

Podstawy projektowania systemów wbudowanych

Valery Salauyou
Irena Bułatowa
Tomasz Grześ
Witali Bułatow



Politechnika
Białostocka

Valery Salauyou, Irena Bułatowa,
Tomasz Grześ, Witali Bułatow

PODSTAWY PROJEKTOWANIA SYSTEMÓW WBUDOWANYCH



OFICyna WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2024

Recenzent:
prof. dr hab. inż. Ołeksandr Barkalov

Redaktor naukowy dyscypliny informatyka techniczna i telekomunikacja:
prof. dr hab. Jarosław Stepaniuk

Tabele i rysunki:
prof. Valery Salauyou

Korekta językowa:
Edyta Chrzanowska

Skład i okładka:
Marcin Dominów

Zdjęcie na okładce: franganillo
<https://pixabay.com/pl/illustrations/ai-generowane-p%C5%82ytka-%C5%BCeton-edyt%C5%82or-8505529/>

© Copyright by Politechnika Białostocka, Białystok 2024

ISBN 978-83-68077-32-2 (e-Book)
DOI: 10.24427/978-83-68077-32-2



Publikacja jest udostępniona na licencji
Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 4.0
(CC BY-NC-ND 4.0).

Pełną treść licencji udostępniono na stronie
creativecommons.org/licenses/by-nc-nd/4.0/legalcode.pl.
Publikacja jest dostępna w Internecie na stronie Oficyny Wydawniczej PB.

Spis treści

Wstęp	11
1. Systemy wbudowane i układy scalone z logiką programowalną	16
1.1. Systemy wbudowane	16
1.1.1. Pojęcie systemu wbudowanego	16
1.1.2. Funkcje systemów wbudowanych.....	17
1.1.3. Podstawowe komponenty systemów wbudowanych.....	18
1.1.4. Urządzenia peryferyjne.....	23
1.1.5. Niezawodne działanie systemów wbudowanych	23
1.1.6. Metodologia projektowania systemów wbudowanych opartych na FPGA	25
1.2. Programowalne układy logiczne	27
1.2.1. Klasyfikacja układów programowalnych.....	28
1.2.2. Układy PLD	28
1.2.3. Złożone układy programowalne CPLD	31
1.2.4. Układy FPGA	31
1.2.5. Układy SoC.....	33
1.2.6. Wykorzystanie układów programowalnych do budowy systemów wbudowanych	34
1.3. Podsumowanie rozdziału	35
2. Język opisu sprzętu Verilog	39
2.1. Moduły.....	39
2.1.1. Deklarowanie modułów.....	39
2.1.2. Elementy modułów.....	40
2.1.3. Behawioralny oraz strukturalny styl opisu projektu	41
2.1.4. Przykład projektowania jednobitowego sumatora	41
2.1.5. Deklaracja portów.....	45
2.1.6. Instancje modułów	46
2.1.7. Przekazywanie sygnałów do instancji modułów	47
2.1.8. Parametry modułu.....	49
2.1.9. Niejawne przekazywanie parametrów	51
2.1.10. Tablice instancji modułów.....	51
2.1.11. Równoległość języka Verilog.....	53

2.2.	Podstawowe elementy języka Verilog	54
2.2.1.	Identyfikatory	54
2.2.2.	Znaki białe	54
2.2.3.	Komentarze.....	54
2.2.4.	Sygnały, sieci, sterowniki.....	55
2.2.5.	Wartości logiczne	55
2.2.6.	Liczby całkowite	55
2.2.7.	Liczby rzeczywiste.....	57
2.2.8.	Prymitywy języka Verilog.....	57
2.3.	Typy danych	58
2.3.1.	Dwie klasy typów danych	58
2.3.2.	Sieciowe typy danych.....	59
2.3.3.	Znaczenie sygnałów sieci.....	60
2.3.4.	Zmienne typy danych.....	62
2.3.5.	Parametry.....	64
2.3.6.	Parametry lokalne.....	64
2.3.7.	Zmienne generacji	65
2.3.8.	Łańcuchy znaków.....	65
2.3.9.	Skalary, wektory i tablice.....	65
2.3.10.	Wybór bitów i pól bitowych	66
2.3.11.	Wybór elementów tablicy i pól bitowych elementów tablicy.....	67
2.3.12.	Deklaracja pamięci	68
2.4.	Wyrażenia.....	68
2.5.	Operatory języka Verilog.....	69
2.5.1.	Operatory bitowe	69
2.5.2.	Operatory redukcji	70
2.5.3.	Operatory logiczne	73
2.5.4.	Operatory relacji	74
2.5.5.	Operatory identyczności.....	75
2.5.6.	Operatory arytmetyczne	75
2.5.7.	Operatory różne.....	76
2.5.8.	Wykonywanie operacji.....	77
2.5.9.	Priorytety operatorów	78
2.5.10.	Rozmiar wyrażeń bitowych	78
2.6.	Instrukcja przypisania ciągłego assign.....	79
2.6.1.	Przypisania w języku Verilog.....	79
2.6.2.	Formaty instrukcji przypisania ciągłego.....	79
2.6.3.	Wykorzystanie instrukcji przypisania ciągłego.....	80
2.7.	Instrukcje i bloki proceduralne	83
2.7.1.	Instrukcje proceduralne initial i always, bloki proceduralne.....	83
2.7.2.	Instrukcje begin-end i fork-join	83

2.7.3.	Nazwane bloki proceduralne.....	84
2.7.4.	Format bloków proceduralnych.....	84
2.8.	Zarządzanie czasem	86
2.8.1.	Instrukcja czułości @.....	86
2.8.2.	Lista czułości	87
2.8.3.	Lista czułości w układach kombinacyjnych	88
2.8.4.	Lista czułości w układach sekwencyjnych	89
2.9.	Instrukcje przypisania.....	90
2.9.1.	Instrukcja przypisania blokującego „=”	90
2.9.2.	Instrukcja przypisania nieblokującego „<=”	91
2.9.3.	Specyfika syntezy instrukcji blokujących i nieblokujących.....	93
2.9.4.	Wykorzystanie przypisania blokującego i nieblokującego w jednym bloku proceduralnym.....	95
2.10.	Instrukcje programowania proceduralnego	97
2.10.1.	Instrukcja if-else	97
2.10.2.	Instrukcja case.....	101
2.10.3.	Instrukcje casez i casex.....	104
2.10.4.	Atrybuty full_case i parallel_case	106
2.10.5.	Atrybut full_case.....	106
2.10.6.	Atrybut parallel_case.....	107
2.10.7.	Instrukcja for	111
2.10.8.	Instrukcja while.....	112
2.10.9.	Instrukcja repeat	113
2.10.10.	Instrukcja forever	114
2.10.11.	Instrukcja disable.....	114
2.11.	Bloki generacyjne	115
2.12.	Zadania i funkcje.....	118
2.12.1.	Automatyczne i statyczne zadania i funkcje.....	119
2.12.2.	Zadania.....	119
2.12.3.	Funkcje.....	121
2.12.4.	Funkcje stałe	123
2.13.	Zadania i funkcje systemowe	124
2.14.	Dyrektywy kompilatora.....	125
2.15.	Podsumowanie rozdziału	125
3.	System projektowania Quartus Prime	129
3.1.	Podstawowe etapy realizacji projektów na układach FPGA	129
3.2.	Utworzenie nowego projektu w systemie Quartus	130
3.3.	Wprowadzenie kodu projektu w języku Verilog	136
3.4.	Szablony języka Verilog	137
3.4.1.	Gotowe projekty.....	138
3.4.2.	Konstrukcje języka Verilog.....	139

3.4.3.	Elementy logiczne	139
3.4.4.	Atrybuty syntezy	140
3.4.5.	Prymitywy firmy Altera	141
3.5.	Dodawanie plików do projektu	142
3.6.	Kompilacja projektu	144
3.7.	Przegląd i analiza wyników syntezy	145
3.8.	Komunikaty o błędach kompilacji	147
3.9.	Przyporządkowanie wyprowadzeń	149
3.9.1.	Przyporządkowanie wyprowadzeń za pomocą edytora przypisań	149
3.9.2.	Przyporządkowanie wyprowadzeń za pomocą pliku ustawień	152
3.9.3.	Import i eksport pliku ustawień	153
3.9.4.	Plik ustawień dla płytki prototypowej DE1-SoC	154
3.9.5.	Wykorzystanie nazw sygnałów predefiniowanych w pliku ustawień	155
3.10.	Symulacja projektu	157
3.10.1.	Generowanie wektorów testowych	158
3.10.2.	Grupowanie i rozgrupowanie sygnałów	161
3.10.3.	Zmiana porządku bitowego sygnałów w szynie	162
3.10.4.	Symulacja funkcjonalna	163
3.10.5.	Symulacja czasowa	164
3.11.	Programowanie i konfigurowanie układu FPGA	164
3.12.	Testowanie projektu	167
3.13.	Podsumowanie rozdziału	167
4.	Projektowanie układów kombinacyjnych	170
4.1.	Elementy języka Verilog wykorzystywane w projektowaniu układów kombinacyjnych	170
4.1.1.	Prymitywy	170
4.1.2.	Prymitywy użytkownika	171
4.1.3.	Instrukcja ciągłego przypisania assign	171
4.1.4.	Instrukcje proceduralne	172
4.1.5.	Funkcje i zadania	176
4.2.	Metodologia projektowania układów kombinacyjnych	176
4.2.1.	Opis funkcjonowania układu kombinacyjnego	176
4.2.2.	Minimalizacja funkcji logicznych	179
4.2.3.	Opis układu kombinacyjnego za pomocą systemu funkcji logicznych	179
4.2.4.	Opis układu kombinacyjnego w postaci algorytmu	179
4.2.5.	Opis układu kombinacyjnego w języku Verilog	181

4.3.	Zapobieganie wprowadzeniu zatrząskó na wyjściach układów kombinacyjnych	181
4.4.	Podsumowanie rozdziału	182
5.	Projektowanie standardowych kombinacyjnych bloków funkcjonalnych.....	184
5.1.	Bufory trójstanowe	184
5.2.	Multipleksery	185
5.2.1.	Sposoby opisu multiplexerów	186
5.2.2.	Multipleksery grupowe	189
5.2.3.	Zwiększenie liczby wejść multiplexera	190
5.3.	Demultipleksery	196
5.4.	Kodery i kodery priorytetowe	204
5.4.1.	Kodery	204
5.4.2.	Kodery priorytetowe	212
5.5.	Dekodery	217
5.5.1.	Sposoby opisu dekodérów	218
5.5.2.	Dekoder z wejściem zezwalającym.....	223
5.5.3.	Niepełne dekodery.....	224
5.5.4.	Dekodery wyświetlaczy siedmiosegmentowych.....	227
5.5.5.	Dekodery do wyświetlania tekstu na wyświetlaczu siedmiosegmentowym	230
5.5.6.	Dekodery adresów	231
5.6.	Podsumowanie rozdziału	232
6.	Projektowanie układów sekwencyjnych	235
6.1.	Zatrąski	236
6.1.1.	Zatrąski typu SR.....	236
6.1.2.	Transparentny przerzutnik typu D.....	242
6.1.3.	Niepoprawny opis zatrązku D	245
6.1.4.	Zatrązk master-slave	246
6.1.5.	Błędne umieszczenie zatrązków na wyjściach układów kombinacyjnych.....	248
6.2.	Przerzutniki	250
6.2.1.	Przerzutnik typu D	250
6.2.2.	Inne typy przerzutników (SR, JK i T).....	255
6.2.3.	Określenie stanów początkowych przerzutników	257
6.3.	Rejestry	259
6.4.	Rejestry przesuwne.....	265
6.5.	Cykliczne rejestry przesuwne	270
6.6.	Rejestry przesuwne z liniowym sprzężeniem zwrotnym (LFSR).....	271
6.7.	Bloki rejestrów	273
6.8.	Podsumowanie rozdziału	274

7. Liczniki, schematy opóźnień, dzielniki częstotliwości, czasomierze, zegary czasu rzeczywistego.....	276
7.1. Liczniki	277
7.1.1. Strukturalny opis licznika	277
7.1.2. Behawioralny opis licznika	280
7.1.3. Liczniki modulo M	281
7.1.4. Ładowanie danych do licznika	284
7.1.5. Liczniki rewersyjne	288
7.1.6. Opis licznika w postaci automatu skończonego	291
7.1.7. Wyniki badań realizacji liczników na układach FPGA	295
7.2. Liczniki Graya i Johnsona	297
7.2.1. Licznik Graya	297
7.2.2. Licznik Johnsona.....	299
7.3. Schematy opóźnień	300
7.4. Dzielniki częstotliwości	302
7.5. Generatory impulsów sygnałów długookresowych	303
7.6. Powolne liczniki.....	305
7.7. Czasomierze	307
7.8. Zegar czasu rzeczywistego.....	309
7.9. Podsumowanie rozdziału	312
8. Projektowanie bloków arytmetycznych.....	314
8.1. Sumatory.....	314
8.1.1. Sumatory jednobitowe	315
8.1.2. Sumator RCA z przeniesieniami szeregowymi	317
8.1.3. Sumator z przeniesieniami równoległymi	319
8.1.4. Sumator z wyborem przeniesienia.....	328
8.1.5. Sumator z przeskokiem przeniesień	333
8.1.6. Sumator z sumami warunkowymi.....	336
8.1.7. Badania efektywności realizacji sumatorów na układach FPGA.....	338
8.2. Sumator z przechowywaniem przeniesień.....	341
8.3. Subtraktory	341
8.4. Sumator szeregowy na podstawie pełnego sumatora i rejestrów przesuwnych.....	343
8.5. Sumator akumulujący	346
8.6. Sumator liczb dwójkowo-dziesiętnych BCD	348
8.7. Inkrementatory i dekrementatory	350
8.7.1. Inkrementator	350
8.7.2. Dekrementator	355
8.8. Realizacja licznika dodającego na podstawie inkrementatora	358
8.9. Podsumowanie rozdziału	359

9. Projektowanie układów mnożących.....	361
9.1. Matrycowy układ mnożący z przechowywaniem przeniesień	362
9.2. Układ mnożący z wykorzystaniem sumatorów z przeniesieniem szeregowym	367
9.3. Drzewiasty układ mnożący Wallace'a	371
9.4. Układ mnożący Dadda	380
9.5. Algorytmiczne metody realizacji układów mnożących	385
9.6. Układ mnożący Bootha	391
9.7. Zmodyfikowany układ mnożący Bootha	394
9.8. Badania realizacji układów mnożących na różnych rodzinach układów FPGA	398
9.9. Podsumowanie rozdziału	400
10. Projektowanie automatów skończonych.....	402
10.1. Modele strukturalne automatów skończonych przy ich realizacji na układach FPGA.....	402
10.2. Sposoby opisu automatów skończonych	406
10.3. Projektowanie automatu Mealy'ego.....	409
10.3.1. Opis automatu Mealy'ego.....	409
10.3.2. Synteza automatu Mealy'ego	411
10.3.3. Analiza funkcjonowania automatu Mealy'ego	413
10.4. Projektowanie automatu Moore'a.....	416
10.4.1. Opis automatu Moore'a	416
10.4.2. Synteza automatu Moore'a	418
10.4.3. Analiza funkcjonowania automatu Moore'a.....	419
10.5. Style opisu automatów skończonych.....	422
10.5.1. Opis automatów skończonych z dwoma procesami.....	422
10.5.2. Opis automatów skończonych z jednym procesem.....	426
10.5.3. Opis automatów skończonych z rejestrami na wyjściach.....	429
10.5.4. Opis automatów skończonych za pomocą instrukcji case.....	429
10.5.5. Opis niezawodnych automatów skończonych	431
10.5.6. Porównanie stylów opisu automatów skończonych	435
10.6. Kodowanie stanów wewnętrznych automatów skończonych.....	438
10.6.1. Kodowanie stanów wewnętrznych przy realizacji automatów skończonych na układach FPGA.....	438
10.6.2. Sposoby kodowania stanów wewnętrznych automatów skończonych w systemie Quartus	441
10.6.3. Kodowanie stanów wewnętrznych automatów skończonych w celu minimalizacji kosztów realizacji.....	443
10.6.4. Porównanie sposobów kodowania stanów wewnętrznych automatów skończonych	444
10.7. Projektowanie bezpiecznych automatów skończonych.....	446

10.8. Wykorzystanie wartości zmiennych wejściowych i wyjściowych do kodowania stanów automatów skończonych	447
10.8.1. Synteza automatów klasy C	448
10.8.2. Synteza automatów klasy D	453
10.8.3. Synteza automatów klasy D_1 bez rozszczepienia stanów wewnętrznych	457
10.8.4. Synteza automatów klasy E.....	459
10.8.5. Synteza automatów klasy E_1 bez rozszczepienia stanów wewnętrznych.....	465
10.8.6. Synteza automatów klasy F	467
10.8.7. Porównanie modeli automatów skończonych.....	472
10.9. Ocena efektywności metod projektowania automatów skończonych.....	475
10.9.1. Optymalizacja automatów skończonych za pomocą stylów opisu w języku Verilog.....	476
10.9.2. Optymalizacja automatów skończonych za pomocą metod oferowanych przez system Quartus.....	478
10.9.3. Wybór najbardziej efektywnego podejścia do optymalizacji automatów skończonych.....	479
10.10. Podsumowanie rozdziału	482
Bibliografia	486
Spis tabel	488
Spis rysunków	491
Streszczenie	504
Abstract.....	505

Wstęp

System wbudowany to wyspecjalizowany rodzaj komputera zaprojektowany do wykonywania jednego lub więcej określonych zadań i bezpośrednio zintegrowany ze sterowanym urządzeniem lub obiektem. W przeciwieństwie do komputerów ogólnego przeznaczenia systemy wbudowane są zoptymalizowane pod kątem konkretnych zastosowań. Można je spotkać w każdym urządzeniu elektronicznym, które pełni funkcję kontrolną lub sterującą. W dzisiejszym świecie otaczają nas liczne systemy wbudowane: od elektroniki użytkowej i urządzeń biurowych po zaawansowane systemy transportowe, lotnicze czy telekomunikacyjne. Są one nieodłączną częścią naszego życia, mają istotny wpływ na różnorodne aspekty naszej codzienności.

