JĘZYK VHDL

w praktyce

Praca zbiorowa pod redakcją Józefa Kalisza

WKŁ

Wydawnictwa Komunikacji i Łączności

WARSZAWA

Okładkę projektował: *Dariusz Litwiniec*

Redaktor merytoryczny: *Elżbieta Gawin*

Redaktor techniczny: *Ewa Kęsicka*

621.38.184.4

Podręcznik zawiera opis języka VHDL, szeroko stosowanego do komputerowego projektowania układów i urządzeń cyfrowych, ilustrowany wieloma przykładami. Omówiono cztery popularne systemy programowe, wykorzystujące język VHDL do projektowania w oparciu o programowalne układy cyfrowe PLD, CPLD i FPGA. Opisano proces instalacji tych systemów, ich zasoby biblioteczne oraz sposoby realizacji projektów i ich testowania poparte licznymi przykładami.

Książka jest przeznaczona dla studentów na kierunkach elektronika, telekomunikacja i informatyka, a także dla profesjonalistów i amatorów - elektroników.

ISBN 83-206-1440-6

M.4088

© Copyright by Wydawnictwa Komunikacji i Łączności sp. z o.o.
Warszawa 2002

Wydawnictwa Komunikacji i Łączności sp. z o.o.
ul. Kazimierzowska 52, 02-546 Warszawa
tel. (0-22) 849-27-51, fax (0-22) 849-23-22
Dział handlowy tel. (0-22) 849-27-51 w. 555
tel./fax (0-22) 849-23-45
Prowadzimy sprzedaż wysyłkową książek
Księgarnia firmowa w siedzibie wydawnictwa
tel. (0-22) 849-20-32, czynna pon.-pt. 10.00 -18.00
e-mail: *wkl@wkl.com.pl*
Pełna oferta WKŁ w INTERNECIE: http://www.wkl.com.pl
Wydanie 1. Warszawa 2002.

Zakłady Graficzne im. KEN S.A.
85-067 Bydgoszcz, ul. Jagiellońska 1

SPIS TREŚCI

WSTĘP *1*
1 PROCES PROJEKTOWANIA UKŁADÓW CYFROWYCH *5*
2 PODSTAWY OPISU UKŁADÓW CYFROWYCH W JĘZYKU VHDL *9*
3 JĘZYK VHDL *23*
3.1 Obiekty *23*
3.2 Typy *25*
3.2.1 Typy skalarne *26*
3.2.2 Typy złożone *31*
3.2.3 Typy wektorowo-skalarne *33*
3.3 Atrybuty *33*
3.4 Pakiety i biblioteki *35*
3.5 Instrukcje współbieżne *38*
3.6 Instrukcje sekwencyjne *43*
3.7 Funkcje i procedury *50*
4 MODELOWANIE ZŁOŻONYCH UKŁADÓW CYFROWYCH *57*
4.1 Sposoby opisu *57*
4.2 Przykłady opisu bloków funkcjonalnych *67*
4.3 Weryfikacja projektów *72*
5 SYSTEM PROJEKTOWY *WebPACK ISE* *77*
5.1 Instalacja systemu *79*
5.2 Organizacja i zasoby biblioteczne systemu *79*
5.3 Płytka testowa *81*
5.4 Proces projektowania *84*
5.4.1 Uruchomienie systemu *84*
5.4.2 Model licznika modulo 10 *84*
5.4.3 Model konwertera kodu BCD na kod wyświetlacza 7-segmentowego LED *86*
5.4.4 Opis projektu w postaci schematu *87*
5.4.5 Symulacja funkcjonalna *89*
5.4.6 Synteza i implementacja projektu *91*
5.4.7 Symulacja czasowa *92*
5.4.8 Programowanie układu *93*
5.5 Przykłady projektów *94*
5.5.1 Konwerter 3-bitowego kodu NB na kod „1 z 8" *94*
5.5.2 Rotator *99*
5.5.3 Licznik modulo 100 w kodzie BCD *102*
5.5.4 Blok liczników równoległych *107*
5.6 Zadania *115*
6 SYSTEM PROJEKTOWY *MAX+plus II* *117*
6.1 Instalacja systemu *118*
6.2 Organizacja i zasoby biblioteczne systemu *119*
6.3 Płytka testowa *123*
6.4 Proces projektowania *128*
6.4.1 Uruchomienie systemu *129*
6.4.2 Opis schematowy sumatora/subtraktora *129*
6.4.3 Symulacja funkcjonalna *133*
6.4.4 Opis w języku VHDL konwertera kodu *138*
6.4.5 Opis schematowy projektu hierarchicznego *142*
6.4.6 Kompilacja projektu *145*
6.4.7 Symulacja czasowa *146*
6.4.8 Programowanie układu *149*

6.5 Przykłady projektów *151*
6.5.1 Ośmiobitowy licznik rewersyjny *151*
6.5.2 Jednostka aytmetyczno-logiczna *158*
6.5.3 Moduł pamięci RAM *162*
6.6 Zadania *163*
7 SYSTEM PROJEKTOWY *IspDesignEXPERT* *165*
7.1 Instalacja systemu *166*
7.2 Organizacja i zasoby biblioteczne systemu *167*
7.3 Płytka testowa *170*
7.4 Proces projektowania *174*
7.4.1 Uruchomienie systemu *174*
7.4.2 Model multipleksera grupowego *177*
7.4.3 Opis schematowy *179*
7.4.4 Opis mieszany *180*
7.4.5 Określanie wymagań projektowych *182*
7.4.6 Kompilacja projektu i tworzenie plików wynikowych *185*
7.4.7 Symulacja funkcjonalna i czasowa *186*
7.4.8 Programowanie układu *188*
7.4.9 Dodatkowe programy użytkowe *189*
7.5 Przykłady projektów *190*
7.5.1 Konwerter kodu BIN/BCD *190*
7.5.2 Generator ciągu pseudolosowego *197*
7.5.3 Generator sygnału prostokątnego *203*
7.5.4 Zamek szyfrowy *207*
7.5.5 Sterownik świateł ulicznych *211*
7.6 Zadania *216*
8 SYMULACJA UKŁADÓW CYFROWYCH W SYSTEMIE PROJEKTOWYM *Active-HDL 5.1* *217*
8.1 Instalacja systemu *218*
8.2 Organizacja i zasoby biblioteczne systemu *219*
8.3 Proces projektowania *220*
8.3.1 Uruchomienie systemu *220*
8.3.2 Organizacja projektu *222*
8.3.3 Opis i kompilacja projektu *222*
8.4 Weryfikacja projektu *225*
8.4.1 Symulacja funkcjonalna *225*
8.4.2 Program testowy *230*
8.4.3 Symulacja czasowa *235*
8.4.4 Schemat procesu projektowania *237*
8.5 Przykłady projektów *239*
8.5.1 Układ sterowania ruchem wózka *239*
8.5.2 Detektor ciągu bitów *243*
8.6 Zadania *248*
LITERATURA *249*
DODATEK. PAKIET WAT *250*
SKOROWIDZ *258*
VHDL QUICK REFERENCE CARD, Qualis Design Corp.
1164 PACKAGES QUICK REFERENCE CARD, Qualis Design Corp.

WSTĘP

Współczesne urządzenia cyfrowe projektuje się często w oparciu o *programowalne* układy cyfrowe. Do zastosowań wielkoseryjnych, lub gdy trzeba spełnić specyficzne wymagania techniczne (niemożliwe do spełnienia przez układy programowalne), stosuje się układy *specjalizowane* (ASIC – *Application Specific Integrated Circuit*).

Klasyczne układy cyfrowe o małej i średniej skali integracji stosuje się już rzadko. Ich obecność w urządzeniu zazwyczaj świadczy o przestarzałości technologicznej projektu.

Projektowanie urządzeń cyfrowych z układami programowalnymi i specjalizowanymi wykonuje się metodami komputerowymi, przy użyciu odpowiednich programów. Do opisu projektu stosuje się różne *języki opisu sprzętu* (HDL – *Hardware Description Language*). Najczęściej stosowane języki to ABEL, VHDL i VERILOG.

Użycie HDL do projektowania urządzeń cyfrowych umożliwia opis struktury lub funkcji urządzenia, jego dekompozycji na mniejsze bloki, oraz połączeń między blokami. Opis ten realizowany jest przy użyciu instrukcji typowych dla języków programowania wysokiego poziomu. Możliwa jest komputerowa symulacja działania oraz sprawdzenie projektu bez potrzeby budowy i badania prototypu, co znacznie skraca czas projektowania i obniża jego koszt.

Język ABEL (i jego późniejsze odmiany firmowe jak AHDL i XABEL) stosuje się zazwyczaj do projektowania przy stosowaniu prostych układów programowalnych PLD (*Programmable Logic Device*) i niewielkich układów CPLD (*Complex PLD*). Język ten nie umożliwia opisu opóźnień czasowych w projektowanych układach, co jest potrzebne do dokumentacji, symulacji i testowania. Niemożliwy jest również opis procesów współbieżnych, występujących powszechnie w układach cyfrowych. Język VHDL nie ma tych wad i może być stosowany uniwersalnie, to znaczy do opisu zarówno prostych jak i bardziej złożonych układów CPLD oraz układów FPGA (*Field-Programmable Gate Array*). Język VERILOG ma zbliżone właściwości do języka VHDL, lecz najczęściej jest stosowany przy projektowaniu układów specjalizowanych.

Język ABEL jest znacznie łatwiejszy do opanowania i stosowania niż język VHDL. Dlatego do prostych projektów lepiej jest stosować ten pierwszy. Mówiąc obrazowo, nie warto używać siekiery do wbicia małego gwoździa. Język ABEL jest opisany m.in. w książce *„Podstawy elektroniki cyfrowej"* (J. Kalisz, WKŁ, wydanie III).

Język VHDL służy do sformalizowanego opisu tekstowego układów cyfrowych, to znaczy ich sposobu działania lub/i struktury logicznej. Opis układu (lub w ogólnym przypadku systemu) cyfrowego w języku VHDL może służyć do celów dokumentacyjnych, jak również do celów projektowych. W ostatnim przypadku opis VHDL stanowi dane wejściowe dla systemu projektowego, który umożliwia analizę, syntezę i weryfikację projektu. W końcowym etapie procesu projektowania tworzone są dane do programatora układu programowalnego albo do wykonania układu specjalizowanego.

Akronim VHDL pochodzi od dość złożonej zbitki słów angielskich *Very (High Speed Integrated Circuit) Hardware Description Language*. Pierwsza wersja języka została opracowana w połowie lat osiemdziesiątych w ramach programu badawczego VHSIC zleconego przez amerykański Departament Obrony (*Department of Defence – DOD*), aby zapewnić kompletność i jednolitość dokumentacji układów i systemów cyfrowych, zamawianych na potrzeby armii amerykańskiej. Szybko się okazało, że język ten jest przydatny także w zastosowaniach cywilnych. Tworzeniem norm tego języka zajął się specjalny komitet normalizacyjny w ramach amerykańskiej organizacji IEEE (*Institute of Electrical and Electronic Engineers*). W latach 1987, 1993 i 2001 ogłoszono kolejne wersje normy IEEE Standard 1076, która jest stale rozwijana i udoskonalana.

Część I (rozdziały 2-4) tej książki zawiera zwięzły opis podstaw języka VHDL w zakresie odpowiadającym studiom wyższym na kierunku elektronika i telekomunikacja. Opis ten jest ilustrowany licznymi przykładami praktycznymi. W części II (rozdziały 5-8) są zawarte opisy systemów projektowych, oferowanych przez firmy amerykańskie *Aldec, Altera, Lattice* i *Xilinx* oraz przykładowe zadania projektowe. Przedstawione zadania odpowiadają typowym studenckim ćwiczeniom laboratoryjnym.

Podany opis języka VHDL oraz przykłady i ćwiczenia projektowe mogą być wykorzystane przez wszystkich konstruktorów, którzy pragną podnieść swoje kwalifikacje i stosować nowoczesne metody projektowania urządzeń cyfrowych.

Zakres książki jest ograniczony do opisu podstaw języka VHDL i niektórych jego zastosowań praktycznych. Istnieje wiele obszernych i

znacznie bardziej wyczerpujących książek poświęconych językowi VHDL i jego aplikacjom. Część z nich jest wymieniona w załączonym wykazie literatury. Aby zapoznać się z bogatą ofertą książek anglojęzycznych można wywołać w Internecie witrynę księgarni wysyłkowej www.amazon.com i wpisać do wyszukiwania słowo VHDL. Jedną z najlepszych książek jest obszerna monografia *„The Designer's Guide to VHDL"* (autor Peter J. Ashenden, *Academic Press*, 2001).

W Internecie można znaleźć wiele informacji związanych z językiem VHDL. Wystarczy wpisać słowo VHDL w dowolnej przeglądarce (na przykład www.altavista.com), aby uzyskać dużo linków do odnośnych witryn. Przykładowo można wymienić następujące:

- www.vhdl.org – międzynarodowa organizacja VHDL;
- tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html – z Uniwersytetu w Hamburgu: znakomite archiwa dokumentów (w tym norm VHDL), liczne linki do doskonałych źródeł;
- www.vhdl-online.de/~vhdl – z Uniwersytetu w Erlangen;
- www.angelfire.com/electronic/in/vlsi/vhdl.html – z Indii.

Firmy produkujące układy programowalne (np. *Xilinx* – www.xilinx.com) zazwyczaj oferują również bezpłatne wersje oprogramowania do projektowania z użyciem języka VHDL.

Niniejszy tekst powstał jako pomoc dydaktyczna w Zakładzie Techniki Cyfrowej Instytutu Systemów Łączności w Wojskowej Akademii Technicznej. Inne książki z zakresu techniki cyfrowej, napisane przez pracowników ZTC i opublikowane przez Wydawnictwa Komunikacji i Łączności, to *„Podstawy elektroniki cyfrowej"* (J. Kalisz), *„Komputerowe projektowanie układów cyfrowych"* (współautor T. Łuba), *„Układy programowalne w praktyce"* (współautor R. Pasierbiński) i *„Mikrokontrolery"* (R. Pełka).

Publikacja tej książki stała się możliwa dzięki aktywnemu zaangażowaniu ulubionego Wydawnictwa (www.wkl.com.pl). Autorzy serdecznie dziękują Red. Elżbiecie Gawin za znakomitą opiekę edytorską, Red. Nacz. Krzysztofowi Wiśniewskiemu za nieocenioną pomoc i życzliwość oraz Prezesowi Bogumiłowi Zielińskiemu za niezwykle przyjazne i kompetentne kierowanie publikacją tej książki.

Autorzy będą wdzięczni Czytelnikom za uwagi i propozycje ulepszeń. Można je przesyłać pod adres jkalisz@wel.wat.waw.pl.

Józef Kalisz

1

PROCES PROJEKTOWANIA UKŁADÓW CYFROWYCH

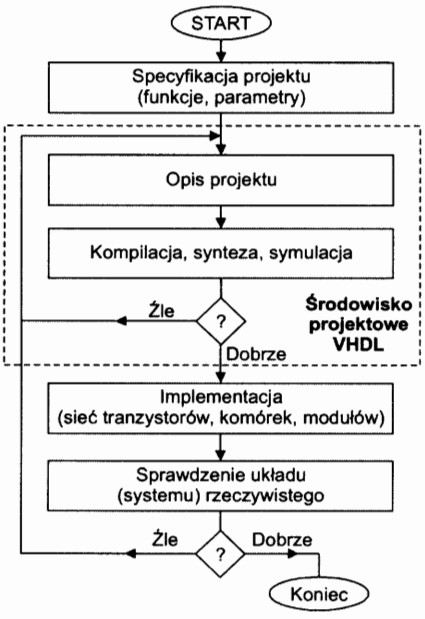
Józef Kalisz

Wykonanie opisu układu (lub systemu) cyfrowego w języku VHDL jest częścią ogólnego procesu projektowania, który ilustruje rys. 1.1. Proces ten rozpoczyna się od sformułowania założeń i wymagań projektowych oraz utworzenia listy funkcji układu. Projekt tworzy się najpierw w postaci opisu w języku VHDL. Jest to plik tekstowy z charakterystycznym rozszerzeniem `.vhd`. Po kompilacji, symulacji i syntezie układowej otrzymuje się opis projektu gotowy do realizacji praktycznej. Do realizacji tych trzech czynności stosuje się odpowiednio odrębne programy, które tworzą *system projektowy.*

Bardziej szczegółowy opis czynności wchodzących w skład projektowania przy stosowaniu języka VHDL przedstawia rys. 1.2. Przed rozpoczęciem tworzenia opisu najpierw tworzy się hierarchię projektu lub odpowiedni schemat blokowy, co umożliwia równoległą pracę projektantów nad różnymi blokami. Następnie tworzy się w języku VHDL opis bloków składowych i ich połączeń.

Podczas kompilacji sprawdzana jest poprawność opisu pod kątem składni języka VHDL i zgodność z możliwościami programu użytego do kompilacji. Jeśli wynik jest poprawny, to przechodzi się do realizacji głównych celów projektowania, którymi są *symulacja* i *synteza.*

Do sprawdzenia poprawności działania projektowanego układu stosuje się programy testowe (*test bench*) oraz symulację *funkcjonalną* z użyciem wprowadzonych w opisie parametrów czasowych. Synteza służy do przetworzenia kodu wynikowego z kompilatora na opis

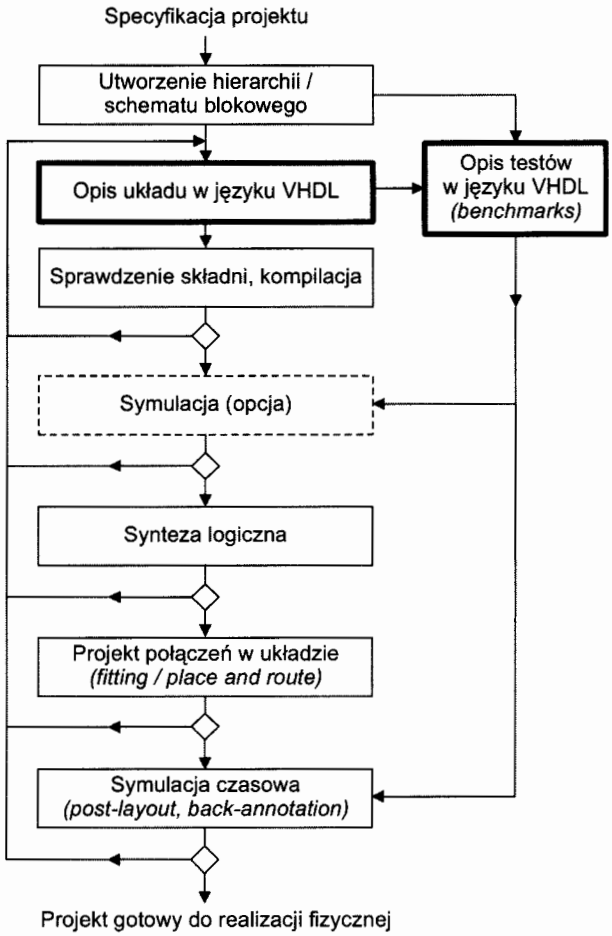


Rys.1.1 Proces projektowania układu/systemu cyfrowego

połączeń sieci elementów (np. bramek, multiplekserów i przerzutników) przyjętych do realizacji fizycznej. Opis ten jest zazwyczaj tworzony przez program w postaci pliku danych w znormalizowanym formacie EDIF (*Electronic Design Interchange Format*). Format EDIF jest ogólnie przyjęty do opisu połączeń w układach elektronicznych i jest objęty normą amerykańską EIA-682 oraz normą międzynarodową IEC 61690.

Następnym krokiem jest odwzorowanie (*mapping*) projektu logicznego do postaci odpowiedniej dla wybranej technologii układu fizycznego. Program o nazwie „fiter" (*fitter*) „wpasowuje" projekt logiczny w układ fizyczny, czyli określa topografię rozkładu i połączeń elementów układowych.

Ponieważ system projektowy z reguły zawiera biblioteki modeli dynamicznych dla układów danej firmy, zatem otrzymany opis projektu fizycznego pozwala na oszacowanie już w miarę dokładnie parametrów czasowych projektowanego układu (*back-annotated, post-layout*). Można je odczytać z raportu końcowego, tworzonego przez system projektowy po zakończeniu procesu projektowania. Na przykład w raporcie mogą być podane typowe czasy propagacji między końcówkami układu, jak również czasy propagacji dla najgorszego przypadku.

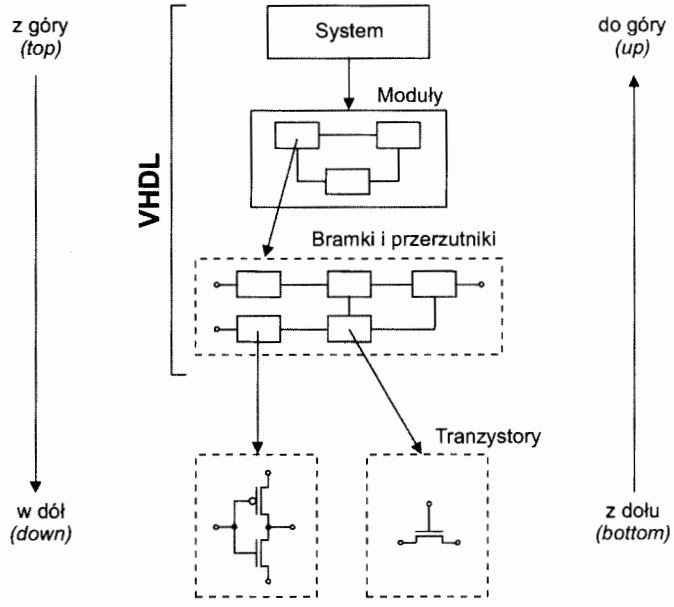


Rys. 1.2 Projektowanie w środowisku VHDL

Uzyskane po syntezie parametry mogą być również wykorzystane do finalnej symulacji *czasowej*.

Podczas procesu projektowania stale wykonuje się różne testy weryfikacyjne. Jeśli wynik jest negatywny, to zazwyczaj trzeba wracać do opisu VHDL, aby tam wprowadzić niezbędne ulepszenia.

Tworząc hierarchię projektu (rys. 1.3) można wybrać jeden z dwu trybów projektowania. Tryb „z góry w dół" (*top-down*) rozpoczyna się od dekompozycji systemu na mniejsze bloki (moduły), te z kolei na jeszcze mniejsze (np. bramki i przerzutniki), a te nawet na sieci tranzystorowe, reprezentujące poziom *fizyczny* (nie uwzględniany w języku VHDL).



Rys. 1.3 Hierarchia w procesie projektowym

Tryb „z dołu do góry" (*bottom-up*) jest odwrotny. W praktyce te dwa tryby są często stosowane łącznie, gdyż z jednej strony dekompozycja systemu na mniejsze bloki jest zazwyczaj konieczna, a z drugiej strony często tworzy się moduły „optymalne" z dostępnych elementów bibliotecznych, na przykład bramek i przerzutników.

2

PODSTAWY OPISU UKŁADÓW CYFROWYCH W JĘZYKU VHDL

Józef Kalisz

Opis w języku VHDL projektowanego układu jest tworzony w postaci pliku tekstowego, zawierającego zbiór modułów o zdefiniowanych *portach* wejściowych i wyjściowych, tworzących sprzęgi do łączenia modułów w większe struktury. Działanie każdego modułu jest opisywane wewnątrz modułu i nie jest widoczne dla innych modułów projektu.

Opis w języku VHDL tworzy się przy użyciu edytora tekstowego i zazwyczaj stosując nieproporcjonalną czcionkę `Courier`. Każda litera o tej czcionce zajmuje dokładnie taką samą szerokość w zapisie, dzięki czemu można precyzyjnie dopasowywać w pionie wcięcia tekstu. Ułatwia to czytelność złożonych opisów. Komentarze zapisuje się po dwóch kreskach (--), tylko do końca wiersza. Dłuższy komentarz wymaga wprowadzenia takich kresek na początku każdego następnego wiersza lub wierszy. Programy komputerowe do projektowania w języku VHDL zazwyczaj nie umożliwiają wyświetlania polskich znaków diakrytycznych. Dlatego polskojęzyczne komentarze w przykładach są pisane bez „polskich liter".

W języku VHDL nie odróżnia się liter małych od dużych (za wyjątkiem znaków literowych i łańcuchów znakowych), ani kroju czcionki. Aby zapewnić lepszą czytelność programów w języku VHDL stosuje się wyróżnienie *słów kluczowych*, ujętych w standardzie języka, poprzez **wytłuszczenie** czcionek. Innym sposobem wyróżnienia słów kluczowych jest zastosowanie DUŻYCH LITER. W edytorach systemów projektowych stosuje się wyróżnienie kolorem czcionki.

Poniższy przykład przedstawia najprostszy opis w języku VHDL trzywejściowej bramki NAND (rys. 2.1).

```
-- L2.1  (umowne oznaczenie numeru przykladu)
entity nand3 is                -- deklaracja jednostki
port(a,b,c   : in bit;         -- opis wejsc
          y  : out bit);       -- opis wyjscia
end nand3;

architecture a1 of nand3 is    -- deklaracja archit.
begin
   y <= not(a and b and c);    -- cialo architektury
end a1;
```



Rys. 2.1 Opis portów przykładowej bramki NAND

Pokazany opis nosi nazwę *jednostki projektowej* (*design entity*). Nazwa ta może się odnosić do modułu elementarnego (bramki), ale również do bardziej złożonych bloków funkcjonalnych. Jednostka projektowa, zwana również *modułem*, zawiera *deklarację jednostki* (**entity**) oraz *deklarację i opis* (*body* – „ciało") *architektury* (**architecture**). Składnia jednostki projektowej jest następująca:

```
[klauzule uzycia bibliotek i pakietów]
entity nazwa_jednostki is -- deklaracja jednostki
   [generic(lista_deklarowanych_stalych);]
   [port  (lista_sygnalow_wej. in | inout nazwa_typu;
   lista_sygnalow_wyj. out | buffer | inout
   nazwa_typu);]
end nazwa_jednostki;

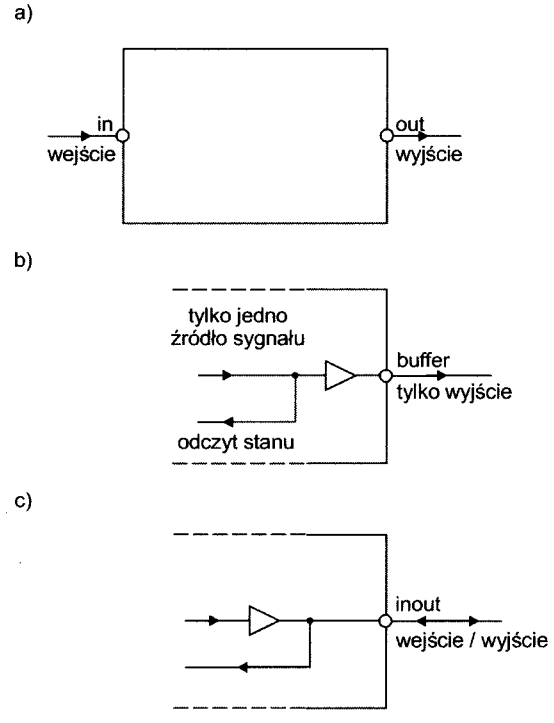
architecture nazwa_architektury of nazwa_jednostki is
[czesc_deklaracyjna: typy, sygnaly_wewnetrzne, stale]
begin
   {instrukcja_wspolbiezna;}
end nazwa_architektury;
```

Przy opisie składni stosuje się pewne charakterystyczne znaki. W nawiasach kwadratowych [ ] zawarte są pojedyncze elementy opcjonalne, a w nawiasach klamrowych { } powtarzalne elementy opcjonalne. Pionowa kreska | oznacza tu alternatywę („albo-albo").

Klauzule **port** i **generic** stanowią *nagłówek* jednostki. Klauzula **port** służy do opisu sprzęgu danej jednostki projektowej z otoczeniem. Na związanej z klauzulą liście podaje się

- nazwy *portów*, które są równoznaczne z nazwami *końcówek* opisywanego układu i nazwami odpowiednich *sygnałów* na końcówkach;
- rodzaje portów: wejściowych – **in**, wyjściowych – **out** i **buffer**, dwukierunkowych – **inout**;
- nazwy typów dla wymienionych sygnałów.

W podanym wyżej przykładzie zadeklarowano porty `a,b,c` i `y`, ich rodzaje **in** i **out** oraz typ `bit`, którego wartościami są dwa znaki numeryczne `'0'` i `'1'` (a nie liczby 0 i 1!). Pokazany na rys. 2.1 symbol prostokąta („czarnej skrzynki") z odpowiednio oznaczonymi końcówkami stanowi odpowiednik graficzny deklaracji jednostki.



Rys. 2.2 Rodzaje portów: a) wejściowy – **in** i wyjściowy – **out**, b) wyjściowy z odczytem – **buffer**, c) dwukierunkowy – **inout**

Rysunek 2.2 ilustruje główne rodzaje portów stosowanych w języku VHDL. Porty **in** i **out** są jednokierunkowe, a port **inout** jest dwukierunkowy. Port wyjściowy **buffer** dopuszcza do odczytu tylko wewnętrzne źródło sygnału, a na zewnątrz może być użyty tylko do odczytu na przewodzie, na którym nie występują inne źródła sygnałów. Port **inout** nie wprowadza takich ograniczeń, ale aby go wykorzystać jako port wejściowy dla sygnału zewnętrznego, należy sygnał wewnętrzny doprowadzać do tego portu poprzez wyjście trójstanowe. Aby odczytać stan sygnału wewnętrznego, w czasie odczytu do portu nie mogą być doprowadzane sygnały zewnętrzne o wartościach mogących zmienić ten stan.

Stałe wprowadzane na liście podanej w klauzuli **generic** najczęściej wykorzystuje się do łatwej specyfikacji liczby portów wejściowych lub/i wyjściowych, liczby komponentów w bloku lub wielkości opóźnień czasowych dla stosowanych elementów logicznych. Jest to wygodny sposób deklarowania takich stałych, który umożliwia tworzenie modeli VHDL o charakterze uniwersalnym, np. sumatorów lub rejestrów o dowolnej liczbie bitów, układów logicznych o różnych opóźnieniach itp. Odpowiednie przykłady podane są w dalszej części tekstu.

Architektura opisuje sposób działania danej jednostki projektowej. Odpowiednikiem graficznym opisu architektury może być *schemat* bloku zdefiniowanego końcówkowo przez deklarację jednostki.

W podanym przykładzie jest to proste wyrażenie opisujące funktor NAND z użyciem operatorów boolowskich. Symbol lewostronnej strzałki (`<=`) oznacza operację *przypisania* wartości otrzymanej z *wyrażenia* po prawej stronie do *sygnału* o nazwie podanej po lewej stronie. Trzeba zaznaczyć, że ten sam symbol jest również stosowany jako operator relacji „mniejsze lub równe" (p. 3.2.1), lecz różne konteksty stosowania wykluczają możliwość powstania pomyłek.

Porty jednostki projektowej mogą być opisane przez jedną deklarację jednostki, ale działanie jednostki przez więcej niż jedną architekturę. Różne architektury dla danej jednostki można opisywać jedna po drugiej w dowolnej kolejności, a wyróżnia się je swobodnie wybieranymi nazwami, np. `a1`, `a2` i `a3` albo `arch1` i `arch2` itp. Do wyboru określonej architektury do danego projektu stosuje się deklarację *konfiguracji* (**configuration**). Przy braku tej deklaracji domyślnie stosowana jest ostatnia opisana architektura.

Instrukcje wypisane w opisie architektury między słowami **begin** i **end** `nazwa_architektury` są wykonywane *współbieżnie* i tworzą *obszar współbieżny*, określany jako *ciało architektury*.

Współbieżność jest bardzo istotną właściwością opisu układów cyfrowych w języku VHDL. Podczas symulacji jakakolwiek zmiana sygnału w instrukcjach współbieżnych powoduje ich wykonanie w taki sposób, że *ich wyniki mogą być przypisane do tego samego momentu czasu*. Kolejność wypisania instrukcji współbieżnych nie ma znaczenia. Współbieżność nie ma związku z procesem kompilacji programu napisanego w języku VHDL, co w komputerze realizowane jest sekwencyjnie.

Symulacja realizowana jest w dyskretnych krokach na skali czasu. Aby zmniejszyć obciążenie systemu, niezbędne obliczenia wykonywane są tylko dla takich momentów czasu i tylko dla takich instrukcji współbieżnych, gdzie występują zmiany stanu sygnałów będących argumentami tych instrukcji. Takie zmiany sygnałów (wraz z przyporządkowanymi im momentami czasu) określa się jako *zdarzenia* (*event*). Każdy krok symulacji realizowany jest dwuetapowo. Najpierw przygotowuje się *transakcje*, czyli stany sygnałów będących argumentami instrukcji współbieżnych, odpowiednio do wyników poprzedniego kroku symulacji. Jeśli nastąpiła zmiana sygnału, to transakcja staje się zdarzeniem. Następnie wykonywane są tylko te instrukcje, w których wystąpiły zdarzenia. Otrzymane wyniki są przypisywane do odpowiednio opóźnionych momentów czasu. W ten sposób tworzone są nowe transakcje w postaci *planu transakcji* i ustalany jest moment czasu dla następnego kroku symulacji.

Wykonaniu każdej instrukcji współbieżnej system przydaje wirtualny odcinek czasu równy zeru w symulowanym czasie i zwany czasem *delta*. W obrębie tego czasu nie mogą wystąpić żadne zmiany sygnałów, co jest warunkiem prawidłowego działania symulatora.

Na przykład, można utworzyć architekturę równoległego układu kombinacyjnego o trzech wejściach `a,b,c` i czterech wyjściach `y1,y2,y3,y4` następująco:

```
architecture ar1 of uklad is
begin
   y1 <= a xor c;
   y2 <= b nand c;
   y3 <= c nor a;
   y4 <= a and b;
end ar1;
```

Tak opisane bramki mają zerowe opóźnienia, czyli podczas symulacji przy zmianie sygnału `a` nastąpią jednocześnie zmiany sygnałów `y1`, `y3` i `y4`. W istocie następują one po czasie *delta*.

Na przykład, dla sygnału y1 w powyższej architekturze oddziaływanie opóźnienia *delta* (Δ) można zilustrować następującą sekwencją czynności w kolejnych krokach symulacji:

Czas	a	c	y1	Komentarz
0	'1'	'1'	'0'	umowny stan początkowy
1	'0'	'1'	'0'	zdarzenie (zmiana a z '1' na '0')
1 + Δ	'0'	'1'	'1'	przypisanie po czasie Δ, transakcja y1

Suma opóźnień *delta* jest także równa zeru. Na przykład, jeśli w ciele architektury są instrukcje współbieżne wymagające kolejnych iteracji:

```
y1 <= a xor c;        -- te instrukcje mozna wypisac
y5 <= y1 nand b;      -- w odwrotnej kolejnosci!
```

to zmiana sygnału y5 zostanie przypisana po czasie 2Δ = 0:

Czas	a	b	c	y1	y5
0	'1'	'1'	'1'	'0'	'1'
1	'0'	'1'	'1'	'0'	'1'
1 + Δ	'0'	'1'	'1'	'1'	'1'
1 + 2Δ	'0'	'1'	'1'	'1'	'0'

W ten sposób otrzymuje się *cykl delta*, który jest równy zeru w symulowanym czasie.

Opóźnienia występujące w układach cyfrowych można opisywać wprowadzając opóźnienia przypisania do sygnału, co wykonuje się przy użyciu klauzuli **after**. Na przykład, jeśli zapiszemy

```
y1 <= a xor c after 5 ns;
y5 <= y1 nand b after 5 ns;
```

to po zmianie a zmiana y1 wystąpi po czasie 5 ns + Δ, a zmiana y5 po czasie 10 ns + 2 Δ. Opóźnienia wpisane po słowie **after** będą widoczne podczas symulacji, ale nie będą uwzględniane przy syntezie układu. Oznacza to, że są one „niesyntezowalne".

Zasada współbieżności może czasem stwarzać pewne problemy. Na przykład, można opisać działanie multipleksera *2 na 1* modułem

```
-- L2.2
entity mux2_1 is
   port (a,b,s  :  in bit;
               y  :  out bit);
end mux2_1;
```

```
architecture a1 of mux2_1 is
begin
   y <= a and s;
   y <= b and not s;
end a1;
```

Zależnie od wartości sygnału s na wyjściu pojawi się wartość sygnału a lub b. Jednakże przy zmianie sygnału s obydwie instrukcje w ciele architektury będą wykonywane współbieżnie, co oznacza, że sygnałowi y mogłyby być jednocześnie przypisane przeciwstawne wartości '0' i '1'. Mówimy wtedy, że sygnał y jest jednocześnie sterowany przez dwa *sterowniki*, co może prowadzić do niejednoznaczności wyniku. Problem można rozwiązać przez odrębne zdefiniowanie *funkcji rozdzielczej*, lecz prostszym rozwiązaniem jest wprowadzenie dwu dodatkowych sygnałów wewnętrznych:

```
-- L2.3
architecture a2 of mux2_1 is
   signal p,r:  bit;     --  sygnaly wewnetrzne
begin
   p <= a and s;
   r <= b and not s;
   y <= p or r;
end a2;
```

Teraz sygnał y ma już tylko jeden sterownik. Najprostszym sposobem zapewnienia pojedynczego sterownika jest użycie pojedynczej, współbieżnej instrukcji przypisania warunkowego **when-else**:

```
-- L2.4
architecture a3 of mux2_1 is
begin
   y <= a when s = '1' else
        b when s = '0';    -- trzeba wypisac wszystkie
end a3;                    -- warunki!
```

Warunki wypisane w instrukcji **when-else** są badane *kolejno* tak, że najwyższy priorytet ma pierwszy warunek. Jeśli jest on spełniony, to dalsze nie są już badane.

Ogólnie biorąc, układ lub system cyfrowy może być opisany w języku VHDL przez architekturę zawierającą ciąg instrukcji współbieżnych. Użyte wyżej przykładowe instrukcje współbieżne przypisania do sygnału mogą być stosowane w kilku odmianach. Każda taka instruk-

cja reprezentuje tylko *pojedynczą operację*. W praktyce często jednak istnieje potrzeba opisu *ciągu operacji* wykonywanych kolejno, czyli sekwencyjnie. W szczególności dotyczy to układów sekwencyjnych.

Sekwencyjność można opisać za pomocą współbieżnej instrukcji *procesu*. Podobnie jak architektura, proces zawiera „ciało" zawarte między słowami **begin** i **end**. W odróżnieniu od instrukcji współbieżnych, instrukcje zawarte w ciele procesu są instrukcjami *sekwencyjnymi*, które są wykonywane *kolejno*. Proces stanowi jednak instrukcję współbieżną i jest stosowany w opisie architektury na równi z innymi instrukcjami współbieżnymi, w tym również z innymi procesami.

W szczególności proces może zawierać ciąg instrukcji podstawowego przypisania do sygnału. Instrukcje te, zależnie od zastosowania, są współbieżne (w ciele architektury) lub sekwencyjne (w ciele procesu). Proces zawierający tylko jedną instrukcję podstawowego przypisania do sygnału jest równoważny tej instrukcji wypisanej bezpośrednio w ciele architektury.

Poniższy moduł opisuje przykładowy przerzutnik T, wyzwalany narastającym zboczem zegara C, z asynchronicznym wejściem zerującym R.

```
-- L2.5
entity fft is
   port (T,C,R   :   in bit;
               Q   :   out bit);
end fft;

architecture a1 of fft is
begin
   process (C,R)     -- (C,R): lista wrazliwosci
      variable tq  :  bit;    -- czesc deklaracyjna,
                      -- wprowadzenie zmiennej lokalnej
   begin              -- poczatek opisu ciala procesu
      if  R = '1' then tq := '0';   -- zerowanie
         elsif C'event and C = '1' and T = '1' then
-- jesli jest narastajace zbocze zegara i T = '1', to
         tq := not tq;     -- zmiana stanu
      end if;
      Q <= tq;          -- przypisanie do sygnalu Q
   end process;         -- koniec procesu
end a1;
```

Instrukcja procesu o składni podanej w p. 3.5 wymusza sekwencyjność wykonywania instrukcji zawartych w ciele procesu, czyli wypisa-

nych od słowa **begin** do słów **end process**. Pokazany proces zawiera dwie instrukcje, wykonywane kolejno.

Przy opisie układów kombinacyjnych trzeba na liście wrażliwości wymienić *wszystkie* sygnały wejściowe użyte w danym procesie.

Podczas symulacji proces jest wykonywany przez komputer w sposób ciągły od początku do końca i znowu od początku itd. („w kółko"). Zatrzymywanie tego obiegu ma miejsce w dwu przypadkach.

Pierwszy przypadek ma miejsce, gdy jest użyta opcjonalna lista wrażliwości. W powyższym przykładzie lista `(C,R)` zawiera te sygnały, które powodują zmiany stanu modelowanego układu. Użycie tej listy oznacza, że proces jest wykonywany do końca, a następnie czeka na zmianę stanu któregokolwiek z sygnałów wymienionych na liście wrażliwości. Po wykryciu takiej zmiany proces jest wykonywany od początku i znowu czeka na końcu.

Drugi przypadek ma miejsce, gdy w obrębie procesu użyta jest sekwencyjna instrukcja czekania (**wait**). Wtedy jednak nie można używać listy wrażliwości związanej z instrukcją **process**.

Zmienna (**variable**) lokalna `tq` została wprowadzona w celu pamiętania „obecnego" stanu przerzutnika. W przerzutnikach T i JK stan następny zależy również od stanu obecnego, który musi być znany przed każdym kolejnym impulsem zegarowym. Nie można jednak – przy definicji portu `Q : out bit` – napisać legalnego wyrażenia o postaci `Q <= not Q`. Port wyjściowy **out** nie umożliwia bowiem odczytu sygnału wyjściowego do wykorzystania w ciele architektury. Podobnie port wejściowy **in** nie umożliwia zapisu (przypisania) wartości sygnału.

Zmienne mogą być deklarowane tylko w części deklaracyjnej procesu lub podprogramu i nie mogą służyć do komunikacji między procesami. Zmienne uzyskują wartości w rezultacie operacji *przypisania do zmiennej* (symbol `:=`). Wartości zmiennych użyte w procesie są pamiętane między kolejnymi krokami symulacji, a więc mogą być użyte do pamiętania stanu obecnego przy opisie przerzutników. Zakres i/lub wartość początkową zmiennej można wprowadzić w jej deklaracji, na przykład

```
variable aa : integer range 5 to 200 := 24;
variable bb : integer;
```

Jeśli nie zdefiniuje się wartości początkowej, to domyślnie staje się ona równa najmniejszej (lub lewostronnej) wartości dla danego typu.

Aby wykryć moment pojawienia się narastającego zbocza sygnału zegara wykorzystuje się wyrażenie **if** `(C'event and C = '1')`,

gdzie zamiast litery C można wprowadzić dowolny inny symbol sygnału zegarowego, zadeklarowany w opisie jednostki. Zapis 'event stanowi *atrybut*, który służy do wykrycia *zdarzenia*, czyli zmiany stanu sygnału związanego z atrybutem. Wynikiem wyrażenia C'event jest wartość boolowska true (jeśli wykryte jest zdarzenie) lub false (w przeciwnym razie). Do wykrywania zbocza opadającego należy użyć zapisu C = '0'.

Inny sposób pamiętania stanu obecnego wykorzystano w nieco zmodyfikowanej architekturze przerzutnika T:

```
-- L2.5_a2
architecture a2 of fft is
   signal  tp   :  std_logic;  -- sygnal wewnetrzny
begin
   process (C,R)
   begin
      if  R = '1' then tp <= '0';
         elsif C'event and C = '1' then
         tp <= tp xor T;
      end if;
   end process;
   Q <= tp;
end a2;
```

W architekturze a2 nie użyto zmiennej lokalnej wewnątrz procesu, natomiast poza procesem wprowadzono *sygnał wewnętrzny* tp (w części deklaracyjnej architektury). Pokazane przypisanie Q <= tp stanowi instrukcję współbieżną z procesem.

Sygnały wewnętrzne nie mogą być deklarowane wewnątrz procesu (czyli w jego części deklaracyjnej). Wszystkie sygnały wewnętrzne mają predefiniowany rodzaj **inout**, czyli umożliwiają zarówno zapis jak i odczyt ich wartości. Jednym z typowych zastosowań sygnałów wewnętrznych jest modelowanie połączeń wewnętrznych przy opisie bardziej złożonych układów (p. 4.2).

Do odczytu stanu obecnego przerzutnika poprawne jest zarówno pokazane użycie zmiennej lokalnej jak i użycie sygnału wewnętrznego. Można zauważyć, że przypisanie do sygnału wewnątrz procesu jest instrukcją sekwencyjną i musi być wykonywane zawsze podczas symulacji (nawet gdy nie następuje zmiana stanu przerzutnika), natomiast przypisanie do sygnału poza procesem jest instrukcją współbieżną, która jest wykonywana tylko wtedy, gdy następuje zmiana stanu sygnału zegarowego lub zerującego przerzutnika.

Jeśli istnieje alternatywa wyboru zmiennej lokalnej lub sygnału wewnętrznego, to zaleca się stosowanie zmiennej lokalnej. Wynika to

stąd, że zmienna lokalna wymaga użycia znacznie mniej zasobów pamięciowych w symulatorze niż sygnał lokalny. Ponadto podczas symulacji przypisanie do zmiennej jest wykonywane natychmiastowo, podczas gdy przypisanie do sygnału jest wykonywane dwuetapowo, podobnie jak wykonanie instrukcji współbieżnej. Nieuwzględnienie tego mechanizmu może prowadzić do błędów. Na przykład, można wypisać ciąg instrukcji sekwencyjnych

```
Q <= '0';
.........
if Q = '0' then ... -- warunek nie bedzie spelniony!
```

albo

```
Q <= D;
P <= Q;   --   P <= D?
```

i w obydwu przypadkach można „intuicyjnie" oczekiwać sekwencyjnej realizacji tych instrukcji, zgodnie z zapisem. Tak jednak nie będzie. Podobnie jak wykonanie instrukcji współbieżnych, również wykonanie instrukcji sekwencyjnych przebiega dwuetapowo. Najpierw kolejno we wszystkich instrukcjach procesu przygotowuje się transakcje, czyli stany „obecne" sygnałów będących argumentami instrukcji. Stany te są wynikiem poprzedniego kroku symulacji. Następnie proces się zatrzymuje i dopiero wtedy następuje w czasie *delta* faktyczne wykonanie instrukcji dla tak przygotowanych danych.

W pierwszym z podanych przykładów warunek `Q = '0'` nie będzie zatem spełniony w bieżącym kroku symulacji, lecz dopiero w następnym. Może to odpowiadać opóźnieniu jednego okresu zegara. Analogicznie w drugim przykładzie, równość `P = D` będzie uzyskana dopiero w wyniku dwu kroków symulacji (okresów zegara). Dlatego też w architekturze `L3.5_a2` wprowadzono przypisanie `Q <= tp` poza procesem.

Stan obecny w przerzutnikach T i JK można również wykorzystać do ich opisu bezpośrednio, to znaczy bez użycia zmiennej lokalnej lub sygnału wewnętrznego. W tym celu zamiast portu **out** trzeba zadeklarować port **buffer** lub **inout**, który umożliwia odczyt stanu. Ilustruje to kolejny przykład opisu przerzutnika T:

```
-- L2.6
entity fft1 is
   port (T,C,R   :   in bit;
         Q       :   buffer bit);
end fft1;
```

```
architecture a1 of fft1 is
begin
   process (C,R)
   begin
      if  R = '1' then Q <= '0';
         elsif C'event and C = '1' and T = '1' then
         Q <= not Q;  --  odczyt i negacja stanu Q
      end if;
   end process;
end a1;
```

Kolejnym przykładem stosowania instrukcji procesu jest opis przerzutnika D:

```
-- L2.7
entity ffd is
   port (D,C,R : in bit;
               Q : out bit);
end ffd;

architecture a1 of ffd is
begin
   process (C,R)
   begin
      if  R = '1' then Q <= '0' ;
         elsif (C'event and C = '1') then Q <= D;
      end if;
   end process;
end a1;
```

W tym przykładzie nie użyto ani zmiennej lokalnej, ani sygnału wewnętrznego, ani wyjścia **buffer** lub **inout**, gdyż w opisie przerzutnika D nie ma potrzeby odczytu stanu obecnego.

Przedstawione modele podstawowych układów i bloków logicznych, takich jak bramki, przerzutniki i multipleksery, zalicza się do kategorii „prymitywów" (*primitives*). Aby nie trzeba było tworzyć takiego opisu za każdym razem, gdy jest on potrzebny, w systemach projektowych stosuje się zazwyczaj *pakiety* firmowe, zawierające zbiory najczęściej stosowanych prymitywów o postaci modułów, funkcji i procedur. Pakiety takie mogą być również tworzone indywidualnie. Jak będzie to opisane w p. 3.4, wystarczy wpisać na początku modułu klauzulę do używania określonego pakietu, aby móc korzystać z jego zawartości bez potrzeby wprowadzania odrębnych opisów dla prymitywów używanych w projekcie.

W niniejszym rozdziale poprawność działania modelowanych układów była oceniana przez śledzenie procesu ich symulacji. W zasadzie model w języku VHDL, który jest poprawnie interpretowany przez symulator, powinien być również akceptowany przez program służący do syntezy. To jednak nie zawsze ma miejsce i istnieją formalnie poprawne lecz „niesyntezowalne" modele układów cyfrowych w języku VHDL, których przykłady będą podane w następnym rozdziale.

Jak wynika z podanych poprzednio przykładów, działanie układu można opisać różnymi sposobami (czyli różnymi architekturami). Aby sklasyfikować te sposoby, zdefiniowane zostały pewne „style" opisu architektury. Nie są one znormalizowane, lecz w praktyce wyróżnia się następujące **style architektury**:

- behawioralny (*behavioral*),
- przepływowy (*dataflow*),
- strukturalny (*structural*).

Styl **behawioralny** stosuje się do opisów algorytmicznych, z użyciem instrukcji procesu i instrukcji *sekwencyjnych* (p. 3.8). Dlatego styl ten określa się również jako **sekwencyjny**. W ten sposób opisuje się sekwencje stanów, układy cyfrowe z pamięcią - automaty, programy testujące oraz podprogramy. Przykładami takiego stylu są pokazane poprzednio proste opisy przerzutników (`L2.5`, `L2.6` i `L2.7`).

Styl **przepływowy** stosuje się do opisu przepływu danych podczas ich przetwarzania przez układy *kombinacyjne*, takie jak sieci bramkowe, multipleksery i układy arytmetyczne. Do opisu stosuje się instrukcje *współbieżne* (p. 3.5). Przykłady takich opisów podano wcześniej (`L2.1` – `L2.4`).

Styl przepływowy stosuje się również do opisu układów *sekwencyjnych*, w szczególności układów zawierających rejestry (zespoły przerzutników), które są opisywane przez odpowiednie procedury. W tych przypadkach styl przepływowy często określa się akronimem **RTL** (*Register Transfer Logic* lub *RT Level* lub *RT Language*). Przykłady takich opisów podane są w rozdziale 4.

Styl **strukturalny** stosuje się do opisu układu utworzonego z pewnej liczby układów składowych, których opisy VHDL przygotowano wcześniej. Jest to w istocie opis połączeń układów składowych (komponentów). Styl ten będzie omówiony w rozdziale 4.

Styl przepływowy i strukturalny są naturalnymi sposobami opisu układów cyfrowych, zwłaszcza dla projektantów operujących typowymi

modułami układowymi (jak multiplekser, sumator, licznik itp.) przy projektowaniu bardziej złożonych bloków. Prowadzi to jednak często do długich i złożonych opisów.

Zalety języka VHDL w pełni wykorzystuje się stosując styl behawioralny, który zazwyczaj umożliwia tworzenie opisów znacznie krótszych i łatwiejszych do interpretacji przez człowieka. Wtedy główny ciężar interpretacji opisu i syntezy układu spada na użyty system projektowy. Przy trudniejszych projektach jakość wyniku (w szczególności złożoność układowa) istotnie zależy od możliwości użytego systemu projektowego.

3

JĘZYK VHDL

Józef Kalisz

W niniejszym rozdziale będą omówione najistotniejsze elementy semantyki i składni języka VHDL. Ścisłe przestrzeganie tych zasad jest niezbędne do tworzenia poprawnych (to znaczy akceptowanych przez kompilator) opisów układów cyfrowych w języku VHDL.

3.1 OBIEKTY

Do zapisu i pamiętania danych w języku VHDL służą *obiekty*. Podstawowymi obiektami języka VHDL są *sygnały, zmienne, stałe* i *pliki*. Podczas deklarowania obiektów muszą być także zdefiniowane *typy*, które służą do wyrażania wartości obiektów. Jeśli początkowe wartości obiektów nie są odrębnie deklarowane, to są one przyjmowane domyślnie jako najmniejsze z zakresu danego typu, a dla typów wyliczeniowych jako pierwsza wartość z listy.

Sygnały są funkcjami czasu, co oznacza że wartości sygnałów są przyporządkowane wartościom na osi czasu podczas symulacji działania opisywanych układów. W części deklaracyjnej jednostki wprowadzane są nazwy sygnałów *zewnętrznych*. Wartości sygnałów *wejściowych* są wprowadzane do wykorzystania w modułach przez *porty wejściowe* w sposób opisany poprzednio. Wartości sygnałów *wyjściowych* są uzyskiwane z *portów wyjściowych*, w rezultacie *operacji przypisania* (<=), na przykład `y <= '1'`. W części deklaracyjnej

architektury mogą być także wprowadzane sygnały *wewnętrzne*. Składnia deklaracji sygnału jest następująca:

```
signal nazwa_sygnalu : nazwa_typu [ograniczenie][:=
wyrazenie];
```

Opcjonalne `ograniczenie` umożliwia zawężenie zakresu predefiniowanego dla danego typu. Opisuje się je podając granice nowego zakresu (**range**). Opcjonalne `wyrazenie` służy do *inicjalizacji* wartości sygnału, która zastępuje wartość domyślną (bez inicjalizacji) dla danego typu. Przykłady:

```
signal zegar : bit; -- domyslna wartosc poczatkowa '0'
signal reset : bit := '1';  -- inicjalizacja '1'
signal x : integer; -- dom. wartosc pocz. - 2147483647
signal y  :  bit_vector(0 to 3) := ('0','1','1','0');
signal alfa : integer range 0 to 255;  -- ograniczenie
signal mi : bit_vector(9 downto 0) := (others => '1');
```

W ostatnim przykładzie, charakterystyczne wyrażenie o postaci (**others** => '1') umożliwia ustalenie wartości początkowej wszystkich elementów wektora `mi` jako równej '1'. Operator prawostronnej strzałki (=>) oznacza *przyporządkowanie*.

Zmienne nie są funkcjami czasu (podobnie jak w standardowych językach programowania) i mogą być stosowane pomocniczo *tylko w obrębie procesu lub podprogramu* do zapamiętywania wyników pośrednich.

Wartości zmiennych są uzyskiwane w rezultacie *operacji przypisania do zmiennej* (:=), na przykład `pp := x + 1`. Ta operacja nie jest przypisana do osi czasu podczas symulacji i jest wykonywana natychmiastowo. Składnia deklaracji zmiennej jest następująca:

```
variable nazwa_zmiennej : nazwa_typu [ograniczenie]
[:= wyrazenie];
```

gdzie ograniczenie i inicjalizacja są realizowane tak samo jak dla sygnałów. Na przykład

```
variable u  :  integer range 0 to 127 := 5;
variable pp  :  integer range 500 downto 5 := 100;
```

Stałe umożliwiają przypisanie nazwy wartościom stałym określonego typu. Służy do tego deklaracja **constant** o składni

```
constant nazwa_stalej  :  typ := wyrazenie;
```

Przykłady:

```
constant UCC  :  bit := '1';
constant  fi  :  integer := 12;
constant  vec :  bit_vector(7 downto 0) := "10010011";
```

Pliki służą do trwałego przechowywania danych, co wykorzystuje się głównie przy wykonywaniu testów. Standardowy pakiet `textio` z biblioteki standardowej `std` definiuje do operacji na plikach typ `text` oraz szereg procedur. Plik tworzy się instrukcją **`file`** o składni

```
file nazwa_pliku: text [open tryb] is
"lancuch_znakowy";
```

gdzie `tryb` może przyjmować wartości `read_mode` (odczyt), `write_mode` (zapis) lub `append_mode` (dodaj), a `lancuch_znakowy` stanowi logiczną nazwę pliku. Na przykład

```
file mplik: text [open write_mode] is "dane.dat";
```

3.2 TYPY

Obiekty operują na różnych *typach danych*, które określa się poprzez podanie *nazwy*, *zbioru wartości* i *zbioru operacji*. Typy dzieli się na następujące klasy:

- *typy skalarne*, takie jak liczby całkowite i zmiennoprzecinkowe, typy wyliczalne (definiowane przez wyliczenie ich wartości w zbiorze), oraz typy fizyczne;
- *typy złożone*, takie jak *tablice* (*array*) elementów tego samego typu skalarnego, np. *wektory*. Do typów złożonych zalicza się również *rekordy* (*records*), które zawierają elementy różnych typów, oraz *pliki* (*files*);
- *typy wektorowo-skalarne*: wektory zerojedynkowe, które do operacji arytmetycznych reprezentują liczby naturalne lub całkowite, zapisane w naturalnym kodzie dwójkowym.

Poniżej są przedstawione typy danych ujęte w normie IEEE Std 1076-2001. Ponadto ze stosownymi odnośnikami wymieniono typy objęte normami IEEE Std 1164-1993 i IEEE Std 1076.3-1997. Definicje typów są zawarte w odpowiednich pakietach standardowych.

3.2.1 TYPY SKALARNE

Istnieją trzy rodzaje typów skalarnych: *numeryczne*, *wyliczeniowe* i *fizyczne*, zależnie od rodzaju stosowanych elementów.

Liczby całkowite: `integer`

Zakres: `- 2147483647 to + 2147483647`,
czyli od $-(2^{31}-1)$ do $+(2^{31}-1)$
Operatory arytmetyczne: `+` (dodawanie), `-` (odejmowanie), `*` (mnożenie), `/` (dzielenie), `**` (potęgowanie), **abs**, **rem**, **mod**.
Operatory relacyjne: `=` (równe), `/=` (nierówne), `<` (mniejsze), `<=` (mniejsze lub równe), `>` (większe), `>=` (większe lub równe).
Przykłady: `a + b + c, (a*3 - 5)*b, 2**c`.

Operator **abs** (*absolute value*) zwraca wartość absolutną argumentu. Operator **rem** (*remainder*) zwraca resztę z dzielenia całkowitoliczbowego, to znaczy `A` **rem** `B` zwraca wynik `A - (A/B)*B`, gdzie `/` oznacza dzielenie całkowitoliczbowe (typ wyniku jest taki sam jak argumentów). Wynik ma znak argumentu `A` i wartość bezwzględną mniejszą od wartości bezwzględnej `B`. Na przykład `6` **rem** `4 = 2, (-6)` **rem** `4 = -2, 6` **rem** `(-4) = 2, (-6)` **rem** `(-4) = -2`.

Operator **mod** (*modulo*) jest stosowany w postaci `A` **mod** `B` i zwraca wynik `A - B*N`, gdzie `N` jest liczbą całkowitą tak dobraną, aby wynik miał znak argumentu `B` i wartość bezwzględna wyniku była mniejsza od wartości bezwzględnej `B`. Na przykład `6` **mod** `4 = 2` (czyli `N = +1`), `(-6)` **mod** `4 = 2` (czyli `N = -2`), `6` **mod** `(-4) = -2` (czyli `N = +2`), `(-6)` **mod** `(-4) = -2` (czyli `N = +1`). Operacja `A` **mod** `B` zwraca wynik o takiej samej wartości bezwzględnej jak operacja `A` **rem** `B`, lecz znak wyniku ma znak argumentu `B`.

Istnieją również **podtypy standardowe** dla liczb całkowitych, które są predefiniowane przy użyciu słów kluczowych **subtype** i **range** (zakres). Są to liczby naturalne i dodatnie:

```
subtype natural is integer range 0 to 2147483647;
subtype positive is integer range 1 to 2147483647;
```

Zapis dużych liczb całkowitych można nieco ułatwić, wprowadzając pomijane przez kompilator znaki podkreślenia (_), na przykład `31_215_754`.

Liczby rzeczywiste: `real`

Liczby rzeczywiste są zapisywane w 32-bitowym formacie zmiennoprzecinkowym, w którym jeden bajt opisuje wykładnik, a pozostałe trzy bajty mantysę. Maksymalny wykładnik jest równy $2^{127} \approx 1.7 \cdot 10^{38}$, a minimalny nieujemny $2^{-128} \approx 2.9 \cdot 10^{-39}$. Gwarantowany zakres wynosi `(- 1.0E+38 to + 1.0E+38)`. Precyzja jest określona przez mantysę ($2^{-23} \approx 1.19 \cdot 10^{-7}$), czyli wynosi co najmniej 6 cyfr dziesiętnych. Operacje i przykłady są analogiczne jak dla liczb całkowitych. Typ ten stosuje się bardzo rzadko, tylko do symulacji.

Przykład: `constant limit : real := 6.3;`

Typ bitowy: `bit`

Zakres: `('0', '1')`, będący zbiorem znaków numerycznych, a nie liczb!

Operatory logiczne: **not**, **and**, **or**, **nand**, **nor**, **xor**, **xnor** (operator unarny **not** ma wyższy priorytet od pozostałych).

Operatory relacyjne: `=, /=, <, <=, >, >=`.

Operacje relacyjne (predykaty) mają wyższy priorytet niż logiczne, a ich wynik jest typu boolowskiego (`boolean`).

Przykłady:

```
a and (b or c) -- funkcja logiczna, wynik typu bit
(not a or c) /= '1'  -- predykat, wynik typu boolean
```

Typy wielowartościowe: `std_ulogic i std_logic`

Wadą typu `bit` jest brak możliwości zapisu innych stanów poza standardowymi `'0'` i `'1'`, na przykład stanu wysokiej impedancji lub stanu dowolnego. Dlatego w normie IEEE Std 1164-1993 zdefiniowano dziewięciowartościowy typ `std_ulogic`, służący do reprezentowania wartości logicznej lub metalogicznej *na jednym przewodzie*:

```
type std_ulogic is   (
    'U',   -- stan niezainicjowany
    'X',   -- wymusza stan nieznany
    '0',   -- wymusza stan 0
    '1',   -- wymusza stan 1
    'Z',   -- stan wysokiej impedancji
    'W',   -- slaby stan nieznany (odczyt)
    'L',   -- slabe 0 (odczyt), rownowazne polaczeniu
           -- przez rezystor z masa)
    'H',   -- slabe 1 (odczyt), rownowazne polaczeniu
           -- przewodu przez rezystor
```

```
            -- z napieciem zasilania)
    '-'     -- stan dowolny (don't care));
```

Wartości 'U', 'X', 'W' i '-' są *metalogiczne* i służą do opisu modelu podczas symulacji, a nie do opisu działania realnego układu.

Typ std_ulogic jest „niejednoznaczny" (*unresolved*, stąd **u**logic) w tym sensie, że w przypadkach, gdy jeden przewód jest sterowany różnymi wartościami, wówczas wartość wynikowa nie jest ściśle określona. Typem „jednoznacznym" (*resolved*) jest dziewięciowartościowy typ std_logic, który jest zdefiniowany w pakiecie standardowym std_logic_1164. Są tam wprowadzone deklaracje funkcji resolved i podtypu std_logic w postaci:

```
function resolved ( s : std_ulogic_vector ) return
std_ulogic;
subtype std_logic is resolved std_ulogic;
   --  formalnie jest to podtyp!
```

jak również podana jest tablica, umożliwiającą jednoznaczne określenie wyniku:

```
constant resolution_table : stdlogic_table := (

-- U    X    0    1    Z    W    L    H    -         |   |
-------------------------------------------------------------
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
       );
```

oraz podany jest opis *funkcji rozdzielczej*

```
function resolved ( s : std_ulogic_vector ) return
std_ulogic is
   variable result : std_ulogic := 'Z';
begin
   if (s'length = 1) then return s(s'low); -- test
      else
        for i in s'range loop
            result := resolution_table(result, s(i));
        end loop;
```

```
        end if;
        return result;
    end resolved;
```

Jeśli jest tylko jeden stan sterujący, to funkcja zwraca jego wartość bez zmiany. Dominują stany „wymuszające" 'X', '0' i '1', a w żadnym przypadku wynikiem nie jest wartość '-'. Wartość 'Z' jest powszechnie stosowana do modelowania stanu wysokiej impedancji w buforach trójstanowych i podczas przesyłania sygnałów przez magistrale systemowe. Niektóre programy do syntezy nie akceptują wartości '-', lecz zazwyczaj wartości 'X' i '-' są traktowane jednakowo.

W normie IEEE Std 1164-1993 zdefiniowane są również użyteczne w praktyce **podtypy standardowe**

```
subtype X01 is resolved std_ulogic range 'X' to '1';
          --  czyli zbior wartosci ('X','0','1')
subtype X01Z is resolved std_ulogic range 'X' to 'Z';
          --  czyli zbior wartosci ('X','0','1','Z')
```

i analogiczne podtypy UX01 oraz UX01Z. Kolejność wyliczania znaków wynika z definicji typu ulogic, podanej poprzednio. W tej samej normie zdefiniowane są również funkcje standardowe do odpowiednich konwersji.

Typ boolowski: boolean

Zakres: (false, true).
Operatory: głównie = , /=, lecz dopuszcza się również inne operacje jak dla typu bitowego.
Przykłady: (a = b), (b /= c)

Typ znakowy: character

Zakres:
(...,'0','1','2',...,'A','B','C',...,'a','b','c',...),
czyli 128 znaków ze zbioru ASCII.
Operator: & (złożenie lub konkatenacja).
Przykład: jeśli a = "elektro" i b = "nika", to
a&b = "elektronika".
Uwaga: Norma IEEE Std 1076-2001 rozciąga zbiór znakowy na tablicę 256 znaków ujętych w normie ISO 8859 (wersja zachodnia).

Typ fizyczny czasowy: `time`

Zakres: `(0 to 2147483647)` jednostek:
`fs,ps,ns,us,ms,sec,min,hr.`
Przykład: `x <= not a and b after 12 ns;`

Typy wyliczeniowe są często stosowanymi typami definiowanymi przez projektanta. Są one definiowane przez listę *nazw* lub *znaków* wypisanych w odpowiedniej kolejności. Na przykład

```
type kolory is (niebieski, zielony, czerwony);
type trojkowy is (3,6,9,12,15);  --  nazwy
type duze_litery is ('L','H','W','X','Z');  -- znaki
```

W ostatnim przykładzie wartościami typu są znaki (literały), którymi są DUŻE litery. Gdybyśmy sformułowali typ, którego wartościami są nazwy o postaci pojedynczych dużych liter, na przykład

```
type dlitery is (L,H,W,X,Z);  -- nazwy
```

to typ ten byłby identyczny z typem

```
type mlitery is (l,h,w,x,z);
```

gdyż w języku VHDL małe i duże litery w nazwach i słowach kluczowych traktowane są jednakowo.

W praktyce, aby skrócić czas kompilacji i uzyskać po syntezie najprostsze struktury logiczne, należy celowo zawężać zakresy typów standardowych do faktycznie używanych podzakresów poprzez definiowanie *własnych podtypów*. Na przykład, można zdefiniować własny podzakres liczb całkowitych od 0 do 255 jako podtyp

```
subtype bajtowy is integer range 0 to 255;
```

albo podzakres małych liter w obrębie typu znakowego

```
subtype mlitery is character range 'a' to 'z';
```

Można również tworzyć *własne typy* liczb całkowitych przez podanie odpowiedniego zakresu, np.

```
type zakres256 is range 0 to 255;
type dekada is range 0 to 9;
```

Trzeba podkreślić, że operacje na dwu lub więcej argumentach (np. dodawanie) można realizować na danych *tylko tego samego typu*. Aby zatem można było dodawać wartości sygnałów wymienionych dwu typów, lepiej zdefiniować typ dekada jako podtyp

```
subtype dekada is zakres256 range 0 to 9;
```

Można także zawężać zakres bezpośrednio przy definiowaniu typu w deklaracji jednostki, na przykład

```
port (x_we : in integer range 0 to 15);
```

UWAGA: jeśli nie korzysta się z odpowiednio dobranego typu lub podtypu, ograniczającego zakres np. liczb całkowitych i przyjmuje się typ standardowy (integer), to synteza takiego modelu zajmie więcej czasu z uwagi na większe obciążenie obliczeniowe systemu projektowego. Otrzymany projekt może być również nieoptymalny w sensie większej złożoności logicznej. Oznacza to niepotrzebne zużycie większych zasobów logicznych w realnym układzie, np. programowalnym.

3.2.2 TYPY ZŁOŻONE

Typy złożone służą do zapisu zbiorów elementów określonego typu lub typów. Zbiory te są *indeksowane* w tym sensie, że każdy element zbioru ma przyporządkowaną liczbę (indeks), określającą *pozycję* elementu w zbiorze. Jeśli elementy zbioru mają tylko jeden typ, to zbiór taki stanowi *tablicę* (**array**). Jeśli natomiast elementy zbioru mają różne typy, to zbiór taki jest *rekordem* (**record**). Przykład:

```
type slowo16 is array(0 to 15) of bit;
   -- definicja slowa 16-bitowego
```

W nawiasach podaje się zakres i kierunek indeksowania. Można również definiować typy o zakresie *nieokreślonym*, używając słowa kluczowego **range** i symbolu „skrzynki" <>. Na przykład

```
type memory16 is array(natural range <>) of slowo16;
   -- typ tablicy pamieciowej ze slowami 16-bitowymi;
```

Aby jednak wykorzystać taki typ, to jego zakres trzeba sprecyzować podczas deklaracji obiektu lub subtypu. Na przykład

```
variable memo :  memory16(0 to 2**n-1);
   --  pamiec zawierajaca 2^n slow 16-bitowych
```

Predefiniowanymi typami złożonymi są typ *łańcuchowy* (`string`) oraz typ *bitwektorowy* (`bit_vector`). Pierwszy z nich definiuje łańcuchy znakowe, czyli słowa o deklarowanym formacie, np. **type** `string(1 to 4)` o przykładowej postaci `"adam"`, `"elka"`. Łańcuchy znakowe wymagają użycia cudzysłowów (").

Podobnie typ bitwektorowy definiuje wektory o elementach typu `bit`, na przykład

```
signal a  :  bit_vector(3 downto 0);
signal b  :  bit_vector(0 to 3);
```

Wartości typu `bit_vector` mogą być wyrażane w postaci wektora, np. `a = ('1','0','1','0')` lub w postaci równoważnego łańcucha, np. `b = "1010"`. Bit o oznaczeniu `a(0)` będzie skrajny z prawej strony, czyli równy `0`. W formacie sygnału `b` bit `b(0)` jest umieszczony z lewej strony, czyli `b(0) = 1`. Indeksowanie umożliwia różnorodne operacje na bitach. Na przykład, jeśli jest

```
signal c  :  bit_vector(0 to 3);
```

to można wykonać działanie

```
c <= b(1 to 2)&"11"
```

co w naszym przykładzie oznacza, że `c` przyjmie wartość równą `"0111"`.

Zaleca się stosowanie formatu wektorów o indeksowaniu malejącym (**downto**). W ten sposób wektory reprezentujące liczby dwójkowe mają najwyższy indeks na pozycji najbardziej znaczącego bitu (MSB – *Most Significant Bit*).

Niektóre kompilatory dopuszczają możliwość zapisu wartości wektorów jako liczb dwójkowych w kodzie szesnastkowym, co zaznacza się dużą literą `X` przed liczbą szesnastkową. Na przykład, można zadeklarować stałą `pik` w postaci

```
constant pik : bit_vector(7 downto 0) := X"5A";
          -- rownowazne zapisowi "01011010"
```

W normie IEEE Std 1164-1993 zdefiniowany jest typ złożony std_logic_vector(format), który ma elementy typu std_logic. Na przykład, jeśli zadeklarować zmienną

```
variable dwabajty : std_logic_vector(15 downto 0);
```

to można zrealizować przypisanie

```
dwabajty := "011XX10UWU11ZZ10";
```

3.2.3 TYPY WEKTOROWO-SKALARNE

Aby wektory o elementach typu bit lub std_logic mogły być stosowane do operacji arytmetycznych jak równoważne liczby całkowite (ze znakiem) lub naturalne (bez znaku), w normie IEEE Std 1076.3-1997 zdefiniowano specjalne typy złożone signed (dla liczb ze znakiem w kodzie ZU2) i unsigned (dla liczb bez znaku) oraz odpowiednie operacje arytmetyczne i funkcje. Na skrajnej prawej pozycji w wektorze jest najmniej znacząca cyfra. Definicje typów o elementach std_logic są takie same:

```
type signed | unsigned is array (natural range <>) of
std_logic;
```

Typy te są ujęte w pakietach standardowych numeric_bit (dla elementów typu bit) i numeric_std (dla elementów typu std_logic). Aby je użyć, trzeba wpisać odpowiednią klauzulę na początku modułu (np. **use** ieee.numeric_std.**all**;).

Na przykład, zmienna zadeklarowana w postaci

```
variable m : unsigned(7 downto 0);
```

przybiera wartości w zakresie od 0 do $2^8 - 1$. Możliwe są operacje m + 1, m*7, **if** m = 30... i podobne. Niemożliwe jest jednak przypisanie liczby całkowitej, np. m := 14. Wtedy trzeba użyć typu łańcuchowego, np. m := "00001110" (niekiedy jest akceptowany zapis m := X"0E"). Można również użyć predefiniowanej funkcji to_unsigned.

3.3 ATRYBUTY

W języku VHDL predefiniowane są liczne atrybuty, pozwalające na identyfikację pewnych cech deklarowanych typów i obiektów. Pozwala

to na łatwiejsze tworzenie bardziej ogólnych modeli układowych. Atrybut związany jest z typem lub obiektem przez apostrof:

```
(typ | obiekt)'atrybut
```

i zwraca wynik stanowiący poszukiwaną cechę typu lub obiektu. Atrybuty są najczęściej związane z typami i sygnałami.

Popularnym atrybutem jest `'event`, który zwraca wartość `true` jeśli w danym kroku symulacji miało miejsce *zdarzenie*, czyli zmiana wartości związanego sygnału. W przeciwnym razie atrybut zwraca wartość `false`. Atrybut ten był już wprowadzony w przykładach `L2.5`, `L2.6` i `L2.7` do wykrycia zbocza sygnału zegarowego.

Do rozpoznawania granic typów i podtypów skalarnych stosuje się atrybuty:

`'left` – zwraca lewostronną granicę zakresu danego typu,
`'right` – zwraca prawostronną granicę zakresu danego typu,
`'high` – zwraca górną granicę zakresu danego typu,
`'low` – zwraca dolną granicę zakresu danego typu.

