważną informacją jest, że język VHDL **nie jest językiem typu case sensitive**. Oznacza to, że wielkość liter nie ma znaczenia. Dla przykładu, zapis *x* jest równoznaczny zapisowi *X*. Należy o tym pamiętać podczas deklarowania nazw portów, sygnałów, zmiennych.

Układy w języku VHDL można opisywać operując wprost na bramkach logicznych. Wynik naszej tablicy Karnaugh można zapisać w następujący sposób:

```
y <= (x(0) and not x(1) and not x(2)) or (not x(0) and x(1) and not x(2))
 or (not x(0) and not x(1) and x(2));
```

Jak widać nasz układ opiera się na trójwejściowej bramce OR oraz trzech trójwejściowych bramkach AND. Klauzula *not* symbolizuje inwerter. Ten sposób projektowania jest wymagany podczas pierwszego projektu lub też dla projektów wykonywanych w schemacie (zamiast tekstu wykorzystuje się wtedy bramki oraz odpowiednie ich połączenie).

Język opisu sprzętu umożliwia nam jednak pisanie kodu na wyższym poziomie abstrakcji, dzięki czemu kod jest czytelniejszy i szybszy w projektowaniu. Do opisania funkcjonalności naszego układu możemy użyć również konstrukcji *with-select*. Proszę zapoznać się z poniższym kodem.

```
31 architecture Behavioral of d1z3_main is
32
33 begin
34
35     with x select
36         y <= '1' when "001",
37              '1' when "010",
38              '1' when "100"
39              '0' when others;
40
41 end Behavioral;
```

Konstrukcja *with-select* ustala stan na porcie *y* w zależności od stanu portu *x*. Trzy bity portu *x* są trzema wejściami naszego układu jeden z trzech. Jeśli tylko na jednym z nich odczytamy stan '1', na wyjściu *y* również ustalamy stan '1'. Klauzula *others* oznacza wszystkie inne możliwe stany. Jest bardzo użyteczna, bowiem pozwala na uniknięcie niejasności – sytuacji, gdy dla jednej lub więcej kombinacji wejściowych nie zostanie ustalony stan wyjściowy. W takim przypadku, na wyjściu pozostanie stan ustalony wcześniej (układ sekwencyjny) lub stan nieokreślony 'X' lub wysokiej impedancji 'Z'. Proszę również zauważyć, że pojedynczy bit logiczny bierzemy w apostrof (np. '0' lub '1'), a ciąg bitów (vector) - w cudzysłów (np. "010").

Możemy też użyć prostszej i bardziej uniwersalnej konstrukcji z słowami kluczowymi when else:

```
architecture Behavioral of d1z3 is
begin
  y<= '1' when x="100" or x="010" or x="001" else '0';
end Behavioral;
```

Więcej możliwych sposobów zapisu jest dostępnych w menu: Tools / Language Templates / VHDL, gdzie można znaleźć opis podstawowej składni języka VHDL pod kątem zarówno symulacji jak i syntezy.

W naszym projekcie użyjemy jednak konstrukcji strukturalnej z deklaracją dodatkowego sygnału *yp*, co przedstawia poniższy kod:

```
architecture Behavioral of d1z3 is
 signal yp: std_logic_vector(2 downto 0); -- sygnał wewnętrzny – tylko w tym miejscu można deklarować sygnały
begin
 yp(0)<= x(0) and not x(1) and not x(2);  -- aktywny dla 100
 yp(1)<= not x(0) and x(1) and not x(2);  -- aktywny dla 010
 yp(2)<= not x(0) and x(1) and x(2); -- aktywny dla 001?
 y<= yp(0) or yp(1) or yp(2);
end Behavioral;
```

Proszę zapisać plik poprzez naciśnięcie kombinacji klawiszy *Ctr + s* lub wybranie menu File/ Save File.

4. Plik symulacyjny *testbench*

Następnym krokiem jest stworzenie pliku vhdl tzw. *testbench* sprawdzającego poprawność działania naszego modułu. W pliku tym musimy określić zbiór stanów wejściowych (wymuszeń czasowych). Wymuszenia są również określane w języku VHDL, wynik (wyjściowe przebiegi czasowe) można obejrzeć podczas symulacji.

Aby dodać plik *testbench* należy kliknąć ikonę *Add Sources* (podobnie jak przy pliku d1z3.vhd).

Pojawi się okno kreatora nowego pliku. Proszę wybrać opcję *Add or create simulation sources* i kliknąć przycisk *Next*. W kolejnym oknie klikamy przycisk *Create file*. W nowym okienku wpisujemy nazwę pliku – zwyczajowo nazwa pliku symulowanego z końcówką *_tb* (czyli *d1z3_tb*). Klikamy *Finish* i możemy zmienić nazwę architecture na *sim*, nie dodajemy portów we/wy (plik symulacyjny z reguły nie posiada portów), klikamy *OK*.