Tradycyjnie systemy wbudowane opierają się na mikrokontrolerach. Jednak systemy wbudowane zaprojektowane w układach FPGA (*Field Programmable Gate Arrays*) wyróżniają się największą wydajnością. Ponieważ tego rodzaju systemom stawia się spore wymagania, szczególnie w obszarze wydajności, układy FPGA są coraz bardziej atrakcyjnym rozwiązaniem. Oferują one nie tylko elastyczność konfiguracji, lecz także wyjątkową wydajność, która przewyższa możliwości tradycyjnych mikrokontrolerów. Ma ona szczególne znaczenie w zastosowaniach specjalnych, gdzie wymagane są szybkie obliczenia, reakcja na zmienne warunki lub obsługa równoległych operacji. Przykłady takich zastosowań obejmują systemy przetwarzania w czasie rzeczywistym, systemy komunikacji oraz zaawansowane systemy obliczeniowe. Dzięki elastycznej i rozbudowanej architekturze układów FPGA projektanci mogą tworzyć systemy wbudowane o złożonych funkcjonalnościach, które efektywnie działają nawet w najbardziej wymagających warunkach.

Niniejsza książka poświęcona jest problemom projektowania systemów wbudowanych na układach FPGA. W sposób systematyczny, krok po kroku, od prostych do bardziej zaawansowanych zagadnień omawia proces projektowania poszczególnych elementów i bloków funkcjonalnych systemów wbudowanych wykorzystujących elastyczność i możliwości, jakie oferują układy FPGA. Autorzy starają się przybliżyć czytelnikowi nie tylko teoretyczne aspekty projektowania, lecz także praktyczne zastosowania, wykorzystując liczne przykłady opisane w języku Verilog. Pozwalają one czytelnikowi zrozumieć teoretyczne zagadnienia, jak też zastosować je w praktyce, co ułatwia przyswajanie materiału.

W celu demonstracji wyników syntezy i symulacji prezentowanych przykładów w książce skorzystano z programu Quartus Prime (wersje 18.1–23.1), a do fizycznej implementacji użyto płytki prototypowej Intel DE1-SoC. Niemniej jednak materiał

zawarty w pracy nie jest ograniczony do jednego konkretnego zestawu narzędzi ani do systemu projektowego. Stanowi on zbiór ogólnych zasad projektowania systemów wbudowanych na układach FPGA, który może być z powodzeniem stosowany na różnych platformach i w różnych środowiskach projektowych. Warto podkreślić, że autorzy nie skupiają się wyłącznie na układach FPGA jednej firmy ani na określonym systemie projektowym. Aby opanować materiał zawarty w książce, wystarczy używać narzędzia projektowego akceptującego Verilog jako język opisu sprzętu. Wszystkie przykłady mogą być zaimplementowane także na układach FPGA firmy Xilinx, przy użyciu systemów ISE lub Vivado, podobnie jak na układach innych producentów, zapewniających odpowiednie narzędzia i wsparcie dla języka Verilog. Dzięki takiej uniwersalności podejścia czytelnicy mogą elastycznie dostosować materiał do swoich własnych preferencji i środowiska pracy.

W książce zastosowano wszechstronne podejście do analizy każdego zagadnienia oraz zaproponowano najefektywniejsze metody ich rozwiązania. Poparto je badaniami eksperymentalnymi, które pozwalają na wyciągnięcie praktycznych wniosków i zaleceń dotyczących optymalnego wykorzystania różnych podejść projektowych. Dodatkowo w publikacji znajdują się zadania praktyczne, które czytelnik może wykonać samodzielnie. Ich realizacja nie tylko pogłębia zrozumienie prezentowanych materiałów, lecz także umożliwia zdobycie cennego doświadczenia w projektowaniu elementów i bloków systemów wbudowanych na układach FPGA.

Książka składa się z 10 rozdziałów. Pierwszy wprowadza czytelnika w tematykę systemów wbudowanych, omawia ich zastosowanie oraz bazę charakterystycznych elementów. Ponadto przedstawiono w nim ogólną metodologię projektowania systemów wbudowanych, z naciskiem na podejście oparte na układach FPGA. W ramach tego rozdziału omówiono także klasyfikację układów FPGA, co pozwala czytelnikowi zrozumieć różnice między ich poszczególnymi klasami oraz ich potencjalne zastosowania. Dodatkowo czytelnik otrzymuje informacje dotyczące możliwego miejsca każdej z klas układów FPGA w projekcie systemu wbudowanego, co pozwala lepiej zrozumieć wpływ odpowiedniego typu układu FPGA na projektowany system.

W drugim rozdziale dokładnie omówiono możliwości języka Verilog w kontekście opisu projektów przeznaczonych do syntezy. Pierwsze zastosowanie tego języka miało charakter modelowania sprzętu, lecz jego wygoda i ekspresyjność szybko przekonały projektantów do wykorzystania go również do syntezy projektów. W rozdziale ukazano też liczne przykłady małych projektów, które są powszechnie spotykane w systemach wbudowanych. Takie podejście pozwala na praktyczne zrozumienie sposobu działania języka Verilog oraz jego możliwości w odniesieniu do rzeczywistych zastosowań w projektowaniu systemów wbudowanych.

Z myślą o początkujących użytkownikach w rozdziale trzecim zawarto wprowadzenie do systemu projektowania Quartus. Przeznaczony jest on głównie dla osób, które dopiero zaczynają swoją przygodę z tym narzędziem. Doświadczeni użytkownicy zaznajomieni z systemem Quartus lub innymi systemami projektowymi mogą pominąć ten rozdział. Ci, którzy jeszcze nie mieli styczności z tym systemem,

z rozdziału tego mogą zaczerpnąć informacji dotyczących interfejsu, podstawowych funkcji oraz sposobu pracy z tym narzędziem. Dzięki temu będą w stanie szybko i efektywnie rozpocząć pracę nad własnymi projektami.

Rozdział czwarty poświęcony jest projektowaniu układów kombinacyjnych przy użyciu języka Verilog, z naciskiem na ich praktyczną realizację. Omówiono tu możliwości tego języka w kontekście opisu układów kombinacyjnych, metodyki ich projektowania oraz potencjalne błędy prowadzące do automatycznego wprowadzenia zatrząsków na ich wyjściach. Szczególną uwagę zwrócono na sposoby reprezentacji układów kombinacyjnych, które są najczęściej stosowane w praktyce projektowania. Dodatkowo podkreślono osobliwości opisu układów kombinacyjnych za pomocą algorytmów z użyciem instrukcji proceduralnych. Dzięki temu czytelnik może lepiej zrozumieć proces projektowania układów kombinacyjnych oraz uniknąć potencjalnych trudności występujących w praktyce.

W rozdziale piątym omówiono budowę standardowych bloków funkcjonalnych, a także konstrukcję buforów trójstanowych. Scharakteryzowano projekty multiplekserów, demultiplekserów, koderów, koderów priorytetowych oraz różnego rodzaju dekodów. Multipleksery i demultipleksery stanowią integralne elementy wielu systemów wbudowanych, zwłaszcza w urządzeniach telekomunikacyjnych, w których odgrywają kluczową rolę w przesyłaniu i odbieraniu danych. Z kolei koderów i dekodów używa się jako konwerterów kodów, które mają szerokie zastosowanie w systemach wbudowanych do przetwarzania informacji. Przedstawienie tych klasycznych bloków funkcjonalnych pozwala czytelnikowi na lepsze zrozumienie ich roli oraz znaczenia w kontekście projektowania systemów wbudowanych.

Rozdział szósty poświęcony jest projektowaniu elementów układów sekwencyjnych. Szczególną uwagę zwrócono w nim na projektowanie zatrząsków (transparentnych przerzutników), przerzutników, rejestrów, rejestrów przesuwanych oraz bloków rejestrów na układach FPGA. Projektując zatrząski i przerzutniki, nie można ich opisać jawnie w języku Verilog – są one automatycznie wstawiane do schematu projektu przez kompilator na podstawie kodu projektu. Ta specyfika wymaga precyzji podczas procesu projektowania, gdyż wpływa to na sposób, w jaki projektant musi planować i implementować te elementy w swoim projekcie.

W rozdziale siódmym omówiono różnorodne bloki systemów wbudowanych związane z pomiarem czasu, takie jak liczniki, układy opóźniające, dzielniki częstotliwości, timery i zegary czasu rzeczywistego. Warto zauważyć, że FPGA, w przeciwieństwie do mikroprocesorów i mikrokontrolerów, nie dysponują wbudowanymi licznikami ani generatorami sygnałów zegarowych, które mogłyby posłużyć do pomiaru interwałów czasowych. W związku z tym w przypadku systemów opartych na FPGA konieczne jest korzystanie z zewnętrznego źródła sygnałów zegarowych w celu dokładnego pomiaru czasu. Dzielniki częstotliwości są wykorzystywane do zmniejszania częstotliwości zegara zewnętrznego lub referencyjnego. Dodatkowo obwody opóźniające w układach FPGA również operują na bazie sygnału zegarowego. Aby skutecznie zaimplementować te bloki, niezbędne są liczniki, czasomierze oraz zegary czasu rzeczywistego. Dlatego też bardzo dokładnie przyjrano się projektom liczników, przeprowadzono

również badania implementacji liczników na różnych rodzinach FPGA, co pozwala na optymalne zaprojektowanie i wykorzystanie tych elementów w praktyce.

Rozdział ósmy poświęcono projektowaniu urządzeń arytmetycznych. Sumatory odgrywają niezwykle istotną rolę w dziedzinie inżynierii komputerowej, znajdują zastosowanie w jednostkach arytmetyczno-logicznych, mnożarkach, przetwornikach cyfrowych oraz w kryptografii. W rozdziale omówiono najczęściej wykorzystywane metody syntezy sumatorów, obejmujące sumatory z przeniesieniem sekwencyjnym, równoległym, selektywnym, z pomijaniem przeniesienia oraz z dodawaniem warunkowym. Przedstawiono również wyniki badań dotyczące implementacji sumatorów na różnych rodzinach FPGA, co pozwala na wybór optymalnej strategii projektowej. Opisano także sumatory zachowujące przeniesienie oraz metody projektowania subtraktorów, akumulatorów, sumatorów dwójkowo-dziesiętnych, inkrementatorów i dekrementatorów.

W rozdziale dziewiątym skupiono się na znanych metodach konstrukcji układów mnożenia, obejmujących między innymi mnożarki macierzowe z zachowaniem przeniesień, mnożarki oparte na sumatorach, oraz metodach mnożenia wykorzystujących drzewa Wallace'a i Dadda. Dodatkowo przedstawiono cztery algorytmiczne strategie realizacji układów mnożenia, w tym algorytm Bootha oraz jego zmodyfikowaną wersję. Rozdział zawiera również wyniki badań eksperymentalnych dotyczących implementacji 16-bitowych układów mnożenia na różnych rodzinach układów FPGA, co pozwala na ocenę efektywności poszczególnych metod konstrukcji w praktyce.

Rozdział dziesiąty poświęcony jest implementacji automatów skończonych w układach FPGA. Automaty skończone odgrywają kluczową rolę w projektach systemów wbudowanych, pełniąc funkcje urządzeń sterujących i kontrolerów. W odróżnieniu od standardowych jednostek funkcjonalnych, które mogą być wielokrotnie wykorzystywane w różnych projektach, urządzenia sterujące muszą być projektowane od podstaw dla każdego nowego projektu systemu wbudowanego. W rozdziale przedstawiona została klasyfikacja strukturalnych modeli automatów skończonych realizowanych na FPGA, obejmująca automaty klas A, B, C, D, E i F. Omówione są również różne style opisu automatów skończonych w języku Verilog oraz sposoby kodowania ich stanów wewnętrznych. W rozdziale zaprezentowano także metody syntezy automatów klas C–F. Przedstawiono też wyniki badań nad metodami optymalizacji automatów skończonych dla różnych rodzin FPGA wykorzystujących różne style opisu Verilog, metody optymalizacji systemu Quartus oraz różne sposoby kodowania stanów wewnętrznych. Zawarto również zalecenia dotyczące najbardziej efektywnego wykorzystania każdego z tych podejść.

Każdy rozdział kończy się wnioskami, które stanowią syntezę kluczowych punktów omówionych w danym rozdziale, podsumowują główne wyniki prezentowane w tekście, oraz koncentrują uwagę czytelnika na istotnych kwestiach, które mogły umknąć podczas lektury. Wnioski pełnią funkcję swoistej rekapitulacji omawianego materiału, ułatwiając czytelnikowi jego lepsze zrozumienie. Publikację bogato

zilustrowano przykładami projektów z podaniem wyników ich syntezy i symulacji. Wszystkie przykłady są dokładnie przetestowane, co pozwala na ich praktyczne zastosowanie podczas projektowania systemów wbudowanych.

Książka jest skierowana głównie do studentów uczelni technicznych studiujących kierunki związane z elektroniką, informatyką oraz telekomunikacją. Materiał w niej zawarty może być wykorzystywany przez nauczycieli do prowadzenia wykładów, zajęć laboratoryjnych i praktycznych. Wiele omawianych zagadnień nie zostało jeszcze w pełni zbadanych, co może zainteresować studentów i młodych naukowców. Ponadto programiści mogą korzystać z pracy jako samodzielnego źródła nauki, aby zgłębić różne aspekty związane z projektowaniem systemów wbudowanych na układach FPGA.

1. Systemy wbudowane i układy scalone z logiką programowalną

1.1. Systemy wbudowane

1.1.1. Pojęcie systemu wbudowanego

We współczesnym świecie komputery oraz różnorodne urządzenia cyfrowe stanowią nieodłączny element naszego codziennego życia, przenikając niemal każdy obszar ludzkiej aktywności. Od zaawansowanych systemów kosmicznych przez precyzyjne pojazdy głębinowe aż po proste panele kontrolne w naszych gospodarstwach domowych czy interaktywne zabawki dla dzieci – systemy wbudowane są wszędzie. Pojęcie *systemu wbudowanego* obejmuje całą różnorodność elektronicznych urządzeń, które otaczają człowieka.

System wbudowany to wyspecjalizowany rodzaj komputera, który został zaprojektowany do rozwiązywania jednego lub więcej specyficznych zadań, zazwyczaj w ścisłej współpracy z kontrolowanym urządzeniem bądź systemem. W odróżnieniu od komputerów ogólnego przeznaczenia, takich jak komputery osobiste, systemy wbudowane są zoptymalizowane pod kątem konkretnej funkcji lub zestawu zadań, które mają wykonywać. Ich projektowanie i implementacja są szczegółowo dopasowane do specyficznych wymagań danego zastosowania, co sprawia, że są one nie tylko bardziej efektywne, lecz także wydajniejsze w realizacji określonych zadań.

Za pierwszy system wbudowany uważa się komputer pokładowy Autonetics D-17, który został zainstalowany na rakiecie kosmicznej Minuteman w ramach projektu kosmicznego Apollo w 1961 roku. Ten przełomowy moment, jak powszechnie wiadomo, poprzedził sukces lotu na Księżyc. Masowe zastosowanie komputera D-17 w przemyśle rozpoczęło się już w 1966 roku, co zapoczątkowało erę powszechnego wykorzystania systemów wbudowanych w różnych dziedzinach.

Obecnie systemy wbudowane pełnią kluczową funkcję w wielu obszarach działalności, obejmując szeroki zakres aplikacji:

- elektronika użytkowa: odtwarzacze audio, telefony komórkowe (smartfony), gry wideo, cyfrowe aparaty fotograficzne i kamery, kuchenki mikrofalowe, pralki, lodówki, klimatyzatory;
- urządzenia komputerowe i biurowe: drukarki, skanery, faksy, modemy, dyski twarde, myszy, klawiatury;

- systemy transportowe: różne urządzenia kontrolne (sterowniki) w samochodach (systemy zarządzania silnikiem, oświetlenie, ABS, systemy ESP/ESC, nawigatory, systemy parkowania), systemy sterowania pociągami i samolotami;
- systemy telekomunikacyjne: centrale telefoniczne, urządzenia sterujące transmisją danych w sieciach, hosty, modemy, adaptery, routery, huby;
- sprzęt medyczny: systemy monitorowania życia pacjenta, elektrokardiografy, tomografy komputerowe, urządzenia ultradźwiękowe, urządzenia do tomografii rezonansu magnetycznego, aparaty słuchowe, systemy korekcji wzroku, różnorodny sprzęt diagnostyczny;
- robotyka: roboty przemysłowe, ramiona robotyczne wykorzystywane w montażu i produkcji, systemy robotyczne pracujące w niebezpiecznych środowiskach, takie jak pojazdy głębinowe, stacje kosmiczne i satelity, urządzenia do badania kraterów wulkanicznych, urządzenia do produkcji chemicznej, reaktory jądrowe, elektrownie jądrowe i inne.

Dzięki systemom wbudowanym te różnorodne urządzenia stają się inteligentne, interaktywne i łatwe w obsłudze. Zapewniają one bezpieczeństwo, komfort oraz skuteczne i precyzyjne działanie nawet w najbardziej wymagających warunkach środowiskowych.