Przykłady:

```
type zakres is range 15 downto -5;
variable P: integer := zakres'left;      -- P := 15
variable R: integer := zakres'right;     -- R := -5
variable S: integer := zakres'high;      -- S := 15
variable T: integer := zakres'low;       -- T := -5
```

W podobny sposób wykorzystuje się te atrybuty do określania granic zakresów dla typów złożonych, takich jak wektory i tablice. Przykłady:

```
type alfa is array(12 downto 0) of bit;
variable P: integer := alfa'left;        -- P := 12
variable R: integer := alfa'right;       -- R := 0
variable S: integer := alfa'high;        -- S := 12
variable T: integer := alfa'low;         -- T := 0
```

Atrybut `'length` zwraca długość wektora określonego typu:

```
variable U: integer := alfa'length;      -- U := 13
```

Atrybut może również zwracać nazwę elementu w typie wyliczeniowym, na przykład

```
architecture a1 of atrybuty is
   type beta is (czerwony,zielony,niebieski,czarny);
```

```
    signal P,R:  beta;
begin
    P <= beta'left;              -- P <= czerwony
    R <= beta'right;             -- R <= czarny
end a1;
```

Identyfikację formatu wektora umożliwia atrybut `'range`. Na przykład, jeśli

```
signal W : std_logic_vector(0 to 15);
```

to `W'range` zwraca zakres `(0 to 15)`, który można bezpośrednio wykorzystać w instrukcji pętli (**loop**). Analogicznie można otrzymać zakres `(15 downto 0)`. W obydwu przypadkach zakres rozpoczyna się od indeksu lewostronnego i kończy się na indeksie prawostronnym. Analogiczny atrybut `'reverse_range` zwraca odwrócony zakres. Kierunek indeksowania wykrywa atrybut `'ascending`, zwracając wartość `true` lub `false`:

```
type beta is array(2 to 14) of bit;
type gama is array(15 downto 4) of bit;
variable be: boolean := beta'ascending; -- be := true
variable ga: boolean := gama'ascending; -- ga := false
```

W języku VHDL są predefiniowane również inne atrybuty (p. 4.2). W komercyjnych systemach projektowych są także stosowane specyficzne atrybuty, opisane w dokumentacji użytkowej. Można również definiować własne atrybuty.

3.4 PAKIETY I BIBLIOTEKI

Obiekty, podprogramy i modele przeznaczone do stosowania w różnych projektach, takie jak deklaracje typów i podtypów, stałych i sygnałów, a także funkcje i procedury, często umieszcza się w zbiorach zwanych *pakietami* (**package**). Aby nie opisywać w każdym projekcie typowych prostych modeli (przerzutniki, rejestry, multipleksery itd.), często umieszcza się te modele zwane „prymitywami" w odrębnym pakiecie (np. pakiet `wat` podany w załączniku 1).

Zawartość pakietu może być łatwo wykorzystana przez różne (również wszystkie) moduły różnych projektów, bez potrzeby każdorazowego wpisywania określonych deklaracji lub podprogramów do każdego projektu.

Pakiety mogą być predefiniowane, czyli *standardowe* (ujęte w normach IEEE) lub *niestandardowe* (tworzone indywidualnie). Każdy pakiet ma swoją *nazwę*.

Pakiet napisany w języku VHDL składa się z dwu części: *deklaracji* i *ciała pakietu* (podobnie do deklaracji jednostki i ciała architektury). Ilustruje to poniższy przykład pakietu o nazwie `pak1_pkg`.

```
-- L3.1
package pak1_pkg is     -- deklaracja, podanie nazwy
   subtype bv4 is bit_vector(3 downto 0);
   subtype bv8 is bit_vector(7 downto 0);
   subtype r4 is integer range (0 to 3);
   type kolory is (czerwony, zielony, niebieski);
   constant max : integer := 511;
   constant jc : time := 10 ns;
   function testmax(x: integer) return bit;
   -- [dalsze deklaracje]
end pak1_pkg;

package body pak1_pkg is       -- cialo pakietu
   function testmax (x: integer) return bit is
      variable z : bit;   -- zmienna lokalna
   begin
      if (x > max) then z := '1' else z := '0';
      end if;
      return(z);
   end testmax;
   -- [opisy innych funkcji i procedur]
end pak1_pkg;
```

Wyniki kompilacji pakietów, jak również realizowanych projektów, są umieszczane w systemie projektowym w zbiorach zwanych *bibliotekami* (`library`). Każda biblioteka ma przyporządkowaną *nazwę*.

Indywidualnie tworzone pakiety są podczas kompilacji umieszczane zazwyczaj w bibliotece *roboczej* o domyślnej nazwie `work` lub w innej bibliotece, wcześniej zadeklarowanej w danym systemie projektowym. Biblioteka `work` jest domyślnie dostępna w każdym projekcie, bez potrzeby specjalnej deklaracji. W bibliotece `work` domyślnie umieszczane są również wyniki realizowanych projektów.

Aby użyć określonego pakietu z biblioteki, trzeba wpisać na początku tworzonego modułu klauzule o składni

```
library nazwa_biblioteki;
use nazwa_biblioteki.nazwa_pakietu.nazwa_elementu(ów);
```

Polecenie użycia całego pakietu ma postać klauzuli

```
use nazwa_biblioteki.nazwa_pakietu.all;
```

Tak wybrany pakiet lub jego element jest dostępny („widoczny") dla modułu bezpośrednio umieszczonego po klauzuli **use**.

Zaleca się umieszczanie indywidualnie tworzonych pakietów w odrębnych bibliotekach (o różnych nazwach), z których każda zawiera pakiety zbliżone funkcjonalnie lub tworzone przez jednego projektanta. To samo dotyczy projektów wykonywanych przez różnych projektantów. W ten sposób osiąga się lepszą czytelność projektów oraz ukrycie informacji źródłowych, zawartych w ciele pakietu.

Domyślnie przed każdym projektem istnieje (niewidoczny) zapis

```
library std, work;
use std.standard.all;
```

Biblioteka standardowa `std` jest jedną z bibliotek zasobów języka VHDL, którą dysponuje *każdy* kompilator. Jest ona objęta normą IEEE o aktualnej wersji oznaczonej jako 1076-2001 i zawiera dwa pakiety standardowe (predefiniowane): `standard` i `textio`.

Pakiety standardowe zawierają szereg predefiniowanych typów, wymienionych poprzednio. Poza *dwuwartościowym* typem `bit`, ujętym w pakiecie `std.standard`, istnieją również typy *wielowartościowe* (p. 3.2.1), ujęte w pakiecie o nazwie `std_logic_1164`, zawartym w standardowej bibliotece o nazwie `ieee`. Odpowiednikiem typu skalarnego `bit` jest dziewięciowartościowy typ `std_logic`. Analogicznie odpowiednikiem typu złożonego `bit_vector` o elementach typu `bit` jest typ `std_logic_vector` o elementach typu `std_logic`.

Aby użyć wymienionych typów wielowartościowych i ich podtypów należy na początku modułu umieścić klauzule:

```
library ieee;
use ieee.std_logic_1164.all;
```

Zaleca się stosowanie tego pakietu oraz typów `std_logic` i `std_logic_vector` we wszystkich projektach w języku VHDL. W ten sposób otrzymuje się przenośność modeli między różnymi projektantami oraz dokładniejszą symulację opisywanych układów.

UWAGA: Pakiet `std_logic_1164` będzie stosowany w dalszych przykładach, które do kompilacji powinny być uzupełnione przez dopisanie na początku dwu podanych wyżej klauzul.

3.5 INSTRUKCJE WSPÓŁBIEŻNE

Instrukcjami *współbieżnymi* są wszystkie instrukcje, które można bezpośrednio wypisać w ciele architektury. W języku VHDL istnieje sześć takich instrukcji:

- przypisanie do sygnału (<=),
- instrukcja procesu **process**,
- instrukcja współbieżnego wywołania procedury,
- instrukcja łączenia komponentów **port map**,
- instrukcja powielania **generate**,
- instrukcja blokowa **block**.

Współbieżną instrukcję **przypisania do sygnału** stosuje się w następujących postaciach:

- przypisanie podstawowe,
- przypisanie warunkowe **when-else**,
- przypisanie selektywne **with-select**.

Przypisanie podstawowe do sygnału ma składnię

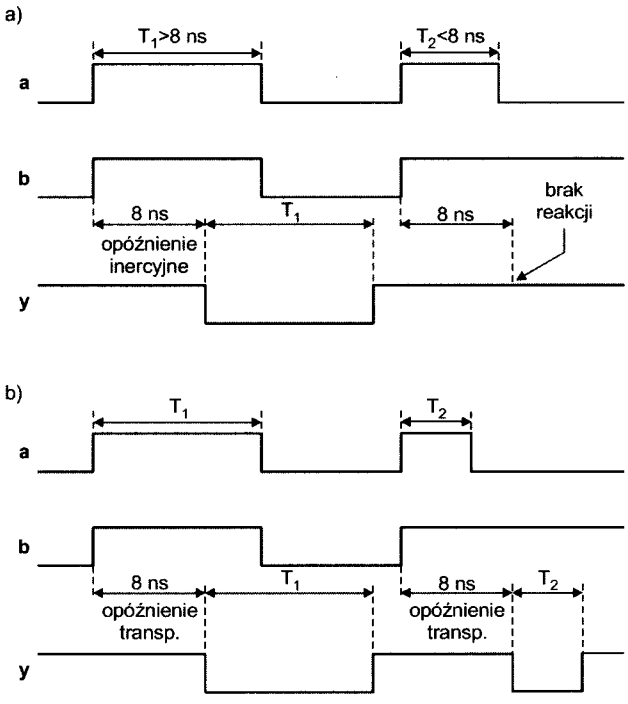
```
sygnal <= [transport] wyrazenie [after odcinek_czasu];
```

gdzie `wyrazenie` musi dawać wynik tego samego typu co `sygnal`. Opcjonalna klauzula **after** służy do opisu opóźnień w realizacji operacji przypisania. `Odcinek_czasu` może być deklarowany bezpośrednio w jednostkach czasu (typu `time`) lub może stanowić wyrażenie dające wynik typu `time`.

Trzeba podkreślić, że wprowadzane w ten sposób opóźnienia są pomijane przez programy służące do syntezy i mogą być wykorzystywane tylko do symulacji. Nie można bowiem oczekiwać, że jakikolwiek program zapewni opóźnienie np. bramki równe *dokładnie* 10 ns. Można natomiast narzucić wymaganie, aby dany blok miał opóźnienie nie dłuższe niż np. 20 ns, a następnie sprawdzić to podczas końcowej symulacji czasowej.

Istnieją dwa rodzaje opóźnień opisywanych z użyciem klauzuli **after**: inercyjne i transportowe. Opóźnienie *inercyjne* odwzorowuje działanie typowej, rzeczywistej bramki w tym sensie, że sygnał wejściowy o czasie trwanie krótszym niż `odcinek_czasu` nie jest transmitowany na jej wyjście. Na przykład, jeśli zapiszemy

```
y <= a nand b after 8 ns;
```



Rys. 3.1 Przykłady ilustrujące opóźnienia: a) inercyjne i b) transportowe

to wyniki wyrażenia `a` **`nand`** `b` nie będą przypisane do sygnału `y`, jeśli czasy trwania stanów sygnałów `a` i `b` będą krótsze niż 8 ns, co ilustruje rys. 3.1a. Jeśli jednak będą równe 8 ns lub dłuższe niż 8 ns, to przypisania będą zrealizowane.

Opóźnienie *transportowe* odwzorowuje czas propagacji przewodu, a więc jest to tak zwane „czyste opóźnienie". Na przykład (rys. 3.1b)

```
y <= transport a nand b after 8 ns;
```

Klauzulę **after** można wykorzystać do utworzenia modelu prostego zegara do celów symulacyjnych:

```
-- L3.2
entity zegar is
   generic(op : time := 50 ns);
   port (Q    :  inout std_logic := '1');
         -- inicjalizacja stanu poczatkowego '1'
end zegar;
```

```
architecture a1 of zegar is
begin
   Q <= not Q after op; -- f = 1/(2*op) = 10 MHz
end a1;
```

Rodzaj **inout** dla portu `Q` umożliwia zarówno zapis jak i odczyt stanów sygnału Q. Stany `'0'` i `'1'` na wyjściu `Q` są równe odcinkowi czasu `op` zadeklarowanemu po klauzuli **generic** i wpisanemu po klauzuli **after**. Model ten jest „niesyntezowalny", to znaczy nie można oczekiwać, że w wyniku syntezy otrzymamy projekt fizycznego układu generatora.

Klauzula **after** może być wykorzystywana różnorodnie, na przykład do tworzenia ciągu impulsów do symulacji:

```
y <=   '1' after 12 ns,
       '0' after 25 ns,
       '1' after 40 ns;
       . . . . . .
```

Przypisanie warunkowe ma składnię

```
sygnal <=
   {wyrazenie when warunek [and | or warunek] else}
   wyrazenie_koncowe;
```

Wynik sprawdzenia warunku jest typu `boolean`, a więc może być równy `false` lub `true`. Gdy warunek po słowie **when** jest spełniony (`true`), to następuje przypisanie wyrażenia sprzed słowa **when** i wykonanie instrukcji się kończy. W przeciwnym razie (**else**) sprawdzany jest kolejny warunek i tak dalej.

Jeśli jedno `wyrazenie` jest deklarowane alternatywnie dla dwu lub więcej warunków, to można te warunki wymienić po słowie **when** w postaci ciągu `warunek` **or** `warunek` **or** `warunek...` . Podobnie, jeśli dane `wyrazenie` jest deklarowane dla iloczynu warunków, to można te warunki wymienić w postaci ciągu `warunek` **and** `warunek` **and** `warunek...` .

Symulator bada ciągi warunków *kolejno* i realizuje przypisanie przy pierwszym spełnionym warunku. Przy stosowaniu tej instrukcji należy wymienić *wszystkie* możliwe warunki, gdyż w przeciwnym razie w wyniku syntezy mogą zostać utworzone przerzutniki zatrzaskowe dla sygnału o nietestowanych wartościach. Jeśli nie chcemy definiować wszystkich możliwych warunków, to na końcu należy wpisać `wyrazenie_koncowe`, które domyślnie spełnia warunek **when others,** co oznacza „dla pozostałych warunków".

Przykład stosowania tej instrukcji był podany wcześniej (L2.4). Dla sygnału s był tam zastosowany dwuwartościowy typ bit tak, że dwa warunki wyczerpywały wszystkie możliwości. Jednakże przy stosowaniu zalecanego typu std_logic, tych wartości jest już dziewięć. W tym przypadku można opisać architekturę multipleksera w postaci

```
architecture a4 of mux2_1 is
begin
   y <= a when s = '0' else b;
end a4;
```

albo w postaci

```
architecture a5 of mux2_1 is
begin
   y <= a when s = '0' else
        b when s = '1' else 'X';
end a5;
```

Można również spróbować wykorzystać klauzulę **unaffected**, czyli „bez zmiany" (niektóre kompilatory nie rozpoznają jednak tej klauzuli):

```
architecture a6 of mux2_1 is
begin
   y <=  a when s = '0' else
         b when s = '1' else
         unaffected;
end a6;
```

W przypisaniu warunkowym można również wykorzystać klauzulę **after** do modelowania opóźnienia, na przykład

```
architecture a7 of mux2_1 is
begin
   y <= a after 6 ns when s = '0' else
        b after 7 ns when s = '1' else unaffected;
end a7;
```

Przypisanie selektywne ma składnię

```
with wybor select
   {sygnal <=  wyrazenie when wartosc_wyboru,}
   sygnal <=  wyrazenie_koncowe when others;
```

Podobnie jak poprzednio, należy wymienić w tej instrukcji *wszystkie* możliwe wartości wyboru, aby nie dopuścić do tworzenia zbędnych przerzutników zatrzaskowych podczas syntezy.

Na przykład, multiplekser *4 na 1* bajtów można opisać modułem

```
-- L3.3
entity muxg8_4_1 is
  port (a,b,c,d  : in std_logic_vector(7 downto 0);
              adr  : in std_logic_vector(1 downto 0);
                y  : out std_logic_vector(7 downto 0));
end muxg8_4_1;
architecture a1 of muxg8_4_1 is
begin
    with adr select
    y <=   a when "00",
           b when "01",
           c when "10",
           d when "11",
           "XXXXXXXX" when others;
end a1;
```

Istotną cechą przypisania *warunkowego* jest kolejność (priorytet) badanych warunków, zgodnie z zapisem. Właściwość ta może powodować podczas syntezy tworzenie bardziej złożonego układu o strukturze szeregowej. *Zaleca się* stosowanie przypisania selektywnego zamiast warunkowego, zwłaszcza przy opisie złożonych układów.

W przypisaniu selektywnym wartości sygnału `wybor` mogą być określone w postaci wyrażenia, będącego alternatywą kilku wartości. Można to wykorzystać do względnie prostego *zapisu tablicy stanów układu kombinacyjnego*. Na przykład, jeśli zadeklarowany jest wektor wejściowy dla funkcji 6 zmiennych

```
a   :   in std_logic_vector(5 downto 0);
```

i sygnał wyjściowy

```
y   :   out std_logic;
```

to można opisać tablicę stanów takiej funkcji następująco:

```
with a select
    y <= '1' when
        --  tu trzeba wypisac wszystkie stany a,
        --  dla ktorych y = 1, na przyklad
              "001100" | "10100X" | "1110XX",
          '0' when
        --  tu trzeba wypisac wszystkie stany a,
        --  dla ktorych y = 0, na przykład
              "X01010" | "1100X0" | "100010",
          'X' when others;
```

Symbol pionowej kreski (|) oznacza *wybór* („albo"). Nie oznacza on jednak funkcji **or**.

Można też zastosować równoważniki dziesiętne *L*(a), ale bez możliwości zapisu stanów X. Nie można tu podstawiać wartości typu unsigned (p. 3.2.3), lecz można użyć funkcję vec_int, służącą do konwersji wektora na liczbę całkowitą (L3.7). Na przykład, tablicę stanów funkcji zupełnej y = *f*(a) o 5 zmiennych można zapisać

```
with vec_int(a) select
   y <=   '1' when 1 | 2 | 6 | 13 | 24 | 27 | 30,
          '0' when others;
```

Instrukcja **procesu** została już wprowadzona i objaśniona w rozdziale 2. Przypomnijmy, że wewnątrz procesu wykonywane są instrukcje sekwencyjne, ale sam proces, jako instrukcja zapisana w ciele architektury, jest współbieżny z pozostałymi instrukcjami zapisanymi w tym ciele. Składnia procesu jest następująca:

```
[etykieta:]  process [(lista_wrazliwosci)][is]
                [czesc_deklaracyjna]
             begin
                {instrukcja_sekwencyjna;}
             end process [etykieta];
```

Lista wrażliwości zawiera nazwy sygnałów, które podczas symulacji powodują wykonanie instrukcji sekwencyjnych wewnątrz procesu, gdy tylko którykolwiek z tych sygnałów zmieni wartość. Lista ta jest opcjonalna w tym sensie, że gdy jej nie ma, to musi być wprowadzona wewnątrz procesu sekwencyjna instrukcja czekania wait. Instrukcja ta ma trzy odmiany, omówione poniżej w p. 3.6. Przy braku listy wrażliwości i braku wewnątrz instrukcji wait proces trwałby stale i teoretycznie nieskończenie długo. Opcjonalna etykieta służy tylko do polepszenia czytelności zapisu i generalnie zaleca się jej stosowanie.

W części deklaracyjnej mogą być umieszczone deklaracje typów (**type**), stałych (**constant**) i zmiennych lokalnych (**variable**).

Pozostałe instrukcje współbieżne będą omówione w rozdziale 4, przy opisywaniu złożonych struktur układowych.

3.6 INSTRUKCJE SEKWENCYJNE

Opisane poprzednio instrukcje współbieżne są wykonywane jednocześnie i odwzorowują działanie układów cyfrowych o strukturze koncep-

cyjnie równoległej. Natomiast instrukcje sekwencyjne umożliwiają opis działania układów w postaci *uporządkowanej sekwencji zdarzeń w czasie*. Instrukcje sekwencyjne stosuje się do opisu *procesów* oraz *podprogramów*, takich jak *procedury* i *funkcje*.

Instrukcje sekwencyjne służą nie tylko do opisu sekwencyjnych układów cyfrowych. W praktyce są też stosowane do opisu kombinacyjnych układów cyfrowych, jeśli trzeba wprowadzić kolejność operacji lub uwarunkowania czasowe wewnątrz opisywanych układów.

Lista instrukcji sekwencyjnych zawiera dwie instrukcje wykorzystywane również jako współbieżne, a mianowicie instrukcję *przypisania podstawowego do sygnału* (p. 3.5) oraz instrukcję *wywołania procedury*. Pozostałe instrukcje sekwencyjne to:

- przypisanie do zmiennej (:=), opisane w p. 3.1;
- instrukcja czekania **wait**;
- instrukcja warunkowa **if-then-else**;
- instrukcja wyboru **case**;
- instrukcja pętli **loop** i związane instrukcje **exit** i **next**;
- instrukcja pusta **null**;
- instrukcja testowa **assert**.

Instrukcja czekania **wait** oznacza przejściowe wstrzymanie wykonania procesu lub podprogramu. Ma ona trzy opcje:

```
wait [until warunek]          -- czekaj dopoki
     [on nazwa_sygnalu {,nazwa_sygnalu}] - czekaj na
     [for odcinek_czasu];  -- czekaj przez
```

Na przykład, architekturę przerzutnika D (L2.7) można równorzędnie opisać posługując się instrukcją **wait on**, czyli „czekaj na (zmianę sygnału (sygnałów))":

```
architecture a2 of ffd is
begin
   process        -- nie ma listy wrazliwosci!
   begin
      wait on (R,C);     -- czekaj na zmiane R,C
      if  R = '1' then Q <= '0';
         elsif (C'event and C = '1') then Q <= D;
      end if;
   end process;
end a2;
```

Instrukcja **wait for**, czyli „czekaj przez" (odcinek czasu), służy do wprowadzenia opóźnienia działania w układzie cyfrowym. Odcinek

czasu może być opisany wyrażeniem, którego wynik ma typ czasowy (p. 3.2), albo bezpośrednio podając wartość stałą o typie czasowym, czyli z użyciem odpowiedniej jednostki czasu. Na przykład, można opisać wirtualny zegar o częstotliwości 20 MHz modułem

```
-- L3.4
entity clock is
   port (Q : out std_logic);
end clock;

architecture a1 of clock is
begin
   process
      variable tq    : std_logic := '0' ;
      constant period : time := 50 ns;
   begin
      wait for period/2;    -- czekaj przez pol okresu
      tq := not tq;
      Q <= tq;
   end process;
end a1;
```

Oczywiście można zrezygnować z deklarowania zmiennej `period` i napisać po prostu **wait for** `25 ns`. Można również zrezygnować z instrukcji **wait**:

```
architecture a2 of clock is
begin
   process(Q)
      variable tq    : std_logic := '0';
   begin
      tq := not tq after 25 ns;
      Q <= tq;
   end process;
end a2;
```

Opisane postacie instrukcji **wait** można również łączyć w jednym wyrażeniu, na przykład

```
wait on a for 250 ns;  -- czekaj na a przez 250 ns
wait until b = '0' for 150 ns;
--  czekaj na spelnienie warunku b = '0' przez 150 ns
wait on a until d = '1' for 66 ns;
--  czekaj na a dopóki nie spelni sie warunek d = '1'
--  przez 66 ns
```

Stosując instrukcję **wait** trzeba pamiętać, aby *zawsze* wprowadzać odpowiednią opcję (warunek, nazwę sygnału lub odcinek czasu). W przeciwnym razie czekanie może być nieskończenie długie.

Proces bez listy wrażliwości wykonuje się od początku (**begin**) do pierwszego wystąpienia instrukcji **wait**, a następnie zatrzymuje się na zadeklarowany odcinek czasu. Gdy proces zawiera tylko jedną instrukcję **wait**, to po odczekaniu tego odcinka czasu proces wykonuje się do końca (**end process**), a następnie znowu rozpoczyna działanie od początku, zatrzymuje się ponownie na instrukcji **wait** itd. W opisie procesu lub podprogramu można wprowadzić szereg różnych instrukcji czekania, co umożliwia tworzenie bardzo złożonych modeli, np. zegarów wielofazowych.

Instrukcja warunkowa **if-then-else** stanowi sekwencyjny odpowiednik współbieżnej instrukcji przypisania warunkowego **when-else**, której nie można stosować w obrębie procesu ani podprogramu. Składnia jest następująca:

```
if warunek then {instrukcja sekwencyjna;}
    {elsif warunek then {instrukcja sekwencyjna;}}
    [else {instrukcja sekwencyjna;}]
end if;
```

Wynik warunku jest typu `boolean`, a więc `false` lub `true`. Gdy warunek jest spełniony (wynik `true`), to wykonywana jest sekwencja instrukcji następująca bezpośrednio po słowie **then**. W przeciwnym razie (**else**) wykonywana jest inna sekwencja albo sprawdzany jest kolejny warunek (**elsif** warunek **then**) i tak dalej. Sprawdzanie warunków następuje *kolejno*, a więc pierwsze wypisane warunki mają odpowiednio wyższy priorytet. Przykłady podano w rozdziale 2 (`L2.5`, `L2.6` i `L2.7`).

Instrukcja wyboru **case** stanowi sekwencyjny odpowiednik współbieżnej instrukcji przypisania selektywnego **with-select-when**, której nie można stosować w obrębie procesu ani podprogramu. Składnia jest następująca:

```
case wyrazenie is
    {when wybor => {instrukcja sekwencyjna;}}
    [when others => {instrukcja sekwencyjna;}]
end case;
```

W tej instrukcji wybor jest pojedynczą wartością wyrazenia albo grupą takich wartości. Przy opisie powinny być wymienione *wszystkie* wzajemnie wyłączające się wartości, albo dla wartości nieużywanych należy wprowadzić zapis

```
when others => sekwencja_instrukcji
```

lub

```
when others => null;  -- brak dzialania
```

Sprawdzanie wartości następuje *równolegle* (jednocześnie), czyli żadna z nich nie ma priorytetu względem innych.

Jako wybor można użyć wyrażenia alternatywnego, grupującego kilka wartości przy użyciu symbolu *albo* (|). Na przykład, jeśli c jest liczbą całkowitą i y jest sygnałem **out** typu std_logic

```
case c is
   when 1 | 3 | 4 | 14 => y <= '1';
   when 2 | 5 | 6 => y <= '0';
   when 7 | 9 => y <= 'X';
   when others => null;
end case;
```

Instrukcja pętli loop umożliwia zapis powtarzania sekwencji instrukcji, co behawioralnie odzwierciedla działanie układu. W języku VHDL istnieją trzy rodzaje pętli: **for**, **while** i pętle nieskończone.

Jeśli liczba iteracji (obiegów pętli) jest z góry znana, to stosuje się instrukcję **for-loop** o składni

```
[etykieta:] for parametr in zakres loop
   {instrukcja sekwencyjna;}
end loop [etykieta];
```

Parametr pętli jest zazwyczaj indeksem zakresu definiowanego przez granice i kierunek indeksowania (to | downto). Na przykład, inwersję wszystkich bitów wektora A można opisać za pomocą pętli

```
for i in 0 to A'length - 1 loop
   B(i) <= not A(i);   -- B zawiera negacje bitow A
end loop;
```

Indeks (w tym przykładzie i) nie musi być odrębnie deklarowany i jest rozpoznawany wyłącznie w ramach pętli. Atrybut 'length umożliwia identyfikację długości wektora. Opcjonalna etykieta zwiększa czytelność programu, zwłaszcza jeśli stosuje się w nim więcej pętli.

Jeśli liczba iteracji zależy od wyniku sprawdzania warunku przed powtórzeniem pętli, to stosuje się instrukcję **while-loop** o składni

```
[etykieta:] while warunek loop
              {instrukcja sekwencyjna;}
            end loop [etykieta];
```

Na przykład można opisać wirtualny zegar, który działa tylko wtedy gdy zmienna boolowska `flaga` ma wartość **true**:

```
process
begin
   while flaga loop
      zegar <= not zegar;  -- typ bit lub std_logic
      wait for okres_zegara/2;  -- typ time
   end loop;
end process;
```

Instrukcja wyjścia z pętli

```
exit[etykieta_petli][when warunek];
```

służy do wyjścia z pętli, jeśli warunek jest spełniony. Można zrezygnować z opcji `[when warunek]`, ale oznacza to bezwarunkowe wyjście z pętli przy pierwszym napotkaniu słowa **exit** podczas wykonywania instrukcji pętli. Warunek można też wprowadzić stosując instrukcję warunkową **if-then**:

```
if warunek then exit;
end if;
```

Instrukcja wyjścia z iteracji pętli

```
next [etykieta_petli][when warunek]
```

służy do zakończenia wykonywania bieżącej iteracji pętli i przejście do wykonywania następnej, jeśli warunek jest spełniony. Jednocześnie indeks pętli zwiększa się o jeden.

Instrukcja pusta `null` wskazuje, że nie jest wykonywane żadne działanie poza przejściem do wykonania następnej instrukcji. Jak pokazano wyżej, często jest ona stosowana na końcu instrukcji **case**, gdy trzeba uwzględnić nieokreślone wartości wyrażenia sterującego.

Instrukcja testowa **assert** służy do sprawdzania poprawności symulacji i alarmowania w razie błędów przez odpowiednie komunikaty tekstowe. Składnia jest następująca:

```
[etykieta:] assert warunek
    [report "tekst_komunikatu"]
    [severity severity_level];  --  poziom zagrożenia
```

Jeśli warunek nie jest spełniony, to generowany jest komunikat alarmowy. Do wyboru są dwie opcje, z których można wybrać jedną lub obydwie. Opcja **report** umożliwia wyświetlenie wpisanego komunikatu, a opcja **severity** (zagrożenie) wyświetla słowo informujące o stopniu zagrożenia. Może być sygnalizowany jeden z czterech zadeklarowanych poziomów zagrożenia: Failure (uszkodzenie), Error (błąd), Warning (ostrzeżenie) i Note (uwaga). Na przykład

```
assert (t_setup < 2 ns)
    report "Za krotki czas ustalenia!"
    severity Warning;
```

i jeśli warunek nie jest spełniony, to symulator zareaguje komunikatem Assertion violation.

Na przykład, można opisać przerzutnik SR modułem

```
-- L3.5
entity ffsr is
    port(S,R : in std_logic;
             Q : out std_logic);
end ffsr;

architecture test of ffsr is
begin
    process(S,R)
    begin
        assert S='1' nand R='1';  -- test S=R='1'
        if S='1' then Q<='1';
        end if;
        if R='1' then Q<='0';
        end if;
    end process;
end test;
```

Jeśli podczas symulacji wystąpi niedozwolony stan S = R = '1', to zostanie wygenerowany komunikat alarmujący.

3.7 FUNKCJE I PROCEDURY

W języku VHDL funkcje i procedury są definiowane i stosowane w podobny sposób jak w innych językach wysokiego poziomu, takich jak Pascal i C. Funkcja jest podprogramem, którego parametrami są wyłącznie sygnały wejściowe i który jako wynik zwraca jedną wartość określonego typu. W języku VHDL funkcja jest traktowana jak *wyrażenie*, które można stosować tylko w obrębie procesu lub podprogramu. Procedura jest podprogramem, który na liście parametrów ma zarówno wejścia i wyjścia. W języku VHDL procedura jest traktowana jak *instrukcja*. Ściślej mówiąc, *wywołanie procedury* jest instrukcją, która może być współbieżną lub sekwencyjną, zależnie od miejsca jej użycia.

Do tworzenia podprogramów można stosować wyłącznie instrukcje sekwencyjne. W obrębie podprogramów, podobnie jak wewnątrz procesów, nie mogą być deklarowane sygnały wewnętrzne, lecz tylko zmienne lokalne. Jednak w odróżnieniu od procesów, w podprogramach wartości zmiennych lokalnych są ustalane od nowa przy każdym wywołaniu podprogramu. Oznacza to, że zmienne lokalne nie mogą być wykorzystane do „pamiętania" stanów w podprogramach.

Funkcja przed użyciem (czyli wywołaniem) musi być najpierw zadeklarowana. *Deklarację* funkcji umieszcza się *lokalnie* w części deklaracyjnej architektury (przed słowem **begin**) opisywanego modułu albo w nagłówku procesu, albo *globalnie* w odrębnym i zadeklarowanym pakiecie (jeśli chce się łatwo używać danej funkcji w większej liczbie modułów). *Wywołanie* funkcji wykonuje się używając jej nazwy jako *wyrażenia* z podaniem listy parametrów przechodzących. Wartości tych parametrów nie są zmieniane przez funkcję. Wyrażenie może być użyte tak jak inne, na przykład w instrukcji przypisania do sygnału.

Składnia deklaracji funkcji jest następująca:

```
function nazwa [(lista_parametrow)] return typ is
    [czesc deklaracyjna]
    --  zazwyczaj deklaracje zmiennych lokalnych
    --  nie mozna deklarowac sygnalow!
begin
    {instrukcja sekwencyjna;}
    -- przynajmniej jedna instrukcja musi
    -- zawierać słowo return(wyrazenie)
end [function][nazwa];
```

Argumenty funkcji, czyli *parametry formalne* wymieniane na liście są rodzaju **in** i domyślnie (niejawnie) są przyjmowane jako należące do klasy stałych (**constant**), ale muszą być jawnie deklarowane w klasie **signal**, jeśli w obrębie funkcji wykorzystuje się na przykład atrybuty sygnału. Parametry funkcji nie mogą należeć do klasy zmiennych (**variable**). Przy wywołaniu funkcji parametry formalne są zastępowane przez parametry *aktualne* (*przechodzące*), które mogą należeć do klas **constant** lub **signal**. Typ parametrów musi być deklarowany jawnie.

Instrukcja **return** kończy wykonywanie funkcji, a przyporządkowane mu `wyrazenie` zwraca wynik funkcji, czyli wartość zadeklarowanego typu. W obrębie funkcji nie można stosować instrukcji **wait**.

Przykład funkcji komparacji trzech bitów:

```
-- L3.6  Funkcja komparacji trzech bitów
function komp3(a,b,c : std_logic) return boolean is
begin
   if ((a xor b) nor (a xor c)) = '1' then
      return true;
      else return false;
   end if;
end komp3;
```

W powyższym przykładzie funkcja zwraca wynik typu `boolean`, ale można ją łatwo zmodyfikować, aby otrzymać wynik typu `std_logic`.

Poniższa funkcja vec_int służy do konwersji wektora typu `std_logic_vector` na liczbę całkowitą (typu `integer`). Podobną rolę pełni typ wektorowo-numeryczny `unsigned` (p. 3.2.3), ale tylko w odniesieniu do operacji arytmetycznych.

```
-- L3.7
function vec_int(signal a : std_logic_vector) return
integer is
   variable p : integer;
begin
   p := 0;
   for i in a'range loop
      p := p * 2;
      case a(i) is
         when '1' | 'H' => p := p + 1;
         when others => null;
      end case;
   end loop;
   return p;
end vec_int;
```

W tej funkcji wykorzystano znany wzór iteracyjny, służący do zamiany słowa dwójkowego na równoważnik dziesiętny. Jeśli słowo dwójkowe oznaczyć jako $B = b_{n-1}b_{n-2}...b_1b_0$, to

$$L(B) = ((...((b_{n-1}\cdot 2 + b_{n-2})\cdot 2 + ...)\cdot 2 + b_1)\cdot 2 + b_0$$

Niezależnie od długości i sposobu indeksowania wektora a atrybut sygnałowy a'range umożliwia prawidłowe iteracje w pętli, rozpoczynając od lewostronnego bitu (MSB). Ponieważ użyto atrybutu sygnałowego, więc trzeba było zadeklarować parametr formalny a jako **signal**. Również parametr aktualny musi należeć do klasy **signal**.

Wartość 'H' może wystąpić, gdyż sygnał a ma elementy typu std_logic. Przypomnijmy, że symbol | oznacza „albo". Zamiast instrukcji **case** można użyć instrukcji **if-then-else**.

Wiele funkcji jest zdefiniowane w pakiecie standardowym std_logic_1164. Wygodne w użyciu są funkcje wykrywające narastające i opadające zbocze sygnału, np. zegara. Są to funkcje o łatwo interpretowalnych nazwach:

```
function rising_edge(signal s : std_ulogic) return
boolean;
function falling_edge(signal s : std_ulogic) return
boolean;
```

Oznacza to, że zamiast pisać długą instrukcję **if** zegar'event **and** (zegar = '1') **then**..., można to samo wyrazić krótszym zapisem **if** rising_edge(zegar) **then**... . Typem obiektu może oczywiście być std_logic.

Można również utworzyć „uniwersalne" funkcje logiczne, automatycznie rozpoznające liczbę argumentów, jeśli wejścia zadeklaruje się w postaci wektora. Na przykład funktor **nand** o liczbie argumentów (wejść) równej liczbie elementów wektora można opisać funkcją

```
-- L3.8 Funkcja nand
function fnand(signal a: std_logic_vector) return
std_logic is
   variable p : std_logic;
begin
   p := '1';
   for i in a'range loop
      p := p and a(i);
   end loop;
   return not p;
end fnand;
```

Trzeba podkreślić, że *użycie funkcji jest dopuszczalne tylko w ciele procesu lub podprogramu.* Na przykład, jeśli były zadeklarowane sygnały

```
signal s,t    :  in std_logic_vector(0 to 7);
signal r      :  in std_logic;
```

to w ciele procesu (pomiędzy słowami **process begin** i **end process**) można funkcję fnand użyć na przykład w postaciach

```
if fnand(s) = '1' then...;
r <= fnand(s) or fnand(t);
```

Procedura może zwracać więcej wyników niż jeden. Podobnie jak funkcja, procedura najpierw musi być *zadeklarowana* albo globalnie w pakiecie, albo lokalnie w deklaracji architektury (przed słowem **begin**), albo w nagłówku procesu. Następnie procedura może być *wywołana* w ciele architektury jako instrukcja *współbieżna* lub w obrębie procesu bądź innego podprogramu jako instrukcja *sekwencyjna.*

Składnia deklaracji procedury jest następująca:

```
procedure nazwa [(lista_parametrow)] is
[czesc deklaracyjna]
--  zazwyczaj deklaracje zmiennych lokalnych
--  nie mozna deklarowac sygnalow!
begin
   {instrukcja sekwencyjna;}
   --  moze byc uzyte slowo return (bez wyrazenia)
end [procedure][nazwa];
```

Parametry formalne, wymieniane na liście, mogą być rodzaju **in**, **out**, lub **inout**. Rodzaj **in** służy do wprowadzania wartości parametrów *do* procedury, rodzaj **out** służy do zwracania wartości parametrów *z* procedury (czyli jej wyników), a rodzaj **inout** umożliwia modyfikację jednego parametru (wprowadzanego i zwracanego). Parametry formalne mogą należeć do klas **constant**, **variable** lub **signal** i mogą należeć do trzech rodzajów: **in**, **out** lub **inout**. Na liście należy definiować zarówno klasę jak i rodzaj parametru. Wykonanie procedury kończy się po napotkaniu słowa **return** lub **end** [nazwa].

Procedurę można wywołać w ciele architektury jako instrukcję *współbieżną* pod warunkiem, że żaden z parametrów na liście nie należy do klasy **variable**. *Użycie zmiennych nie jest bowiem dopusz-*

czalne poza procesem lub podprogramem. Procedura jest wykonywana gdy wystąpi *zdarzenie*, czyli zmiana stanu choćby jednego parametru z listy. W obrębie procesu lub podprogramu procedurę wywołuje się jako instrukcję *sekwencyjną* i wtedy jest ona wykonywana jak pozostałe instrukcje sekwencyjne.

Poniższy przykład opisuje tester parzystości wektora A o automatycznie rozpoznawalnej długości, generujący dwa sygnały wyjściowe: P (gdy liczba jedynek jest parzysta) i N (gdy jest odwrotnie).

```
-- L3.9
procedure test_par(signal A : in std_logic_vector;
   signal P,N : out std_logic) is
   variable  t : std_logic;
begin
   t := '0' ;
   for i in A'range loop
      t := t xor A(i);
   end loop;
   P <= t;
   N <= not t;
end test_par;
```

Przykład procedury opisującej przerzutnik JK wyzwalany zboczem narastającym, z asynchronicznym zerowaniem i ustawianiem:

```
-- L3.10
procedure FFJK1
      (signal C,J,K,R,S :  in std_logic;
       signal Q  :  inout std_logic;
       signal QN :  out std_logic) is
   variable  JK :  std_logic_vector(0 to 1);
   variable  re :  boolean;
begin
   JK := J & K;
   re := rising_edge(C);
   if R = '1' then Q <= '0';
      elsif S = '1' then Q <= '1';
      elsif re then
         case JK is
            when  "10" => Q <= '1';
            when  "01" => Q <= '0';
            when  "11" => Q := not Q;
            when others  =>  null;
         end case;
   end if;
   QN <= not Q;
end FFJK1;
```

Do pamiętania i odczytu „obecnego" stanu przerzutnika wykorzystano port o rodzaju **inout.** Jest to w istocie *jedyna możliwość do wykorzystania w procedurze opisującej ten przerzutnik*, ponieważ nie można zastosować do tego celu ani sygnału wewnętrznego lub wyjścia **buffer** (niedopuszczalne w procedurze) ani zmiennej lokalnej (nie przechowuje wartości po wykonaniu procedury).

Można do opisu przerzutnika JK wykorzystać również jego logiczne równanie funkcyjne:

```
-- L3.11
procedure FFJK2
    (signal C,J,K,R,S   :  in std_logic;
     signal Q  : inout  std_logic;
     signal QN : out std_logic) is
    variable  re :  boolean;
begin
    re := rising_edge(C);
    if R = '1' then Q <= '0';
       elsif S = '1' then Q <= '1';
       elsif re then
          Q <= (J and not Q) or (not K and Q);
    end if;
    QN <= not Q;
end FFJK2;
```

Niektóre kompilatory nie akceptują używania wewnątrz procedur atrybutu `'event` ani predefiniowanej funkcji `rising_edge()` odnośnie do parametrów klasy **signal**. W niektórych kompilatorach można jednak użyć funkcji `rising_edge()`, ale pod warunkiem że nie będzie ona użyta wewnątrz instrukcji **if-then-else**, lecz z odrębnym przypisaniem do zmiennej lokalnej, w sposób opisany w przykładzie.

Przy wywoływaniu procedury trzeba na liście parametrów zastąpić nazwy parametrów formalnych nazwami parametrów aktualnych. Proste zastąpienie oznacza przyporządkowania *niejawne* (domyślne). Na przykład

```
FFJK2(zegar,a,b,reset,set,Q4,Q4not);
```

W tym przypadku trzeba zwrócić uwagę, aby liczba i kolejność nazw ściśle odpowiadała deklaracji procedury. Bardziej czytelne jest wywołanie procedury z przyporządkowaniem *jawnym* parametrów, gdzie po lewej stronie *symbolu przyporządkowania* (=>) wpisuje się parametr formalny, a po prawej stronie parametr aktualny. Na przykład

```
FFJK2(C=>zegar,J=>a,K=>b,R=>reset,S=>set,Q=>Q4,QN=>
Q4not);
```

Ponieważ wszystkie przyporządkowania są opisane kompletnie, zatem kolejność ich wypisania może być dowolna. Odpowiedni przykład jest podany w następnym rozdziale (L4.5).

4

MODELOWANIE ZŁOŻONYCH UKŁADÓW CYFROWYCH

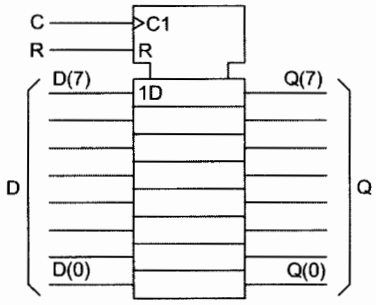
Józef Kalisz

4.1 SPOSOBY OPISU

W poprzednich rozdziałach przedstawiono podstawy języka VHDL oraz sposoby opisu podstawowych układów logicznych, jakimi są bramki, proste multipleksery i przerzutniki. W niniejszym rozdziale będą opisane sposoby modelowania bardziej złożonych układów logicznych.

Łatwo można opisać rejestry równoległe. Wystarczy w tym celu zamienić typ sygnałów wejściowych i wyjściowych ze skalarnego na złożony. Wykorzystując model przerzutnika D z przykładu L2.7, można opisać n-bitowy rejestr równoległy w sposób pokazany niżej.

```
-- L4.1
entity regn is
    generic(n :  positive := 8);    -- deklaracja n = 8
    port (C,R :  in std_logic;
          D   :  in std_logic_vector(n-1 downto 0);
          Q   :  out std_logic_vector(n-1 downto 0));
end regn;
architecture a1 of regn is
begin
    process (R, C)
    begin
        if  R = '1' then Q <= (others => '0');
            elsif rising_edge(C) then
            Q <= D;
        end if;
    end process;
end a1;
```



Rys. 4.1 Ośmiobitowy rejestr równoległy z zerowaniem asynchronicznym

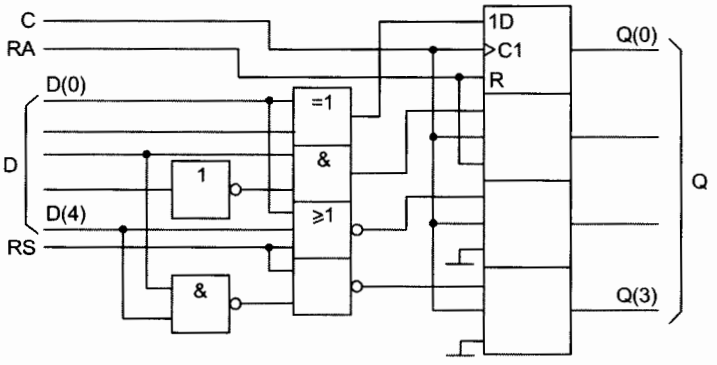
Odpowiedni symbol graficzny pokazuje rys. 4.1. Można również utworzyć uniwersalną procedurę dla rejestru równoległego, przy czym liczba bitów (n) rejestru nie musi być jawnie deklarowana, a do rozpoznania tej liczby stosuje się atrybut 'range:

```
-- L4.2
procedure reg
    (signal C,R  : in  std_logic;
    signal D   : in std_logic_vector;
    signal Q   : out std_logic_vector) is
    variable ff  :  boolean;  --  niekonieczne
begin
    ff := rising_edge(C);      --  niekonieczne
    if R = '1' then Q <= (Q'range => '0');
    -- zerowanie wektora o dowolnej dlugosci
        elsif ff then Q <= D;
        -- zamiast ff mozna uzyc rising_edge(C)
    end if;
end reg;
```

Przy syntezie typowych układów synchronicznych wejścia rejestru są połączone z układami kombinacyjnymi (np. bramkami), realizującymi odpowiednie funkcje wzbudzeń. Przykładowy schemat logiczny takiego układu pokazuje rys. 4.2.

Taką strukturę logiczną można opisać różnymi architekturami. Na przykład, stosując styl *behawioralny* można utworzyć model

```
-- L4.3
entity ex4_3 is
    port (C,RA,RS : in std_logic;
          D : in std_logic_vector(4 downto 0);
          Q : out std_logic_vector(3 downto 0));
end ex4_3;
```



Rys. 4.2 Przykładowy układ cyfrowy z rejestrem i zespołem bramek

```
architecture a1 of ex4_3 is
begin
  p0: process
      begin
         if RA = '1' then Q(0) <= '0';
            else wait until rising_edge(C);
            Q(0) <= D(0) xor D(1);
         end if;
      end process p0;
  p1: process(C,RA)
      begin
         if RA = '1' then Q(1) <= '0';
            elsif rising_edge(C) then
            Q(1) <= D(2) and not D(3);
         end if;
      end process p1;
  p2: process
      begin
         wait until rising_edge(C);
         Q(2) <= not(D(0) or D(4) or RS);
      end process p2;
  p3: process(C)
      begin
         if rising_edge(C) then
            if RS = '1' then Q(3) <= '0';
               else Q(3) <= D(2) and D(4);
            end if;
         end if;
      end process p3;
end a1;
```

W podanym przykładzie wprowadzono procesy bez listy wrażliwości (p0 i p2) oraz z listą wrażliwości (p1 i p3). Trzeba zaznaczyć, że niektóre kompilatory (np. *Warp* firmy *Cypress)* wymagają, aby w procesach bez listy wrażliwości instrukcja wait była *pierwszą* instrukcją procesu. Sygnały RA i RS oznaczają zerowanie odpowiednio asynchroniczne i synchroniczne (wykonywane przy aktywnym zboczu zegara). Proces p3 można również zapisać krócej, stosując opis *logiczny*:

```
p3: process(C)
begin
   if rising_edge(C) then
      Q(3) <= (not RS) and D(2) and D(4);
   end if;
end process p3;
```

Do opisu architektury tego układu można również wykorzystać *procedurę* podobną do podanej w L4.2. Różnica polega na wprowadzeniu odrębnych, asynchronicznych wejść zerujących w postaci wektora R. Założono, że długości wektorów D, R i Q są takie same.

```
-- L4.4
procedure regr
  (signal C   :  in std_logic;
   signal R,D    :  in std_logic_vector;
   signal Q   :  out std_logic_vector) is
   variable      rr :  boolean;
begin
   rr := rising_edge(C);
   for i in R'range loop
      if R(i)= '1' then Q(i) <= '0';
         elsif rr then Q(i) <= D(i);
      end if;
   end loop;
end regr;
```

Załóżmy że procedura ta jest zawarta w roboczym pakiecie pak, umieszczonym w bibliotece work. Aby ją wywołać, trzeba określić parametry aktualne (przechodzące) procedury. Stosując styl *przepływowy* (*RTL*) można utworzyć prostszy opis architektury pokazanego przykładu. Kompletny opis układu jest następujący:

```
-- L4.5
library ieee;
use ieee.std_logic_1164.all;
use work.pak.all;  -- jesli korzystamy z pakietu pak

entity ex4_3 is
```

```
    port (C,RA,RS:  in std_logic;
          D   :  in std_logic_vector(4 downto 0);
          Q   :  out  std_logic_vector(3 downto 0));
end ex4_3;

architecture a2 of ex4_3 is
    -- tutaj trzeba wpisac procedure regr
    -- jesli nie korzystamy z pakietu pak
    signal w,z :  std_logic_vector(3 downto 0);
begin
    w(0) <= D(0) xor D(1);
    w(1) <= D(2) and not D(3);
    w(2) <= not(D(0) or D(4) or RS);
    w(3) <= (not RS) and D(2) and D(4);
    z <= ('0','0',RA,RA);
    regr(C,z,w,Q); -- wywolanie procedury
end a2;
```

Trzecim stylem opisu architektury bloków funkcjonalnych (oprócz stylu *behawioralnego* i *przepływowego* – *RTL*) jest styl *strukturalny*. Opis strukturalny stanowi tekstowy odpowiednik schematu logicznego lub, przy bardziej złożonych komponentach, schematu blokowego.

Przy użyciu stylu strukturalnego architektura opisuje sieć połączeń układów o prostszej strukturze, które są określane jako *komponenty*. Są one opisywane odrębnymi modułami i deklarowane za pomocą deklaracji **component**. Aby opisać sieć „połączeń" sygnałów wyjściowych (typu **out**) komponentów z portami wejściowymi (typu **in**) komponentów trzeba wprowadzić sygnały pomocnicze. Do wejść komponentów można „przyłączyć" nie tylko sygnały, ale również wyrażenia (na przykład wartość stałą), albo można określone wejścia pozostawić otwarte, przypisując im słowo **open**.

Deklarację połączeń komponentów tworzy się wykorzystując współbieżną instrukcję *łączenia* („osadzania") *komponentów* **port map**

```
[etykieta:] nazwa_komponentu port map(lista);
```

Na liście opisuje się połączenia przez odpowiednie przyporządkowania. Można je zapisać w sposób jawny lub niejawny, podobnie jak przy wywołaniu procedury. Stosując zapis *jawny* (kompletny) przyporządkowań trzeba użyć symbolu przyporządkowania (=>):

```
port_wewnetrzny => sygnal_zewnetrzny | wyrazenie | open
```

i wtedy kolejność wypisania przyporządkowań nie jest istotna. Można również pominąć wtedy opis tych portów, których stany mają stałe

wartości, zadeklarowane wcześniej. Stosując zapis *niejawny* (domyślny) pomija się nazwy portów wewnętrznych i symbol przyporządkowania. Wtedy jednak trzeba zachować kolejność i pozycje portów podane w deklaracji komponentów, jak też trzeba opisać wszystkie porty.

Nazwy sygnałów pomocniczych mogą być takie same jak nazwy portów komponentów. Wtedy zapis wydaje się nieco prostszy. Trzeba jednak wtedy opisywać przyporządkowania jawne, jak na przykład C=>C i Q=>Q, co jest trudniejsze w interpretacji.

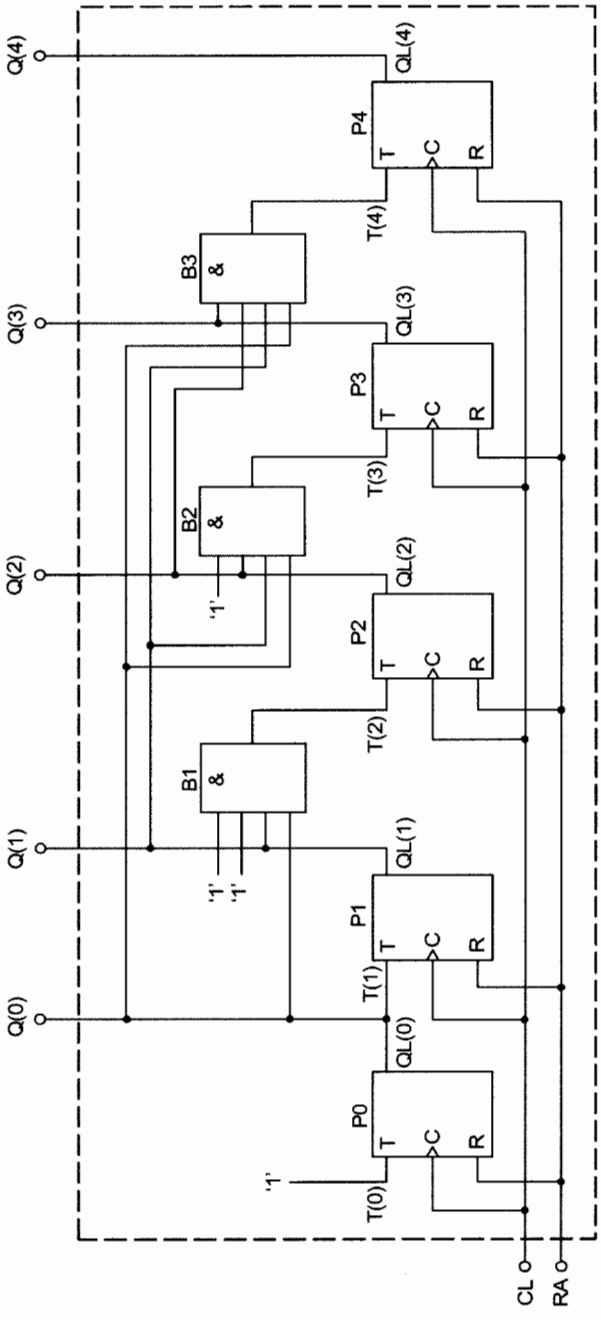
Modele układów wykorzystywanych jako komponenty są standardowymi modułami, składającymi się z deklaracji jednostki i jednej lub więcej architektur. Moduły te mogą być wpisywane bezpośrednio przed ich deklaracją jako komponentów albo mogą być umieszczane w odpowiednich pakietach. Bardzo pomocne są *pakiety prymitywów*, zawierające modele najprostszych bramek i przerzutników. Są one często dostarczane wraz z oprogramowaniem firmowym. Można także tworzyć pakiety indywidualne, zawierające „własne" modele różnych układów.

Podobnie jak podprogramy, modele układów zawarte w pakiecie mogą być wielokrotnie „kopiowane" do opisu złożonych układów.

Na rys. 4.3 pokazano przykładowy schemat logiczny pięciobitowego, dwójkowego licznika synchronicznego z przeniesieniami równoległymi. W jego opisie zastosowano dwa komponenty: czterowejściową bramkę AND i przerzutnik T.

W poniższym przykładzie opisu strukturalnego komponenty są opisane na początku, co oznacza, że ich skompilowane opisy znajdują się w w bieżącej bibliotece roboczej o domyślnej nazwie work. Mogą one być również umieszczone w odpowiednio zadeklarowanym pakiecie (na przykład o nazwie pak).

```
--  L4.6
--  model czterowejsciowej bramki AND
library ieee;
use ieee.std_logic_1164.all;
entity and4 is
   port (A,B,C,D:  in std_logic;
               y   :  out  std_logic);
end and4;
architecture a1 of and4 is
begin
   y <= a and b and c and d;
end a1;
```



Rys. 4.3 Przykład licznika pięciobitowego, opisanego stylem strukturalnym

```
-- model przerzutnika T (analogicznie do L2.5_a2)
library ieee;
use ieee.std_logic_1164.all;
entity fft is
   port (T,C,R   :   in std_logic;
               Q   :   out std_logic);
end fft;
architecture a1 of fft is
   signal tt   :  std_logic;
begin
   process (C,R)
   begin
      if  R = '1' then tt <= '0';
         elsif rising_edge(C)then
         tt <= tt xor T;
      end if;
   end process;
   Q <= tt;
end a1;

-- model licznika
library ieee;
use ieee.std_logic_1164.all;
-- use work.pak.all; gdy komponenty sa w pakiecie pak
entity licznik5 is
   port(CL,RA    :   in std_logic;
         Q       :   out std_logic_vector(4 downto 0));
end licznik5;
architecture struktura of licznik5 is
--  deklaracje komponentow
   component and4
      port (A,B,C,D    :   in std_logic := '1';
   --  domyslnie na wszystkich wejsciach jest stan '1'
                  Y   :   out  std_logic);
   end component;
   component fft
      port (T,C,R   :   in std_logic;
                  Q   :   out std_logic);
   end component;
   --  deklaracja sygnalow pomocniczych
   signal T,QL   :   std_logic_vector(4 downto 0);
begin
   --  opis polaczen przez przyporzadkowania jawne:
   T(0) <= '1';
   P0: fft port map(T=>T(0),C=>CL,R=>RA,Q=>QL(0));
   T(1) <= QL(0);
   P1: fft port map(T=>T(1),C=>CL,R=>RA,Q=>QL(1));
   --  na wejsciach C i D jest domyslny stan '1':
```

```
    B1: and4 port map(A=>QL(0),B=>QL(1),Y=>T(2));
    P2: fft port map(T=>T(2),C=>CL,R=>RA,Q=>QL(2));
    -- przyporzadkowania niejawne:
    B2: and4 port map(QL(0),QL(1),QL(2),'1',T(3));
    -- teraz trzeba bylo opisac wszystkie porty
    P3: fft port map(T(3),CL,RA,QL(3));
    B3: and4 port map(QL(0),QL(1),QL(2),QL(3),T(4));
    P4: fft port map(T(4),CL,RA,QL(4));
    Q <= QL;
end struktura;
```

W tym prostym przykładzie dwa komponenty architektury zdefiniowano stosując te same nazwy, które mają poprzednio opisane, pojedyncze moduły bramki i przerzutnika. Gdyby jednak było kilka modułów dla każdego komponentu, a dodatkowo gdyby każdy moduł miał kilka architektur, to wtedy pojawia się zadanie wyboru i powiązania określonego modułu i jego architektury z każdym komponentem użytym przy opisie połączeń w projekcie. Do tego celu stosuje się specyfikację lub deklarację *konfiguracji*.

Konfiguracja architektury zawierającej strukturalny opis połączeń między komponentami jest zapisem wyboru modułu i jego architektury dla każdego komponentu w projekcie. Taki wybór określa się jako *wiązanie* (*binding*). Przy deklaracji komponentów można wtedy wybrać inne nazwy niż nazwy wiązanych modułów.

Na przykład, jeśli zamiast **component** `and4` zadeklarować **component** `komp1`, to po tej deklaracji (przed słowem **begin**) trzeba opisać wiązanie dla połączeń stosujących ten komponent. Na przykład, dla połączenia `B1` można zadeklarować użycie jednostki `and4`

```
for B1 : komp1 use entity work.and4;
```

i wtedy domyślnie wybierana jest ostatnia architektura opisana w module `and4`. Jeśli dla `B1` chcielibyśmy wykorzystać moduł `and4x` z pakietu `pak`, oraz wybrać określoną architekturę `ar3` tego modułu, to wtedy opis wiązania ma postać

```
for B1 : komp1 use entity pak.and4x(ar3);
```

W podobny sposób można związać komponenty z odpowiednimi modułami pakietu we wszystkich połączeniach. Jeśli kilka połączeń ma mieć identyczne wiązania, to połączenia można wypisać kolejno, np. „**for** `B1,B2` :" itd. Dla pozostałych połączeń można użyć zapisu „**for**

others :". Jeśli wszystkie połączenia mają mieć te same wiązania, to piszemy „**for all** : ".

Opisany wyżej sposób wiązania stanowi *specyfikację konfiguracji* projektu. Można również opisać konfigurację przy użyciu *deklaracji konfiguracji*. Jest to odrębny moduł, dopisywany do projektu po opisie jego architektury. Deklaracja prostej konfiguracji ma składnię

```
configuration nazwa of nazwa_jednostki_projektu is
   for nazwa_architektury_projektu
      {for nazwa_polaczenia :
         nazwa_komponentu klauzula_wiazania;
       end for;}
end [configuration] [nazwa];
```

Na przykład, poprzednią specyfikację można opisać deklaracją K1:

```
configuration K1 of licznik5 is
   for struktura
      for B1 : komp1 use entity pak.and4x(ar3);
         -- dalsze wiazania
   end for;}
end configuration K1;
```

W praktyce często interesujący jest wpływ opisu architektury komponentów na złożoność i szybkość działania zaprojektowanego układu. Taką analizę można wykonać stosując odpowiednie zmiany konfiguracji w sposób opisany wyżej i analizując końcowe raporty projektowe.

Niektóre kompilatory (np. *Leonardo Spectrum*) umożliwiają wykorzystanie konfiguracji również w prostych projektach, w których nie stosuje się łączenia komponentów. W szczególności dotyczy to pojedynczych modułów opisanych różnymi architekturami. Wtedy deklaracja konfiguracji jest bardzo prosta. Na przykład, jeśli jest opisany moduł o jednostce `uklad1` i architekturach `a1`, `a2` i `a3`, to wybór architektury `a2` realizuje się deklaracją

```
configuration konf1 of uklad1 is
   for a2
   end for;
end;
```

W takiej sytuacji niektóre systemy (np. *Warp* i *WebPack*) nie sygnalizują błędu, ale niezależnie od deklaracji zawsze wybierają ostatnio wypisaną, czyli domyślną architekturę. Niektóre kompilatory w ogóle nie akceptują deklaracji konfiguracji, ale realizują jej specyfikację.

Styl strukturalny można również stosować do opisu bardziej złożonych układów, gdzie jako komponenty można użyć dowolnych bloków funkcjonalnych. Na przykład, układ licznika z rys. 4.3 można również użyć jako komponentu o definicji

```
component licznik5 is
   port(CL,RA    :  in std_logic;
             Q   :  out std_logic_vector(4 downto 0));
end component;
```

gdzie porty CL, RA i Q reprezentują sygnały wewnętrzne licznika.

Przy opisie bardzo złożonych układów wygodnie jest wprowadzić dekompozycję opisu na mniejsze części zwane *blokami*. Można je wtedy wymieniać lub składać (stosując edytor tekstowy), co jest przydatne zwłaszcza wówczas, gdy poszczególne bloki tworzą różni projektanci. Instrukcja współbieżna **block** opisuje wybrany blok następująco:

```
[etykieta:]block
[czesc deklaracyjna, na przykład sygnaly lokalne]
begin
   {instrukcja wspolbiezna;}
end block [etykieta];
```

4.2 PRZYKŁADY OPISU BLOKÓW FUNKCJONALNYCH

Przedstawione poniżej przykłady są przedstawione w postaci modeli i procedur, łatwych do zastosowania przy opisach złożonych układów.

Multiplekser

Popularnym kombinacyjnym blokiem funkcjonalnym jest multiplekser. Przykłady opisu multiplekserów były podane w rozdziale 2. Zazwyczaj stosuje się *przepływowy styl opisu funkcyjnego lub logicznego*. W pierwszym przypadku zaleca się stosowanie instrukcji współbieżnej **with-select-when**, która implikuje realizację równoległej struktury logicznej. Model multipleksera „4 na 1" można opisać modułem

```
-- L4.7
entity mux4 is
  port (a,b,c,d  : in std_logic);
              dr : in std_logic_vector(1 downto 0);
               y : out std_logic);
end mux4;
```

```
architecture a1 of mux4 is
begin
   with adr select
   y <=   a when "00",
          b when "01",
          c when "10",
          d when others;    -- albo: d when "11",
                            --       "X" when others;
end a1;
```

W opisie procedury (styl behawioralny) należy użyć instrukcji sekwencyjnej **case-is-when**:

```
--  L4.8
procedure mux4(
   signal a,b,c,d   :  in std_logic;
   signal adr       :  in std_logic_vector(1 downto 0);
   signal y         :  out std_logic) is
begin
   case adr is
      when "00" => y <= a;
      when "01" => y <= b;
      when "10" => y <= c;
      when others => y <= d;
   end case;
end mux4;
```

Ostatni przypadek można sformułować inaczej. Na przykład, jeśli chcemy aby przy niezerojedynkowych stanach wejść adresowych na wyjściu pojawił się stan 'X':

```
when "11" => y <= d;
when others => y <= 'X';
```

Opis *logiczny* odwzorowuje bezpośrednio typową strukturę logiczną multipleksera. W odniesieniu do jednostki mux4 można opisać architekturę

```
architecture a2 of mux4 is
begin
   y <=   a and (not adr(0)) and (not adr(1)) or
          b and adr(0) and (not adr(1)) or
          c and (not adr(0)) and adr(1) or
          d and adr(0) and adr(1);
end a2;
```

Modele multiplekserów o innych liczbach wejść danych tworzy się analogicznie. Opis multipleksera grupowego „4 na 1" bajtów podany jest w L3.3.

Konwertery kodów

Modele prostych konwerterów można utworzyć wypisując odpowiednią tablicę stanów logicznych. Na przykład, konwerter kodu „1 z 10" na kod BCD8421 (koder) można opisać procedurą

```
--  L4.9
procedure conv_to_BCD(
    signal ten : in std_logic_vector(9 downto 0);
    signal bcd : out std_logic_vector(3 downto 0)) is
begin
    case ten is
        when "0000000010" => bcd <= "0001";
        when "0000000100" => bcd <= "0010";
        when "0000001000" => bcd <= "0011";
        when "0000010000" => bcd <= "0100";
        when "0000100000" => bcd <= "0101";
        when "0001000000" => bcd <= "0110";
        when "0010000000" => bcd <= "0111";
        when "0100000000" => bcd <= "1000";
        when "1000000000" => bcd <= "1001";
        when others => bcd <= "0000";
    end case;
end conv_to_BCD;
```

Analogicznie łatwo można opisać dekoder, czyli układ przetwarzający kod BCD8421 (lub inny kod BCD) na kod „1 z 10". Można również opisać logicznie konwerter kodu zapisując jego znaną strukturę logiczną za pomocą odpowiednich wyrażeń logicznych z operatorami **and**, **or**, **not**.

Opis znacznie się komplikuje przy opisie konwerterów wielobitowych, na przykład konwerterów kodu NB na BCD i odwrotnie.

Komparatory cyfrowe

Jednobitowy komparator opisuje się dwuargumentową funkcją XNOR:

```
-- L4.10
entity komp1 is
    port(p,q  :  in std_logic;
         y    :  out std_logic);
end komp1;
```

```
architecture a1 of komp1 is
begin
   y <= p xnor q;
end a1;
```

Analogicznie do L4.2, uniwersalną procedurę komparatora wielobitowego można zapisać następująco:

```
-- L4.11
procedure kompar
   (signal   P  :  in std_logic_vector;
    signal   Q  :  in std_logic_vector;
             y  :  out std_logic) is
   variable  rr :  std_logic;
begin
   rr := '1';
   for i in P'range loop
      if P(i)/= Q(i)then
         rr := '0';
         exit;
      end if;
   end loop;
   y <= rr;
end kompar;
```

Predykat P(i)/= Q(i) można oczywiście również zapisać w postaci funkcji logicznej, analogicznie jak poprzednio: P(i) **xor** Q(i) = '1', co może nieco ułatwić zadanie syntezy. Mimo to model ten jest dość złożony jak na tak prostą funkcję, a ponadto iteracyjne badanie nierówności kolejnych par bitów może prowadzić do syntezy szeregowej, a więc relatywnie powolnej struktury układu komparatora. Najprościej funkcję komparatora wielobitowego opisuje procedura z bezpośrednim sprawdzaniem predykatu P = Q:

```
-- L4.12
procedure komp
   (signal   P  :  in std_logic_vector;
    signal   Q  :  in std_logic_vector;
             y  :  out std_logic) is
begin
   if P = Q then y <= '1' else y <= '0';
   end if;
end komp;
```

Przy takim zapisie można się spodziewać, że w wyniku syntezy otrzyma się układ o strukturze równoległej.

Licznik synchroniczny z przeniesieniami szeregowymi

Na rysunku 4.3 pokazany był przykładowy schemat logiczny pięciobitowego, dwójkowego licznika synchronicznego z przeniesieniami równoległymi, opisany modelem L4.6. Opis strukturalny złożonych układów o powtarzalnych komponentach można znacznie uprościć posługując się instrukcją *powielania* **for-in-generate**. Występuje ona w dwu postaciach. Postać *podstawowa* ma składnię

```
etykieta: for indeks in zakres generate
          [czesc deklaracyjna]
          [begin] {instrukcja wspolbiezna;}
          end generate [etykieta];
```

Instrukcja powielania może być również wykonywana warunkowo, ale wtedy bez definiowania zakresu liczbowego. Warunkowa instrukcja powielania ma składnię

```
etykieta: if warunek generate
          [czesc deklaracyjna]
          [begin] {instrukcja wspolbiezna;}
          end generate [etykieta];
```

Użycie instrukcji podstawowej ilustruje poniższy przykład opisu licznika synchronicznego z przeniesieniami szeregowymi. Liczba bitów licznika jest deklarowana jednorazowo w wierszu **generic** (w przykładzie N = 8). Komponentem jest przerzutnik T, do którego wejścia T jest dołączone wyjście trójwejściowej bramki AND. Sygnał T musi mieć wyjście rodzaju **inout**, ponieważ jest on również sygnałem wejściowym dla bramki w kolejnym komponencie.

```
-- L4.13
library IEEE;
use IEEE.std_logic_1164.all;
entity ffg is
   port (e1,e2,e3,C,R  :  in std_logic;
                 Q     :  out std_logic;
                 T     :  inout std_logic);
end ffg;

architecture a1 of ffg is
   signal tt  :  std_logic;
begin
   T <= e1 and e2 and e3;
   process (C,R)
   begin
```

```
        if  R = '1' then tt <= '0';
            elsif rising_edge(C)then
            tt <= tt xor T;
        end if;
    end process;
        Q <= tt;
    end a1;

    library IEEE;
    use IEEE.std_logic_1164.all;
    entity licz is
        generic (N : positive := 8); -- liczba bitow
        port(CL,RA,EN : in std_logic;
                Q : out std_logic_vector(N-1 downto 0));
    end licz;

    architecture gener of licz is
        component ffg port (e1,e2,e3,C,R : in std_logic;
                            Q  :  out std_logic;
                            T  :  inout std_logic);
        end component;
        signal TL,QL :  std_logic_vector(N-1 downto 0);
    begin
        G0: ffg port map (e1=>EN,e2=>'1',e3=>'1',C=>CL,
        R=>RA,T=>TL(0), Q=>QL(0)); -- pierwszy przerzutnik
        G1: for i in 1 to N-1
            generate  --  powielanie
                G2: ffg port map (e1=>EN,e2=>QL(i-1),
                    e3=>TL(i-1),C=>CL,R=>RA, T=>TL(i),
                    Q => QL(i));
            end generate;
    Q <= QL;
end gener;
```

Jeśli jako komponenty trzeba wykorzystać różne moduły z określonymi architekturami, to można wprowadzić specyfikację lub deklarację konfiguracji w sposób opisany w p. 4.1.

Dalsze przykłady są podane w następnych rozdziałach.

4.3 WERYFIKACJA PROJEKTÓW

Projekty tworzone w języku VHDL mogą być bardzo proste i zawierać na przykład tylko jeden prosty moduł, ale mogą być również bardzo złożone, obejmując pewną liczbę bloków w strukturze hierarchicznej. Do sprawdzenia prostych projektów może wystarczyć najprostszy test,

którym jest poprawny wynik kompilacji, to znaczy nie sygnalizujący błędów składni języka VHDL w opisie projektu. Bardziej złożone projekty wymagają jednak bardziej szczegółowego testowania, ponieważ poprawny składniowo opis projektu jeszcze nie gwarantuje działania układu zgodnego z oczekiwaniami projektanta.

Testowanie projektów realizuje się poprzez *symulację* ich działania na komputerze. Jest to odpowiednik testowania sprzętowego, które tradycyjnie wykonuje się na stanowisku testowym konstruktora. Badany układ jest na takim stanowisku łączony z generatorami odpowiednich sygnałów testowych, a działanie układu ocenia się na podstawie obserwacji sygnałów wyjściowych, na przykład za pomocą wielokanałowego oscyloskopu lub analizatora stanów logicznych.

Istnieją dwie metody symulacji projektów: symulacja *graficzna* i *programowa*. Są one wzajemnie powiązane w tym sensie, że na podstawie symulacji graficznej można utworzyć program testowy i odwrotnie.

Symulacja graficzna jest procesem wykonywanym w środowisku graficznym systemu projektowego lub symulacyjnego i realizowanym w kilku kolejnych krokach. Są to następujące czynności:

- zdefiniowanie graficzne przebiegów czasowych dla określonych sygnałów wejściowych w testowanym modelu układu, czyli tak zwanych „wymuszeń";
- uruchomienie programu symulatora, który generuje przebiegi czasowe sygnałów na zadeklarowanych wyjściach testowanego modelu;
- sprawdzenie, czy przebiegi wyjściowe są zgodne z oczekiwanym działaniem projektu.

Zależnie od użytego systemu projektowego, możliwe są dwa rodzaje symulacji, które umownie określa się jako *funkcjonalna* i *czasowa*.

Symulacja funkcjonalna jest realizowana wyłącznie na podstawie opisu projektu w języku VHDL i uwzględnia opóźnienia czasowe wprowadzone w tym opisie przez projektanta z użyciem klauzuli **`after`** (p. 3.5). Tak deklarowane opóźnienia nie są jednak uwzględniane w czasie syntezy projektu i jego wpasowania do wybranego typu układu scalonego. W wyniku tych kroków powstaje opis topograficzny połączeń między elementami logicznymi na kostce krzemowej docelowego układu scalonego. Z tymi połączeniami i elementami związane są określone opóźnienia, które system oblicza na podstawie posiadanej

biblioteki modeli układów scalonych. Wyniki obliczeń opóźnień czasowych między sygnałami na końcówkach układu są zazwyczaj zawarte w raporcie końcowym, tworzonym przez system po zakończeniu projektu.