Został utworzony nowy plik w folderze Simulation Sources w drzewie naszego projektu. Proszę kliknąć podwójnie na nazwę *d1z3_tb – sim*. Po otwarciu pliku ujrzymy standardowy szkielet kodu przygotowany przez program Vivado.

Podstawowym zadaniem podczas pisania *testbencha* jest osadzenie modułu, który będzie symulowany (UUT – *Unit Under Test*), czyli w naszym przypadku modułu *d1z3*. Osadzenie to jest podobne do wywołania funkcji w języku C/C++: oprócz nazwy modułu *d1z3* należy dodać nazwę etykiety (poprzedzoną dwukropkiem), słowo kluczowe *entity.work,* słowo kluczowe *port map* oraz odpowiednio zmapować porty we/wy (*port map*). Przedstawia to poniższy kod:

UUT: entity work.d1z3 port map (x=> x, y=> y_uut); -- Unit Under Test

Podobnie należy postąpić w przypadku projektu hierarchicznego, czyli takiego który zawiera inne (pod)moduły. Proszę zwrócić uwagę na użycie etykiety ‘UUT:’, która bezpośrednio identyfikuje podmoduł podczas symulacji czy też implementacji – ma to szczególne znaczenie, kiedy użyjemy dwa takie same moduły. Podobnie jest z mapowanymi sygnałami, np. sygnałowi *y* modułu *d1z3*, przypisano nazwę *y_uut* w module *d1z3_tb*.

Pełny tekst pliku *d1z3_tb.vhd* jest podany poniżej (proszę go odpowiednio wkleić do projektu).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; -- umożliwia dokonanie operacji dodawania na std_logic_vector

entity d1z3_tb is  -- brak porów we/wy
end d1z3_tb;

architecture sim of d1z3_tb is
 -- deklaracja sygnałów – muszą one odpowiadać sygnałom modułu testowanego
 signal x: std_logic_vector(2 downto 0):= "000";  -- sygnał wejściowy z wartością początkowa "000"
 signal y_uut, y_correct: std_logic;  -- sygnał wyjściowy z modułu UUT, sygnał wyjściowy poprawny
```

```
 signal error: std_logic;  -- sygnał błędu, świadczący, że logika UUT nie działa poprawnie
begin
 UUT: entity work.d1z3 port map (x=> x, y=> y_uut);  -- Unit Under Test

 -- poprawna wartość - kod nie jest realizowany w sprzęcie więc może być napisany na wyższym poziomie
 y_correct<= '1' when ('0' & x(0)) + x(1) + x(2) = 1 else '0';
 error<= '1' when y_uut /= y_correct else '0'; -- sprawdzenie czy nie ma błędu, '/=' – różny '!='