1.1.2. Funkcje systemów wbudowanych

Podstawowe funkcje wykonywane przez systemy wbudowane obejmują:

- sterowanie różnymi urządzeniami peryferyjnymi;
- odbieranie i przetwarzanie sygnałów pochodzących z przycisków, przełączników oraz czujników;
- generowanie sygnałów sterujących;
- implementację interfejsów urządzeń peryferyjnych, takich jak porty szeregowo, równoległe, oraz obsługę klawiatury i wyświetlaczy LED, LCD, VGA.

Dodatkowo współczesne systemy wbudowane wykazują następujące cechy:

- konieczność przetwarzania dużych ilości danych z wysoką prędkością, co jest istotne szczególnie w przypadku systemów obsługujących *Big Data*;
- elastyczność pozwalająca na szybką adaptację algorytmów przetwarzania informacji, co jest niezbędne na przykład w systemach telekomunikacyjnych przy zmianie protokołów przesyłania danych;
- możliwość dodawania nowych funkcji przetwarzania informacji, co jest ważne na przykład w przypadku smartfonów i umożliwia rozbudowę ich funkcjonalności.

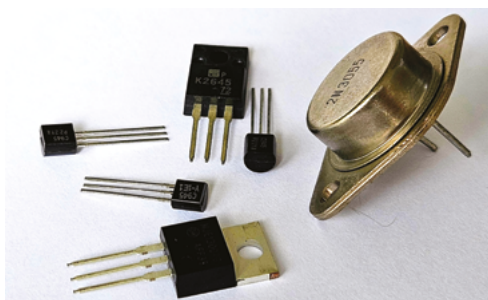
Systemy wbudowane często są poddawane specyficznym wymaganiom, które wynikają z konkretnych zadań do wykonania oraz z zakresu ich zastosowania:

- konieczność pracy w czasie rzeczywistym, gdzie reakcja systemu wbudowanego na zdarzenie musi być szybka ze względu na ściśle określone limity czasowe;

- minimalne zużycie energii, aby system działał jak najdłużej na jednym źródle zasilania;
- minimalne rozmiary i wymiary, aby taki system można było łatwo umieścić w różnych urządzeniach;
- zdolność do pracy w niebezpiecznych środowiskach, gdzie warunki pracy mogą być trudne i wymagają od systemu wyjątkowej wytrzymałości i niezawodności.

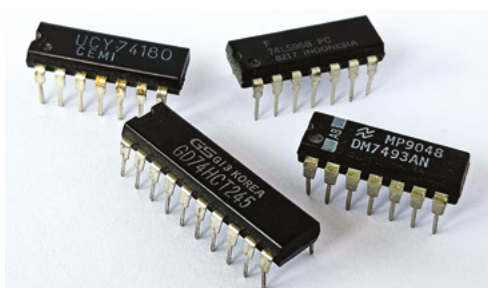
1.1.3. Podstawowe komponenty systemów wbudowanych

Podstawowe elementy składowe systemów wbudowanych są bezpośrednio związane z rozwojem mikrokomputerów i mikrokontrolerów. Początkowo systemy te opierały się głównie na tranzystorach (rysunek 1.1) oraz na układach scalonych o małym i średnim stopniu integracji (rysunek 1.2).



RYS. 1.1. Tranzystory

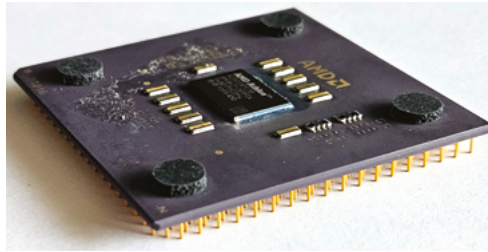
ŹRÓDŁO: zbiory Tomasza Grzesia.



RYS. 1.2. Układy o małym i średnim stopniu scalenia

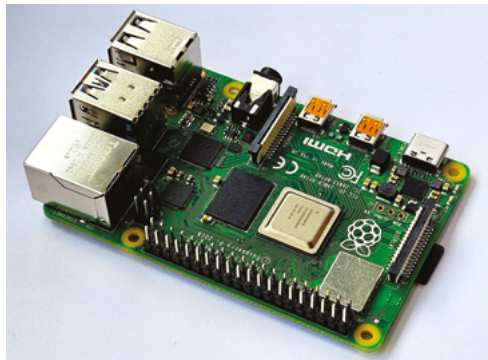
ŹRÓDŁO: zbiory Tomasza Grzesia.

Powszechne stosowanie systemów wbudowanych rozpoczęło się wraz z wprowadzeniem mikroprocesorów (rysunek 1.3). Kolejnym krokiem było wykorzystanie komputerów jednopłytkowych (rysunek 1.4) oraz mikrokontrolerów (rysunek 1.5) do konstruowania systemów wbudowanych.



RYS. 1.3. Mikroprocesory

ŹRÓDŁO: zbiory Tomasza Grzesia.



RYS. 1.4. Komputer jednopłytkowy

ŹRÓDŁO: zbiory Tomasza Grzesia.



RYS. 1.5. Mikrokontrolery

ŹRÓDŁO: zbiory Tomasza Grzesia.

W ostatnich latach coraz częściej systemy wbudowane powstają przy użyciu programowalnych układów scalonych, takich jak układy FPGA (*Field Programmable Gate Array*; rysunek 1.6) oraz układy SoC (*System-on-Chip*). W przypadku produkcji masowej, na przykład w telefonach komórkowych, systemy wbudowane mogą być tworzone na podstawie układów ASIC (*Application Specific Integrated Circuits*). Ta tendencja wynika z rosnącej potrzeby elastyczności i dostosowania

do zmieniających się wymagań, jak również z możliwości szybkiego prototypowania i iteracji w procesie projektowania.



RYS. 1.6. Układ FPGA

ŹRÓDŁO: zbiory Tomasza Grzesia.

W niektórych przypadkach płyty główne komputerów osobistych mogą pełnić funkcję podstawy dla systemów wbudowanych (rysunek 1.7). W takich sytuacjach wykorzystuje się powszechnie dostępne systemy operacyjne ogólnego przeznaczenia oraz popularne języki programowania.



RYS. 1.7. Płyta główna komputera osobistego

ŹRÓDŁO: zbiory Tomasza Grzesia.

Często do tworzenia systemów wbudowanych wykorzystuje się gotowe płyty komputerowe. Takie podejście eliminuje konieczność projektowania systemu od podstaw, jednocześnie zapewnia dostęp do zaawansowanych funkcji i interfejsów oferowanych przez tego typu płyty.

Podsumowując, oto lista głównych komponentów, które stanowią podstawę dla systemów wbudowanych:

- mikrokontrolery;
- mikroprocesory;

- mikrokomputery;
- komputery jednopłytkowe;
- płyty główne do komputerów osobistych;
- układy scalone z logiką programowalną;
- układy SoC;
- wyspecjalizowane układy scalone ASIC;
- gotowe płyty komputerowe.

Ogólnie rzecz biorąc, mikrokontroler to rodzaj mikrokomputera zintegrowany w pojedynczym układzie scalonym. Jego architektura obejmuje kluczowe elementy komputerowe, takie jak procesor, magistrale i pamięć, a także szereg urządzeń peryferyjnych często stosowanych w systemach wbudowanych. Wśród tych urządzeń znajdują się timery, liczniki, porty szeregowy i równoległy, przetworniki analogowo-cyfrowe, przetworniki cyfrowo-analogowe oraz kontrolery standardowych interfejsów. Istotną wadą mikrokontrolerów jest ich ograniczona moc obliczeniowa, która nie jest zazwyczaj wymagana w większości zastosowań systemów wbudowanych.

Mikroprocesory charakteryzują się dużą mocą obliczeniową w porównaniu z mikrokontrolerami, jednak mają zwykle ograniczoną liczbę urządzeń peryferyjnych. Jedną z istotnych zalet mikroprocesorów jest zdolność do obsługi ogólnych systemów operacyjnych, takich jak Windows, Unix czy Linux, oraz popularnych języków programowania, jak na przykład Assembler, C/C++ czy Java.

Te ograniczenia mikroprocesorów rekompensują komputery jednopłytkowe, które zapewniają także możliwość korzystania z ogólnych systemów operacyjnych. Niemniej w większości przypadków takie komputery nie spełniają wymagań stawianych systemom wbudowanym pod względem poboru mocy, wagi i rozmiarów.

Kolejną istotną wadą mikrokontrolerów i mikroprocesorów jest ograniczona liczba zewnętrznych wyprowadzeń do przesyłania danych. Aby rozwiązać ten problem, stosuje się multipleksowanie danych, a do przesyłania większych ilości informacji często wykorzystuje się interfejsy szeregowy. Te techniki mogą jednak prowadzić do spadku wydajności systemu wbudowanego, co jest nieakceptowalne w przypadku systemów czasu rzeczywistego.

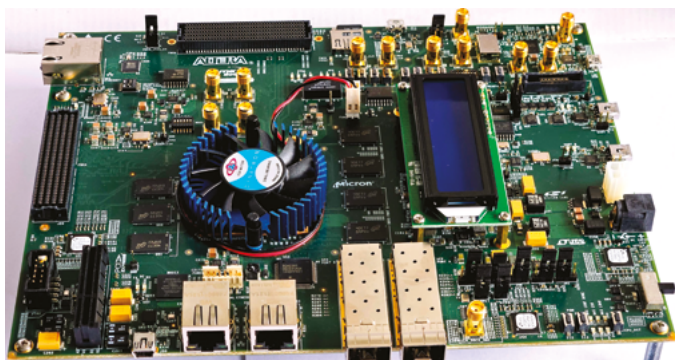
W ostatnich latach układy FPGA i SoC stały się powszechnie wykorzystywanymi układami programowalnymi do tworzenia systemów wbudowanych. Dzięki ich dużej funkcjonalności oraz znacznej liczbie zewnętrznych wyprowadzeń pozwalają one na elastyczną konfigurację ich wewnętrznej struktury, umożliwiając tworzenie kompleksowych systemów mikroprocesorowych z rozbudowanym zestawem urządzeń peryferyjnych na jednym układzie scalonym. Kluczową cechą układów FPGA jest możliwość rozwiązywania problemów systemów wbudowanych nie tylko programowo, jak to ma miejsce w przypadku mikrokontrolerów i mikroprocesorów, lecz także sprzętowo. Oznacza to, że specjalne zadania mogą być przedstawione w postaci układu logicznego, który zostaje sprzętowo zaimplementowany na FPGA. W efekcie prędkość rozwiązywania problemów w układach FPGA, w porównaniu z tradycyjnymi mikroprocesorami, jest wyższa nawet o kilka rzędów wielkości.

Jedną z istotnych cech układów FPGA jest duża liczba zewnętrznych wyprowadzeń ogólnego przeznaczenia. Dzięki temu eliminowana jest konieczność multipleksowania danych oraz stosowania różnorodnych standardowych interfejsów. Ponadto układy FPGA wyróżniają się niskim poborem mocy, a także компактowymi rozmiarami i lekką wagą.

Trendy rozwojowe we współczesnych układach FPGA zmierzają w kierunku szerokiego wykorzystania bloków cyfrowego przetwarzania sygnałów (DSP), wbudowanej pamięci, nadajników-odbiorników oraz kontrolerów popularnych interfejsów.

Przykładami gotowych płytek komputerowych używanych do tworzenia małych systemów wbudowanych są standardy PC/104 i PC/104 PLUS. Płyty te opierają się na procesorach Intel x86 i charakteryzują się znacznie mniejszymi rozmiarami w porównaniu z tradycyjnymi płytami głównymi komputerów osobistych. Na tych płytkach można z powodzeniem stosować systemy operacyjne czasu rzeczywistego, takie jak Linux, NetBSD, a także MicroC/OS-II, QNX i VxWorks. Niektóre z tych płyt mogą również korzystać z mikroprocesorów innych niż Intel x86.

Podczas tworzenia systemów wbudowanych opartych na FPGA często nie ma potrzeby projektowania własnej płytki drukowanej dla przyszłego projektu. Producenci FPGA oraz ich partnerzy oferują szeroki wybór gotowych do użycia płyt dla systemów wbudowanych na FPGA lub SoC (rysunek 1.8). Wystarczy wybrać właściwą płytę z dostępnej listy i odpowiednio skonfigurować układ FPGA.



RYS. 1.8. Gotowa płytka do systemów wbudowanych opartych na FPGA/SoC

ŹRÓDŁO: zbiory Tomasza Grzesia.

Zwróćmy uwagę na kilka korzystnych cech gotowych płyt systemów wbudowanych opartych na układach programowalnych:

- obecność układu SoC;
- obecność układów pamięci zewnętrznej FLASH i DRAM;
- obecność kontrolerów standardowych interfejsów.

W układzie SoC zaimplementowane są wydajny procesor ARM oraz szereg wbudowanych układów logicznych, natomiast układy pamięci służą do przechowywania

danych stałych i tymczasowych. Takie płyty mogą być produkowane masowo i wyróżniają się niskim kosztem.

1.1.4. Urządzenia peryferyjne

Systemy wbudowane wchodzą w interakcję ze środowiskiem i wpływają na sterowany obiekt za pomocą zestawu urządzeń peryferyjnych. Ponieważ obiekty, które mają być sterowane przez systemy wbudowane, charakteryzują się dużą różnorodnością, zestaw urządzeń peryferyjnych systemów wbudowanych jest również bardzo zróżnicowany. Najczęściej spotyka się w nich:

- sterowniki standardowych interfejsów szeregowych: RS-232, RS-422, RS-485;
- kontrolery synchronicznych interfejsów szeregowych: I2C, SPI, SSC i ESSI;
- kontrolery portów USB;
- kontrolery kart multimedialnych SD i Compact Flash;
- kontrolery sieciowe Ethernet i LAN;
- standardowe sterowniki magistrali: CAN-Bus, LIN-Bus i PROFIBUS;
- timery: PLL, Capture/Compare i Time Processing Units;
- przetworniki analogowo-cyfrowe i cyfrowo-analogowe (ADC i DAC);
- wbudowane debuggery: JTAG, ISP, ICSP, BDM Port, BITP, DPS itp.

Dodatkowo na płytach prototypowych dla systemów wbudowanych i płytach szkoleniowych można znaleźć następujące urządzenia peryferyjne:

- przyciski;
- różne typy przełączników;
- diody LED;
- wyświetlacze LCD;
- sterowniki wyświetlaczy VGA;
- kontrolery myszy, kontrolery klawiatury itp.

Dla celów debugowania i wstępnego programowania do systemów wbudowanych można podłączyć różne rodzaje klawiatur, takie jak matrycowa, PS/2 i inne.

1.1.5. Niezawodne działanie systemów wbudowanych

Systemy wbudowane często funkcjonują autonomicznie, bez konieczności interakcji z człowiekiem. W przypadku ewentualnych błędów brakuje możliwości ręcznego resetowania przez operatora. Dlatego też programy dla systemów wbudowanych są tworzone i testowane z większą precyzją niż te przeznaczone do komputerów osobistych. Dodatkowo w systemach tych należy unikać stosowania elementów ruchomych, takich jak dyski twarde, przełączniki i przyciski. Warto jednak zauważyć, że przełączniki i przyciski są często wykorzystywane na płytach prototypowych podczas testowania systemów wbudowanych.

Niektóre klasy systemów wbudowanych mają wyższe wymagania dotyczące niezawodności ich działania. Przykładowo system wbudowany nie może być wyłączony w trakcie naprawy ani gdy staje się niedostępny (jak w przypadku satelitów kosmicznych, kabli podmorskich czy systemów transportowych). Niekiedy musi on działać nawet w przypadku awarii (na przykład w nawigacji samolotu, sterowaniu reaktorem jądrowym czy zarządzaniu paliwem w samolotach jednosilnikowych). Czasem awaria systemu wbudowanego może prowadzić do znaczących strat finansowych, jak w przypadku centrali telefonicznych, sterowników w produkcji przemysłowej, przekazów pieniężnych, transakcji handlowych czy automatyzacji w sprzedaży i serwisie.

Poprawa niezawodności systemów wbudowanych wymaga zastosowania różnych metod. Przede wszystkim konieczne są dokładne planowanie, opracowanie i testowanie we wszystkich fazach projektowania. Kluczowymi etapami są symulacja komputerowa, testy fizyczne na płycie prototypowej oraz eksperymentalna współpraca z kontrolowanym obiektem.

Błędy w systemach wbudowanych można podzielić na programowe (*software errors*) oraz sprzętowe (*hardware errors*). Niektóre z tych ostatnich nie mogą być automatycznie naprawione przez system wbudowany. Te, z których system wbudowany może samodzielnie wyjść, nazywane są „miękkimi błędami” (*soft errors in the hardware*).