Symulacja czasowa jest symulacją takiego projektu, który wykorzystuje konfigurację i dane czasowe modelu układu otrzymane po wykonaniu syntezy i projektu topograficznego. Struktura takiego modelu *wynikowego* jest już inna niż modelu opisanego na początku w języku VHDL, lecz funkcje i opisy końcówek pozostają te same. Symulację czasową określa się również jako symulację z parametrami „przypisanymi wstecz" (*back-annotated*) lub jako symulację „poprojektową" (*post-layout*). W niektórych programach symulacyjnych można również automatycznie sprawdzać, czy są spełnione warunki czasowe, narzucone przez projektanta.

Można zauważyć, że użyte nazewnictwo metod (funkcjonalna, czasowa) nie jest najlepsze, jako że w istocie obydwie metody symulacji są *czasowe*.

Programy do symulacji funkcjonalnej i czasowej zazwyczaj wchodzą w skład uniwersalnych systemów projektowych, ale niektóre są również dostępne jako odrębne pakiety. Odpowiednie przykłady są podane w rozdziałach 5 – 8.

Symulacja programowa polega na utworzeniu i wykonaniu programu testowego, napisanego w języku VHDL. **Program testowy** (*test bench*) jest programowym odpowiednikiem stanowiska testowego na biurku (*bench*) konstruktora. W programie testowym opisuje się sekwencje zmian wirtualnych sygnałów lub wektorów testowych, które służą jako wymuszenia dla testowanego modelu układu. Następnie opisuje się warunki, których spełnienie oznacza prawidłowe działanie modelu. Jeśli podczas wykonania programu testowego dany warunek nie jest spełniony, to generowany jest odpowiedni komunikat alarmowy.

Program testowy, dalej krótko określany jako **test**, jest modelem utworzonym w języku VHDL na tych samych zasadach jak każdy inny model. Jest to jednak oddzielny moduł, który powinien być umieszczony w innej bibliotece niż moduł testowany. Ten ostatni wprowadza się do testu jako komponent.

Najprostszy test polega na zdefiniowaniu sekwencji zmian stanów sygnałów wejściowych testowanego układu i sprawdzaniu jego odpowiedzi na wyjściach. Jako przykład można sformułować test do sprawdzenia trzywejściowej bramki NAND, opisanej przez `L2.1`. Niech

model tej bramki znajduje się w domyślnym pakiecie work. Wtedy test bramki można opisać w postaci modułu z charakterystyczną jednostką bez deklaracji portów:

```
-- L4.14
library ieee;
use ieee.std_logic_1164.all;
use work.nand3;  -- uzyj modelu nand3 z pakietu work

   -- jednostka testu nie ma portow
entity tnand3 is
end tnand3;

architecture at1 of tnand3 is
   component nand3  -- deklaracja komponentu
      port(a,b,c    :  in std_logic;
                y   :  out std_logic);
   end component;
-- deklaracja sygnalow testowych i sygnalu wyjsciowego
   signal x1,x2,x3,z   :  std_logic;
begin
   -- polaczenie komponentu z sygnalami wymuszajacymi
   TU1: nand3 port map(a=>x1,b=>x2,c=>x3,y=>z);
   -- opis procesu symulacji
   process
      constant okres : time := 50 ns;
   begin
      x1<='1'; x2<='1'; x3<='1';
      wait for okres;
      assert(z='0')
         report "TEST NEGATYWNY!" severity ERROR;
      x1<='0'; x2<='1'; x3<='1';
      wait for okres;
      assert(z='1')
         report "TEST NEGATYWNY!" severity ERROR;
      x1<='1'; x2<='0'; x3<='1';
      wait for okres;
      assert(z='0') -- oop! sprawdzmy blad testu...
         report "TEST NEGATYWNY!" severity ERROR;
      x1<='X'; x2<='X'; x3<='0';
      wait for okres;
      assert(z='1')
         report "TEST NEGATYWNY!" severity ERROR;
      -- podobnie mozna opisac inne stany...
      wait;  -- koniec testu
   end process;
end at1;
```

Taki test trzeba skompilować, a następnie wykonać w trybie symulacji użytego systemu projektowego. Jak widać, pełne testowanie w ten sposób nawet prostych modeli może być bardzo złożone, biorąc pod uwagę 9 wartości użytego typu `std_logic`.

Uproszczenie opisu testu można osiągnąć przez zdefiniowanie wielowymiarowej tablicy w postaci stałej (**constant**), która zawiera *wektory testowe* o uprzednio zdefiniowanym typie tablicowym (`array`), oraz przez użycie pętli (**loop**) do wywoływania poszczególnych wektorów z tablicy. Dalszym usprawnieniem jest użycie *plików* do zapisu i odczytu wektorów testowych. Do testowania dużych projektów stosuje się testy *algorytmiczne* z użyciem pewnej liczby pętli, których opis może być stosunkowo krótki i łatwy w interpretacji. Przykłady takich testów są podane w rozdziałach 5 – 8 oraz w literaturze [2 – 6].

5

SYSTEM PROJEKTOWY *WebPACK ISE*

Rafał Szymanowski

System projektowy *WebPACK ISE* firmy *Xilinx* jest zestawem programów umożliwiających realizację projektów w strukturach programowalnych CPLD i FPGA. Prezentowany opis dotyczy wersji *4.1* pracującej w środowisku Windows 98/2000/ME/NT4.0. System *WebPACK ISE* umożliwia wykonanie:

- opisu projektów wieloma sposobami,
- symulacji funkcjonalnej i czasowej projektu,
- projektu dla struktury programowalnej,
- programowania wybranych układów CPLD i FPGA.

System umożliwia projektowanie z zastosowaniem następujących układów:

CPLD:	XC9500, XC9500XL, XC9500XV, CoolRunner-II (XC2C000), CoolRunner XPLA3 (XCR3000XL),
FPGA:	Spartan-II (XC2S000), Spartan-IIE (XC2S000E), Virtex-E (XCV000E), Virtex-II (XC2V000).

Opis projektu może być wykonany z użyciem:

- opisu w formacie EDIF, uzyskanym z innego narzędzia projektowego,
- języka VHDL,
- języka Verilog,
- języka ABEL.

System zawiera także programy graficzne, umożliwiające tworzenie modeli układów w postaci grafu stanów (*StateCAD*) lub schematu (*ECS*). W opisie schematowym można użyć symboli modułów opisanych w językach VHDL, Verilog lub ABEL, jak również symboli bibliotecznych bloków funkcjonalnych (bramki, komparatory, multipleksery, sumatory, przerzutniki, liczniki, rejestry). Pliki wynikowe z tych programów nie są zapisywane w żadnym ze standardowych formatów jak np. EDIF. Ponieważ *WebPACK ISE* zawiera kompilator języków VHDL i Verilog, zatem wymienione pliki wynikowe są poddawane translacji, w wyniku której otrzymuje się moduły opisane w jednym z tych języków, zależnie od wybranego typu projektu. Moduł otrzymany ze schematu zawsze zawiera opis strukturalny.

Modele układów opisane w języku ABEL są również poddawane translacji do modułów VHDL lub Verilog. Język ABEL jest dostępny tylko do projektowania z układami programowalnymi CPLD.

W zależności od wybranego typu układu CPLD lub FPGA, dostępne są różne narzędzia wspomagające proces projektowania. Wybierając docelowy układ CPLD projektant może wykonać symulację funkcjonalną oraz czasową projektu. Pomiędzy procesami syntezy i implementacji program *ChipView* można użyć do przypisania sygnałom określonych końcówek układu scalonego. Po implementacji program umożliwia obejrzenie konfiguracji makrokomórek z opisem realizowanych w nich funkcji logicznych. Wybierając docelowy układ FPGA można wykonać symulację:

- funkcjonalną modelu (VHDL, Verilog),
- funkcjonalną po syntezie,
- czasową po podziale projektu na bloki logiczne, realizowane w makrokomórkach,
- czasową po alokacji bloków logicznych w strukturze FPGA i zaprojektowaniu połączeń.

Po podziale projektu na bloki logiczne, można je alokować w strukturze FPGA używając programu *FloorPlanner*.

Niezależnie od wyboru docelowej struktury (FPGA czy CPLD), dostępny jest program *Timing Analyzer*, który służy do analizy czasów propagacji sygnałów wewnątrz układu programowalnego w oparciu o modele biblioteczne. Nie uwzględnia on jednak czasów ustalenia i trzymania przerzutników.

Dla układów Spatan, Virtex i CoolRunner XPLA3 można przeprowadzić analizę mocy strat używając programu *XPower*.

Do symulacji służy symulator *ModelSim* firmy *Model Technology*. Symulator ten wymaga napisania programu testowego w języku VHDL lub Verilog. Program taki można utworzyć przy użyciu edytora *HDL Bencher*. Jest to program graficzny, służący do określania stanów sygnałów wejściowych i wyjściowych. Ponadto dokonuje on porównania wyników symulacji z określonymi przez projektanta stanami sygnałów wyjściowych.

5.1 INSTALACJA SYSTEMU

System projektowy *WebPACK* i symulator *ModelSim* są dostępne bezpłatnie w Internecie pod adresem www.xilinx.com/sxpresso/webpack.htm. Warunkiem uzyskania dostępu do plików instalacyjnych jest wypełnienie formularza rejestracyjnego, w którym deklarujemy nazwę konta i hasło do niego. Plik instalacyjny systemu *WebPACK* ma pojemność około 110 MB (*WebPACK_41wp30_full_installer.exe*) a symulatora *ModelSim* około 39 MB (*MXE_41wp30_full_installer.exe*). Do zainstalowania obu pakietów wymagana jest pojemność 440 MB na dysku. Po zainstalowaniu system *WebPACK* jest gotowy do uruchomienia, natomiast symulator *ModelSim* wymaga pliku licencyjnego, który można uzyskać w końcowym etapie instalacji. Program instalacyjny łączy się z witryną internetową firmy *Xilinx*, gdzie należy podać nazwę utworzonego wcześniej konta i hasło do niego. Następnie plik licencyjny jest wysyłany do użytkownika pocztą elektroniczną. W przypadku braku dostępu do Internetu podczas instalacji należy wysłać formularz zawarty w pliku *lic_request.txt* pod adres cs_1@xilinx.com. Plik ten znajduje się w folderze, gdzie zainstalowano symulator (*...\Modeltech_xe*).

Używając programu *Model Technology License Wizard* (*Start → Programy → ModelSim XE 5.5b → Licensing Wizard*) wybieramy ścieżkę dostępu do pliku licencyjnego.

Domyślnie instalacja pakietów odbywa się odpowiednio w folderach *C:\xilinx_webpack* i *C:\Modeltech_xe*. Aby zachować porządek na dysku zaleca się utworzyć wspólny folder dla obu programów, na przykład *C:\Xilinx*.

5.2 ORGANIZACJA I ZASOBY BIBLIOTECZNE SYSTEMU

Dla każdego projektu system *WebPACK ISE* tworzy odrębny folder o nazwie projektu. W takim folderze umieszczane są wszystkie pliki

związane z danym projektem. Zestawienie tych plików zawiera tablica 5.1. Symbol strzałki określa plik wynikowy. Dla realizowanych projektów wskazane jest utworzenie dodatkowego folderu (lub folderów), na przykład *C:\Xilinx\Projekty*.

Tablica 5.1 Typy plików tworzonych w procesie projektowania

Etapy procesu projektowania	Typ pliku
Opis projektu:	
- VHDL	.vhd
- Verilog	.v
- ABEL	.abl, .abv → .vhf, .vf
- schemat (*ECS*)	.sch → .vhf, .vf
- graf stanów (*StateCAD*)	.dia → .vhd, .v, .abl
Ograniczenia (*Constraints Editor*)	.ucf
Synteza (*Synthesize*)	.vhd, .v → .ngc
Implementacja projektu:	
- połączenie wszystkich części projektu z uwzględnieniem ograniczeń (*Translate*)	.ucf + .ngc → .ngd .edn, .edf → .ngd
- wpasowanie (*Fit* – CPLD)	.ngd → .vm6
- podział na bloki logiczne (*Map* – FPGA)	.ngd → _map.ncd
- alokacja bloków logicznych i zaprojektowanie połączeń (*Place&Root* – FPGA)	_map.ncd → .ncd
Symulacja:	
- funkcjonalna	.vhd, .v
- czasowa	.vhd, .v, .sdf
- program testowy:	
- *HDL Bencher*	.tbw
- VHDL	.vhd
- Verilog	.tf
- ABEL	.abv
Programowanie:	
- CPLD	.vm6 → .jed
- FPGA	.ncd → .bit
Struktura projektu	.npl

System *WebPACK ISE* zawiera obszerny zestaw pomocy (*Help*) i podręczników (*Tutorial*), których zestawienie znajduje się w pliku *...\xilinx_webpack\doc\manuals.pdf*. Po uruchomieniu systemu wybierając *Help → Project Navigator Contents* uzyskujemy dostęp do pomocy dla wszystkich programów pakietu. Pliki tych pomocy (*.hlp*) znajdują się w folderze *...\xilinx_webpack\help\nt\usenglish*. Poprzez *Help → Project Navigator Contents* uzyskujemy dostęp do zbioru pomocy i podręczników projektowania w strukturach CPLD i FPGA. Jako przykład opisany jest licznik Johnsona (*File → Open Example*). Pliki projektowe licznika umieszczone są w folderze *...\xilinx_webpack\WPexamples*.

Podręczniki dla programów *HDL Bencher* i *XPower* znajdują się w następujących folderach w formacie PDF:

- *HDL Bencher* – *...\xilinx_webpack\data\Bencher\bench.pdf*,
- *XPower* –
  ...\xilinx_webpack\help\nt\usenglish\xpowercpldtutorial.pdf
  oraz
  ...\xilinx_webpack\help\nt\usenglish\xpowerfpgatutorial.pdf.

Podczas projektowania można korzystać z dołączonych do systemu *WebPACK ISE* pakietów. Tablica 5.2 zawiera wykaz tych pakietów z opisem ich zawartości. Pierwsze trzy pakiety są pakietami standardowymi IEEE. Pozostałe pakiety są produktami firmy *Synopsys* i nie są objęte normami IEEE. Wszystkie wymienione pakiety znajdują się w bibliotece `IEEE`.

Tablica 5.2 Pakiety systemu *WebPACK ISE*

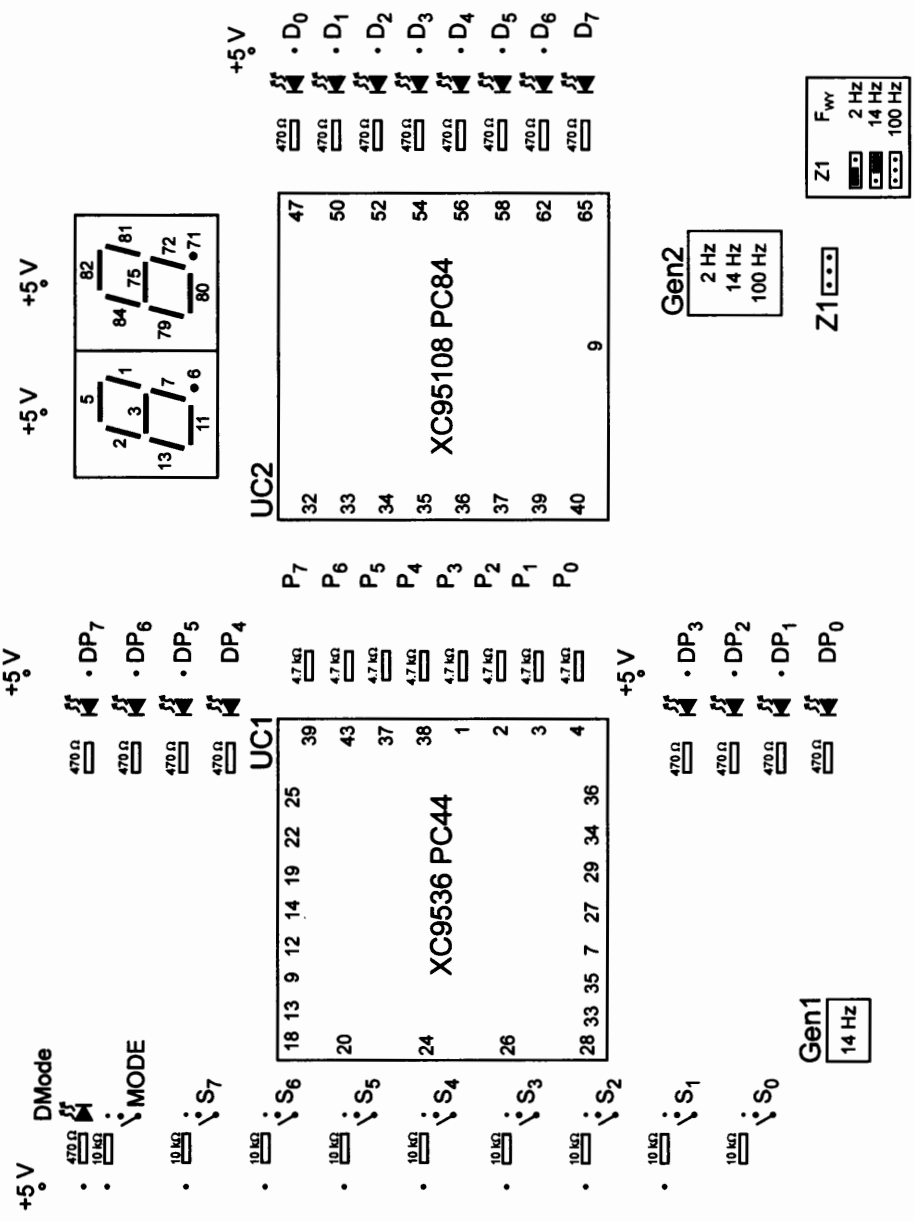
Pakiet	Zawartość
`std_logic_1164`	definiuje typy `std_logic`, `std_ulogic`, `std_ulogic_vector` i `std_logic_vector`, funkcje logiczne i funkcję rozdzielczą
`numeric_bit`	definiuje typ wektorowo-skalarny `signed` i `unsigned` na bazie typu `bit`, operacje arytmetyczne, logiczne, konwersji, przesuwu
`numeric_std`	definiuje typ wektorowo-skalarny `signed` i `unsigned` na bazie typu `std_logic`, operacje arytmetyczne, logiczne, konwersji, przesuwu
`std_logic_arith`	jak pakiet `numeric_std`
`std_logic_unsigned`	operacje arytmetyczne, logiczne, przesuwu i konwersji dla typu `std_logic_vector` – bez znaku
`std_logic_signed`	operacje arytmetyczne ze znakiem, logiczne, przesuwu i konwersji dla typu `std_logic_vector` – ze znakiem
`std_logic_misc`	definiuje dodatkowe typy, stałe, funkcje dla pakietu `std_logic_1164`

5.3 PŁYTKA TESTOWA

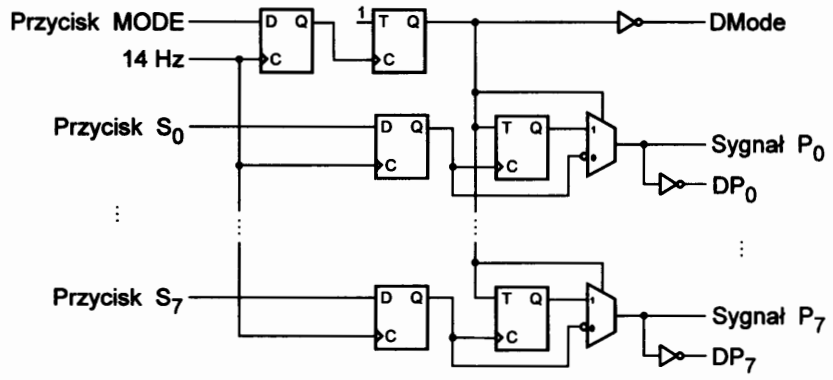
Do praktycznego testowania realizowanych projektów laboratoryjnych przeznaczona jest opracowana przez autora (rafal@wel.wat.waw.pl) płytka testowa RAF1_01, której uproszczony schemat przedstawia rys. 5.1. Na płytce znajdują się dwa układy programowalne CPLD:

- UC1 – XC9536 PC44,
- UC2 – XC95108 PC84.

Układ UC1 wytwarza sygnały $P_0 \div P_7$ na podstawie stanu przycisków $S_0 \div S_7$ eliminując wpływ drgań ich styków. Diody $DP_0 \div DP_7$ świecą, gdy odpowiedni sygnał $P_0 \div P_7$ ma poziom wysoki. Przycisk MODE służy do wyboru pracy sygnałów $P_0 \div P_7$: generacja pojedynczego impulsu, ustawianie poziomu niskiego lub wysokiego. Generator Gen1 (NE555)



Rys. 5.1 Uproszczony schemat płytki testowej RAF1_01

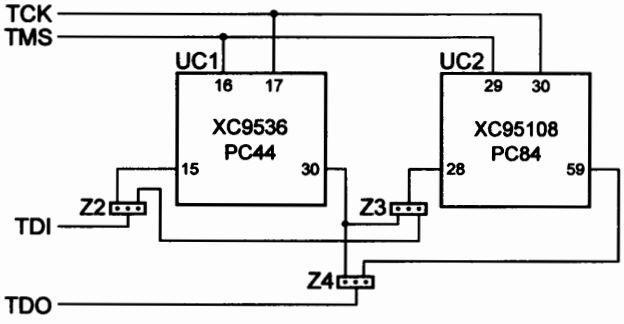


Rys. 5.2 Schemat projektu wykonanego w układzie UC1

wytwarza sygnał prostokątny o częstotliwości 14 Hz. Schemat projektu wykonanego w układzie UC1 pokazuje rys. 5.2.

Układ UC2 przeznaczony jest do testowania realizowanych projektów. Sygnały $P_0 \div P_7$ doprowadzone są do wybranych końcówek układu UC2 poprzez rezystory ograniczające prąd (w przypadku zaprogramowania końcówek jako wyjść). Do układu przyłączono osiem diod świecących $D_0 \div D_7$ oraz dwa wyświetlacze 7-segmentowe LED, które świecą po podaniu poziomu L na odpowiednie końcówki. Ponadto do układu UC2 doprowadzono z generatora Gen2 (NE555) sygnał zegarowy o częstotliwości wybieranej za pomocą zworki Z1.

Programowanie układów wykonuje się za pomocą komputera PC ze złączem JTAG. Sposób połączenia sygnałów złącza i wyboru programowanego układu za pomocą zworek Z2, Z3 i Z4 przedstawia rys. 5.3.



Programowany układ	Konfiguracja zworek Z2, Z3 i Z4
UC1	Z2 ▯ Z3 ■ Z4 ■
UC2	Z2 ■ Z3 ■ Z4 ▯
UC1 i UC2	Z2 ■ Z3 ▯ Z4 ■

Rys. 5.3 Konfigurowanie interfejsu JTAG na płytce testowej RAF1_01

5.4 PROCES PROJEKTOWANIA

Możliwości systemu projektowego *WebPACK ISE* ilustruje projekt licznika modulo 10, którego stan poddawany jest konwersji na kod 7-segmentowy i przesyłany do wyświetlacza LED. Opis licznika i konwertera jest przygotowany w języku VHDL. Po kompilacji i utworzeniu symboli tych elementów utworzono schemat i umieszczono projekt w układzie CPLD.

5.4.1 Uruchomienie systemu

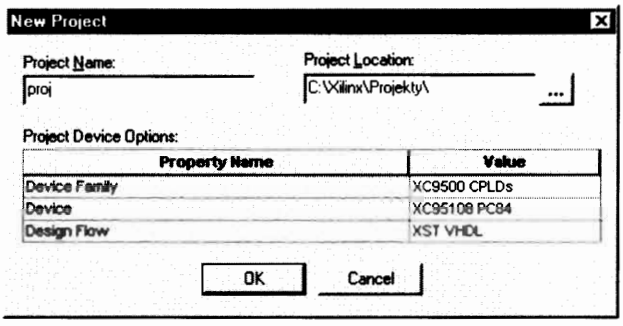
System *WebPACK ISE* uruchamia się dwukrotnym kliknięciem lewego klawisza myszki na ikonie:



Pojawia się okno *Project Navigator*, z którego wybieramy *File* → *New Project*. Pojawia się okno *New Project* (rys. 5.4). W polu *Project Location* wybieramy folder *C:\Xilinx\Projekty*, a w polu *Project Name* wpisujemy nazwę projektu *proj*. Wybieramy w polu *Device Family* rodzinę układów XC9500 CPLDs, układ typu XC95108 PC84 (*Device*) i projekt typu XST VHDL (*Design Flow*). Zatwierdzamy ustawienia przyciskiem *OK* powracając do okna *Project Navigator*.

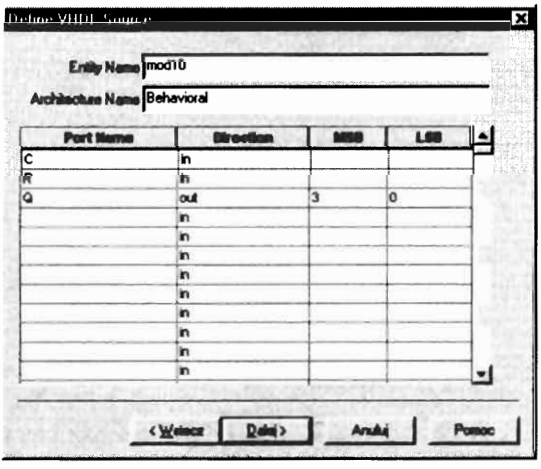
5.4.2 Model licznika modulo 10

Edytor tekstowy uruchamiamy w oknie *Project Navigator* wybierając *Project* → *New Source*. Pojawia się okno dialogowe *New*, w którym wybieramy typ pliku *VHDL Module*. W polu *File Name* wpisujemy nazwę



Rys. 5.4 Okno *New Project*

mod10. Domyślnie powinien być wybrany folder *C:/Xilinx/Projekty/proj*. Ustawienia zatwierdzamy przyciskiem *Dalej*. Następnie pojawia się okno *Define VHDL Source* (rys. 5.5). W kolumnie *Port Name* wpisujemy nazwy sygnałów C, R i Q, a w kolumnie *Direction* wybieramy kierunek ich przepływu (*in*, *out* lub *inout*). W kolumnach *MSB* i *LSB* ustawiamy liczbę bitów wektora Q. Po naciśnięciu *Dalej* pojawia się okno *New Source Information* informujące o wybranych nazwach sygnałów i modułu oraz ich typie. Po naciśnięciu *Zakończ* w oknie *Project Navigator* pojawia się szablon pliku *mod10.vhd*. Należy teraz



Rys. 5.5 Okno *Define VHDL Source*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity mod10 is
  port ( C,R : in std_logic;
           Q : out std_logic_vector(3 downto 0) );
end mod10;
architecture Behavioral of mod10 is
begin
  process(C,R)
    variable cnt : unsigned(3 downto 0);
  begin
    if R='1' then
      cnt:="0000";
    elsif rising_edge(C) then
      if cnt<9 then
        cnt:=cnt+1;
      else
        cnt:="0000";
      end if;
    end if;
    Q<=std_logic_vector(cnt);
  end process;
end Behavioral;
```

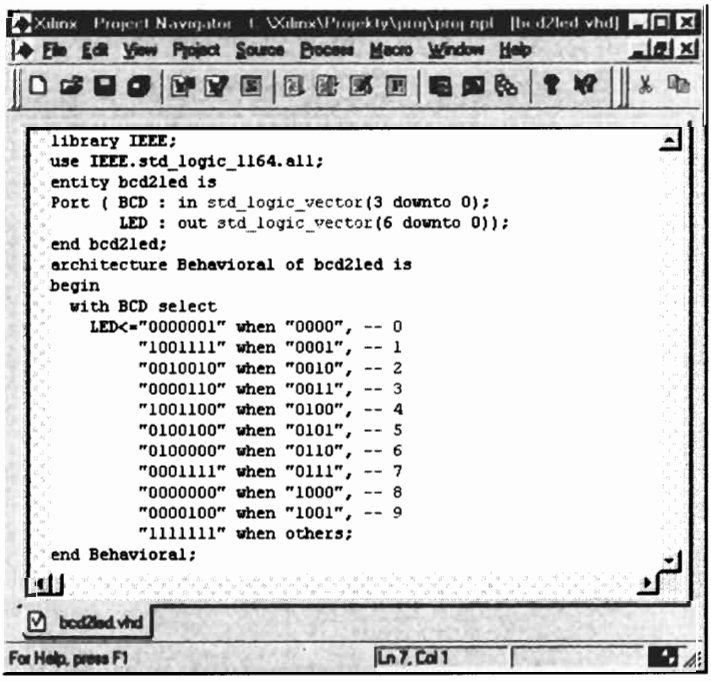
mod10.vhd

Rys. 5.6 Opis licznika modulo 10 (*mod10.vhd*)

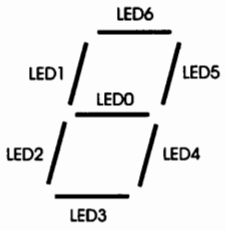
opisać działanie licznika uzupełniając brakujące wiersze utworzonego modułu w sposób pokazany na rys. 5.6. Następnie zapisujemy plik *mod10.vhd* (*File* → *Save*). W oknie *Project Navigator* zaznaczamy w obszarze *Sources in Project* moduł *mod10*. Poprawność opisu pod kątem składni języka VHDL sprawdzamy przy użyciu *Check Syntax* w procesie *Synthesize*. Tworzymy symbol graficzny, który użyjemy podczas rysowania schematu projektu. W tym celu w procesie *Design Entry Utilities* uruchamiamy *Create Schematic Symbol* dwukrotnym kliknięciem lewego klawisza myszy.

5.4.3 Model konwertera kodu BCD na kod wyświetlacza 7-segmentowego LED

Postępujemy podobnie jak przy opisie licznika modulo 10. Różnica polega na zastosowaniu nazwy modułu *bcd2led* oraz innych sygnałów wejściowych i wyjściowych. Model konwertera kodu BCD na kod wyświetlacza 7-segmentowego LED pokazuje rys. 5.7. Rysunek 5.8 przedstawia sposób połączenia odpowiednich bitów sygnału LED z segmentami wyświetlacza (aktywny jest poziom niski).



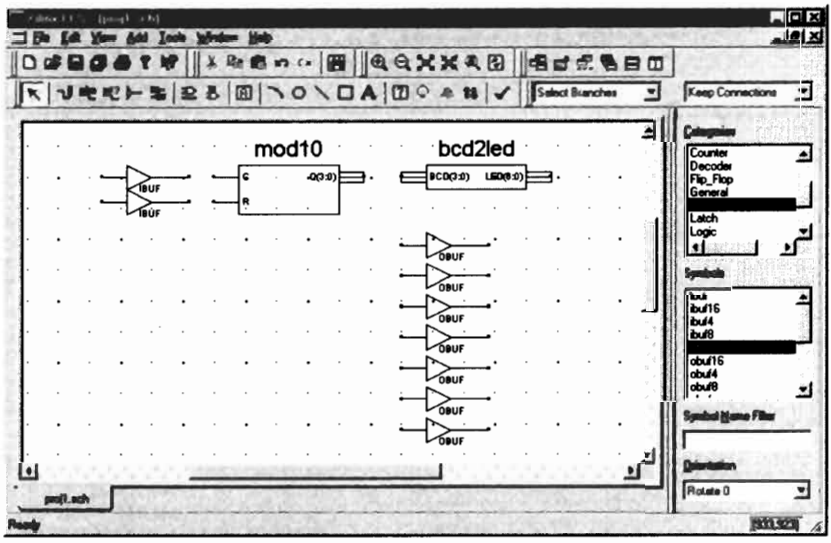
Rys. 5.7 Okno *Project Navigator* – opis w języku VHDL konwertera kodu BCD na kod wyświetlacza 7-segmentowego LED



Rys. 5.8 Segmenty wyświetlacza LED

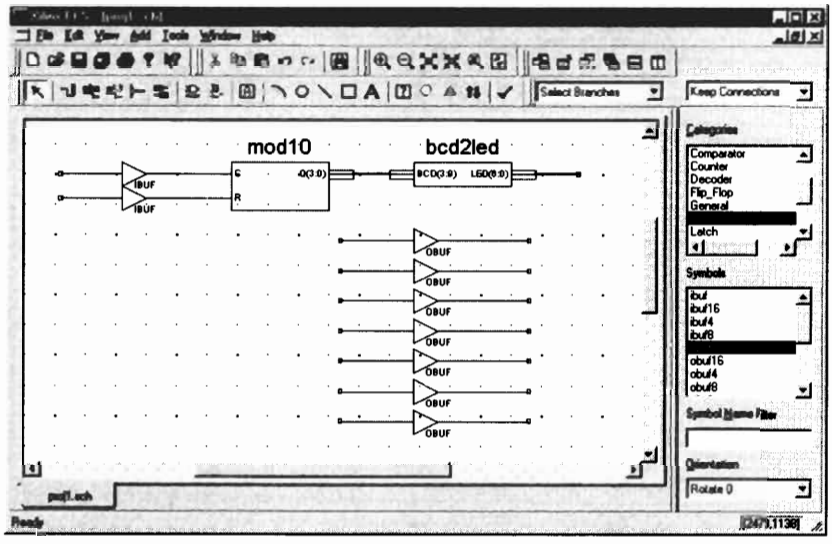
5.4.4 Opis projektu w postaci schematu

W oknie *Project Navigator* wybieramy *Project* → *New Source*. W oknie dialogowym *New* wybieramy typ *Schematic* oraz nadajemy nazwę *proj1*. Pojawia się okno edytora schematu *ECS* (*Engineering Capture System*). W obszarze *Categories* ustawiamy opcję *<All Symbols>*. Z obszaru *Symbols* pobieramy utworzone symbole *mod10* i *bcd2led*. Rozmieszczamy je na schemacie jak na rys. 5.9. Ponadto umieszczamy symbole buforów *IBUF* i *OBUF*. W kolejnym kroku wybieramy opcję *Add* → *Wire* i prowadzimy połączenia jak na rys. 5.10. W zależności od wybranej opcji *Select Branches* lub *Select Wires* można zaznaczyć sieć połączeń lub pojedyncze połączenie. Podczas przesuwania symbolu można zachować połączenia z pozostałymi symbolami po wybraniu opcji *Keep Connections* na pasku narzędzi.

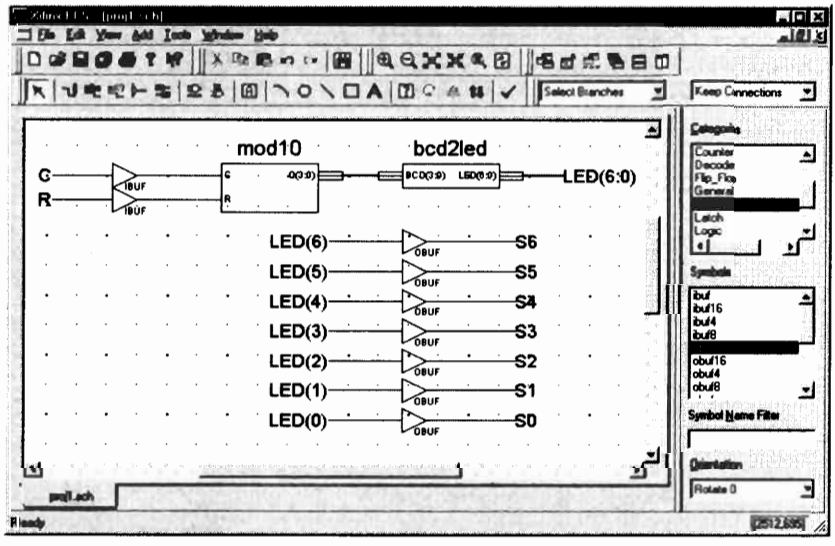


Rys. 5.9 Okno edytora schematu po umieszczeniu symboli *mod10*, *bcd2led*, *IBUF* i *OBUF*

Następnie wybieramy *Add* → *Net Name* i nadajemy nazwy poprowadzonym połączeniom jak na rys. 5.11. Ostatni krok to oznaczenie portów wejść i wyjść. W tym celu wybieramy *Add* → *I/O Marker*. Na pasku narzędziowym ustawiamy opcję *Input*. Porty C i R oznaczamy jako wejścia klikając lewym klawiszem myszy na początku połączenia. Pozostałe porty S_0+S_6 oznaczamy jako wyjścia. Zapisujemy schemat *File* → *Save* i zamykamy okno *ECS*.



Rys. 5.10 Okno edytora schematu po wykonaniu połączeń



Rys. 5.11 Okno edytora schematu po nadaniu nazw połączeniom

Następnie w obszarze *Processes for Current Source* rozwijamy *Synthesize*. Dwukrotnie klikamy lewym klawiszem myszy na *Analyze Hierarchy* uruchamiając proces, który tworzy plik VHDL (*.vhf*) ze schematu, a następnie kompiluje wszystkie moduły VHDL projektu.

5.4.5 Symulacja funkcjonalna

Dostępny w systemie *WebPACK ISE* symulator *ModelSim* działa opierając się na programie testowym. Aby go napisać tworzymy nowy moduł VHDL poprzez *Project → New Source* w oknie *Project Navigator*. Wybieramy moduł typu *VHDL Test Bench*, nadajemy nazwę *proj1tb* i naciskamy *Dalej*. W następnym oknie wybieramy moduł *proj1*, aby skojarzyć z nim program testowy. Po utworzeniu programu testowego *proj1tb.vhd* uzupełniamy jego brakujące wiersze zgodnie z modułem `L5.1`. Moduł ten opisuje test z tablicą wektorów oczekiwanych `odp_oczekiwana`, która zawiera stany logiczne wyświetlacza 7-segmentowego. Na początku testu układ jest zerowany, a następnie generowany jest sygnał zegarowy w każdym obiegu pętli. Dla danej liczby obiegów pętli na wyjściach powinien być odpowiedni stan logiczny wyświetlacza 7-segmentowego po czasie propagacji mniejszym od wartości 8 ns. Sygnały `OdpUkladu` i `OdpOczekiwana` wprowadzono do wizualizacji otrzymanego i oczekiwanego stanu logicznego wyświetlacza 7-segmentowego w postaci wektora. Sygnał `error` przyjmuje poziom wysoki w przypadku błędnej odpowiedzi testowanego układu.

```
-- L5.1
library IEEE;
use IEEE.std_logic_1164.all;
entity testbench is
end testbench;
architecture behavioral of testbench is
  component proj1
    port( C,R : in std_logic;
          S0,S1,S2,S3,S4,S5,S6 : out std_logic );
  end component;
  signal error : bit;
  signal C,R,S0,S1,S2,S3,S4,S5,S6 : std_logic;
  type tablica is array (0 to 9) of std_logic_vector
                                        (6 downto 0);
  constant odp_oczekiwana: tablica:=
  ("0000001",   --0
   "1001111",   --1
   "0010010",   --2
   "0000110",   --3
```

```
        "1001100",    --4
        "0100100",    --5
        "0100000",    --6
        "0001111",    --7
        "0000000",    --8
        "0000100" ); --9
    signal OdpUkladu,OdpOczekiwana: std_logic_vector
                                          (6 downto 0);
begin
    UUT: proj1 port map ( C=>C,R=>R,S0=>S0,S1=>S1,
                S2=>S2,S3=>S3,S4=>S4,S5=>S5,S6=>S6 );
    OdpUkladu<=S6&S5&S4&S3&S2&S1&S0;
    process
    begin
      C<='0'; R<='1'; error<='0';
      OdpOczekiwana<=odp_oczekiwana(0);
      wait for 100 ns;
      if OdpUkladu/=odp_oczekiwana(0) then     -- test
        error<='1';
        assert false report "Bledna odpowiedz ukladu";
      else error<='0';
      end if;
      error<='0';
      R<='0';
      wait for 50 ns;
      for i in 1 to 9 loop
        C<='1'; OdpOczekiwana<=odp_oczekiwana(i);
        wait for 8 ns;  -- czas propagacji 8 ns;
        if OdpUkladu/=odp_oczekiwana(i) then   -- test
          error<='1';
          assert false
              report "Bledna odpowiedz ukladu";
        else error<='0';
        end if;
        wait for 8 ns;
        C<='0'; wait for 10 ns;
        error<='0';
      end loop;
      wait;
    end process;
end;
```

Aby uruchomić symulator *ModelSim* zaznaczamy moduł *proj1tb* w oknie *Project Navigator* w obszarze *Sources in Project*. W obszarze *Processes for Current Source*, rozwijamy *ModelSim Simulator* i dwukrotnie klikamy lewym klawiszem myszy na *Simulate Behavioral VHDL Model*. Wynik symulacji funkcjonalnej projektu ma postać komunikatów tekstowych. W postaci graficznej przebiegi sygnałów można

zaobserwować w oknie *Wave*. Sygnał `error` ma poziom niski, co świadczy o poprawności opisu działania układu.

5.4.6 Synteza i implementacja projektu

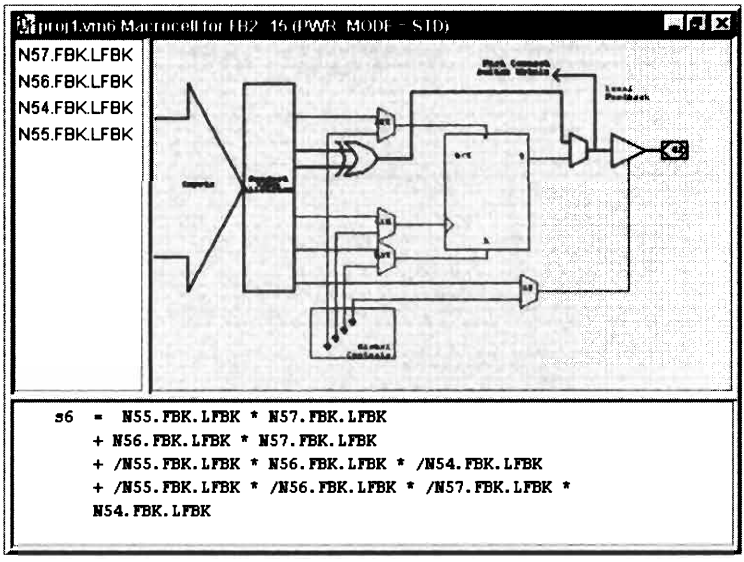
Następnym etapem procesu projektowania jest synteza. Rozpoczynamy od zaznaczenia w oknie *Project Navigator* w obszarze *Sources in Project* modułu *proj1*. Następnie w obszarze *Processes for Current Source* dwukrotnie klikamy lewym klawiszem myszy na *Synthesize* uruchamiając aplikację, która tworzy binarny opis projektu w postaci bramek i przerzutników.

Przypisanie końcówek układu scalonego do portów projektu dokonujemy używając aplikacji *Constraints Editor*. W procesie *Design Entry Utilities* rozwijamy *User Constraints*. Okno *Constraints Editor* otwieramy dwukrotnym kliknięciem lewego klawisza myszy na *Edit Implementation Constraints (Constraints Editor)*. Następnie wybieramy zakładkę *Ports*. Porty projektu przypisujemy wpisując w kolumnie *Location* numer końcówki układu scalonego poprzedzony literą P (rys. 5.12). Po zamknięciu okna *Constraints Editor* pojawia się komunikat *Notice*, w którym wybieramy *Reset* do wprowadzenia zmian do pliku *proj1.ucf* zawierającego informację o przypisaniu końcówek, potrzebną w następnych etapach procesu projektowania.

W kolejnym kroku klikając dwukrotnie na *Implement Design* uruchamiamy dwa procesy. Pierwszy *Translate* tworzy binarny opis projektu z uwzględnieniem ograniczeń zawartych w pliku *proj1.ucf* – w tym przypadku przypisania portów do końcówek układu scalonego. Następnie proces *Fit* wpasowuje projekt w układ CPLD. Z każdego etapu generowany jest raport.

Port Name	Port Direction	Location	Pad to Setup	Clock to Pad
c	INPUT	P9	N/A	N/A
r	INPUT	P40		N/A
s0	OUTPUT	P75	N/A	
s1	OUTPUT	P84	N/A	
s2	OUTPUT	P79	N/A	
s3	OUTPUT	P80	N/A	
s4	OUTPUT	P72	N/A	
s5	OUTPUT	P81	N/A	
s6	OUTPUT	P82	N/A	

Rys. 5.12 Przypisanie końcówek do portów projektu



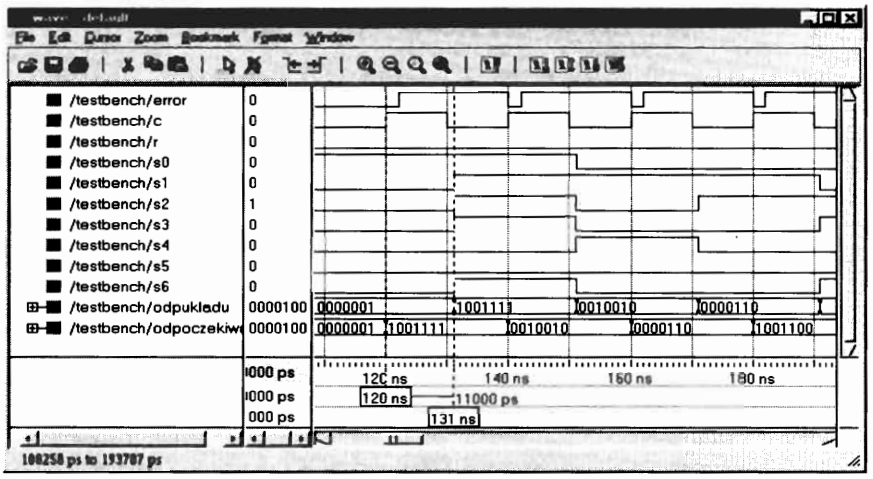
Rys. 5.13 Widok makrokomórki FB2_15

Pozostałe procesy nie są wykonywane automatycznie. *Generate Timing* tworzy raport o czasach propagacji od sygnałów wejściowych do wyjściowych. Uruchamiając *Analyze Post-Fit Static Timing* możemy określić czasy propagacji wybranych połączeń wewnątrz układu i znaleźć krytyczne połączenia ograniczające szybkość działania projektu. Proces *Generate Post-Fit Simulation Model* nie jest konieczny do przeprowadzenia, ponieważ jest uruchamiany automatycznie podczas symulacji czasowej. *Generate IBIS Model* tworzy model elektryczny zastosowanych w projekcie buforów wejściowych i wyjściowych.

Graficzne przedstawienie rozmieszczenia projektu w układzie programowalnym umożliwia aplikacja *ChipView* uruchamiana poprzez *View Fitted Design*. Ponadto za jej pomocą można obejrzeć makrokomórki z opisem realizowanych w nich funkcji logicznych (rys. 5.13).

5.4.7 Symulacja czasowa

Symulację czasową można przeprowadzić dopiero po implementacji, ponieważ wówczas otrzymujemy model czasowy układu. Do symulacji użyjemy program testowy *proj1tb.vhd* (`L5.1`). Symulator *ModelSim* uruchamiamy podobnie jak podczas symulacji funkcjonalnej z tą różnicą, że dwukrotnie klikamy lewym klawiszem myszy na *Simulate Post-Fit VHDL Model*. Obserwując w oknie *Wave* przebiegi sygnałów zauważamy, że sygnał `error` przyjmuje poziom wysoki (rys. 5.14).

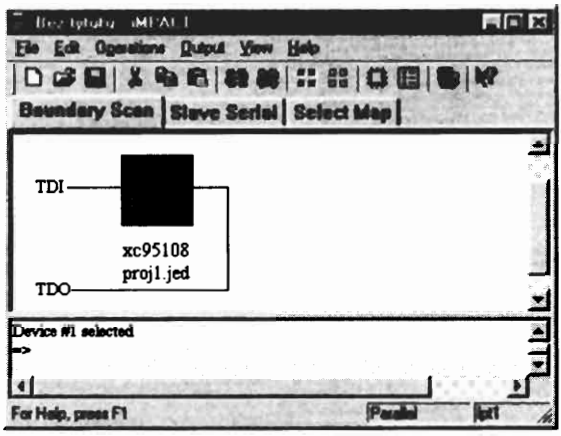


Rys. 5.14 Wynik symulacji czasowej – okno *Wave*

Świadczy to o błędnym działaniu naszego projektu. Mierząc czas, po którym testowany układ odpowiada na zmianę sygnału zegarowego `C` z poziomu L na H, otrzymujemy wynik 11 ns. Przyjmowanie poziomu wysokiego przez sygnał `error` jest więc wynikiem zbyt wczesnego porównywania odpowiedzi oczekiwanej z odpowiedzią testowanego układu. Należy zwiększyć czas propagacji do wartości 12 ns, czyli **`wait for`** `12 ns`. Przeprowadzając ponownie symulację czasową otrzymuje się wynik pozytywny.

5.4.8 Programowanie układu

Przed programowaniem układu należy utworzyć plik dla programatora. W tym celu zaznaczamy w obszarze *Sources in Project* moduł *proj1* i uruchamiamy proces *Generate Programming File*. Następnie uruchamiamy aplikację programatora poprzez *Configure Device (iMPACT)*. Pojawia się okno *iMPACT* jak na rys. 5.15. Po poprawnym dołączeniu przewodu programującego do układu scalonego i komputera PC, wybieramy *Operations* → *Program*. Pojawia się okno *Program Options*, w którym ustawiamy aktywną opcję *Erase Before Programing*. Zatwierdzamy ustawienia poprzez *OK*. Po zaprogramowaniu układu zamykamy okno *iMPACT*.



Rys. 5.15 Okno *iMPACT*

5.5 PRZYKŁADY PROJEKTÓW

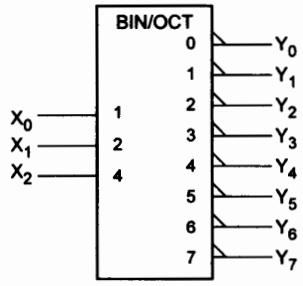
5.5.1 Konwerter 3-bitowego kodu NB na kod „1 z 8"

Należy zaprojektować konwerter 3-bitowego kodu NB na kod „1 z 8" (rys. 5.16). Sygnał wyjściowy ma być aktywny na poziomie L. Tablica 5.3 pokazuje zależność sygnału wyjściowego `Y` od wejściowego `X`.

Tablica 5.3 Tablica stanów konwertera

Sygnał wejściowy X[2:0]	Sygnał wyjściowy Y[7:0]
000	11111110
001	11111101
010	11111011
011	11110111
100	11101111
101	11011111
110	10111111
111	01111111

Język VHDL umożliwia opisanie takiego konwertera wieloma sposobami. Moduł `L5.2` przedstawia opis z zastosowaniem stylów przepływowego i behawioralnego.



Rys. 5.16 Konwerter 3-bitowego kodu NB na kod „1 z 8"

```
-- L5.2
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity konw is
  port( X: in  std_logic_vector(2 downto 0);
        Y: out std_logic_vector(7 downto 0) );
end;
architecture beh1 of konw is  -- styl przeplywowy
begin
    Y <= "11111110" when X="000" else
         "11111101" when X="001" else
         "11111011" when X="010" else
         "11110111" when X="011" else
         "11101111" when X="100" else
         "11011111" when X="101" else
         "10111111" when X="110" else
         "01111111";
end beh1;
architecture beh2 of konw is  -- styl przeplywowy
begin
    with X select
      Y <= "11111110" when "000",
           "11111101" when "001",
           "11111011" when "010",
           "11110111" when "011",
           "11101111" when "100",
           "11011111" when "101",
           "10111111" when "110",
           "01111111" when others;
end beh2;
architecture beh3 of konw is  -- styl behawioralny
begin
    process(X)
    begin
      case X is
        when "000" => Y <= "11111110";
        when "001" => Y <= "11111101";
```

```
            when "010" => Y <= "11111011";
            when "011" => Y <= "11110111";
            when "100" => Y <= "11101111";
            when "101" => Y <= "11011111";
            when "110" => Y <= "10111111";
            when others => Y <= "01111111";
          end case;
      end process;
end beh3;
architecture beh4 of konw is  -- styl behawioralny
begin
      process(X)
      begin
        Y<=(others=>'1');
        Y(TO_INTEGER(unsigned(X)))<='0';
      end process;
end beh4;
architecture beh5 of konw is  -- styl behawioralny
begin
    process(X)
    begin
      for i in 0 to 5 loop      -- tu wprowadzono blad
        if i=unsigned(X) then
          Y(i)<='0';
        else
          Y(i)<='1';
        end if;
      end loop;
    end process;
end beh5;
```

Ponieważ w architekturze `beh1` warunki są sprawdzane kolejno, więc rezultatem takiego zapisu może być układ o architekturze szeregowej, o długim czasie opóźnienia. Lepiej więc stosować współbieżną instrukcję wyboru jak w `beh2`. Analogiczny opis można utworzyć wewnątrz procesu jak w architekturze `beh3`. Funkcja `TO_INTEGER` z pakietu `numeric_std` pozwala na konwersję typu `unsigned` na typ `integer`. Użycie tej funkcji umożliwia sformułowanie bardzo krótkiego opisu w stylu `beh4`. W pierwszym wierszu procesu przypisywany jest stan 1 wszystkim bitom sygnału `Y`. W kolejnym wierszu stan 0 przypisywany jest wybranemu bitowy sygnału `Y`, zależnie od wartości sygnału `X`. Styl `beh5` jest opisem powtarzającym w pętli instrukcję warunkową **if-then-else**. Gdy warunek jest spełniony, to następuje przypisanie stanu 0 określonemu bitowi sygnału `Y`. Pozostałym bitom przypisywany jest stan 1.

Styl opisu układu cyfrowego ma znaczenie w procesie kompilacji i syntezy. System projektowy *WebPACK ISE* zawiera kompilator języka

VHDL *XST* firmy *Xilinx*. Liczbę bloków logicznych potrzebnych do realizacji konwertera z użyciem wymienionych architektur po syntezie przedstawia tablica 5.4. Do dalszej realizacji projektu wybrano architekturę beh5.

Tablica 5.4 Liczba bloków logicznych po syntezie

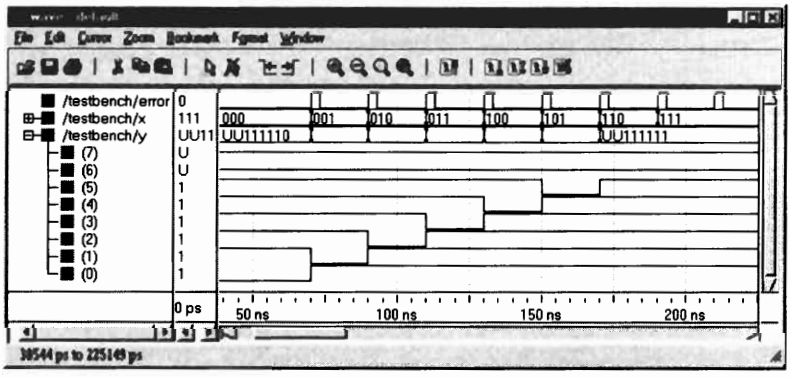
Architektura	Liczba bloków logicznych
beh1	34
beh2	34
beh3	34
beh4	23
beh5	23

Program testowy L5.3 opisuje test projektu z sekwencją sygnału wejściowego X generowaną w pętli. Indeks pętli typu integer zamieniany jest na typ std_logic_vector i przypisany sygnałowi X. Odpowiedź układu (sygnał Y) porównywana jest z elementami tablicy test_vectors. Sygnał error przyjmuje stan 1 w przypadku błędnej odpowiedzi konwertera.

Po wykonaniu symulacji funkcjonalnej okazało się, że sygnał error ma stan 1 (rys. 5.17). Sygnały Y(6) i Y(7) mają stan U. Analizując moduły L5.2 i L5.3 nie znajdujemy przypisania tym sygnałom stanu U. Należy zwrócić uwagę na to, że stan U jest pierwszy na liście definiującej typ std_logic (p. str. 27). Ponieważ sygnały Y(6) i Y(7) są typu std_logic, zatem gdy nie są im przypisywane żadne wartości, to stanem domyślnym jest stan U. Rzeczywiście, celowo wprowadzony błąd w architekturze beh5 powoduje taki skutek. Pętla dokonuje sześciu obiegów zamiast ośmiu powodując, że sygnały Y(6) i Y(7) są pomijane. Po zwiększając liczby obiegów pętli do ośmiu (**for** i **in** 0 **to** 7 **loop**) uzyskuje się poprawny wynik symulacji.

```
-- L5.3
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity testbench is
end testbench;
architecture behavior of testbench is
    component konw
      port( X : in std_logic_vector(2 downto 0);
            Y : out std_logic_vector(7 downto 0) );
    end component;
    signal error : std_logic;
```

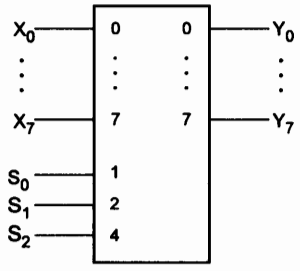
```
    signal X : std_logic_vector(2 downto 0);
    signal Y : std_logic_vector(7 downto 0);
    type test_vector_array is array (natural range <>)
                        of std_logic_vector(7 downto 0);
    constant test_vectors: test_vector_array:=
    ( "11111110",    -- 0
      "11111101",    -- 1
      "11111011",    -- 2
      "11110111",    -- 3
      "11101111",    -- 4
      "11011111",    -- 5
      "10111111",    -- 6
      "01111111" ); -- 7
begin
    UUT : konw port map ( X=>X, Y=>Y );
    process
    begin
      error<='0';
      X<="000";
      wait for 50 ns;
      for i in 0 to 7 loop
        X<=std_logic_vector(TO_UNSIGNED(i,3));
        wait for 20 ns;
        if Y/=test_vectors(i) then      -- test
          error<='1','0' after 3 ns;
          assert false
            report "Bledna odpowiedz";
        else
          error<='0';
        end if;
      end loop;
      wait;
    end process;
end behavior;
```



Rys. 5.17 Przykładowe wyniki symulacji funkcjonalnej

5.5.2 Rotator

Należy zaprojektować układ kombinacyjny obracający w lewo 8-bitowy sygnał wejściowy o zadaną liczbę bitów (rys. 5.18). W języku angielskim układ ten zwany jest *barrel shifter* („przesuwnik beczkowy"). Zależność sygnału wyjściowego `Y` od sygnału sterującego `S` pokazuje tablica 5.5.



Rys. 5.18 Rotator

Tablica 5.5 Tryby obrotu

Sygnał sterujący S[2:0]	Sygnał wyjściowy Y[7:0]
0 0 0	$X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$
0 0 1	$X_6 X_5 X_4 X_3 X_2 X_1 X_0 X_7$
0 1 0	$X_5 X_4 X_3 X_2 X_1 X_0 X_7 X_6$
0 1 1	$X_4 X_3 X_2 X_1 X_0 X_7 X_6 X_5$
1 0 0	$X_3 X_2 X_1 X_0 X_7 X_6 X_5 X_4$
1 0 1	$X_2 X_1 X_0 X_7 X_6 X_5 X_4 X_3$
1 1 0	$X_1 X_0 X_7 X_6 X_5 X_4 X_3 X_2$
1 1 1	$X_0 X_7 X_6 X_5 X_4 X_3 X_2 X_1$

W pakiecie `standard` jest zdefiniowanych sześć operatorów przesunięć, których zestawienie zamieszczono w tablicy 5.6. Są to działania na wektorach typu `bit_vector`. Lewy operand to dany wektor, a prawy to liczba pozycji do przesunięcia.

Tablica 5.6 Operatory przesunięć

Operator	Opis działania
`sll`	przesuw logiczny w lewo
`srl`	przesuw logiczny w prawo
`sla`	przesuw arytmetyczny w lewo
`sra`	przesuw arytmetyczny w prawo
`rol`	obrót logiczny w lewo
`ror`	obrót logiczny w prawo

Do realizacji projektu potrzebny jest operator `rol` (`L5.4`). Wykorzystując funkcje `To_StdLogicVector` i `To_BitVector` (zawarte w pakiecie `std_logic_1164`) realizuje się konwersje między typami `bit_vector` i `std_logic_vector`.

```
-- L5.4
library IEEE;
use IEEE.std_logic_1164.all;
entity przes is
  port( X : in std_logic_vector(7 downto 0);
        S : in std_logic_vector(2 downto 0);
        Y : out std_logic_vector(7 downto 0) );
end przes;
architecture beh of przes is
  signal Y_bitv : bit_vector(7 downto 0);
begin
  with S select
    Y_bitv <= To_BitVector(X) when "000",
              To_BitVector(X) rol 1 when "001",
              To_BitVector(X) rol 2 when "010",
              To_BitVector(X) rol 3 when "011",
              To_BitVector(X) rol 4 when "100",
              To_BitVector(X) rol 5 when "101",
              To_BitVector(X) rol 6 when "110",
              To_BitVector(X) rol 7 when others;
  Y<=To_StdLogicVector(Y_bitv);
end beh;
```

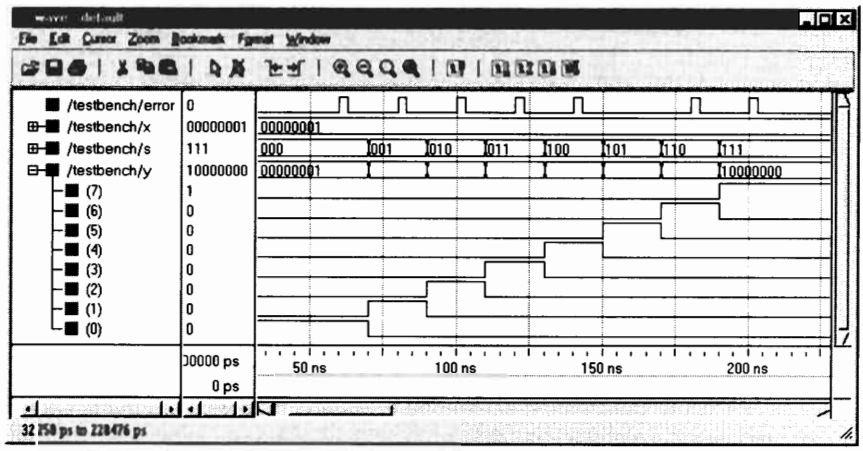
Program testowy `L5.5` wykorzystuje sekwencję sygnału sterującego `S` generowaną w pętli. Indeks pętli typu `integer` zamieniany jest na typ `std_logic_vector` i przypisywany sygnałowi `S`. Sygnałowi wejściowemu `Y` przypisano stałą wartość `"00000001"` zapisaną w postaci szesnastkowej (`X"01"`). Program testowy przesuwa tę wartość i porównuje z odpowiedzią testowanego układu. Przeprowadzając symulację funkcjonalną otrzymano wyniki jak na rys. 5.19. Błąd nie występuje tylko dla `S="101"`. W programie testowym celowo wprowadzono błędny warunek porównania odpowiedzi układu z wartością poprawną. Program testowy przesuwa wartość `X"01"` zawsze o pięć pozycji, a powinien przesuwać o liczbę pozycji równą indeksowi pętli. Poprawny zapis warunku testu jest następujący:

```
if (To_BitVector(Y)/=To_BitVector(X) rol k) then
```

Przeprowadzając ponownie symulację otrzymujemy wynik pozytywny.

```
-- L5.5
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.numeric_std.all;
entity testbench is
end testbench;
architecture behavior of testbench is
   component przes
      port( X : in std_logic_vector(7 downto 0);
            S : in std_logic_vector(2 downto 0);
            Y : out std_logic_vector(7 downto 0) );
   end component;
   signal error : std_logic;
   signal X,Y : std_logic_vector(7 downto 0);
   signal S : std_logic_vector(2 downto 0);
begin
   UUT: przes port map ( X=>X, Y=>Y, S=>S );
   process
   begin
      X<=X"01"; error<='0'; S<="000";
      wait for 50 ns;
      for k in 0 to 7 loop
         S<=std_logic_vector(TO_UNSIGNED(k,3));
         wait for 10 ns;
         -- test odpowiedzi UUT - wprowadzono blad
         if (To_BitVector(Y)/=To_BitVector(X) rol 5)
          then
            error<='1','0' after 3 ns;
            assert false report "Bledna odpowiedz UUT";
         else error<='0';
         end if;
         wait for 10 ns;
      end loop;
      wait;
   end process;
end behavior;
```



Rys. 5.19 Wynik symulacji funkcjonalnej

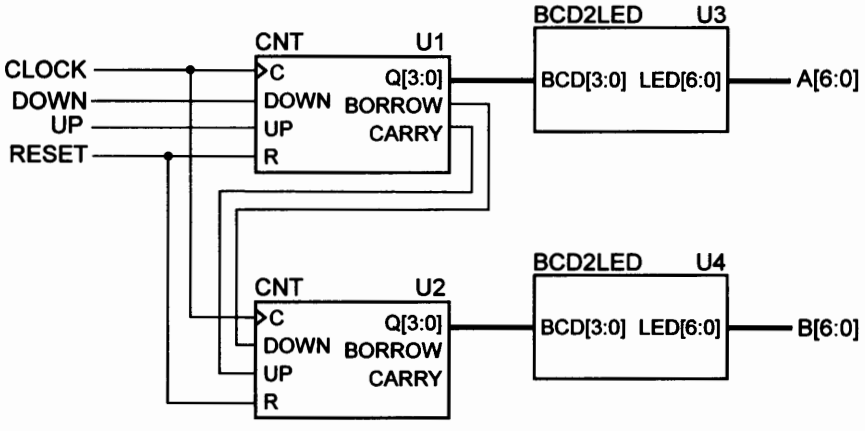
5.5.3 Licznik modulo 100 w kodzie BCD

Należy zaprojektować układ synchronicznego, rewersyjnego licznika modulo 100 w kodzie BCD. Stan licznika ma być przedstawiony na wyświetlaczu 7-segmentowym LED. Licznik ma cztery wejścia: `UP`, `DOWN`, `CLOCK` i `RESET`. Wejścia `UP` i `DOWN` określają tryby pracy zdefiniowane w tablicy 5.7. Wejście `CLOCK` jest wejściem zegarowym. Wejście `RESET` jest asynchronicznym wejściem zerującym. Schemat blokowy licznika przedstawia rys. 5.20.

Tablica 5.7 Tryby pracy licznika

UP	DOWN	Q_{n+1}
0	0	Q_n
0	1	Q_n-1
1	0	Q_n+1
1	1	Q_n

Do konwersji kodu BCD na kod wyświetlacza 7-segmentowego LED użyto procedury `BCD2LED` zawartej w utworzonym przez autora pakiecie `rafszym` (`L5.6`). Jako licznik **CNT** zastosowano równoległy licznik modulo 10 z asynchronicznym zerowaniem (`L5.7`). Użycie pakietu `numeric_std` z biblioteki `IEEE` pozwoliło na inkrementację i dekrementację zmiennej lokalnej `Q_int` typu `unsigned`. Zmienna po konwersji na typ `std_logic_vector` jest przypisywana do sygnału `Q`.



Rys. 5.20 Schemat blokowy licznika modulo 100

```
-- L5.6
-- moj pakiet "rafszym"
library IEEE;
use IEEE.std_logic_1164.all;
package rafszym is
   -- procedura dla wyswietlacza 7-segmentowego
   procedure BCD2LED
      (signal BCD: in std_logic_vector(3 downto 0);
       signal LED: out std_logic_vector(6 downto 0) );
end rafszym;
package body rafszym is
   procedure BCD2LED
      (signal BCD: in std_logic_vector(3 downto 0);
       signal LED: out std_logic_vector(6 downto 0)) is
   begin
       case BCD is
          when "0000" => LED<="0000001"; -- 0
          when "0001" => LED<="1001111"; -- 1
          when "0010" => LED<="0010010"; -- 2
          when "0011" => LED<="0000110"; -- 3
          when "0100" => LED<="1001100"; -- 4
          when "0101" => LED<="0100100"; -- 5
          when "0110" => LED<="0100000"; -- 6
          when "0111" => LED<="0001111"; -- 7
          when "1000" => LED<="0000000"; -- 8
          when "1001" => LED<="0000100"; -- 9
          when others => LED<="1111111";
       end case;
   end BCD2LED;
end rafszym;
```

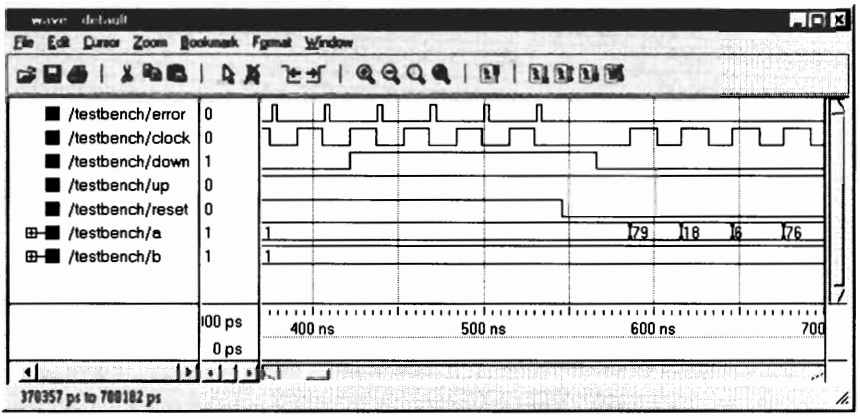
Opis licznika modulo 10 jest następujący:

```
-- L5.7
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity CNT is
   port( UP,DOWN,C,R: in std_logic;
         Q: inout std_logic_vector(3 downto 0);
         BORROW,CARRY: out std_logic );
end;
architecture beh of CNT is
   signal F: std_logic_vector(1 downto 0);
begin
    F<=UP&DOWN;
    process(C,R)
      variable Q_int: unsigned(3 downto 0);
    begin
      if R='1' then
```

```
            Q_int:="0000";
          elsif rising_edge(C) then
            case F is
              when "10" => if Q_int<"1001" then
                             Q_int:=Q_int+1;
                           else
                             Q_int:="0000";
                           end if;
              when "01" => if Q_int>"0000" then
                             Q_int:=Q_int-1;
                           else
                             Q_int:="1001";
                           end if;
              when others => Q_int:=Q_int;
            end case;
          end if;
          Q<=std_logic_vector(Q_int);
      end process;
      CARRY<='1' when (Q="1001" and F="10") else '0';
      BORROW<='1' when (Q="0000" and F="01") else '0';
end;
```

Opis strukturalny projektowanego licznika z wykorzystaniem opisanej jednostki projektowej `L5.7` i procedury `BCD2LED` przedstawia moduł `L5.8`. Nie wykorzystane wyjścia licznika **U2** pozostawiamy nie połączone za pomocą słowa kluczowego **open**.

```
-- L5.8
library IEEE;
use IEEE.std_logic_1164.all;
use work.rafszym.all;
entity top is
  port( CLOCK,DOWN,UP,RESET: in std_logic;
        A,B: out std_logic_vector(6 downto 0) );
end;
architecture top_arch of top is
    component CNT
      port( UP,DOWN,C,R: in std_logic;
            Q: inout std_logic_vector(3 downto 0);
            BORROW,CARRY: out std_logic );
    end component;
    signal qa,qb: std_logic_vector(3 downto 0);
    signal x0,x1: std_logic;
begin
    U1: CNT port map (UP,DOWN,CLOCK,RESET,qa,x1,x0);
    U2: CNT port map (x0,x1,CLOCK,RESET,qb,open,open);
    -- procedury wyswietlacza
    BCD2LED(qa,A);  -- U3
    BCD2LED(qb,B);  -- U4
end;
```



Rys. 5.21 Wynik symulacji funkcjonalnej

Moduł L5.9 opisuje test licznika z tablicą wektorów oczekiwanych test_vectors, którymi są stany logiczne wyświetlacza 7-segmentowego. Sygnał zegarowy CLOCK jest generowany w każdym obiegu pętli. Na wyjściach A i B powinny być odpowiednie stany logiczne wyświetlacza 7-segmentowego, zależnie od liczby zliczonych impulsów zegarowych. Sygnał wyjściowy error przyjmuje stan 1 w przypadku błędnej odpowiedzi testowanego układu licznika. Przykładowe wyniki symulacji funkcjonalnej przedstawiono na rysunku 5.21, z których wynika błędne działa układu podczas zerowania. Na wyjściach A i B są poprawne stany logiczne wyświetlacza 7-segmentowego dla wartości zero. Błąd popełniono w programie testowym w części testowania sygnału RESET. Poprawny zapis testu jest następujący:

```
if A/=test_vectors(0) or B/=test_vectors(0) then
```

Przeprowadzając ponownie symulację otrzymano wynik pozytywny.