 -- wymuszenia
 -- SPOSOB A
 x<= "000", "001" after 10ns, "010" after 20ns;
 -- SPOSOB B
 process begin
  wait for 10ns;  -- czekaj przez 10ns
  x<= "001";
  wait for 10ns;
  x<= "001";
  wait; -- czekaj w nieskonczonosc
 end process;
 -- sposob C
 process begin
  wait for 10ns;
  x<= x+1;
 end process;
end sim;
```

Kolejnym krokiem jest określenie poprawnego sygnału wyjściowego, w naszym wypadku *y_correct*. Nie jest to teoretycznie wymagane do przeprowadzenia symulacji, niemniej w wielu przypadkach jest to konieczne – obliczenie wartości poprawnej sygnału dla milionów możliwych kombinacji i sprawdzenie poprawności przebiegów w sposób ręczny jest niezmiernie kłopotliwe. Proszę zwrócić uwagę, że sygnał *y_correct* może być obliczany w sposób dużo prostszy (na wyższym poziomie abstrakcji) niż w module UUT, który jest implementowany w sprzęcie. Bardzo często poprawny sygnał jest generowany np. w matlabie i odczytywany z pliku. Dlatego kod VHDL zdecydowanie różni się dla symulacji i dla implementacji. Sygnał wykrywający błędne działanie modułu: *error* jest w stanie wysokim '1' w momencie, kiedy wartości rzeczywista *y_uut* i poprawna *y_correct* są różne.

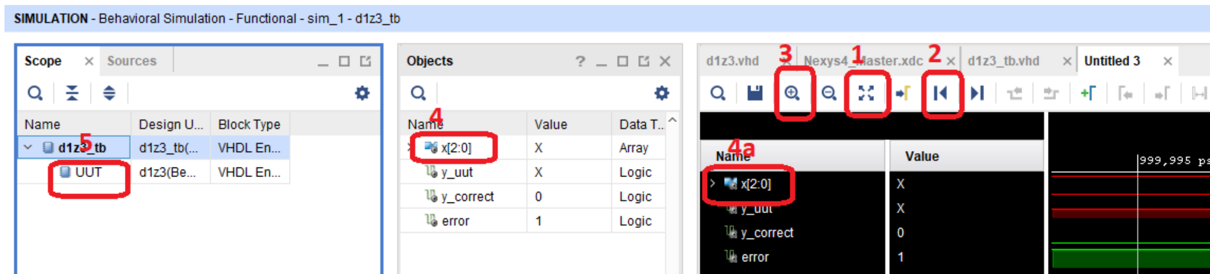
Kolejnym krokiem jest określenie sygnałów wymuszających (wejściowych dla testowanego modułu, czyli *x*). Warto zwrócić uwagę, że w VHDL standardowo instrukcje wykonywane są równolegle (współbieżnie), tak działają moduły sprzętowe, np. każda z bramek logicznych działa niezależnie od innych i nie czeka na podanie wyniku poprzedniej bramki, jak to odbywa się w programie np. w języku C/C++. Dlatego kolejność instrukcji w VHDL nie ma znaczenia. Wyjątek stanowią instrukcje czekania: *wait, after* lub instrukcje w bloku *process*. W bloku *process*, kolejność wykonywania operacji może (ale nie musi) mieć znaczenie. Dlatego aby opisać układ z zależnością czasową (układ sekwencyjny lub wektor wymuszeń zmieniający się w czasie), konieczne jest użycie bloku *process*. W nim ustalane są kolejne wartości sygnałów np. (x <= "001") na okres 10ns (instrukcja wait for). Instrukcja czekania (wait for) może być wykonywana tylko podczas symulacji i w bloku process – nie ma możliwości implementacji bezpośredniej tej instrukcji w sprzęcie.

5. Symulacja

Należy się upewnić, który moduł będzie symulowany (jest nadrzędny - top level). W oknie *Sources* należy sprawdzić czy moduł *d1z3_tb* jest napisany pogrubioną czcionką. Jeśli moduł *d1z3_tb* nie jest modułem nadrzędnym, to należy kliknąć go prawym przyciskiem myszki i wybrać menu: *set as top*. Powyższa czynność

jest szczególnie ważna przy projektach, które zawierają więcej niż jeden plik *testbench*. W przeciwieństwie do np. języka C/C++ moduł nadrzędny nie jest określony za pomocą nazwy funkcji: *main*().

Następnie klikamy w lewym oknie: *Run Simulation*, a później *Run Behavioral Simulation*. Po chwili oczekiwania wyświetli się widok symulacji przedstawiony na poniższym rysunku.