Rozważmy kilka z najczęściej stosowanych metod poprawy niezawodności systemów wbudowanych:

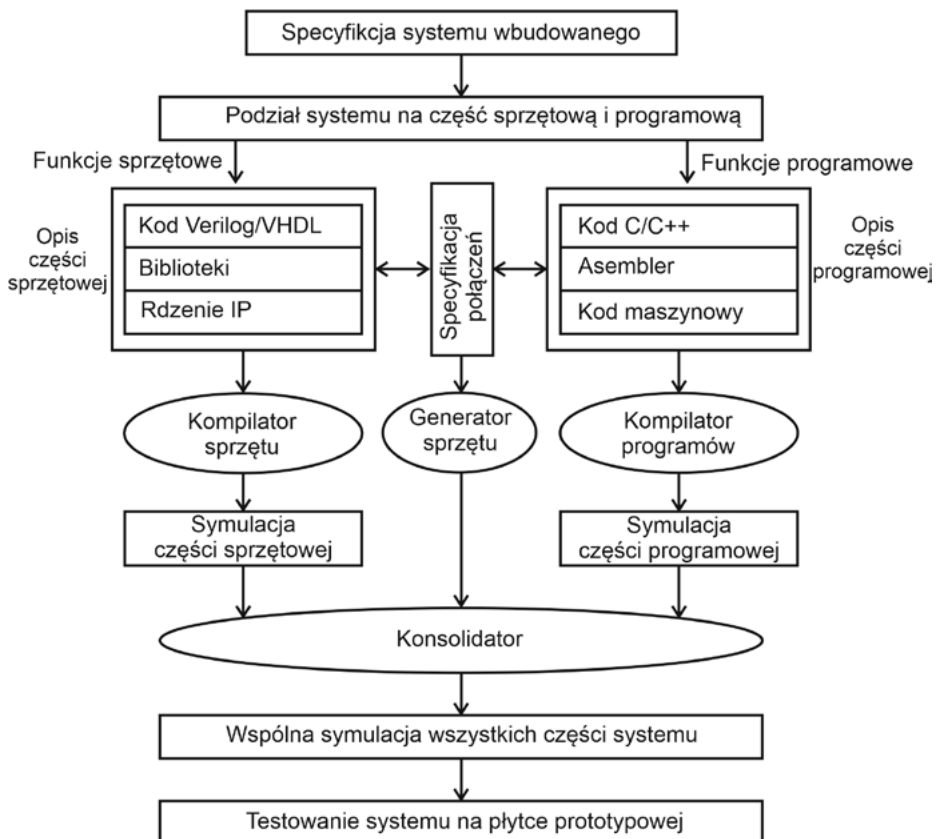
- użycie układu czuwającego *watchdog*, który zrestartuje procesor, jeśli program nie ostrzeże *watchdog*a w odpowiednim czasie;
- zwielokrotnienie jednostek systemu wbudowanego i szybkie przełączanie na jednostki będące w stanie czuwania w przypadku niedostępności innych;
- przełączenie oprogramowania w tryb *limp mode*, który zapewnia częściowe działanie systemu wbudowanego w przypadku wystąpienia błędu;
- wykorzystanie koncepcji TCB (*Trusted Computing Base*) przy projektowaniu architektury systemu wbudowanego, co daje wysoki stopień niezawodności projektowanego systemu;
- wykorzystanie wbudowanego hipernadzorcy (*hypervisor*), który zapewnia ukrywanie informacji, dzięki czemu komponenty oprogramowania nie mogą kolidować z innymi podsystemami na poziomie uprzywilejowanym;
- zapewnienie autonomicznych podsystemów w przypadku błędów automatycznego uruchamiania (resetowanie i ponowne uruchomienie) itp.

Należy podkreślić, że w praktyce można stosować różne kombinacje wymienionych metod w celu zwiększenia niezawodności systemów wbudowanych.

1.1.6. Metodologia projektowania systemów wbudowanych opartych na FPGA

Ogólna metodyka projektowania systemów wbudowanych opartych na FPGA została przedstawiona w postaci ścieżki projektowej na rysunku 1.9. Składa się ona z następujących głównych etapów:

- specyfikacja systemu wbudowanego;
- rozdzielanie systemu wbudowanego na oprogramowanie i sprzęt;
- opis części sprzętowej;
- opis części programowej;
- specyfikacja połączeń;
- symulacja części sprzętowej;
- symulacja części programowej;
- wspólna symulacja całego systemu;
- fizyczne testowanie systemu na płycie prototypowej.



RYŚ. 1.9. Proces projektowania systemów wbudowanych

Ponieważ system wbudowany to wyspecjalizowany komputer, zazwyczaj można go podzielić na dwie części: sprzętową i programową. Proste systemy wbudowane mogą składać się tylko z części sprzętowej.

Pierwszym krokiem w projektowaniu systemu wbudowanego jest zdefiniowanie jego specyfikacji, w której określa się jego główne funkcje i ogólne parametry. Następnie system wbudowany zostaje podzielony na części sprzętową i programową, zazwyczaj w sposób ręczny. Skutkuje to zdefiniowaniem i rozdzieleniem funkcji, które mają być realizowane przez poszczególne części.

Dalsze etapy projektowania systemu wbudowanego mogą przebiegać równolegle: projektanci sprzętu zajmują się opracowaniem części sprzętowej, podczas gdy programiści tworzą część programową.

Projektowanie sprzętowe systemu wbudowanego polega na opisanu wszystkich jego komponentów sprzętowych za pomocą języka opisu sprzętu, takiego jak Verilog lub VHDL. W tym celu można korzystać z modułów bibliotecznych narzędzi projektowych, takich jak Quartus lub Vivado, oraz z rdzeni IP-Core dostarczanych przez różnych producentów. Do opisu elementów sprzętowych można również wykorzystać schematy ideowe tworzone w edytorach graficznych oraz oddzielne elementy logiczne znane jako logika klejowa (*glue logic*).

Sprzęt w systemach wbudowanych może być opisany na różnych poziomach szczegółowości:

- systemowym;
- przesłań międzyrejestrowych (*register transfer level* – RTL);
- bramek;
- tranzystorów.

Oprócz tego można także zastosować mieszany sposób opisu, w którym na przykład na poziomie systemowym stosowane są oddzielne bramki logiki klejowej.

Wszystkie wymienione poziomy szczegółowości opisu sprzętu są obsługiwane przez język Verilog (choć poziom systemowy w mniejszym stopniu). Ponadto można w nim wyróżnić następujące style opisu modułów sprzętowych [7]:

- strukturalny;
- behawioralny, z wykorzystaniem instrukcji proceduralnych;
- równań logicznych, z użyciem instrukcji przypisania.

Należy zauważyć, że style opisu języka Verilog nie są bezpośrednio związane z poziomami szczegółowości opisu sprzętu.

Każdy komponent sprzętu w systemie wbudowanym jest modelowany za pomocą narzędzi symulacyjnych w ramach systemu projektowego. Dodatkowo przeprowadza się symulację całej części sprzętowej systemu wbudowanego. Wynikiem projektu sprzętowego systemu wbudowanego są pliki konfiguracyjne zawierające informacje o konfiguracji FPGA.

Projektowanie części programowej systemu wbudowanego polega na opisaniu wszystkich komponentów programowych systemu. Do tego celu można wykorzystać języki wysokiego poziomu, takie jak C i C++, język assemblera, a także kod maszynowy.

Komponenty programowe systemu wbudowanego są przeznaczone do pracy na wbudowanym procesorze układu programowalnego. Dlatego po skompilowaniu każdego modułu oprogramowania przeprowadza się symulację przy użyciu emulatora odpowiedniego procesora.

Po opracowaniu wszystkich modułów oprogramowania systemu wbudowanego następuje symulacja części programowej. Stworzone programy są przechowywane w pamięci systemu wbudowanego i zazwyczaj przeznaczone do uruchamiania na wbudowanym procesorze.

Części sprzętowe i programowe systemu wbudowanego muszą być w stanie komunikować się ze sobą. W tym celu wykorzystuje się wewnętrzne połączenia, które mogą przybierać różne formy, takie jak kilka połączeń przewodowych, wspólną magistralę lub złożoną strukturę przełączającą. Na przykład w układach SoC firmy Intel do komunikacji między sprzętem zaimplementowanym na FPGA a wbudowanym procesorem stosuje się standardy interfejsów AMBA AXI i Avalon Memory-Mapped (Avalon-MM). Zazwyczaj wewnętrzne połączenia w SoC są syntetyzowane automatycznie przez generator sprzętowy zawarty w systemie projektowym.

Po oddzielnym opracowaniu sprzętu i oprogramowania wszystkie komponenty systemu są integrowane ze sobą za pomocą konsolidatora. Następnie kompletny system wbudowany jest symulowany jako całość. Ostatecznym etapem jego projektowania jest fizyczny test na płycie prototypowej. Wówczas dokładnie ocenia się działanie systemu w realnych warunkach oraz identyfikuje i usuwa ewentualne błędy czy nieprawidłowości w jego funkcjonowaniu.

1.2. Programowalne układy logiczne

Główną cechą charakterystyczną układów programowalnych jest ich zdolność do zmiany struktury w celu realizacji funkcji, które są definiowane przez użytkownika. Co istotne, może on samodzielnie programować lub konfigurować układy FPGA bez potrzeby angażowania specjalistycznych organizacji. W przeciwieństwie do układów programowalnych specjalizowane układy scalone ASIC są konfigurowane fabrycznie i nie umożliwiają użytkownikowi modyfikacji ich struktury po produkcji. Dodatkowo innym rodzajem nowoczesnych układów są mikroprocesory i mikrokontrolery, które mają stałą strukturę, a ich funkcje są realizowane poprzez odczyt i wykonanie instrukcji zapisanych w pamięci programu.

1.2.1. Klasyfikacja układów programowalnych

Programowalne układy logiczne są zwyczajowo dzielone na następujące klasy:

- PLD (*Programmable Logic Device*);
- CPLD (*Complex PLD*);
- FPGA (*Field Programmable Gate Array*);
- SoC (*System-on-Chip*).

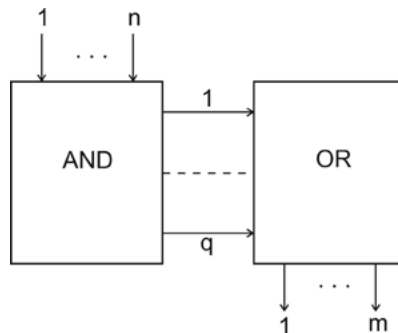
Natomiast układy PLD obejmują następujące klasy:

- PLA (*Programmable Logic Array*);
- PROM (*Programmable Read Only Memory*);
- PAL (*Programmable Array Logic*).

1.2.2. Układy PLD

Podstawowymi elementami układu PLA są dwie macierze programowalne: AND i OR (rysunek 1.10). Macierz AND jest programowalna w taki sposób, aby generować dowolne iloczyny logiczne n zmiennych na swoich wyjściach. Wyjścia macierzy AND są połączone z wejściami macierzy OR i są nazywane *szynami pośrednimi* (*product terms*).

Macierz OR jest także programowalna i na swoich wyjściach umożliwia realizację dowolnej funkcji logicznej OR q argumentów.



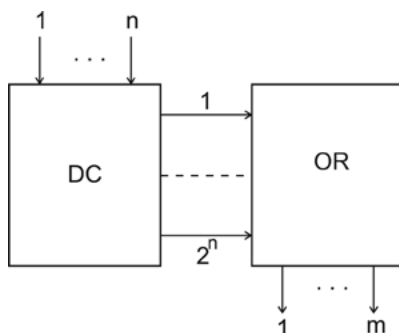
RYS. 1.10. Struktura PLA

Niech $Y = \{y_1, \dots, y_N\}$ oznacza pewien zestaw funkcji logicznych, a $X = \{x_1, \dots, x_L\}$ – ich argumenty. Ponieważ każdą funkcję logiczną można przedstawić w postaci sumy iloczynów, układ PLA z rysunku 1.10 może realizować dowolny zestaw funkcji logicznych Y , spełniający następujące ograniczenia:

$$\begin{aligned} L &\leq n; \\ N &\leq m; \\ Q &\leq q, \end{aligned} \tag{1.1}$$

gdzie: L – liczba argumentów funkcji logicznych, N – liczba funkcji, a Q – liczba różnych iloczynów w wyrażeniach funkcji.

Gdy macierz AND jest stale skonfigurowana do realizacji funkcji pełnego dekodera DC, uzyskujemy układ PROM (rysunek 1.11).



RYS. 1.11. Struktura układu PROM

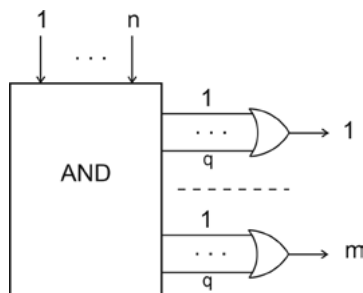
W układzie PROM liczba szyn pośrednich jest stała i równa 2^n . Na takim układzie można zrealizować dowolny zestaw funkcji logicznych, które są reprezentowane w kanonicznej postaci sumy (KPS) przy spełnieniu następujących nierówności:

$$\begin{aligned} L &\leq n; \\ N &\leq m. \end{aligned} \quad (1.2)$$

W strukturze PAL (rysunek 1.12) macierz OR ma stałe ustawienie, a jedynie macierz AND jest programowalna. Każdy układ funkcji logicznych Y spełniający poniższe nierówności może być zaimplementowany na PAL:

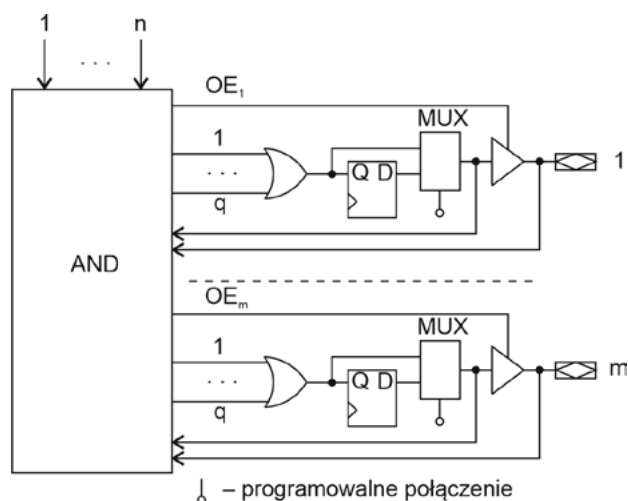
$$\begin{aligned} L &\leq n; \\ N &\leq m; \\ Q(y_i) &\leq q, \end{aligned} \quad (1.3)$$

gdzie $Q(y_i)$ – liczba różnych iloczynów w układzie funkcji logicznych Y .



RYS. 1.12. Struktura PAL

Historycznie układy PROM powstały jako pierwsze. Następnie opracowano układy PLA, gdyż uznano, że układy PROM są szczególnym przypadkiem układów PLA. W miarę rozwoju eksperymentów z programowaniem macierzy AND i OR pojawiły się układy PAL. Ich architektura została udoskonalona, co doprowadziło do powstania uniwersalnych układów PAL (rysunek 1.13).



RYS. 1.13. Struktura uniwersalnego PAL

Struktura uniwersalnego PAL składa się z programowalnej macierzy AND oraz m makrokomórek wyjściowych. Każda makrokomórka zawiera bramkę OR (część macierzy OR), przerzutnik, multiplexer MUX, bufor trójstanowy sterowany przez jedną z szyn pośrednich oraz jeden lub dwa obwody sprzężenia zwrotnego. Wyjście makrokomórki jest połączone z dwukierunkowym wyprowadzeniem układu PAL poprzez bufor trójstanowy.

Przerzutnik i multiplexer w każdej makrokomórce umożliwiają realizację funkcji kombinacyjnej lub rejestrowej. Bufor trójstanowy pozwala na wykorzystanie dwukierunkowego wyprowadzenia PAL jako wyjścia bądź wejścia. W drugim przypadku wartość zrealizowanej funkcji podawana jest na wejście macierzy AND.

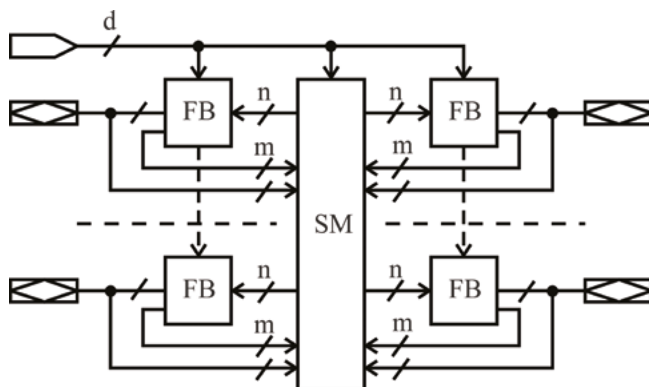
Udoskonalenie architektury makrokomórek uniwersalnych PAL pozwoliło na znaczne zwiększenie ich możliwości architektonicznych:

- przerzutnik obecny w strukturze makrokomórek umożliwia realizację *funkcji rejestrowych* (funkcji kombinacyjnych z przerzutnikiem na wyjściu);
- multiplexer pozwala na wybór rodzaju funkcji realizowanej przez makrokomórkę: kombinacyjnej lub rejestrowej;
- bufor trójstanowy umożliwia wykorzystanie dwukierunkowego wyprowadzenia makrokomórki jako wyjścia lub wejścia;
- sprzężenia zwrotne dają możliwość realizacji funkcji wewnętrznej, gdy dwukierunkowe wyjście makrokomórki jest wykorzystywane jako wejście.

W praktyce układy PLD głównie wykorzystywane były do zastępowania w projektach elementów o niskiej i średniej skali integracji. Funkcjonalność układów PLD nie była wystarczająca do budowy pełnowartościowych systemów wbudowanych.

1.2.3. Złożone układy programowalne CPLD

Postępy w mikroelektronice i zwiększenie skali integracji umożliwiły umieszczenie wielu układów PLD w jednym układzie scalonym. W rezultacie powstała nowa architektura, znana jako CPLD (rysunek 1.14).



RYŚ. 1.14. Struktura CPLD

Struktura CPLD składa się z zestawu *bloków funkcyjnych* (*functional blocks* – FB), które są połączone *macierzą przełączników* (*switch matrix* – SM). Każdy blok funkcyjny ma architekturę uniwersalnego PAL. Dane wejściowe z dwukierunkowych wyprowadzeń są najpierw kierowane do macierzy przełączników SM, a następnie do wejść bloków funkcyjnych FB.

Jeden blok funkcyjny umożliwia generowanie m funkcji kombinacyjnych lub rejestrowych, których wyniki mogą być przekazywane na wyjścia układu CPLD bądź przez układ sprzężenia zwrotnego do macierzy przełączników SM.