```
-- L5.9
library IEEE;
use IEEE.std_logic_1164.all;
entity testbench is
end testbench;
architecture behavior of testbench is
    component top
      port( CLOCK,DOWN,UP,RESET : in std_logic;
            A,B : out std_logic_vector(6 downto 0) );
    end component;
    signal error : std_logic;
    signal CLOCK,DOWN,UP,RESET : std_logic;
    signal A,B : std_logic_vector(6 downto 0);
    type test_vector_array is array (natural range <>)
                    of std_logic_vector(6 downto 0);
```

```
    constant test_vectors: test_vector_array:=
    ( "0000001",    -- 0
      "1001111",    -- 1
      "0010010",    -- 2
      "0000110",    -- 3
      "1001100",    -- 4
      "0100100",    -- 5
      "0100000",    -- 6
      "0001111",    -- 7
      "0000000",    -- 8
      "0000100",    -- 9
      "0000001" ); -- dodatkowy kod cyfry 0
begin
  UUT: top port map (CLOCK=>CLOCK,DOWN=>DOWN,UP=>UP,
                     RESET=>RESET,A=>A,B=>B);
  process
  begin
    -- test sygnalu RESET
    error<='0';
    RESET<='1';
    wait for 50 ns;
    for i in 0 to 3 loop  -- liczba obiegow dowolna
      case i is
        when 0 => UP<='0'; DOWN<='0';
        when 1 => UP<='0'; DOWN<='1';
        when 2 => UP<='1'; DOWN<='0';
        when 3 => UP<='1'; DOWN<='1';
      end case;
      for j in 0 to 3 loop  -- liczba obiegow dowolna
        CLOCK<='1','0' after 15 ns;--impuls zegarowy
        wait for 16 ns;
        -- test odpowiedzi UUT - tu wprowadzono blad
        if A/=test_vectors(0) or B/=test_vectors(1)
        then
          error<='1','0' after 3 ns;
          assert false
            report "Bledna odpowiedz ukladu UUT";
        else
          error<='0';
        end if;
        wait for 15 ns;
      end loop;
    end loop;
    RESET<='0';
    wait for 20 ns;
    -- test liczenia w gore
    error<='0';
    UP<='1'; DOWN<='0';
    wait for 20 ns;
    for i in 0 to 9 loop
      for j in 1 to 10 loop
```

```
            CLOCK<='1','0' after 10 ns;--impuls zegarowy
            wait for 16 ns;
            -- test odpowiedzi UUT
            if A/=test_vectors(j)
                or B/=test_vectors(i+j/10) then
              error<='1','0' after 3 ns;
              assert false
                report "Bledna odpowiedz ukladu UUT";
            else
              error<='0';
            end if;
            wait for 15 ns;
          end loop;
        end loop;
        -- test liczenia w dol
        error<='0';UP<='0';DOWN<='1';wait for 20 ns;
        for i in 9 downto 0 loop
          for j in 9 downto 0 loop
            CLOCK<='1','0' after 15 ns;--impuls zegarowy
            wait for 16 ns;
            -- test odpowiedzi UUT
            if A/=test_vectors(j) or B/=test_vectors(i)
            then
              error<='1','0' after 6 ns;
              assert false
                report "Bledna odpowiedz ukladu UUT";
            else
              error<='0';
            end if;
            wait for 15 ns;
          end loop;
        end loop;
        wait;
      end process;
    end behavior;
```

5.5.4 Blok liczników równoległych

Należy zaprojektować układ zbudowany z czterech liczników: modulo 256, Johnsona oraz dwóch pierścieniowych z „krążącą jedynką" i „krążącym zerem". Schemat układu przedstawiono na rys. 5.22. Za pomocą sygnału SEL wybierany jest jeden z liczników, którego wyjścia będą podawane na wyjście układu Y. Sygnały EN1, EN2, EN3 i EN4 sterują procesem liczenia odpowiednich liczników. Wejście RESET jest asynchronicznym wejściem zerującym.

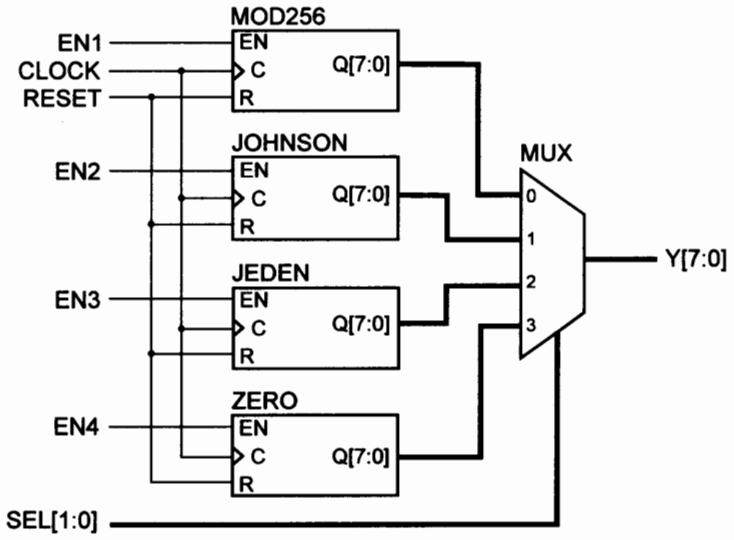
Multiplekser grupowy MUX można opisać w języku VHDL z użyciem instrukcji przypisania selektywnego lub warunkowego. Moduł L5.10

opisuje multiplekser z zastosowaniem współbieżnego przypisania warunkowego:

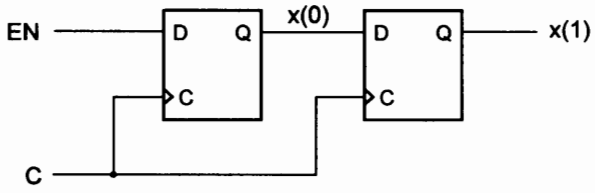
```
-- L5.10
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX is
  port( A,B,C,D: in std_logic_vector(7 downto 0);
        SEL: in std_logic_vector(1 downto 0);
        Y: out std_logic_vector(7 downto 0) );
end;
architecture beh of MUX is
begin
     Y <=  A when SEL="00" else
           B when SEL="01" else
           C when SEL="10" else
           D;
end;
```

Dalej zamieszczono moduły opisujące działanie liczników. W celu eliminacji zjawiska metastabilności, zastosowano synchronizację sygnału EN jak na rysunku 5.23. Jest to klasyczny przykład rejestru przesuwającego z wejściem i wyjściem szeregowym. Do opisu przesuwu n-bitowego wektora Q w języku VHDL można zastosować następujące przypisania:

```
Q<=Q(n-1 downto 0)&X;   -- przesuw w lewo
Q<=X&Q(n downto 1);     -- przesuw w prawo
```



Rys. 5.22 Schemat blokowy układu z czterema licznikami



Rys. 5.23 Schemat synchronizatora

Sygnał X jest wejściem szeregowym. Wyjściem szeregowym jest bit Q(n) lub Q(0) zależnie od kierunku przesuwu. Wyjściem równoległym jest wektor Q.

Pierwszy moduł opisuje licznik modulo 256:

```
-- L5.11
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity MOD256 is
  port( C,EN,R: in std_logic;
        Q: out std_logic_vector(7 downto 0) );
end;
architecture beh of MOD256 is
  signal x: std_logic_vector(1 downto 0);
begin
  process(C,R)
    variable Q_int: unsigned(7 downto 0);
  begin
    if R='1' then
      Q_int:=(others=>'0');
    elsif rising_edge(C) then
      x<=x(0)&EN;  -- synchronizator
      if x(1)='1' then
        Q_int:=Q_int+1;
      end if;
    end if;
    Q<=std_logic_vector(Q_int);
  end process;
end;
```

Drugi moduł opisuje licznik Johnsona:

```
-- L5.12
library IEEE;
use IEEE.std_logic_1164.all;
entity JOHNSON is
  port( C,EN,R: in std_logic;
        Q: inout std_logic_vector(7 downto 0) );
end;
```

```
architecture beh of JOHNSON is
  signal x: std_logic_vector(1 downto 0);
begin
  process(C,R)
  begin
    if R='1' then
      Q<="00000001";
    elsif rising_edge(C) then
      x<=x(0)&EN;  -- synchronizator
      if x(1)='1' then
        Q<=Q(6 downto 0)&(not Q(7));  -- przesuw
      end if;
    end if;
  end process;
end;
```

Trzeci moduł opisuje licznik pierścieniowy z „krążącą jedynką":

```
-- L5.13
library IEEE;
use IEEE.std_logic_1164.all;
entity JEDEN is
  port( C,EN,R: in std_logic;
        Q: inout std_logic_vector(7 downto 0)  );
end;
architecture beh of JEDEN is
  signal x: std_logic_vector(1 downto 0);
begin
  process(C,R)
  begin
    if R='1' then Q<="00000001";
    elsif rising_edge(C) then
      x<=x(0)&EN;  -- synchronizator
      if x(1)='1' then
        Q<=Q(6 downto 0)&Q(7);  -- przesuw
      end if;
    end if;
  end process;
end;
```

Czwarty moduł opisuje licznik pierścieniowy z „krążącym zerem":

```
-- L5.14
library IEEE;
use IEEE.std_logic_1164.all;
entity ZERO is
  port( C,EN,R: in std_logic;
        Q: inout std_logic_vector(7 downto 0) );
end;
architecture beh of ZERO is
  signal x: std_logic_vector(1 downto 0);
```

```
begin
  process(C,R)
  begin
    if R='1' then
      Q<="11111110";
    elsif rising_edge(C) then
      x<=x(0)&EN;  -- synchronizator
      if x(1)='1' then
        Q<=Q(6 downto 0)&Q(7);  -- przesuw
      end if;
    end if;
  end process;
end;
```

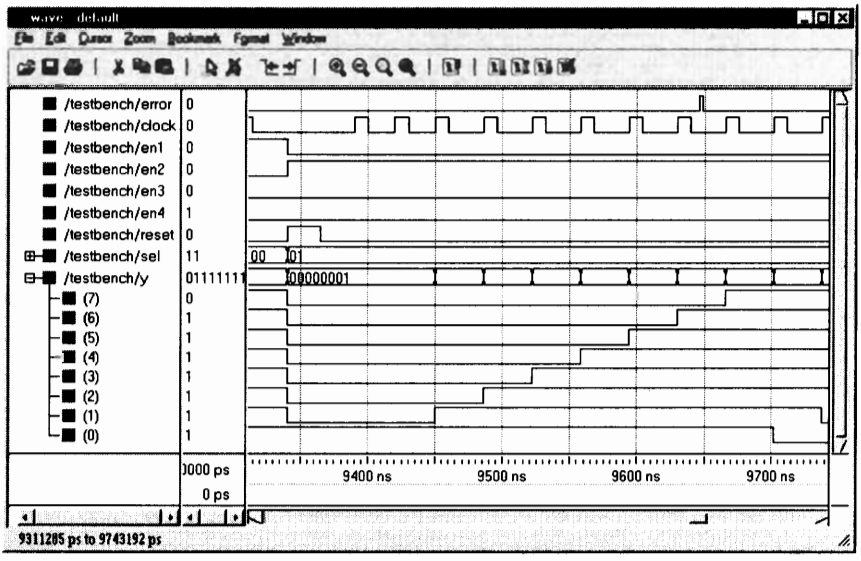
Przy użyciu opisanych modułów można sformułować opis strukturalny projektowanego układu:

```
-- L5.15
library IEEE;
use IEEE.std_logic_1164.all;
entity TOP is
  port( CLOCK,EN1,EN2,EN3,EN4,RESET: in std_logic;
        SEL: in std_logic_vector(1 downto 0);
        Y: out std_logic_vector(7 downto 0) );
end;
architecture arch of TOP is
  component MOD256
    port( C,EN,R: in std_logic;
          Q: out std_logic_vector(7 downto 0) );
  end component;
  component JOHNSON
    port( C,EN,R: in std_logic;
          Q: inout std_logic_vector(7 downto 0) );
  end component;
  component JEDEN
    port( C,EN,R: in std_logic;
          Q: inout std_logic_vector(7 downto 0) );
  end component;
  component ZERO
    port( C,EN,R: in std_logic;
          Q: inout std_logic_vector(7 downto 0) );
  end component;
  component MUX
    port( A,B,C,D: in std_logic_vector(7 downto 0);
          SEL: in std_logic_vector(1 downto 0);
          Y: out std_logic_vector(7 downto 0) );
  end component;
  signal a,b,c,d: std_logic_vector(7 downto 0);
begin
   U1: MOD256 port map (CLOCK,EN1,RESET,a);
```

```
    U2: JOHNSON port map (CLOCK,EN2,RESET,b);
    U3: JEDEN port map (CLOCK,EN3,RESET,c);
    U4: ZERO port map (CLOCK,EN4,RESET,d);
    U5: MUX port map (a,b,c,d,SEL,Y);
end;
```

Moduł `L5.16` opisuje program testowy. Pojedynczy impuls sygnału zegarowego generowany jest w każdym obiegu pętli. Stan licznika modulo 256 porównywany jest z indeksem pętli `i`. Wektory testowe liczników Johnsona i pierścieniowych zawiera tablica `test_vectors`. Sygnał wyjściowy `error` przyjmuje stan 1 w przypadku błędnej odpowiedzi. Przebiegi sygnałów otrzymanych podczas symulacji funkcjonalnej przedstawia rysunek 5.24. Widoczny jest pojedynczy impuls sygnału błędu `error` w części testu licznika Johnsona. Impuls ten powstaje w wyniku błędnej wartości, która została wpisana do tablicy wektorów testowych `test_vectors`. Po jej zmianie na poprawną symulacja daje wynik pozytywny.

```
-- L5.16
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity testbench is
end testbench;
architecture behavior of testbench is
  component top
    port( CLOCK,EN1,EN2,EN3,EN4,RESET : in std_logic;
```



Rys. 5.24 Przykładowe wyniki symulacji funkcjonalnej

```
            SEL : in std_logic_vector(1 downto 0);
            Y : out std_logic_vector(7 downto 0) );
    end component;
    signal error : std_logic;
    signal CLOCK,EN1,EN2,EN3,EN4,RESET : std_logic;
    signal SEL : std_logic_vector(1 downto 0);
    signal Y : std_logic_vector(7 downto 0);
    type test_vector_array is array (natural range <>)
                        of std_logic_vector(7 downto 0);
    constant test_vectors: test_vector_array:=
     ( "00000001",  -- licznik Johnsona
       "00000011",
       "00000111",
       "00001111",
       "00011111",
       "00111111",
       "00000000", -- blad, powinno być "01111111"
       "11111111",
       "11111110",
       "11111100",
       "11111000",
       "11110000",
       "11100000",
       "11000000",
       "10000000",
       "00000000",
       "00000001",  -- licznik z „krazaca jedynka"
       "00000010",
       "00000100",
       "00001000",
       "00010000",
       "00100000",
       "01000000",
       "10000000",
       "11111110",  -- licznik z „krazacym zerem"
       "11111101",
       "11111011",
       "11110111",
       "11101111",
       "11011111",
       "10111111",
       "01111111" );
begin
    UUT: top port map (CLOCK,EN1,EN2,EN3,EN4,RESET,
                       SEL,Y);
    process
    begin
      error<='0';CLOCK<='0';
      -- test licznika MOD256
      EN1<='1';EN2<='0';EN3<='0';EN4<='0';SEL<="00";
```

```
RESET<='1'; wait for 50 ns;
RESET<='0'; wait for 50 ns;
-- dwa impulsy zegarowe dla synchronizatora
CLOCK<='1','0' after 10 ns; wait for 30 ns;
CLOCK<='1','0' after 10 ns; wait for 30 ns;
for i in 1 to 255 loop
  CLOCK<='1','0' after 10 ns; -- impuls zegarowy
  wait for 16 ns;
  -- test odpowiedzi UUT
  if Y/=std_logic_vector(TO_UNSIGNED(i,8)) then
    assert false
      report "Sygnal Y jest nieprawidlowy";
    error<='1','0' after 3 ns;
  else
    error<='0';
  end if;
  wait for 20 ns;
end loop;
-- test licznika Johnsona
EN1<='0';EN2<='1';EN3<='0';EN4<='0';SEL<="01";
RESET<='1'; wait for 25 ns;
RESET<='0'; wait for 25 ns;
-- dwa impulsy zegarowe dla synchronizatora
CLOCK<='1','0' after 10 ns; wait for 30 ns;
CLOCK<='1','0' after 10 ns; wait for 30 ns;
for i in 0 to 1 loop
  for j in 0 to 14+i loop
    CLOCK<='1','0' after 10 ns;--impuls zegarowy
    wait for 16 ns;
    -- test odpowiedzi UUT
    if Y/=test_vectors(j+1-i) then
      assert false
        report "Sygnal Y jest nieprawidlowy";
      error<='1','0' after 3 ns;
    else
      error<='0';
    end if;
    wait for 20 ns;
  end loop;
end loop;
-- test licznika piersc. z kraz. jedynka
EN1<='0';EN2<='0';EN3<='1';EN4<='0';SEL<="10";
RESET<='1'; wait for 25 ns;
RESET<='0'; wait for 25 ns;
-- dwa impulsy zegarowe dla synchronizatora
CLOCK<='1','0' after 10 ns; wait for 30 ns;
CLOCK<='1','0' after 10 ns; wait for 30 ns;
for i in 0 to 1 loop
  for j in 16 to 22+i loop
    CLOCK<='1','0' after 10 ns;--impuls zegarowy
    wait for 16 ns;
```

```
            -- test odpowiedzi UUT
            if Y/=test_vectors(j+1-i) then
              assert false
                report "Sygnal Y jest nieprawidlowy";
              error<='1','0' after 3 ns;
            else
              error<='0';
            end if;
            wait for 20 ns;
          end loop;
        end loop;
        -- test licznika piersc. z kraz. zerem
        EN1<='0';EN2<='0';EN3<='0';EN4<='1';SEL<="11";
        RESET<='1'; wait for 25 ns;
        RESET<='0'; wait for 25 ns;
        -- dwa impulsy zegarowe dla synchronizatora
        CLOCK<='1','0' after 10 ns; wait for 30 ns;
        CLOCK<='1','0' after 10 ns; wait for 30 ns;
        for i in 0 to 1 loop
          for j in 24 to 30+i loop
            CLOCK<='1','0' after 10 ns;--impuls zegarowy
            wait for 16 ns;
            -- test odpowiedzi UUT
            if Y/=test_vectors(j+1-i) then
              assert false
                report "Sygnal Y jest nieprawidlowy";
              error<='1','0' after 3 ns;
            else
              error<='0';
            end if;
            wait for 20 ns;
          end loop;
        end loop;
      end process;
    end behavior;
```

5.6 ZADANIA

1. Zaprojektować priorytetowy konwerter kodu „1 z 8" na kod NB.

2. Zmodyfikować układ z zadania 5.5.3 tak, aby dla liczby zliczonych impulsów mniejszej od 10 zero było wygaszane na pozycji dziesiątek wyświetlacza.

3. Zaprojektować dwukierunkowy 8-bitowy rejestr przesuwający z szeregowym wejściem i równoległym wyjściem.

4. Zaprojektować 4-bitowy licznik w kodzie NB o zadawanym stanie początkowym i końcowym.

5. Zaprojektować sumator dwóch czterodekadowych liczb BCD.

6. Zaprojektować układ stopera elektronicznego wyposażonego w przyciski START/STOP i RESET. Maksymalna liczba zliczonych impulsów wynosi 99.

7. Zaprojektować multiplikator dwóch liczb 4-bitowych.

6

SYSTEM PROJEKTOWY *MAX+PLUS II*

Tomasz Orżanowski

System *MAX+plus II 10.1* (***M**ultiple **A**rray Matri**X** **P**rogrammable **L**ogic **U**ser **S**ystem)* firmy *Altera* (www.altera.com) jest przeznaczony do projektowania z użyciem programowalnych układów cyfrowych z rodzin: CLASSIC, MAX3000A, MAX7000, MAX7000B, MAX7000E, MAX7000AE, MAX7000S, MAX9000, FLEX6000, FLEX8000, FLEX10K, FLEX10KA, FLEX10KE i ACEX1K.

W tym rozdziale jest opisana studencka wersja programu *MAX+plus II 10.1* (*06/12/2001*). Aktualną wersję można bezpłatnie uzyskać przez Internet z serwera firmy *Altera* (www.altera.com/education/univ/unv-software.html) po wypełnieniu krótkiego formularza osobowego. Wersja 10.1 jest oferowana w pliku skompresowanym *student101.exe* o wielkości 46,2 MB. Instrukcja użytkowania systemu *MAX+plys II Getting Started version 8.1* w postaci pliku *81_gs.pdf* (5.4 MB) jest dostępna pod adresem internetowym www.altera.com/support/software/sof-maxplus2.html.

System *MAX+plus II 10.1* umożliwia przygotowanie opisu układu cyfrowego w postaci:

- pliku tekstowego w językach opisu sprzętu: VHDL, VERILOG lub AHDL (*Altera* HDL),
- schematu,
- przebiegów czasowych (np. opis automatu),

z uwzględnieniem hierarchicznej struktury projektu. Możliwa jest wieloparametrowa optymalizacja logiczna i topograficzna, synteza z minimalizacją opóźnień czasowych lub powierzchni struktury programo-

walnej, dekompozycja, symulacja funkcjonalna i czasowa, symulacja struktury wieloukładowej, automatyczna lokalizacja błędów oraz programowanie układów. System umożliwia tworzenie i odczytywanie plików wynikowych w następujących formatach: EDIF (*Electronic Design Interchange Format*), SDF (*Standard Delay Format*), XNF (*Xilinx Netlist Files*) oraz *schematic OrCad*.

Do programowania układów użyto zestawu laboratoryjnego *Altera University Program Design Laboratory Package*. Zestaw ten zawiera płytkę testową *UP1 Education Board* z dwoma układami programowalnymi CPLD oraz przewód firmowy *ByteBlaster* do programowania układów za pomocą komputera PC.

6.1 INSTALACJA SYSTEMU

Przed zainstalowaniem systemu na komputerze PC należy uzyskać plik licencyjny (*license.dat*) z serwera firmy *Altera* (www.altera.com/support/licensing/lic-university.html). Licencja na oprogramowanie jest związana z numerem dysku twardego (woluminu), który należy podać podczas procedury uzyskiwania pliku licencyjnego. Numer ten można odczytać po uruchomieniu na komputerze trybu *MS-DOS*, wpisaniu polecenia `cd c:\`, a następnie polecenia `vol`. Na ekranie zostanie wyświetlony komunikat z numerem woluminu. Jest to ośmiocyfrowa liczba szesnastkowa np. 1E3A-3C72.

Instalację systemu *MAX+plus II 10.1* wykonuje się następująco:

1) Uruchom plik instalacyjny *student101.exe*.
2) Program instalacyjny określa dwa domyślne foldery: *c:\maxplus2* dla plików systemu *MAX+plus II 10.1* oraz *c:\max2work* dla plików z projektami (można wybrać inne nazwy). Uruchom instalację.
3) Po zainstalowaniu systemu, umieść plik licencyjny *license.dat* w folderze podstawowym *c:\maxplus2*.
4) Uruchom system wybierając z menu Start program *MAX+plus II 10.1 Baseline*.
5) Pojawi się okno z komunikatem *„Would you like information on how to obtain a free MAX+plus II Baseline license?"*. Kliknij przycisk *No*.
6) Wybierz z menu *Options* polecenie *License Setup*. Pojawi się okno *License Setup*, w którym kliknij przycisk *Browse* i podaj ścieżkę dostępu do pliku licencyjnego np. *c:\maxplus2\license.dat*.
7) Zatwierdź ustawienia przyciskiem OK.

Od tej pory wszystkie opcje systemu *MAX+plus II 10.1* są już dostępne. Przykład realizacji projektu w systemie *MAX+plus II* jest przedstawiony w rozdziale 3 pt. *MAX+plus II Tutorial* w instrukcji użytkowania systemu.

6.2 ORGANIZACJA I ZASOBY BIBLIOTECZNE SYSTEMU

Zasoby biblioteczne systemu *MAX+plus II 10.1* znajdują się w trzech podfolderach: *max2lib*, *vhdl93* i *vhdl87*. Dwa ostatnie foldery zawierają pakiety VHDL zgodne ze standardem z 1993 i 1987 roku. Przedstawione poniżej tablice zawierają wybrane zasoby biblioteczne systemu *MAX+plus II 10.1*. Więcej informacji na ten temat można znaleźć w menu *Help* wybierając kolejno opcje: *VHDL*, *Megafunctions/LPM*, *Old-Style Macrofunctions* lub *Primitives*.

Tablica 6.1 Struktura folderów zawierających biblioteki i przykłady projektów w systemie *MAX+plus II 10.1* (wersja studencka)

Nazwa folderu	Zawartość
max2lib\edif	Biblioteka prymitywów i makromodułów do opisów w standardzie EDIF
max2lib\mega_lpm	Biblioteka symboli graficznych (*.sym*) dla megamodułów i modułów parametryzowanych z opisami w języku AHDL
max2lib\mf	Biblioteka symboli graficznych i schematów (*.gdf*) modeli układów TTL z serii 74 oraz innych bloków funkcjonalnych
max2lib\prim	Biblioteka graficznych symboli prymitywów
vhdl93\altera	Biblioteka `altera` z pakietem VHDL `maxplus2`
vhdl93\ieee	Biblioteka `ieee` z pakietami VHDL (p. tabl. 6.2)
vhdl93\std	Biblioteka `std` z pakietami VHDL: `standard` i `textio`
vhdl93\lpm	Biblioteka `lpm` z pakietem VHDL `lpm_components`
max2work\ahdl	Przykłady projektów opisanych w języku AHDL
max2work\chiptrip	Projekt *chiptrip*, który jest przedstawiony w instrukcji użytkowania systemu (*MAX+plus II Tutorial*)
max2work\edif	Przykłady plików w standardzie EDIF
max2work\tutorial	Folder plików tworzonych przez użytkownika podczas nauki obsługi systemu na przykładzie projektu *chiptrip*
max2work\vhdl	Przykłady projektów opisanych w języku VHDL
max2work\verilog	Przykłady projektów opisanych w języku VERILOG

Tablica 6.2 Pakiety VHDL zawarte w systemie *MAX+plus II* 10.1

Nazwa pliku	Nazwa pakietu	Biblioteka	Zawartość
maxplus2.vhd megacore.vhd	maxplus2 megacore	altera	Pakiety zawierają prymitywy, bloki funkcjonalnych TTL, makromoduły i megamodułów w postaci komponentów.
lpm_pack.vhd	lpm_components	lpm	Pakiet zawiera moduły parametryzowane w postaci komponentów.
std1164.vhd	std_logic_1164	ieee	Pakiet standardowy
arith.vhd	std_logic_arith	ieee	Pakiet zawiera deklaracje operacji arytmetycznych dla danych typu signed i unsigned.
unsigned.vhd	std_logic_unsigned	ieee	Pakiet zawiera deklaracje operacji arytmetycznych dla danych typu wektorowo-skalarnego, interpretowanych jak typ unsigned.
signed.vhd	std_logic_signed	ieee	Pakiet zawiera deklaracje operacji arytmetycznych dla danych typu wektorowo-skalarnego, interpretowanych jak typu signed.

Pakiet maxplus2 zawiera tylko deklaracje komponentów różnych primitywów (buforów, przerzutników). Opisy VHDL tych elementów nie są dostępne dla użytkownika, ponieważ zostały skompilowane (pliki z rozszerzeniem *.dls*).

Tablica 6.3 Prymitywy zawarte w pakiecie maxplus2

Typ prymitywu	Nazwa prymitywu
Bufor (*Buffer*)	CARRY, CASCADE, EXP, GLOBAL, LCELL, SOFT, TRI
Przerzutnik (*Flip-Flop*)	DFF, DFFE, JKFF, JKFFE, SRFF, SRFFE, TFF, TFFE
Zatrzask (*Latch*)	LATCH

Tablica 6.4 Makromoduły zawarte w pakiecie maxplus2

Typ makromodułu	Nazwa makromodułu	Opis działania
Sumator (*Adder*)	a_8fadd a_7480 a_74283 a_7483	Sumator 8-bitowy Sumator 1-bitowy Sumator 4-bitowy z przeniesieniami równoległymi
Jednostka arytm.-log. (*Arithmetic Logic Unit*)	a_74181 a_74182	Jednostka arytmetyczno-logiczna Generator przeniesień równoległych
Bufor (*Buffer*)	a_74240 a_74241	8 buforów trójstanowych, odwracających 8 buforów trójstanowych
Komparator (*Comparator*)	a_8mcomp a_7485 a_74688	Komparator 8-bitowy Komparator 4-bitowy Komparator 8-bitowy

cd. tablicy 6.4

Konwerter (*Converter*)	a_74184	Konwerter kodu BCD na kod NB
Licznik (*Counter*)	gray4	4-bitowy licznik w kodzie Graya
	a_7468	Dwa liczniki dziesiętne
	a_7493	4-bitowy licznik dwójkowy
	a_74191	4-bitowy licznik rewersyjny z asynchronicznym wpisem
	a_74669	4-bitowy licznik rewersyjny
Dekoder (*Decoder*)	a_16dmux	Dekoder „1 z 16"
	a_7446	Konwerter kodu BCD na kod 7-segmentowy
EDAC (*Error Detection and Correction Circuit*)	a_74630	16-bitowy układ równoległej detekcji i korekcji błędów
Koder (*Encoder*)	a_74148	Koder „8 na 3"
	a_74348	Koder priorytetowy „8 na 3" z wyjściami trójstanowymi
Frequency Divider	a_7456	Dzielnik częstotliwości
Zatrzask (*Latch*)	inpltch	Zatrzask
	a_7475	Zatrzask 4-bitowy
	a_74259	Zatrzask 8-bitowy z zerowaniem
	a_74845	8-bitowy interfejs magistrali (zatrzaski typu D z wyjściami trójstanowymi)
Multiplikator (*Multiplier*)	mult4	4-bitowy multiplikator równoległy
	a_74261	2-bitowy multiplikator równoległy
Multiplekser (*Multiplexer*)	a_21mux	Multiplekser "2 na 1"
	a_74151	Multiplekser "8 na 1"
	a_74157	Cztery multipleksery "2 na 1"
	a_74845	Multiplekser "8 na 1", selektor/rejestr z wyjściami trójstanowymi
Generator parzystości *Parity generator*	a_74180	9-bitowy generator bitu parzystości
Rejestr (*Register*)	a_7470	Przerzutnik JK z bramkami AND na wejściach J i K oraz z wejściami zerowania i ustawiania
	a_7473	Dwa przerzutniki JK z zerowaniem
	a_74171	Cztery przerzutniki D z zerowaniem
	a_74173	4-bitowy rejestr typu D
	a_74396	Ośmioelementowy rejestr
Rejestr przesuwający (*Shift register*)	barrelst	8-bitowy programowalny rejestr przesuwający
	a_7491	Rejestr przesuwający szeregowo-szeregowy
	a_7495	4-bitowy rejestr przesuwający równoległo-równoległy
	a_74198	8-bitowy dwukierunkowy rejestr przesuwający
Układy SSI (*SSI Functions*)	inhb	Iloczyn z zakazem
	a_7400	Dwuwejściowa bramka NAND
	a_7421	Czterowejściowa bramka AND
	a_7432	Dwuwejściowa bramka OR
	a74386	Cztery bramki XOR

System *MAX+plus II* zawiera liczne moduły parametryzowane *lpm* (*Library of Parametrized Modules*), które są zadeklarowane w pakiecie `lpm_components` jako komponenty. Moduły te mają rozbudowaną listę parametrów (stałych) wymienionych w klauzuli **`generic`** w jedno-

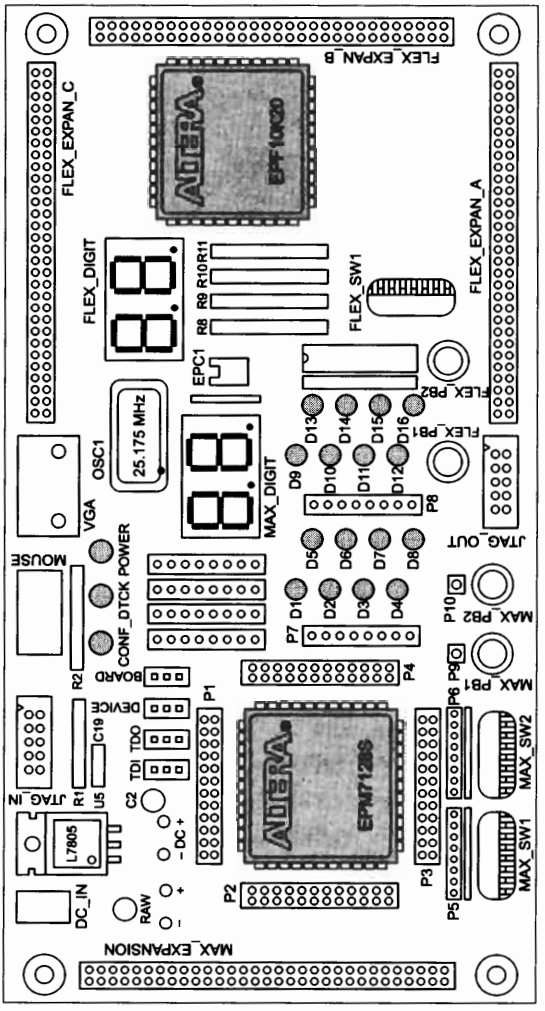
stce deklaracji komponentu. Projektant wykorzystujący komponent *lpm* określa wartości tych parametrów przy użyciu klauzuli **generic map** umieszczonej odpowiednio w opisie architektury jednostki projektowej (patrz p. 6.5.3).

Tablica 6.5 Moduły parametryzowane zawarte w pakiecie `lpm_components`

Nazwa modułu *lpm*	Opis działania
Gates	*Podstawowe komponenty logiczne*
`lpm_and`	Wielobitowa bramka AND
`lpm_bustri`	Wielobitowy bufor trójstanowy
`lpm_clshift`	Uniwersalny rejestr przesuwający z przesuwem logicznym, arytmetycznym lub obrotem
`lpm_constant`	Generator stałej
`lpm_decode`	Dekoder 1 z n
`lpm_or`	Wielobitowa bramka OR
`lpm_xor`	Wielobitowa bramka XOR
`lpm_inv`	Wielobitowy inwerter
`busmux`	Multiplekser grupowy „2 na 1"
`lpm_mux`	Uniwersalny multiplekser grupowy „*n* na *m*"
`mux`	Pojedynczy multiplekser „*n* na 1"
Arithmetic Components	*Komponenty arytmetyczne*
`lpm_abs`	Funkcja wartości bezwzględnej
`lpm_add_sub`	Wielobitowy sumator/subtraktor
`lpm_compare`	Komparator dwóch liczb *n*-bitowych
`lpm_counter`	Wielobitowy licznik dwójkowy z różnymi opcjami sterowania
`lpm_mult`	Wielobitowy multiplikator
Storage Components	*Moduły pamięciowe*
`lpm_ram_dq`	Pamięć RAM z oddzielnymi portami wejściowym i wyjściowym
`lpm_ram_io`	Pamięć RAM z jednym portem dwukierunkowym I/O
`lpm_rom`	Pamięć ROM
`csdpram`	Pamięć RAM z oddzielnymi portami wejściowym i wyjściowym, współdzielona w jednym cyklu zegara
`csfifo`	Pamięć FIFO
Other Functions	*Moduły specjalnego zastosowania*
`ntsc`	Generator sygnału video NTSC
`clocklock`	Układ redukcji opóźnienia i skosu sygnału zegara oraz do powielania (×2) częstotliwości zegara wewnętrznego

6.3 PŁYTKA TESTOWA

Płytka testowa *UP1 Education Board* (rys. 6.1) zawiera dwa układy programowalne firmy *Altera*: MAX7128S i EPF10K20. Szczegółowe opisy tych układów są dostępne w katalogu firmowym.



Rys. 6.1 Płytka testowa *UP1 Education Board*

Układ EPM7128S z rodziny MAX7000S jest programowany w technologii EEPROM. Oznacza to, że po wyłączeniu zasilania informacja o konfiguracji układu jest zachowana. Układ składa się z 128 makrokomórek, z których każda zawiera programowalną matrycę AND, nieprogramowalną matrycę OR i przerzutnik D z niezależnie progra-

mowanymi wejściami: zegarowym (*Clock*), zezwalającym dla zegara (*Clock Enable*), zerującym (*Reset*), ustawiającym (*Preset*) i wejściem informacyjnym *D*. Złożoność układu EPM7128S odpowiada liczbie 2500 bramek równoważnych.

Układ EPF10K20 z rodziny FLEX10K jest programowany w technologii SRAM. Oznacza to, że po wyłączeniu zasilania informacja o konfiguracji układu jest tracona. Układ EPF10K20 składa się z 1152 elementów logicznych LE (*Logic Element*) oraz z 6 bloków matryc wbudowanych EAB (*Embedded Array Block*). Każdy element LE zawiera 4-wejściową tablicę logiczną LUT (*Look-Up Table*), programowalny przerzutnik oraz dedykowane ścieżki sygnałowe do realizacji funkcji przeniesienia (*carry*) i do kaskadowego (*cascade*) łączenia elementów logicznych LE. Każdy blok EAB jest pamięcią 2048 × 1 bitów i może być użyty do realizacji pamięci RAM, ROM lub FIFO o organizacji np.: 512 × 4 bitów lub 256 × 8 bitów. Zastosowany na płytce testowej układ EPF10K20RC240-4 ma obudowę RQFP z 240 wyprowadzeniami.

Płytka ma własny stabilizator +5 V i może być zasilana ze źródła napięcia stałego od +7 V do +9 V o wydajności prądowej co najmniej 350 mA. Jeśli zasilanie jest włączone, to świeci zielona dioda LED – *POWER*. Na płytce znajduje się generator kwarcowy o częstotliwości 25,175 MHz (częstotliwość taka jest potrzebna do realizacji interfejsu VGA), którego wyjście jest doprowadzone do układu EPM7128S (końcówka nr 83) i do układu EPF10K20 (końcówka nr 91).

Przewód firmowy *ByteBlaster* stanowi interfejs w standardzie JTAG (*Joint Test Action Group*) między portem równoległym komputera PC, a płytką testową i służy do programowania układów. Jest on dołączany do płytki poprzez 10-wtykowe gniazdo *JTAG_IN*.

Płytka testowa ma 4 zwory: *TDI*, *TDO*, *Device*, *Board* do konfiguracji interfejsu JTAG w czterech trybach pracy, które są wymienione w tablicy 6.6.

Gdy układ na płytce jest programowany, to dioda LED – *TCK* świeci sygnalizując transfer danych z komputera. Dioda LED – *CONF_D* świeci po zaprogramowaniu układu.

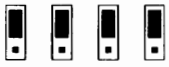
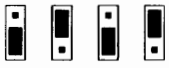
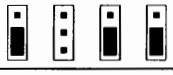
Ponadto płytka *UP1* umożliwia realizacje interfejsu VGA do monitora ekranowego oraz może być sterowana myszą lub klawiaturą PS/2. Użytkownik musi najpierw opisać (np. w języku VHDL) wyżej wymienione interfejsy i zaprogramować układy na płytce testowej.

Układ EPM7128S

Końcówki układu EPM7128S nie są połączone z przełącznikami *MAX_SW*, *MAX_PB* i diodami LED *D1÷D16*. Są natomiast dołączone do

dwurzędowych listew łączeniowych *P1÷P4*, umieszczonych dookoła układu EPM7128S. Odpowiednie sygnały wejściowe i wyjściowe dla układu są dołączane przewodami. W tablicy 6.7 podano numery końcówek układu EPM7128S wyprowadzone na listwy łączeniowe *P1÷P4*.

Tablica 6.6 Konfiguracje ustawień interfejsu JTAG na płytce testowej

Opis konfiguracji	*TDI, TDO, Device, Board*
1. Programowany jest tylko układ EPM7128S	
2. Programowany jest tylko układ EPF10K20	
3. Programowane są dwa układy na płytce testowej[1)]	
4. Połączenie kilku płytek testowych razem[2)]	

Uwagi:
1) Pierwszym układem w łańcuchu JTAG jest EPF10K20, a drugim EPM7128S.
2) Pierwszym układem w łańcuchu JTAG jest EPF10K20, a drugim EPM7128S. Ostatnia płytka testowa w łańcuchu JTAG musi być skonfigurowana w jednym z trzech trybów 1÷3; nie może być ustawiona w trybie 4.

Tablica 6.7 Numery końcówek układu EPM7128S wyprowadzone na listwy łączeniowe *P1÷P4*

P1		P2		P3		P4	
Zewn.	*Wewn.*	*Zewn.*	*Wewn.*	*Zewn.*	*Wewn.*	*Zewn.*	*Wewn.*
75	76	12	13	33	34	54	55
77	78	14	15	35	36	56	57
79	80	16	17	37	38	58	59
81	82	18	19	39	40	60	61
83	84	20	21	41	42	62	63
1	2	22	23	43	44	64	65
3	4	24	25	45	46	66	67
5	6	26	27	47	48	68	69
7	8	28	29	49	50	70	71
9	10	30	31	51	52	72	73
11	–	32	–	53	–	74	–

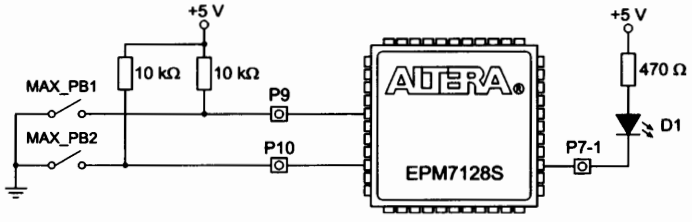
Wewn. oznacza wewnętrzny rząd gniazd listwy łączeniowej, tj. gniazda leżące bliżej układu EPM7128S, a *Zewn.* oznacza zewnętrzny rząd gniazd listwy łączeniowej.

Przyciski *MAX_PB*

Przyciski *MAX_PB1* i *MAX_PB2* są dołączone do napięcia zasilania +5 V przez rezystory 10 kΩ (rys. 6.2) i wyprowadzone na zaciski *P9* i *P10* na płytce testowej. Przyciśnięcie oznacza zmianę z poziomu H na L.

Przełączniki *MAX_SW*

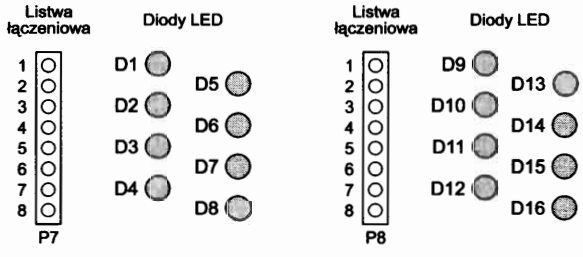
Przełączniki grupowe *MAX_SW1* i *MAX_SW2* zawierają po 8 niezależnych przełączników dwupołożeniowych, które są wyprowadzone na listwy łączeniowe *P5* i *P6*. Gdy przełącznik jest otwarty, to na jego zacisku łączeniowym jest poziom H. W przeciwnym razie jest poziom L. Zaciski łączeniowe przełączników *MAX_SW* łączy się przewodami z tymi końcówkami układu EPM7128S, które zostały zaprogramowane jako wejścia.



Rys. 6.2 Uproszczony schemat połączenia przycisków *MAX_PB* z układem programowalnym

Diody LED

Na płytce testowej znajduje się 16 diod LED *D1÷D16*, których anody są dołączone przez rezystory 470 Ω do napięcia zasilania U_{CC}, a katody wyprowadzono na listwy łączeniowe *P7* i *P8*. Dioda LED świeci, gdy poziom L jest podany na jej zacisk łączeniowy. Na rysunku 6.3 pokazano numerację zacisków łączeniowych dla diod LED *D1÷D16*.



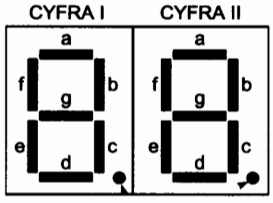
Rys. 6.3 Rozmieszczenie zacisków łączeniowych dla diod LED na płytce testowej

Wyświetlacz 7-segmentowy *MAX_DIGIT*

Podwójny wyświetlacz 7-segmentowy LED jest bezpośrednio dołączony do układu EPM7128S. Segmenty LED świecą, gdy sygnał wyjściowy z układu jest na poziomie L.

Na rysunku 6.4 pokazano oznaczenia poszczególnych segmentów, a w tablicy 6.8 podano numery końcówek układu EPM7128S połączo-

nych z wyświetlaczem *MAX_DIGIT*. Niewykorzystane końcówki I/O i sygnały globalne w układzie EPM7128S są doprowadzone do dwóch rzędów otworów lutowniczych *MAX_EXPANSION*.



Rys. 6.4 Oznaczenia segmentów w wyświetlaczu 7-segmentowym LED

Tablica 6.8 Sposób przyporządkowania końcówek układu EPM7128S do wyświetlacza *MAX_DIGIT*

Segment wyświetlacza	Numer końcówki (cyfra I)	Numer końcówki (cyfra II)
a	58	69
b	60	70
c	61	73
d	63	74
e	64	76
f	65	75
g	67	77
Kropka dziesiętna	68	79

Układ EPF10K20

Część końcówek układu EPF10K20 jest połączona z wyświetlaczem 7-segmentowym *FLEX_DIGIT*, przełącznikami *FLEX_SW* i *FLEX_PB*, portem *VGA* i portem *PS/2*. Pozostałe końcówki układu są dołączone do trzech zestawów otworów lutowniczych: *FLEX_EXPAN_A*, *FLEX_EXPAN_B*, *FLEX_EXPAN_C*.

Przyciski *FLEX_PB*

Przycisk *FLEX_PB1* jest połączony z końcówką 28, a przycisk *FLEX_PB2* z końcówką 29 układu EPF10K20. Każdy z przycisków *FLEX_PB* po naciśnięciu ustala poziom L na końcówce układu EPF10K20.

Przełącznik *FLEX_SW*

Przełącznik grupowy *FLEX_SW1* składa się z 8 niezależnych przełączników, które po włączeniu (w pozycji dolnej) ustalają poziom L na

określonych końcówkach układu EPF10K20. W tablicy 6.9 podano numery końcówek układu EPF10K20, które są połączone z przełącznikami *FLEX_SW1*.

Tablica 6.9 Sposób przyporządkowania końcówek układu EPF10K20 do przełączników *FLEX_SW1*

Numer przełącznika	Numer końcówki
1	41
2	40
3	39
4	38
5	36
6	35
7	34
8	33

Wyświetlacz 7-segmentowy *FLEX_DIGIT*

Podwójny wyświetlacz 7-segmentowy LED jest bezpośrednio dołączony do układu EPF10K20. Aktywnym poziomem sygnału do sterowania segmentami LED jest poziom L. W tablicy 6.10 podano sposób przyporządkowania końcówek układu EPF10K20 do poszczególnych segmentów wyświetlacza *FLEX_DIGIT*.

Tablica 6.10 Sposób przyporządkowania końcówek układu EPF10K20 do wyświetlacza *FLEX_DIGIT*

Segment wyświetlacza	Numer końcówki (cyfra I)	Numer końcówki (cyfra II)
a	6	17
b	7	18
c	8	19
d	9	20
e	11	21
f	12	23
g	13	24
Kropka dziesiętna	14	25

6.4 PROCES PROJEKTOWANIA

Kolejne kroki procesu projektowania w systemie *MAX+plus II 10.1* będą omówione na podstawie przykładowych projektów. Aby rozpocząć projektowanie, najpierw należy uruchomić system w sposób opisany dalej.

6.4.1 Uruchomienie systemu

1) Wybierz polecenie *Start* → *Programy* → *MAX+plus II 10.1 Baseline* → *MAX+plus II 10.1 Baseline* lub kliknij dwukrotnie ikonę programu *MAX+plus II 10.1 Baseline* na pulpicie. Na ekranie pojawi się okno główne programu (rys. 6.5).
2) Wybierz z menu polecenie *Options* → *Preferences* i zaznacz (✓) pola: *Show Toolbar*, *Show Status Bar*, aby paski narzędzi i pasek statusu były wyświetlane we wszystkich aplikacjach systemu *MAX+plus II*. Zatwierdź ustawienia przyciskiem *OK*. Paski narzędzi zawierają przyciski najczęściej używanych poleceń. Pasek statusu wyświetla zwięzły opis poleceń menu oraz dostarcza opisu dla przycisków pasków narzędzi.



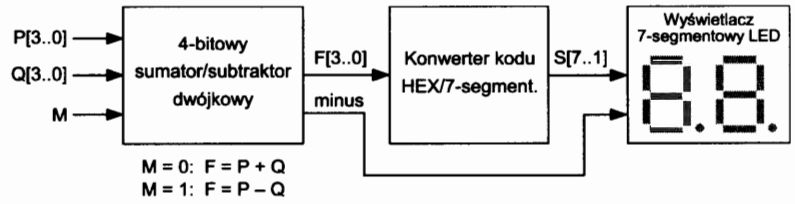
Rys. 6.5 Okno główne programu *MAX+plus II*

W systemie *MAX+plus II* projekty znajdują się w oddzielnych folderach, które zwykle umieszcza się w folderze roboczym *c:\max2work*. Aby utworzyć odrębny folder (np. *sumb4*) dla nowego projektu (np. *sumsub4.vhd*) należy wybrać polecenie *File* → *Save As*, a następnie napisać w polu *File Name* pełną ścieżkę dostępu do projektu (np. *c:\max2work\sumb4\sumsub4.gdf*) i przycisnąć *OK*.

6.4.2 Opis schematowy sumatora/subtraktora

Należy zaprojektować 4-bitowy sumator/subtraktor i zaprogramować układ CPLD na płytce *UP1*. Liczby wejściowe *P* i *Q* są w naturalnym kodzie binarnym (NB) bez znaku. Subtraktor wykonuje operację odejmowania w kodzie U2. W przypadku różnicy ujemnej zostaje ustawiony bit flagi (*minus* = 1), a wynik odejmowania jest w kodzie NB. Wynik

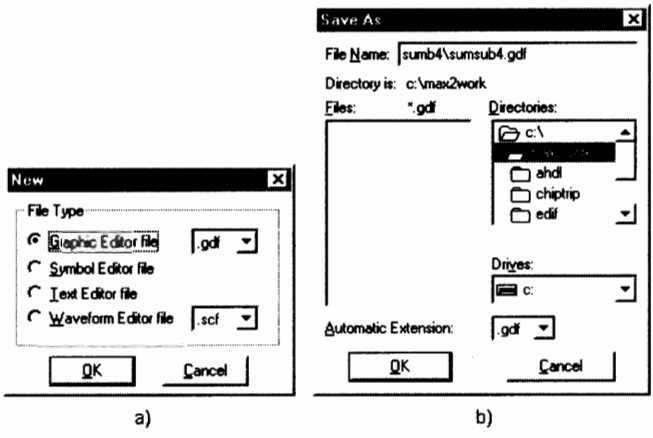
sumowania lub odejmowania jest wyświetlany na wyświetlaczu 7-segmentowym LED w kodzie szesnastkowym (HEX) (rys. 6.6).



Rys. 6.6 Schemat blokowy układu sumatora/subtraktora dwójkowego z wyświetlaczem 7-segmentowym LED

Ponieważ w bibliotece (*max2lib\mf*) systemu *MAX+plus II* jest dostępny gotowy komponent (*7483*), realizujący funkcję 4-bitowego sumatora dwójkowego z przeniesieniami równoległymi, zatem projekt sumatora/subtraktora dwójkowego wykonano przy użyciu opisu schematowego zamiast opisu w języku VHDL. Kolejne kroki projektowania są następujące:

1) Uruchom system *MAX+plus II* i wybierz z menu *File* polecenie *New*.
2) W oknie dialogowym *New* zaznacz opcję *Graphic Editor file* i wybierz rozszerzenie *.gdf* (rys. 6.7a).
3) Zatwierdź wybór przyciskiem *OK*. Następnie pojawi się okno edytora schematowego *Untitled1 – Graphic Editor*.



Rys. 6.7 Utworzenie nowego pliku: a) schematowego, b) zapis pliku do folderu

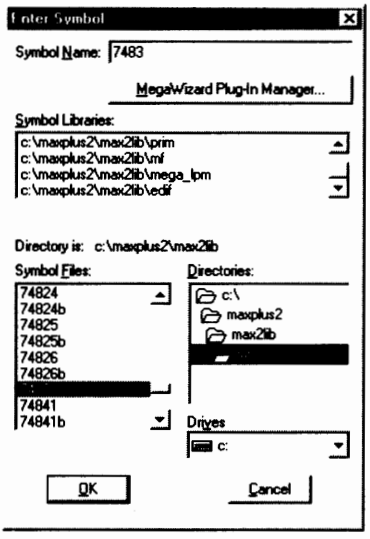
4) Wybierz polecenie *File* → *Save As*, a następnie w polu *Directories* (rys. 6.7b) kliknij dwa razy na folder *max2work*. W polu *File Name* wpisz *sumb4\sumsub4.gdf* i przyciśnij *OK*. Projekt zostanie zapi-

sany pod przykładową nazwą *sumsub4.gdf* w nowym folderze *sumb4*.

5) Wybierz z menu *File* polecenie *Project* → *Set Project to Current File* i ustaw plik *sumsub4.gdf* jako projekt bieżący.

Wstawianie symboli bramek oraz bloków funkcjonalnych do schematu

1) Wybierz polecenie *Symbol* → *Enter Symbol* lub kliknij dwukrotnie myszą w wolnym miejscu na schemacie. Pojawi się okno dialogowe *Enter Symbol* (rys. 6.8). W polu *Symbol Libraries* są pokazane biblioteki systemu zawierające symbole graficzne: prymitywów (*max2lib\prim*), makromodułów i układów TTL (*max2lib\mf*) oraz modułów parametryzowanych (*max2lib\mega_lpm*). Po dwukrotnym kliknięciu myszą na bibliotece *mf* jej zawartość jest wyświetlana w polu *Symbol Files*. Wstawianie symboli graficznych do schematu wykonuje się przez wpisanie jego nazwy w polu *Symbol Name* lub przez dwukrotne kliknięcie myszą na wybranym symbolu w polu *Symbol Files*.
2) W polu *Symbol Files* wybierz nazwę *7483* i przyciśnij *OK*. Na schemacie pojawi się symbol układu 7483 (sumator 4-bitowy).



Rys. 6.8 Okno dialogowe wstawiania symboli do schematu

3) Postępując jak wyżej, wstaw do schematu symbole bramek: *XOR*, *AND* i *NOT* z biblioteki *prim* (wybierając w polu *Symbol Files* odpowiednio nazwy: *xor*, *and2*, *not*).

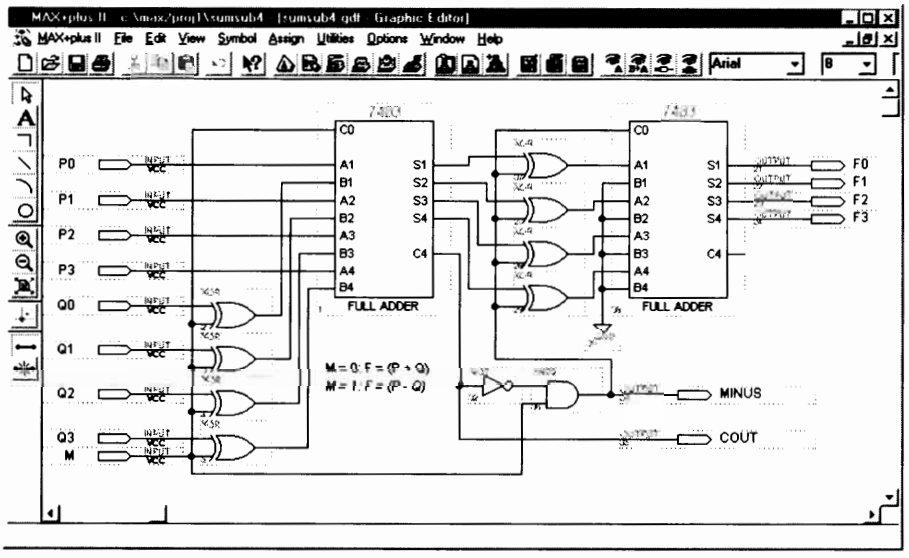
4) Przyciśnij klawisz *Ctrl* i kliknij lewym klawiszem myszy na symbolu *XOR*. Trzymając wciśnięty klawisz *Ctrl* przesuń kursor i zwolnij klawisz myszy. Utworzona zostanie kopia symbolu bramki *XOR*.
5) Umieść na schemacie 8 bramek *XOR* oraz drugi symbol sumatora 7483. Przemieszczanie symboli graficznych na schemacie wykonuje się przez kliknięcie i przytrzymanie lewego klawisza myszy na wybranym obiekcie, a następnie przesunięcie go myszą w określone miejsce.

Wstawianie symboli wejść i wyjść układu do schematu

1) Wstawianie symbolu wejścia *(Input)* jest realizowane poleceniem *Symbol → Enter Symbol*. W polu *Symbol Name* wpisz nazwę *Input* i przyciśnij *OK*. Na schemacie pojawi się symbol wejścia. Skopiuj i umieść na schemacie 9 symboli wejść.
2) Wstawianie symboli wyjść *(Output)* wykonuje się podobnie jak symboli wejść z tym, że w polu *Symbol Name* wpisuje się nazwę *Output*. Wstaw 6 symboli wyjść do schematu *sumsub4.gdf*.
3) Nadawanie nazw końcówkom wejściowym i wyjściowym układu wykonuje się przez podwójne kliknięcie na etykiecie *PIN_NAME* wybranego wyprowadzenia i wpisanie poprawnej nazwy. W projekcie *sumsub4.gdf* poszczególne wejścia mają nazwy: *P0*, *P1*, *P2*, *P3*, *Q0*, *Q1*, *Q2*, *Q3*, *M*. Wyjścia mają nazwy: *F0*, *F1*, *F2*, *F3*, *MINUS*, *COUT*.

Prowadzenie połączeń na schemacie

1) Z paska narzędzi znajdującego się po lewej stronie okna edytora schematowego wybierz przycisk do rysowania linii poziomych i pionowych. Kursor myszy zmieni się w „+".
2) Naprowadź kursor na końcówkę wejścia *P0*, przyciśnij lewy klawisz myszy i trzymając go, utwórz połączenie między wejściem *P0* i końcówką *A1* układu *7483*. Wykonaj pozostałe połączenia na schemacie tak, jak na rys. 6.9.
3) Kasowanie wybranej linii połączenia wykonuje się przez wskazanie (kliknięcie) myszą na wybraną linię (linia zmieni kolor) i przyciśnięcie klawisza *Del* lub *Backspace*.



Rys. 6.9 Schemat układu *sumsub4*

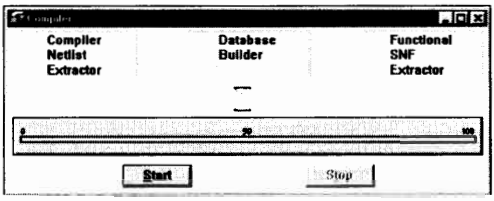
Sprawdzenie poprawności wykonania schematu i utworzenie symbolu graficznego całego projektu

1) Sprawdzenie poprawności wykonania schematu jest wykonywane poleceniem *File* → *Project* → *Save & Check* albo przez kliknięcie klawisza na pasku narzędziowym. Wykryte błędy należy usunąć.
2) Jeśli sprawdzenie schematu zakończy się sukcesem, wtedy kliknij przycisk *OK* i zamknij okno *Compiler*.
3) Ponieważ projekt *sumsub4.gdf* jest blokiem składowym w projekcie hierarchicznym, zatem należy utworzyć symbol graficzny bloku *sumsub4*, wybierając w menu *File* polecenie *Create Default Symbol*.

6.4.3 Symulacja funkcjonalna

Symulacja funkcjonalna sprawdza działanie układu bez uwzględnienia parametrów czasowych układu programowalnego wybranego do realizacji projektu. Aby wykonać symulację, najpierw należy wykonać kompilację projektu:

1) Wybierz z menu *MAX+plus II* polecenie *Compiler*. Pojawi się okno *Compiler*.
2) W menu *Processing* włącz (✓) opcję *Functional SNF Extractor*. Okno kompilatora powinno wyglądać jak na rys. 6.10.



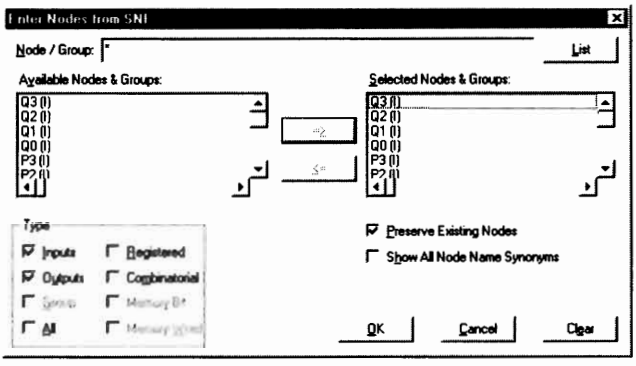
Rys. 6.10 Okno kompilatora dla symulacji funkcjonalnej

3) Uruchom kompilację projektu przyciskiem *Start*. Po jej pozytywnym zakończeniu zamknij okno kompilatora.

Przygotowanie wektora testowego

Następnie należy przygotować wektor testowy sygnałów wejściowych (wektor wirtualnych wymuszeń) w postaci pliku o rozszerzeniu *.scf*. Wektor ten jest tworzony za pomocą edytora przebiegów *MAX+Plus II Waveform Editor* w następujący sposób:

1) Wybierz polecenie *File* → *New*, zaznacz opcję *Waveform Editor file* w oknie *New* oraz określ rozszerzenie *.scf* dla tworzonego pliku. Zatwierdź ustawienia przyciskiem *OK*. Pojawi się okno edytora przebiegów *Waveform Editor*.
2) Wybierz polecenie *File* → *End Time* i określ długość czasu symulacji równą np. 1 µs, wpisując w polu *Time 1.0us* (w oknie *End Time*).
3) Wybierz polecenie *Options* → *Grid Size* i określ rozmiar siatki w edytorze przebiegów równy np. *100 ns*.
4) W menu *Options* włącz (✓) opcje: przyciąganie (*Snap to Grid*) i wyświetlanie (*Show Grid*) siatki w edytorze przebiegów.
5) Wybierz polecenie *Node* → *Enter Nodes from SNF*. Pojawi się okno dialogowe *Enter Nodes from SNF*. Powinny być włączone (✓) tylko opcje: *Inputs* i *Outputs*.
6) Kliknij przycisk *List*, aby wyświetlić listę dostępnych sygnałów wejściowych i wyjściowych w projekcie *sumsub4.gdf*.
7) Zaznacz w polu *Available Nodes & Groups* sygnały: *P0*, *P1*, *P2*, *P3*, *Q0*, *Q1*, *Q2*, *Q3*, *F0*, *F1*, *F2*, *F3*, *MINUS*, *COUT* i przenieś je do pola *Selected Nodes & Groups* za pomocą przycisku „=>" (rys. 6.11). Przycisk „<=" służy do usuwania sygnałów z pola *Selected Nodes & Groups*. Podwójne kliknięcie na wybranym sygnale w polu *Available Nodes & Groups* przenosi go automatycznie do pola *Selected Nodes & Groups*.



Rys. 6.11 Wybór sygnałów wejściowych i wyjściowych do symulacji

8) Zatwierdź wybór przyciskiem *OK*. W oknie edytora przebiegów pojawią się wybrane sygnały wejściowe i wyjściowe.
9) Wybierz polecenie *File* → *Save As*. W polu *File Name* pojawi się domyślnie nazwa pliku *sumsub4.scf*. Zapisz plik pod tą nazwą w bieżącym folderze *c:\max2work\sumb4*, wybierając przycisk *OK*.

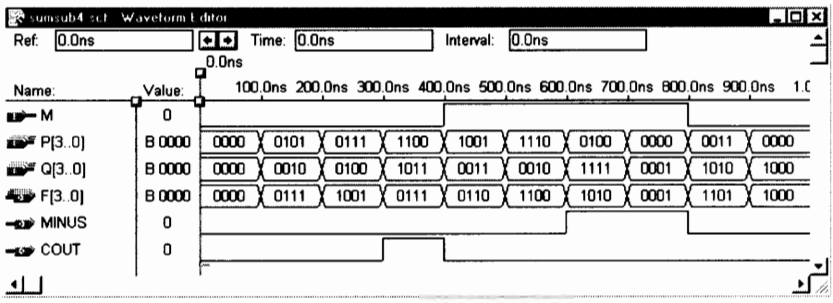
Edycja wektora testowego

1) Zaznacz sygnały *P3*, *P2*, *P1*, *P0* i połącz je w grupę *P[3..0]* wybierając polecenie *Node* → *Enter Group*. Ustaw zapis binarny dla stanu magistrali *P[3..0]* zaznaczając opcję *BIN* w polu *Radix*. Zatwierdź ustawienia przyciskiem *OK*.
2) Podobnie zgrupuj sygnały wejściowe *Q3*, *Q2*, *Q1*, *Q0* i wyjściowe *F3*, *F2*, *F1*, *F0*.
3) Zaznacz odcinek czasu od 400 ns do 800 ns w przebiegu sygnalu *M* i ustaw poziom H wybierając polecenie *Edit* → *Overwrite* → *High (1)* lub przycisk z paska narzędzi.
4) Zaznacz odcinek czasu od 100 ns do 200 ns w grupie *P[3..0]* i ustaw stan testowy *0101* wybierając polecenie *Edit* → *Overwrite* → *Group Value* lub przycisk .
5) Ustaw pozostałe stany testowe sygnałów *P[3..0]* i *Q[3..0]* jak na rys. 6.12.

Symulator systemu *MAX+plus II* umożliwia automatyczną weryfikację stanów wyjściowych określonych przez projektanta i oczekiwanych po wykonaniu symulacji. W tym celu należy wprowadzić oczekiwane stany wyjściowe w układzie *sumsub4* dla zadanych stanów wejściowych, edytując plik *sumsub4.scf* w następujący sposób:

1) Ustaw stany wyjściowe: *F[3..0] = 0111*, *MINUS = 0* i *COUT = 0* dla stanów wejściowych: *P[3..0] = 0101*, *Q[3..0] = 0010* i *M* = 0.

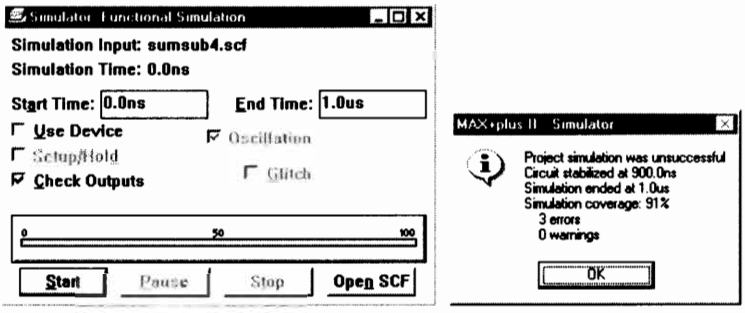
2) Pozostałe oczekiwane stany wyjściowe ustaw jak na rys. 6.12. Aby zademonstrować opcję weryfikację stanów wyjściowych w symulatorze, celowo wprowadzono nieprawidłowe stany na wyjściach *COUT* i *F[3..0]*.
3) Zapisz ponownie tak przygotowany plik pod nazwą *sumsub4.scf*, wybierając polecenie *File* → *Save*.



Rys. 6.12 Wektor wymuszeń i oczekiwane stany na wyjściach (*sumsub4.scf*)

Uruchomienie symulatora

1) Wybierz polecenie *MAX+Plus II* → *Simulator*. Pojawi się okno *Simulator: Functional Simulation*. Plik wektora wymuszeń *sumsub4.scf* jest pobierany automatycznie.
2) W aktywnym oknie symulatora włącz opcję *Check Outputs* (rys. 6.13a) i uruchom symulację przyciskiem *Start*.



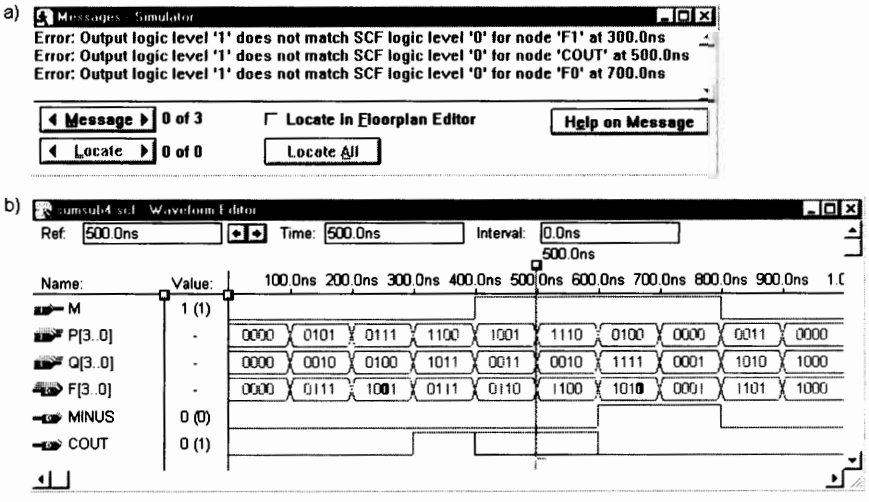
a) b)

Rys. 6.13 Symulacja funkcjonalna: a) wybór opcji *Check Outputs*, b) okno z komunikatami o błędach po zakończeniu symulacji

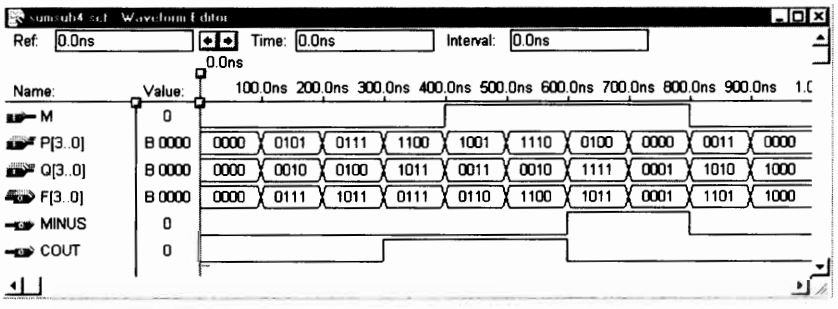
3) Gdy symulacja zostanie zakończona, wówczas pojawi się okno z komunikatami (rys. 6.13b) o: błędach (*Errors*), ostrzeżeniach (*Warnings*) i liczbie sygnałów (w %), które zmieniły swoją wartość w

trakcie symulacji (*Simulation coverage*). Zamknij okno komunikatów przyciskiem *OK*.

4) Symulator wykrył trzy stany wyjściowe źle określone przez projektanta. Ich lokalizacja jest podanie w postaci komunikatów w oknie *Messages – Simulator* (rys. 6.14a). Jeśli klikniemy dwukrotnie myszą na jednym z komunikatów o błędzie, wówczas automatycznie zostanie pokazana jego lokalizacja na wykresie przebiegów czasowych (rys. 6.14b).



Rys. 6.14 Wyniki symulacji funkcjonalnej: a) komunikaty o błędach, b) przebiegi czasowe z wykrytymi błędami



Rys. 6.15 Poprawne wyniki symulacji funkcjonalnej dla projektu *sumsub4*

5) Wybierz przycisk *Open SCF* w aktywnym oknie symulatora. Otworzony zostanie plik *sumsub4.scf*, zawierający wyniki symulacji (rys. 6.14b). Popraw błędne stany wyjściowe i wykonaj ponownie symulację. Poprawne wyniki symulacji są pokazane na rys. 6.15.

Wyniki symulacji mogą być zapisane w postaci tabeli, po wybraniu polecenia *File → Create Table File*. W polu *File Name* pojawi się domyślnie nazwa pliku *sumsub4.tbl* w folderze *c:\max2work\sumb4*.

6.4.4 Opis w języku VHDL konwertera kodu

Zadanie polega na przygotowaniu opisu w języku VHDL konwertera kodu szesnastkowego (HEX) na kod wyświetlacza 7-segmentowego.

Edytor języka VHDL w systemie *MAX+PLUS II 10.1* ma gotowe szablony: deklaracji jednostki projektowej, ciała architektury, instrukcji warunkowych itp., które wywołuje się poleceniem *Template → VHDL Template*.

1) Wybierz polecenie *File → New* i zaznacz opcję *Text Editor file*. Zatwierdź ustawienia przyciskiem *OK*.
2) Wybierz polecenie *File → Save As*. W polu *Automatic Extension* wybierz rozszerzenie *.vhd*. Zapisz projekt pod przykładową nazwą *hex7a.vhd* do folderu *c:\max2work\sumb4*. Pojawi się okno edytora tekstowego *Text Editor*.
3) Ustaw plik *hex7a.vhd* jako projekt bieżący, wybierając z menu *File* polecenie *Project → Set Project to Current File*.
4) Opisz w języku VHDL konwerter kodu szesnastkowego na kod wyświetlacza 7-segmentowego LED o wspólnej anodzie (`L6.1`). W tym przypadku aktywnym stanem na wyjściach konwertera kodu jest stan 0 (wtedy segment LED świeci). W systemie *MAX+plus II* nazwa jednostki projektowej musi być taka sama jak nazwa pliku *.vhd*.

```
-- L6.1
library ieee;
use ieee.std_logic_1164.all;
entity hex7a is
  port(i : in std_logic_vector (3 downto 0);
       s : out std_logic_vector (7 downto 1));
end hex7a;
architecture hex7a of hex7a is
  begin
    -- Numeracja segmentow
    --      s1
    --     ----
    -- s6| s7 |s2
    --     ----
    -- s5|     |s3
    --     ----
    --      s4
```

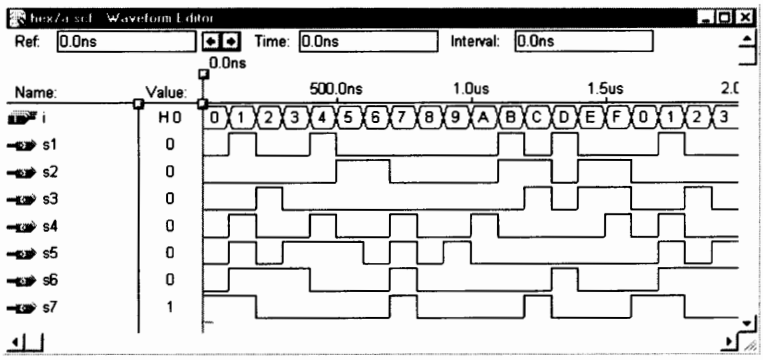
```
    with i select
    -- aktywnym stanem na wyjsciach s jest stan 0
      s <= "1000000" when "0000", -- 0
           "1111001" when "0001", -- 1
           "0100100" when "0010", -- 2
           "0110000" when "0011", -- 3
           "0011001" when "0100", -- 4
           "0010010" when "0101", -- 5
           "0000010" when "0110", -- 6
           "1111000" when "0111", -- 7
           "0000000" when "1000", -- 8
           "0010000" when "1001", -- 9
           "0001000" when "1010", -- A
           "0000011" when "1011", -- b
           "1000110" when "1100", -- C
           "0100001" when "1101", -- d
           "0000110" when "1110", -- E
           "0001110" when "1111", -- F
           "1100011" when others; -- n - wartosc
end hex7a;                        -- niezdefiniowana
```

5) Sprawdź poprawność opisu VHDL wybierając polecenie *File* → *Project* → *Save & Check*. Usuń błędy składni wykryte przez kompilator.
6) Wybierz polecenie *File* → *Create Default Symbol* i utwórz symbol graficzny bloku *hex7a*.

Symulacja funkcjonalna projektu *hex7a*

1) Wybierz polecenie *MAX+plus II* → *Compiler* i ustaw (✓) opcję *Functional SNF Extractor* w menu *Processing*.
2) Wykonaj kompilację projektu *hex7a*. Zamknij okno kompilatora.
3) Utwórz plik wektora wymuszeń *hex7a.scf* (polecenie *File* → *New*, opcja *Waveform Editor file*, rozszerzenie *.scf*).
4) Wybierz polecenie *File* → *End Time* i określ długość czasu symulacji równą 2 µs.
5) Ustaw rozmiar siatki w edytorze przebiegów równy *100 ns* (polecenie *Options* → *Grid Size*).
6) Wybierz polecenie *Node* → *Enter Nodes from SNF*, kliknij przycisk *List*, a następnie wstaw sygnały wejściowe *i3*, *i2*, *i1*, *i0* oraz sygnały wyjściowe *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7* do okna edytora przebiegów.
7) Wybierz polecenia *File* → *Save As* i zapisz plik pod domyślnie podaną nazwą *hex7a.scf* w folderze *c:\max2work\sumb4*.
8) Połącz sygnały wejściowe *i3*, *i2*, *i1*, *i0* w grupę *i[3..0]* wybierając polecenie *Node* → *Enter Group*. Wybierz zapis szesnastkowy *HEX* dla stanu magistrali *i[3..0]*.

9) Ustaw stany na magistrali *i[3..0]* wybierając polecenie *Edit* → *Overwrite* → *Count Value* z parametrami: *Starting Value = 0, Increment By = 1, Multiplied By = 1, Count Type: Binary* lub przycisk z paska narzędziowego. Zatwierdź ustawienia przyciskiem *OK*.
10) Zapisz ponownie przygotowany plik pod bieżąca nazwą, wybierając polecenie *File* → *Save*.
11) Uruchom symulator poleceniem *MAX+Plus II* → *Simulator*. Otwórz plik *hex7a.scf*, zawierający wyniki symulacji (rys. 6.16).



Rys. 6.16 Wyniki symulacji funkcjonalnej projektu *hex7a*

W tym przypadku stany na wyjściach konwertera kodu nie zostały wcześniej określone i poddane automatycznej weryfikacji. A zatem projektant musi sam sprawdzić poprawność otrzymanych wyników symulacji.

Program testowy do konwertera kodu

Aby wykonać weryfikację funkcjonalną projektu *hex7a*, przygotowano również w języku VHDL odpowiedni program testowy *hex7a_tb.vhd* (L6.2).

System *MAX+plus II 10.1* nie umożliwia weryfikacji projektów opisanych w języku VHDL za pomocą programów testowych. Takie testowanie jest możliwe z użyciem symulatora VHDL. Zastosowanie do tego celu symulatora *Active-HDL* firmy *Aldec* jest opisane w rozdziale 8.