Najpierw musimy odpowiednio przeskalować przebiegi czasowe. Dlatego przyciskamy ikonę 1 (na powyższym rysunku) *Zoom Fit*, a następnie ikonę 2 (Go to time 0) oraz ikonę 3 (Zoom in) tak aby oglądnąć pierwsze 80 ns symulacji, które są najbardziej interesujące. Niestety wektor wymuszeń *x* jest określony tylko przez 10ns, a po tym czasie ustawiony jest stan 'X' – niezdefiniowany. Stan 'X' często świadczy, że w układzie następuje „zwarcie" (wymuszenie tego samego sygnału przez dwa lub więcej wyjść), w przeciwieństwie do stanu 'U', który często świadczy o rozwarciu (braku wymuszenia lub też braku inicjalizacji stanu początkowego).

Dlatego, najlepiej nie zamykając symulacji, należy poprawić plik *d1z3_tb*, w którym sygnał *x* jest wymuszany na trzy różne sposoby. Należy wybrać tylko jeden ze sposobów – najlepiej sposób C, który wymusza wszystkie możliwe stany sygnału *x*. Pozostałe sposoby należy albo usunąć albo zakomentować. Po poprawieniu i zapisaniu pliku *d1z3_tb* (Ctr + s) należy przejść do symulacji i wybrać menu: *Run / Relaunch Simulation*, co spowoduje ponowne uruchomienie symulacji z zachowaniem ustawień okna symulacji.

Inną ciekawą funkcją symulatora jest możliwość zmiany formatu wyświetlanych sygnałów. W tym celu należy np. kliknąć prawym przyciskiem myszki sygnał *x* (ikona 4 lub 4a) i wybrać *radix / binary*. Można też rozwinąć magistralę *x* przyciskając strzałkę obok sygnału *x* (ikona 4a). Można też wyświetlić sygnał *x* w sposób analogowy klikając prawym przyciskiem myszki sygnał *x* (ikona 4a) i wybierając *Waveform Style / Analog*.

Podczas symulacji można również wyświetlać sygnały wewnętrzne poszczególnych podmodułów. W tym celu należy wybrać moduł UUT (ikona 5). Wtedy w oknie *Objects* pojawi się sygnał wewnętrzny *yp.* Sygnał ten należy przenieś myszką do okna z przebiegiem czasowym. Niestety przebieg sygnału *yp* nie zostanie wyświetlony. Dlatego należy restartować symulację poprzez menu: *Run / Restart* oraz uruchomić ją powtórnie *Run / Run for* (100ns).

Czy nasza symulacja jest poprawna? Kiedy sygnał *error* wskazuje błąd, dla jakiego *yp* to występuje. Popraw kod w pliku *d1z3.vhd* i uruchom symulację powtórnie (Run / Relaunch). Po otrzymaniu poprawnego wyniku symulacji, zamykamy okno symulacji symbolem X (proszę odczekać parę sekund).

6. Edycja pliku .xdc

Bitstream jest plikiem wynikowym całego procesu projektowania. Jest to plik, którym programowany jest układ FPGA. Aby wykonać implementację projektu, musimy jeszcze dokonać zmian w pliku .xdc., w tym celu w drzewie hierarchii w folderze *Constraints* znajdujemy plik *Nexys4_Master.xdc*. Klikamy go podwójnie aby go otworzyć.

Plik ten przygotowany jest przez firmę Digilent i służy przypisaniu portów wejścia i wyjścia do fizycznych wyprowadzeń (ang. pin) na płytce ewaluacyjnej. Dla naszego układu jeden z trzech potrzebujemy jednej diody LED która będzie sygnalizować stan wyjścia *y* oraz trzech przełączników (ang. switch) które posłużą jako porty wejściowe *x*. Proszę przygotować plik zgodnie ze wzorem.