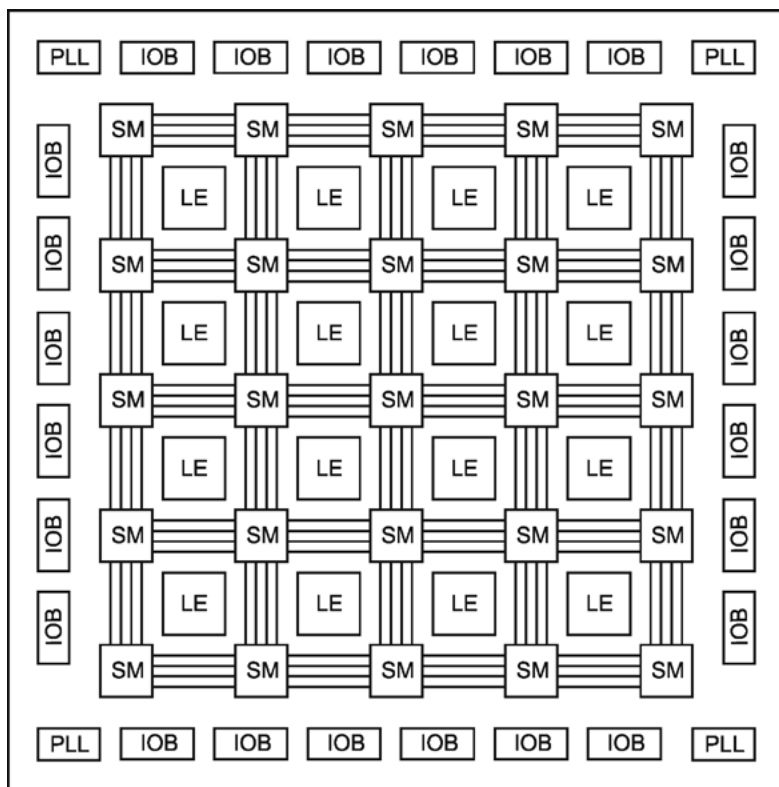
Oprócz dwukierunkowych wyprowadzeń architektura CPLD zawiera również d wejść sterujących, które dostarczają sygnały synchronizacji (*clk*) i zerowania (*reset*) przerzutników oraz sygnały sterowania buforami trójstanowymi (*output enable* – OE).

1.2.4. Układy FPGA

Równolegle do architektury CPLD rozwinęła się inna klasa układów programowalnych zwanych *bezpośrednio programowalną macierzą bramek* – FPGA. Podstawową różnicą między FPGA a CPLD jest to, że w FPGA głównym elementem funkcyjnym nie jest para programowalnych macierzy AND–OR, lecz mały element pamięciowy nazywany *tablicą LUT* (*Look-Up Table*) lub *generatorem funkcyjnym LUT*.

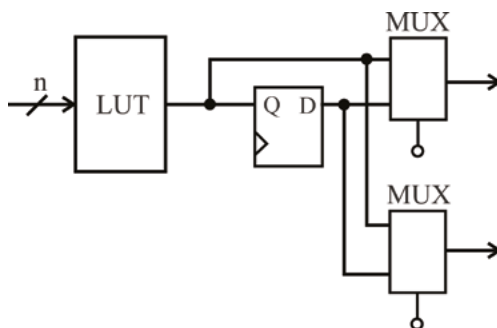
Element LUT to pamięć o dostępie swobodnym (RAM) z jednym wyjściem i ograniczoną liczbą n wejść adresowych (w przypadku FPGA zwykle n wynosi 4 lub 6). W zależności od danych zapisanych w elemencie LUT może on realizować dowolną funkcję logiczną n argumentów. Ponieważ te informacje są zapisywane natychmiast po włączeniu zasilania FPGA, proces ustawiania struktury FPGA nie jest nazywany programowaniem, lecz konfigurowaniem.

W swojej najbardziej ogólnej postaci struktura FPGA obejmuje macierz elementów logicznych (*logic elements* – LE) zbudowanych na bazie generatorów funkcyjnych LUT, które są rozmieszczone pomiędzy *polem połączeń* (rysunek 1.15). Składa się ono z zestawu pionowych i poziomych linii kanałów połączeniowych. Mogą one obejmować całą szerokość (lub wysokość) urządzenia FPGA albo być skonfigurowane jako pojedyncze segmenty o zróżnicowanej długości. Małe macierze przełączające SM służą do przesyłania sygnałów pomiędzy kanałami, a na obrzeżach układu FPGA znajdują się bloki wejścia/wyjścia (IOB), które zarządzają wyprowadzeniami dwukierunkowymi. Dodatkowo w strukturze FPGA mogą być również zawarte pętle synchronizacji fazowej PLL (*Phase-Locked Loop*).



RYS. 1.15. Uogólniona struktura FPGA

Typowy element logiczny FPGA (rysunek 1.16) zazwyczaj zawiera generator funkcyjny LUT, przerzutnik oraz kilka multiplexerów. Każdy taki element umożliwia realizację funkcji kombinacyjnej lub rejestrowej o n argumentach, przy czym ich wartości mogą być przekazywane na wyprowadzenia FPGA bądź na wejścia innych elementów logicznych za pomocą kanałów połączeniowych.



RYS. 1.16. Uogólniona struktura elementu logicznego FPGA

Chociaż pojedynczy element logiczny ma ograniczoną funkcjonalność, FPGA oferuje ogromną liczbę tych elementów, co zapewnia większą elastyczność w implementacji systemów wbudowanych niż w przypadku CPLD.

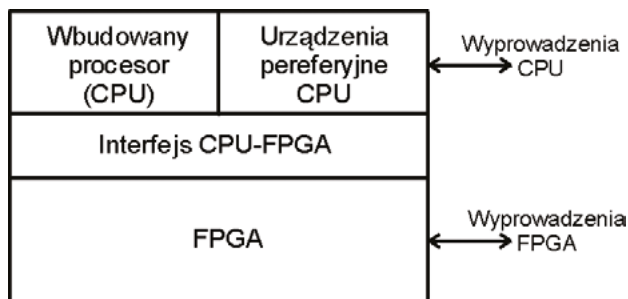
Konkretny układ FPGA, na którym ma być zrealizowany projekt systemu wbudowanego, nazywany jest *docelowym* FPGA. Architektura FPGA różnych producentów, a także różnych rodzin FPGA mogą się znacząco różnić. Dlatego też, aby efektywnie zaimplementować projekt na FPGA, istotne jest dokładne zbadanie wszystkich cech architektonicznych docelowego FPGA.

1.2.5. Układy SoC

Wraz ze wzrostem stopnia integracji układów scalonych architektura FPGA była intensywnie rozwijana poprzez dodawanie pętli PLL, bloków pamięci, bloków cyfrowego przetwarzania sygnałów (DSP) itp. Te dodatkowe bloki w terminologii FPGA określane są jako bloki wbudowane. W pewnym momencie stało się możliwe zintegrowanie na jednym układzie procesora i FPGA (rysunek 1.17). Takie układy zaczęto nazywać *systemami na układzie scalonym* (System-on-chip – SoC), a procesor – procesorem wbudowanym (*embedded processor*).

Wbudowany procesor w SoC zazwyczaj stanowi pewną wersję popularnego mikrokontrolera. Na przykład w rodzinie Cyclone V FPGA zaimplementowano dwurdzeniowy mikrokontroler ARM Cortex-A9.

Podobnie jak każdy mikrokontroler, procesor wbudowany dysponuje zestawem urządzeń peryferyjnych oraz własnym zestawem wyprowadzeń zewnętrznych, które nie pokrywają się z wyprowadzeniami FPGA.



RYS. 1.17. Uogólniona struktura SoC

Wewnętrzny interfejs CPU-FPGA, oparty na standardowych protokołach, służy do komunikacji pomiędzy CPU a FPGA. Przykładowo rodzina Cyclone V wykorzystuje protokół interfejsu AMBA AXI 3 do komunikacji pomiędzy wbudowanym procesorem a FPGA.

1.2.6. Wykorzystanie układów programowalnych do budowy systemów wbudowanych

Układy PLD charakteryzują się ograniczoną mocą funkcjonalną i są wykorzystywane do implementacji bardzo prostych systemów wbudowanych. Ich główną zaletą jest relatywnie niski koszt.

Układy CPLD doskonale sprawdzają się w implementacji automatów skończonych oraz różnego rodzaju kontrolerów. Ich przewagą nad FPGA jest to, że są programowane na stałe, co oznacza, że są gotowe do pracy natychmiast po włączeniu zasilania. Dlatego w bardziej złożonych systemach wbudowanych układy CPLD są często montowane na płytce obok FPGA, aby zarządzać ich procesem konfiguracji po włączeniu zasilania.

Układy FPGA wyróżniają się największą elastycznością i różnorodnością zastosowań. Nawet w obrębie jednej rodziny istnieją podrodziny układów FPGA, które są specjalizowane w konkretnych obszarach zastosowań, takich jak:

- cyfrowe przetwarzanie sygnałów;
- telekomunikacja;
- zastosowania ogólne.

Układy SoC znajdują zastosowanie w systemach wbudowanych, gdzie obecność procesora jest niezbędna. W bardzo złożonych systemach wbudowanych układy CPLD, FPGA, SoC i mikroprocesor mogą być zainstalowane na tej samej płytce jednocześnie.

W książce omówiono różne zagadnienia związane z projektowaniem systemów wbudowanych opartych na FPGA. Istotność tych zagadnień wynika z faktu, że FPGA mogą stanowić samodzielną bazę elementów systemów wbudowanych, jak również mogą być integralną częścią układu SoC.

1.3. Podsumowanie rozdziału

System wbudowany to wyspecjalizowany rodzaj komputera, który został zaprojektowany do rozwiązywania jednego lub więcej specyficznych zadań i jest bezpośrednio zintegrowany z kontrolowanym obiektem.

Główne obszary zastosowań systemów wbudowanych to: elektronika użytkowa, urządzenia komputerowe i biurowe, systemy transportowe, systemy telekomunikacyjne, robotyka.

Funkcje systemów wbudowanych:

- sterowanie urządzeniami peryferyjnymi;
- odbieranie i przetwarzanie sygnałów ze źródeł zewnętrznych;
- generowanie sygnałów sterujących;
- implementacja interfejsów urządzeń peryferyjnych.

Cechy nowoczesnych systemów wbudowanych:

- szybkie przetwarzanie dużych ilości danych;
- szybka zmiana algorytmów przetwarzania informacji;
- możliwość dodawania nowych funkcji przetwarzania informacji.

Specyficzne wymagania dla niektórych systemów wbudowanych obejmują:

- konieczność pracy w czasie rzeczywistym;
- minimalne zużycie energii i rozmiar;
- zdolność do pracy w trudnych warunkach środowiskowych.

Podstawowe komponenty do projektowania systemów wbudowanych:

- mikrokontrolery;
- mikroprocesory;
- mikrokomputery;
- komputery jednopłytkowe;
- płyty główne do komputerów osobistych;
- układy scalone z logiką programowalną;
- układy SoC;
- wyspecjalizowane układy scalone ASIC;
- gotowe płyty komputerowe.

Istotną cechą układów FPGA jest możliwość rozwiązywania zadań systemów wbudowanych w sposób sprzętowy zamiast wykonywania sekwencji poleceń programowo. Szybkość rozwiązywania problemów przy realizacji systemu wbudowanego na FPGA jest o kilka rzędów wielkości większa w porównaniu z mikrokontrolerami i mikroprocesorami. Układy FPGA wyróżniają się również niskim poborem mocy oraz kompaktowymi rozmiarami.

Trendy rozwojowe we współczesnej architekturze FPGA skierowane są ku szerokiemu wykorzystaniu bloków cyfrowego przetwarzania sygnałów (DSP), wbudowanych pamięci, nadajników-odbiorników, kontrolerów popularnych interfejsów i innych komponentów.

Systemy wbudowane mają zróżnicowany zestaw urządzeń peryferyjnych, który przewyższa ten dostępny w mikrokontrolerach i mikroprocesorach.

Ponieważ wymagania dotyczące niezawodności systemów wbudowanych są wyższe niż w przypadku komputerów osobistych, programy przeznaczone dla tych systemów są opracowywane i testowane z większą dokładnością. W systemach wbudowanych zaleca się unikanie stosowania ruchomych elementów, takich jak dyski twarde, przełączniki i przyciski.

Poprawa niezawodności systemów wbudowanych wymaga zastosowania różnych metod. Konieczne są dokładne planowanie, konstruowanie i testowanie we wszystkich fazach projektowania. Kluczowymi etapami są symulacja komputerowa, testy fizyczne na płycie prototypowej oraz współdziałanie z kontrolowanym obiektem.

Metodologia projektowania systemów wbudowanych opartych na FPGA obejmuje następujące podstawowe etapy:

- specyfikacja systemu wbudowanego;
- rozdzielenie systemu wbudowanego na oprogramowanie i sprzęt;
- opis części sprzętowej;
- opis części programowej;
- specyfikacja połączeń;
- symulacja części sprzętowej;
- symulacja części programowej;
- wspólna symulacja całego systemu;
- fizyczne testowanie systemu.

Część sprzętowa systemu wbudowanego może być opisana na różnych poziomach szczegółowości:

- systemowy;
- przesłań międzyrejestrowych (RTL);
- bramek;
- tranzystorów.

Ważną cechą charakterystyczną układów programowalnych jest ich zdolność do zmiany struktury w celu realizacji funkcji, które są definiowane przez użytkownika. Może on samodzielnie programować lub konfigurować układy programowalne bez potrzeby angażowania specjalistycznych organizacji.

Programowalne układy logiczne są zwyczajowo dzielone na następujące klasy:

- PLD (*Programmable Logic Device*);
- CPLD (*Complex PLD*);
- FPGA (*Field Programmable Gate Array*);
- SoC (*System-on-Chip*).

Układy PLD obejmują następujące klasy:

- PLA (*Programmable Logic Array*);
- PROM (*Programmable Read Only Memory*);
- PAL (*Programmable Array Logic*).

Podstawą struktury PLA są dwie programowalne macierze: AND i OR. W układzie PROM macierz AND ma stałą konfigurację do realizacji pełnego dekodera. Z kolei w strukturze PAL macierz ta jest programowalna, a stałą konfigurację ma macierz OR.

Makrokomórki uniwersalnego układu PAL zawierają bramkę OR (część macierzy OR), przerzutnik, multiplexer MUX, bufor trójstanowy oraz jeden lub dwa obwody sprzężenia zwrotnego.

Struktura CPLD jest zbiorem *bloków funkcjonalnych*, które są połączone *matrycą przełączającą* SM. Każdy blok funkcjonalny ma architekturę uniwersalnego PAL.

Podstawową różnicą pomiędzy FPGA a CPLD jest to, że głównym elementem funkcyjnym FPGA nie jest para programowalnych macierzy AND–OR, lecz mały element pamięciowy nazywany *tablicą LUT* lub *generatorem funkcyjnym LUT*.

Uogólniona struktura FPGA zawiera macierz *elementów logicznych* LE, zbudowanych na bazie generatorów funkcyjnych LUT, które są rozmieszczone pomiędzy *polem połączeń*. Składa się ono z zestawu pionowych i poziomych linii kanałów połączeniowych, które mogą obejmować całą szerokość (lub wysokość) urządzenia FPGA bądź być skonfigurowane jako pojedyncze segmenty o zróżnicowanej długości. Małe macierze przełączające SM służą do przesyłania sygnałów pomiędzy kanałami, a na obrzeżach układu FPGA znajdują się bloki wejścia/wyjścia (IOB), które zarządzają wyprowadzeniami dwukierunkowymi. W strukturze FPGA obecne są również pętle synchronizacji fazowej PLL.

FPGA zawiera znaczną liczbę elementów logicznych połączonych ze sobą, co umożliwia większą elastyczność w implementacji systemów wbudowanych w porównaniu z CPLD. Konkretny układ FPGA, na którym ma być zaimplementowany projekt systemu wbudowanego, nazywany jest *docelowym* FPGA.

SoC to układ FPGA z wbudowanym procesorem, który ma zestaw urządzeń peryferyjnych oraz własne wyprowadzenia zewnętrzne, niepokrywające się z wyprowadzeniami FPGA. Do komunikacji między procesorem a FPGA wykorzystywany jest wewnętrzny interfejs CPU-FPGA, który oparty jest na standardowych protokołach.

Układy CPLD sprawdzają się dobrze w implementacji automatów skończonych i kontrolerów. W systemach wbudowanych układy te są często montowane na jednej płycie z FPGA, aby kontrolować proces konfiguracji FPGA po włączeniu zasilania.

Układy FPGA są powszechnie wykorzystywane w projektowaniu systemów wbudowanych. Często w obrębie jednej rodziny układów FPGA istnieją różne podrodziny, które specjalizują się w konkretnych obszarach zastosowań, takich jak:

- cyfrowe przetwarzanie sygnałów;
- telekomunikacja;
- ogólne zastosowania.

Układy SoC stosuje się w systemach wbudowanych, gdzie obecność procesora jest niezbędna.

W bardzo złożonych systemach wbudowanych układy CPLD, FPGA, SoC i mikroprocesor mogą być zainstalowane na tej samej płytce jednocześnie.

2. Język opisu sprzętu Verilog

Język Verilog został opracowany przez Phila Moorby'ego oraz Prabhu Goela w latach 1983–1984 jako narzędzie do opisu sprzętu. Jego składnia bazuje na języku programowania C. Początkowo język Verilog był przeznaczony głównie do symulacji układów cyfrowych. Symulator Verilog XL zdobył ogromną popularność wśród projektantów systemów cyfrowych. W 1987 roku firma Synopsys rozpoczęła wykorzystywanie języka Verilog również do opisu projektów systemów cyfrowych. W 1995 roku został ustanowiony pierwszy standard języka Verilog – IEEE 1364-1995. W 2001 roku wydano kolejną wersję standardu – IEEE 1364-2001 [3], która wprowadziła szereg istotnych ulepszeń do języka Verilog i została nazwana Verilog-2001.