```
-- L6.2
library ieee;
use ieee.std_logic_1164.all;
entity hex7a_tb is
end hex7a_tb;
architecture tb of hex7a_tb is
  component hex7a
    port (i : in std_logic_vector(3 downto 0);
```

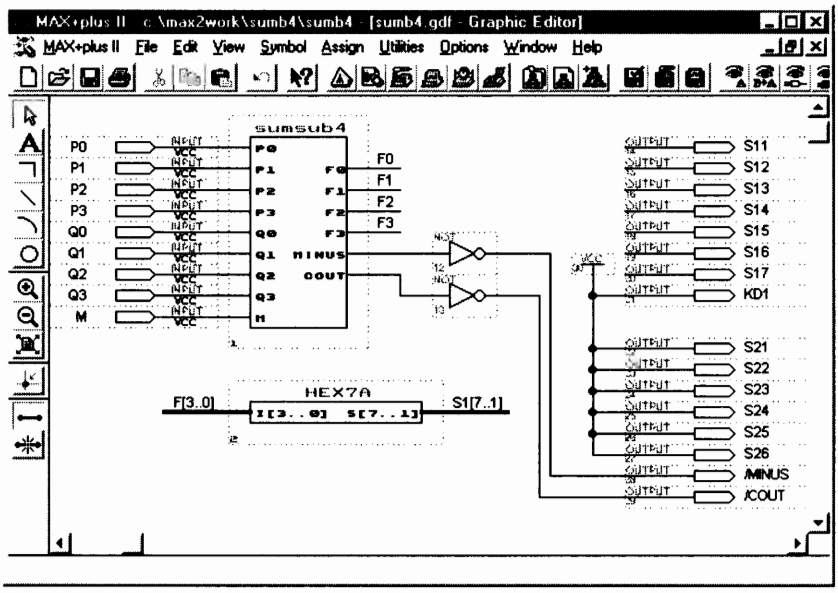
```
            s : out std_logic_vector(7 downto 1));
    end component;
    signal i : std_logic_vector(3 downto 0);
    signal s : std_logic_vector(7 downto 1);
    -- deklaracja typu rekordowego tv,
    -- ktory zawiera sygnaly: i, s
    type tv is record
      i : std_logic_vector(3 downto 0);
      s : std_logic_vector(7 downto 1);
    end record;
    -- deklaracja tablicy rekordow tva typu tv
    type tva is array (natural range <>) of tv;
      constant tvec : tva := (
      -- tablica wektorow testujacych
      (i => "0000", s => "1000000"),
      (i => "0001", s => "1111001"),
      (i => "0010", s => "0100100"),
      (i => "0011", s => "0110000"),
      (i => "0100", s => "0011001"),
      (i => "0101", s => "0010010"),
      (i => "0110", s => "0000010"),
      (i => "0111", s => "1111000"),
      (i => "1000", s => "0000000"),
      (i => "1001", s => "0010000"),
      (i => "1010", s => "0001000"),
      (i => "1011", s => "0000011"),
      (i => "1100", s => "1000110"),
      (i => "1101", s => "0100001"),
      (i => "1110", s => "0000110"),
      (i => "1111", s => "0001110"));
  begin
  -- deklaracja jednostki testowanej
  U1 : hex7a port map (i=>i, s=>s);
    -- wprowadzanie wektorow testujacych
    -- i sprawdzanie wynikow
    spr: process
      variable vec : tv;
      variable errors : boolean := false;
    begin
      for m in tvec'range loop
        vec := tvec(m);
        i <= vec.i;  -- przypisanie i-tego wektora
        wait for 20 ns;
        -- sprawdzenie stanu wyjsc s
        if s /= vec.s then
           assert false
           report "Stan wyjsc s jest nieprawidlowy";
           errors := true;
        end if;
      end loop;
      -- utworzenie komunikatu o bledach podczas
```

```
        -- symulacji
        assert not errors
           report "Wektory testujace sa przetwarzane
                   nieprawidlowo"
        severity note;
        assert errors
           report "Wektory testujace sa przetwarzane
                   prawidlowo"
        severity note;
        wait;
      end process spr;
    end tb;
```

6.4.5 Opis schematowy projektu hierarchicznego

Celem jest utworzenie projektu hierarchicznego w postaci schematu przez odpowiednie połączenie poprzednio wykonanych bloków funkcjonalnych *sumsub4* i *hex7a*.

1) Utwórz nowy plik schematowy i zapisz go pod przykładową nazwą *sumb4.gdf* w folderze *c:\max2work\sumb4*.
2) Ustaw plik *sumb4.gdf* jako projekt bieżący (polecenie *File* → *Project* → *Set Project to Current File*).
3) Wybierz z menu *Symbol* polecenie *Enter Symbol* i wstaw do schematu symbole *sumsub4* i *hex7a*. Symbole te znajdują się w folderze *c:\max2work\sumb4*.
4) Wstaw do schematu symbole wejść i wyjść oraz wykonaj odpowiednie połączenia jak na rys. 6.17.
5) Magistrale na schemacie rysuje się odpowiednio grubszą linią wybierając polecenie *Options* → *Line Style*. Połączenia magistralowe mogą być ustanawiane *bez konieczności fizycznego łączenia poszczególnych linii* (np. połączenie z magistralą *F[3..0]* polega na nadaniu końcówkom: *F0*, *F1*, *F2*, *F3* odpowiednio nazw: *F0*, *F1*, *F2*, *F3*. Aby nadać pojedynczej linii określoną nazwę trzeba kliknąć myszą na tę linię (pojawi się kursor „▪"), a następnie wpisać nazwę.
6) Sprawdź poprawność wykonania schematu wybierając z menu *File* polecenie *Project* → *Save & Check*.

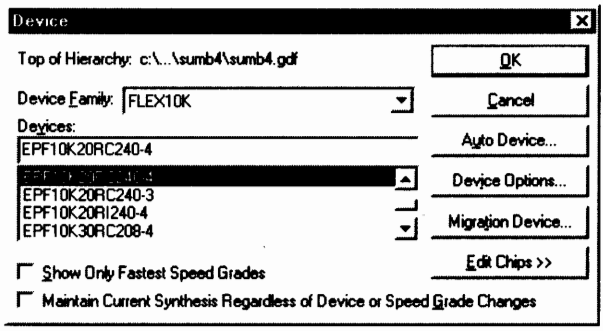


Rys. 6.17 Połączenie bloków *sumsub4* i *hex7a*

Wybór układu programowalnego i przyporządkowanie końcówek

Projekt *sumb4* zostanie zrealizowany w strukturze EPF10K20, ponieważ odpowiednie końcówki układu są już połączone z przełącznikami i wyświetlaczem na płytce testowej. Pozwoli to uniknąć wykonywania niezbędnych połączeń przewodowych.

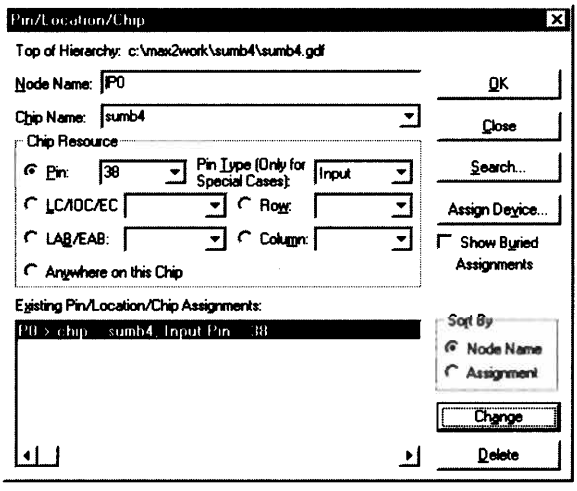
1) Wybierz w menu *Assign* polecenie *Device*. Pojawi się okno dialogowe *Device*. Opcję *Show Only Fastest Speed Grades* należy wyłączyć. W polu *Device Family* wybierz rodzinę układów FLEX 10K, a z listy *Devices* wybierz układ *EPF10K20RC240-4* (rys. 6.18).
2) Zatwierdź ustawienia przyciskiem *OK*.



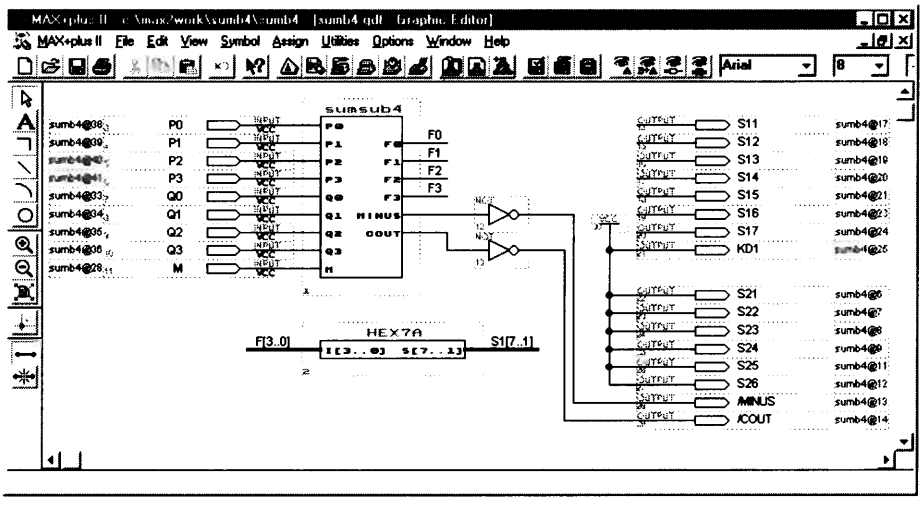
Rys. 6.18 Wybór układu programowalnego

Przyporządkowanie końcówek układu EPF10K20 do wejść i wyjść

1) Kliknij myszą na symbol wejścia *P0* i wybierz polecenie *Asign* → *PIN/Location/Chip*. Pojawi się okno *PIN/Location/Chip* (rys. 6.19).
2) W sekcji *Chip Resource* w polu *Pin* wpisz numer końcówki *38* dla wejścia *P0*. Zatwierdź ustawienia przyciskiem *Add*.



Rys. 6.19 Przyporządkowanie końcówek układu programowalnego



Rys. 6.20 Schemat projektu *sumb4* z przyporządkowanymi końcówkami układu

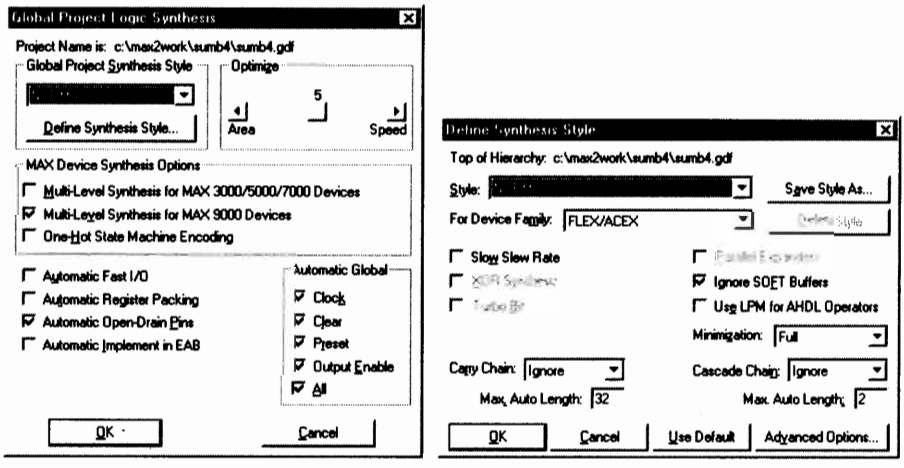
3) Pozostałe końcówki układu EPF10K20 przyporządkuj w projekcie *sumb4* w następujący sposób: *P1* – 39, *P2* – 40, *P3* – 41, *Q0* – 33, *Q1* – 34, *Q2* – 35, *Q3* – 36, *M* – 28, *S11* – 17, *S12* – 18, *S13* – 19, *S14* – 20, *S15* – 21, *S16* – 23, *S17* – 24, *KD1* – 25, *S21* – 6, *S22* –

7, *S23* – 8, *S24* – 9, *S25* – 11, *S26* – 12, */MINUS* – 13, */COUT* – 14. Schemat projektu *sumb4* z przyporządkowanymi końcówkami układu EPF10K20RC240-4 pokazano na rys. 6.20.

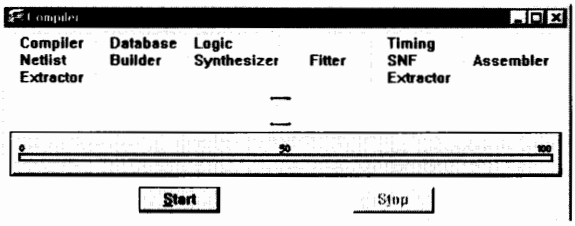
6.4.6 Kompilacja projektu

Kompilator w systemie *MAX+Plus II* najpierw sprawdza projekt z punktu widzenia błędów składniowych. Po pozytywnym wyniku sprawdzenia wykonywana jest synteza logiczna, wpasowanie projektu do wybranego układu programowalnego oraz utworzenie plików dla symulatora i programatora. Przebieg przetwarzania jest ustalany w menu *Processing*, które jest aktywne dopiero po wywołaniu okna *Compiler*. Styl syntezy logicznej jest określany w menu *Assign* → *Global Project Logic Synthesis* (rys. 6.21). Domyślnym stylem syntezy logicznej jest *Normal*. Suwak w polu *Optimize* pozwala ukierunkować optymalizację na minimalizację powierzchni struktury programowalnej (*Area*) lub na zapewnienie maksymalnej szybkości działania układu (*Speed*).

1) Wybierz z menu *MAX+plus II* polecenie *Compiler* lub przyciśnij klawisz na górnym pasku narzędziowym.
2) Włącz (✓) opcję *Timing SNF Extractor* w menu *Processing*, aby podczas kompilacji został utworzony plik wejściowy (*sumb4.snf*) dla symulatora i analizatora opóźnień czasowych *Timing Analyzer*.
3) Uruchom kompilator przyciskiem *Start* (rys. 6.22).



Rys. 6.21 Okna ustawień syntezy logicznej: a) parametrów ogólnych, b) stylu syntezy

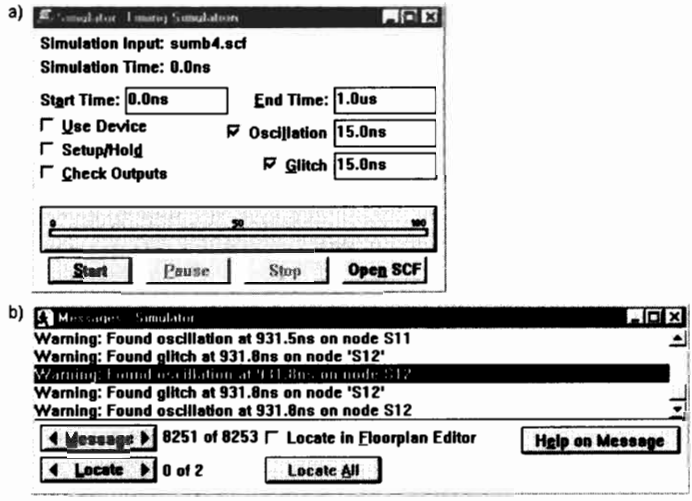


Rys. 6.22 Okno kompilatora

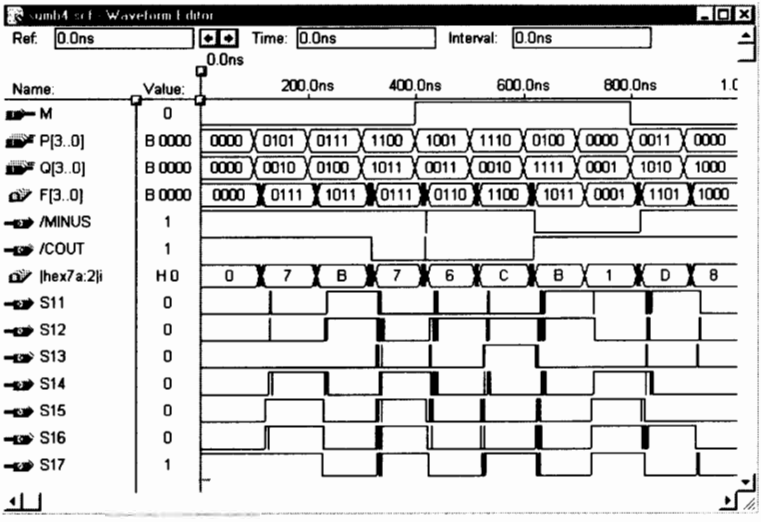
6.4.7 Symulacja czasowa

Symulacja czasowa (*timing simulation*) umożliwia sprawdzenie działania projektu z uwzględnieniem parametrów czasowych zastosowanego układu programowalnego. Wartości tych parametrów są ujęte w modelach układów o postaci plików z rozszerzeniem *.dmf* (*Device Model Files*), które znajdują się w głównym folderze *c:\maxplus2* systemu *MAX+plus II*. Przygotowanie symulacji jest wykonywane podobnie jak dla symulacji funkcjonalnej z tą różnicą, że przy symulacji czasowej plik wejściowy (np. *sumb4.snf*) dla symulatora zawiera czasy opóźnień propagacji dla projektu zrealizowanego w wybranym układzie programowalnym. Te parametry czasowe są automatycznie pobierane przez system z biblioteki modeli.

1) Utwórz odpowiedni plik wektora wymuszeń *sumb4.scf*. W tym przypadku można wykorzystać poprzednio używany plik *sumsub4.scf*, modyfikując go przez dodanie sygnałów wyjściowych *s11 ... s17* z konwertera kodu oraz sygnałów: */MINUS* i */COUT*.
2) Wybierz z menu polecenie *MAX+Plus II* → *Simulator* i włącz w oknie *Simulator: Timing Simulation* (rys. 6.23a) opcje wykrywania oscylacji (*Oscillation*) trwających powyżej np. 15 ns i impulsów hazardowych (*Glitch*) w zakresie np. 15 ns.
3) Uruchom symulator przyciskiem *Start*. Ostrzeżenia o wykrytych oscylacjach i impulsach hazardowych są wyświetlane w oknie *Messages – Simulator* (rys. 6.23b)



Rys. 6.23 Symulacja czasowa: a) wybór opcji *Oscillation* i *Glitch*, b) okno z komunikatami po zakończeniu symulacji



Rys. 6.24 Wyniki symulacji czasowej projektu *sumb4*

Z symulacji czasowej projektu *sumb4* pokazanej na rys. 6.24 wynika, że na wyjściach konwerterów kodu szesnastkowego na kod wyświetlacza 7-segmentowego występują impulsy hazardowe. W tym przypadku impulsy hazardowe są nieszkodliwe, ponieważ ich czas trwania jest bardzo krótki (kilkanaście ns) w porównaniu do czasu odczytu wyniku operacji arytmetycznej z wyświetlaczy 7-segmentowych.

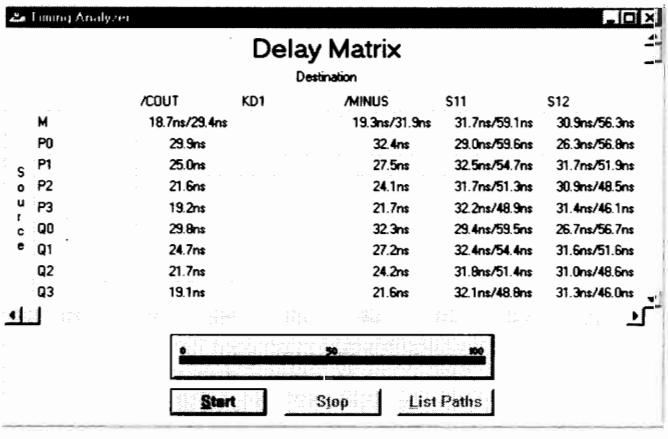
Analiza parametrów dynamicznych projektu (*Analyze Timing*)

Parametry dynamiczne projektu są wyznaczane przez analizator czasowy *Timing Analyzer*. Analizator czasowy wykonuje trzy rodzaje analiz:

1) *Delay Matrix* – analiza opóźnienia propagacji poszczególnych ścieżek sygnałowych (dla najlepszego i najgorszego przypadku).
2) *Registered Performance* – analiza parametrów logiki sekwencyjnej w projekcie, tj.: minimalny okres zegara (maksymalna częstotliwość zegara).
3) *Setup/Hold Matrix* – wyznaczenie minimalnej wartości czasów ustalania (*setup*) i podtrzymania (*hold*) dla poszczególnych sygnałów wejściowych w przerzutnikach (*flip-flops*) lub w zatrzaskach (*latches*).

Uruchomienie analizatora parametrów czasowych jest bardzo proste:

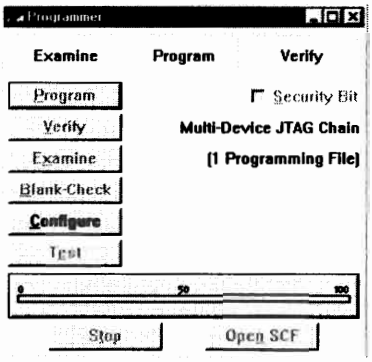
1) Wybierz polecenie *MAX+Plus II* → *Timing Analyzer*. Wyświetlona zostanie tablica *Delay Matrix*. W menu *Options* można określić parametry analizy czasowej np. opcja *Cut Off I/O Pin Feedback* powoduje, że sprzężenia zwrotne wewnątrz układu nie są analizowane.
2) Uruchom *Timing Analyzer* przyciskiem *Start*. Pojawi się informacja o zakończeniu analizy czasowej *Timing analysis is completed*. Kliknij przycisk *OK*. Wyniki analizy czasowej projektu *sumb4*, zrealizowanego w układzie EPF10K20RC240-4 pokazano na rys. 6.25.



Rys. 6.25 Wyniki analizy czasów opóźnień propagacji w projekcie *sumb4*

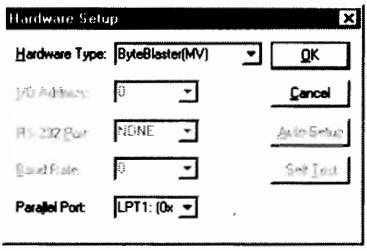
6.4.8 Programowanie układu

1) Połącz płytkę testową przewodem *ByteBlaster* z portem drukarkowym LPT w komputerze PC.
2) Sprawdź ustawienie zwór: *TDI*, *TDO*, *Device*, *Board* na płytce. Interfejs JTAG powinien być skonfigurowany dla przypadku, gdy programowany jest tylko układ EPF10K20 (tablica 6.6).
3) Włącz zasilanie płytki testowej. Dioda LED – *POWER* świeci.
4) Wybierz polecenie *MAX+Plus II* → *Programmer*. Pojawi się okno *Programmer* (rys. 6.26).



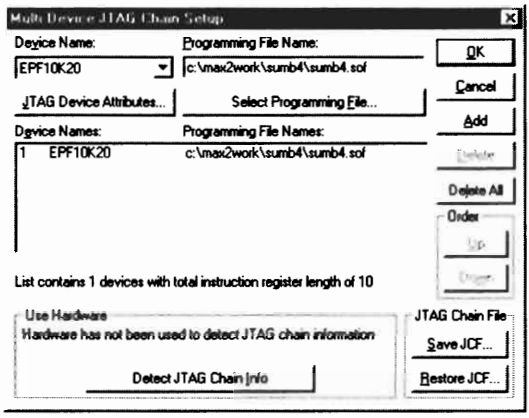
Rys. 6.26 Okno programatora *Programmer*

5) Wybierz polecenie *Option* → *Hardware Setup* i określ typ interfejsu sprzętowego, wybierając *ByteBlaster(MV)* w polu *Hardware Type* (rys. 6.27). Przyciśnij *OK*.



Rys. 6.27 Okno wyboru interfejsu sprzętowego

6) Przy aktywnym oknie *Programmer* zaznacz (✓) w menu *JTAG* opcję *Multi-Device JTAG Chain.*
7) Wybierz polecenie *JTAG* → *Multi-Device JTAG Chain Setup*. Pojawi się okno *Multi-Device JTAG Chain Setup* (rys. 6.28), które służy do konfiguracji interfejsu JTAG.



Rys. 6.28 Okno konfiguracji interfejsu JTAG

8) Wybierz z listy *Device Name* układ *EPF10K20.*
9) Wybierz przycisk *Select Programming File* i podaj ścieżkę dostępu (*c:\max2work\sumb4\sumb4.sofj* do pliku programującego.
10) Kliknij przycisk *Add*, aby umieścić układ EPF10K20 i plik programujący w oknie dialogowym *Device Names & Programming File Names* (rys. 6.28).
11) Kliknij przycisk *Detect JTAG Chain Info*, żeby sprawdzić połączenie *ByteBlaster.* System powinien wyświetlić komunikat *„JTAG chain information confirmed by hardware check".*
12) Kliknij przycisk *Save JCF* i zapisz bieżące ustawienia interfejsu JTAG w pliku *sumb4.jcf* (*JTAG Chain File*). Zatwierdź wybór przyciskiem *OK.*
13) Kliknij przycisk *OK*, aby zatwierdzić wykonane zmiany.
14) Uruchom programowanie układu EPF10K20 przyciskiem *Configure* w oknie *Programmer* (rys. 6.26).
15) Po zakończeniu programowania układu, system powinien wyświetlić informację *Configuration Complete.*
16) Sprawdź działanie zaprojektowanego układu sumatora/subtraktora. Sygnał *M* jest zmieniany przyciskiem *FLEX_PB1* na płycie testowej.

6.5 PRZYKŁADY PROJEKTÓW

Przedstawione dalej przykłady projektów są wykonane wyłącznie z użyciem języka VHDL. Ośmiobitowy licznik rewersyjny z konwerterami kodu na wyjściu jest przykładem projektowania strukturalnego w systemie *MAX+plus II* z użyciem języka VHDL.

6.5.1 Ośmiobitowy licznik rewersyjny

Zadanie polega na zaprojektowaniu 8-bitowego, rewersyjnego, synchronicznego licznika dwójkowego przy użyciu stylu behawioralnego w języku VHDL. Stan licznika jest wyświetlany na dwóch wyświetlaczach 7-segmentowych. Ponieważ częstotliwość sygnału zegara (*OSC*) na płytce testowej wynosi 25,175 MHz, należy zaprojektować odpowiedni dzielnik częstotliwości dla sygnału zegarowego ze współczynnikiem podziału np. K = 25175000, aby można było obserwować zmiany stanu licznika z częstotliwością 1 Hz. Dla sygnału sterującego liczeniem (*ENCNT*) należy zastosować synchronizator w celu eliminacji efektu metastabilności. Cały projekt należy zrealizować w układzie EPF10K20RC240-4.

Projekt dzielnika częstotliwości dla K = 25175000

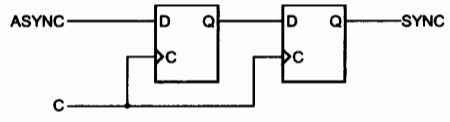
1) Wybierz polecenia *File* → *New* i zaznacz *Text Editor file*. Zatwierdź przyciskiem *OK*.
2) Wybierz polecenie *File* → *Save As* i rozszerzenie *.vhd* w polu *Automatic Extension*.
3) Zapisz plik pod przykładową nazwą *clkdiv* w nowym katalogu np. *cnt8rev*, wpisując w polu *File Name* pełną ścieżkę dostępu *c:\max2work\cnt8rev\clkdiv.vhd*.
4) Ustaw plik *clkdiv.vhd* jako projekt bieżący, wybierając polecenie *File* → *Project* → *Set Project to Current File*.
5) Przygotuj w języku VHDL opis synchronicznego dzielnika częstotliwości o współczynniku podziału K = 25175000 (`L6.3`).
6) Skompiluj projekt *clkdiv*.

```
-- L6.3, czestotliwosc sygnalu OSC = 25,175 MHz
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity clkdiv is
  port(OSC : in std_logic;  -- wejscie zegara
```

```
        F1Hz : out std_logic);  -- wyjscie dzielnika
end clkdiv;
architecture freq_div of clkdiv is
  -- Licznik musi byc co najmniej 25-bitowy,
  -- aby zrealizowac dzielnik z K = 25175000
  signal Q: std_logic_vector (24 downto 0);
begin
  process (OSC)
  begin
    if rising_edge(OSC) then
      if Q < 25174999 then
        Q <= Q + 1;
      else
        Q <= (others => '0');
      end if;
    end if;
  end process;
  -- F1Hz = 1 dla Q = 16777216 ,... , 25174999
  -- wspolczynnik wypelnienia ti/T0 = 0,33
  F1Hz <= Q(24);
end freq_div;
```

Projekt synchronizatora

1) Utwórz nowy plik o przykładowej nazwie *synch.vhd* i ustaw go jako projekt bieżący.
2) Opisz w języku VHDL układ przedstawiony na rys. 6.29.
3) Zapisz plik do folderu *c:\max2work\cnt8rev*.
4) Skompiluj projekt *synch*.



Rys. 6.29 Schemat logiczny synchronizatora

Opis synchronizatora ma postać

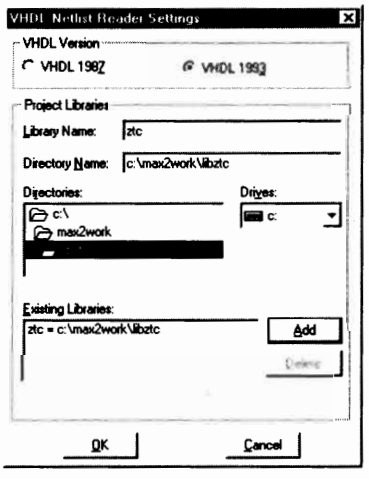
```
-- L6.4
library ieee;
use ieee.std_logic_1164.all;
entity synch is
  port(ASYNC, C : in std_logic;
       SYNC : out std_logic);
end synch;
architecture shift_reg of synch is
  -- signal pomocniczy tq
  signal tq : std_logic_vector(1 downto 0);
begin
```

```
    process (C)
    begin
      if rising_edge (C) then
       tq <= ASYNC & tq(1);
      end if;
    end process;
    SYNC <= tq(0);
  end shift_reg;
```

Moduły licznika rewersyjnego (L6.5) oraz konwertera kodu szesnastkowego na kod wyświetlacza 7-segmentowego (L6.1) są również zawarte w pakiecie wat, który trzeba skompilować do odpowiedniej biblioteki np. ztc. Odpowiednie pliki źródłowe w języku VHDL (*hex7a.vhd, cntnbrev.vhd, wat.vhd*) należy umieścić w przykładowym folderze *c:\max2work\libztc*.

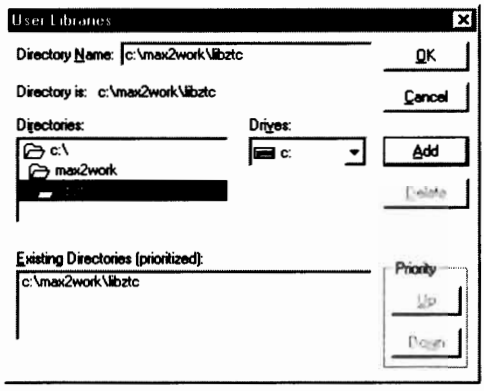
Utworzenie biblioteki ztc wykonuje się w następujący sposób:

1) Wybierz polecenie *File* → *Open* i wczytaj plik *wat.vhd*. Ustaw go jako projekt bieżący.
2) Otwórz okno kompilatora poleceniem *MAX+plus II* → *Compiler*.
3) Wybierz polecenie *Interfaces* → *VHDL Netlist Reader Settings*. W oknie dialogowym *VHDL Netlist Reader Settings* podaj nazwę biblioteki oraz ścieżkę dostępu do biblioteki, jak na rys. 30. Zaznacz opcję wersji języka *VHDL 1993*.
4) Kliknij przycisk *Add*, aby dodać bibliotekę do istniejących (*Existing Libraries*).
5) Zamknij okno *VHDL Netlist Reader Settings* i uruchom kompilator przyciskiem *Start*. Biblioteka ztc zostanie utworzona. Zamknij okno kompilatora.



Rys. 6.30 Okno ustawień bibliotek projektu i wersji języka VHDL

6) Wybierz polecenie *Options* → *User Libraries*. Pojawi się okno dialogowe *User Libraries* (rys. 6.31). W polu *Directories* wybierz folder *libztc*, który zawiera jednostki projektowe komponentów (`hex7a`, `cntnbrev`) użytych w projekcie `cnt8rev`. Przyciśnij *Add*, aby dodać folder *libztc* do istniejących w polu *Existing Directories*. Przyciski *Up* i *Down* służą do ustalenia priorytetu bibliotek użytkownika. Przyciśnij *OK*, aby zatwierdzić wykonane ustawienia.



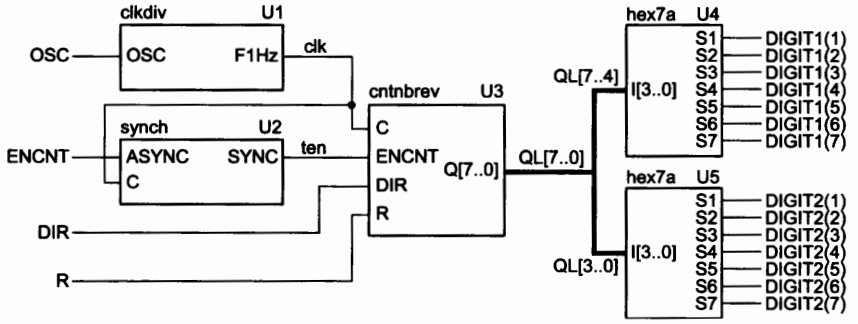
Rys. 6.31 Okno ustawień folderów z bibliotekami użytkownika

```
-- L6.5
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity cntnbrev is
  generic (N: positive := 8); -- deklaracja N = 8 bit
  port(C, R, ENCNT, DIR : in std_logic;
       Q : out std_logic_vector(N-1 downto 0));
end cntnbrev;
architecture cnt_rev of cntnbrev is
  signal cnt: std_logic_vector(N-1 downto 0);
begin
  process (C, R, DIR)
  begin
    if R = '0' then  -- zerowanie asynchroniczne
        cnt <= (others => '0');
      elsif rising_edge(C) then
        if ENCNT = '1' then  -- zezwolenie na liczenie
          case DIR is
           -- granice liczenia ustala parametr N = 8
           -- operacja cnt - 1 jest wykonywana
           -- w kodzie U2
            when '1' => cnt <= cnt + 1;
            when '0' => cnt <= cnt - 1;
            when others => null;
```

```
        end case;
      end if; -- ENCNT = '0' pamietanie poprzed. stanu
    end if;
  end process;
  Q <= cnt;
end cnt_rev;
```

Projekt strukturalny bloku

1) Utwórz nowy plik o nazwie *cnt8rev.vhd* i ustaw go jako projekt bieżący.
2) Opisz układ przy użyciu komponentów zgodnie z rys. 6.32.



Rys. 6.32 Schemat blokowy projektu *cnt8rev*

Opis tej struktury ma postać

```
-- L6.6
library ieee, ztc; -- Deklaracja bibliotek ieee i ztc
use ieee.std_logic_1164.all;
use ztc.wat.all;      -- Klauzula uzycia pakietu wat
entity cnt8rev is
  port (OSC, ENCNT, R, DIR: in std_logic;
    DIGIT1, DIGIT2: out std_logic_vector(7 downto 1));
end cnt8rev;
architecture cnt8rev of cnt8rev is
  -- deklaracje komponentow
  component clkdiv
    port(OSC : in std_logic;
         F1Hz : out std_logic);
  end component;
  component synch
    port(ASYNC, C : in std_logic;
         SYNC : out std_logic);
  end component;
  component cntnbrev
    generic (N: positive:= 8);
    port(C, R, ENCNT, DIR : in std_logic;
         Q : out std_logic_vector(N-1 downto 0));
```

```
  end component;
  component hex7a
    port(I : in std_logic_vector(3 downto 0);
         S : out std_logic_vector(7 downto 1));
  end component;
  -- deklaracje sygnalow pomocniczych
  signal clk, ten : std_logic;
  signal QL : std_logic_vector(7 downto 0);
begin
  U1:clkdiv port map (OSC=>OSC, F1Hz=>clk);
  U2:synch port map (ASYNC=>ENCNT, C=>clk, SYNC=>ten);
  U3:cntnbrev port map (C=>clk, R=>R, ENCNT=>ten,
                        DIR=>DIR, Q=>QL);
  U4:hex7a port map (I=>QL(7 downto 4), S=>DIGIT1);
  U5:hex7a port map (I=>QL(3 downto 0), S=>DIGIT2);
end;
```

3) Przy aktywnym oknie *Compiler*, wybierz z menu polecenie *Interfaces* → *VHDL Netlist Reader Settings* i ustaw bibliotekę `ztc` dostępną dla projektu *cnt8rev*, a następnie wybierz opcję *VHDL 1993* (jak na rys. 6.30).
4) Sprawdź poprawność opisu VHDL poleceniem *File* → *Project* → *Save & Check*.
5) Wybierz układ programowalny *EPF10K20RC240-4* i przyporządkuj końcówki do poszczególnych portów w następujący sposób: `OSC` – 91, `ENCNT` – 41, `R` – 28, `DIR` – 29, `DIGIT1(1)` – 6, `DIGIT 1(2)` – 7, `DIGIT1(3)` – 8, `DIGIT1(4)` – 9, `DIGIT1(5)` – 11, `DIGIT1(6)` – 12, `DIGIT1(7)` – 13, `DIGIT2(1)` – 17, `DIGIT2(2)` – 18, `DIGIT2(3)` – 19, `DIGIT2(4)` – 20, `DIGIT2(5)` – 21, `DIGIT2(6)` – 23, `DIGIT2(7)` – 24.
6) Zapisz plik *cnt8rev.vhd* do folderu *c:\max2work\cnt8rev* i uruchom kompilator.
7) Wykonaj symulację projektu bez dzielnika `clkdiv` (powodem jest zbyt duży współczynnik podziału $K = 25175000$).
8) Zaprogramuj układ.

Program testowy do projektu `cnt8rev`

Na wydruku `L6.7` przedstawiono przykład programu testowego do projektu `cnt8revm` bez dzielnika częstotliwości `clkdiv`.

```
-- L6.7
library ieee;
use ieee.std_logic_1164.all;
entity cnt8rev_tb is end;
architecture cnt8rev_tb of cnt8rev_tb is
```

```
  component cnt8revm
   port(OSC, R, ENCNT, DIR: in std_logic;
    DIGIT1, DIGIT2: out std_logic_vector(7 downto 1));
  end component;
  signal OSC: std_logic := '0';
  signal R, ENCNT, DIR: std_logic := '1';
  signal DIGIT1,DIGIT2: std_logic_vector (6 downto 0);
  type digit is array (natural range <>) of
       std_logic_vector (6 downto 0);
  constant LED : digit := ("1000000",   -- 0
                           "1111001",   -- 1
                           "0100100",   -- 2
                           "0110000",   -- 3
                           "0011001",   -- 4
                           "0010010",   -- 5
                           "0000010",   -- 6
                           "1111000",   -- 7
                           "0000000",   -- 8
                           "0010000",   -- 9
                           "0001000",   -- A
                           "0000011",   -- b
                           "1000110",   -- C
                           "0100001",   -- d
                           "0000110",   -- E
                           "0001110");  -- F
begin
  U1: cnt8revm port map (OSC=>OSC, R=>R, ENCNT=>ENCNT,
            DIR=>DIR, DIGIT1=>DIGIT1, DIGIT2=>DIGIT2);
  OSC <= not (OSC) after 20 ns;  -- zegar 25MHz
  spr: process
    variable errors : boolean := false;
  begin
    -- zerowanie licznika
    R <= '0';
    wait for 30 ns;
    if DIGIT1 /= LED(0) or DIGIT2 /= LED(0) then
      assert false
        report "Licznik nie zostal wyzerowany";
        errors := true;
    end if;
    assert errors
      report "Licznik zostal wyzerowany"
    severity note;
    -- test liczenia w gore
    R <= '1';
    ENCNT <= '1';
    DIR <= '1';
    wait until OSC = '1';
    for j in 0 to 15 loop
      for i in 0 to 15 loop
        wait until OSC = '1';
```

```
            if DIGIT1 /= LED(j) or DIGIT2 /= LED(i) then
              assert false
                report "Bledy przy liczeniu w gore";
                errors := true;
            end if;
            assert errors
              report "Liczenie w gore poprawne"
            severity note;
          end loop;
        end loop;
        -- test liczenia w dol
        R <= '1';
        ENCNT <= '1';
        DIR <= '0';
        wait until OSC = '1';
        for j in 15 downto 0 loop
          for i in 15 downto 0 loop
            wait until OSC = '1';
            if DIGIT1 /= LED(j) or DIGIT2 /= LED(i) then
              assert false
                report "Bledy przy liczeniu w dol";
                errors := true;
            end if;
            assert errors
              report "Liczenie w dol poprawne"
            severity note;
          end loop;
        end loop;
        wait;
      end process spr;
    end cnt8rev_tb;
```

Przedstawiony wyżej program testowy `cnt8rev_tb` do projektu `cnt8revm` można wykonać z użyciem symulatora *Active-HDL* (rozdz. 8).

6.5.2 Jednostka arytmetyczno-logiczna

W niniejszym przykładzie przedstawiono opis VHDL jednostki arytmetyczno-logicznej (ALU – *Arithmetic-Logic Unit*), która wykonuje następujące operacje:

- podstawowe funkcje logiczne: *NOT*, *AND*, *NAND*, *OR*, *NOR*, *XOR* i *XNOR* na *N*-bitowych wektorach *P* i *Q*,
- operacje arytmetyczne: *P* + *Q* i *P* – *Q* na liczbach bez znaku (U2) i ze znakiem (ZU2),
- inkrementację i dekrementację argumentu *P*,
- przesuw w lewo lub w prawo argumentu *P*.

Do sygnalizacji wyników operacji arytmetycznych wprowadzono dodatkowe sygnały wyjściowe (flagi): SGN – bit znaku przy dodawaniu liczb w kodzie ZU2, OV – bit przepełnienia przy dodawaniu liczb w kodzie ZU2, CO – wyjście przeniesienia, ZERO – zerowy wynik operacji. Tablica 6.11 zawiera zestawienie operacji realizowanych przez ALU. Wybór operacji jest realizowany za pomocą czterech wejść sterujących $M_3 - M_0$, tworzących wektor `M(3 downto 0)`.

Tablica 6.11 Operacje ALU

M_3 M_2 M_1 M_0	Operacja
0 0 0 0	P
0 0 0 1	$\overline{P}$
0 0 1 0	PQ
0 0 1 1	$\overline{PQ}$
0 1 0 0	$P \vee Q$
0 1 0 1	$\overline{P \vee Q}$
0 1 0 1	$P \oplus Q$
0 1 1 1	$\overline{P \oplus Q}$
1 0 0 0	P + Q
1 0 0 1	P – Q
1 0 1 0	P + 1
1 0 1 1	P – 1
1 1 0 0	SHL P
1 1 0 1	SHR P
1 1 1 0	0 0 0 0
1 1 1 1	1 1 1 1

System *MAX+plus II 10.1* nie zawiera pakietu `numeric_std`, który zwykle jest stosowany w operacjach numerycznych na danych typu `std_logic_vector`. Zamiast niego użyto pakietu `std_logic_unsigned`, który jest zawarty w systemie. Pakiet ten zawiera deklaracje operatorów i funkcji do obliczeń arytmetycznych na danych typu `std_logic_vector`.

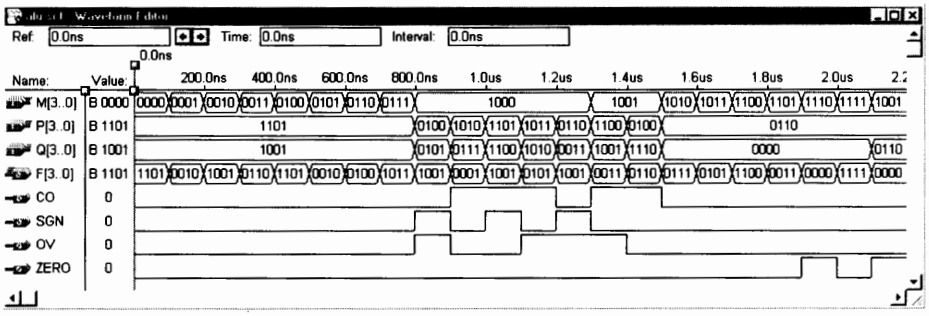
Model *N*-bitowego ALU opisuje się modułem

```
-- L6.8
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity alu is
  -- Deklaracja liczby bitow N
  generic (N: positive := 4);
  port (P, Q : in std_logic_vector (N-1 downto 0);
        M : in std_logic_vector (3 downto 0);
        F : out std_logic_vector (N-1 downto 0);
```

```
        SGN, OV, CO, ZERO : out std_logic);
end alu;
architecture a1 of alu is
  signal tF: std_logic_vector (N-1 downto 0);
  signal tCO: std_logic;
begin
  ALU: process (M, P, Q)
  begin
    case M is
      -- operacje logiczne:
      when "0000" => tF <= P;          -- P
      when "0001" => tF <= not P;      -- /P
      when "0010" => tF <= P and Q;    -- AND
      when "0011" => tF <= P nand Q;   -- NAND
      when "0100" => tF <= P or Q;     -- OR
      when "0101" => tF <= P nor Q;    -- NOR
      when "0110" => tF <= P xor Q;    -- XOR
      when "0111" => tF <= not (P xor Q); -- XNOR
      -- operacje arytmetyczne:
      when "1000" => tF <= P + Q; -- suma P+Q
      when "1001" => tF <= P - Q; -- roznica P-Q
      when "1010" => tF <= P + 1; -- inkrementacja
      when "1011" => tF <= P - 1; -- dekrementacja
      -- SHL P - przesuw w lewo argumentu P
      when "1100" => tF <= P(N-2 downto 0) & '0';
      -- SHR P - przesuw w prawo argumentu P
      when "1101" => tF <= '0' & P(N-1 downto 1);
      -- ustawienie stanu 0 na wyjsciach F
      when "1110" => tF <= (others => '0');
      -- ustawienie stanu 1 na wyjsciach F
      when "1111" => tF <= (others => '1');
      when others => tF <= (others => '0');
    end case;
      F <= tF;
    -- flagi:
    if M > "0111" and M < "1100" then
      -- przeniesienie
      tCO <= (P(N-1) and Q(N-1)) or
             (not tF(N-1) and (P(N-1) xor Q(N-1)));
      SGN <= tF(N-1);  -- bit znaku (ZU2)
      -- przepelnienie (ZU2)
      OV <= tCO xor (tF(N-1) xor P(N-1) xor Q(N-1));
    else
      tCO <= '0';
      SGN <= '0';
      OV <= '0';
    end if;
    CO <= tCO;  -- wyjscie przeniesienia
    if tF = 0 then
       ZERO <= '1';
    else
```

```
        ZERO <= '0';
      end if;
    end process ALU;
end a1;
```

Dla funkcji *XOR* obowiązuje reguła: jeśli $F = P \oplus Q$, to $Q = F \oplus P$. Pozwala to w łatwy sposób wykryć szkodliwe przepełnienie przy dodawaniu liczb w kodzie ZU2. Załóżmy, że dodajemy czterobitowe liczby P i Q w kodzie ZU2, gdzie P_3 i Q_3 są bitami znaku. Otrzymujemy czterobitowy wynik F i przeniesienie CO. Bit znaku wyniku jest równy $SGN = F_3$. Bit przeniesienia z pozycji znaku jest równy $c_z = CO$. Bit przeniesienia z najwyższego bitu modułu sumy jest równy $c_m = F_3 \oplus (P_3 \oplus Q_3)$. Nadmiar powodujący błędny wynik ma miejsce wtedy, gdy $c_z \oplus c_m = 1$. A zatem równanie funkcji logicznej dla sygnału przepełnienia OV można sformułować w postaci $CO \oplus (F_3 \oplus (P_3 \oplus Q_3))$. Natomiast przeniesienie można wyznaczyć jako $CO = (P_3 Q_3 \vee \overline{F}_3 (P_3 \oplus Q_3))$.



Rys. 6.33 Wyniki symulacji funkcjonalnej projektu `alu`

Na rysunku 6.33 pokazano wyniki symulacji funkcjonalnej dla wszystkich operacji realizowanych przez ALU. Na przykład dodajemy dwie liczby w kodzie ZU2: $P = 1.010$ (-6_{10}) i $Q = 0.111$ ($+7_{10}$). Otrzymuje wtedy wynik dodatni $F = 0.001$ ($+1_{10}$) w kodzie ZU2, a odpowiednie bity flag są równe: $SGN = 0$, $OV = 0$, $CO = 1$. Szczegółową analizę pozostałych wyników symulacji oraz napisanie programu testowego dla projektu `alu` pozostawiono Czytelnikowi do samodzielnego wykonania.

W zasobach bibliotecznych systemu jest dostępny również układ ALU 74181 w postaci komponentu (w pakiecie `maxplus2`) oraz symbolu graficznego (w bibliotece *max2lib\mf*).

6.5.3 Moduł pamięci RAM

Biblioteka `lpm` w system *MAX+plus II 10.1* zawiera liczne deklaracje modułów parametryzowanych w postaci komponentów, na przykład bloków pamięci RAM i ROM. Poniżej przedstawiony jest przykładowy opis VHDL (`L6.9`) modułu pamięci RAM 256 × 8 bitów z użyciem pakietu `lpm_components`. Komponent `lpm_ram_dq` jest parametryzowanym modułem pamięci RAM z oddzielnymi portami wejściowymi `data` i wyjściowymi `q`. Może być użyty do realizacji pamięci asynchronicznej lub pamięci z synchronicznymi wejściami i/lub wyjściami. Parametry komponentu są wymienione w liście klauzuli **generic**. Projektant podczas konkretyzacji komponentu określa wartości tych parametrów za pomocą klauzuli **generic map**. W przedstawionym poniżej przykładzie określono tylko liczbę portów wejściowych i wyjściowych (`lpm_width => 8`) oraz liczbę wejść adresowych `address` (`lpm_widthad => 8`). Pozostałe parametry podczas kompilacji przyjmują wartości zdefiniowane w klauzuli **generic**. Na przykład parametry: `lpm_indata`, `lpm_address_control` i `lpm_outdata` określają, czy porty wejściowe, adresowe i wyjściowe są typu rejestrowego (`"REGISTERED"`) lub nie są typu rejestrowego (`"UNREGISTERED"`). Sygnały zegarowe `INCLOCK` i `OUTCLOCK` realizują odpowiednio operację synchronicznego zapisu (gdy `WE = '1'`) i odczytu danych z pamięci.

```
-- L6.9
library ieee;
use ieee.std_logic_1164.all;
-- Deklaracja biblioteki lpm
library lpm;
-- Klauzula uzycia pakietu lpm_components
use lpm.lpm_components.all;
entity ram256x8 is
  port (DATA : in std_logic_vector (7 downto 0);
        ADDR : in std_logic_vector (7 downto 0);
        WE, INCLOCK, OUTCLOCK : in std_logic;
        Q : out std_logic_vector (7 downto 0));
end ram256x8;
architecture a1 of ram256x8 is
  -- Deklaracja komponentu
  component lpm_ram_dq
    generic (lpm_width: positive;
          lpm_type: string := "LPM_RAM_DQ";
          lpm_widthad: positive;
          lpm_numwords: natural := 0;
          lpm_file: string := "UNUSED";
          lpm_indata: string := "REGISTERED";
```