```
## Switches
##Bank = 34, Pin name = IO_L21P_T3_DQS_34,                Sch name = SW0
set_property PACKAGE_PIN U9 [get_ports {x[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[0]}]
##Bank = 34, Pin name = IO_25_34,                         Sch name = SW1
set_property PACKAGE_PIN U8 [get_ports {x[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[1]}]
##Bank = 34, Pin name = IO_L23P_T3_34,                    Sch name = SW2
set_property PACKAGE_PIN R7 [get_ports {x[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[2]}]

## LEDs
##Bank = 34, Pin name = IO_L24N_T3_34,             Sch name = LED0
set_property PACKAGE_PIN T8 [get_ports {y}]
set_property IOSTANDARD LVCMOS33 [get_ports {y}]
```

Aby „przypiąć" port do któregoś z wyprowadzeń, należy odnaleźć odpowiednią linijkę wedle oznaczeń z prawej strony – my przypisujemy bity portu *x* do switchy *SW0*, *SW1*, *SW2* a port y do diody oznaczonej *LED0*. Po odnalezieniu linii należy ją odkomentować usuwając znak krzyżyka ('#'), następnie po klauzuli *get_ports* w nawiasach klamrowych należy wpisać nazwę portu. Jako, że nasz port *x* jest wektorem sklejonym z trzech bitów, a jeden switch może ustawiać tylko jeden bit musimy je w jakiś sposób „wyłuskać". Do tego służą nawiasy kwadratowe, gdzie *x[0]* jest bitem najmłodszym a *x[2]* - najstarszym. Tak przygotowany plik należy zapisać.

7. Synteza, implementacja i symulacja czasowa

Proszę kliknąć napis Synthesis / Run Synthesis i OK w lewym oknie programu Vivado. Spowoduje to, że nasz moduł *d1z3* opisany w języku VHDL zostanie zsyntesowany, czyli zostanie określony jego odpowiednik schematu, składającego się z podstawowych elementów dostępnych w układach FPGA. Aby zobaczyć wynik syntezy należy wybrać: *Open Synthesized Design / Schematic*. W wyniku zostanie wyświetlony schemat naszego modułu, niestety nie ma w nim bramek a jedynie bufory wejściowe IBUF i wyjściowe OBUF oraz moduł LUT3 – pamięć Look-Up Table (LUT), która realizuje naszą logikę, nie ma żadnych bramek logicznych. Następnie proszę zamknąć schemat znakiem x.

Następnym etapem jest uruchomienie implementacji: *Implementation / Run Implementation*. W wyniku implementacji każdy element logiczny zostanie osadzony w odpowiednim miejscu w układzie FPGA i zostaną wyznaczone połączenia pomiędzy tymi elementami (Place & Route). W raporcie implementacji pokazane będą zużyte zasoby układ FPGA: *Implementation / Open Implemented Design / Report Utilization*.

W wyniku implementacji zostaną też wyznaczone czasy propagacji przez elementy logiczne i połączenia. Aby sprawdzić skąd wynikają te opóźnienia w otwartej implementacji wybierz *Report Timing Summary*, w sekcji *Path Limits* zwiększ *Maximum number of worst paths per endpoint* do 12, kliknij OK. W zakładce *Timing*, sekcji *Unconstrained Paths/NONE to NONE/Setup* wybierz sygnał x[0] (poprzez dwukrotne kliknięcie) i porównaj go z sygnałem x[1]. Aby sprawdzić jak zostały poprowadzone ścieżki przejdź do zakładki *Device* (jeżeli nie jest otwarta wybierz menu *Window/Device*) a następnie na górze zakładki zaznacz ikonkę *Routing Resources*, klikając na rożne sygnały w karcie *Timing* można zobaczyć konkretne ścieżki którymi zostały poprowadzone. Po dużym przybliżeniu można prześledzić jakie elementy cyfrowe zostały użyte do zaimplementowania naszego projektu.

Następnym krokiem jest przeprowadzenie symulacji czasowej poprzez uruchomienie: *Simulation / Run Simulation / Run Post-Implementation Timing Simulation.* W wyniku tego zostanie przeprowadzona symulacja, w której widoczne będą czasy propagacji oraz hazardy na sygnale *y_uut.* Prześledź wynik tej symulacji. Warto podkreślić, że czasy propagacji podczas symulacji (i raportów czasowych) są z reguły określone na poziomie

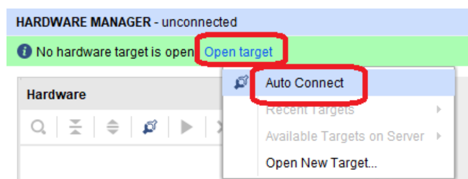
maksymalnych wartości. Dlatego rzeczywiste przebiegi mogą się różnić, nie tylko poprzez skrócenie czasów propagacji ale również mogą powstawać dodatkowe hazardy i wyścigi. Symulacja czasowa trwa dłużej (program wymaga więcej obliczeń) i może wykrywać błędy w projekcie, które nie ujawniają się na poziomie symulacji funkcjonalnej (behawioralnej). Niemniej nawet symulacja czasowa nie daje nam 100% pewności, że układ działa poprawnie. Dlatego bardzo ważne jest aby unikać podczas projektowania możliwości wystąpienia zjawiska wyścigów, hazardów czy metastabilności (przejścia pomiędzy blokami taktowanymi różnymi niezsynchronizowanymi zegarami) ponieważ wykrycie tych zjawisk w rzeczywistym układzie jest bardzo trudne.

8. Programowanie układu FPGA

Następnym etapem jest zaprogramowanie układu FPGA. Aby tego dokonać należy wygenerowanie plik *bistreamu*, czyli plik z rozszerzeniem .bit. Aby to dokonać w lewym oknie należy wybrać: *Program and Debug / Generate Bitstream* a następnie w otwartym oknie wybrać OK. W kolejnym oknie (*Bitstream Generation Completed*) sugerującym następną czynność należy wybrać *Cancel*.