Język Verilog służy zarówno do syntezy systemów cyfrowych, jak i do symulacji ich działania. W niniejszym podrozdziale przedstawione zostaną elementy i konstrukcje języka Verilog-2001 stosowane do syntezy projektów systemów wbudowanych.

2.1. Moduły

Moduł jest podstawową konstrukcją języka Verilog. Wszystkie modele układów cyfrowych, projekty poszczególnych bloków systemów cyfrowych, a nawet same systemy są reprezentowane za pomocą modułów. Kojarzone są one z poszczególnymi komponentami projektu lub z funkcjami w językach programowania.

Moduł powinien być najpierw opisany, a dopiero potem można go wykorzystać, tworząc jego *instancje*. Dla każdej instancji należy określić punkty podłączenia wejść i wyjść. Jeśli projekt składa się z jednego modułu, można pominąć tworzenie jego instancji.

2.1.1. Deklarowanie modułów

Składnia opisu modułu przyjmuje następującą formę:

```
module module_name
    #( parameter_declaration, parameter_declaration, ...)
    ( port_declaration port_name, port_name, ...,
```

```
port_declaration port_name, port_name, ...);
module items
```

endmodule

Deklaracja modułu zawsze rozpoczyna się od słowa kluczowego **module** i kończy się słowem **endmodule**. Po słowie kluczowym **module** umieszczona jest etykieta *module_name* definiowana przez użytkownika. Dalej w nawiasach okrągłych wymienione są parametry modułu. Ich lista poprzedzona jest znakiem # (dla rozróżnienia spisu parametrów od spisu portów modułu). Po spisie parametrów, także w nawiasach okrągłych, wymieniane są porty modułu. Lista portów kończy się średnikiem, po którym następuje opis elementów modułu (*module items*).

W powyższym formacie deklaracja portów odbywa się poprzez umieszczenie ich na liście portów modułu (w nawiasach okrągłych). Starsza forma deklaracji modułu pozwala na opis portów w ciele modułu, jak to pokazano poniżej:

```
module module_name (port_name, port_name, ...);
    port_declaration port_name, port_name, ... ;
    port_declaration port_name, port_name, ... ;
    module items
```

endmodule

Lista parametrów jest elementem opcjonalnym w deklaracji modułu. Jeśli moduł nie ma żadnych parametrów, spis portów następuje bezpośrednio po etykiecie. Jednocześnie lista portów również nie jest wymagana; na przykład moduł najwyższego poziomu, opisujący współpracę układu cyfrowego lub systemu z otoczeniem, może nie mieć portów.

2.1.2. Elementy modułów

Jako elementy modułu (*module items*) mogą być stosowane następujące konstrukcje języka Verilog:

- deklaracja typów danych;
- deklaracja parametrów;
- instancje (*instances*) modułów;
- instancje prymitywów;
- bloki generujące (*generate blocks*);
- bloki proceduralne (*procedural blocks*);
- przypisania ciągłe (*continuous assignments*);
- zadania (*tasks*);
- funkcje;
- bloki specyfikacji (*specify blocks*).

Elementy modułu mogą się pojawiać w kodzie w dowolnej kolejności. Jedynym wyjątkiem jest deklaracja typów danych i parametrów, które powinny występować przed ich pierwszym użyciem.

2.1.3. Behawioralny oraz strukturalny styl opisu projektu

W języku Verilog istnieją dwa główne style tworzenia modułów: behawioralny (*behavioral*) i strukturalny (*structural*). Styl behawioralny opisu (nazywany także funkcjonalnym bądź algorytmicznym) pozwala przedstawiać funkcjonowanie oraz zachowanie systemu bez uwzględniania jego budowy i szczegółów realizacji. Styl strukturalny opisuje instancje modułów i prymitywów oraz połączenia pomiędzy nimi. Do przekazywania sygnałów między komponentami struktury mogą być wykorzystywane zmienne wewnętrzne. Za pomocą stylu strukturalnego schematy logiczne, strukturalne oraz ideowe są opisywane w sposób zbliżony do edytorów graficznych. Dopuszcza się też przenikanie się stylów opisu modułu behawioralnego i strukturalnego.

2.1.4. Przykład projektowania jednobitowego sumatora

Działanie jednobitowego sumatora można zaprezentować w postaci *tablicy prawdy*, przedstawionej w tabeli 2.1, gdzie *a* i *b* reprezentują pojedyncze bity słów *A* i *B*, *cin* to bit przeniesienia z poprzedniej pozycji, *s* to bit sumy, a *cout* to przeniesienie na kolejną pozycję.

TABELA 2.1. Tablica prawdy sumatora jednobitowego

Wejścia			Wyjścia	
<i>cin</i>	<i>a</i>	<i>b</i>	<i>cout</i>	<i>s</i>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

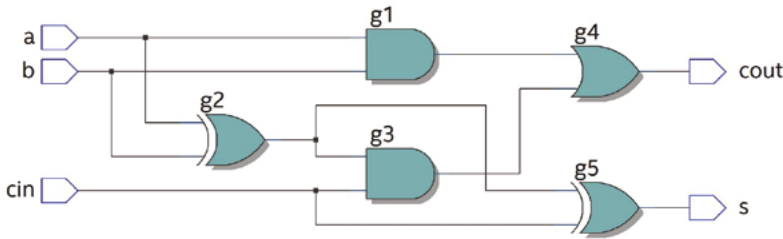
Na podstawie tabeli 2.1 można stworzyć następujące równania logiczne funkcji wyjściowych:

$$\begin{aligned}
 s &= \overline{cin} \cdot \overline{a} \cdot b + \overline{cin} \cdot a \cdot \overline{b} + cin \cdot \overline{a} \cdot \overline{b} + cin \cdot a \cdot b \\
 cout &= \overline{cin} \cdot a \cdot b + cin \cdot \overline{a} \cdot b + cin \cdot a \cdot \overline{b} + cin \cdot a \cdot b
 \end{aligned}
 \tag{2.1}$$

które w wyniku minimalizacji i przekształceń logicznych można zaprezentować w następującej postaci:

$$\begin{aligned} s &= a \oplus b \oplus cin \\ cout &= a \cdot b + (a \oplus b) \cdot cin \end{aligned} \quad (2.2)$$

gdzie znak $\oplus$ oznacza operację logicznej alternatywy wykluczającej (XOR). Przewagą równań (2.2) nad równaniami (2.1) jest to, że mogą być realizowane za pomocą bramek dwuwejściowych (rysunek 2.1).



RYS. 2.1. Schemat jednobitowego sumatora na poziomie bramek logicznych

Poniższy kod przedstawia strukturalny opis jednobitowego sumatora z rysunku 2.1 w języku Verilog:

```

module add_1_1 (input cin, a, b,    // opis portów wejściowych
                output s, cout);    // opis portów wyjściowych

    /* inicjalizacja */
    wire g1_o, g2_o, g3_o;          // zmienne wewnętrzne

    /* operatory */
    and g1 (g1_o, a, b);             // pierwsza bramka AND
    xor g2 (g2_o, a, b);             // pierwsza bramka XOR
    and g3 (g3_o, g2_o, cin);        // druga bramka AND
    or g4 (cout, g1_o, g3_o);        // bramka OR
    xor g5 (s, g2_o, cin);           // druga bramka XOR
endmodule

```

Po słowie kluczowym **module** następuje nazwa projektu *add_1_1*, po której w nawiasie wymienione są porty modułu: wejściowe (**input**) i wyjściowe (**output**).

Jeżeli do opisu projektu wymagane są dodatkowe zmienne, należy je zainicjalizować przed ich pierwszym użyciem. W przedstawionym przykładzie są to węzły (**wire**) odpowiadające wyjściom poszczególnych bramek: *g1_o*, *g2_o* i *g3_o*. Właściwie utworzony opis projektu składa się z opisu każdej bramki z rysunku 2.1. Bramki logiczne

odpowiadają prymitywom języka Verilog, więc mogą być stosowane bez wcześniejszej inicjalizacji czy opisu.

Opis każdej pojedynczej bramki składa się z nazwy jej typu (**and**, **or**, **xor**), nazwy instancji (*g1*, *g2*, *g3*, *g4*, *g5*) oraz spisu portów danej instancji. Spis portów przedstawia wyprowadzenia konkretnej instancji bramki i jednocześnie określa schemat łączenia elementów. Charakterystyczną cechą opisu bramek logicznych jest to, że w spisie portów sygnał wyjściowy znajduje się na pierwszym miejscu, następnie zaś są wymienione wejścia. Taka organizacja portów jest w pełni zrozumiała, ponieważ bramki mogą mieć dowolną liczbę wejść, lecz tylko jedno wyjście. Do przekazywania sygnałów pomiędzy bramkami wykorzystuje się zainicjalizowane wcześniej dodatkowe zmienne wewnętrzne *g1_o*, *g2_o* i *g3_o*. Zauważmy, że skalarne (jednobitowe) wewnętrzne połączenia w języku Verilog nie muszą być inicjalizowane, gdyż kompilator robi to automatycznie.

Opis strukturalny dokładnie odzwierciedla graficzną prezentację projektu, co czasem utrudnia dostrzeżenie zalet korzystania z języka projektowania zamiast edytora graficznego. Dlatego też powyższy styl stosuje się bardzo rzadko do opisu projektu.

Ten sam projekt można również opisać za pomocą równań logicznych (2.2):

```
module add_1_2 (input cin, a, b,           // opis portów wejściowych
               output s, cout);          // opis portów wyjściowych

    assign s = a ^ b ^ cin;               // opis funkcji sumy
    assign cout = a & b | (a ^ b) & cin;   // opis funkcji przeniesienia
endmodule
```

W powyższym kodzie do opisu projektu jednobitowego sumatora wykorzystano instrukcje przypisania ciągłego **assign**. Instrukcje te są zawsze wykonywane równolegle, niezależnie od ich umieszczenia w kodzie projektu. Z prawej strony znaku równości przedstawiono wyrażenia korzystające z operacji logicznych, które są oznaczone symbolami: | – OR, & – AND oraz ^ – XOR. Warto zauważyć, że negację oznacza się za pomocą znaku ~. Taki styl opisu jest najczęściej stosowany do projektowania układów kombinacyjnych na podstawie *równań logicznych*. Realizację modułu *add_1_2* pokazano na rysunku 2.2. Można zauważyć, że układy z rysunków 2.1 oraz 2.2 są identyczne, chociaż zostały przedstawione za pomocą różnych stylów opisu.

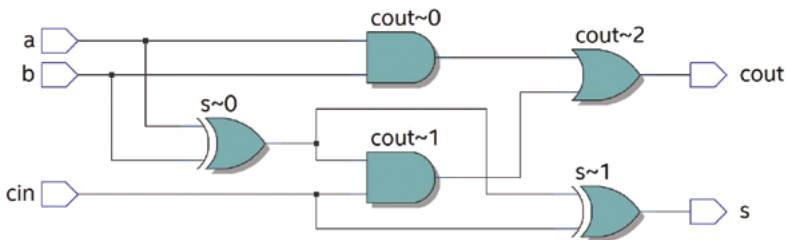
Język Verilog umożliwia także opisywanie projektów na poziomie behawioralnym (zachowania), co polega na przedstawieniu algorytmu funkcjonowania modułu za pomocą *instrukcji proceduralnych*. Aby opisać przykładowy projekt jednobitowego sumatora na poziomie zachowań, należy określić reguły jego działania. Na przykład funkcja przeniesienia *cout* będzie równa 1, jeżeli na wejściach sumatora jednocześnie znajdują się co najmniej dwie wartości 1. W takiej sytuacji opis jednobitowego sumatora może być następujący:

```

module add_1_3 (input cin, a, b,                // opis portów wejściowych
                output reg s, cout);           // opis portów wyjściowych

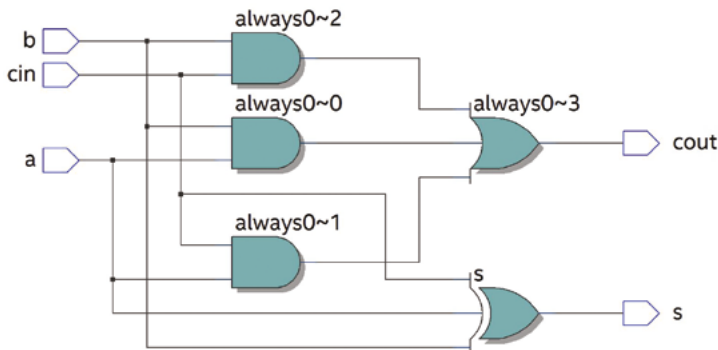
    always @(cin, a, b) begin
        if (a & b | cin & a | cin & b) cout=1; // opis funkcji przeniesienia
        else cout=0;
        s = a ^ b ^ cin;                       // opis funkcji sumy
    end
endmodule

```



RYS. 2.2. Realizacja modułu *add_1_2*: przykład opisu sumatora jednobitowego za pomocą równań logicznych

W przedstawionym kodzie wykorzystano instrukcję proceduralną **if-else** oraz instrukcję blokowanego przypisania (=). Dodatkowo dla funkcji wyjściowych *s* i *cout* został wykorzystany typ zmiennych **reg**. Zgodnie ze standardem języka Verilog zmienne zapisane po lewej stronie znaku równości w instrukcjach proceduralnych przypisania powinny mieć typ **reg**. Realizacja modułu *add_1_3* została przedstawiona na rysunku 2.3.



RYS. 2.3. Realizacja modułu *add_1_3*: przykład opisu jednobitowego sumatora za pomocą instrukcji proceduralnych

Kolejny przykład opisu jednobitowego sumatora prezentuje stary styl deklaracji modułu, w którym porty modułu są opisywane w jego ciele:

```
module add_1_4 (cin, a, b, s, cout); // deklaracja nagłówka modułu
```

```
    input cin, a, b;           // deklaracja portów wejścia  
    output s, cout;          // deklaracja portów wyjścia
```

```
    assign s = a ^ b ^ cin;  
    assign cout = a & b | (a ^ b) & cin;
```

```
endmodule
```

2.1.5. Deklaracja portów

Deklaracja portów modułu ma następującą składnię:

```
port_direction data_type signed range port_name, port_name, ...,
```

gdzie: *port_direction* – kierunek przekazywania sygnału, *data_type* – typ przekazywanych danych, **signed** – słowo kluczowe, *range* – specyfikacja zakresu pola bitowego, *port_name* – nazwa portu.

Stara notacja deklaracji portów przybiera następującą formę:

```
port_direction signed range port_name, port_name, ...;  
data_type_declarations
```

Kierunek przekazywania sygnału (*port_direction*) może przyjmować następujące wartości:

- **input** – dla portów wejściowych;
- **output** – dla portów wyjściowych;
- **inout** – dla portów dwukierunkowych.

Jako typ przekazywanych danych (*data_type*) można używać dowolnego typu danych z języka Verilog z wyjątkiem typu **real**. Wartość *port_direction* ogranicza typy danych przekazywanych za pośrednictwem określonego portu. Tylko sieciowe typy danych (*net*) mogą być przypisane do portów wejściowych oraz dwukierunkowych. Na portach wyjściowych mogą być generowane dowolne typy danych języka Verilog z wyjątkiem typu **real**.

Słowo kluczowe **signed** wskazuje, że wartości przekazywane przez port należy interpretować jako liczbę ze znakiem (*signed value*) w kodzie uzupełnień do 2. Jeżeli port lub typ danych wewnętrznego sygnału podłączonego do portu deklarowane są jako **signed**, to oba stają się wartościami ze znakiem.

Specyfikacja zakresu pola bitowego (*range*) portu ma następujący format:

```
[msb:lsb],
```

gdzie: *msb* – numer najbardziej znaczącego bitu, *lsb* – numer najmniej znaczącego bitu.

Jeżeli specyfikacja zakresu nie jest określona, to port ma szerokość jednego bitu, a przekazywane sygnały są traktowane jako wartości skalarne. W przeciwnym zaś wypadku jako wartości wektorowe.

Numerami najstarszego (*msb*) oraz najmłodszego (*lsb*) bitu mogą być: liczby, stałe, wyrażenia oraz odwołania do funkcji zwracających wartości stałe. Specyfikacja zakresu dopuszcza zarówno rosnący (*big-endian*), jak i malejący (*little-endian*) porządek wymiany bitów (na przykład [0:7] oraz [8:1] – dopuszczalne rozmiary zakresu). Maksymalny rozmiar zakresu bitów portu jest ograniczony wartością 256 bitów. Jednakże większość kompilatorów języka Verilog pozwala na liczbę bitów portu do miliona.