```
            lpm_address_control: string := "REGISTERED";
            lpm_outdata: string := "REGISTERED";
            lpm_hint: string := "UNUSED");
        port (data: in std_logic_vector
                (lpm_width-1 downto 0);
              address: in std_logic_vector
                (lpm_widthad-1 downto 0);
              we: in std_logic;
              inclock: in std_logic := '0';
              outclock: in std_logic := '0';
              q: out std_logic_vector
                (LPM_WIDTH-1 downto 0));
    end component;
  begin
    U1: lpm_ram_dq
      -- Okreslenie rozmiaru wektora danych i adresu
      generic map (lpm_width=>8, lpm_widthad=>8)
      port map (data=>DATA, address=>ADDR, we=>WE,
              inclock=>INCLOCK, outclock=>OUTCLOCK, q=>Q);
  end a1;
```

Wykaz modułów parametryzowanych zawartych w systemie *MAX+plus II* oraz szczegółowy ich opis jest wywoływany poleceniem *Help → Megafunctions/LPM*.

6.6 ZADANIA

1. Zmodyfikować układ sumatora/subtraktora z p. 6.3.2 tak, aby realizował dodawanie/odejmowanie oraz testowanie przepełnienia na wyjściu dla liczb 8-, 16-bitowych w kodzie ZU2.

2. Zaprojektować układ dwupozycyjnego sumatora/subtraktora w kodzie BCD 8421. Operacja odejmowania jest wykonywana w kodzie U9. Wynik działania arytmetycznego jest wyświetlany na wyświetlaczu 7-segmentowym LED.

3. Zaprojektować multiplikator 3×3. Wynik działania arytmetycznego jest wyświetlany na dwóch wyświetlaczach 7-segmentowych LED.

4. Napisać program testowy dla projektu `alu` z p. 6.5.1.

5. Zaprojektować programowany generator odcinków czasu (*timer*) o zakresie do 3 min i rozdzielczości 1 s. Odliczanie nastawionego czasu jest inicjowane sygnałem *Start* = 0. Ustawienie minut (*Min*) i sekund (*Sek*) odmierzanego czasu jest realizowane przez wpis równoległy z wejść *m7..m0* po sygnale *Load* = 0. Po upłynięciu nastawionego czasu na wyjściu *Alarm* pojawia się zbocze narastające (lub impuls dodatni).

7

SYSTEM PROJEKTOWY *IspDesignEXPERT*

Zbigniew Jachna

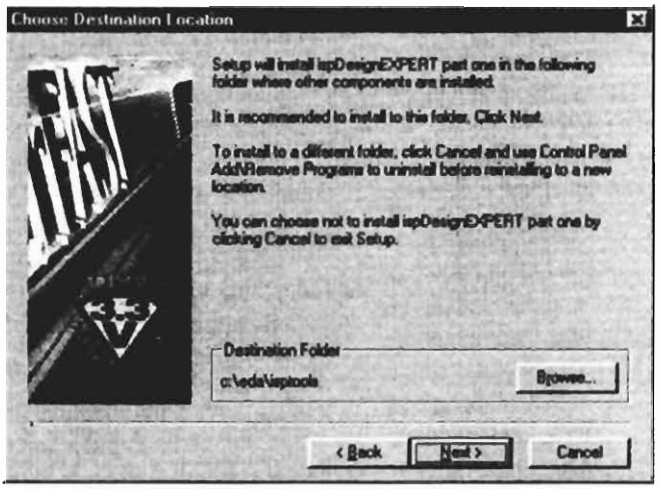
System projektowy *IspDesignEXPERT* firmy *Lattice Semiconductor* służy do projektowania układów cyfrowych w strukturach programowalnych PLD i CPLD. Ten rozdział zawiera opis wersji 8.4, działającej w środowisku Windows 95/98 i Windows NT. Wersja ta zawiera odpowiednie narzędzia projektowe umożliwiające zastosowanie układów ispLSI 1000, 2000, 3000, 5000V, 6000, 8000, GAL, MACH oraz PAL [10,11,17].

System *IspDesignEXPERT* jest dostępny w kilku pakietach różniących się zestawem narzędzi projektowych, wielkością biblioteki układów oraz ceną. Ceny pakietów rozciągają się od 500 do 4000 dolarów. Istnieje również bezpłatna wersja *IspDesignEXPERT Starter*, która jest adresowana głównie do studentów i ewentualnych przyszłych nabywców systemu komercyjnego. Można ją znaleźć w Internecie na stronie www.latticesemi.com. Na tej stronie jest udostępniana również obszerna informacja techniczna o systemach projektowych i układach programowalnych.

Dalej będzie omawiana wersja systemu *IspDesignEXPERT Starter*, która w stosunku do wersji komercyjnych ma pewne ograniczenia. Nie są one jednak istotne wtedy, gdy system jest wykorzystywany do celów dydaktycznych. Za pomocą tego systemu można wykonać wszystkie etapy projektowania (p. rys. 1.2) z użyciem dość dużej liczby układów programowalnych. Z uwagi na tematykę książki szerzej będą opisane zagadnienia związane z językiem VHDL.

7.1 INSTALACJA SYSTEMU

System *IspDesignEXPERT Starter* w wersji 8.4 składa się z czterech wykonywalnych plików instalacyjnych: *dexp84p1.exe* (17.8MB), *dexp84p2.exe* (22.7MB), *dexp84p3.exe* (15.3MB) i *dexp84p4.exe* (6.2MB). Trzy pierwsze pliki zawierają integralną część systemu i muszą być zainstalowane. Czwarty plik, zawierający kompilator języka VHDL *Synplify* firmy *Synplicity*, może być instalowany opcjonalnie. Należy go jednak zainstalować, gdyż będzie wykorzystywany w przykładowych projektach pokazanych dalej.



Rys. 7.1 Okno instalatora systemu *IspDesignEXPERT*

Do konfigurowania układów programowalnych, umieszczonych na płytce drukowanej, potrzebny jest program ładujący *IspVM System*. W tym celu należy ściągnąć i uruchomić plik instalacyjny *ispvmsystem10_b.exe* (10.60 MB). Dodatkowo można ściągnąć również asembler *GASM* udostępniany w postaci archiwum *gdspkg.zip* (265 KB). Asembler ten jest przeznaczony do projektowania przełączników ispGDS i będzie przedstawiony w dalszej części opisu.

Do uruchomienia systemu projektowego jest niezbędna licencja, którą trzeba otrzymać ze strony internetowej producenta.

Podczas instalacji systemu *IspDesignEXPERT* pojawia się okno standardowego instalatora (rys. 7.1), którego obsługa jest intuicyjna. Domyślnie system jest umieszczany w folderze *c:\isptools*. Jest to lokalizacja zalecana dla mniej doświadczonych projektantów. Jednak gdy w jednym komputerze znajduje się kilka różnych systemów projektowych, można je instalować we wspólnym folderze, np. *c:\eda*.

Po zainstalowaniu oprogramowania i ponownym uruchomieniu komputera (ponowne uruchomienie komputera jest konieczne, gdyż program instalacyjny dokonuje zmian w pliku *autoexec.bat*) system projektowy *IspDesignEXPERT Starter* jest gotowy do pracy.

7.2 ORGANIZACJA I ZASOBY BIBLIOTECZNE SYSTEMU

Obsługa systemu projektowego *IspDesignEXPERT* odbywa się za pomocą graficznego interfejsu użytkownika nazywanego nawigatorem projektu (*Project Navigator*). Jego głównym zadaniem jest ułatwienie sposobu projektowania w zakresie doboru odpowiednich narzędzi projektowych i zasobów bibliotecznych. Interfejs ten jest podobny do popularnego w końcu lat 90. systemu *Synario* [13], z którego wywodzi się nazwa rozszerzenia *.syn* stosowana do głównych plików projektowych.

Kompletny projekt składa się z wielu plików. Nazwa ich rozszerzenia mówi zazwyczaj o przeznaczeniu. Przykładowa lista najczęściej stosowanych nazw rozszerzeń plików wygląda następująco:

- *.sch* – schemat,
- *.abl* – opis źródłowy w języku ABEL,
- *.abv* – plik wektorów testowych w języku ABEL,
- *.vhd* – opis źródłowy lub program testowy w języku VHDL,
- *.v* – opis źródłowy w języku Verilog,
- *.ed** – opis połączeń w formacie EDIF,
- *.wdl* – graficzny opis przebiegów czasowych,
- *.pin* – przyporządkowanie końcówek,
- *.jed* – wynikowy plik do programowania.

Projekty tworzone za pomocą systemu *IspDesignEXPERT* mogą być przechowywane w dowolnym folderze na dysku twardym. System pozwala na tworzenie wielu projektów w tym samym folderze. Czasami jest to wygodne, na przykład gdy projekty zawierają takie same części składowe. Jednak bardzo szybko okazuje się, że w folderze zawierającym kilka projektów pojawia się ogromna liczba plików. Próba usunięcia jednego z projektów może wtedy prowadzić do niezamierzonego uszkodzenia innych projektów umieszczonych w tym samym folderze. Dlatego zaleca się, aby każdy projekt znajdował się w osobno deklarowanym folderze. Warto też przy nazywaniu plików starać się nie stosować zbyt długich nazw, zwłaszcza takich, które zawierają znaki niedozwolone w systemie DOS. Z doświadczeń autora wynika, że nie-

które programy systemu nie akceptują wszystkich nazw plików dozwolonych w systemie Windows.

System projektowy *IspDesignEXPERT* zawiera liczne przykłady gotowych projektów. Ich analiza może znacznie ułatwić i przyśpieszyć opanowywanie systemu. Dostęp do przykładów uzyskuje się po uruchomieniu systemu i wybraniu opcji *File*, a następnie *Open Example...*. System kieruje wtedy użytkownika do folderu zawierającego przykłady gotowych projektów zgrupowane w dwóch podfolderach *ispLSI_GAL* i *Mach_pal*. Dodatkowo w folderze *isptools\ispsys\tutorial* można znaleźć podręcznik do nauki obsługi systemu *ispDesignExpert Tutorial* (*ispdegs.pdf*) zawierający dwanaście ściśle zaplanowanych czasowo lekcji tematycznych. Przeglądanie tego podręcznika można rozpocząć za pomocą menu nawigatora projektu wybierając opcję *Help* a następnie *Tutorials*. Za pomocą opcji *Help* możliwe jest również przeglądanie standardowego pliku pomocy w stylu Windows oraz dostęp do dokumentów zamieszczonych na stronie WWW producenta.

Opisywany system projektowy jest systemem hybrydowym, składającym się z różnych narzędzi programowych. Mechanizm wymiany danych między poszczególnymi narzędziami wykorzystuje pliki o różnych formatach zapisu danych. Często stosowanym formatem jest format EDIF, który jest m.in. stosowany do zapisu wyniku syntezy układu w postaci listy połączeń prostych komponentów bibliotecznych, tzw. prymitywów. Format ten stosuje też kompilator języka VHDL *Synplify* wchodzący w skład systemu projektowego.

System projektowy pozwala na tworzenie czterech typów projektów: *Schematic/ABEL*, *Schematic/VHDL*, *Schematic/Verilog HDL* oraz *EDIF*. W tym opisie będzie stosowany typ *Schematic/VHDL*.

Rysowanie schematów odbywa się z użyciem symboli elementów logicznych zawartych w bibliotekach (pliki z rozszerzeniem *.lib*). W folderze *generic* znajdują się biblioteki symboli elementów uniwersalnych (*misc.lib*, *arith.lib*, *gates.lib*, *iopads.lib* i *regs.lib*), które mogą być użyte we wszystkich projektach. Anglojęzyczne nazwy bibliotek kojarzą się z ich zawartością, np. biblioteka *regs.lib* zawiera różne typy przerzutników. Oprócz folderu *generic* istnieją również foldery zawierające symbole elementów przeznaczonych tylko dla określonej grupy układów programowalnych, np. folder *plsi* zawiera biblioteki przeznaczone dla układów ispLSI: *arith.lib*, *coder.lib*, *counter.lib*, *gate.lib*, *iopin.lib*, *mux.lib* i *register.lib*. Symbole z tych bibliotek dotyczą modeli wykorzystujących unikalne właściwości danej grupy układów. W związku z tym, powinny być stosowane tylko w projektach, w których nie przewiduje się użycia układów należących do innej grupy.

Osobną biblioteką jest biblioteka (*Local*), która jest tworzona lokalnie w folderze projektu. Zawiera ona symbole modułów projektu tworzone na podstawie opisu VHDL lub schematu.

Zasoby systemu związane ze stosowaniem języka VHDL dotyczą programu syntezy Synplify. Są to biblioteki standardowe IEEE 1076-1993 i 1164 oraz kilka bibliotek niestandardowych. Występują one w postaci skompilowanej jak i nieskompilowanej (tablica 7.1). Postać nieskompilowana może być potrzebna do symulacji z użyciem symulatora VHDL.

Tablica 7.1 Wykaz bibliotek VHDL stosowanych w systemie *IspDesignEXPERT*

Nazwa biblioteki i pakietu	Zawartość
`Std` `- standard`	Typy predefiniowane i funkcje standardowe

`IEEE`
`- std_logic_1164`
`- numeric_std,`
	Typy std_logic, std_ulogic, signed, unsigned, funkcje konwersji typów, funkcje logiczne i arytmetyczne

`IEEE`
`- std_logic_unsigned,`
`- std_logic_signed,`
`- std_logic_arith,`
	Typy std_logic, std_ulogic, signed, unsigned, funkcje konwersji typów, funkcje logiczne i arytmetyczne opracowane przez firmę *Synopsys*

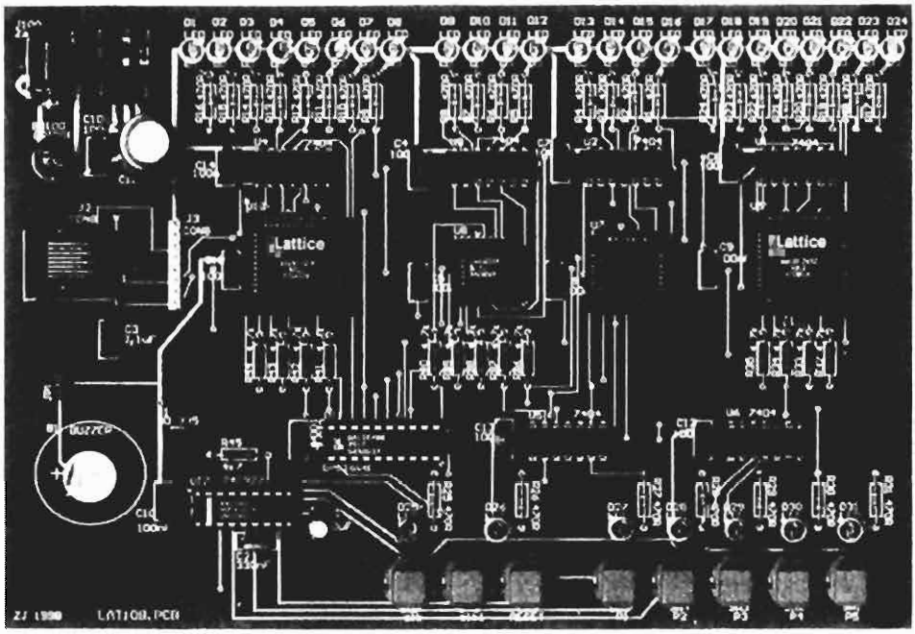
`Synplify` `- attributes`	Atrybuty projektowe wykorzystywane przez kompilator *Synplify*
`Lpm` `- lpm_components`	Parametryzowane makra (przerzutniki, liczniki, sumatory, komparatory, multiplikatory, pamięci, dekodery itd.)
`Lattice` `- components`	Komponenty (bramki, liczniki, przerzutniki itd.) rozpoznawane przez kompilator *Synplify* jako tzw. „czarne skrzynki"

Sposób korzystania z wymienionych pakietów jest standardowy, to znaczy, że używa się deklaracji bibliotek **`library`** oraz klauzuli **`use`** w nagłówkach plików VHDL. Opis pakietów standardowych został przedstawiony w czterech pierwszych rozdziałach książki. Na etapie poznawania systemu projektowego nie ma potrzeby sięgania do innych pakietów. Jednak po nabyciu pewnych doświadczeń można wykorzystywać również i pakiety niestandardowe. W tym celu może być pomocny podręcznik *IspEXPERT Compiler and Synplicity Design Manual* udostępniony na stronie internetowej producenta systemu projektowego (www.latticesemi.com).

7.3 PŁYTKA TESTOWA

Do praktycznego testowania projektów utworzonych za pomocą opisywanego oprogramowania *IspDesignEXPERT* stosuje się płytki testowe. Widok płytki testowej LAT10, zaprojektowanej przez autora, pokazano na rys. 7.2. Znajdują się na niej cztery reprogramowalne układy ISP (*In System Programmable*) firmy Lattice:

- ispGAL22V10 – z serii układów GAL,
- ispGDS14 – z serii przełączników programowalnych,
- ispLSI1016 i ispLSI2032 – z serii układów CPLD.



Rys. 7.2 Widok płytki testowej LAT10

Na płytce LAT10 znajduje się również układ programowany w sposób tradycyjny (z użyciem programatora) – GAL16V8. Układ ten nie będzie obiektem testów, gdyż jego zadaniem jest generowanie odpowiednich sygnałów sterujących do testowania wymienionych wcześniej czterech układów ISP. Tablica 7.2 zawiera zestawienie podstawowych parametrów zastosowanych układów programowalnych.

Układy programowalne umieszczone na płytce cechują się stosunkowo małym stopniem złożoności, mieszczącym się w zakresie od 300 do 2000 bramek. Liczby określające zasoby logiczne, do których zalicza się liczbę przerzutników bądź makrokomórek, liczbę wejść i wyjść,

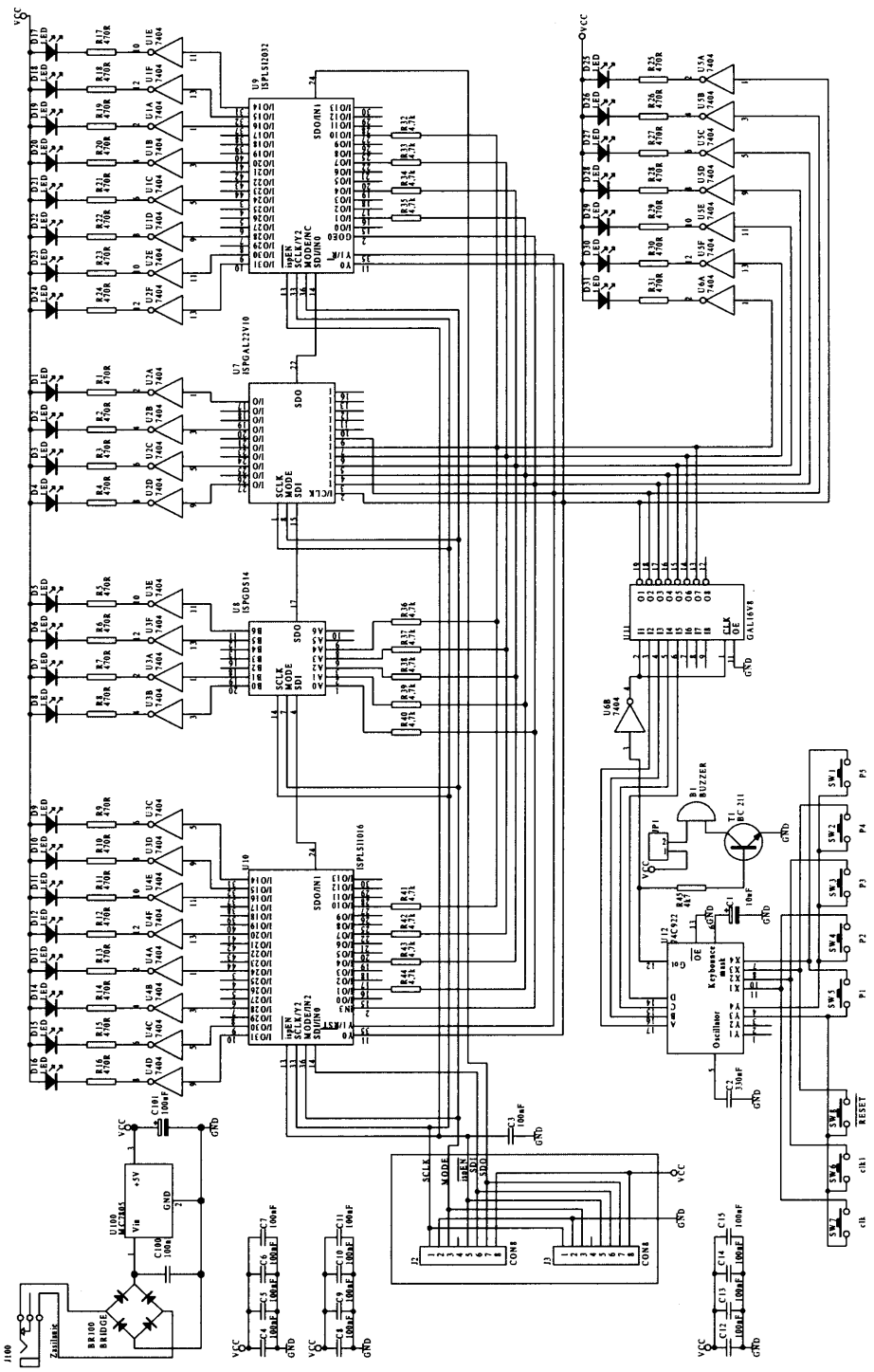
wielkość i liczbę matryc programowalnych itp., decydują o tym, jak duże projekty można realizować. Należy to uwzględnić podczas projektowania z użyciem płytki LAT10.

Tablica 7.2 Parametry układów programowalnych umieszczonych na płytce LAT10

CECHA	GAL 16V8	IspGAL 22V10	IspLSI 1016	IspLSI 2032	IspGDS 14
Isp	nie	tak	tak	tak	tak
Złożoność (liczba bramek)	300	500	2000	1000	7×7 (matryca)
Czas propagacji (ns)	3,5	7,5	7,5	5	7,5
Częstotliwość maksymalna (MHz)	250	111	125	180	50
Liczba makrokomórek	8	10	64	32	----
Liczba przerzutników	8	10	96	32	----
Liczba wyprowadzeń I/O	18	22	36	34	14
Opcje 5 V i 3.3 V	tak	tak	nie	tak	nie
Liczba cykli reprogramowania	100	10000	1000	1000	10000
Typy obudów	PDP, PLCC SOIC, LCC	PLCC, SSOP	PLCC, TQFP, JLCC	PLCC, TQFP	PDIP, PLCC

Schemat elektryczny płytki LAT10 pokazano na rysunku 7.3. Płytka jest zasilana z zewnętrznego zasilacza i umożliwia programowanie umieszczonych na niej układów poprzez łącze JTAG. Sterowanie układów odbywa się za pomocą ośmiu klawiszy *SW1÷SW8*, natomiast sygnalizacja stanów wyjściowych przez 31 diod świecących LED. Do rozpoznawania wciśniętych klawiszy zastosowano układ klawiaturowy 74922, który – zapewniając eliminację drgań styków – generuje na wyjściu sygnał naciśniętego klawisza. Następnie sygnał ten podawany jest do układu GAL16V8, który steruje wejściami badanych układów reprogramowalnych. Stany wejść tych układów są sygnalizowane siedmioma diodami LED (*D25÷D31*).

Podczas tworzenia projektów z użyciem płytki LAT10 należy pamiętać o właściwym przyporządkowaniu sygnałów odpowiednim końcówkom układu. „Sztywne" przyporządkowanie wyprowadzeń dotyczy zarówno sygnałów wejściowych jak i wyjściowych. Tablica 7.3 zawiera przyporządkowanie sygnałów generowanych za pośrednictwem klawiszy do końcówek poszczególnych układów.



Rys. 7.3. Schemat elektryczny płytki testowej LAT10
(projekt: Zbigniew Jachna, zjachna@wel.wat.waw.pl)

Tablica 7.3 Przyporządkowanie klawiszy do końcówek układów programowalnych

Nazwa klawisza	Numer końcówki układu			
	IspLSI 1016	IspGDS 14	IspGAL 22V10	IspLSI 2032
clk	11		2	11
clk1	35		9	35
$\overline{\text{RESET}}$	35		9	35
P1	2	1	3	2
P2	16	2	4	16
P3	19	3	5	19
P4	22	6	6	22
P5	27	8	7	27

Przyciskom umieszczonym na płytce przypisano następujące funkcje:

- clk, clk1 – generacja impulsu ⎍,
- $\overline{\text{RESET}}$ – generacja impulsu ⎎,
- P1+ P5 – ustawianie poziomów logicznych 'L' i 'H'.

Każde naciśnięcie przycisku może być sygnalizowane za pomocą sygnalizatora dźwiękowego *B1*, dołączanego po zwarciu zworki *JP1*.

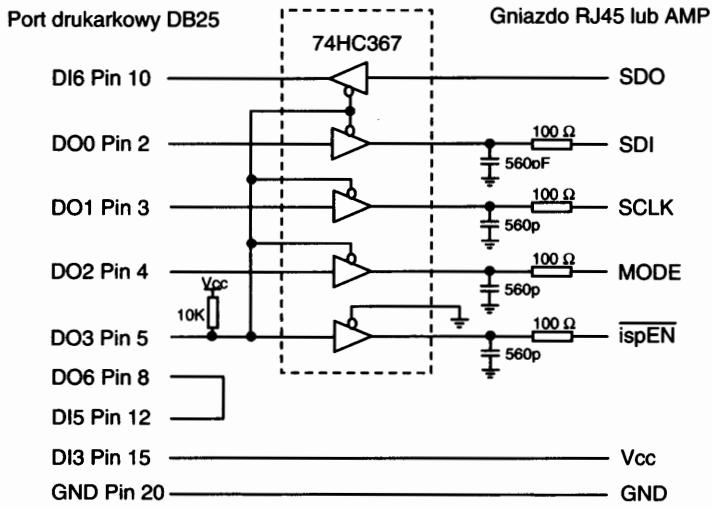
Stan wyjść układów testowych jest sygnalizowany za pomocą diod LED *D1+D24* dołączonych poprzez bufory 7404. Tablica 7.4 zawiera przyporządkowanie numerów końcówek układów do odpowiadających im diod świecących.

Tablica 7.4 Przyporządkowanie wyjść układów programowalnych do diod LED

Numer końcówki układu – numer diody LED			
IspLSI 1016	IspGDS 14	IspGAL 22V10	IspLSI 2032
10 – D16	20 – D8	27 – D4	10 – D24
9 – D15	19 – D7	25 – D3	9 – D23
7 – D14	12 – D6	20 – D2	7 – D22
3 – D13	11 – D5	17 – D1	3 – D21
41 – D12			41 – D20
37 – D11			37 – D19
32 – D10			32 – D18
31 – D9			31 – D17

Płytka testowa LAT10 jest zabezpieczona przed uszkodzeniem wejść, które może wystąpić na skutek pomyłkowego zaprogramowania ich jako wyjść. Do tego celu służą rezystory *R32 ÷ R44*, ograniczające wartości płynących przez nie prądów.

Przeznaczone do programowania końcówki testowanych układów są przyłączone do dwóch gniazd (jedno typu AMP, drugie typu RJ45). Za pośrednictwem dowolnego z nich łączy się płytkę LAT10 z komputerem za pomocą dedykowanego interfejsu (rys. 7.4).



Rys. 7.4 Schemat interfejsu

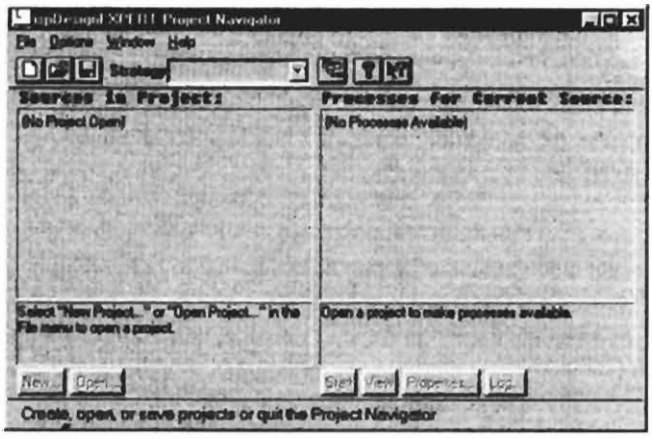
7.4 PROCES PROJEKTOWANIA

Proces projektowania układu za pomocą systemu *IspDesignEXPERT* zostanie pokazany na przykładzie 8-bitowego multipleksera grupowego *4 na 1* (p. L3.3).

7.4.1 Uruchomienie systemu

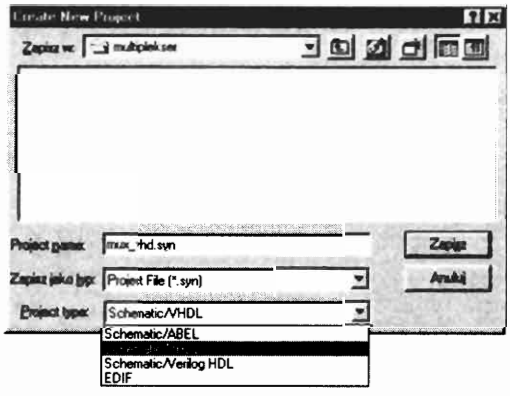
Uruchomienie nawigatora projektu następuje z menu *START* przez wybór pozycji *Programy → Lattice Semiconductor → IspDesignEXPERT System*. Na ekranie pojawia się okno nawigatora (rys. 7.5). Okno jest podzielone na dwa obszary *Sources in Project:* oraz *Processes for Cur-*

rent Source. W obszarze *Sources in Project*: są wyświetlane dane dotyczące źródeł projektu (nazwa, typ układu programowalnego) i jego elementów składowych (nazwy plików źródłowych, nazwy poszczególnych bloków układu w projekcie hierarchicznym, wektory testowe). W obszarze *Processes for Current Source* są wyświetlane procesy. Są to polecenia uruchamiające (przyciskiem *Start*) przetwarzanie plików źródłowych wskazanych w obszarze *Sources in Project* lub też polecenia aktywujące przeglądanie raportów generowanych w trakcie takiego przetwarzania (przyciskiem *View*).



Rys. 7.5 Okno programu zarządzającego – nawigatora projektu

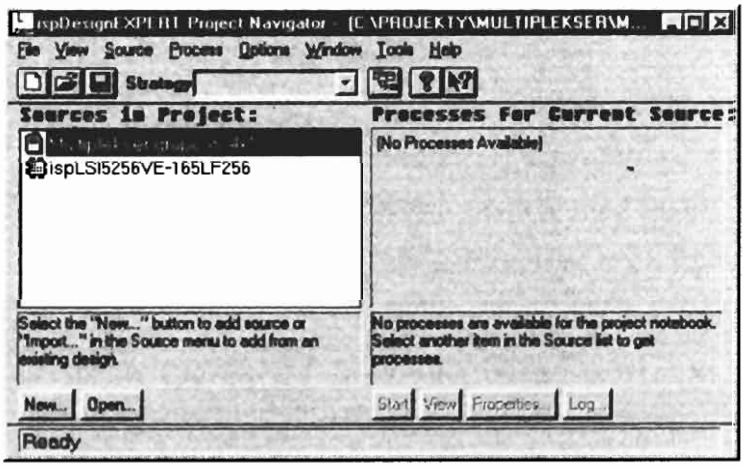
Nowy projekt rozpoczynamy wybierając opcję *New Project...* z menu *File*, po czym na ekranie pojawia się okno *Create New Project* (rys. 7.6). Teraz należy wpisać nazwę projektu (*Project name:*) oraz określić jego typ (*Zapisz jako typ:*). Jako nazwę wpisujemy `mux_vhd`. Ustalamy też lokalizację projektu tworząc nowy folder `multiplekser`.



Rys. 7.6 Okno tworzenia nowego projektu

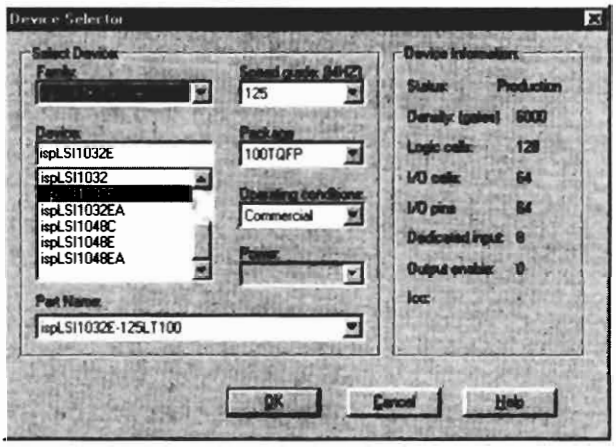
Typ projektu decyduje o tym, jakie sposoby opisu będziemy stosować. W tym opisie będzie to *Schematic/VHDL*. Po zatwierdzeniu ustawień (*Zapisz*), pojawia się okno nawigatora projektu z nazwą pliku o rozszerzeniu *.syn* w pasku tytułowym. Plik ten jest głównym plikiem projektu i w nim są zawarte informacje potrzebne do określenia kompletnej struktury projektu.

W lewym obszarze okna nawigatora są tylko dwa wiersze: *Untitled* i *ispLSI5256VE–165LF256*. Pojawiają się one zawsze, gdy tworzony jest nowy projekt. Po wybraniu *Untitled* i naciśnięciu przycisku *Open* (lub dwukrotne kliknięcie lewym klawiszem myszki) otwiera się okno dialogowe *Project Properties*, w którym wprowadza się nazwę nowego projektu (*Multiplekser grupowy 4x1*). Po zatwierdzeniu (*OK*) pojawia się okno nawigatora pokazane na rys. 7.7. Nazwa projektu służy tylko do identyfikacji, toteż nie ma konieczności jej precyzowania w każdym projekcie. Jednakże stosowanie takich nazw jest zalecane przez producenta systemu.



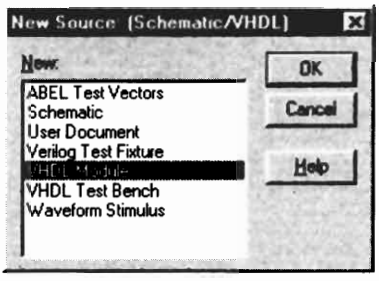
Rys. 7.7 Okno nawigatora po wpisaniu nazwy projektu

Kolejny krok w przygotowaniu projektu to określenie typu układu programowalnego. Po wybraniu wiersza z domyślną nazwą układu *ispLSI5384V–125LB388* otwiera się okno bibliotek elementów (rys. 7.8). Należy wskazać rodzinę układów (*Family:*), a następnie wybrać układ (*Device:*) i nacisnąć *OK*. System nie ma możliwości automatycznego wyboru układu. Projektant powinien więc wstępnie oszacować złożoność projektu i wybrać układ programowalny, który będzie najlepiej spełniał założenia projektowe. Przy wyborze należy również uwzględnić ceny układów.



Rys. 7.8 Okno wyboru układu programowalnego

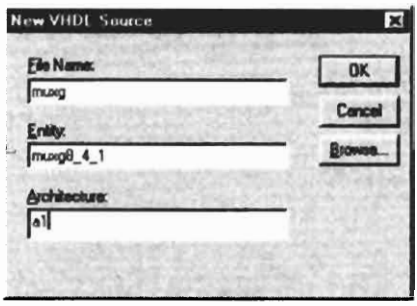
Pliki źródłowe z opisem projektu tworzy się i dołącza korzystając z opcji *Source → New...* . Pojawia się wówczas okno wyboru typu opisu (rys. 7.9) w postaci listy, która zawiera siedem pozycji: *VHDL Module* – do opisu modułu projektu w języku VHDL, *Schematic* – do rysowania schematów, *ABEL Test Vectors*, *Verilog Text Fixture*, *VHDL Test Bench*, *Waveform Stimulus* – do programów testowych i symulacji oraz *User Document* – do tworzenia plików tekstowych. Na podstawie dokonanego wyboru uruchamiany jest właściwy edytor tekstowy do opisów HDL lub edytor graficzny do edycji schematów lub przebiegów czasowych.



Rys. 7.9 Okno wyboru sposobu opisu projektu

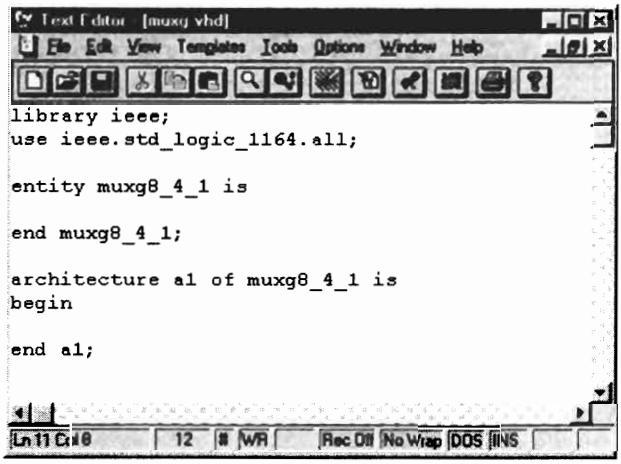
7.4.2 Model multipleksera grupowego

Opis układu w języku VHDL rozpoczynamy wybierając opcję *Source → New* oraz określając typ pliku *VHDL Module*. Po pojawieniu się okna *New VHDL Source* (rys. 7.10) wpisujemy nazwę pliku `muxg`, nazwę modułu `muxg8_4_1` oraz nazwę architektury `a1`. System w sposób automatyczny dopisuje do nazwy pliku rozszerzenie *.vhd*.



Rys. 7.10 Okno dołączania pliku *.vhd*

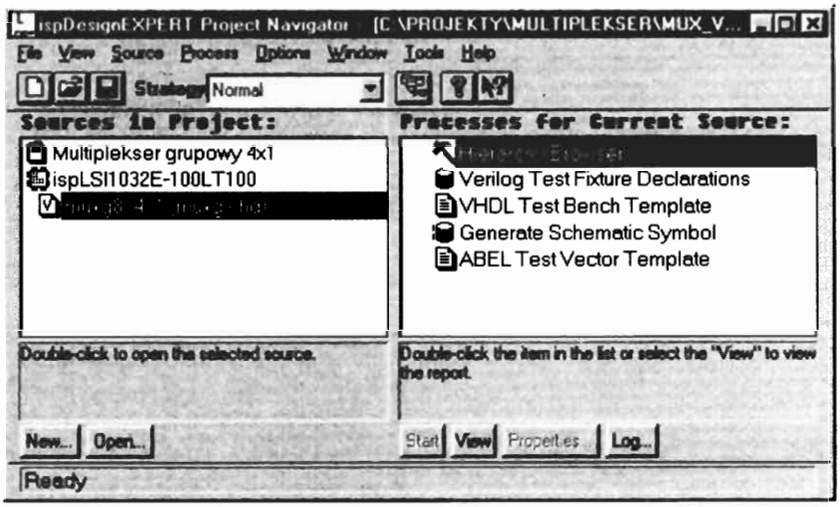
Na podstawie wprowadzonych danych po naciśnięciu przycisku *OK* pojawia się okno edytora tekstowego z automatycznie wygenerowanym szablonem modułu multipleksera (rys. 7.11) Opis ten należy uzupełnić.



Rys. 7.11 Okno edytora VHDL

W lewym obszarze nawigatora pojawia się wiersz reprezentujący stworzony przed chwilą plik. Natomiast w prawym obszarze pojawiają się wiersze aktywujące odpowiednie procesy, które można wykonać z użyciem tego pliku (rys. 7.12).

W praktyce często podczas tworzenia nowego projektu wykorzystuje się fragmenty innych projektów. Wtedy potrzebne pliki źródłowe dołącza się w inny sposób poprzez wybór opcji *Source → Import....* Po otwarciu okna *Import File* należy odszukać i zaznaczyć dołączane pliki. Zatwierdzenie przyciskiem *Otwórz* powoduje, że wybrane pliki stają się dostępne do edycji bezpośrednio w oknie nawigatora projektu.



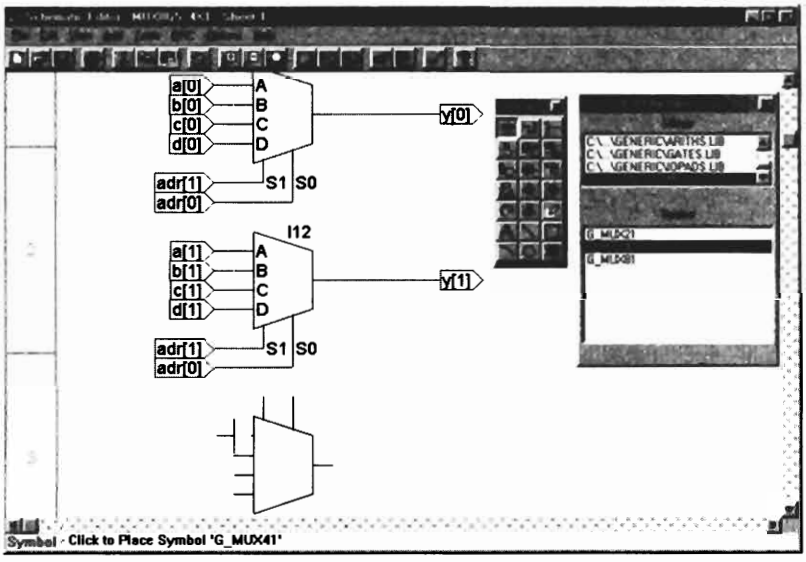
Rys. 7.12 Okno po wczytaniu pliku *.vhd*

Z poziomu nawigatora można uruchomić edytor w celu dokonania poprawek poprzez wskazanie wiersza z nazwą pliku umieszczoną w nawiasie i naciśnięcie przycisku *Open...* lub dwukrotne kliknięcie myszką na wskazanym obiekcie.

W polu *Processes for Current Source* znajduje się pięć opcji: *Hierarchy Browser* – do przeglądania hierarchii modułu, *Verilog Test Fixture Declarations*, *VHDL Test Bench Template* i *ABEL Test Template* – do szybkiego wspomagania pisania testów oraz *Generate Schematic Symbol* – do automatycznego tworzenia symbolu schematowego.

7.4.3 Opis schematowy

Projekt można również opisać graficznie w postaci schematu logicznego. Aby narysować schemat układu należy wybrać opcję *Source* → *New* i uruchomić edytor schematów wybierając wiersz *Schematic* w oknie dialogowym. Po wpisaniu nazwy pliku poleceniem *View* → *Drawing Toolbar* wyświetla się okno z ikonami narzędziowymi do rysowania schematów. Wskazanie ikony powoduje pojawienie się jej opisu w dolnej części okna edytora. Po wybraniu ikony elementów logicznych otwiera się okno biblioteki (rys. 7.13).



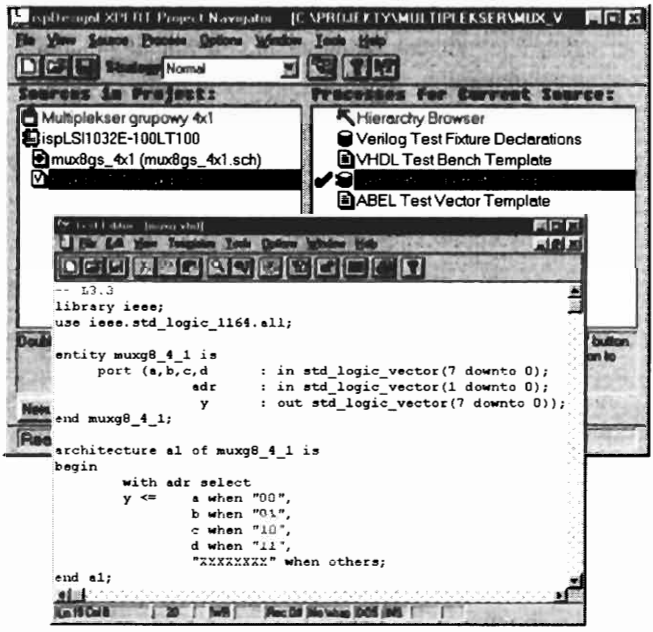
Rys. 7.13 Okno edytora schematowego podczas wstawiania symbolu z biblioteki elementów

Po wybraniu elementu w polu *Symbol*, można go umieścić na schemacie przeciągnięciem myszki w żądane miejsce i kliknięciem. Elementy łączymy połączeniami drutowymi lub magistralowymi po wybraniu odpowiednich ikon. Można nadać nazwy poszczególnym połączeniom oraz wprowadzić specjalne znaczniki portów I/O. Edytor schematowy umożliwia również dodawanie atrybutów związanych z doprowadzeniami, połączeniami lub symbolami. Atrybuty te są wykorzystywane w procesach optymalizacji i dopasowania. Użyteczną opcją jest *DRC → Consistency Check*, która służy do automatycznego wykrywania błędów na schemacie. Można wygenerować symbol schematu, który może być użyty na wyższych poziomach hierarchii projektu. Realizuje się to za pomocą opcji *File → Matching symbol* edytora schematowego lub *Generate Schematic Symbol* w nawigatorze projektu. Utworzony symbol jest wówczas umieszczany w lokalnej bibliotece elementów symboli *Symbol Libraries*.

7.4.4 Opis mieszany

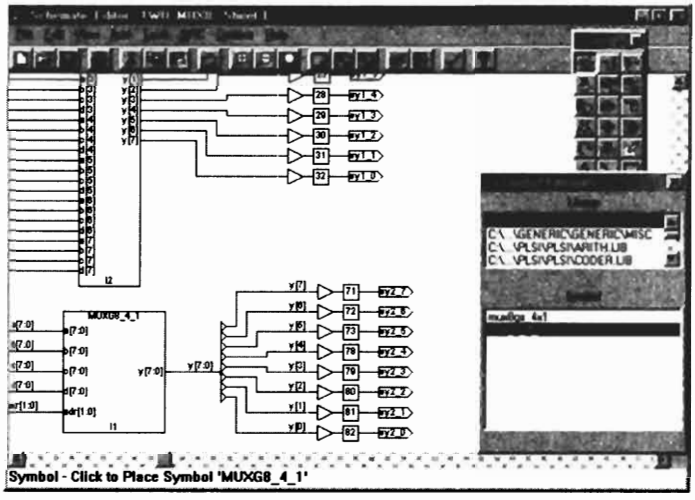
Złożone projekty mogą być opisywane hierarchicznie z zastosowaniem różnych metod opisu poszczególnych jego bloków. W takich projektach na najwyższym poziomie stosowany jest schemat blokowy, natomiast na niższych poziomach bloki mogą być opisywane za pomocą schema-

tu lub języka HDL. Dla każdego z tych opisów należy przygotować symbol schematowy (rys. 7.14).



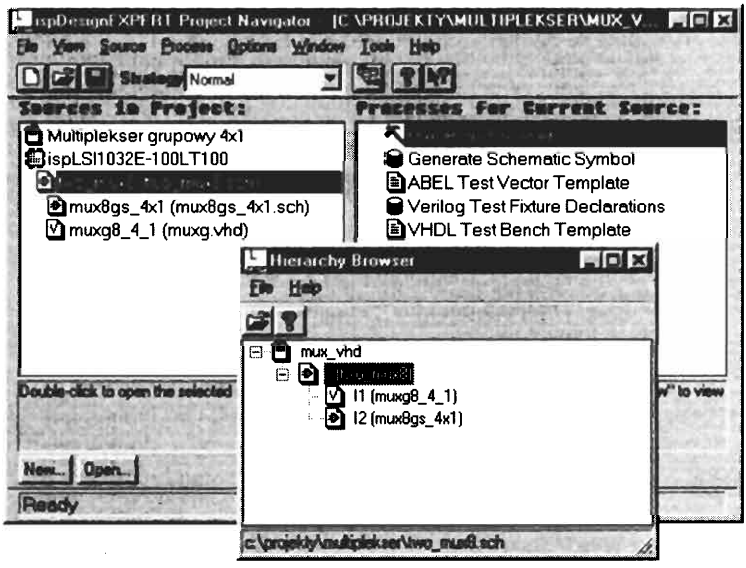
Rys. 7.14 Tworzenie symbolu elementu muxg8_4_1

Utworzone symbole blokowe umieszczane są w lokalnej bibliotece symboli (*Local*) i mogą być używane w schematach o wyższym poziomie hierarchii (rys. 7.15).



Rys. 7.15 Przykład schematowego opisu zadania, w którym użyto symboli dwóch 8-bitowych multiplekserów grupowych

Projekt w postaci schematu, na którym umieszczono dwa multipleksery grupowe, ma strukturę hierarchiczną. Na rysunku 7.16 pokazano okno nawigatora oraz okno prezentera hierarchii.



Rys. 7.16 Przykładowe okno z hierarchią projektu

7.4.5 Określanie wymagań projektowych

Podczas projektowania układu można określać wymagania projektowe za pomocą dodatkowych atrybutów projektowych. Są to predefiniowane słowa, które wpływają na jakość implementacji. Zestawienie przykładowych atrybutów zawiera tablica 7.5.

Tablica 7.5 Zestawienie przykładowych atrybutów systemu *IspDesignExpert*

Nazwa atrybutu	Opis	Przykładowe wartości
CLK	przypisanie zegara	CLK0, CLK1,CLK2
GROUP	nazwa grupy	GRUPA_G1
PRESERVE	zarezerwowanie połączenia	Y lub N
SAP/EAP	początek i koniec ścieżki asynchronicznej	S,nazwa/E,nazwa
SCP/ECP	początek i koniec ścieżki krytycznej	S,nazwa/E,nazwa
PROTECT	zarezerwowanie symbolu	Y lub N
XOR	użycie makra hard XOR	Y lub N
REGTYPE	przypisanie typu przerzutnika	GLB lub IOC
LOCK	przypisanie numeru końcówki	numer
OPTIMIZE	wybór rodzaju makra soft/hard	Y lub N
PULL	dołączenie rezystora podciągającego	UPP, OFF, HOLD

Atrybuty mogą być związane z realizacją całego projektu lub też poszczególnych jego fragmentów. Najmniejszymi jednostkami, którym nadaje się atrybuty są: symbol schematowy, połączenie i końcówka I/O. Istnieje możliwość tworzenia grup złożonych z wielu elementów i nadawania atrybutów grupie. Największą grupą jest cały projekt. Dokładny opis wymienionych atrybutów jest podany w podręczniku użytkownika systemu *„IspDesignEXPERT User Manual", Lattice Semiconductor Corporation, 1999.* Dalej jest pokazany sposób użycia najczęściej stosowanego atrybutu LOCK, którego zadaniem jest określenie końcówek układu.

Atrybuty można wprowadzać do projektu kilkoma sposobami: za pomocą edytora schematowego, w postaci odpowiednich zapisów w modelach VHDL i za pomocą specjalnych plików *.ppr* i *.ppn* zawierających wyłącznie atrybuty projektowe. W edytorze schematowym wprowadzanie atrybutów wspomagane jest poprzez okno kontekstowe, które wyświetla nazwy atrybutów dostępnych do edycji dla wybranego elementu schematowego.

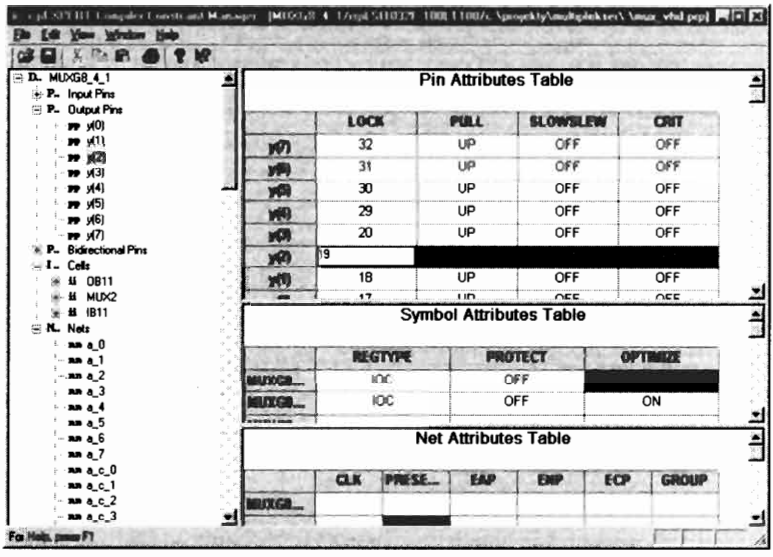
W języku VHDL przypisanie odpowiednich numerów wyprowadzeń układu realizuje się następująco:

```
entity sum1 is
  port( a,b,ci:     in std_logic;
        s,co: out std_logic);
  attribute loc: string;
  attribute loc of a,b,ci: signal is "P5,P6,P7";
  attribute loc of s:  signal is "P8";
  attribute loc of co: signal is "P9";
end;
```

W opisie VHDL atrybut LOCK jest stosowany przy użyciu trzyliterowego skrótu loc, a numery wyprowadzeń są poprzedzone literą P (od słowa pin - końcówka). W łańcuchu znakowym będącym wartością atrybutu nie wolno używać spacji. Zaletą wprowadzania atrybutów projektowych za pomocą osobnych plików tekstowych jest możliwość umieszczenia wszystkich atrybutów w jednym miejscu. Pliki te można tworzyć za pomocą narzędzia *Constraint Manager*, wywoływanego w oknie nawigatora (rys. 7.17).

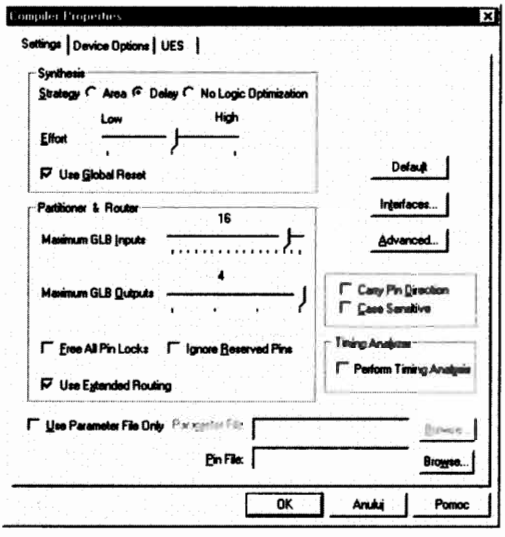
Zamiast stosowania atrybutów, wystarczy dokonać odpowiednich ustawień parametrów przetwarzania za pomocą okna *Properties* (rys. 7.18). Zawartość tego okna nie jest stała, gdyż zależy ona od wybranego układu programowalnego. Przykładowe parametry, które można ustawić za pomocą tego okna to strategia przetwarzania, maksymalna

liczba wejść i wyjść w blokach logicznych oraz rezerwowanie końcówek układu.



Rys. 7.17 Wprowadzanie atrybutów

Strategia przetwarzania określa sposób optymalizacji układu. W oknie *Properties* można wybrać minimalizację powierzchni (*Area*) zajmowanej przez projektowany układ lub minimalizację opóźnień (*Delay*). Zaleca się, aby zawsze stosować jedną z tych strategii.

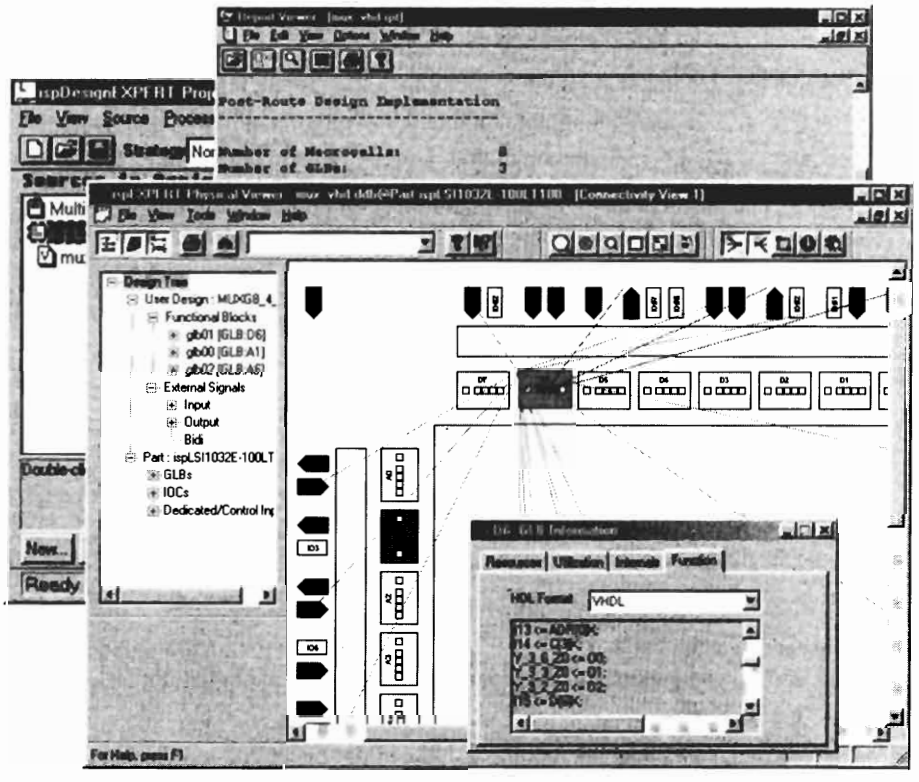


Rys. 7.18 Przykład ustawiania parametrów przetwarzania za pomocą okna *Properties*

7.4.6 Kompilacja projektu i tworzenie plików wynikowych

Po zaznaczeniu typu układu w lewym oknie nawigatora, w prawym oknie pojawi się lista czynności projektowych. Wśród nich znajdują się: *Update All Schematic Files* – do aktualizacji plików schematowych gdy korzystamy z opisu schematowego, *Merged EDIF Netlist* – do scalania projektu hierarchicznego i *Compile Design* – do wpasowania projektu w układ programowalny i wytworzenia pliku konfiguracyjnego. Plik konfiguracyjny ma rozszerzenie *.jed* i zawiera mapę programowania w standardzie JEDEC [17].

Po zakończeniu procesu kompilacji generowany jest raport, który można obejrzeć po podwójnym kliknięciu na pozycji *Compiler Report*. Ponadto można obejrzeć graficzne rozmieszczenie projektu w strukturze programowalnej po użyciu opcji *Physical Viewer* (rys. 7.19).

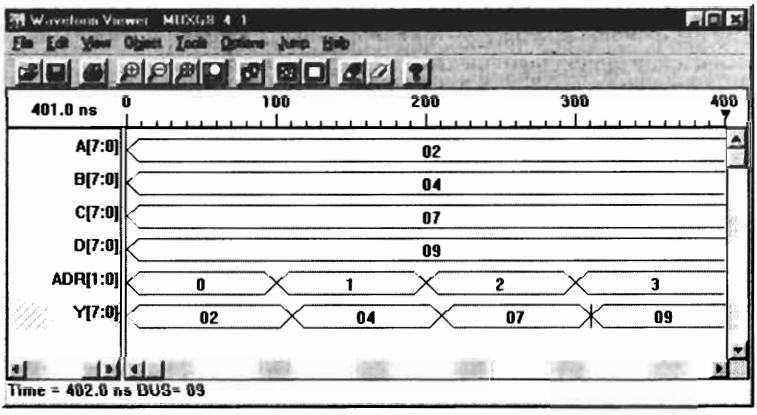


Rys. 7.19 Przykładowy wygląd ekranu po kompilacji projektu

W wyniku syntezy tworzony jest również model czasowy układu. Model ten może być wykorzystany do symulacji dynamicznej oraz analizy czasowej.

7.4.7 Symulacja funkcjonalna i czasowa

System projektowy jest wyposażony w symulator umożliwiający wykonanie symulacji funkcjonalnej i czasowej. Symulacja funkcjonalna ma na celu sprawdzenie projektu bez uwzględniania czasów propagacji wybranego układu programowalnego, czyli przed uruchomieniem fitera za pomocą procesu *Compile Design*. W tym systemie proces *Compile Design* może być traktowany jako fiter, gdyż przetwarza kompletny projekt reprezentowany listą połączonych komponentów w formacie EDIF. Symulacja czasowa może być wykonana po zakończeniu syntezy i służy do sprawdzenia układu z uwzględnieniem występujących w nim czasów propagacji.



Rys. 7.20 Przykładowe wyniki symulacji czasowej multipleksera grupowego

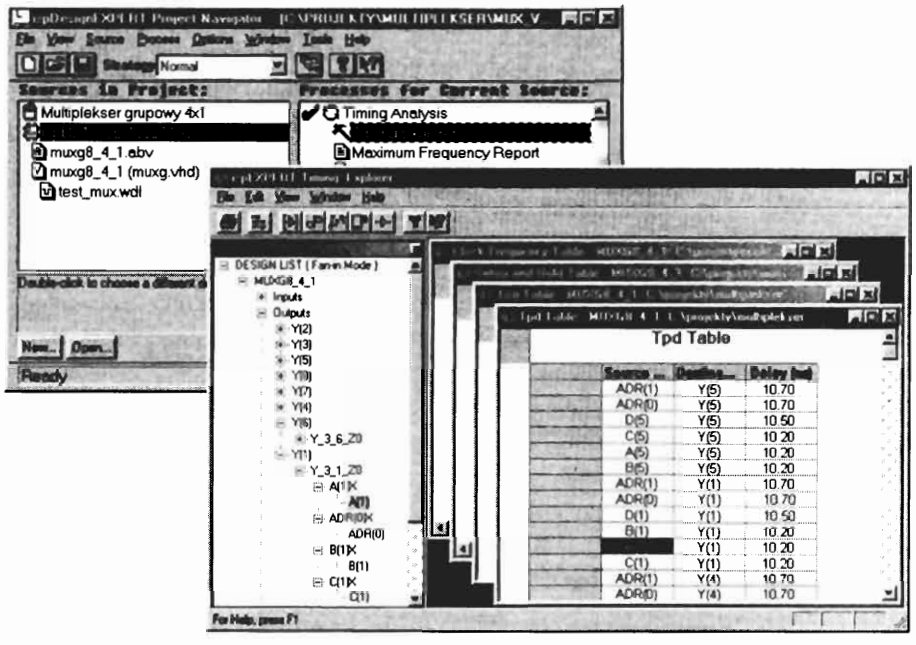
W celu wykonania symulacji należy przygotować plik z wektorami testowymi, czyli opisem wirtualnych sygnałów wejściowych. Są dwie drogi postępowania: tworzymy graficzny plik *.wdl* za pomocą edytora przebiegów lub plik tekstowy *.abv* zawierający opis wektorów testowych w języku ABEL [10]. Wektory testowe *.wdl* mogą być przygotowane dla każdego bloku występującego w projekcie, natomiast wektory *.abv* tylko dla całego projektu. W pliku *.abv* oprócz opisu sygnałów wymuszeń można zdefiniować również oczekiwane wartości odpowiedzi układu.

Symulator uruchamia się w oknie nawigatora po zaznaczeniu pliku z wektorami testowymi, *.abv* lub *.wdl*, oraz dwukrotnym kliknięciu opcji *Functional Simulation* lub *Timing Simulation*. Start symulacji następuje po naciśnięciu przycisku *RUN* w pasku narzędziowym lub wybraniu opcji *Simulate → Run*. Wynik z przebiegami czasowymi pojawia się automatycznie po zakończeniu symulacji (o ile jest zaznaczona

opcja *View → Show Waveforms* w menu symulatora). Wszystkie przypadki wystąpienia niezgodności odpowiedzi układu wobec wartości oczekiwanych są sygnalizowane w oknie komend symulatora.

Symulator umożliwia prześledzenie działania układu z wykorzystaniem wbudowanego debugera do zakładania pułapek oraz ustawiania i obserwacji wymaganych wartości sygnałów. W języku ABEL można definiować wiele sekcji wektorów testowych. Symulator umożliwia wybór kolejnych sekcji następujących po sobie. Niestety nie można wybrać sekcji poprzedzającej. W takim przypadku należy powtórzyć symulację od początku. Przykładowy ekran z wynikami symulacji czasowej przedstawiono na rys. 7.20.

Po odwzorowaniu projektu w wybranym układzie scalonym można przeprowadzić jego analizę czasową. W oknie nawigatora należy zaznaczyć i dwukrotnie kliknąć opcję *Timing Analysis*.



Rys. 7.21 Przykładowe okno z wynikami pracy analizatora czasowego

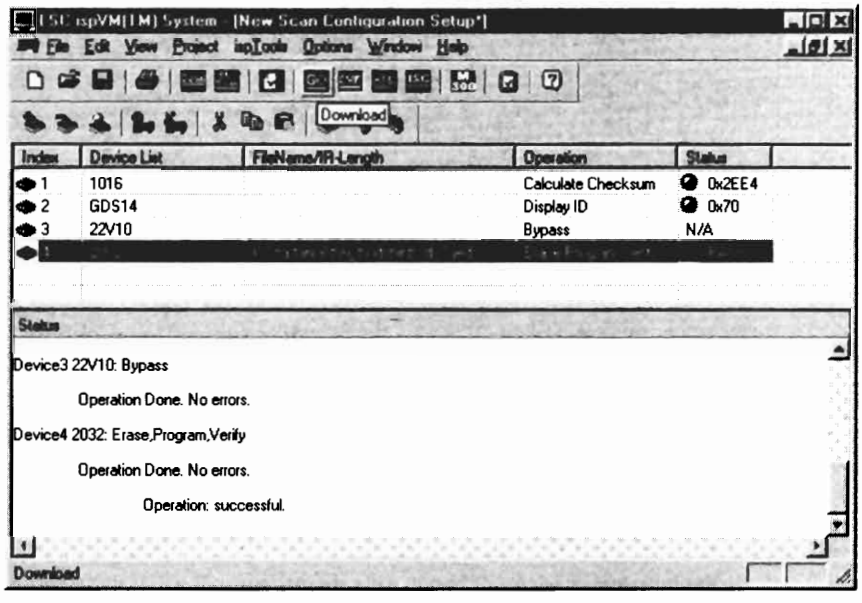
Analiza czasowa ma na celu wyznaczenie właściwości czasowych układu odwzorowanego w strukturze programowalnej. Można określić na przykład takie parametry jak: maksymalna częstotliwość zegara, wartości czasów propagacji, czasy ustalenia i podtrzymania w przerzutnikach. Wyniki analizy czasowej można przeglądać w postaci raportów generowanych osobno dla określonych parametrów (*Maximum Frequency Report, Setup/Hold ReportTpd Report, Tco Report*)

Frequency Report, Setup/Hold ReportTpd Report, Tco Report) lub za pośrednictwem *Timing Explorer* (rys. 7.21).

7.4.8 Programowanie układu

Programowanie układu realizuje się za pomocą programu ładującego i kabla łączącego komputer z układami ISP. Program ładujący *IspVM System* uruchamia się z menu *START* komputera. W programie tym dostępne są opcje umożliwiające automatyczne skanowanie łańcucha ISP w celu identyfikacji układów oraz opcje do obsługi programowania, weryfikacji, odczytu i kasowania wskazanych układów.

Przed programowaniem należy połączyć płytkę LAT10 z komputerem, zgodnie z wcześniejszym opisem, i włączyć zasilanie płytki. Uruchamiamy program i naciskamy przycisk z napisem *SCAN*. Jeśli połączenie programujące jest poprawne, to na ekranie powinno pojawić się okno podobne do przedstawionego na rysunku 7.22. Widzimy, że w wyniku tej operacji pojawił się wykaz układów programowalnych umieszczonych na płytce LAT10. Nastąpiło automatyczne rozpoznanie elementów zgodnie z porządkiem ich występowania w łańcuchu programującym. Dla każdego z tych elementów możemy wskazać plik programujący oraz określić typ operacji, którą będziemy przeprowadzać (np. programowanie, weryfikacja, odczyt, pomiń).



Rys. 7.22 Okno programu ładującego

Podczas jednej „sesji programowania" można realizować różne operacje na różnych układach. Na przykład, można programować układ 1 wykorzystując plik *A1.JED*, weryfikować konfigurację układu 2 z zawartością plikiu *A2.JED*, sprawdzać sumę kontrolną układu 3 i pominąć układ 4. Takie konfiguracje można oczywiście zapamiętywać na dysku i wykorzystywać przy okazji kolejnych uruchomień programu. Sesję programowania rozpoczynamy naciśnięciem przycisku *GO* (*Download*) umieszczonego w polu narzędziowym okna programu ładującego. Po zakończeniu operacji przy każdym układzie wyświetlany jest status informujący o jej wyniku. Dodatkowo przebieg „programowania" można śledzić w oknie komunikatów, gdzie najbardziej oczekiwanym komunikatem jest *SUCCESSFUL*.

7.4.9 Dodatkowe programy użytkowe

System *IspDesignExpert* nie zawiera narzędzi do projektowania z użyciem niektórych mniej popularnych typów układów produkowanych przez firmę Lattice. Do takich należą układy z rodziny przełączników programowalnych IspGDS (*Generic Digital Switch*), których działanie opiera się na wykorzystaniu programowalnej matrycy przełączającej (krosownicy). Za pomocą takiej krosownicy można zmieniać konfigurację połączeń bez konieczności używania mechanicznych przełączników. Matryca może być zaprogramowana tak, by na jej wyjściach pojawiały się sygnały cyfrowe w postaci prostej bądź zanegowanej lub też poziomy logiczne 'L' lub 'H'.

Do projektowania konfiguracji układów przełączników programowalnych IspGDS należy utworzyć plik tekstowy w kodzie asemblera *GASM*. Asembler przetwarza takie pliki źródłowe opisujące konfigurację połączeń do formatu JEDEC. Poniższy przykładowy plik *TEST.GDS*, ilustruje możliwości specyfikacji układu `IspGDS14`.

```
" To jest komentarz
title = 'IRQ DIP SWITCH'

" typ ukladu ispgds (IspGDS14)
device = IspGDS14

" sygnal z wejscia 11 jest doprowadzany do wyjsc 1 i 3
" UWAGA! Zapis pin 1,2 = pin 11 jest niedozwolony
pin 1 = pin 11
pin 3 = pin 11

" sygnal z wejscia 12 jest doprowadzany
```

```
" do wyjsc 8,9 i 10 w postaci zanegowanej
!pin 8 = pin 12
pin 9 != pin 12
pin 10 = !pin 12

" na wyprowadzeniu 19 jest zawsze poziom wysoki (high)
pin 19 = h

" na wyprowadzeniu 18 jest zawsze poziom niski (low)
pin 18 = l

" przyklady deklarowania etykiet
a_label pin 13
b_label pin 2
c_label pin 20

" przyklady stosowania etykiet
a_label = pin 16
c_label = b_label
pin 6   = b_label
```

W celu zaprogramowania układu GDS należy wykonać następujące czynności:

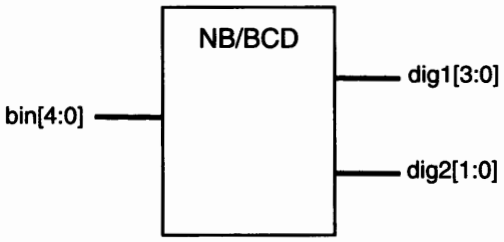
- utworzyć plik tekstowy w kodzie asemblera, np. *TEST.GDS*,
- przetłumaczyć plik podając komendę <GASM TEST>,
- otrzymany plik wynikowy *TEST.JED* wykorzystać do zaprogramowania układu z użyciem programu *IspVM System*.

7.5 PRZYKŁADY PROJEKTÓW

7.5.1 Konwerter kodu BIN/BCD

Zadaniem jest wykonanie projektu konwertera naturalnego kodu dwójkowego na kod BCD z użyciem płytki LAT10 (rys. 7.23). Należy ograniczyć wielkość magistrali wejściowej do pięciu bitów oraz zapewnić sztywne przyporządkowanie końcówek w wybranym układzie programowalnym, aby umożliwić ustawianie wartości sygnałów wejściowych za pomocą przycisków na płytce.

Dla tego typu zadania często stosowanym opisem układu jest tablica stanów. Niestety w standardzie języka VHDL nie przewidziano takiej specyfikacji. Nie oznacza to jednak braku możliwości stosowania takiego opisu z wykorzystaniem typowych konstrukcji językowych. Prosty sposób z wykorzystaniem instrukcji przypisania selektywnego **with-select** oraz operatora „albo" (|) jest omówiony w p. 3.5.



Rys. 7.23 Konwerter kodu NB/BCD

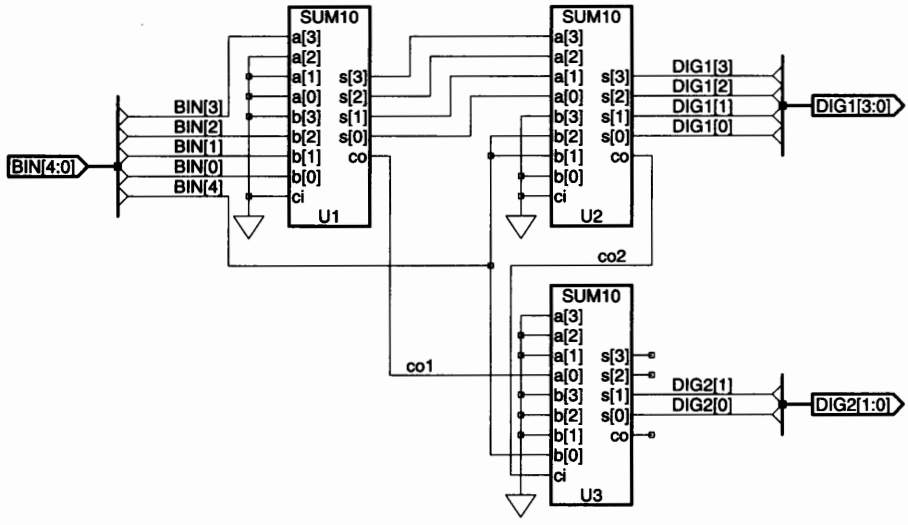
Układ konwertera można opisać różnymi sposobami, uzyskując w rezultacie skrócenie zapisu w języku VHDL. Najkrótszy opis można otrzymać stosując styl behawioralny. W tym przykładzie można zastosować operatory `mod` i `rem` (p. 3.2.1) w celu obliczania części całkowitej oraz reszty z dzielenia przez 10. Jednak korzystanie z tych operatorów nie zawsze jest możliwe, gdyż kompilatory VHDL zazwyczaj ograniczają zakres ich stosowania (np. wymaganie aby prawy operand był potęgą liczby 2).

Można jednak nie używać wymienionych operatorów tworząc opis w stylu behawioralnym (`L7.1a`). W opisie tym stosuje się sprawdzenie, do którego zakresu dziesiętnego należy dana wejściowa i zależnie od wyniku oblicza się wartości sygnału wyjściowego.

```
-- L7.1a
library ieee;
use ieee.std_logic_1164.all;
entity bin1 is
   port( bin: in std_logic_vector(4 downto 0);
         dig1: out std_logic_vector(3 downto 0);
         dig2: out std_logic_vector(1 downto 0));
end bin1;
use ieee.numeric_std.all;
architecture behavioral of bin1 is
begin
  process(bin)
  variable bin_var: unsigned(4 downto 0);
  begin
      bin_var := unsigned(bin);
      if bin_var < 10 then
         dig2 <= "00";
      elsif bin_var < 20 then
         bin_var := bin_var - 10;
         dig2 <= "01";
      elsif bin_var < 30 then
         bin_var := bin_var - 20;
         dig2 <= "10";
```

```
        else
           bin_var := bin_var - 30;
           dig2 <= "11";
        end if;
        dig1 <= std_logic_vector(bin_var(3 downto 0));
    end process;
end;
```

Stosując różne sposoby opisu układu poszukuje się możliwie najefektywniejszego pod względem zajmowanej powierzchni lub szybkości działania [15]. W wielu przypadkach przydatne okazuje się wtedy przeprowadzenie dekompozycji zadania na zadania prostsze. W projektowanym konwerterze kodu zastosowano dekompozycję z użyciem sumatorów dziesiętnych [10], których model umieszczono w pakiecie `wat`. Powstał w ten sposób opis strukturalny (`L7.1b`), odpowiadający schematowi pokazanemu na rysunku 7.24.



Rys. 7.24 Schemat konwertera NB/BCD zbudowanego z sumatorów dziesiętnych

```
-- L7.1b
use work.wat.all;
architecture structural of bin1 is
   component sum10
      port(a:  in std_logic_vector(3 downto 0);
           b:  in std_logic_vector(3 downto 0);
           ci: in std_logic;
           s:  out std_logic_vector(3 downto 0);
           co: out std_logic);
   end component;
```

```
    -- sygnaly wyjsciowe sumatorow U1, U2 i U3
    signal s1,s3: std_logic_vector (3 downto 0);
    signal co1,co2: std_logic;
begin
  U1: sum10 port map (bin(3) & "000",
       '0' & bin(2 downto 0), '0', s1, co1);
  U2: sum10 port map (s1, '0'&bin(4)&bin(4)&'0',
       '0', dig1, co2);
  U3: sum10 port map ("000"&bin(4), "000"&co1,
       co2, s3, open);
  dig2 <= s3(1 downto 0);
end structural;
```

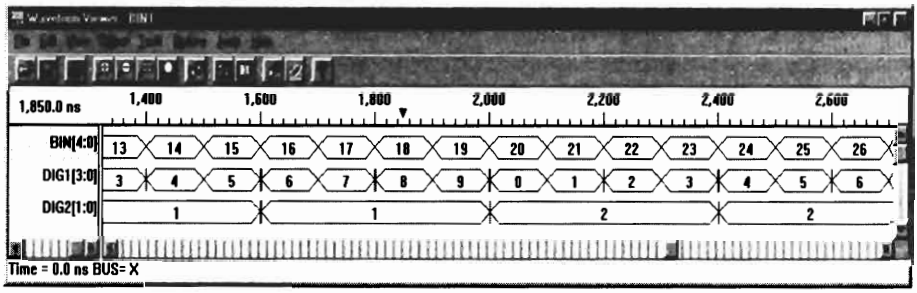
Pomimo różnego sposobu opisu obu architektur układu konwertera nie uzyskano spodziewanych różnic w dokonanych implementacjach. Realizacja konwertera w strukturze ispLSI1016 wymaga w obu przypadkach użycia sześciu makrokomórek zawartych w dwóch blokach GLB. Jednak uzyskany identyczny wynik nie oznacza, że zaproponowany tok postępowania okaże się równie nieskuteczny przy innych znacznie bardziej złożonych zadaniach.

W celu sprawdzenia projektu zawsze najpierw należy wykonać symulację funkcjonalną, która zazwyczaj pozwala na skrócenie czasu projektowania, wskazując miejsca błędnego działania modelu. Do symulacji z użyciem *Lattice Logic Simulator* z systemu *IspDesignEXPERT* można utworzyć wektory testowe w języku ABEL:

```
module bin1
"  wejscia
   bin_4_,bin_3_,bin_2_,bin_1_,bin_0_ pin;
"  wyjscia
   dig1_3_,dig1_2_,dig1_1_,dig1_0_pin;
   dig2_1_,dig2_0_ pin;
   bin  = [bin_4_,bin_3_,bin_2_,bin_1_,bin_0_];
   dig1 = [dig1_3_,dig1_2_,dig1_1_,dig1_0_];
   dig2 = [dig2_1_,dig2_0_];
test_vectors ([bin] -> [dig2, dig1])
   @const licznik=0;
   @repeat 32
         {[licznik] -> [licznik/10,licznik%10];
         @const licznik=licznik+1;}
end
```

W celu sprawdzenia poprawności działania modelu użyto pętli **repeat** do generacji wszystkich stanów wejść. Stany wyjść wyznaczone przez symulator są porównywane z wynikami prawidłowymi, które w tym przypadku są reprezentowane w postaci działań `licznik/10` i `licz-`

`nik%10` (część całkowita i reszta z dzielenia przez 10). Symulator umożliwia wyświetlanie wyników w postaci graficznej (rys. 7.25) jak również w postaci tekstowej.



Rys. 7.25 Przykładowy wynik symulacji konwertera w postaci przebiegów czasowych

Obserwacja wyników symulacji w postaci tekstowej jest szczególnie użyteczna, gdy w trakcie symulacji wykrywane są błędy. Wówczas w miejscach wystąpień błędów wyświetlane są wiersze zawierające oczekiwane oraz nieprawidłowe wartości odpowiedzi układu. Poniżej przedstawiono początkowy fragment pliku zawierającego błędy.

```
// Trace file generated by Simulator
// Simulation result
                          dddddd
                    bbbbb iiiiii
                    iiiii ggggg
                    nnnnn 221111
                    _____ ______
                    43210 103210
                    _____ ______
                    BBBBB BBBBBB
          0.000 ns: 00000->xxxxxx
          0.000 ns: 00000->xxxxx0
          0.000 ns: 00000->0xxxx0
          0.000 ns: 00000->0xxx00
          0.000 ns: 00000->0x0000
          0.000 ns: 00000->000000
at 0.000 ns, DIG1_0_  expected: 1, actual: 0
        100.000 ns: 00001->000000
        100.000 ns: 00001->000001
at 100.000 ns, DIG1_1_  expected: 1, actual: 0
at 100.000 ns, DIG1_0_  expected: 0, actual: 1
        200.000 ns: 00010->000001
        200.000 ns: 00010->000000
        200.000 ns: 00010->000010
at 200.000 ns, DIG1_0_  expected: 1, actual: 0
        300.000 ns: 00011->000010
```

Do symulacji z użyciem symulatora VHDL spoza systemu *ispDesign Expert* niezbędny jest program testowy o przykładowej postaci:

```
--L7.1_TB
library  ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_textio.all;
use std.textio.all;
entity testbench is
   generic (czekaj: time:=15 ns);
end testbench;
architecture testbench_arch of testbench is
   file results: text is out "results.txt";
   component bin1
      port (bin : in  std_logic_vector (4 downto 0);
          dig1 : out  std_logic_vector (3 downto 0);
          dig2 : out  std_logic_vector (1 downto 0));
   end component;
   signal bin : std_logic_vector (4 downto 0);
   signal dig1 : std_logic_vector (3 downto 0);
   signal dig2 : std_logic_vector (1 downto 0);
begin
   uut : bin1  port map
         (bin => bin, dig1 => dig1, dig2 => dig2);
process
   variable tx_out : line;
   variable tx_error : integer := 0;
   procedure check_dig1
         (next_dig1 : std_logic_vector(3 downto 0)) is
      variable tx_loc : line;
      variable tx_str : string(1 to 512);
   begin
      if dig1 /= next_dig1 then
          tx_error := tx_error + 1;
          write(tx_loc, now);
          write(tx_loc, string'(": [Blad "));
          write(tx_loc, tx_error);
          write(tx_loc, string'("] dig1="));
          write(tx_loc, dig1);
          write(tx_loc, string'(" zamiast "));
          write(tx_loc, next_dig1);
          write(tx_loc, string'(" "));
          tx_str(tx_loc.all'range) := tx_loc.all;
          writeline(results, tx_loc);
          deallocate(tx_loc);
          assert false report tx_str severity error;
      end if;
   end;
   procedure check_dig2
```

```
        (next_dig2 : std_logic_vector(1 downto 0)) is
      variable tx_str : string(1 to 512);
      variable tx_loc : line;
    begin
      if dig2 /= next_dig2 then
          tx_error := tx_error + 1;
          write(tx_loc, now);
          write(tx_loc, string'(": [Blad "));
          write(tx_loc, tx_error);
          write(tx_loc, string'("] dig2="));
          write(tx_loc, dig2);
          write(tx_loc, string'(" zamiast "));
          write(tx_loc, next_dig2);
          write(tx_loc, string'(" "));
          tx_str(tx_loc.all'range) := tx_loc.all;
          writeline(results, tx_loc);
          deallocate(tx_loc);
          assert false report tx_str severity error;
      end if;
    end;
  begin
    for k in 0 to 31  loop
      bin <= std_logic_vector(to_unsigned(k,5));
      wait for czekaj;
      check_dig1
        (std_logic_vector(to_unsigned(k rem 10,4)));
      check_dig2
        (std_logic_vector(to_unsigned(k/10,2)));
    end loop;
    if (tx_error = 0) then write(tx_out, string'
     ("Symulacja zakonczona. Bledow nie wykryto."));
      writeline(results, tx_out);
      assert false report
        "Symulacja zakonczona. Bledow nie wykryto."
        severity warning;
    else
      write(tx_out, string'("Wykryto "));
      write(tx_out, tx_error);
      if tx_error = 1 then
        write(tx_out, string'(" blad"));
      elsif (tx_error rem 10) < 5 then
        write(tx_out, string'(" bledy"));
      else write(tx_out, string'(" bledow"));
      end if;
      write(tx_out, string'(" podczas symulacji."));
      writeline(results, tx_out);
      assert false
      report "Wykryto bledy podczas symulacji."
          severity failure;
    end if;
    wait;
```

```
    end process;
    end testbench_arch;
```

W trakcie symulacji są sprawdzane odpowiedzi badanego układu na wszystkie możliwe stany pięciu wejść. Sprawdzenie odbywa się z użyciem procedur `check_dig1` oraz `check_dig2`, wyświetlających błędy na ekranie oraz zapisujących je do pliku *results.txt*. Na koniec jest wyświetlany komunikat `"Wykryto bledy podczas symulacji."` lub `"Symulacja zakonczona. Bledow nie wykryto."`.

Podczas symulacji został utworzony plik *results.txt*:

```
15 ns: [Blad 1] dig1=0UU0 zamiast 0000
15 ns: [Blad 2] dig2=0U zamiast 00
45 ns: [Blad 3] dig1=0000 zamiast 0010
75 ns: [Blad 4] dig1=0110 zamiast 0100
135 ns: [Blad 5] dig1=1010 zamiast 1000
165 ns: [Blad 6] dig2=00 zamiast 01
195 ns: [Blad 7] dig1=0000 zamiast 0010
225 ns: [Blad 8] dig1=0110 zamiast 0100
435 ns: [Blad 9] dig1=1100 zamiast 1000
Wykryto 9 bledow podczas symulacji.
```

Zostały wykryte błędy, ponieważ w pętli **for-loop** zastosowano instrukcję czekania **wait for** `15 ns;`. W tym przypadku przyczyną wykrycia błędów był błąd testu polegający na tym, że nie uwzględniono w wystarczającym stopniu czasów propagacji badanego układu. Po zwiększeniu czasu oczekiwania do 20 ns błędów nie wykryto i na ekranie jest wyświetlony komunikat:

```
# ** Warning: Symulacja zakonczona. Bledow nie wykryto.
#    Time: 640 ns  Iteration: 0  Instance: /testbench.
```

a plik *results.txt* zawiera tylko jedną linię tekstu:

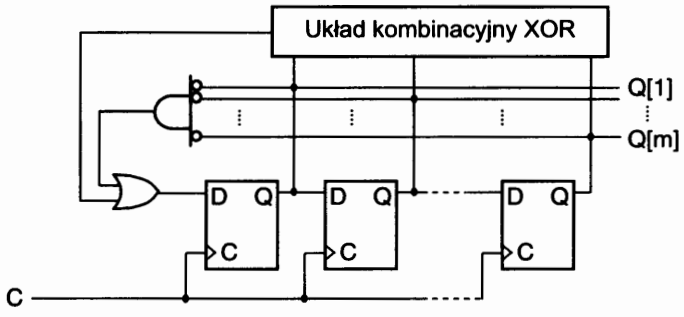
```
Symulacja zakonczona. Bledow nie wykryto.
```

7.5.2 Generator ciągu pseudolosowego

Generatory ciągów liczb pseudolosowych wykorzystuje się do testowania systemów transmisyjnych oraz urządzeń akustycznych. Wytwarza się je za pomocą rejestru przesuwającego z odpowiednim sprzężeniem zwrotnym (rys. 7.26). Po każdym impulsie zegarowym zawartość rejestru przesuwa się o jedną pozycję w prawo i stan na wyjściu układu

XOR zmienia się zależnie od stanów wybranych przerzutników wchodzących w skład rejestru.

Bramki `OR` i `Negative AND` służą do ustawienia startowej wartości rejestru w przypadku, gdy wartość ta jest zerowa. Można zrezygnować z ich stosowania, jednak wtedy konieczne jest wystąpienie przynajmniej jednego stanu `'1'` w sekwencji startowej. W przypadku, gdy warunek ten nie będzie spełniony, generacja kodu pseudolosowego nie wystąpi, gdyż `0 xor 0 = 0`.



Rys. 7.26 Generator ciągu pseudolosowego

Generowany ciąg pseudolosowy zależy od długości rejestru *m*. Największą długość cyklu losowego wynoszącą $2^m - 1$ uzyskuje się stosując odpowiedni układ sprzężenia zwrotnego, którego strukturę można wyznaczyć korzystając z tablic przedstawionych w [12].

Zadanie polega na zaprojektowaniu generatora ciągu pseudolosowego oraz jego praktycznym wykonaniu z użyciem płytki testowej LAT10. Podczas projektowania należy uwzględnić ograniczenia związane z niewielkim wyborem układów, sztywnym przyporządkowaniem końcówek i małą liczbą diod świecących. W związku z tym można ograniczyć długość rejestru przesuwającego do ośmiu bitów, aby wyjście każdego z przerzutników można było podłączyć do diod świecących LED.

Przedstawiony dalej model generatora ciągu pseudolosowego w języku VHDL zawiera układ kombinacyjny XOR realizujący funkcję `q(8) xor q(4) xor q(3) xor q(2)`. Sygnał ten jest doprowadzany na wejście pierwszego przerzutnika rejestru przesuwającego, natomiast sygnały `q` pochodzą z odpowiednio wybranych wyjść przerzutników.

```
-- L7.2
library ieee;
use ieee.std_logic_1164.all;
```

```
entity gener is
   port(clk: in std_logic;
        q: out std_logic_vector(1 to 8));
   attribute loc: string;
   attribute loc of clk: signal is "P11";
   attribute loc of q: signal is
      "P10,P9,P7,P3,P41,P37,P32,P31";
end gener;
architecture arch1 of gener is
signal sig_q: std_logic_vector(1 to 8):= "00000000";
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if sig_q = "00000000" then
         sig_q(1) <= '1';
      else
         -- uklad kombinacyjny XOR
         sig_q(1) <= sig_q(8) xor sig_q(4)
                     xor sig_q(3) xor sig_q(2);
      end if;
      -- przesuniecie bitow w prawo
      sig_q(2 to 8) <= sig_q(1 to 7);
    end if;
end process;
  q <= sig_q;
end arch1;
```

W deklaracji jednostki zawarto przyporządkowania poszczególnym sygnałom odpowiednich numerów wyprowadzeń układu za pomocą atrybutów `loc`. Występowanie takiego przyporządkowania pozbawia pierwotny opis cechy wirtualności tego projektu, gdyż dowiązuje go do konkretnego układu scalonego. W praktyce częściej takie i inne atrybuty definiuje się w osobnym pliku, który można przygotować przy pomocy narzędzia *Constraint Manager* lub też dowolnego edytora tekstowego. Fragment takiego pliku (*gener.prp*) przedstawiono poniżej.

```
property gener pin clk lock 11 endproperty
property gener pin q(7) lock 32 endproperty
property gener pin q(4) lock 3 endproperty
```

Poprawność działania układu można sprawdzić z użyciem testu pokazanego dalej. W trakcie symulacji są wyświetlane komunikaty oraz są tworzone raporty w postaci plików tekstowych *results1* i *results2*. Dla celów symulacji założono, że ośmiobitowe liczby pseudolosowe są tworzone na podstawie ciągu bitów `q(i)` uzyskiwanych z badanego układu za pomocą wzoru

$$L = \sum_{i=1}^{8} 2^{-i} q(i)$$

Dzięki takiej reprezentacji uzyskiwane są liczby ułamkowe o wartościach w przedziale (0,1).

```
--L7.2_TB
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_textio.all;
use std.textio.all;
entity testbench is
   generic (n: integer:=255);
end testbench;
architecture testbench_arch of testbench is
   type t_hist is
         array(integer range 1 to 10) of integer;
   type c_real is
         array(integer range 1 to n) of real;
   file results1: text is out "results1.txt ";
   file results2: text is out "results2.txt ";
   component gener
         port (q : out  std_logic_vector (1 to 8);
               clk : in  std_logic);
   end component;
   signal q : std_logic_vector (1 to 8);
   signal clk : std_logic;
begin
   uut : gener port map (q => q, clk => clk
   process
         variable tab_hist: t_hist;
         variable tab_liczb: c_real;
         variable suma, liczbaulamkowa: real;
         variable w_srednia, sumakw, wariancja: real;
         variable tx_out : line;
         variable tx_str : string(1 to 512);
   begin
   for i in 1 to 10 loop
      tab_hist(i):=0;
   end loop;
   for i in 1 to 1000 loop
      clk <= '0';      wait for 10 ns;
      clk <= '1';      wait for 10 ns;
      liczbaulamkowa := 0.0;
      for j in 1 to 8 loop
        if q(j)='1' then
          liczbaulamkowa := liczbaulamkowa + 2.0**(-j);
        end if;
      end loop;
```

```
    if i <= n then
        tab_liczb(i):= liczbaulamkowa;
        for j in 1 to 10 loop
          if (liczbaulamkowa <= real(j) * 0.1 and
            liczbaulamkowa > real(j-1) * 0.1 ) then
            tab_hist(j):= tab_hist(j)+1;
          end if;
        end loop;
    end if;
    write(tx_out,i);
    write(tx_out, string'(". "));
    write(tx_out,liczbaulamkowa);
    writeline(results1, tx_out);
  end loop;
write(TX_OUT,string'("Histogram"));
writeline(results2, tx_out);
    for i in 1 to 10 loop
      write(tx_out, string'(" (0."));
      write(tx_out, i-1);
      if i < 10 then
         write(tx_out, string'(" - 0."));
         write(tx_out, i);
         write(tx_out, string'("] - "));
      else
         write(tx_out, string'(" - 1.0) - "));
      end if;
      write(tx_out, tab_hist(i));
      writeline(results2, tx_out);
      suma:=suma+tab_liczb(i);
    end loop;
write(tx_out,string'("Dlugosc ciagu: "));
write(tx_out, n);
tx_str(tx_out.all'range):=tx_out.all;
writeline(results2, tx_out);
assert false report tx_str severity warning;
write(tx_out,string'("Wartosc srednia: "));
suma:=0.0;
for i in 1 to n loop
     suma:=suma + tab_liczb(i);
end loop;
w_srednia:=suma/real(n);
write(tx_out, w_srednia);
tx_str(tx_out.all'range):=tx_out.all;
writeline(results2, tx_out);
assert false report tx_str   severity warning;
sumakw:=0.0;
for i in 1 to n loop
     sumakw:=sumakw + (tab_liczb(i)-w_srednia)**2;
end loop;
write(tx_out,string'("Wariancja: "));
```

```
    wariancja:=sumakw/real(n);
    write(tx_out, Wariancja);
    write(tx_out,string'("                    "));
    tx_str(tx_out.all'range):=tx_out.all;
    writeline(results2, tx_out);
     assert false report tx_str    severity warning;
    write(tx_out,string'("Symulacja zakonczona."));
    writeline(results2, tx_out);
    assert false report "Symulacja zakonczona
           (results1.txt, results2.txt)."
            severity warning;
   wait;
end process;
end testbench_arch;
```

Generowane liczby pseudolosowe są zapisywane w pliku *results1.txt*:

```
1. 4.960938e-001
2. 2.460938e-001
3. 6.210938e-001
.......
254. 9.921875e-001
255. 9.960938e-001
.......
```

W pliku *results2.txt* są zapisywane wyniki obliczeń statystycznych dla ciągu `n=255` kolejnych liczb pseudolosowych uzyskanych z generatora.