Następnie należy poprosić prowadzącego o płytkę z układem FPGA i podłączyć ją kablem z portem USB komputera. Zasilanie jest dostarczane również przez ten kabel USB. Powinna się zapalić czerwona dioda świadcząca o włączonym zasilaniu, jeśli nie należy ustawić przełącznik zasilania (*POWER*) w pozycję *ON*.

Następnie programujemy układ FPGA wygenerowanym uprzednio plikiem **.bit*. Można tego dokonać wybieramy w lewym oknie: *Program and Debug / Open Hardware Manager.* W otwartym oknie (zob. poniższy rysunek) wybieramy *Open target* a później *Auto Connect*, co spowoduje wykrycie podłączonej płyty z układem FPGA.



Następnie w oknie *Hardware manager* wybieramy *Program Device.* W otwartym oknie wybieramy przycisk *Program*, spowoduje to zaprogramowanie naszego układu FPGA co zostanie potwierdzone zapaleniem zielonej diody *DONE* na płytce z układem FPGA. Proszę sprawdzić poprawność działania naszej logiki *d1z3* zmieniając stany przełącznikami *SW0-SW2* i obserwując diodę *LD0*. Przełącznik przesunięty w górę jest logiczną ‘1’, w dół logicznym ‘0’. Świecąca dioda LED jest logiczną ‘1’, zgaszona logicznym ‘0’. Po sprawdzeniu działania należy zamknąć projekt (menu *File / Close Project*).

9. Archiwizacja projektu

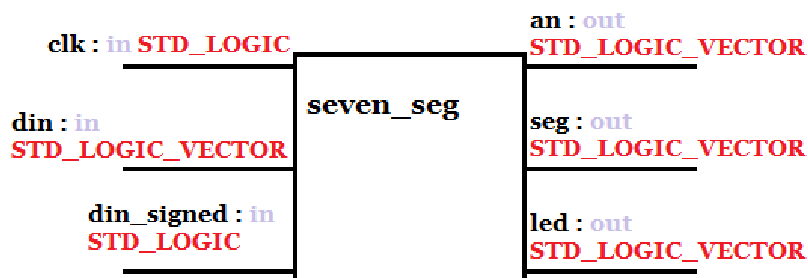
Jeżeli chcemy zapisać projekt w celu przeniesienia go na inne urządzenie lub archiwizacji, możemy użyć opcji *File/Archive Project…* zapisuje ona nasz projekt w skompresowanej formie. Jeżeli chcemy zapiać również wyniki syntezy i implementacji możemy zaznaczyć *Include run results*, dzięki czemu nie będziemy musieli wykonywać ich jeszcze raz, jest to szczególnie przydatne w przypadku dużych projektów których implementacja trwa bardzo długo (wiele godzin). Odbywa się to jednak kosztem dużo większej wielkości pliku wynikowego.

Część dodatkowa (wykonywana jeśli starczy czasu)

10. Wstęp teoretyczny do wyświetlacza siedmiosegmentowego.

Niniejszy punkt jest czysto teoretyczny – nic nie należy robić. Do instrukcji dołączony jest plik *seven_seg.vhd*. Jest to plik z kodem VHDL który dodamy do naszego kodu jako gotowy moduł tzw. czarna skrzynka, której wewnętrznej zawartości nie musimy rozumieć, natomiast ważna jest jego funkcjonalność z punktu widzenia użycia tego modułu w projekcie (czyli porty wejścia/wyjścia). Funkcjonalność układu *seven_seg*:

- wyświetlanie wyniku z wykorzystaniem wyświetlacza siedmiosegmentowego
- wyświetlanie wyniku z wykorzystaniem diod LED
- tryby wyświetlania: binarny (LED), decymalny lub heksadecymalny (7seg)
- na starszych 4 cyfrach wyświetlana jest liczba w kodzie dziesiętnym, a na młodszych czterech cyfrach wyświetlana jest liczba w kodzie heksadecymalnym.



Seven_seg posiada 6 portów. Wejściowe:

- *clk* typu *STD_LOGIC* – należy dopiąć do niego sygnał zegarowy płytki Nexys4 (100MHz)
- 8 bitowy *din* typu *STD_LOGIC_VECTOR(7 downto 0)* – należy dopiąć sygnał ośmiobitowy zawierający liczbę do wyświetlenia
- *din_signed* typu *STD_LOGIC* – podanie '0' oznacza, że dana wejściowa *din* jest liczbą bez znaku, podanie '1' oznacza, że jest liczbą zapisaną w kodzie uzupełnień do dwóch

Porty wyjściowe:

- 8bitowy *an* typu *STD_LOGIC_VECTOR*. Bity należy przypisać kolejno do pinów anod wyświetlacza 7segmentowego
- 8bitowy *seg* typu *STD_LOGIC_VECTOR*. Bity należy przypisać kolejno do pinów segmentu wyświetlacza 7segmentowego
- 8bitowy *led* typu *STD_LOGIC_VECTOR*. Bity należy przypisać kolejno do pinów diod LED, wyświetlają one stan sygnału *din* w sposób binarny na diodach LED.

Powód występowania portów *an* i *seg* wykracza poza zakres tego kursu. Porty te obsługują wyświetlacz 7segmentowy i należy potraktować je jako część modułu. Dla dociekliwych: krótki opis działania wyświetlacza 7segmentowego można znaleźć w pliku Reference Manual dla płytki Nexys4, punkt 9.1, strony 18-20.

11. Przygotowanie nowego projektu i dodanie komponentu wyświetlacza

Proszę utworzyć nowy projekt o nazwie *bin_arith* w programie Vivado. Typ projektu: *RTL Project*. W oknie *Add sources* za pomocą *Create file* dodajemy plik *bin_arith_main*. Za pomocą *Add files* dodajemy do projektu plik *seven_seg.vhd*. Pozostawiamy odznaczoną opcję *Scan and add RTL include files into project*. Pozostawiamy zaznaczoną opcję *Copy sources into project*. Dalsze czynności wykonujemy analogicznie do pierwszej części ćwiczenia. W kreatorze portów dodajemy:

- wejściowy *clk* typu *STD_LOGIC*
- wyjściowy *an* typu *STD_LOGIC_VECTOR(7 downto 0)*
- wyjściowy *seg* typu *STD_LOGIC_VECTOR(7 downto 0)*
- wyjściowy *led* typu *STD_LOGIC_VECTOR(7 downto 0)*

Proszę otworzyć do edycji plik *bin_arith_main*. Proszę pozostawić w kodzie linie 25-27 i usunąć linie 29-32. Proszę dodać deklarację komponent *seven_seg* w bloku *architecture* – wykonywaliśmy podobne zadanie przy kreowaniu *testbencha* w części pierwszej ćwiczenia. W tym celu najlepiej jest skopiować sekcję entity z pliku *seven_seg.vhd* i zamienić słowo kuczowe *entity* na *component*. Kod dodania komponentu powinien wyglądać następująco:

```
44 architecture Behavioral of bin_arith_main is
45
46     component seven_seg is
47         Port ( clk : in STD_LOGIC;
48                din : in STD_LOGIC_VECTOR(7 downto 0);
49                din_signed : in STD_LOGIC;
50                an : out STD_LOGIC_VECTOR(7 downto 0);
51                seg : out STD_LOGIC_VECTOR(7 downto 0);
52                led : out STD_LOGIC_VECTOR(7 downto 0)
53            );
54     end component;
```

Nie wszystkie porty komponentu *seven_seg* zmapujemy bezpośrednio do portów *bin_arith_main*. Część z nich zmapujemy do sygnałów wewnętrznych bloku *architecture*. Sygnały te trzeba zadeklarować poniżej deklaracji komponetu a przed słowem kluczowym *begin.* Sygnały nie muszą mieć takiej samej nazwy jak porty komponentu, ale mogą – ułatwia to pracę z kodem. Deklaracja sygnałów powinna wyglądać następująco:

```
signal din :  STD_LOGIC_VECTOR(7 downto 0); --wynik do wyswietlenia
signal din_signed :  STD_LOGIC :='0'; -- wybór trybu, '0' binarny bez znaku, '1' uzupełnienie do 2
signal a :  STD_LOGIC_VECTOR(7 downto 0); -- argument 1 operacji dodawania
signal b :  STD_LOGIC_VECTOR(7 downto 0); -- argument 2 operacji dodawania
signal result : STD_LOGIC_VECTOR(7 downto 0); -- wynik operacji dodawania
signal tmp0 : STD_LOGIC_VECTOR(6 downto 0); -- wynik dodawania, młodsze bity
signal tmp1 : STD_LOGIC_VECTOR(1 downto 0); -- wynik dodawania, starsze bity
```

Po definicji sygnału *din_signed* widzimy dopisek *:= '0'* - jest to domyślna wartość sygnału po uruchomieniu układu. Sygnały warto opisać komentarzem – jest to dobry nawyk programistyczny nawet przy małych projektach. Po klauzuli *begin* proszę zmapować komponent *seven_seg*.

```
69 begin
70     mapping: seven_seg port map ( clk => clk,
71                                   din => result,
72                                   din_signed => din_signed,
73                                   an => an,
74                                   seg => seg,
75                                   led => led);
```

12.Przełączniki i przyciski, konstrukcja warunkowa *if*

1. Przełączniki

Do przekazywania 8-bitowych liczb binarnych do układu użyjemy przełączników (ang. switch). Płytka Nexys4 posiada ich 16 sztuk – ilość dokładnie wystarczającą na dwie 8-bitowe liczby. Proszę dodać do układu port wejściowy *sw* typu *STD_LOGIC_VECTOR(15 downto 0)*. Aby to zrobić należy dodać w bloku entity, w miejscu definicji portów linijkę:

```
31          sw : in STD_LOGIC_VECTOR(15 downto 0);
```

Będą one reprezentować nasze liczby. Następnie, w bloku *architecture* po klauzuli *begin* i linii mapowania proszę dodać linijki:

```
79    a <= sw(15 downto 8);
80    b <= sw(7 downto 0);
```

Jak widać, pierwsze osiem przełączników (patrząc na płytkę od lewej) zostało przypisane sygnałowi *a.* Kolejne osiem – *b.* Takie przypisanie powoduje, że *sw[15]* zostaje przypisane *a[7]* itd. Oznacza to, że najbardziej znaczący bit pierwszej liczby jest ustawiany przełącznikiem nr 15, a najbardziej znaczący bit drugiej liczby jest ustawiany przełącznikiem nr 7.

2. Przyciski

Płytka Nexys4 wyposażona jest w 5 przycisków (ang. pushbutton) w kształcie znaku '+'. Dwa z nich użyjemy do wybierania działania (proponowane: górny – dodawanie, dolny – odejmowanie). Kolejne dwa do wybierania typu zapisu liczb (proponowane: lewy – binarne bez znaku, prawy – kod uzupełnień do dwóch).

Proszę dodać do deklaracji portów cztery nowe porty wejściowe typu STD_LOGIC:

```
32          btnU : in STD_LOGIC;
33          btnD : in STD_LOGIC;
34          btnL : in STD_LOGIC;
35          btnR : in STD_LOGIC;
```

Oznaczenia symbolizują słowo *button* i kolejno: *Upper, Down, Left, Right*.

Teraz zadeklamowane są już wszystkie porty. Pozostało nam zmodyfikowanie istniejący pliku .xdc, przypisujący nazwy portu do konkretnego wyprowadzenia układu FPGA. Można to wykonać ręcznie tak jak poprzednio, ale ponieważ liczba użytych sygnałów jest duża najlepiej przekopiować gotowy już plik *bin_arith_main.xdc*, załączony do tego ćwiczenia.

3. Konstrukcja *if*

Do opisania działania kolejnych przycisków potrzebna nam będzie konstrukcja warunkowa *if*. Działa ona podobnie jak w innych językach programowania, charakterystyczne jest jednak słowo kluczowe *elsif*. Oto jej składnia:

Składnia konstrukcji *if:*

```
if warunek then
        polecenia do wykonania;
elsif warunek then
        polecenia do wykonania;
else
```

polecenia do wykonania;

end if;

Konstrukcja *if* musi znajdować się w bloku *process*. Kod dla wyboru typu zapisu liczb (przyciski *btnL* i *btnR*) wygląda następująco:

```
123    process(clk)
124    begin
125        if(rising_edge(clk)) then
126            if (btnL = '1') then -- wybor binarny bez znaku
127                din_signed <= '0';
128            elsif (btnR = '1') then -- wybór U2
129                din_signed <= '1';
130            end if;
131        end if;
132    end process;
```

Jako, że w bloku *process* nie używamy funkcji *wait for*, musimy określić sygnały aktywacji procesu, są to sygnały występujące w nawiasach po słowie kluczowym *process*. Zmiana któregokolwiek sygnału aktywującego spowoduje wykonanie kodu wewnątrz procesu. Teoretycznie przyspiesza to proces symulacji ponieważ kod wewnątrz procesu jest wykonywany tylko przy zmianie wybranych sygnałów. W naszym przypadku wybieramy sygnał *clk* jako sygnał aktywujący, ponieważ używamy automatu synchronicznego, czyli takiego który reaguje tylko na narastające zbocze sygnału zegarowego. Funkcja *rising_edge()* zwraca ‘1’ tylko w razie wykrycia zbocza narastającego sygnału. Można również zauważyć, że konstrukcja *if* opisująca przycisk jest zagnieżdżona w kolejnej konstrukcji *if*. Proszę również zauważyć, że klauzula *if then else end if* może być użyta tylko wewnątrz procesu. Poza procesem można użyć podobnej klauzuli *when else*, np. *y <= ‘1’ when x = ‘1’ else ‘0 when x = ‘0’ else ‘X’.*

Proszę samodzielnie, w tym samym (lub innym) bloku *process*, napisać kod obsługujący wybór działania (dodawanie, odejmowanie). W tym celu należy użyć przycisków *btnU* i *btnD* oraz utworzyć kolejny sygnał w bloku *architecture* o nazwie np. *op_choice*. Wartość domyślna powinna wynosić ‘0’ (dodawanie). Po wykonaniu zadania proszę zapisać plik i uruchomić syntezę w celu sprawdzenia poprawności kodu.

13. Operacje arytmetyczne

W tej części instrukcji zajmiemy się napisaniem kodu zawierającego główną funkcjonalność naszego układu – dodawanie i odejmowanie liczb.

```
82    tmp0 <= ('0' & a(6 downto 0)) +  b(6 downto 0) when op_choice = '0' else
83            ('0' & a(6 downto 0)) - b(6 downto 0);
84    tmp1 <= ('0' & a(7) ) + ('0' & b(7)) + tmp0(7) when op_choice = '0' else
85            ('0' & a(7) ) - ('0' & b(7)) - tmp0(7);
86    result <= tmp1(0) & tmp0(6 downto 0);
87    zero <= '1' when result = 0 else '0';
88    carry <= tmp1(1);
89    overflow <= tmp1(1) xor tmp0(7);
```

Do poprawnego działania kodu należy dodać bibliotekę *IEEE.STD_LOGIC_UNSIGNED.ALL*. Celem działania układu oprócz wykonywania operacji dodawania / odejmowania jest ustawianie flag: *zero carry* oraz *overflow*. Proszę dodać do naszego układu trzy kolejne porty wyjściowe typu *STD_LOGIC*: *zero, carry, overflow*. W pliku .xdc są one przypisane odpowiednio do diod LED 9, 10 i 11. Operacje dodawania i odejmowania są wykonywane tak samo zarówno w przypadku liczb bez znaku jak i kodu U2. W liniach kodu nr 82 i 84 realizowane jest dodawanie, w liniach 83 i 85 odejmowanie. Operator *&* służy do łączenia wektorów (tworzenie szerszego wektora z dwóch innych wektorów).

Na początku, w liniach kodu nr 82-83 zajmujemy się pierwszymi <u>7 bitami</u> naszych liczb. Do liczby *a* doklejany jest 8 bit o wartości '0'. W przypadku dodawania, gdy wynik nie zmieści się na 7 bitach, ustawiony zostanie ósmy bit (przeniesienie). W przypadku odejmowania, bit zostanie ustawiony jeśli zajdzie potrzeba pożyczki (*borrow*). Podobnie odbywa się dla następnego 8 bitu i przeniesienia wyjściowego w liniach kodu nr 84-85. Rozbicie działań na 2 etapy jest konieczne, aby móc prawidłowo ustawiać flagę *overflow*. Proszę pamiętać, że w kodzie U2 najbardziej znaczący bit jest bitem znaku. Flaga *zero* ustawiana jest za pomocą konstrukcji *when-else* w linii kodu nr 87.

Aby zrozumieć warunki ustawiania flag *carry* i *overflow* proszę uzupełnić poniższą tabelę:

dane wejściowe	operacja	>0	≥0	przekroczenie zakresu
bez znaku	dodawanie	not zero	zawsze	carry
bez znaku	odejmowanie			
ze znakiem	dodawanie			
ze znakiem	odejmowanie			