Rozmiary zakresu typu danych i typu portu powinny mieć jednakowe wartości. Jeśli się od siebie różnią, niektóre kompilatory mogą przyjąć rozmiar typu danych bez wyświetlenia ostrzeżenia lub błędu. Poniżej zaprezentowano przykłady deklaracji portów:

```
input a, b, enable;           // trzy porty skalarne
input signed [15:0] m1, m2;   /* dwa 16-bitowe porty, które przekazują dane
                               w kodzie uzupełnień do 2;
                               porządek wymiany – little endian */
output signed [31:0] wyn;     /* podobnie jak poprzednio,
                               jednak jest to 32-bitowy port */
output reg signed [32:1] sum; /* 32-bitowy port, podłączone do portu
                               wewnętrzne sygnały mogą przyjmować
                               wartości typu reg;
                               porządek wymiany – little endian */
inout [0:15] mag;            /* dwukierunkowy 16-bitowy port
                               z porządkiem wymiany big endian */
input [15:12] adres;         /* 4-bitowy wektor adres wykorzystuje się
                               jako wejście (msb oraz lsb mogą mieć
                               dowolne wartości) */

parameter WORD = 32;
input [WORD-1:0] adres;      /* deklaracja portu może zawierać stałe */
parameter SIZE=4096;
input [log2(SIZE)-1:0] adr;  /* deklaracja portu może odwoływać się
                               do stałych funkcji */
```

2.1.6. Instancje modułów

Zanim będzie można wykorzystywać zdefiniowany moduł jako część systemu cyfrowego, należy stworzyć jego instancję (*instance*). Jest ona w pewnym sensie odpowiednikiem wywołania funkcji w językach programowania. W programowaniu wywołanie funkcji uruchamia kod należący do danej funkcji, podobnie tworzenie instancji modułu oznacza umieszczenie w strukturze FPGA egzemplarza układu opisanego przez ten moduł.

Podobnie jak przy implementacji układu w strukturze FPGA należy podłączyć go do poszczególnych wyprowadzeń, również podczas tworzenia instancji modułu konieczne jest wskazanie sygnałów przyporządkowanych odpowiednim portom.

2.1.7. Przekazywanie sygnałów do instancji modułów

Istnieją dwa sposoby wskazania połączenia sygnałów z portami instancji: za pomocą lokalizacji oraz za pomocą nazwy portu.

Format tworzenia instancji modułu, gdy sygnał jest przekazywany za pomocą lokalizacji, wygląda następująco:

```
module_name instance_name (signal, signal, ...);
```

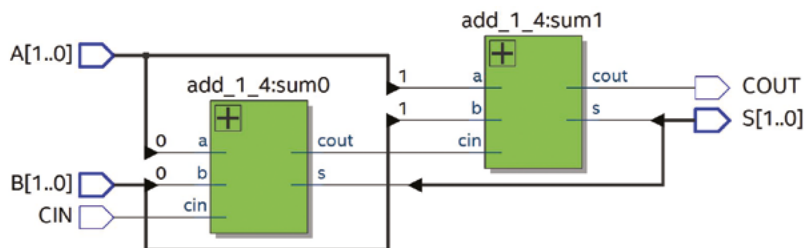
gdzie: *module_name* – nazwa modułu, *instance_name* – nazwa instancji modułu; (*signal, signal, ...*) – lista podłączanych sygnałów.

W przypadku gdy niektóre porty instancji modułu nie są podłączone, oznacza się to na liście poprzez umieszczenie dwóch przecinków z rzędu. Technika ta nazywana jest *pomijaniem sygnału*. Jeżeli konieczne jest pominięcie kilku sygnałów, wówczas w spisie sygnałów umieszcza się kilka przecinków.

Na przykład projekt dwubitowego sumatora składającego się z dwóch instancji sumatora jednobitowego można przedstawić następująco:

```
module add_2 (input wire [1:0] A, B, input CIN,
              output wire [1:0] S, output COUT);
    wire C0;          /* przeniesienie */
    add_1_4 sum0 (A[0], B[0], CIN, S[0], C0);
    add_1_4 sum1 (A[1], B[1], C0, S[1], COUT);
endmodule
```

W danym przykładzie *add_1_4* jest nazwą opisanego wcześniej modułu sumatora jednobitowego, natomiast *sum0* oraz *sum1* są nazwami dwóch instancji tego modułu. Warto zauważyć, że w danym projekcie do przekazu sygnałów między dwoma instancjami sumatora jednobitowego wykorzystuje się wewnętrzny węzeł *C0*. Realizacja modułu *add_2* została przedstawiona na rysunku 2.4.



RYS. 2.4. Realizacja modułu *add_2* dwubitowego sumatora stworzonego z dwóch sumatorów jednobitowych

Format tworzenia instancji modułu, gdy sygnał przekazywany jest za pomocą nazwy portu, wygląda następująco:

```
module_name instance_name ( .port_name (signal), .port_name (signal), ...);
```

gdzie *port_name* – nazwa portu, któremu zostaje przekazany odpowiedni sygnał.

W danym formacie każda nazwa portu zaczyna się od kropki. Jeżeli do pewnego portu instancji modułu nie jest podłączony żaden sygnał, to nazwa tego portu zostaje pominięta w spisie portów instancji modułu. Przekazywanie sygnałów za pomocą nazwy portu wymaga więcej kodu, jednak nie potrzebuje od projektanta wiedzy o kolejności występowania portów w deklarowanym module. Ponadto ten sposób zmniejsza się liczbę błędów przy pisaniu kodu projektu, dlatego jest wykorzystywany on przy tworzeniu instancji modułów z dużą liczbą portów.

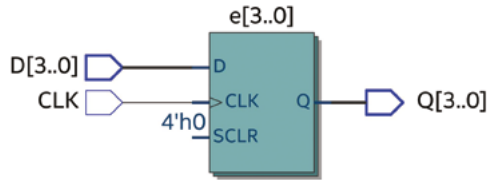
Jako przykład rozważmy projekt 4-bitowego rejestru opartego na prymitywie przerzutnika typu D **dff**. Nie jest znana kolejność występowania portów przerzutnika, jednak znane są ich nazwy: *d* – wejście danych; *q* – wyjście; *clk* – wejście sygnału synchronizacji.

```
module reg_4_1 (input wire [3:0] D,  
               input wire CLK,  
               output wire [3:0] Q);  
    dff e0 (.clk(CLK), .d(D[0]), .q(Q[0]));  
    dff e1 (.clk(CLK), .d(D[1]), .q(Q[1]));  
    dff e2 (.clk(CLK), .d(D[2]), .q(Q[2]));  
    dff e3 (.clk(CLK), .d(D[3]), .q(Q[3]));  
endmodule
```

W użytym przykładzie wykorzystano jedynie trzy porty prymitywu **dff**: *clk*, *d* oraz *q*. Kolejność portów oraz nazwy innych portów nie muszą być znane. Realizacja modułu *reg_4_1* przedstawiona została na rysunku 2.5.

W ogólnym przypadku instancje modułów mogą być tworzone poprzez wymienienie ich nazw oddzielonych od siebie przecinkami, bez powtarzania nazwy modułu, jak pokazano poniżej:

```
module reg_4_2 (input wire [3:0] D,  
               input wire CLK,  
               output wire [3:0] Q);  
    dff e0 (.clk (CLK), .d (D[0]), .q(Q[0])),  
    e1 (.clk (CLK), .d (D[1]), .q(Q[1])),  
    e2 (.clk (CLK), .d (D[2]), .q(Q[2])),  
    e3 (.clk (CLK), .d (D[3]), .q(Q[3]));  
endmodule
```



RYS. 2.5. Realizacja modułów *reg_4_1*, *reg_4_2* oraz *reg_4_3*

Nazwa instancji modułu nie jest wymagana, oznacza to, że można stworzyć instancję projektu bez wskazania nazwy instancji modułów:

```

module reg_4_3 (input wire [3:0] D,
               input wire CLK,
               output wire [3:0] Q);
    dff      (.clk (CLK), .d (D[0]), .q(Q[0]),
             (.clk (CLK), .d (D[1]), .q(Q[1]),
             (.clk (CLK), .d (D[2]), .q(Q[2]),
             (.clk (CLK), .d (D[3]), .q(Q[3]));
endmodule

```

2.1.8. Parametry modułu

Parametry modułów pozwalają znacznie zwiększyć elastyczność korzystania z nich: ten sam moduł może być użyty wielokrotnie do tworzenia instancji modułu z różnymi parametrami. Wykorzystanie parametrów często skraca kod projektu, co zwiększa jego czytelność i zmniejsza możliwość popełnienia błędów podczas pisania.

W deklaracji modułu lista parametrów rozpoczyna się znakiem # i poprzedza spis portów modułu. Na przykład opis parametryzowanego modułu sumatora może wyglądać następująco:

```

module add_N # ( parameter N = 4)           // deklaracja parametru N z domyślną
                                                //wartością 4
               (input [N-1: 0] a, input cin,   // deklaracja zakresu słów wejścia
               output [N-1: 0] s, output cout); // deklaracja zakresu słów wyjścia

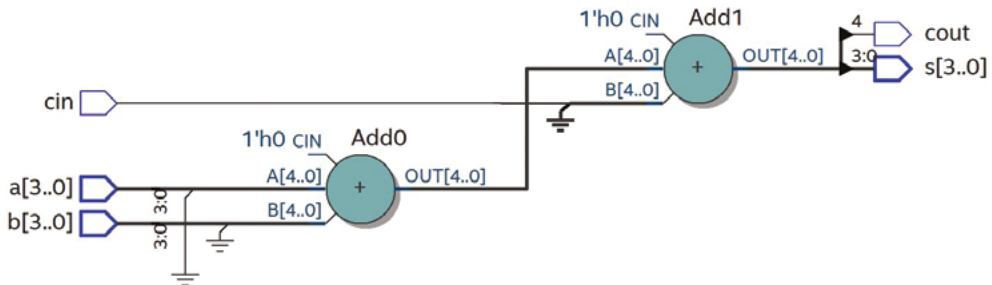
    assign {cout, s} = a + b + cin;
endmodule

```

W powyższym przykładzie parametr *N* wykorzystuje się do określenia zakresu słów wejściowych *a* i *b* oraz sumy *s*. Podczas deklaracji modułu można określać wartości domyślne poszczególnych parametrów, które są wykorzystywane, gdy przy tworzeniu instancji modułu dany parametr nie zostanie ustawiony. W podanym przykładzie

domyślna wartość parametru N wynosi 4. Realizacja modułu add_N przedstawiona została na rysunku 2.6.

W danym przykładzie wykorzystano także następujące instrukcje i operatory: przypisania ciągłego **assign**, konkatencji (sklejania bitów i pól bitowych) $\{ \}$ oraz sumy arytmetycznej $+$. Operator konkatencji znajdujący się po lewej stronie instrukcji **assign** jest niezbędny, gdyż w wyniku dodawania może się pojawić sygnał przeniesienia z najstarszego bitu sumy.



RYS. 2.6. Realizacja modułu add_N

W ogólnym przypadku moduł może mieć wiele parametrów, wtedy przy deklaracji modułu należy zamieścić listę wszystkich parametrów.

Stworzenie instancji parametryzowanego modułu również może mieć dwa formaty, pozycyjny oraz przez nazwę parametru:

```
module_name # (value, value, ...) instance_name (signal, ...);
module_name # ( .parameter_name (value), .parameter_name (value), ...)
instance_name (signal, ...);
```

gdzie: *value* – przekazywana wartość parametru dla konkretnej instancji modułu, *parameter_name* – nazwa parametru.

Jak można zauważyć, postacie przypisywania parametrów instancjom modułu w rzeczywistości są niemal identyczne jak formaty przekazywania sygnałów portom instancji modułu. Różnica wynika jedynie z obecności znaku $\#$ przed spisem ustawianych parametrów.

Poniższy kod demonstruje przykład stworzenia instancji 32-bitowego sumatora:

```
wire [31:0] A, B, S;
wire CIN, COUT;
add_N # (32) sum_32_v1 (A, B, CIN, S, COUT); // ustalenie parametrów według
// ich lokalizacji
add_N # (.N (32)) sum_32_v2 (A, B, CIN, S, COUT); // ustalenie parametrów
// według ich nazwy
```

2.1.9. Niejawne przekazywanie parametrów

W języku Verilog możliwa jest także niejawna deklaracja wartości parametrów instancji modułu. W tym celu wykorzystuje się instrukcję **defparam**, która ma następującą składnię:

defparam *hierarchy_path.parameter_name* = *value*;

gdzie *hierarchy_path* – hierarchiczna nazwa instancji modułu, *parameter_name* oraz *value* – nazwa oraz wartość odpowiedniego parametru.

Zaletą wykorzystania instrukcji **defparam** jest to, że może ona być używana w dowolnym miejscu kodu oraz może nadpisywać wartości parametrów dowolnej instancji modułu. Należy jednak uważać, stosując instrukcję **defparam**, gdyż jej wykorzystanie zmniejsza czytelność kodu.

Przykład wykorzystania instrukcji **defparam**:

```
add_N S_64 (Cin, A, B, S, Cout);    // utworzenie instancji modułu
                                   add_N o nazwie S_64
defparam S_64.N = 64;             // redefinicja wartości parametru N instancji S_64
```

2.1.10. Tablice instancji modułów

Oprócz tworzenia pojedynczych instancji modułów język Verilog pozwala na tworzenie tablic instancji modułów (*array of instances*). W danym przypadku składnia tworzenia instancji modułów wygląda następująco:

module_name instance_name instance_array_range (*signal, signal, ...*);

module_name instance_name instance_array_range

(*.port_name* (*signal*), *.port_name* (*signal*), ...);

gdzie: *module_name*, *instance_name*, *port_name* i *signal* – oznaczają odpowiednio nazwę modułu, nazwę instancji, nazwę portu i przekazywany sygnał; *instance_array_range* – zakres tablicy instancji.

Zakres tablicy instancji ma następującą składnię:

[*left_index* : *right_index*]

gdzie *left_index* i *right_index* – lewy i prawy indeks zakresu tablicy. Liczba tworzonych instancji modułów określa się wyrażeniem:

$\text{abs}(\text{left_index} - \text{right_index}) + 1$

gdzie *abs* – funkcja zwracająca wartość bezwzględną.

Jeżeli rozmiar portu modułu tablicy instancji jest równy szerokości przekazywanego do tego portu sygnału, wówczas ten sam sygnał jest podłączany do każdej instancji modułu. Warto zauważyć, że ta właściwość wykorzystywana jest do podłączenia sygnałów sterujących do wszystkich instancji modułu.

Jeżeli rozmiar portu modułu różni się od szerokości przekazywanego sygnału, wówczas port każdej instancji modułu otrzymuje jedynie część sygnału, począwszy od prawego indeksu portu instancji, który jest podłączany do prawej części wektora. Proces podłączania kolejnych modułów rozszerza się do lewej części wektora. Aby możliwe było podłączenie wszystkich instancji, suma bitów portów instancji powinna odpowiadać szerokości wektora (rozmiar portu należy pomnożyć przez liczbę instancji).

Zbiorowe instancje modułów mogą być także tworzone za pomocą bloku generowania kodu (*generate block*).

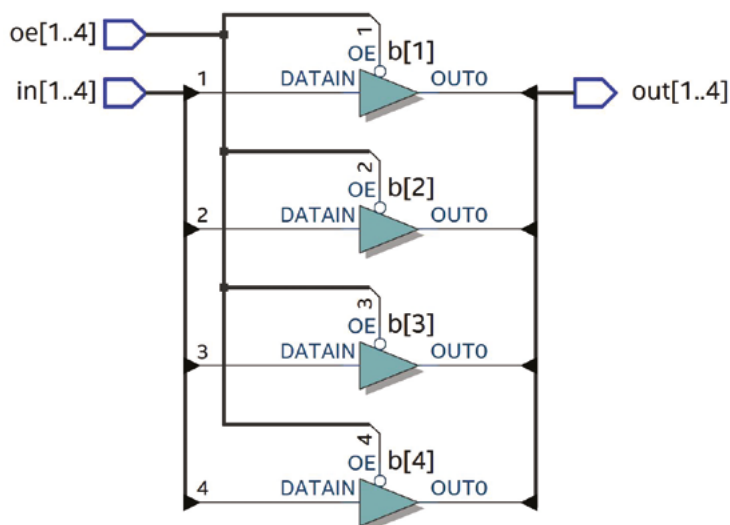
Poniższy kod przedstawia przykład tworzenia tablic instancji modułów przy projektowaniu 4-bitowego bufora składającego się z czterech jednobitowych prymitywów **bufif0**:

```
module buf_4_1 (input [1:4] in, oe,  
               output [1:4] out);  
  
    bufif0 b[1:4] (out, in, oe);    // stworzenie czterech instancji bufif0  
endmodule
```

Powyższy przykład jest równoważny następującemu fragmentowi kodu:

```
module buf_4_2    (input [1:4] in, oe,  
                 output [1:4] out);  
bufif0 (out[1], in[1], oe[1]),  
        (out[2], in[2], oe[2]),  
        (out[3], in[3], oe[3]),  
        (out[4], in[4], oe[4]);  
  
endmodule
```

Realizacja modułu *buf_4_1* została ukazana na rysunku 2.7.



RYS. 2.7. Realizacja modułu *buf_4_1*

2.1.11. Równoległość języka Verilog

W odróżnieniu od standardowych języków programowania język Verilog umożliwia równoległe wykonywanie swoich instrukcji. Do konstrukcji wykonywanych równoległe można zaliczyć:

- instancje modułów;
- instancje prymitywów;
- instrukcje przypisania ciągłego (**assign**);
- bloki proceduralne (**initial**, **always**).

Równoległe wykonywanie poszczególnych konstrukcji języka Verilog jest brane pod uwagę przez kompilator systemu Quartus podczas syntezy schematu projektu. W języku Verilog instancje prymitywów i modułów odpowiadają komponentom systemu, instrukcje przypisania pełnią funkcję połączeń, a bloki proceduralne są równoważne blokom funkcjonalnym urządzeń cyfrowych.

Równoległość języków projektowania doskonale odzwierciedla fizyczną naturę układów elektronicznych: po podaniu napięcia zasilania wszystkie komponenty schematu działają jednocześnie, a sygnały elektryczne przekazywane są równocześnie i równoległe przez złącza. Dodatkowo równoległe operacje w języku Verilog są niezwykle przydatne podczas symulacji urządzeń.