```
Histogram
 (0.0 - 0.1] - 25
 (0.1 - 0.2] - 26
 (0.2 - 0.3] - 25
 (0.3 - 0.4] - 26
 (0.4 - 0.5] - 26
 (0.5 - 0.6] - 25
 (0.6 - 0.7] - 26
 (0.7 - 0.8] - 25
 (0.8 - 0.9] - 26
 (0.9 - 1.0) - 25
Dlugosc ciagu: 255
Wartosc srednia: 5.000000e-001
Wariancja: 8.268229e-002
Symulacja zakonczona.
```

Opisany test można rozbudować i przystosować do sprawdzania różnych generatorów liczb. Jeżeli wartość średnia liczb w ciągu mało się różni od wartości 0.5, a wariancja od wartości 1/12 = 0.08333..., i w wyznaczonych podprzedziałach przedziału (0,1) nie ma istotnych różnic liczności, to można uznać, że wygenerowany ciąg pochodzi z gene-

ratora losowego o rozkładzie wystarczająco równomiernym. Jak widać, przedstawiony w zadaniu generator może spełniać warunki tego testu.

Opracowano wiele testów statystycznych do weryfikacji hipotezy, czy rozkład liczb generowanego ciągu jest zgodny z rozkładem równomiernym. W Internecie pod adresem http://stat.fsu.edu/~geo/diehard.html jest udostępniona strona poświęcona generatorom liczb losowych, zawierająca m.in. bogatą bibliotekę testów statystycznych.

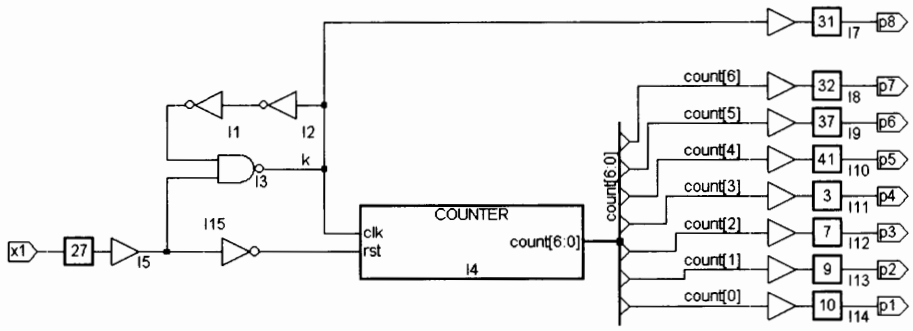
7.5.3 Generator sygnału prostokątnego

Należy zaprojektować i zrealizować układ generatora w układzie programowalnym, umieszczonym na płytce testowej LAT10. Częstotliwość sygnału generatora powinna wynosić 20+30 MHz. Na wyjściu generatora należy dołączyć dzielnik częstotliwości do ilustracji pracy generatora za pomocą diod świecących, dołączonych do wyjść układu scalonego. Podczas projektowania należy:

- opisać blok generatora w postaci schematu, natomiast układ dzielnika częstotliwości za pomocą języka VHDL,
- umożliwić włączanie i wyłączanie generatora sygnałem sterującym,
- dobrać współczynnik podziału częstotliwości w dzielniku tak, aby praca generatora mogła być uwidoczniona za pomocą diod LED,
- uwzględnić ograniczenia płytki uruchomieniowej, sztywne przyporządkowanie końcówek i ograniczoną liczbę diod świecących.

Poniżej przedstawiono fragmenty rozwiązania zadania: schemat układu (rys. 7.27), opis projektowania generatora oraz opis układu dzielnika w języku VHDL.

Układ generatora jest zbudowany z połączonych ze sobą bramek w taki sposób, że w pętli istnieje dodatnie sprzężenie zwrotne. Może być usunięty przez kompilator podczas syntezy. Aby tego uniknąć trzeba wprowadzić do schematu atrybuty, które skojarzone z odpowiednimi symbolami schematowymi bądź połączeniami, zapobiegną redukcji układu nawet wtedy, gdy z punktu widzenia oprogramowania opis ten jest nadmiarowy. W tym celu trzeba nadać połączeniom w pętli atrybuty `PRESERVE=Y`.



Rys. 7.27 Przykładowy schemat układu

Częstotliwość generatora można wyznaczyć na podstawie danych katalogowych korzystając z zależności

$$f = \frac{1}{2\,t_p}$$

gdzie t_p – czas opóźnienia pętli sprzężenia zwrotnego.

Przykładowo dla układu ispLSI1016-60 można wyznaczyć t_p dla trzech różnych konfiguracji pętli generatora: bez inwerterów, z dwoma i z czterema inwerterami. W tym celu należy odczytać z katalogu dane o czasach opóźnień bloków logicznych oraz połączeń. Dla uproszczenia przyjmujemy, że bramka NAND zostanie zrealizowana z użyciem programowalnej matrycy AND oraz bramki XOR zawartych w tym samym bloku logicznym GLB. Wtedy model opóźnienia bramki składa się z czasów t_{lptxor} = 9,3 ns oraz t_{gbp} = 1,3 ns. Jeśli ten sam model opóźnień zastosujemy również do opisu łańcucha dwóch inwerterów, wtedy do wyznaczenia częstotliwości generatorów brakuje już tylko opóźnienia połączeń t_{grpl} = 2 ns. W tablicy 7.6 zestawiono wyniki obliczeń oraz wyniki uzyskane na drodze symulacji.

Tablica 7.6 Zestawienie obliczeń częstotliwości generatora

Liczba inwerterów	t_p	$f_{obliczone}$	$f_{symulatora}$
0	9,3 ns+1,3 ns+2 ns	~ 40 MHz	~ 50 MHz
2	2(9,3 ns+1,3 ns+2 ns)	~ 20 MHz	~ 23 MHz
4	3(9,3 ns+1,3 ns+2 ns)	~ 13 MHz	~ 15 MHz

Jak widać, częstotliwości generatora otrzymane w wyniku obliczeń i symulacji różnią się. Różnica może wynikać z tego, że do obliczeń przyjęto wartości opóźnień dla najprostszych konfiguracji bloków logicznych, gdy tymczasem symulator poddał analizie modele czasowe układu właściwe dla bardziej złożonych konfiguracji wyznaczonych

przez fiter. Należy również uwzględnić, że do obliczeń przyjęto maksymalne wartości opóźnień (w katalogu są zamieszczone tylko wartości maksymalne t_{lptxor}, t_{gbp}, t_{grpl}).

Z tablicy 7.5 wynika, że warunki postawionego zadania spełnia układ generatora z dwoma inwerterami generujący sygnał prostokątny o częstotliwości równej 23 MHz.

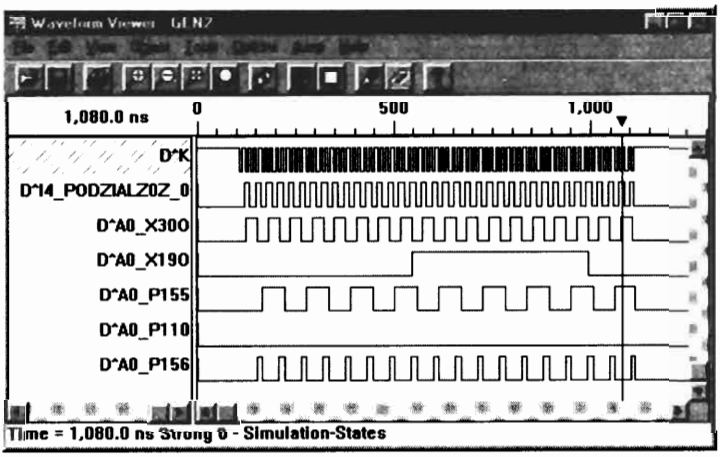
```
-- L7.3
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity counter is
   port ( clk, rst: in std_logic;
          count  : out std_logic_vector (6 downto 0));
end counter;
architecture behave of counter is
   signal count_i : unsigned(6 downto 0);
   signal podzial : integer range 0 to 999999;
begin
   process(clk, rst)
   begin
      if rst='1' then
               count_i <= "0000000";
               podzial <=0;
      elsif rising_edge(clk) then
           podzial <= podzial + 1;
           if podzial = 999999 then
                count_i <= count_i + 1;
                podzial <=0;
           end if;
      end if;
   end process;
   count <= std_logic_vector(count_i);
end behave;
```

Moduł `counter` pełni dwie funkcje: dzielnika częstotliwości przez 1000000 oraz 7-bitowego licznika impulsów doprowadzanych z wyjścia dzielnika. Po dołączeniu wyjść układu licznika do diod LED można oszacować wartość częstotliwości sygnału generatora, która powinna na wejściu licznika wynosić około 23 Hz.

Do sprawdzenia działania układu można zastosować plik z wektorami testowymi w języku ABEL. Przykładowy wynik symulacji wykonanej z użyciem tych wektorów pokazano na rys. 7.28.

```
module gen2
    x1 pin;
    p8, p7 pin;
```

```
      p6, p5 pin;
      p4, p3 pin;
      p2, p1 pin;
test_vectors(x1 -> [p8,p7,p6,p5,p4,p3,p2,p1])
  0 -> [.x.,.x.,.x.,.x.,.x.,.x.,.x.,.x.];
wait 100;
 1 -> [.x.,.x.,.x.,.x.,.x.,.x.,.x.,.x.];
wait 1000;;
 0 -> [.x.,.x.,.x.,.x.,.x.,.x.,.x.,.x.];
wait 600;
end
```



Rys. 7.28 Przykładowy wynik symulacji projektu

Do sprawdzenia działania generatora z użyciem symulatora VHDL można zastosować program testowy z opisem wymuszeń (L7.3_TB). W celu odczytania wyników należy uruchomić interfejs graficzny symulatora umożliwiający wyświetlanie przebiegów czasowych.

```
--L7.3_TB
library ieee;
use ieee.std_logic_1164.all;
entity testbench is
end testbench;
architecture behavior of testbench is
   signal   x1 : std_logic;
   signal   p1, p2, p3, p4, p5, p6, p7, p8: std_logic;
   component gen2
     port (x1: in std_logic;
       p1, p2, p3, p4, p5, p6, p7, p8: out std_logic);
   end component;
begin
   uut : gen2 port map ( p1=>p1,p2=>p2,p3=>p3,p4=>p4,
        p5=>p5,p6=>p6,p7=>p7,p8=>p8,x1=>x1 );
```

```
    tb : process
    begin
       x1 <= '0', '1' after 100 ns, '0' after 200 ns;
       wait for 1000 ns; wait;
    end process tb;
end behavior;
```

Częstotliwość sygnału na wyjściu P8 została zmierzona częstościomierzem. Wyniosła 26,8 MHz. Migotanie diod LED dołączonych do pozostałych wyjść układu pozwala zgrubnie ocenić prawidłowość pracy generatora oraz dzielnika bez konieczności stosowania przyrządów pomiarowych.

7.5.4 Zamek szyfrowy

Należy zaprojektować cyfrowy zamek szyfrowy o dużej odporności na próbę złamania hasła. Projekt należy wykonać z wykorzystaniem jednego z układów programowalnych umieszczonych na płytce testowej LAT10. Po sprawdzeniu układu na płytce testowej, należy wybrać docelowy układ programowalny, którego użycie jest najbardziej opłacalne.

Założenia projektowe:

- hasłem jest ciąg cyfr dziesiętnych,
- długość hasła jest ustalana w zakresie od 5 do 8 poprzez wybór jednego z czterech wyjść układu,
- wprowadzanie hasła odbywa się za pomocą 10-przyciskowej klawiatury na zewnątrz ochranianego obiektu,

Należy ograniczyć zakres projektu do części cyfrowej i zastosować odrębny układ dekodera klawiatury dostarczający sygnały przyciśniętych klawiszy.

Idea pracy zamka szyfrowego polega na sprawdzaniu hasła, czyli zgodności liczby wprowadzanej z klawiatury z liczbą zawartą w pamięci zamka, oraz odpowiednim sterowaniu urządzeniem wykonawczym (zasuwą).

W przedstawionym projekcie zastosowano układ programowalny ispGAL22V10, w którego strukturze została zapamiętana 8-cyfrowa liczba stanowiąca hasło, oraz układ decyzyjny. Założono, że cyfry z klawiatury wprowadzane są za pośrednictwem odrębnego układu dekodera. Dekoder ten tworzy 4-bitową liczbę dwójkową numeru klawisza oraz impulsowy sygnał gotowości danych, generowany w takt

kolejnych naciśnięć klawiszy. Każdej cyfrze liczby dziesiętnej przyporządkowany jest jeden przerzutnik. Przerzutniki są kolejno ustawiane w przypadku poprawnego wprowadzania sekwencji kodowej. Opis zamka w języku VHDL zamieszczono dalej.

W przypadku wprowadzenia pierwszej cyfry zgodnej z pierwszą cyfrą hasła następuje ustawienie pierwszego przerzutnika. Gdy druga cyfra również będzie poprawna, wówczas nastąpi ustawienie drugiego przerzutnika. Przy wprowadzaniu kolejnych poprawnych cyfr stan '1' propaguje się aż do ostatniego przerzutnika. Jeśli we wprowadzanej sekwencji wystąpi błąd, to odpowiedni przerzutnik nie zostanie ustawiony, a ponadto wystąpi wyzerowanie wszystkich przerzutników. Należy więc zacząć wprowadzanie kodu od początku.

```
-- L7.4
-- Projekt zamka szyfrowego w ukladzie ispGAL22V10
-- Zbigniew Jachna
-- zjachna@wel.wat.waw.pl
-- Warszawa 2002
--
library ieee;
use ieee.std_logic_1164.all;
entity zamek is
port ( bcd : in std_logic_vector(3 downto 0);
     clk,reset : in std_logic;
     wy : out std_logic_vector(3 downto 0));
   attribute loc: string;
   attribute loc of bcd: signal is "P4,P5,P6,P7";
   attribute loc of clk: signal is "P2";
   attribute loc of reset: signal is "P9";
   attribute loc of wy: signal is "P17,P20,P25,P27";
end zamek;
architecture a1 of zamek is
   type tablica is array(1 to 8)
      of std_logic_vector(3 downto 0);
   constant kod: tablica :=
    ("0011", "0001", "1001", "0000", "0000", "0101",
     "0101", "0100");      -- haslo: 31900554
   signal y: std_logic_vector(8 downto 1);
begin
   process (clk,reset)
   variable zgodny: boolean;
   begin
      if reset='1' then
         y<= (others => '0');
      elsif rising_edge(clk) then
         for i in 1 to 8 loop
            zgodny := bcd = kod(i);
            if i>1 then
```

```
                zgodny := zgodny and y(i-1)='1';
            end if;
            if zgodny  then
                y(i) <= '1';
            else
                y(i) <= '0';
            end if;
        end loop;
    end if;
  end process;
  wy <= y(8 downto 5);
end;
```

Zmiana hasła zamka wymaga zmiany stałej kod, czyli przeprogramowania układu. Długość hasła można wybrać łącząc układ wykonawczy z jednym z czterech wyjść wy układu programowalnego.

Prawdopodobieństwo otwarcia zamka przez osoby nie znające hasła można wyznaczyć na podstawie liczby kombinacji $n = N^K$, gdzie N jest liczbą klawiszy i K jest długością sekwencji kodowej. Przykładowo dla K = 6 i N = 10 otrzymujemy n = 1000000.

Opisywany układ jest przystosowany do współpracy z klawiaturą o 16 klawiszach oznaczonych 0..9 i A..F. Każdemu przyciskowi przyporządkowana jest czterobitowa liczba dwójkowa. Przy wykorzystaniu takiej klawiatury liczba kombinacji wynosi 16^8. Oczywiście układ będzie pracował poprawnie również z mniejszymi klawiaturami, na przykład z pięcioma lub dziesięcioma klawiszami (jak w postawionym zadaniu). Wówczas należy ograniczyć zakres stosowanych cyfr, np. od zera do dziewięciu. Warunkiem zadziałania zamka jest wystąpienie odpowiedniej sekwencji kodowej, której wprowadzanie można rozpoczynać w dowolnym czasie. Fakt otwarcia drzwi powinien być rozpoznawany w celu wyzerowania układu zamka niskim poziomem sygnału reset. Dalej przedstawiono zawartość pliku testowego, który można wykorzystać do weryfikacji projektu.

```
--L7.4_TB
library  ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use work.wat.all;
entity testbench is
  generic (n: integer:= 1000);
end testbench;
architecture testbench_arch of testbench is
type tablica is array(1 to 8) of integer range 0 to 9;
```

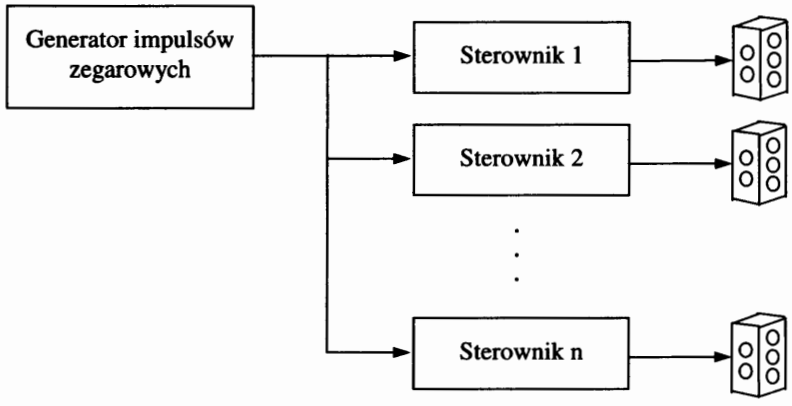
```
file results: text is out "results.txt";
  component zamek
   port (bcd : in  std_logic_vector (3 downto 0);
         clk, reset: in  std_logic;
         wy : out  std_logic_vector (3 downto 0));
  end component;
  signal bcd : std_logic_vector (3 downto 0);
  signal clk, reset: std_logic;
  signal wy : std_logic_vector (3 downto 0);
begin
  uut : zamek port map (  bcd => bcd, clk => clk,
                          reset => reset, wy => wy);
  process
   constant Modul  : INTEGER := 100000000;
   variable Seed,m   : INTEGER := 0;
   variable Seed2    : INTEGER := 45500913;
   variable tx_out : line;
   variable tx_str : string(1 to 512);
   variable haslo : tablica;
begin
   reset <= '1'; clk <= '0';
   wait for 10 ns; reset <= '0';
   for k in 0 to n loop
   if m=8 then
     Random(Seed, Modul);  --generacja l.losowej
     seed2:=seed; m:=0;
   end if;
   m:=m+1;
 bcd <= std_logic_vector(to_unsigned(Seed2 mod 10,4));
   haslo(1 to 7):= haslo(2 to 8)
   haslo(8):=Seed2 mod 10;
   Seed2:= Seed2/10;
   wait for 10 ns; clk <= '1';
   wait for 10 ns; clk <= '0';
   for i in 1 to 8 loop
        write(tx_out,haslo(i));
   end loop;
   if wy(3)='1' then
     write(tx_out,string'(" - znalezione haslo"));
     tx_str(tx_out.all'range):=tx_out.all;
     assert false report tx_str severity warning;
   end if;
   write(tx_out,"        ");
   writeline(results, tx_out);
end loop;
   assert false report
        "Symulacja zakonczona (results.txt)."
         severity warning;
   wait;
   end process;
end testbench_arch;
```

Test polega na sprawdzeniu zamka dla prawidłowej kombinacji cyfr hasła oraz dla 1000 cyfr dostarczonych z generatora pseudolosowego. Wszystkie znalezione sekwencje cyfr, otwierające zamek są wyświetlane na ekranie. Przykładowy komunikat wygląda następująco:

```
Selected Top-Level: testbench (testbench_arch)
: WARNING: 31900554 - znalezione haslo
: Time: 170 ns,  Iteration: 0,  TOP instance.
: WARNING: Symulacja zakonczona (results.txt).
: Time: 20030 ns,  Iteration: 0,  TOP instance.
```

7.5.5 Sterownik świateł ulicznych

Do kontroli ruchu drogowego na skrzyżowaniu dróg powszechnie stosuje się synchronizowane urządzenia sygnalizacyjne (rys. 7.29). Wszystkie sterowniki systemu są sterowane z tego samego generatora synchronizującego, aby ustawiać odpowiednio fazy świateł.



Rys. 7.29 System sygnalizacji świetlnej

Celem zadania jest projekt sterownika pojedynczego panelu sygnalizacyjnego, przystosowanego do pracy synchronicznej.

Sterownik powinien generować sygnały sterujące do sygnalizacji ruchu pojazdów oraz do sygnalizacji ruchu pieszych. Należy więc wprowadzić korekcję fazy świateł w celu zapewnienia synchronizacji z innymi sterownikami.

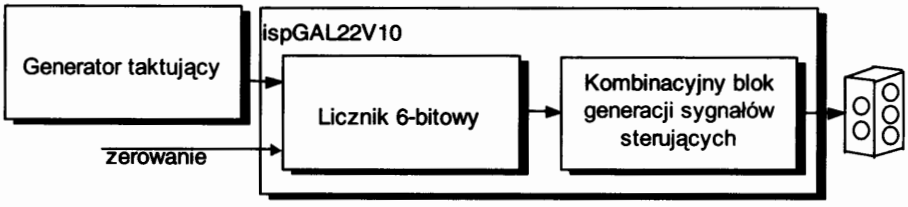
W praktycznych urządzeniach sygnalizacyjnych występuje wiele odmian sekwencji sterujących o różnym czasie trwania pełnego okresu, jak też o różnym rozmieszczeniu wartości sygnałów wyjściowych w cyklu. Poprawnie działające urządzenie nie może dopuszczać do sytu-

acji kolizyjnych. Najczęściej do sterowania panelu sygnalizacyjnego stosuje się sekwencję stanów przedstawioną w tablicy 7.7.

Tablica 7.7 Przykładowa sekwencja sterowania światłami

Faza	Pojazdy	Piesi
I	zielone	czerwone
II	żółte	czerwone
III	czerwone	zielone
IV	czerwone	zielone przerywane
V	czerwone + żółte	czerwone

W przedstawionym projekcie sterownika zastosowano układ programowalny ispGAL22V10, który zawiera blok licznika 6-bitowego oraz blok kombinacyjny, generujący wartości sygnałów sterowania na pięciu wyjściach układu (rys. 7.30).



Rys. 7.30 Schemat blokowy układu sterownika

Układ jest taktowany zegarem zewnętrznym, którego częstotliwość należy dobrać w zależności od zamierzonego czasu trwania pełnego cyklu. Na przykład jeśli cykl sterowania trwa 40 sekund, to sygnał zegarowy powinien mieć częstotliwość 64 : 40 = 1,6 Hz. Cykl zliczania licznika podzielono w taki sposób, aby przypisać wszystkim stanom licznika odpowiednie wartości sygnałów sterujących (zgodnie z tablicą 7.7). Układ ma wejście zerujące, które jest przeznaczone do ustalenia fazy świateł względem sygnału zegara. Opis sterownika w języku VHDL ma postać

```
-- L7.5
library ieee;
use ieee.std_logic_1164.all;
entity sterownik is port (
   clock,reset: in std_logic;
   sam_czerwone,sam_zielone,sam_zolte: out std_logic;
   pie_czerwone,pie_zielone: out std_logic);
        --     sam_czerwone - czerwone dla samochodow
        --     sam_zielone  - zielone dla samochodow
        --     sam_zolte    - zolte dla samochodow
```

```
        --     pie_czerwone - czerwone dla pieszych
        --     pie_zielone  - zielone dla pieszych
end sterownik;
architecture swiatla of sterownik is
   signal licznik : integer range 0 to 63;
begin
  process(clock, reset)
  begin
    if reset='1' then
       licznik <= 0;
    elsif (clock'event and clock = '1') then
       if licznik = 63 then
          licznik <= 0;
       else
          licznik <= licznik + 1;
       end if;
    end if;
  end process;
  process(licznik)
  begin
    if (licznik>31) then
       sam_czerwone<='1';
    else sam_czerwone<='0';
    end if;
    if (licznik<30) then
       sam_zielone<='1';
    else sam_zielone<='0';
    end if;
    if (licznik=30) or (licznik=31) or (licznik>61)
       then sam_zolte<='1';
    else sam_zolte<='0';
    end if;
    if (licznik<32) or (licznik>61) then
       pie_czerwone<='1';
    else pie_czerwone<='0';
    end if;
    if ((licznik>31) and (licznik<53)) or (licznik=54)
       or (licznik=56) or (licznik=58) or (licznik=60)
       then pie_zielone<='1';
    else
       pie_zielone<='0';
    end if;
  end process;
end swiatla;
```

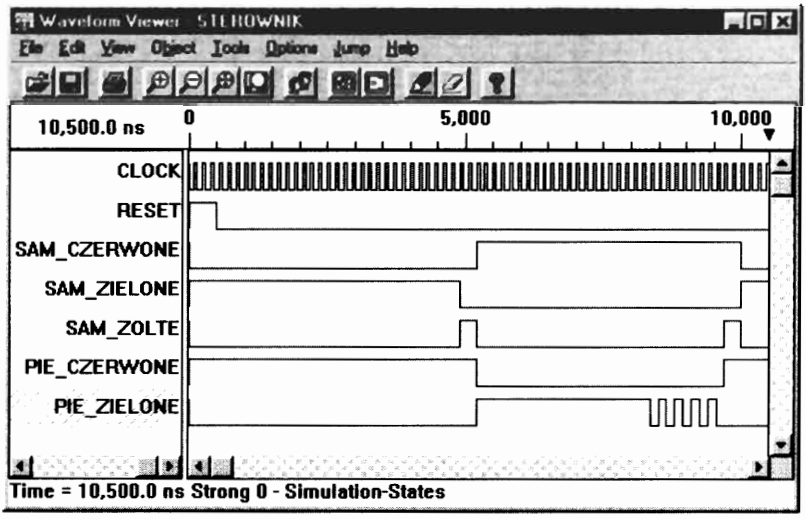
Opisany układ sterownika można zaprogramować w pojedynczym i niedrogim układzie scalonym ispGAL22V10. W tym układzie znajduje się tylko 10 wyjść. Po zarezerwowaniu pięciu z nich dla sygnałów sterujących pozostaje 5 przerzutników do budowy licznika. W takim razie

pokazany opis, wykorzystujący licznik 6-bitowy, nie powinien zostać właściwie odwzorowany we wskazanym układzie. Można to jednak zrobić dzięki bezpośredniemu wykorzystaniu najbardziej znaczącego bitu licznika do wytwarzania jednego z sygnałów sterujących, np. światłem czerwonym dla samochodów.

Przykładowy opis wektorów testowych dla symulatora w języku ABEL pokazany jest dalej. Rysunek 7.31 ilustruje działanie układu w postaci przebiegów czasowych uzyskanych w wyniku symulacji.

```
module sterownik
" wejscia
    clock pin;
    reset pin;
" wyjscia
    sam_czerwone pin;
    sam_zielone pin;
    sam_zolte pin;
    pie_czerwone pin;
    pie_zielone pin;

test_vectors([reset] -> [sam_czerwone,sam_zielone,
sam_zolte,pie_czerwone,pie_zielone])
   cycle clock (0,100) (1,50);
   [1] -> [.x.,.x.,.x.,.x.,.x.];
   wait 500;
   [0] -> [.x.,.x.,.x.,.x.,.x.];
   wait 10000;
end
```



Rys. 7.31 Przykładowy wynik symulacji projektu

Do symulacji układu w symulatorze VHDL można wykorzystać test przedstawiony poniżej.

```
--L7.5_TB
library ieee;
use ieee.std_logic_1164.all;
entity testbench is
end testbench;
architecture behavior of testbench is
   component sterownik
   port( clock : in std_logic;
         reset : in std_logic;
         sam_czerwone : out std_logic;
         sam_zielone : out std_logic;
         sam_zolte : out std_logic;
         pie_czerwone : out std_logic;
         pie_zielone : out std_logic);
   end component;
   signal clock :  std_logic;
   signal reset :  std_logic;
   signal sam_czerwone :  std_logic;
   signal sam_zielone :  std_logic;
   signal sam_zolte :  std_logic;
   signal pie_czerwone :  std_logic;
   signal pie_zielone :  std_logic;
begin
   uut: sterownik port map(
         clock => clock,
         reset => reset,
         sam_czerwone => sam_czerwone,
         sam_zielone => sam_zielone,
         sam_zolte => sam_zolte,
         pie_czerwone => pie_czerwone,
         pie_zielone => pie_zielone);
    tb : process
begin
     reset <= '1'; clock <= '0';
     wait for 10 ns;
     reset <= '0';
     wait for 10 ns;
     for i in 0 to 63 loop
        clock <= '1'; wait for 10 ns;
        clock <= '0'; wait for 10 ns;
     end loop;
     wait;
   end process;
end;
```

7.6 ZADANIA

1. Zaprojektować 8-bitowy licznik programowalny stosując język VHDL. Lista funkcji zawiera: wybór rodzaju kodu, wybór kierunku zliczania, deklarację długości cyklu zliczania. Zrealizować projekt w jednym z układów programowalnych umieszczonych na płytce LAT10.

2. Napisać program testowy w języku VHDL do weryfikacji układu z zadania 1. Program powinien sprawdzać poprawność stanów licznika przy skokowo rosnącej częstotliwości impulsów podawanych na jego wejście zliczające. Celem jest wyznaczenie maksymalnej częstotliwości pracy układu. Następnie należy porównać wartość częstotliwości maksymalnej licznika wyznaczoną za pomocą programu testowego z wartością wyznaczoną za pomocą analizatora czasowego systemu *IspDesignEXPERT*.

3. Zaprojektować w języku VHDL układ stopera elektronicznego. Założyć, że układ dokonuje pomiaru pomiędzy dwoma impulsami (START i STOP) w zakresie od 0 do 30 s z rozdzielczością 0.1s. Przedstawić rozwiązanie przystosowane do możliwości płytki LAT10.

4. Przedstawić projekt sterownika windy do 4-piętrowego budynku. Założyć, że na kondygnacjach 0 i 4 znajduje się po jednym przycisku przywoławczym, natomiast na pozostałych są po dwa przyciski do wskazywania oczekiwanego kierunku ruchu windy. W windzie należy przewidzieć istnienie 5 przycisków do wyboru pięter, na których winda ma się zatrzymać, oraz jeden przycisk do sygnalizacji awarii. Do sterowania silnika zastosować dwa sygnały binarne: włączenia oraz kierunku ruchu. Sterownik powinien podejmować decyzję o kierunku ruchu windy oraz o miejscu zatrzymania.

5. Zaprojektować zamek szyfrowy z użyciem płytki LAT10. Wprowadzanie hasła może odbywać się tylko za pomocą klawiszy P1+P5. Przewidzieć różną długość hasła oraz możliwość sygnalizacji alarmu, np. po trzech nieudanych próbach otwarcia zamka.

8

SYMULACJA UKŁADÓW CYFROWYCH W SYSTEMIE PROJEKTOWYM *Active-HDL* 5.1

Dariusz Chalaba

System projektowy *Active–HDL* firmy *Aldec* stanowi zestaw interaktywnych programów do opisu oraz symulacji układów cyfrowych. W skład tego zestawu wchodzą: edytor schematu, edytor tekstu, edytor grafu stanów oraz symulator funkcjonalny i czasowy. System umożliwia stosowanie różnych metod opisu poszczególnych fragmentów projektu (schemat, graf stanów, języki *VHDL* i *Verilog*), a ponadto zapewnia możliwość automatycznej generacji programów testowych. Dzięki tym cechom system *Active–HDL* jest wszechstronnym i komfortowym narzędziem projektowym.

W lutym 2002 roku dostępne były następujące wersje systemu: *Active_HDL 5.1*, *Active_HDL 5.1XE*, *Active_HDL with Synplify* oraz *Riviera*. Producent oferował do wymienionych wersji (z wyjątkiem *Riviera*) bezpłatnie licencje na pełne wersje systemu na okres 10 dni. Ponadto oferowano darmową wersję demonstracyjną *Active_HDL 5.1* na okres 20 dni. Ma ona jednak pewne ograniczenia:

- maksymalny czas symulacji wynosi 10 μs;
- maksymalny rozmiar pliku do kompilacji wynosi 5 kB;
- brak możliwości wydruku projektów.

Środowisko *Active-HDL* zawiera następujące narzędzia: *Design Explorer* – do rozmieszczania projektów na dysku komputera, *Library Manager* – do zarządzania bibliotekami i ich zawartością, *Design Browser* (przeglądarka) – do przeglądania plików projektu, *Console* –

do wydawania poleceń dla systemu i wyświetlania generowanych komunikatów, oraz *Design Flow Manager* – graficzny interfejs użytkownika.

System zawiera również narzędzia do wykrywania błędów w projektach podczas symulacji: *Process* – do wyświetlania stanu procesów, *Call Stack* – okno wyświetlające listę podprogramów (funkcji i procedur) wykonywanych w czasie symulacji, *Watch* – okno do śledzenia wybranych sygnałów lub zmiennych zawartych w opisie projektu oraz *Data Flow* – okno do graficznego podglądu sygnałów wejściowych i wyjściowych w postaci przebiegów czasowych.

Projektant może korzystać z pomocy komfortowego narzędzia *Language Assistant,* które udostępnia wiele gotowych szablonów dla złożonych instrukcji i deklaracji języka VHDL i Verilog, jak również wiele gotowych przykładów opisu podstawowych układów cyfrowych i bloków funkcjonalnych: między innymi multipleksera, przerzutników, rejestru przesuwającego i trzech liczników.

System *Active–HDL* umożliwia symulację projektu przed i po syntezie logicznej, to znaczy zarówno symulację funkcjonalną jak i czasową. Systemy projektowe firm *Altera, Xilinx* i *Lattice* opisane w rozdziałach 5-7 wykonują symulacje projektu układu dopiero po przetworzeniu go do formatu EDIF.

8.1 INSTALACJA SYSTEMU

System *Active-HDL5.1* może zostać zainstalowany w komputerze PC z systemem Windows 95/98/ME/2000/XP/NT 4.0 z *Service Pack* co najmniej *v.3.* Program wymaga 128 MB pamięci RAM (producent zaleca 256 MB) i co najmniej 198 MB wolnego miejsca na dysku. Aby go zainstalować należy włożyć dysk do napędu CD-ROM z wersją instalacyjną systemu lub kliknąć na plik instalacyjny „ściągnięty" ze strony www.aldec.com firmy *Aldec.* Po uruchomieniu instalacji pojawia się okno kreatora informujące o kolejnych krokach, jakie użytkownik musi wykonać, aby prawidłowo zainstalować system. Wybieramy *Next* i przechodzimy do kolejnego okna. Producent przedstawia warunki użytkowania. W kolejnym oknie wpisujemy dane użytkownika programu instalacyjnego, a następnie wybieramy jedną z dostępnych opcji tj.: *VHDL, VERILOG* lub *MIX* (oba języki). Następnie wybieramy biblioteki z jakich chcemy korzystać w przyszłości. Określamy folder, w którym ma zostać zainstalowany system, oraz folder gdzie będą zapisywane projekty użytkownika.

Aby uniknąć zaśmiecania dysku zaleca się utworzyć następujące foldery do instalacji *C:\eda\aldec\Aldec5_1* oraz *C:\eda\aldec\ projekty*. W następnym oknie określamy licencję z jakiej będziemy korzystać podczas pracy z tym systemem. Dostępne wersje licencji to:

- demonstracyjna na okres 20 dni lub pełna na 10 dni;
- sieciowa;
- w postaci klucza sprzętowego umieszczonego w porcie USB lub w porcie równoległym.

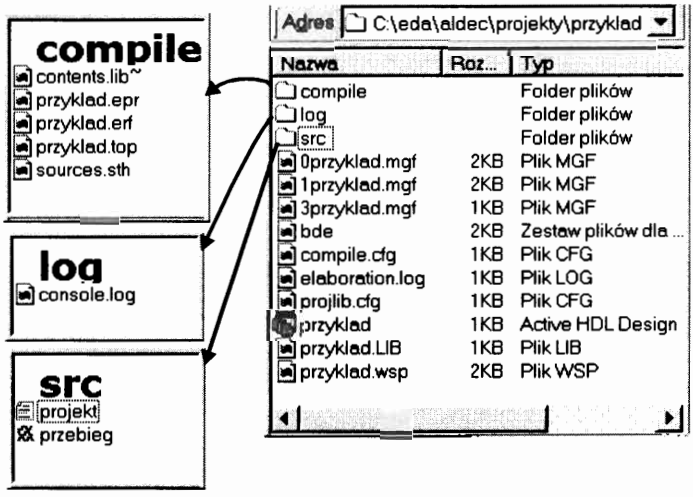
Dalej wybieramy formaty plików, które będą dostępne dla użytkownika, po czym program rozpoczyna instalację. Po zakończeniu instalacji systemu na pulpicie użytkownika pojawia się ikona *Active-HDL*. System jest gotowy do pracy.



Active-HDL5.1

8.2 ORGANIZACJA I ZASOBY BIBLIOTECZNE SYSTEMU

Każdy projekt w systemie *Active-HDL* ma własny folder. Na rys. 8.1 przedstawiono strukturę przykładowego projektu, który zawiera foldery *compile, log* i *src* oraz szereg plików projektu. Wymienione foldery zawierają: *compile* – pliki kompilacji, *log* – pliki komunikatów z okna *Console* oraz *src* – pliki źródłowe zawierające opis projektu i pliki z wynikami symulacji.



Rys. 8.1 Struktura folderu przykładowego projektu

Najczęściej używane typy plików w systemie projektowym:

- .vhd, .vho, .vhq, .tvhd, .thm, .vhi – tekst w języku VHDL;
- .v, .vei, .veo, .vcd, .vo, .vm – tekst w języku Verilog;
- .awf – graficzny opis przebiegów sygnałów;
- .cfg – dane konfiguracyjne;
- .adf – struktura projektu;
- .erf – błędy i uwagi z kompilacji;
- .asf – opis projektu w postaci grafu stanów;
- .do – polecenia dla symulacji;
- .edf, .edn, .edo – lista połączeń w formacie EDIF.

Aby ułatwić użytkownikowi poznanie systemu producent umieścił w menu *Help → On-line Documentation* podręcznik użytkowania systemu. W menu tym dostępny jest również krótki kurs języka VHDL i Verilog (*Interactive VHDL Tutorial* i *Interactive Verilog Tutorial*). Użytkownik ma do dyspozycji pakiety wymienione w tablicy 8.1.

Tablica 8.1 Wykaz bibliotek VHDL stosowanych w systemie *Active-HDL*

Nazwa biblioteki i pakietu	Zawartość

`Std`
`- standard`
`- textio`
	Typy predefiniowane i funkcje standardowe

`IEEE`
`- std_logic_1164`
`- numeric_std`
	Typy `std_logic, std_ulogic, signed, unsigned`, funkcje konwersji typów, funkcje logiczne i arytmetyczne

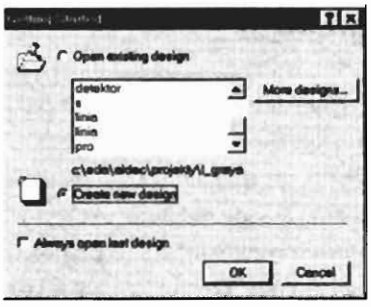
`IEEE`
`- std_logic_unsigned,`
`- std_logic_signed,`
`- std_logic_arith`
	Typy `std_logic, std_ulogic, signed, unsigned`, funkcje konwersji typów, funkcje logiczne i arytmetyczne opracowane przez firmę *Synopsys*

`IEEE`
`- waves_1164_declarations,`
`- waves_1164_frames,`
`- waves_1164_utilities,`
`- waves_interface`
	Standardowe funkcje i procedury związane z opisem wymuszeń i wartości oczekiwanych sygnałów do weryfikacji projektów

8.3 PROCES PROJEKTOWANIA

8.3.1 Uruchomienie systemu

Proces projektowania zostanie przedstawiony na przykładzie projektu licznika Graya. System projektowy *Active–HDL* uruchamia się



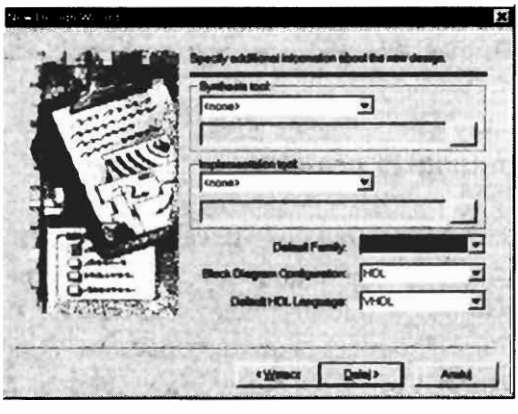
Rys. 8.2 Okno startowe programu

podwójnym kliknięciem na ikonę *Active–HDL*. Uruchomiony program wyświetla okno *Getting Started* (rys. 8.2). Aby otworzyć istniejący projekt zaznaczamy opcję *Open existing design* i wybieramy odpowiednią nazwę projektu. Nowy projekt tworzymy zaznaczając *Create new design* i naciskamy *OK*.

Projekt można zrealizować wybierając jedną z trzech możliwości przedstawionych w kolejnym oknie:

- dodawanie istniejących plików do tworzonego projektu (*Add existing resource files*),
- import projektu utworzonego w systemie *Active-CAD* (*Import a design from Active-CAD*);
- tworzenie projektu od podstaw (*Create an empty design*).

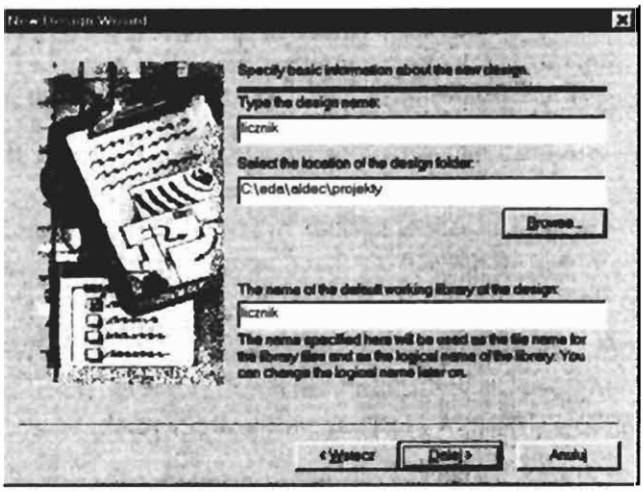
Wybieramy *Create an empty design* i naciskamy *Dalej*. W polu *Block Diagram Configuration* wybieramy opcję *HDL*, a w polu *Default HDL Language* opcję *VHDL* (rys. 8.3). Naciskamy przycisk *Dalej*.



Rys. 8.3 Wybór ustawień wstępnych

8.3.2 Organizacja projektu

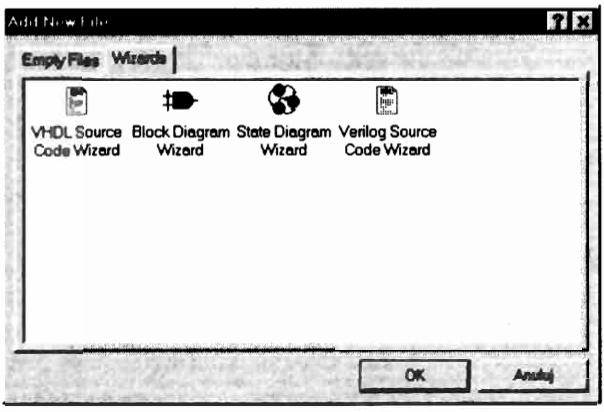
W oknie na rys. 8.4 wpisujemy nazwę folderu tworzonego projektu – *licznik*, określamy ścieżkę dostępu do niego – *C:\eda\aldec\projekty* oraz ustalamy nazwę biblioteki roboczej – *licznik*. System dla każdego projektu tworzy własną, roboczą bibliotekę i domyślnie jej nazwę przyjmuje taką samą jak nazwę folderu projektu. Po wykonaniu tych czynności naciskamy przycisk *Zakończ* i program przechodzi do głównego okna. W utworzonym folderze będą zapisywane wszystkie operacje związane z opisem oraz przetwarzaniem modelu tworzonego projektu. W pierwszej kolejności powstaje plik z rozszerzeniem *.adf* zawierający informacje o tworzonym projekcie. Z okna przeglądarki wybieramy opcję *Add New File* i tworzymy moduł opisujący nasz projekt.



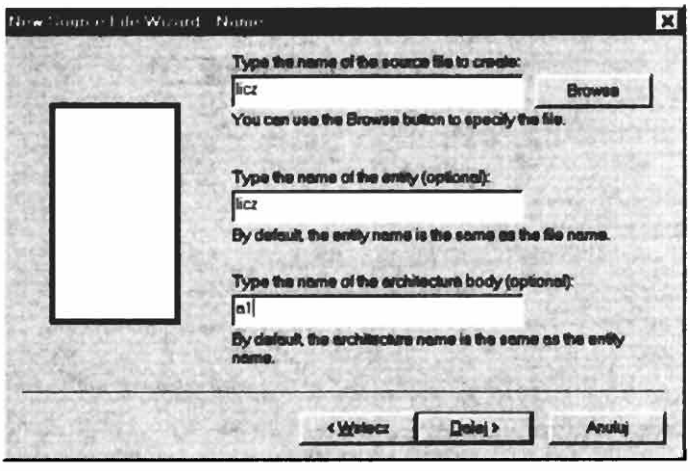
Rys. 8.4 Wprowadzenie nazw folderów projektu

8.3.3 Opis i kompilacja projektu

System projektowy *Active–HDL* umożliwia opisanie projektu z użyciem wygodnego narzędzia wspomagającego zwanego kreatorem (*Wizard*). Dostępne są kreatory schematu blokowego (*Block Diagram Wizard*), grafu stanów (*State Diagram Wizard*), opisu w języku VHDL (*VHDL Source Code Wizard*) lub w języku Verilog (*Verilog Source Code Wizard*). Wybieramy *VHDL Source Code Wizard* (rys. 8.5) i naciskamy przycisk *OK*.

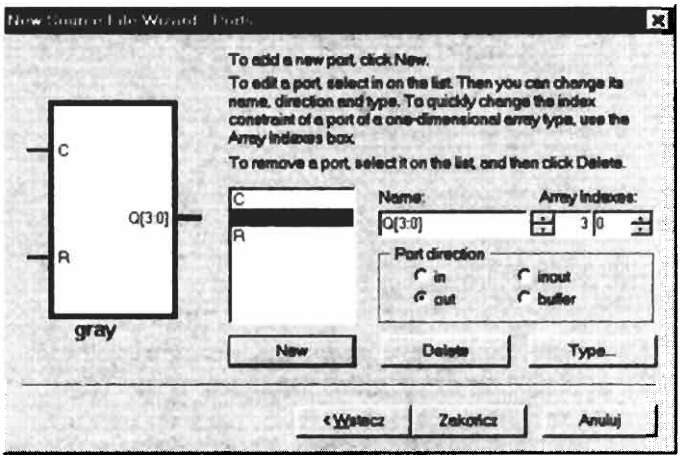


Rys. 8.5 Wybór sposobu opisu projektu.

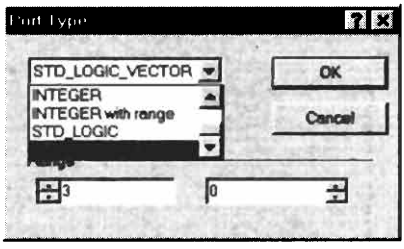


Rys. 8.6 Określenie nazw pliku projektu, jego jednostki i architektury

Zgodnie z rys. 8.6 wpisujemy nazwy: pliku źródłowego projektu (*licz*), jednostki projektowej (`licz`) oraz architektury (`a1`). Jeśli nie zostaną podane nazwy jednostki i jej architektury to system domyślnie przyjmie nazwę projektu. Zaleca się nie używać różnych nazw dla pliku i jednostki projektu, gdyż część kompilatorów uznaje to za błąd. W dalszej kolejności określamy konfigurację wejść oraz wyjść układu (rys. 8.7). Naciskamy przycisk *New*, wpisujemy nazwę sygnału w polu *Name*, określamy rodzaj sygnału: `in`, `out`, `inout` lub `buffer`. Program automatycznie przypisuje sygnałom typ `std_logic`. Aby wybrać inny typ naciskamy przycisk *Type* (rys. 8.8). Dodawanie kolejnych sygnałów odbywa się przyciskiem *New*, aby je zakończyć wciskamy *Zakończ*.

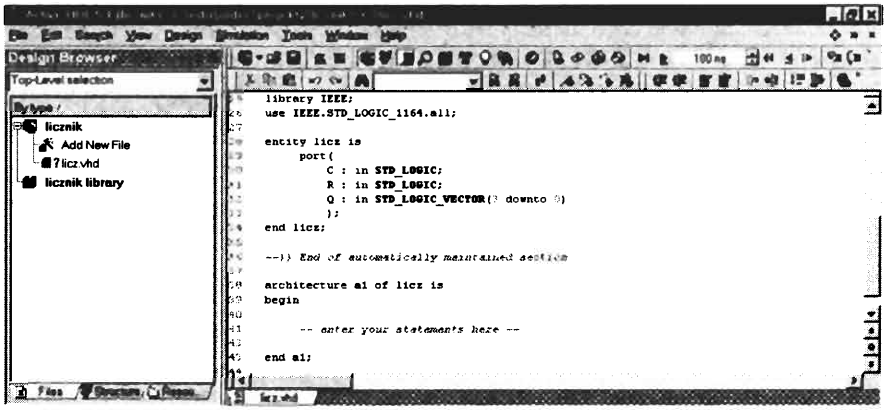


Rys. 8.7 Tworzenie portów projektu



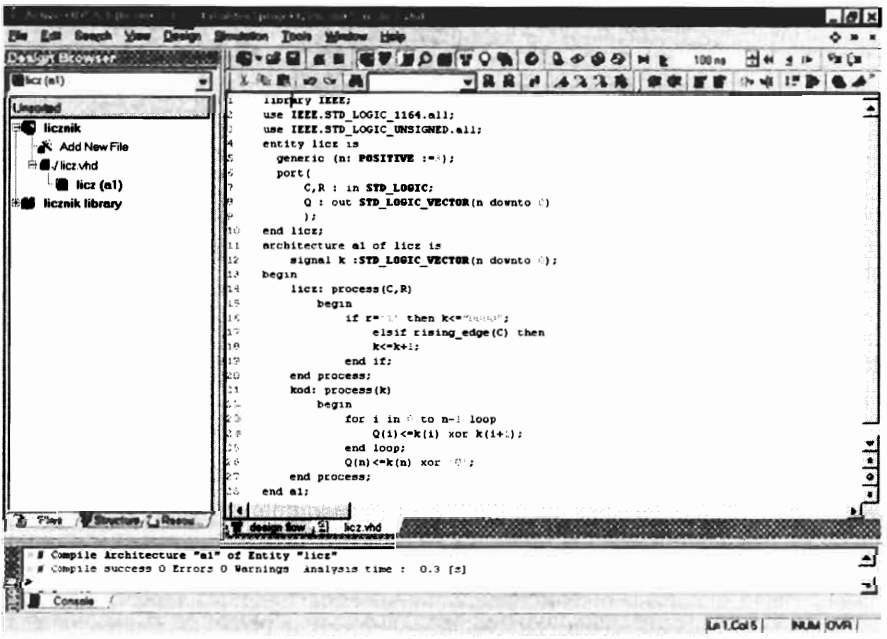
Rys. 8.8 Wybór typu sygnału

Program automatycznie generuje szablon w języku VHDL (rys. 8.9). Domyślnie w każdym szablonie umieszcza się klauzule użycia pakietu `IEEE.STD_LOGIC_1164.all`. Następnie uzupełniamy opis architektury w języku VHDL jak na rys. 8.10 – w naszym przypadku jest to opis licznika Graya.



Rys. 8.9 Szablon projektu

Proces kompilacji programu rozpoczynamy od wybrania z menu głównego (rys. 8.10) opcji *Design* → *Compile* (klawisz funkcyjny F11 lub ikona o nazwie *Compile* na pasku narzędziowym). Poprawiamy ewentualne błędy, jakie rozpoznał kompilator i wyświetlił w oknie *Console*. Jeśli wynik kompilacji był pomyślny to obok nazwy pliku pojawia się znak √ w przeciwnym razie – znak *x*.



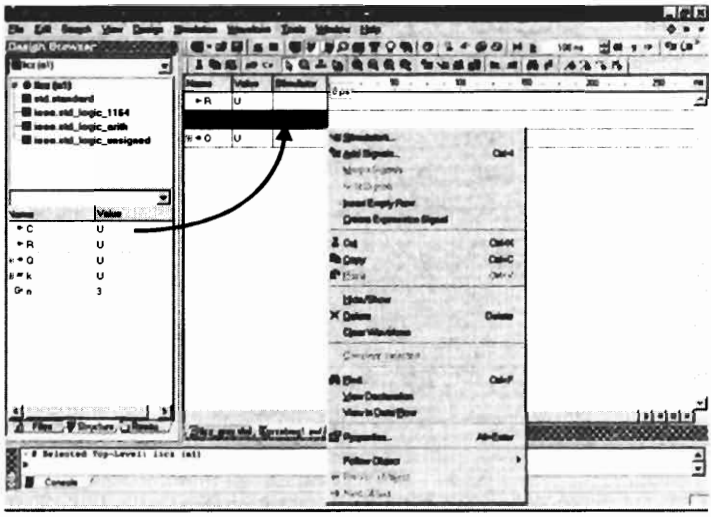
Rys. 8.10 Opis licznika Graya w języku VHDL

8.4 WERYFIKACJA PROJEKTU

8.4.1 Symulacja funkcjonalna

Symulacja funkcjonalna służy do sprawdzenia poprawności działania projektowanego układu. Stosując ją można zredukować czas potrzebny na usunięcie błędów logicznych i prześledzenie poprawności opisu przed wykonaniem syntezy projektu.

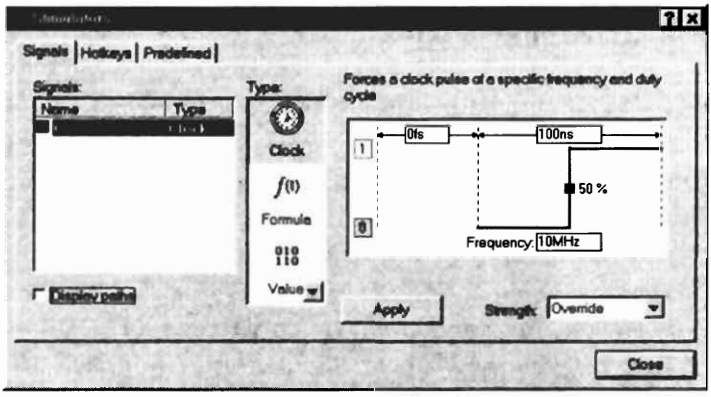
Aby wykonać symulacje funkcjonalną wybieramy architekturę projektu *licz(a1)* za pomocą opcji *Design* → *Settings* → *General* w polu *Top-Level Selection for Simulation* i potwierdzamy *OK*. Następnie z menu głównego wybieramy opcję *Simulation* → *Initialize Simulation*. Pojawia się okno symulatora (rys. 8.11).



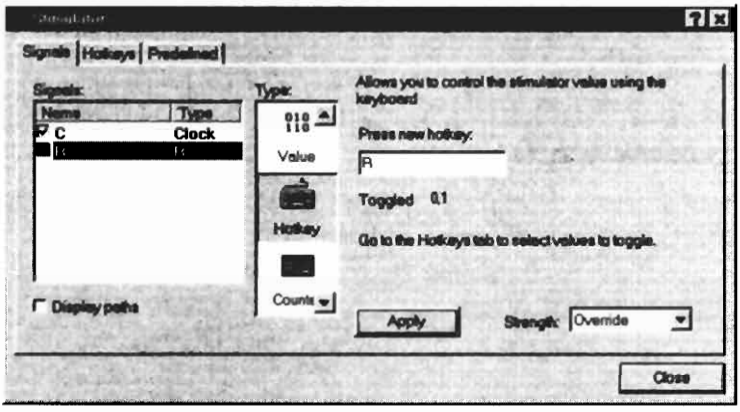
Rys. 8.11 Okno symulatora

System *Active-HDL* oferuje kilka sposobów definiowania przebiegów czasowych dla wirtualnych sygnałów wejściowych. Wykorzystamy je do opisu sygnałów wejściowych naszego modelu.

Z menu *File* wybieramy opcję *New* → *Waveform*, myszką przeciągamy nazwy sygnałów wejściowych i wyjściowych z dolnego okna przeglądarki (rys. 8.11) do tablicy sygnałów w oknie symulatora. Następnie definiujemy przebiegi sygnałów. Zaznaczamy sygnał zegara C lewym klawiszem myszki, wybieramy z menu głównego *Waveform* → *Stimulators* lub klikamy na ikonę o nazwie *Stimulators* na pasku narzędziowym (rys. 8.11). Wybieramy typ *Clock*, wpisujemy w polu *Frequency* wartość częstotliwości, np. 10MHz (rys. 8.12). W polu *Strength* wybieramy nadrzędny priorytet sygnału C (*Override*)



Rys. 8.12 Definiowanie sygnału zegara C



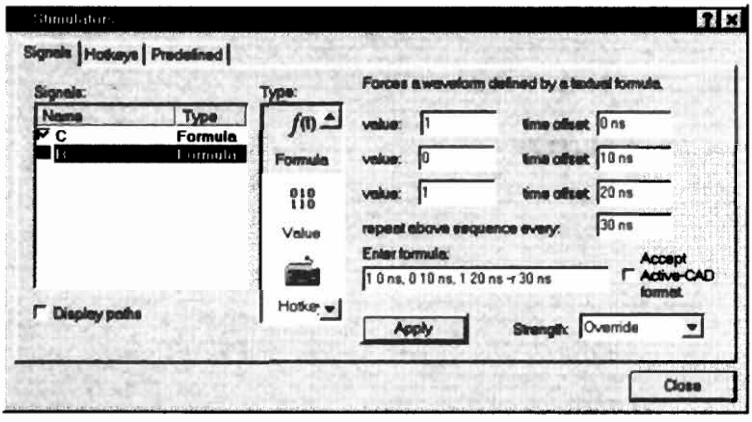
Rys. 8.13 Przypisanie klawisza typu *Hotkey* sygnałowi R

w stosunku do sygnału wytworzonego w czasie symulacji (sygnał przyjmuje wartość zdefiniowaną przez nas, a jego wartość z symulacji jest pomijana). Zatwierdzamy przyciskiem *Apply*.

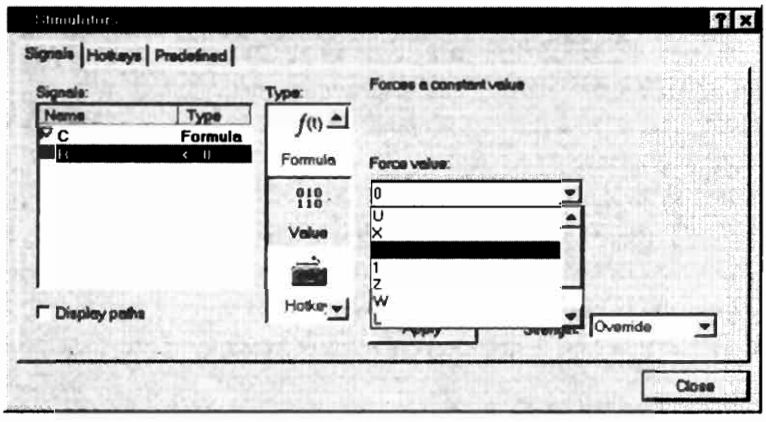
Wartość sygnału R może być zmieniana klawiszem *R* z klawiatury. W tym celu wybieramy typ *Hotkey*, a w polu *Press new hotkey* wpisujemy literkę klawisza, który będzie dokonywał zmiany dwóch stanów sygnału (0, 1). Zatwierdzamy przyciskiem *Apply* (rys. 8.13). Wybranie z okna *Stimulators* (rys. 8.13) zakładki *Hotkeys* powoduje wyświetlenie listy przypisanych klawiszy typu *Hotkey*.

Wartości sygnałów wejściowych można również definiować używając typów:

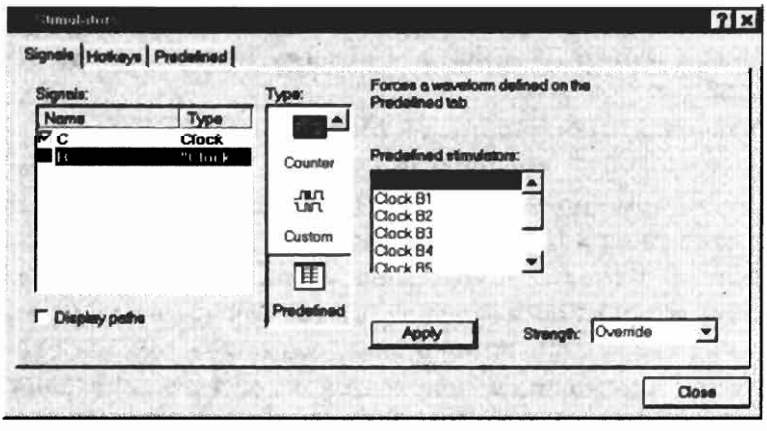
- *Formula* – podajemy maksymalnie trzy kolejne wartości sygnału, czas ich trwania, oraz czas po którym następuje powtórzenie sekwencji (rys. 8.14);
- *Value* – wybieramy jedną z 9 możliwych wartości sygnału (rys. 8.15);
- *Predefined* – korzystamy ze zdefiniowanych przebiegów czasowych o różnych częstotliwościach (rys. 8.16). Zmiany wartości częstotliwości przebiegów B0-B9 dokonujemy wybierając zakładkę *Predefined* w oknie *Stimulators*;
- *Counter* – wybieramy typ licznika w polu *Count type*, kierunek liczenia w polu *Count Direction* (*Up* lub *Down*), wartość początkową licznika w polu *Starting value*, wartość o jaką ma być dokonywana jego zmiana w polu *Modify by*, okres w polu *Count every* oraz podstawę systemu liczbowego w polu *Use radix* (rys. 8.17).



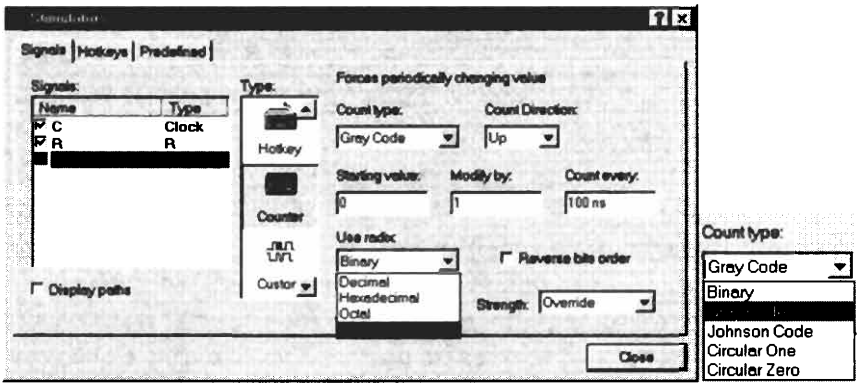
Rys. 8.14 Określenie zmian wartości sygnału z użyciem typu *Formula*



Rys. 8.15 Określenie zmian wartości sygnału z użyciem typu *Value*

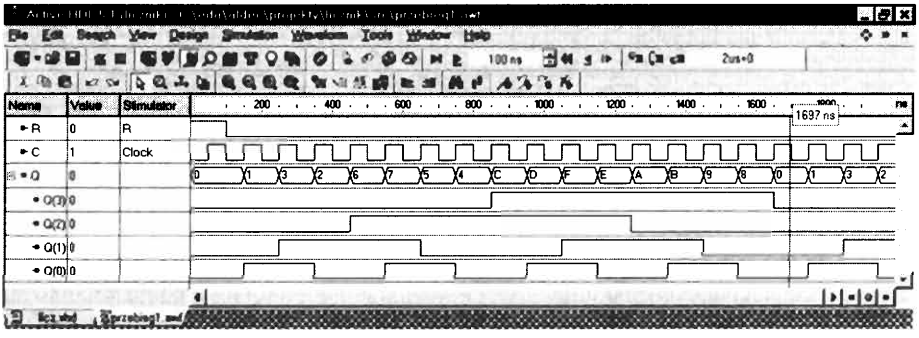


Rys. 8.16 Wybór zegara z listy *Predefined stimulators*

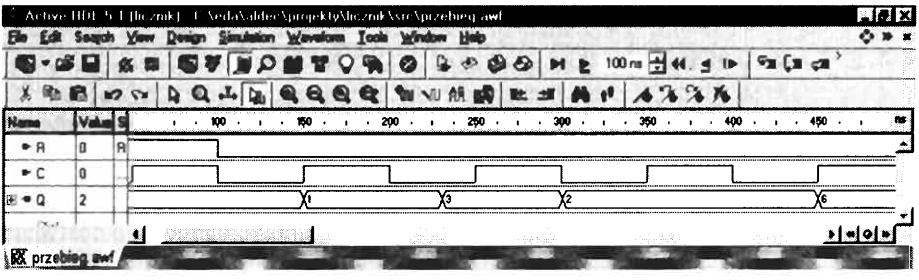


Rys. 8.17 Określenie przebiegu sygnału z użyciem typu *Counter*

Aby rozpocząć symulację z menu głównego wybieramy *Simulation* → *Run For* (ikona *Run For* na pasku narzędziowym). Wartość sygnału R możemy zmieniać po każdym kroku symulacji (klawisz *R*). Wyniki symulacji przedstawia rys. 8.18. Zapisujemy je do pliku *przebieg1.awf* (*Save As*).

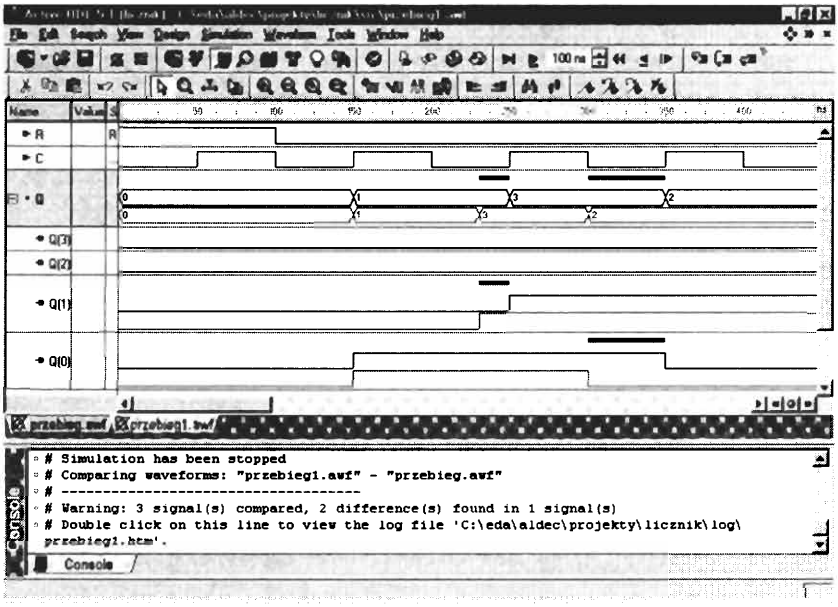


Rys. 8.18 Wynik symulacji funkcjonalnej



Rys. 8.19 Przebieg sygnału po modyfikacji

System umożliwia automatyczne porównanie przebiegów. Dla ilustracji tego procesu posłużymy się przykładem. Modyfikujemy odpowiedź układu Q za pomocą *Waveform* → *Edit Mode* lub ikony na pasku narzędziowym, jak na rys. 8.19 i zapisujemy (*Save As*) zmiany do pliku *przebieg.awf*. W oknie symulatora wyświetlamy zawartość pliku *przebieg1.awf* i z menu *Waveform* wybieramy opcję *Compare Waveform*. Zaznaczamy myszką plik *przebieg.awf* w oknie *Compare Waveform* i naciskamy przycisk *Otwórz*. Następuje porównanie przebiegów sygnałów zawartych w obu plikach. Wykryte różnice w kształtach przebiegów są oznaczane grubą kreską (rys. 8.20). W oknie *Console* jest dodatkowo wyświetlany raport o wykrytych różnicach.



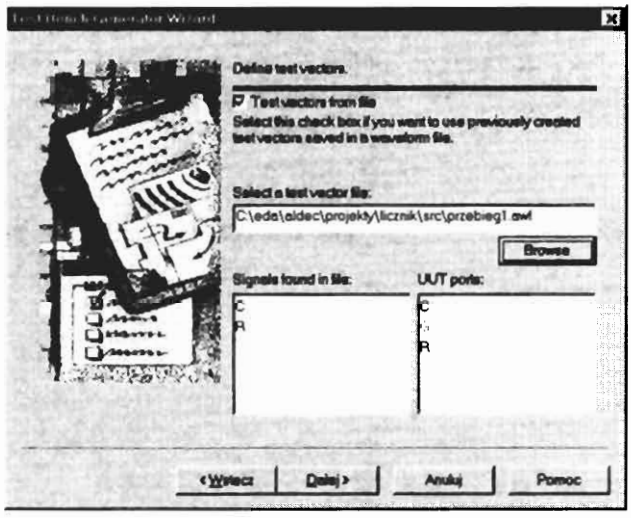
Rys. 8.20 Wynik porównania sygnałów

8.4.2 Program testowy

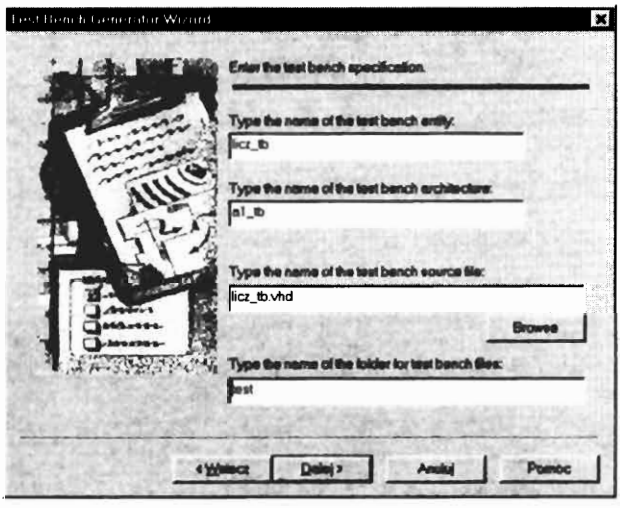
System *Active-HDL* umożliwia tworzenie i wykorzystanie programów testowych (testów) do weryfikacji projektu. Test jest modułem VHDL, zawierającym opis tekstowy zmian sygnałów wejściowych w postaci wektorów testowych. Dalej zostanie przedstawiona opcja automatycznego generowania testu.

Po kompilacji naszego projektu klikamy prawym klawiszem myszy w oknie przeglądarki na wiersz z nazwą architektury układu

przeznaczonego do testowania – *licz(a1)*. W pojawiającym się oknie wybieramy opcję *Generate Test Bench*. Na ekranie pojawia się okno, w którym ustawiamy nazwę jednostki `licz` i architektury `a1`, oraz wybieramy *Single Process* w polu *Test Bench Type*.
Wybieramy opcję *Test vectors from file*, aby program wczytał wektory testowe z pliku zawierającego zdefiniowane przebiegi sygnałów wejściowych. W polu *Select a test vector file* wpisujemy ścieżkę lokalizacji pliku, na przykład: *C:\eda\aldec\projekty\licznik\src\ przebieg1.awf*.



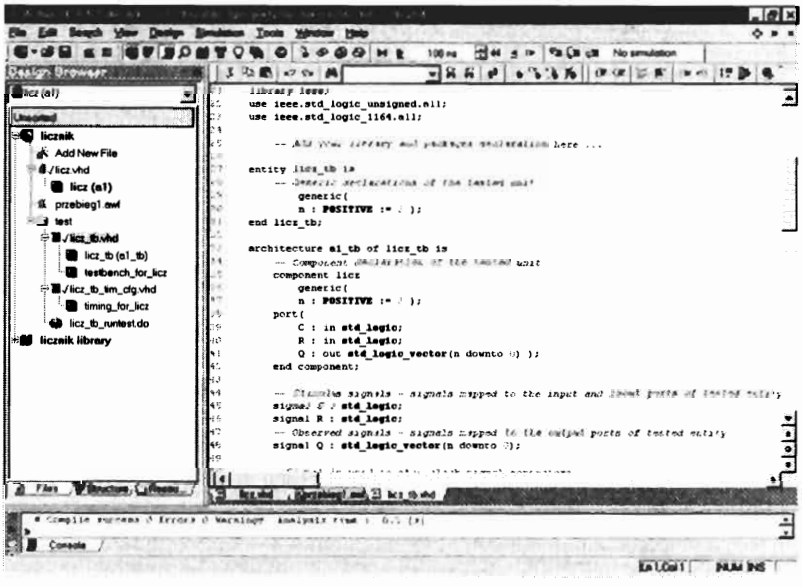
Rys. 8.21 Określenie lokalizacji pliku wektorów testowych



Rys. 8.22 Określenie nazw jednostki, architektury, pliku i folderu testu

W polu *Signals found in file* (rys. 8.21) wyświetlane są nazwy sygnałów zdefiniowane w podanym pliku, a w polu *UUT ports* ich odpowiedniki w programie testowym. Kontynuujemy naciskając *Dalej*.

Program proponuje nazwy dla testu, akceptujemy je lub wprowadzamy własne (rys. 8.22). W pojawiającym się oknie potwierdzającym gotowość systemu do utworzenia testu zaznaczamy opcję *Generate*. Program tworzy plik konfiguracyjny wykorzystany dalej w rozdziale 8.4.3 do symulacji czasowej. Naciskamy przycisk *Zakończ*. Utworzone przez program pliki znajdują się w folderze test w oknie przeglądarki (rys. 8.23). Aby je zobaczyć klikamy na nazwę wybranego pliku (np. *licz_tb.vhd*). Jego zawartość pojawia się w oknie po prawej stronie ekranu. Poniżej przedstawiono wydruk testu `L8.1` wygenerowanego do testowania projektu licznika Graya. Obok komentarzy utworzonych przez program dodano ich objaśnienie w języku polskim.



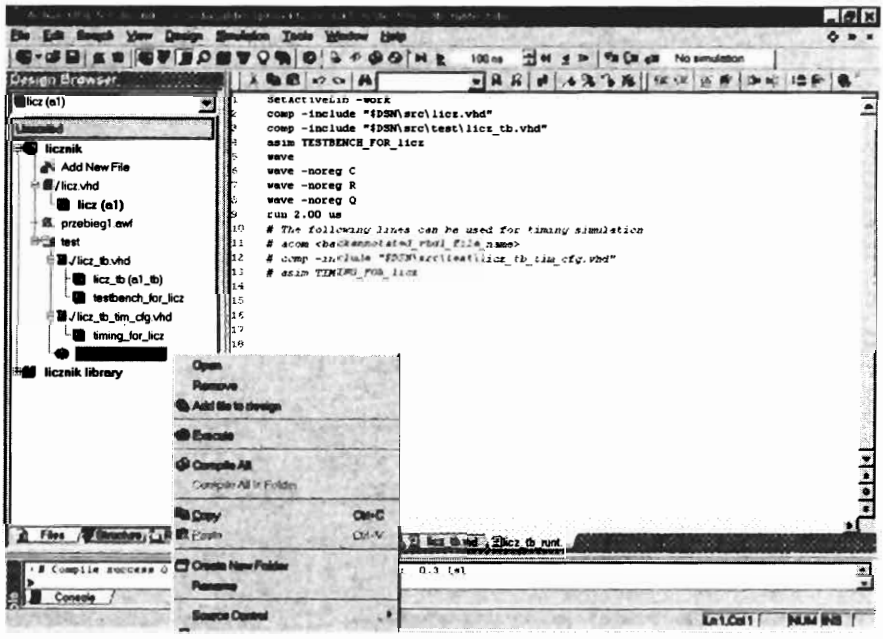
Rys. 8.23 Główne okno po wygenerowaniu programu testowego

```
--  L8.1
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
-- Add your library and packages declaration here
--(Dodaj wlasna biblioteke lub pakiet)
entity licz_tb is
   generic(
         N : POSITIVE := 3 );
end licz_tb;
```

```
architecture a1_tb of licz_tb is
-- Component declaration of the tested unit
-- (Deklaracja testowanego komponentu)
component licz
   generic(
         N : POSITIVE := 3 );
   port( C : in std_logic;
         R : in std_logic;
         Q : out std_logic_vector(N downto 0) );
end component;
-- Stimulus signals - signals mapped to the input and
-- inout ports of tested entity
-- (Wymuszenia - sygnaly podawane na wejscia)
   signal C : std_logic;
   signal R : std_logic;
-- Observed signals - signals mapped to the output
-- ports of tested entity
-- (Odpowiedzi ukladu - sygnaly obserwowane na
-- wyjsciach)
   signal Q : std_logic_vector(N downto 0);
-- Signal is used to stop clock signal generators
-- (Sygnal do zatrzymania symulacji)
   signal END_SIM: BOOLEAN:=FALSE;
-- Add your code here ...
-- (Wstaw wlasne deklaracje)
begin
-- Unit Under Test port map
-- (Okreslenie polaczen testowanego modulu)
UUT : licz
   generic map (N => N)
   port map (
         C => C,
         R => R,
         Q => Q);
STIMULUS: process
   begin  -- of stimulus process --(Opis wymuszen)
         (R <= '1'; -- (Kasowanie)
         wait for 100 ns;
         R <= '0';
         wait for 2 us;
         END_SIM <= TRUE;
-- end of stimulus events
-- (Zakonczenie symulacji po 2 us)
         wait;
end process;
CLOCK_C : process
   begin
         if END_SIM = FALSE then
               C <= '0';
```

```
            wait for 50 ns;
            else
                  wait;
            end if;
            if END_SIM = FALSE then
                  C <= '1';
                  wait for 50 ns;
            else
                  wait;
            end if;
   end process;
   -- Add your stimulus here ...
   -- (Dodaj wlasne opisy wymuszen)
end a1_tb;
configuration TESTBENCH_FOR_licz of licz_tb is
   for a1_tb
      for UUT : licz
            use entity work.licz(a1);
      end for;
   end for;
end TESTBENCH_FOR_licz;
```

Aby wykonać symulację projektu zaznaczamy w oknie przeglądarki plik poleceń *licz_tb_runtest.do*, klikamy prawym klawiszem myszki i z rozwiniętego menu wybieramy polecenie *Execute* (rys. 8.24).



Rys. 8.24 Uruchomienie programu testowego

Program automatycznie wykonuje symulację projektu, a następnie tworzy plik z wynikami. Otrzymane wyniki są takie same jak otrzymane w rozdziale 8.4.1 (rys. 8.18). Niestety przedstawiony test `L8.1` nie umożliwia automatycznego sprawdzenia poprawności działania projektu. Weryfikacji dokonuje projektant obserwując wyniki symulacji. W celu wykonania automatycznego sprawdzenia poprawności działania układu należy zmodyfikować program testowy wprowadzając dodatkowy opis `L8.1a` w miejsce `Wstaw wlasne deklaracje` oraz `Dodaj wlasne opisy wymuszen`.

```
--L8.1a
-- Add your code here
-- (Wstaw wlasne deklaracje)
type test_vec is array (natural range <> )of
std_logic_vector(N downto 0));
constant  test: test_vec:=
-- tablica wektorow - wartosci oczekiwane
("0000","0001","0011",
 "0010","0110","0111",
 "0101","0100","1100",
 "1101","1111","1110",
 "1010","1011","1001",
 "1000","0000","0001");

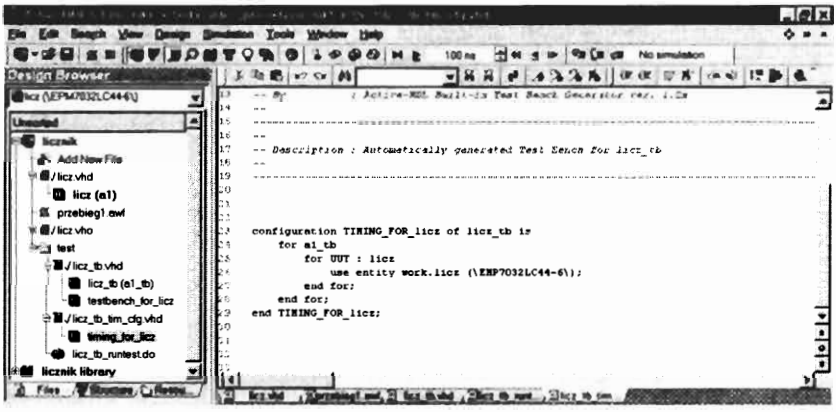
-- Add your stimulus here ...
-- (Dodaj wlasne opisy wymuszen)
automat: process
   begin
         wait for 60 ns;
         for i in test'range loop
 -- sprawdzanie wektorow wyjsciowych
               if Q/=test(i) then
               assert false report "Wystapil blad";
         else report "Uklad dziala poprawnie";
         end if;
         wait for 100 ns;
      end loop;
end process;
```

8.4.3 Symulacja czasowa

System *Active-HDL* nie umożliwia wykonania syntezy projektu. Symulację czasową w takim przypadku należy wykonać opierając się na modelu czasowy otrzymany z innego systemu projektowego lub korzystając ze schematu procesu projektowego *Design Flow Manager* opisanego w rozdziale 8.4.4.

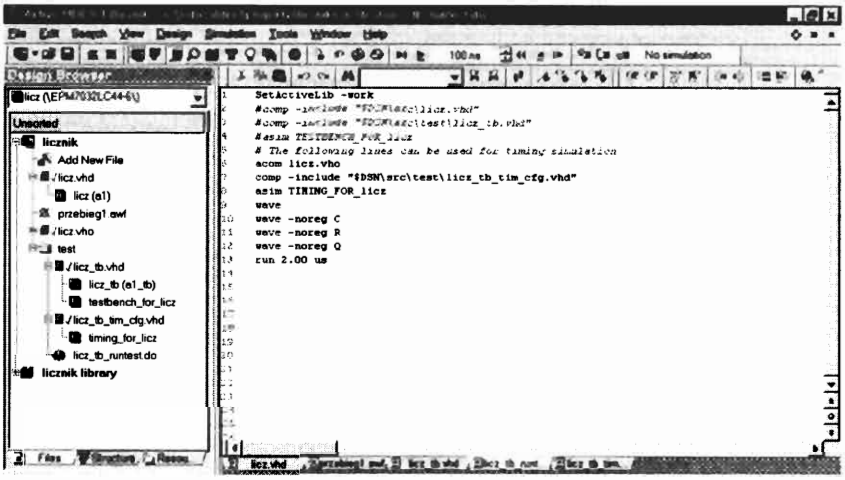
Do uzyskania modelu czasowego wykorzystujemy system *Max+PlusII* firmy *Altera*. Mając model projektu z parametrami czasowymi możemy do jego weryfikacji zastosować już utworzony program testowy w rozdziale 8.4.2.

Uruchamiamy system *Max+PlusII* i otwieramy nasz projekt *licz.vhd* (*File* → *Open*). Ustawiamy plik jako bieżący *File* → *Project* → *Set Project Current File*. Z menu wybieramy opcję *Assign* → *Device*, a w oknie *Device Family* układ *EPM7032LC44-6*. Z menu głównego otwieramy okno kompilacji (*Compiler*), wybieramy opcję *Interface* → *VHDL Netlist Writer Settings* i zaznaczamy *VHDL Output File[.vho]*. Następnie zaznaczamy opcję *Interface* → *VHDL Netlist Writer* i naciskamy przycisk *Start* w oknie *Compiler*. System projektowy *Max+PlusII* generuje plik *licz.vho*, który należy dołączyć do okna z naszym projektem w systemie *Active-HDL*.

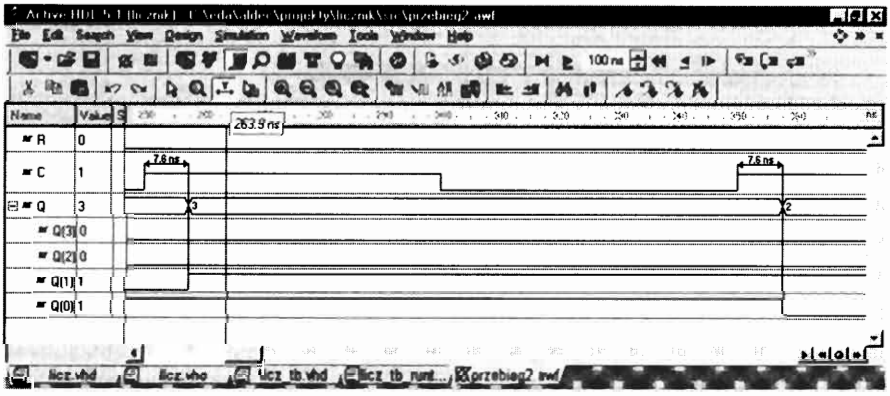


Rys. 8.25 Ustawienie konfiguracji do symulacji czasowej

W tym celu wybieramy opcję *Add New File*, a następnie opcję *Add Existing File*. Zaznaczamy dołączony plik *licz.vho* (rys. 8.25) i kompilujemy go. Plik ten nie zawiera już instrukcji `generic` używanej w module *licz.vhd*. W czasie syntezy w miejsce zmiennej `n` jest wpisywana jej wartość (`n=3`). Przeprowadzamy symulację czasową wykonując czynności jak podczas symulacji funkcjonalnej (rozdział 8.4.1). Aby wykonać symulację z użyciem programu testowego edytujemy plik konfiguracyjny *licz_tb_tim_cfg.vhd*. Wstawiamy nazwę architektury *\EPM7032LC44-6* jak na rys. 8.25. Modyfikujemy również plik *licz_tb.vhd* zmieniając wiersze z instrukcją `generic` na komentarze, natomiast w miejsce zmiennej `n` wpisujemy liczbę `3`. Zmian dokonujemy także w pliku *licz_tb_runtest.do* jak na rys. 8.26.



Rys. 8.26 Dołączenie pliku z parametrami czasowymi do testu

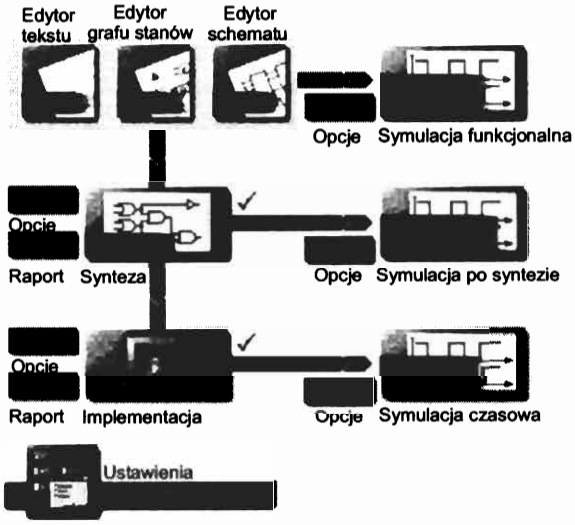


Rys. 8.27 Wynik symulacji czasowej

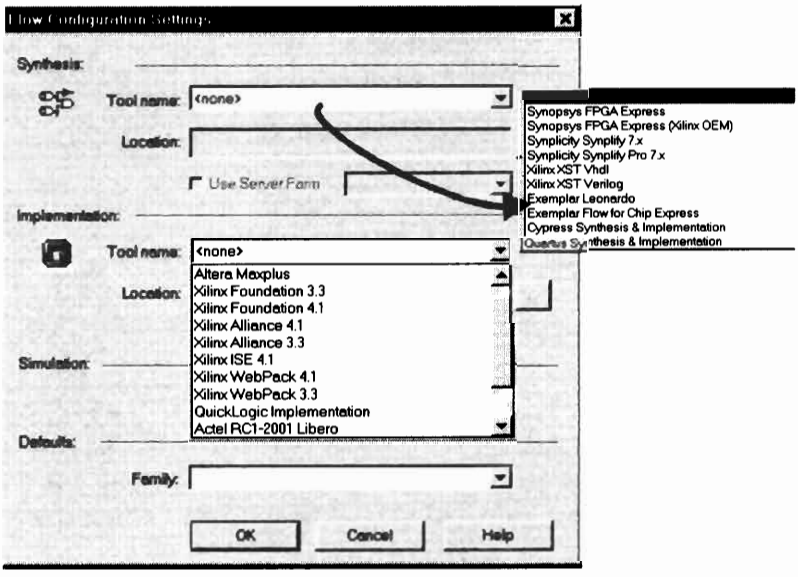
Kompilujemy wszystkie pliki w folderze test używając opcji *Compile All*, a następnie dla pliku *licz_TB_runtest.do* wykonujemy polecenie *Execute*. Wynik symulacji modelu czasowego projektu przedstawia rys. 8.27.

8.4.4 Schemat procesu projektowania

System *Active-HDL* zawiera również graficzny interfejs *Design Flow Menager*. Interfejs ten pozwala w łatwy sposób korzystać z różnych narzędzi potrzebnych w kolejnych etapach projektowania (rys. 8. 28).



Rys. 8.28 Okno schematu projektowania



Rys. 8.29 Okno wyboru narzędzi projektowych

Interfejs *Design Flow Menager* umożliwia opis projektów w postaci tekstowej (ikona *HDE*), grafu stanów (ikona *FSM*) oraz schematu graficznego (ikona *BDE*). Aby przeprowadzić symulacje funkcjonalną opisanego projektu należy kilknać na przycisk *options*, ustawić bieżący moduł w polu *Top-Level Unit* i wybrać *OK*. Po kliknięciu na ikonę *functional simulation* postępujemy tak jak w rozdziale 8.4.1.

Przed przystąpieniem do syntezy i implementacji należy wybrać odpowiednie narzędzia. Klikamy myszką na ikonę *Flow Settings* (rys. 8.29) i w polu *Synthesis Tool name* wybieramy jedno z zainstalowanych narzędzi do syntezy. Program automatycznie lokalizuje jego położenie. Podobnie postępujemy dla pola *Tool name* w obszarze *Imlementation.* W polu *Family* wybieramy rodzinę układów programowalnych i naciskamy *OK.* W celu przeprowadzenia syntezy klikamy na ikonę *synthesis,* a implementacji na ikonę *implementation.* System umożliwia przeprowadzenie symulacji po syntezie (ikona *post-synthesis simulation*) oraz symulacji czasowej po implementacji (ikona *timing simulation*). Przed wykonaniem symulacji należy za pomocą przycisku *options* ustawić odpowiedni moduł w polu *Top-Level Unit* (rys. 8.28).

8.5 PRZYKŁADY PROJEKTÓW

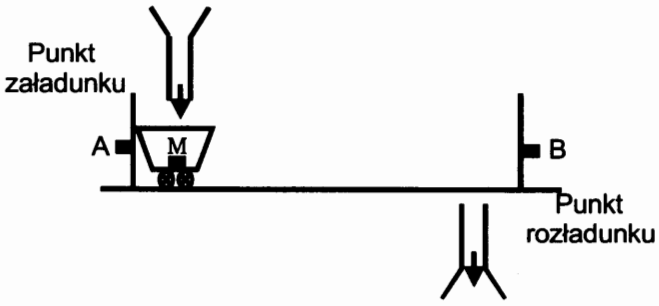
8.5.1 Układ sterowania ruchem wózka

Zaprojektować układ sterowania ruchem wózka transportującego sypki materiał (np. zboże) między punktem rozładunku samochodów A, a miejscem składowania materiału B (rys. 8.30). Załadowanie i rozładowanie wózka odbywa się automatycznie. Położenie wózka wykrywane jest przez czujniki w punktach A i B, które przyjmują stan 1 gdy wózek znajduje się w ich pobliżu. We wnętrzu wózka umieszczono czujnik ciężaru M. Gdy wózek jest załadowany, to M = 1, gdy pusty to M = 0.

Sterowanie ruchem wózka odbywa się według następującego algorytmu:

- pusty wózek zatrzymuje się w położeniu A do momentu, aż zostanie załadowany (czujnik M zmienia stan z 0 na 1);
- po załadowaniu w punkcie A wózek przesuwany jest do punktu B;
- gdy wózek dojedzie do punktu B zatrzymuje się i czeka, aż zostanie rozładowany;
- po rozładowaniu (M = 0), wózek wraca do punktu A;
- cykl powtarza się, gdy wózek znajdzie się w punkcie A.

Schemat rozmieszczenia czujników przedstawia rys. 8.30.



Rys. 8.30 Schemat transportu materiałów sypkich

Wózek sterowany jest za pomocą dwóch sygnałów `s(0)` i `s(1)` według opisu w tablicy 8.2:

Tablica 8.2 Tryby pracy układu

s(0)s(1)	Sterowanie
00	Stop
10	Ruch w lewo
01	Ruch w prawo
11	Niedozwolony

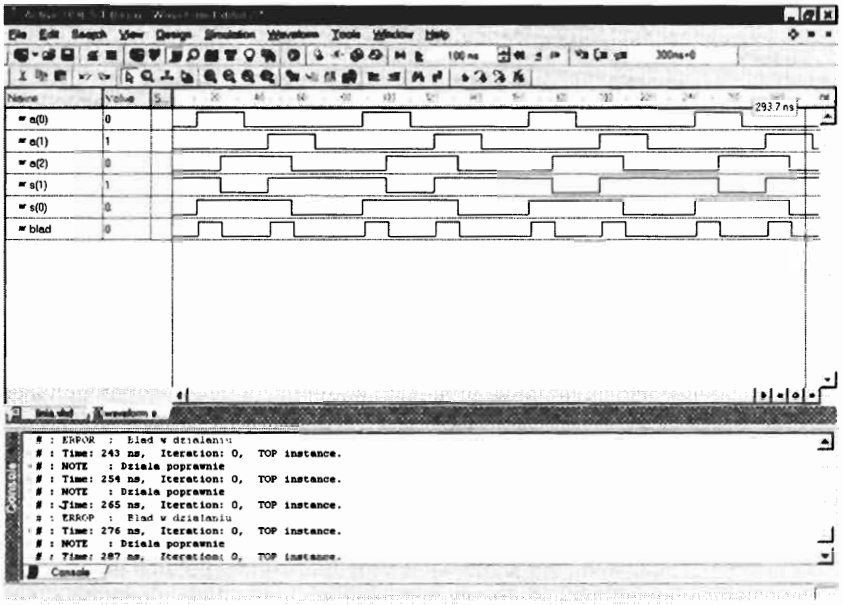
Na postawie tablicy 8.2 utworzono tablicę stanów układu (tablica 8.3).

Tablica 8.3 Tablica stanów układu

A(0)	A(1)	A(2)	s(0)	s(1)
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	1
1	1	0	-	-
1	1	1	-	-

Opis działania układu sterowania wózkiem w języku VHDL jest następujący:

```
-- L8.2
library IEEE;
use IEEE.std_logic_1164.all;
```



Rys. 8.31 Wyniki symulacji funkcjonalnej układu

```
entity linia is
      port (a: in std_logic_vector (0 to 2);
            s: out std_logic_vector (1 downto 0));
   end linia;
   architecture reg of linia is
    begin
      with a select
            s<= "10" when "000" | "010",
                "01" when "001" | "101",
                "00" when others;
   end reg;
```

Do sprawdzenia poprawności funkcjonalnej projektu napisano program testowy L8.3. Do sygnalizacji błędów wprowadzono sygnał blad. Test ten pozwala śledzić wyniki symulacji w postaci graficznej oraz tekstowej. Aby zademonstrować sposób jego działania w opisie L.8.2 wprowadzono błąd (L.8.2a). Wykrycie błędu przez program testowy obserwujemy na rys. 8.31. W miejscach wystąpień błędów generowany jest impuls sygnału blad.

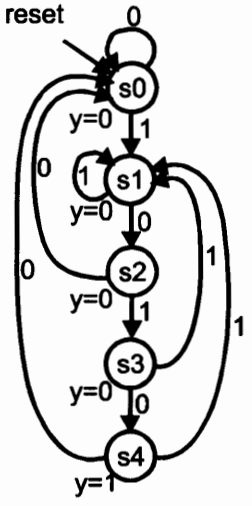
```
   --  L.8.2a
   with a select
      s<= "10" when "000" | "010",
          "01" when "001" | "101",
          "11" when others;-- wprowadzony blad
```

```
-- L8.3
library IEEE;
use IEEE.std_logic_1164.all;
entity linia_tb is
end linia_tb;
architecture TB_ARCHITECTURE of linia_tb is
component linia
    port(a : in  std_logic_vector (0 to 2);
         s : out std_logic_vector (1 downto 0));
end component;
    signal a : std_logic_vector (0 to 2);
    signal s : std_logic_vector (1 downto 0);
    signal blad : std_logic;
    type test_vec is record
         a : std_logic_vector (0 to 2);
         s : std_logic_vector (1 downto 0);
    end record;
    type test_vec_array is array (natural range <>)
                                    of test_vec;
    constant test: test_vec_array:=
         -- a,    s  - tablica wektorow testowych
        (("000","10"),
         ("100","00"),
         ("101","01"),
         ("001","01"),
         ("011","00"),
         ("010","10"),
         ("000","10"));
begin
 UUT : linia
   port map
          (a => a,
           s => s );
STIMULUS: process
   variable vec: test_vec;
begin
   blad<='0';
   for i in test'range loop
         vec:=test(i);
         a<=vec.a;
         wait for 1 ns;
-- sprawdzanie wektorow wyjsciowych
         if s/=vec.s then
           assert false report " Blad w dzialaniu ";
           blad<='1';
         else report "Dziala poprawnie";
           blad<='0';
         end if;
         wait for 10 ns;
```

```
   end loop;
end process;
end TB_ARCHITECTURE;
configuration TESTBENCH_FOR_linia of linia_tb is
for TB_ARCHITECTURE
   for UUT : linia
      use entity work.linia(reg);
   end for;
end for;
end TESTBENCH_FOR_linia;
```

8.5.2 Detektor ciągu bitów

Zaprojektować układ detektora ciągu bitów 1,0,1,0 pojawiających się szeregowo na wejściu danych X. Detektor został opisany grafem stanów jak na rys. 8.32. Układ ten ma na wyjściu Y stan 1 po wystąpieniu sekwencji x=1,0,1,0. We wszystkich innych przypadkach wyjście Y ma stan 0.



Rys. 8.32 Graf stanów detektora

W układzie można wyróżnić następujące stany:

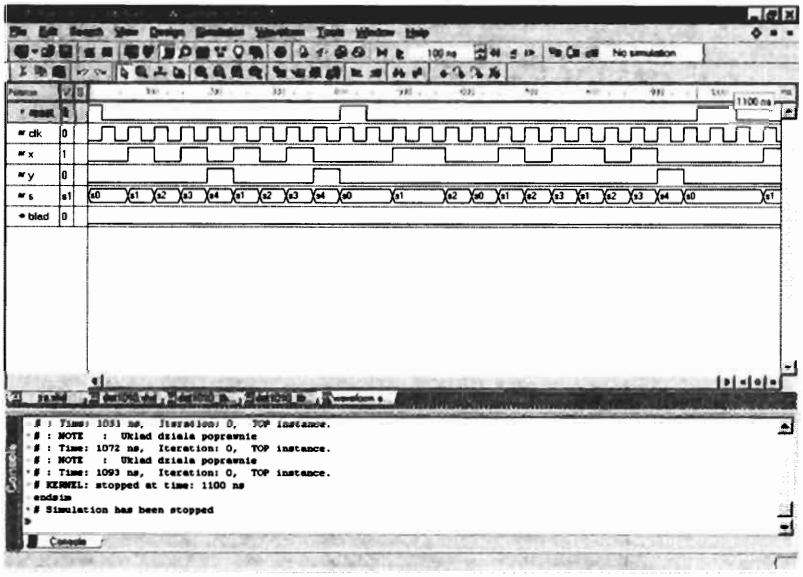
- S0 – nie wystąpił pierwszy element ciągu, y=0;
- S1 – wystąpił pierwszy element ciągu, y=0;
- S2 – wystąpiły dwa pierwsze elementy ciągu, y=0;
- S3 – wystąpiły trzy pierwsze elementy ciągu, y=0;
- S4 – wystąpił cały ciąg, y=1.

Opis działania detektora ciągu bitów w języku VHDL jest następujący:

```
-- L8.4
library IEEE;
use IEEE.std_logic_1164.all;
entity det1010 is
   port (x,reset: in std_logic;
         clk: in std_logic;
         y: out std_logic);
end det1010;
architecture det1010arch of det1010 is
   type stan is (s0, s1, s2, s3, s4);
   signal s : stan;
begin
stany: process (clk, x, reset)
 begin
   if reset='1' then s<=s0;
   elsif rising_edge(clk) then
      case s is
         when s0=>
           if x='0' then s<=s0;
           else s<=s1;
           end if;
         when s1=>
           if x='1' then s<=s1;
           else s<=s2;
           end if;
         when s2=>
           if x='1' then s<=s3;
           else s<=s0;
           end if;
         when s3=>
           if x='0' then s<=s4;
           else s<=s1;
           end if;
         when s4=>
           if x='1' then s<=s1;
           else s<=s0;
           end if;
    end case;
   end if;
end process;
wyjscia : process (s)
    begin
      case s is
         when s0=> y<='0';
         when s1=> y<='0';
         when s2=> y<='0';
         when s3=> y<='0';
         when s4=> y<='1';
```

```
        end case;
    end process;
    end det1010arch;
```

Program testowy L8.5 pozwala na sprawdzenie poprawności funkcjonalnej i wykrycie ewentualnych błędów. Wyniki symulacji funkcjonalnej projektu przedstawia rys. 8.33.



Rys. 8.33 Wynik symulacji detektora ciągu bitów

```
--  L8.5
library IEEE;
use IEEE.std_logic_1164.all;
entity det1010_tb is
end det1010_tb;
architecture TB_ARCHITECTURE of det1010_tb is
component det1010
   port(x,reset, clk : in std_logic;
        y : out std_logic);
end component;
  signal x,reset : std_logic;
  signal clk, y : std_logic;
  signal blad : std_logic;
  type test_vec is record
          x,clk,y,reset :std_logic;
    end record;
    type test_array is array (natural range<>) of
    test_vec;
```

```
 constant test: test_array:=
-- x, clk, y, reset -- tablica wektorow testowych
(('0','0','0','1'),      --s0
 ('0','1','0','0'),      --s0
 ('0','0','0','0'),      --s0
 ('1','1','0','0'),      --s1
 ('1','0','0','0'),      --s1
 ('0','1','0','0'),      --s2
 ('0','0','0','0'),      --s2
 ('1','1','0','0'),      --s3
 ('1','0','0','0'),      --s3
 ('0','1','1','0'),      --s4
 ('0','0','1','0'),      --s4
 ('1','1','0','0'),      --s1
 ('1','0','0','0'),      --s1
 ('0','1','0','0'),      --s2
 ('0','0','0','0'),      --s2
 ('1','1','0','0'),      --s3
 ('1','0','0','0'),      --s3
 ('0','1','1','0'),      --s4
 ('0','0','1','0'),      --s4
 ('0','1','0','1'),      --s0
 ('0','0','0','1'),      --s0
 ('0','1','0','0'),      --s0
 ('0','0','0','0'),      --s0
 ('1','1','0','0'),      --s1
 ('1','0','0','0'),      --s1
 ('1','1','0','0'),      --s1
 ('1','0','0','0'),      --s1
 ('0','1','0','0'),      --s2
 ('0','0','0','0'),      --s2
 ('0','1','0','0'),      --s0
 ('0','0','0','0'),      --s0
 ('1','1','0','0'),      --s1
 ('1','0','0','0'),      --s1
 ('0','1','0','0'),      --s2
 ('0','0','0','0'),      --s2
 ('1','1','0','0'),      --s3
 ('1','0','0','0'),      --s3
 ('1','1','0','0'),      --s1
 ('1','0','0','0'),      --s1
 ('0','1','0','0'),      --s2
 ('0','0','0','0'),      --s2
 ('1','1','0','0'),      --s3
 ('1','0','0','0'),      --s3
 ('0','1','1','0'),      --s4
 ('0','0','1','0'),      --s4
 ('0','1','0','1'),      --s0
 ('0','0','0','1'),      --s0
 ('0','1','0','0'));     --s0
```

```
begin
  UUT : det1010
  port map
    (x => x,
     reset => reset,
     clk => clk,
     y => y );
testuj :process
 variable vec: test_vec;
 begin
  blad<='0';
  for i in test'range loop
     vec:=test(i);
     x<=vec.x;
     reset<=vec.reset;
     clk<=vec.clk;
     wait for 1 ns;
-- sprawdzanie wektorow wyjsciowych
     if y/=vec.y then
         assert false report "Uklad zle dziala";
         blad<='1';
     else report " Uklad dziala poprawnie ";
         blad<='0';
     end if;
     wait for 20 ns;
  end loop;
end process;
end TB_ARCHITECTURE;
configuration TESTBENCH_FOR_det1010 of det1010_tb is
   for TB_ARCHITECTURE
         for UUT : det1010
               use entity work.det1010(det1010arch);
         end for;
   end for;
end TESTBENCH_FOR_det1010;
```

W programie testowym L8.5 zadeklarowano sygnały x, reset, clk, y odpowiadające portom testowanej jednostki det1010 oraz sygnał blad. Zdefiniowano również tablicę wektorów testowych test zawierającą wartości logicznych sygnałów podawanych na wejścia testowanego modelu oraz wartości oczekiwane na jego wyjściu. Wewnątrz procesu testuj umieszczono instrukcję pętli, w której przypisujemy elementy tablicy sygnałom testowanego modelu. Sygnał blad przyjmuje poziom wysoki w przypadku wystąpienia błędnej odpowiedzi układu.

8.6 ZADANIA

1. Zaprojektować i zweryfikować szeregowo-równoległy 4-bitowy rejestr przesuwający.

2. Zaprojektować i zweryfikować konwerter 4-bitowego cyklicznego kodu Graya na naturalny kod dwójkowy

3. Zaprojektować i zweryfikować 8-bitowy sumator równoległy z przeniesieniami równoległymi.

4. Zaprojektować i zweryfikować układ detektora sekwencji bitów 00-01-11-10-00 pojawiających się na wejściu danych X. Układ generuje na wyjściu Y stan 1, gdy wystąpi sekwencja X = 00-01-11-10-00.

5. Zaprojektować i zweryfikować 4-bitowy synchroniczny licznik dwójkowy rewersyjny z przeniesieniami równoległymi.

Do każdego z zadań należy napisać program testowy.

LITERATURA

1. Peter J. Ashenden, *The designer's guide to VHDL*, Second Edition, Academic Press, 2001.
2. K. Skahill, *Język VHDL. Projektowanie programowalnych układów logicznych*, WNT, Warszawa 2001.
3. D. Pellerin, D. Taylor, *VHDL made easy!*, Prentice Hall, 1997.
4. B. Cohen, *VHDL – Coding styles and Methodologies*, Second Edition, Kluwer Academic Publishers, 1999.
5. S. Sjöholm, L. Lindh, *VHDL for Designers*, Prentice Hall, 1997.
6. J. Bergeron, *Writing testbenches – Functional verification of HDL models*, Kluwer Academic Publishers, 2001.
7. M. Zwoliński, *Digital system design with VHDL*, Prentice Hall, 2000.
8. J. O. Hamblen, M. D. Furman, *Rapid prototyping of digital systems*, Kluwer Academic Publisher, 2000.
9. Z. Salcic, *VHDL and FPLDs in digital systems*, Kluwer Academic Publishers, 1998.
10. J. Kalisz, *Podstawy elektroniki cyfrowej*, Wydanie 3, WKŁ, Warszawa 1998.
11. T. Łuba, B. Zbierzchowski, *Komputerowe projektowanie układów cyfrowych*, WKŁ, Warszawa 2000.
12. S. Haykin, *Systemy telekomunikacyjne*, t.2, WKŁ, Warszawa 1998.
13. Z. Jachna, J. Kalisz, T. Orżanowski, J. Pasierbiński, R. Szplet, *Programowalne cyfrowe układy scalone, architektura – oprogramowanie – zadania projektowe*, skrypt WAT, Warszawa 1999.
14. J. Mirkowski, M. Kapustka, Z. Skowroński, A. Biniszkiewicz, *EVITA Interactive VHDL Tutorial rev.2.1 - Active-VHDL™ Series. Book #2.* Aldec, Inc. 1998.
15. Z. Jachna, T. Łuba, *Analiza zasad specyfikacji języka VHDL z punktu widzenia odwzorowania technologicznego*, Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne, nr 12, 1999.
16. T. Łuba, K. Jasiński, B. Zbierzchowski, *Specjalizowane układy cyfrowe w strukturach PLD i FPGA*, WKŁ, Warszawa 1997.
17. J. Pasierbiński, P. Zbysiński, *Układy programowalne w praktyce*, WKŁ, Warszawa 2001.
18. K. Sobolewski, M. Ossysek, *Getting Started Guide - Active-VHDL™ Series. Book #3.* Aldec, Inc. 1998.
19. IEEE, *IEEE Standard VHDL Language Reference Manual, Std 1076-1993*, IEEE 1994.
20. IEEE, *IEEE Standard VHDL Synthesis Package, Std 1076.3-1997*, IEEE 1997.
21. Nota aplikacyjna XAPP338, *Using Xilinx WebPACK and ModelTech ModelSim Xilinx Edition (MXE)*, Xilinx, ver. 2.0, 30 Oct 2000.

DODATEK. PAKIET WAT

```
------------------------------------------------------------
--                                                        --
--     Wojskowa Akademia Techniczna                       --
--     Instytut Systemow Lacznosci                        --
--     Zaklad Techniki Cyfrowej i Mikroprocesorowej       --
--     Nazwa pakietu: wat                                 --
--     Wersja: 2002.1                                     --
--     e-mail: torzanowski@wel.wat.waw.pl                 --
--             zjachna@wel.wat.waw.pl                     --
--                                                        --
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

package wat is

-- Deklaracja funkcji w pakiecie wat

  -- N-wej. bramka AND
  function andN (signal a : in std_logic_vector)
  return std_logic;

  -- N-wej. bramka NAND
  function nandN (signal a : in std_logic_vector)
  return std_logic;

  -- N-wej. bramka OR
  function orN (signal a : in std_logic_vector)
  return std_logic;

  -- N-wej. bramka NOR
  function norN (signal a : in std_logic_vector)
  return std_logic;

-- Deklaracja procedur w pakiecie wat

  -- p. D typu zatrzask z asynch. zerow. poziomem L
  procedure DLnR (signal d, en, r : in std_logic;
      signal q, nq : inout std_logic);

  -- p. D z asynch. zerow. poziomem L
  procedure DnR (signal d, c, r : in std_logic;
      signal q, nq : inout std_logic);

-- p. D asynch. zerow. i ustaw. poziomem L
  procedure DnRnS (signal d, c, r, s : in std_logic;
      signal q, nq : inout std_logic);
```

```
  -- p. T asynch. zerow. poziomem L
  procedure TnR (signal t, c, r : in std_logic;
      signal q, nq : inout std_logic);

  -- p. SR wyzw. zbocz. narast. asynch. zerow.
  procedure SRnCl (signal s, r, c, cl : in std_logic;
      signal q, nq : inout std_logic);

  -- p. JK wyzw. zbocz. narast. asynch. zerow. poziomem L
  procedure JKnR (signal J, K, C, R : in std_logic;
        signal Q, nQ : inout std_logic);

  -- generator liczb pseudolosowych
  procedure random(variable liczba  : inout integer;
                   constant modul : in integer);

-- Deklaracja komponentow

  -- konwerter kodu HEX/7-segment. (wspolna anoda)
  component hex7a
    port(i : in std_logic_vector(3 downto 0);
         s : out std_logic_vector(7 downto 1));
  end component;

  -- N-bitowy synchroniczny, rewersyjny licznik dwojkowy,
  -- asynch. zerowanie
  component cntnbrev
    generic (N: positive := 4); -- np. liczba bitow N = 4
    port (c, r, encnt, dir : in std_logic;
          q : out std_logic_vector(N-1 downto 0));
  end component;

  -- 4-bitowy sumator dwojkowy
  component sum4bin
    port(a: in std_logic_vector(3 downto 0);
      b: in std_logic_vector(3 downto 0);
      ci: in std_logic;
      s: out std_logic_vector(3 downto 0);
      co: out std_logic);
  end component;

  -- uklad korekcji wyniku w sumatorze BCD
  component ksbcd
    port( we: in std_logic_vector(3 downto 0);
          wy: out std_logic);
  end component;

  -- sumator dziesietny
  component sum10
  port(a: in std_logic_vector(3 downto 0);
    b: in std_logic_vector(3 downto 0);
```

```
    ci: in std_logic;
    s: out std_logic_vector(3 downto 0);
    co: out std_logic);
  end component;

end package wat;

package body wat is

-- Funkcje

  -- N-wejsciowa bramka AND, N = dlugosc wektora wej.
  function andN (signal a : in std_logic_vector)
  return std_logic is
    variable f : std_logic;
  begin
    f := '1';
    for i in a'range loop
      f := f and a(i);
    end loop;
    return f;
  end andN;

  -- N-wejsciowa bramka NAND, N = dlugosc wektora wej.
  function nandN (signal a : in std_logic_vector)
  return std_logic is
    variable f : std_logic;
  begin
    f := '1';
    for i in a'range loop
      f := f and a(i);
    end loop;
    return not f;
  end nandN;

  -- N-wejsciowa bramka OR, N = dlugosc wektora wej.
  function orN (signal a : in std_logic_vector)
  return std_logic is
    variable f : std_logic;
  begin
    f := '0';
    for i in a'range loop
      f := f or a(i);
    end loop;
    return f;
  end orN;

  -- N-wejsciowa bramka NOR, N = dlugosc wektora wej.
  function norN (signal a : in std_logic_vector)
  return std_logic is
    variable f : std_logic;
```

```
  begin
    f := '0';
    for i in a'range loop
      f := f or a(i);
    end loop;
    return not f;
  end norN;

-- Procedury

  -- Przerz. D typu zatrzask (latch), asynch. zer. poz. L
  procedure DLnR (signal D, EN, R : in std_logic;
    signal Q, nQ : inout std_logic) is
  begin
    if R = '0' then
      Q <= '0';
    elsif EN = '1' then
      Q <= D;              -- rownanie p. D
    end if;
      nQ <= not Q;
  end DLnR;

  -- Przerz. D wyzw. zb. narast., asynch. zerow. poz. L
  procedure DnR (signal D, C, R : in std_logic;
    signal Q, nQ : inout std_logic) is
  begin
    if R = '0' then
      Q <= '0';
    elsif (C'event and C = '1') then
      Q <= D;              -- rownanie p. D
    end if;
      nQ <= not Q;
  end DnR;

  -- Prz. D wyzw. zb. narast., asyn. zer. i ust. poz. L
  procedure DnRnS (signal D, C, R, S : in std_logic;
    signal Q, nQ : inout std_logic) is
  begin
    if R = '0' then
      Q <= '0';
    elsif S = '0' then
      Q <= '1';
    elsif (C'event and C = '1') then
      Q <= D;
    end if;
    nQ <= not Q;
  end DnRnS;

  -- Przerz. T wyzw. zb. narast., asynch. zerow. poz. L
  procedure TnR (signal T, C, R : in std_logic;
    signal Q, nQ : inout std_logic) is
```

```
  begin
    if R = '0' then
      Q <= '0';
    elsif (C'event and C = '1') then
      Q <= T xor Q;      -- rownanie p. T
    end if;
    nQ <= not Q;
  end TnR;

  -- Przerz. SR wyzw. zb. narast., asynch. zerow. poz. L
  procedure SRnCl (signal S, R, C, Cl : in std_logic;
    signal Q, nQ : inout std_logic) is
  begin
    if Cl = '0' then
      Q <= '0';
    elsif (C'event and C = '1') then
      Q <= S or (not R and Q);       -- rownanie p. SR
    end if;
    nQ <= not Q;
  end SRnCl;

  -- Przerz. JK wyzw. zb. narast., asynch. zerow. poz. L
  procedure JKnR (signal J, K, C, R : in std_logic;
    signal Q, nQ : inout std_logic) is
  begin
    if R = '0' then
      Q <= '0';
    elsif (C'event and C = '1') then    --rownanie p. JK
      Q <= (J and not Q) or (not K and Q);
    end if;
    nQ <= not Q;
  end JKnR;

  -- generator liczb pseudolosowych
  procedure random(variable liczba: inout integer;
                   constant modul : in    integer) is
    constant mnoznik  : integer := 3877;
    constant przyrost : integer := 29573;
  begin
    liczba :=  (mnoznik * liczba + przyrost) mod modul;
  end random;

end package body wat;
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity hex7a is
  port ( i : in std_logic_vector (3 downto 0);
         s : out std_logic_vector (7 downto 1));
end hex7a;
```

```
architecture hex7a of hex7a is
begin
  -- Numeracja segmentow
  --      s1
  --     ----
  -- s6| s7 |s2
  --     ----
  -- s5|     |s3
  --     ----
  --      s4
  with i select
    s <= "1000000" when "0000", -- 0
         "1111001" when "0001", -- 1
         "0100100" when "0010", -- 2
         "0110000" when "0011", -- 3
         "0011001" when "0100", -- 4
         "0010010" when "0101", -- 5
         "0000010" when "0110", -- 6
         "1111000" when "0111", -- 7
         "0000000" when "1000", -- 8
         "0010000" when "1001", -- 9
         "0001000" when "1010", -- A
         "0000011" when "1011", -- b
         "1000110" when "1100", -- C
         "0100001" when "1101", -- d
         "0000110" when "1110", -- E
         "0001110" when "1111", -- F
         "1100011" when others; -- n
end hex7a;
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity cntnbrev is
  generic (N: positive := 8); -- deklaracja N = 8 bit
  port(C, R, ENCNT, DIR : in std_logic;
       Q : out std_logic_vector(N-1 downto 0));
end cntnbrev;
architecture cnt_rev of cntnbrev is
  signal cnt: std_logic_vector(N-1 downto 0);
begin
  process (C, R, DIR)
  begin
    if R = '0' then  -- zerowanie asynchroniczne
        cnt <= (others => '0');
      elsif rising_edge(C) then
        if ENCNT = '1' then  -- zezwolenie na liczenie
          case DIR is
           -- granice liczenia ustala parametr N = 8
           -- operacja cnt - 1 jest wykonywana
```

```
            -- w kodzie U2
             when '1' => cnt <= cnt + 1;
             when '0' => cnt <= cnt - 1;
             when others => null;
           end case;
         end if; -- ENCNT = '0' pamietanie poprzed. stanu
       end if;
  end process;
  Q <= cnt;
end cnt_rev;
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
-- 4-bitowy sumator dwojkowy
entity sum4bin is port(
    a: in std_logic_vector(3 downto 0);
    b: in std_logic_vector(3 downto 0);
    ci: in std_logic;
    s: out std_logic_vector(3 downto 0);
    co: out std_logic);
end sum4bin;

architecture a1 of sum4bin is
  signal c1,c2,c3: std_logic;
begin
    s(0) <= a(0) xor b(0) xor ci;
      c1 <= (a(0) and b(0)) or (ci and (a(0) xor b(0)));
    s(1) <= a(1) xor b(1) xor c1;
      c2 <= (a(1) and b(1)) or (c1 and (a(1) xor b(1)));
    s(2) <= a(2) xor b(2) xor c2;
      c3 <= (a(2) and b(2)) or (c2 and (a(2) xor b(2)));
    s(3) <= a(3) xor b(3) xor c3;
      co <= (a(3) and b(3)) or (c3 and (a(3) xor b(3)));
end a1;
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
-- uklad korekcji wyniku w sumatorze BCD
entity ksbcd is
  port(
    we: in std_logic_vector(3 downto 0);
    wy: out std_logic);
  end ksbcd;

architecture k1 of ksbcd is
begin
    wy <= we(3) or (we(0) and we(2)) or (we(1) and we(2));
end k1;
-------------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
-- sumator dziesietny
entity sum10 is
  port(
    a: in std_logic_vector(3 downto 0);
    b: in std_logic_vector(3 downto 0);
    ci: in std_logic;
    s: out std_logic_vector(3 downto 0);
    co: out std_logic);
  end sum10;

architecture str10 of sum10 is
  component sum4bin
    port(
      a: in std_logic_vector(3 downto 0);
      b: in std_logic_vector(3 downto 0);
      ci: in std_logic;
      s: out std_logic_vector(3 downto 0);
      co: out std_logic);
  end component;

  component ksbcd
    port(
      we: in std_logic_vector(3 downto 0);
      wy: out std_logic);
  end component;

  signal s1: std_logic_vector (3 downto 0);
  signal co1: std_logic;
  signal b2: std_logic_vector (3 downto 0);
  signal i1: std_logic_vector (3 downto 0);
  signal co4: std_logic;
  signal zero: std_logic;
begin
      zero<='0';
   U1: sum4bin port map (a, b, ci, s1, co1);
      i1 <= co1&s1(3 downto 1);
   U2: ksbcd port map (i1,co4);
      b2 <= '0'&co4&co4&'0';
   U3: sum4bin port map (s1,b2,zero,s,open);
   co <= co4;
end str10;
-------------------------------------------------------------
```

SKOROWIDZ

(słowa kluczowe języka VHDL wyróżniono *kursywą*)

A
ABEL, 78,187,205
abs, 26
after, 14,38
albo, 47
ALU, 158
Analiza
  czasowa, 187
  parametrów dynamicznych, 148
array, 31
Atrybut, 18,33-35
  'ascending, 35
  'event, 18,34
  'high, 34
  'left, 34
  'length, 34
  'low, 34
  'reverse_range, 35
  'right, 34
Atrybuty projektowe, 182,199
attribute, 183,199
Asembler, 189
assert, 49,90,98,101, 106,114,196,201,235, 242,246

B
Biblioteka, 35-37, 169
  robocza, 36
  symboli, 168
  użytkownika, 154
bit, 27,37
bit_vector, 32,99
Blok, 67
boolean, 29

C
case, 46,95,103,104,243, 244
character, 29
Ciało architektury, 10
  pakietu, 35
  procesu, 16
Ciąg pseudolosowy, 197
Cykl *delta*, 14
component, 89,97,101, 104,111,112,241,245
constant, 98,106,113, 241,245
CPLD, 77,81,165
Czas
  delta, 13,19
  propagacji, 89,93197, 204
Częstotliwość
  generatora, 204

D
Debuger, 187
Deklaracja
  architektury, 10
  jednostki, 10
  pakietu, 35
  sygnału, 24
delta, 13,14,19
Detektor ciągu bitów, 242
Dzielnik, 203

E
EDIF, 6,77,168
exit, 48

F
for-loop, 47,90,96,98, 101,106,114,196,201, 225, 235,242,246
file, 195,200
Format, 32
  pliku, 168
FPGA, 77
Funkcja, 50,96
  rozdzielcza, 28,81

G
Generator, 82-83,197, 203,210,252
generate, 71
generic, 162,225,232

H
Hierarchia projektu, 7
Hipoteza, 203
Histogram, 202

I
ieee, 37,81,169
if-then-else, 46,225,233, 235,242, 243,246
Instrukcja
  sekwencyjna, 16,43
    czekania, 17,44
    pętli, 47
    przypisania do zmiennej, 17,24
    testowa, 49
    warunkowa, 46
    wyboru, 46
  współbieżna, 10,13,38
    blokowa, 67
    łączenia, 61
    powielania, 71
    procesu, 16
    przypisania do sygnału, 38
integer, 26,96,100
Interfejs, 174

J
JEDEC, 189
Jednostka projektowa, 10
  nagłówek, 11
JTAG, 83,171

K
Klauzula, 11
  a*fter*, 14,38,
  generic, 11
  port, 11
  unaffected, 41
  use, 36
Komparator, 69
Kompilacja, 89,96,145, 185
Komponent, 61,65
Konfiguracja, 12,65, 242,246
  specyfikacja, 66
  deklaracja, 66
Konwerter kodu, 69,86, 138, 190

L
library, 36, 169
Liczby
  całkowite, 26
  rzeczywiste, 27,200
  pseudolosowe, 197
  ułamkowe, 200
Licznik, 62,71,84,102, 107,151,205,225
  modulo 256, 109
  Johnsona, 109
  pierścieniowy, 110
Lista wrażliwości, 17
loop, 47,90,96,98,101, 106,114

M
Makrokomórka, 92
Metastabilność, 108
mod, 26, 191
Model, 84,86,177
  czasowy, 185,204
Moduł, 10,89,94,105,112
  pamięci RAM, 161
MSB, 32
Multiplekser, 14,67,107, 174
Multiplikator, 116

N
natural, 26,105,113
Nawigator projektu, 167
Nazwa projektu, 167,175
next, 48
null, 47-48
numeric_std, 33,81,95, 97,101,103,109,112

O
Obiekty, 23
open, 61,104
Operatory
  przesunięć, 99
  arytmetyczne, 159
Opis logiczny, 68
Opis mieszany, 180
Opis schematowy, 87 129,142,179
Opóźnienie, 96,204
  inercyjne, 38
  transportowe, 39
Optymalizacja, 184

P
Pakiet, 20,35-37,81,103, 120, 169
  numeric_std, 169
  standard, 169
  std_logic_1164, 37,81, 169,225,240,243
  wat, 251
Parametr
  aktualny, 51
  formalny, 51
PLD, 165
Plik, 25,80
  instalacyjny, 166
  raportu, 185
  wynikowy, 185
Płytka testowa, 81,122
  RAF01_01, 81
  LAT10, 170
Podtypy standardowe, 26,29
Port, 10,91,224
  buffer, 10-12,19
  in, 10-12
  inout, 10-12,19
  out, 10-12
  wejściowy, 23
  wyjściowy, 23
port map, 61,90,98,101
positive, 26,57,232
procedura, 53,60,102
  wywołanie, 55,61
Proces, 16,43,91,174
process, 16,43
Program testowy, 5,74, 89,97,100,105,112,140, 156,195,230,241,244
Programowanie, 93,149, 174,188
Prymityw, 20,62,120
Przebieg, 206
Przełączniki
  programowalne, 189
Przeniesienie, 161
Przepełnienie, 161
Przesuw, 108
Przypisanie
  do sygnału, 12,23,38
    podstawowe, 38
    selektywne, 41
    warunkowe, 40
  do zmiennej, 17,24
Przyporządkowanie, 24
  jawne, 55,64
  końcówek, 144
  niejawne, 55,65
  wyprowadzeń, 183
Pułapka, 187

R
range, 24,31,105,113, 235,241,245
real, 27
record, 31,241,245
rejestr, 57,108,115
rekord, 31
rem, 26, 191
report, 49,90,98,101,106, 114,196
rising_edge, 104,109,110
rozkład
  równomierny, 203

S
Schemat, 87,168,179
Sekwencyjność, 16
severity, 49
signed, 33
Składnia, 10
Sprzężenie
  zwrotne, 204
Stała, 24
standard, 37
std_logic, 27,97,232
std_logic_vector, 37,95, 232
std_logic_1164, 37,95
std.standard, 37,99
std_ulogic, 27,81
string, 32
Sterownik świateł, 211
Styl architektury, 21
  behawioralny, 21,58,68, 191
  przepływowy, 21,60,67
  strukturalny, 21,61,192
Sumator dziesiętny, 192
Sumator/subtraktor, 129
Sygnał, 11,23
  wejściowy, 23
  wewnętrzny, 18,24
  zewnętrzny, 23
Symbol, 86-87,180
Symulacja, 13,73
  graficzna, 73
  czasowa, 7,74,92,146, 186,235
  funkcjonalna, 5,73,89, 133,186,225
  programowa, 74
Synchronizacja, 108
Synchronizator, 109, 152
Synteza, 91,96
System projektowy, 5
  Active-HDL, 217
  IspDesignEXPERT,165
  MAX+plus II, 117
  WebPACK ISE, 77

T
Tablica, 31
Tablica stanów, 42-43
Test, 74,230
  statystyczny, 203
Test bench, 5,74,230
textio, 37,200
time, 30
Transakcja, 13
transport, 38
type, 89,105,113
Typy, 25
  plików, 167
  projektów, 168
  skalarne, 26
    bitowy, 27,37
    boolowski, 29
    fizyczny czasowy, 30

liczby całkowite, 26
  liczby rzeczywiste, 27
  wielowartościowe, 27,37
  własne, 30
  wyliczeniowe, 30
  znakowy, 29
 wektorowo-skalarne, 33
 złożone, 31
  bitwektorowy, 32
  łańcuchowy, 32

U

unaffected, 41
unsigned, 33,81,96,102, 103,109,225
Układy programowalne, ISP, 170
 Lattice, 171
Uruchomienie systemu, 174,220

V

variable, 17,103,109
VHDL, 2,77

W

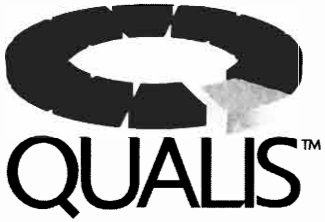
wait, 17,44,90,98,101, 106,114,233,235,242, 246
Wektory testowe, 186, 235,241,245
Weryfikacja projektów, 72, 186
when-else, 15,40,95,108
when others, 46-47,95, 100,103,240
Wiązanie, 65
with-select, 41,67,95, 100,240
work, 36,104
Współbieżność, 13
Wymagania projektowe, 182,218,220
Wyświetlacz, 86,102

X

xor, 161, 198

Z

Zakres, 24,31
 nieokreślony, 31
Zamek, 207
Zasoby
 biblioteczne, 79,167
 logiczne, 170
Zdarzenie, 13,54
Zmienna, 17,24



VHDL QUICK REFERENCE CARD

Revision 2.2

()	Grouping	[ ]	Optional
{}	Repeated	|	Alternative
bold	As is	CAPS	User Identifier
italic	VHDL-1993		

1. LIBRARY UNITS

[{use_clause}]
entity ID **is**
  [**generic** ({ID : TYPEID [:= expr];});]
  [**port** ({ID : **in** | **out** | **inout** TYPEID [:= expr];});]
  [{declaration}]
[**begin**
  {parallel_statement}]
end [***entity***] ENTITYID;

[{use_clause}]
architecture ID **of** ENTITYID **is**
  [{declaration}]
begin
  [{parallel_statement}]
end [***architecture***] ARCHID;

[{use_clause}]
package ID **is**
  [{declaration}]
end [***package***] PACKID;

[{use_clause}]
package body ID **is**
  [{declaration}]
end [***package body***] PACKID;

[{use_clause}]
configuration ID **of** ENTITYID **is**
for ARCHID
  [{block_config | comp_config}]
end for;
end [***configuration***] CONFID;

use_clause::=
  library ID;
  [{**use** LIBID.PKGID[. **all** | DECLID];}]

block_config::=
  for LABELID
    [{block_config | comp_config}]
  end for;

comp_config::=
  for all | LABELID : COMPID
    (**use entity** [LIBID.]ENTITYID [(ARCHID)]
      [[**generic map** ({GENID => expr ,})]
       port map ({PORTID => SIGID | *expr* ,})];
    [**for** ARCHID
      [{block_config | comp_config}]
    end for;]
    end for;) |
    (**use configuration** [LIBID.]CONFID
      [[**generic map** ({GENID => expr ,})]
      port map ({PORTID => SIGID | *expr*,})];)
  end for;

2. DECLARATIONS

2.1. TYPE DECLARATIONS

type ID **is** ({ID,});

type ID **is range** number **downto** | **to** number;

type ID **is array** ({range | TYPEID ,}) **of** TYPEID;

type ID **is record**
  {ID : TYPEID;}
end record;

type ID **is access** TYPEID;

type ID **is file of** TYPEID;

subtype ID **is** SCALARTYPID **range** range;

subtype ID **is** ARRAYTYPID({range,});

subtype ID **is** RESOLVFCTID TYPEID;

range ::=
  (integer | ENUMID **to** | **downto** integer | ENUMID) |
  (OBJID'[**reverse_**]**range**) | (TYPEID **range <>**)

2.2. OTHER DECLARATIONS

constant ID : TYPEID := expr;

[***shared***] **variable** ID : TYPEID [:= expr];

signal ID : TYPEID [:= expr];

file ID : TYPEID (**is in** | **out** string;) |
  (***open read_mode*** | ***write_mode*** |
   append_mode is *string;*)

alias ID : TYPEID **is** OBJID;

attribute ID : TYPEID;

attribute ATTRID **of** OBJID | **others** | **all** : class **is** expr;

class ::=
  entity | **architecture** | **configuration** |
  procedure | **function** | **package** | **type** |
  subtype | **constant** | **signal** | **variable** |
  component | **label**

component ID [***is***]
  [**generic** ({ID : TYPEID [:= expr];});]
  [**port** ({ID : **in** | **out** | **inout** TYPEID [:= expr];});]
end component [*COMPID*];

[***impure*** | ***pure***] **function** ID
  [({[**constant** | **variable** | **signal** | ***file***] ID :
    [**in**]TYPEID [:= expr];})]
  return TYPEID [**is**
begin
  {sequential_statement}
end [***function***] ID];

procedure ID[({[**constant** | **variable** | **signal**] ID :
    in | **out** | **inout** TYPEID [:= expr];})]
[**is begin**
  [{sequential_statement}]
end [***procedure***] ID];

for LABELID | **others** | **all** : COMPID **use**
  (**entity** [LIBID.]ENTITYID [(ARCHID)]) |
  (**configuration** [LIBID.]CONFID)
    [[**generic map** ({GENID => expr,})]
     port map ({PORTID => SIGID | *expr*,})];

3. EXPRESSIONS

expression ::=
  (relation **and** relation) | (relation **nand** relation) |
  (relation **or** relation) | (relation **nor** relation) |
  (relation **xor** relation) | (*relation* ***xnor*** *relation*)

relation ::= shexpr [relop shexpr]

shexpr ::= sexpr [*shop sexpr*]

sexpr ::= [**+**|**-**] term {addop term}

term ::= factor {mulop factor}

factor ::=
  (prim [****** prim]) | (**abs** prim) | (**not** prim)

prim ::=
  literal | OBJID | OBJID'ATTRID | OBJID({expr,})
  | OBJID(range) | ({[choice [{| choice}] **=>**] expr,})
  | FCTID({[PARID **=>**] expr,}) | TYPEID'(expr) |
  TYPEID(expr) | **new** TYPEID['(expr)] | **(** expr **)**

choice ::= sexpr | range | RECFID | **others**

3.1. OPERATORS, INCREASING PRECEDENCE

logop	**and** | **or** | **xor** | **nand** | **nor** | ***xnor***
relop	**=** | **/=** | **<** | **<=** | **>** | **>=**
shop	***sll*** | ***srl*** | ***sla*** | ***sra*** | ***rol*** | ***ror***
addop	**+** | **-** | **&**
mulop	***** | **/** | **mod** | **rem**
miscop	****** | **abs** | **not**

© 1995-2000 Qualis Design Corporation

© 1995-2000 Qualis Design Corporation

1995-2000 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

See reverse side for additional information.

4. Sequential Statements

wait [**on** {SIGID,}] [**until** expr] [**for** time];

assert expr
  [**report** string]
  [**severity note** | **warning** | **error** | **failure**];

report *string*
  [severity note | ***warning*** | ***error*** | ***failure];***

SIGID **<=** [**transport**] | [[***reject*** *TIME*] ***inertial***]
    {expr [**after** time],};

VARID **:=** expr;

PROCEDUREID[({[PARID **=>**] expr,})];

[*LABEL:*] **if** expr **then**
  {sequential_statement}
[{**elsif** expr **then**
  {sequential_statement}}]
[**else**
  {sequential_statement}]
end if [*LABEL*];

[*LABEL:*] **case** expr **is**
{**when** choice [{| choice}] **=>**
  {sequential_statement}}
end case [*LABEL*];

[LABEL:] [**while** expr] **loop**
  {sequential_statement}
end loop [LABEL];

[LABEL:] **for** ID **in** range **loop**
  {sequential_statement}
end loop [LABEL];

next [LOOPLBL] [**when** expr];

exit [LOOPLBL] [**when** expr];

return [expression];

null;

5. Parallel Statements

LABEL: **block** [***is***]
  [**generic** ({ID : TYPEID;});
    [**generic map** ({[GENID **=>**] expr,});]]
  [**port** ({ID : **in** | **out** | **inout** TYPEID });
    [**port map** ({[PORTID **=>**] SIGID | *expr,*})];]
  [{declaration}]
begin
  [{parallel_statement}]
end block [LABEL];

[LABEL:] [***postponed***] **process** [({SIGID,})]
  [{declaration}]
begin
  [{sequential_statement}]
end [***postponed***] **process** [LABEL];

[LBL:] [***postponed***] PROCID({[PARID **=>**] expr,});

© 1995-2000 Qualis Design Corporation

[LABEL:] [***postponed***] **assert** expr
  [**report** string]
  [**severity note** | **warning** | **error** | **failure**];

[LABEL:] [***postponed***] SIGID **<=**
  [**transport**] | [[***reject*** *TIME*] ***inertial***]
  [{{expr [**after** TIME,]} | ***unaffected*** **when** expr **else**}]
  {expr [**after** TIME,]} | ***unaffected***;

[LABEL:] [***postponed***] **with** expr **select**
  SIGID **<=** [**transport**] | [[***reject*** *TIME*] ***inertial***]
    {{expr [**after** TIME,]} | ***unaffected***
      when choice [{| choice}]};

LABEL: COMPID
    [[**generic map** ({GENID **=>** expr,})]
     port map ({[PORTID **=>**] SIGID | *expr,*})];

LABEL: ***entity*** *[LIBID.]ENTITYID [(ARCHID)]*
    *[[**generic map** ({GENID **=>** expr,})]*
     port map *({[PORTID **=>**] SIGID | expr,})];*

LABEL: ***configuration*** *[LIBID.]CONFID*
    *[[**generic map** ({GENID **=>** expr,})]*
     port map *({[PORTID **=>**] SIGID | expr,})];*

LABEL: **if** expr **generate**
  [{parallel_statement}]
end generate [LABEL];

LABEL: **for** ID **in** range **generate**
  [{parallel_statement}]
end generate [LABEL];

6. Predefined Attributes

TYPID'**base**	Base type
TYPID'**left**	Left bound value
TYPID'**right**	Right-bound value
TYPID'**high**	Upper-bound value
TYPID'**low**	Lower-bound value
TYPID'**pos**(expr)	Position within type
TYPID'**val**(expr)	Value at position
TYPID'**succ**(expr)	Next value in order
TYPID'**pred**(expr)	Previous value in order
TYPID'**leftof**(expr)	Value to the left in order
TYPID'**rightof**(expr)	Value to the right in order
*TYPID'**ascending***	*Ascending type predicate*
*TYPID'**image**(expr)*	*String image of value*
*TYPID'**value**(string)*	*Value of string image*
ARYID'**left**[(expr)]	Left-bound of [nth] index
ARYID'**right**[(expr)]	Right-bound of [nth] index
ARYID'**high**[(expr)]	Upper-bound of [nth] index
ARYID'**low**[(expr)]	Lower-bound of [nth] index
ARYID'**range**[(expr)]	'left down/to 'right
ARYID'**reverse_range**[(expr)]	'right down/to 'left
ARYID'**length**[(expr)]	Length of [nth] dimension
*ARYID'**ascending**[(expr)]*	*'right >= 'left ?*
SIGID'**delayed**[(TIME)]	Delayed copy of signal
SIGID'**stable**[(TIME)]	Signals event on signal
SIGID'**quiet**[(TIME)]	Signals activity on signal
SIGID'**transaction**	Toggles if signal active

© 1995-2000 Qualis Design Corporation

SIGID'**event**	Event on signal ?
SIGID'**active**	Activity on signal ?
SIGID'**last_event**	Time since last event
SIGID'**last_active**	Time since last active
SIGID'**last_value**	Value before last event
*SIGID'**driving***	*Active driver predicate*
*SIGID'**driving_value***	*Value of driver*
*OBJID'**simple_name***	*Name of object*
*OBJID'**instance_name***	*Pathname of object*
*OBJID'**path_name***	*Pathname to object*

7. Predefined Types

BOOLEAN	True or false
INTEGER	32 or 64 bits
NATURAL	Integers >= 0
POSITIVE	Integers > 0
REAL	Floating-point
BIT	'0', '1'
BIT_VECTOR(NATURAL)	Array of bits
CHARACTER	7-bit ASCII
STRING(POSITIVE)	Array of characters
TIME	hr, min, sec, ms, us, ns, ps, fs
DELAY_LENGTH	*Time >= 0*

8. Predefined Functions

NOW Returns current simulation time

DEALLOCATE(ACCESSTYPOBJ)
Deallocate dynamic object

FILE_OPEN*([status], FILEID, string, mode)*
Open file

FILE_CLOSE*(FILEID)* *Close file*

9. Lexical Elements

Identifier ::= letter { [underline] alphanumeric }

decimal literal ::=integer [. integer] [**E[+|-]** integer]

based literal ::=
  integer **#** hexint [. hexint] **#** [**E[+|-]** integer]

bit string literal ::= **B|O|X** " hexint "

comment ::= -- comment text

© 1995-2000 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

Qualis Design Corporation

Elite Training / Consulting in Reuse and Methodology

Phone: +1-503-670-7200 FAX: +1-503-670-0809
E-mail: info@qualis.com com Web:www.qualis.com

Also available: 1164 Packages Quick Reference Card
Verilog HDL Quick Reference Card



1164 PACKAGES QUICK REFERENCE CARD

Revision 2.2

()	Grouping	[ ]	Optional
{}	Repeated	|	Alternative
bold	As is	CAPS	User Identifier
italic	VHDL-93	c	commutative

b ::= BIT
bv ::= BIT_VECTOR
u/l ::= STD_ULOGIC/STD_LOGIC
uv ::= STD_ULOGIC_VECTOR
lv ::= STD_LOGIC_VECTOR
un ::= UNSIGNED
sg ::= SIGNED
in ::= INTEGER
na ::= NATURAL
sm ::= SMALL_INT (subtype INTEGER range 0 to 1)

1.IEEE's STD_LOGIC_1164

1.1 Logic Values

'U'	Uninitialized
'X'/'W'	Strong/Weak unknown
'0'/'L'	Strong/Weak 0
'1'/'H'	Strong/Weak 1
'Z'	High Impedance
'-'	Don't care

1.2 Predefined Types

STD_ULOGIC Base type
Subtypes:
STD_LOGIC Resolved STD_ULOGIC
X01 Resolved X, 0 & 1
X01Z Resolved X, 0, 1 & Z
UX01 Resolved U, X, 0 & 1
UX01Z Resolved U, X, 0, 1 & Z
STD_ULOGIC_VECTOR(na **to** I **downto** na) Array of STD_ULOGIC
STD_LOGIC_VECTOR(na **to** I **downto** na) Array of STD_LOGIC

© 1995-2000 Qualis Design Corporation

1.3 Overloaded Operators

Description	Left	Operator	Right
bitwise-and	u/l,uv,lv	**and, nand**	u/l,uv,lv
bitwise-or	u/l,uv,lv	**or, nor**	u/l,uv,lv
bitwise-xor	u/l,uv,lv	**xor, xnor**	u/l,uv,lv
bitwise-not		**not**	u/l,uv,lv

1.4 Conversion Functions

From	To	Function
u/l	b	**TO_BIT**(from[, xmap])
uv,lv	bv	**TO_BITVECTOR**(from[, xmap])
b	u/l	**TO_STDULOGIC**(from)
bv,uv	lv	**TO_STDLOGICVECTOR**(from)
bv,lv	uv	**TO_STDULOGICVECTOR**(from)

2.IEEE's NUMERIC_STD

2.1 Predefined Types

UNSIGNED(na **to** I **downto** na) Array of STD_LOGIC
SIGNED(na **to** I **downto** na) Array of STD_LOGIC

2.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg
	-	sg	sg
un	**+,-,*,/,rem,mod**	un	un
sg	**+,-,*,/,rem,mod**	sg	sg
un	**+,-,*,/,rem,mod** c	na	un
sg	**+,-,*,/,rem,mod** c	in	sg
un	**<,>,<=,>=,=,/=**	un	bool
sg	**<,>,<=,>=,=,/=**	sg	bool
un	**<,>,<=,>=,=,/=** c	na	bool
sg	**<,>,<=,>=,=,/=** c	in	bool

2.3 Predefined Functions

SHIFT_LEFT(un, na) un
SHIFT_RIGHT(un, na) un
SHIFT_LEFT(sg, na) sg
SHIFT_RIGHT(sg, na) sg
ROTATE_LEFT(un, na) un
ROTATE_RIGHT(un, na) un
ROTATE_LEFT(sg, na) sg
ROTATE_RIGHT(sg, na) sg
RESIZE(sg, na) sg
RESIZE(un, na) un
STD_MATCH(u/l, u/l) bool
STD_MATCH(uv, uv) bool
STD_MATCH(lv, lv) bool
STD_MATCH(un, un) bool
STD_MATCH(sg, sg) bool

© 1995-2000 Qualis Design Corporation

2.4 Conversion Functions

From	To	Function
un,lv	sg	**SIGNED**(from)
sg,lv	un	**UNSIGNED**(from)
un,sg	lv	**STD_LOGIC_VECTOR**(from)
un,sg	in	**TO_INTEGER**(from)
na	un	**TO_UNSIGNED**(from, size)
in	sg	**TO_SIGNED**(from, size)

3.IEEE's NUMERIC_BIT

3.1 Predefined Types

UNSIGNED(na **to** I **downto** na) Array of BIT
SIGNED(na **to** I **downto** na) Array of BIT

3.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg
	-	sg	sg
un	**+,-,*,/,rem,mod**	un	un
sg	**+,-,*,/,rem,mod**	sg	sg
un	**+,-,*,/,rem,mod** c	na	un
sg	**+,-,*,/,rem,mod** c	in	sg
un	**<,>,<=,>=,=,/=**	un	bool
sg	**<,>,<=,>=,=,/=**	sg	bool
un	**<,>,<=,>=,=,/=** c	na	bool
sg	**<,>,<=,>=,=,/=** c	in	bool

3.3 Predefined Functions

SHIFT_LEFT(un, na) un
SHIFT_RIGHT(un, na) un
SHIFT_LEFT(sg, na) sg
SHIFT_RIGHT(sg, na) sg
ROTATE_LEFT(un, na) un
ROTATE_RIGHT(un, na) un
ROTATE_LEFT(sg, na) sg
ROTATE_RIGHT(sg, na) sg
RESIZE(sg, na) sg
RESIZE(un, na) un

3.4 Conversion Functions

From	To	Function
un,bv	sg	**SIGNED**(from)
sg,bv	un	**UNSIGNED**(from)
un,sg	bv	**BIT_VECTOR**(from)
un,sg	in	**TO_INTEGER**(from)
na	un	**TO_UNSIGNED**(from)
in	sg	**TO_SIGNED**(from)

© 1995 - 2000 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.
See reverse side for additional information.

49,23

4. Synopsys' STD_LOGIC_ARITH

4.1 Predefined Types

UNSIGNED(na **to** I **downto** na)	Array of STD_LOGIC
SIGNED(na **to** I **downto** na)	Array of STD_LOGIC
SMALL_INT	Integer subtype, 0 or 1

4.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg,lv
	-	sg	sg,lv
un	+,-,*	un	un,lv
sg	+,-,*	sg	sg,lv
sg	+,-,*	un	sg,lv
un	+,- c	in	un,lv
sg	+,- c	in	sg,lv
un	+,- c	u/l	un,lv
sg	+,- c	u/l	sg,lv
un	<,>,<=,>=,=,/=	un	bool
sg	<,>,<=,>=,=,/=	sg	bool
un	<,>,<=,>=,=,/= c	in	bool
sg	<,>,<=,>=,=,/= c	in	bool

4.3 Predefined Functions

SHL(un, un)	un	**SHR**(un, un)	un
SHL(sg, un)	sg	**SHR**(sg, un)	sg
EXT(lv, in)	lv	zero-extend	
SEXT(lv, in)	lv	sign-extend	

4.4 Conversion Functions

From	To	Function
un,lv	sg	**SIGNED**(from)
sg,lv	un	**UNSIGNED**(from)
sg,un	lv	**STD_LOGIC_VECTOR**(from)
un,sg	in	**CONV_INTEGER**(from)
in,un,sg,u	un	**CONV_UNSIGNED**(from, size)
in,un,sg,u	sg	**CONV_SIGNED**(from, size)
in,un,sg,u	lv	**CONV_STD_LOGIC_VECTOR**(from, size)

5. Synopsys' STD_LOGIC_UNSIGNED

5.1 Overloaded Operators

Left	Op	Right	Return
	+	lv	lv
lv	+,-,*	lv	lv
lv	+,- c	in	lv
lv	+,- c	u/l	lv
lv	<,>,<=,>=,=,/=	lv	bool
lv	<,>,<=,>=,=,/= c	in	bool

5.2 Conversion Functions

From	To	Function
lv	in	**CONV_INTEGER**(from)

© 1995-2000 Qualis Design Corporation

6. Synopsys' STD_LOGIC_SIGNED

6.1 Overloaded Operators

Left	Op	Right	Return
	abs	lv	lv
	+,-	lv	lv
lv	+,-,*	lv	lv
lv	+,- c	in	lv
lv	+,- c	u/l	lv
lv	<,>,<=,>=,=,/=	lv	bool
lv	<,>,<=,>=,=,/= c	in	bool

6.2 Conversion Functions

From	To	Function
lv	in	**CONV_INTEGER**(from)

7. Synopsys' STD_LOGIC_MISC

7.1 Predefined Functions

AND_REDUCE(lv I uv)	u/l
[X]OR_REDUCE(lv I uv)	u/l
[N]AND_REDUCE(lv I uv)	UX01
OR_REDUCE(lv I uv)	UX01
NOR_REDUCE(lv I uv)	UX01
XOR_REDUCE(lv I uv)	UX01
XNOR_REDUCE(lv I uv)	UX01

8. Exemplar's STD_LOGIC_ARITH

8.1 Overloaded Operators

Left	Op	Right	Return
	+,-,*,	u/l	u/l
	abs	u/l	u/l

8.2 Predefined Functions

sl(u/l, in)	u/l
sl2(u/l, in)	u/l
sr(u/l, in)	u/l
sr2(u/l, in)	u/l
add(u/l)	u/l
add2(u/l)	u/l
sub(u/l)	u/l
sub2(u/l)	u/l
mult(u/l)	u/l
mult2(u/l)	u/l
extend(u/l, in)	u/l
extend2(u/l, in)	u/l
comp2(u/l)	u/l

8.3 Conversion Functions

From	To	Function
bool	uv	**bool2elb**
uv	bool	**elb2bool**
u/l	na	**evec2int**
in	u/l	**int2evec** (size)
uv	na	**elb2int**

© 1995-2000 Qualis Design Corporation

9. Mentor's STD_LOGIC_ARITH

9.1 Predefined Types

UNSIGNED(na **to** I **downto** na)	Array of STD_LOGIC
SIGNED(na **to** I **downto** na)	Array of STD_LOGIC

9.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg
	-	sg	sg
u/l	+,-	u/l	u/l
uv	+,-,*,/,**mod**,**rem**,**	uv	uv
lv	+,-,*,/,**mod**,**rem**,**	lv	lv
un	+,-,*,/,**mod**,**rem**,**	un	un
sg	+,-,*,/,**mod**,**rem**,**	sg	sg
un	<,>,<=,>=,=,/=	un	bool
sg	<,>,<=,>=,=,/=	sg	bool
	not	un	un
	not	sg	sg
un	**and**,**nand**,**or**,**nor**,**xor**	un	un
sg	**and**,**nand**,**or**,**nor**,**xor**,***xnor***	sg	sg
uv	***sla***,***sra***,***sll***,***srl***,***rol***,***ror***	uv	uv
lv	***sla***,***sra***,***sll***,***srl***,***rol***,***ror***	lv	lv
un	***sla***,***sra***,***sll***,***srl***,***rol***,***ror***	un	un
sg	***sla***,***sra***,***sll***,***srl***,***rol***,***ror***	sg	sg

9.3 Predefined Functions

ZERO_EXTEND(uv I lv I un, na)	same
ZERO_EXTEND(u/l, na)	lv
SIGN_EXTEND(sg, na)	sg
AND_REDUCE(uv I lv I un I sg)	u/l
OR_REDUCE(uv I lv I un I sg)	u/l
XOR_REDUCE(uv I lv I un I sg)	u/l

9.4 Conversion Functions

From	To	Function
u/l,uv,lv,un,sg	in	**TO_INTEGER**(from)
u/l,uv,lv,un,sg	in	**CONV_INTEGER**(from)
bool	u/l	**TO_STDLOGIC**(from)
na	un	**TO_UNSIGNED**(from,size)
na	un	**CONV_UNSIGNED**(from,size)
in	sg	**TO_SIGNED**(from,size)
in	sg	**CONV_SIGNED**(from,size)
na	lv	**TO_STDLOGICVECTOR**(from,size)
na	uv	**TO_STDULOGICVECTOR**(from,size)

© 1995 - 2000 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

Qualis Design Corporation

Elite Training and Consulting in Reuse and Methodology

Phone: +1.503.670.7200 FAX: +1.503.670.0809
Email: info@qualis.com Web: http://www.qualis.com

Also available: VHDL Quick Reference Card
Verilog Quick Reference Card