2.2. Podstawowe elementy języka Verilog

2.2.1. Identyfikatory

Identyfikatory języka Verilog służą jako nazwy dla zmiennych, modułów, funkcji, instancji modułów, instancji prymitywów itp. Mogą zawierać wielkie (A–Z) i małe (a–z) litery alfabetu łacińskiego, cyfry (0–9), znak podkreślenia (`_`) oraz znak dolara (`$`). Inne znaki ASCII mogą być również używane jako składowe identyfikatorów, jeżeli poprzedzone są znakiem *backslash* (`\`).

Identyfikatory powinny zaczynać się od litery lub znaku podkreślenia, a kończyć białym znakiem. Długość identyfikatora nie powinna przekraczać 1024 znaków. Verilog rozróżnia wielkość liter, więc etykiety `abc`, `Abc`, `ABC` są traktowane jako trzy różne identyfikatory.

Przykłady poprawnych identyfikatorów:

`add_1`, `g1`, `g_2`, `g_2_1`, `_adder`, `CLK`, `clk`, `XOR`.

Przykłady nieprawidłowych identyfikatorów:

`1_add`, `2012_project`, `$RESET`, `xor`.

Poprawny identyfikator `XOR` został zapisany wielkimi literami, natomiast identyfikator `xor` jest nieprawidłowy, ponieważ jest słowem kluczowym, co uniemożliwia jego wykorzystanie jako etykiety, gdyż wszystkie słowa kluczowe są pisane małymi literami.

2.2.2. Znaki białe

Znaki białe (*white space*) wykorzystuje się do formatowania opisu projektu celem zapewnienia czytelności kodu. Do białych znaków w języku Verilog zalicza się: spację, znak tabulacji, znak nowej linii (*carriage return*) oraz znak przejścia strony (*form-feed*). W praktyce białe znaki mogą znajdować się w dowolnym miejscu kodu. Jednak ciąg symboli zawartych pomiędzy cudzysłowem nie może zawierać znaku nowej linii.

2.2.3. Komentarze

W języku Verilog możliwe są dwa rodzaje komentarzy: dla pojedynczej linii kodu i dla większej ich liczby. Komentarz pojedynczej linii rozpoczyna się od dwóch znaków *slash* (`//`) i kończy się wraz ze znakiem nowej linii. Blok komentarza (podobnie jak w języku C) rozpoczyna się parą znaków `/*`, kończy zaś odwrotną parą znaków `*/`. Blok komentarza może obejmować kilka linii kodu, jednakże bloki komentarzy nie mogą się w sobie zawierać.

2.2.4. Sygnały, sieci, sterowniki

Ogólnie w języku Verilog nie istnieje pojęcie sygnału, chociaż jest to język powszechnie używany do projektowania układów cyfrowych, gdzie *sygnał* jest kluczowym elementem. Zamiast sygnałów w języku Verilog używa się pojęcia *sieci* (*nets*). Są one wykorzystywane do połączeń portów (wyprowadzeń) poszczególnych komponentów projektu. Głównymi komponentami projektu są moduły, ale mogą też nimi być prymitywy, funkcje i zadania, które również mogą mieć porty.

Sieć charakteryzuje się dwoma parametrami: *wartością logiczną* i *mocą logiczną sygnałów*. Wartość logiczna określa stan sygnału w sieci, który pochodzi ze źródeł sygnałów nazywanych *sterownikami* (*drivers*). Dlatego drugi parametr sieci określa logiczną moc (*strength*) sterownika, z którego sygnały docierają do sieci. Jedna sieć może zawierać kilka sterowników. Stąd wynika problem określenia znaczenia sygnału w sieci w zależności od logicznej mocy sterownika. Zwykle jest ona określana dla sygnału o wartości logicznej *jeden* (wysokiego) oraz *zero* (niskiego).

2.2.5. Wartości logiczne

W języku Verilog każdy bit (pojedynczy sygnał) może przyjmować cztery wartości:

- 0 – logiczne zero bądź *fałsz*;
- 1 – logiczna jedynka albo *prawda*;
- X – dowolny znak (to znaczy, że może być zarówno 0, jak i 1) bądź wartość nieokreślona (*don't care*);
- Z – stan wysokiej impedancji, stan trzeci bądź stan przejściowy (*floating*).

Gdy stan Z pojawia się na wejściu lub w wyrażeniu, rezultat będzie taki sam jak przy stanie X. Różnica między stanem X a sygnałem Z jest widoczna głównie dla sygnałów wyjściowych.

W języku Verilog dopuszczalne są jeszcze dwa dodatkowe stany logiczne, które są wykorzystywane jedynie do wewnętrznej symulacji i nie mogą być używane w opisie projektu:

- L – częściowo nieznan stan: albo 0, albo Z, jednak nie 1;
- H – częściowo nieznan stan: albo 1, albo Z, jednak nie 0.

2.2.6. Liczby całkowite

Do reprezentacji liczb w języku Verilog wykorzystuje się postać:

size'base value

gdzie: *size* – liczba dziesiętna określająca rozmiar pola bitowego do reprezentacji wartości, *'base* – symbol określający podstawę systemu liczbowego, w którym

reprezentowana jest wartość, *value* – wartość liczby. Do określenia podstawy systemu liczbowego korzysta się z następujących znaków:

- 'b albo 'B – dla systemu dwójkowego (*binary*);
- 'd albo 'D – dla systemu dziesiętnego (*decimal*);
- 'h albo 'H – dla systemu szesnastkowego (*hexadecimal*);
- 'o albo 'O – dla systemu ósemkowego (*octal*).

Ogólnie rozmiar (*size*) oraz symbol podstawy systemu (*base*) przy prezentacji liczby nie są obowiązkowe. Domyślnie przy zapisie liczby przyjmowany jest system dziesiętny (*base='D*) oraz rozmiar 32 bitów (*size=32*). Jednakże domyślne wartości systemu liczbowego i rozmiar liczby zależą od implementacji kompilatora.

Przykłady liczb całkowitych:

```
8'b01010011      // 8-bitowa liczba binarna
16'h0F2E          // 16-bitowa liczba szesnastkowa
157               // 32-bitowa liczba dziesiętna (domyślnie)
```

Jeżeli binarna postać liczby przekracza wcześniej zadeklarowany rozmiar (*size*), starsze bity, które przekraczają zadany rozmiar, zostają odrzucone. Natomiast w przypadku gdy postać binarna liczby jest mniejsza niż zadeklarowany rozmiar zmiennej, młodsze bity zostają uzupełnione, a starsze wyzerowane. Istnieją wyjątki, kiedy najstarszym bitem jest **Z** lub **X**, wtedy pozostałe bity są uzupełniane odpowiednio **Z** lub **X**. Przykłady bitowej reprezentacji liczb całkowitych zaprezentowano w tabeli 2.2.

TABELA 2.2. Przykłady reprezentacji liczb całkowitych

Liczba	Reprezentacja binarna	Uwagi
12	0...01100	32 bity
'o5	0...00101	32 bity
1'b1	1	
1'b0	0	
8'Hd5	11010101	
6'hF2	110010	odrzucone starszych bitów
6'hA	001010	uzupełnienie zerami starszych bitów
8'b0	00000000	
8'b1	00000001	
8'bx	xxxxxxxx	
8'b1x	0000001x	
8'b0x	0000000x	
8'hx	xxxxxxxx	
8'hzx	zzzzxxxx	
8'hz1	zzzz0001	
8'bx1	xxxxxxxx1	
8'bx0	xxxxxxxx0	
8'hz	zzzzzzzz	
8'h0z	0000zzzz	

Zamiast symbolu **Z** w prezentacji liczby można używać znaku zapytania (?). Dla zwiększenia czytelności przy prezentacji dużych liczb można używać symbolu podkreślenia (), na przykład:

16'b0001_1011_1010_1100 // binarna 16-bitowa liczba

Podczas kompilacji znak podkreślenia zostaje zignorowany, jednak nie może być on pierwszym znakiem liczby. Liczby ujemne przedstawia się w kodzie dopełnień do dwóch.

2.2.7. Liczby rzeczywiste

Do zapisu liczb rzeczywistych w języku Verilog wykorzystuje się dwa formaty: notację dziesiętną (z kropką pomiędzy częścią całkowitą a ułamkiem) oraz notację wykładniczą.

Format zawierający kropkę wygląda następująco:

value_i.value_f

gdzie: *value_i* – część całkowita, *value_f* – część dziesiętna.

Notacja wykładnicza liczby przyjmuje następujący format:

baseeexponent lub

baseEexponent

gdzie: *base* – podstawa liczby, *exponent* – wykładnik.

Przy reprezentacji dziesiętnej cyfry powinny znajdować się po obu stronach kropki dziesiętnej. W notacji wykładniczej nie może być spacji ani przed, ani po znaku **e** bądź **E**.

Przykłady zapisu liczb rzeczywistych:

0.7

2e4=2 · 10⁴=20000

1.8E-2=1.8 · 10⁻²=0,018

2.2.8. Prymitywy języka Verilog

Wbudowane w język prymitywy można podzielić na dwie kategorie: bramki oraz przełączniki. Kategoria prymitywów bramkowych oprócz samych bramek zawiera także bufory. Opis prymitywów bramkowych znajduje się w tabeli 2.3. Ich cechą charakterystyczną jest to, że w liście portów pierwszy port jest wyjściem bramki, a następnie umieszczone są porty wejściowe bramki.

TABELA 2.3. Bramkowe prymitywy języka Verilog

Nazwa	Cechy
<i>and, nand, or, nor, xor, xnor</i>	jedno wyjście, jedno wejście bądź więcej wejść
<i>buf, not</i>	jedno wyjście bądź więcej wyjść, jedno wejście
<i>bufif0, bufif1, notif0, notif1</i>	jedno wejście, jedno wyjście, jedno wejście sterujące
<i>pullup, pulldown</i>	jedno wyjście, jedno wejście bądź więcej wejść

Kategoria przełączników zawiera różne rodzaje tranzystorów. Może się wydawać dziwne, że prymitywami języka projektowania wysokiego poziomu, takiego jak Verilog, są tranzystory. Jednak nie należy zapominać, że Verilog początkowo przeznaczony był do symulacji, więc powinien umożliwiać symulację elementów układów cyfrowych na każdym poziomie, poczynając od tranzystorów, a kończąc na poziomie strukturalnym czy hierarchicznym. Opis przełącznikowych prymitywów języka Verilog umieszczono w tabeli 2.4.

TABELA 2.4. Przełącznikowe prymitywy języka Verilog

Nazwa	Cechy
<i>pmos, nmos, rpmos, rnmos</i>	jedno wejście, jedno wyjście, jedno wejście sterujące
<i>cmos, rcmos</i>	jedno wyjście bądź więcej wyjść, wejście sterujące <i>n</i> , wejście sterujące <i>p</i>
<i>tran, rtran</i>	dwa dwukierunkowe kanały
<i>tranif0, tranif1, rtranif0, rtranif1</i>	dwa dwukierunkowe kanały, jedno wejście sterujące

Bardziej szczegółowy opis prymitywów języka Verilog można znaleźć w [14].

2.3. Typy danych

2.3.1. Dwie klasy typów danych

W języku Verilog występują dwie klasy podstawowych typów danych: sieciowe (*net*) i zmienne (**reg**). Sieciowe typy danych (*net*) wykorzystuje się do tworzenia połączeń pomiędzy różnymi częściami projektu. Sieci odzwierciedlają wartości i poziomy mocy źródeł sygnałów oraz wartość pojemności, jednak nie mają własnej wartości. Mimo to sieci pełnią funkcję określania wartości sygnału w sytuacji, gdy istnieje kilka źródeł sygnałów.

Zmienny typ danych (**reg**) wykorzystuje się do tymczasowego przechowywania danych. Deklaracje wartości tych zmiennych mogą występować w blokach proceduralnych, takich jak **initial** i **always**, a także w ciałach procedur i funkcji. Przykładowo

jeżeli wartość wyjściowego portu jest przypisywana wewnątrz bloku proceduralnego **always**, wówczas powinna ona mieć typ **reg**, nawet jeżeli występuje w układzie kombinacyjnym.

Zmienne mogą przechowywać jedynie wartości logiczne. Nie są one jednak w stanie przechowywać mocy sygnału. Zmienne nie są inicjowane domyślnie. Na początku symulacji mają wartość **X**, dlatego do ich wykorzystania niezbędne jest przypisanie im właściwej wartości.

Zasady wykorzystywania typów danych:

- 1) gdy sygnał sterowany jest wyjściem modułu, wyjściem prymitywu bądź przypisaniem ciągłym (**assign**), należy używać sieciowego typu danych (*net*);
- 2) gdy wartość sygnału ustalana jest wewnątrz bloku proceduralnego (**initial** albo **always**), należy wykorzystywać zmienny typ danych (**reg**).

2.3.2. Sieciowe typy danych

Sieciowe typy danych przeznaczone są do połączeń strukturalnych komponentów projektu. Składnia deklaracji danych typu *net* ma następujący format:

net_type **signed** [*range*] # (*delay*) *net_name* [*array*], ...;

Konstrukcje *signed*, [*range*], *delay*, [*array*] są opcjonalne. Występujący w składni parametr *net_type* przyjmuje postać jednego z następujących słów kluczowych:

- **wire** – połączenie typu węzłowego, wartość sygnału określa się zgodnie z prawami występującymi w technologii CMOS;
- **wor** – połączenie wyjść za pomocą operatora OR, wartość sygnału określa się zgodnie z zasadami logiki ECL;
- **wand** – połączenie wyjść za pomocą operatora AND, wartość sygnału określa się zgodnie z zasadami logiki układu z otwartym kolektorem (*open-collector*);
- **supply0** – stała logiczna 0, wartość logiczna zasilania;
- **supply1** – stała logiczna 1, wartość logiczna zasilania;
- **tri0** – w stanie wysokiej impedancji wyrównanie do niskiego poziomu;
- **tri1** – w stanie wysokiej impedancji wyrównanie do wysokiego poziomu;
- **tri** – zamiennik **wire**;
- **trior** – zamiennik **wor**;
- **triand** – zamiennik **wand**;
- **triereg** – w stanie wysokiej impedancji utrzymanie ostatniej wartości, logiczna wartość sygnału pojemności.

Słowo kluczowe **signed** wskazuje, że wartość interpretowana jest jako liczba całkowita ze znakiem w kodzie uzupełnień do 2. Jeżeli port lub komponent sieciowy podłączony do portu są zadeklarowane jako wartość ze znakiem, to obydwa w tym połączeniu będą traktowane jako wartości ze znakiem.

Specyfikacja *[range]* określa zakres bitów sygnału w formacie *[msb:lsb]*. Maksymalna szerokość zakresu jest ograniczona wartością $2^{16} = 65\,536$ bitów. Jednakże wiele kompilatorów dopuszcza nawet zakresy sięgające miliona bitów.

Struktura *delay* (opóźnienie) może być określona jedynie dla sieciowych typów danych i jest używana wyłącznie w symulacji.

Konstrukcja *[array]* służy do deklaracji tablic i ma następującą składnię:

[first_address:last_address] [first_address:last_address]...

gdzie każdy nawias kwadratowy określa jeden z wymiarów tablicy.

Tablica może mieć dowolną liczbę wymiarów, które są określane przez zakres adresów. Elementy *first_address* i *last_address* mogą być liczbami, stałymi, wyrażeniami bądź odwołaniem do funkcji o stałej wartości. Wartości adresów mogą być specyfikowane zarówno w porządku rosnącym, jak i malejącym. Maksymalny zakres pojedynczego wymiaru macierzy jest ograniczony wartością $2^{24} = 16\,777\,216$, jednak wiele kompilatorów nie narzuca na nią ograniczenia.

Przykłady deklaracji sieciowych typów danych:

```
wire a, b, c;           /* trzy 1-bitowe sieci skalarne */
tri1 [7:0] mag;         /* 8-bitowa sieć, w trzecim stanie
                        zostaje wyrównana do poziomu wysokiego */
wire signed [1:8] wyn;  /* 8-bitowa sieć ze znakiem */
wire [7:0] M[0:15][0:63]; /* dwuwymiarowa tablica 8-bitowych węzłów */
wire [0:15] (strong1, pull0) sum = a + b; /* 16-bitowa sieć z określoną logiczną mocą
                                           źródła i niejawną instrukcją przypisania ciągłego */
```

2.3.3. Znaczenie sygnałów sieci

Znaczenie sygnału sieci określa się za pomocą następujących typów: **wire (tri)**, **wand (triand)**, **wor (trior)**, **tri0** lub **tri1**. Gdy łączy się dwa źródła sygnałów, wynikowa wartość sygnału w sieci zależy od wartości tych źródeł, a także od typu łączącego węzła. Wyniki połączenia dwóch źródeł sygnałów za pomocą różnych typów węzłów zamieszczone są w tabeli 2.5.

TABELA 2.5. Wartość sygnału w różnych węzłach dla dwóch źródeł sygnałów

Wartość źródła		Wartość sygnału w sieci				
<i>a</i>	<i>b</i>	wire (tri)	wand (triand)	wor (trior)	tri0	tri1
0	0	0	0	0	0	0
0	1	X	0	1	X	X
0	Z	0	0	X	0	0
0	X	X	0	X	X	X
1	1	1	1	1	1	1

Wartość źródła		Wartość sygnału w sieci				
<i>a</i>	<i>b</i>	wire (tri)	wand (triand)	wor (trior)	tri0	tri1
1	Z	1	X	1	1	1
1	X	X	X	1	X	X
Z	Z	Z	Z	X	0	1
Z	X	X	X	X	X	X
X	X	X	X	X	X	X

Konieczność określenia wartości sygnału w węźle pojawia się, gdy węzeł łączy kilka źródeł sygnałów. Poniższy przykład ilustruje połączenie wyjść buforów za pomocą różnych typów węzłów:

```

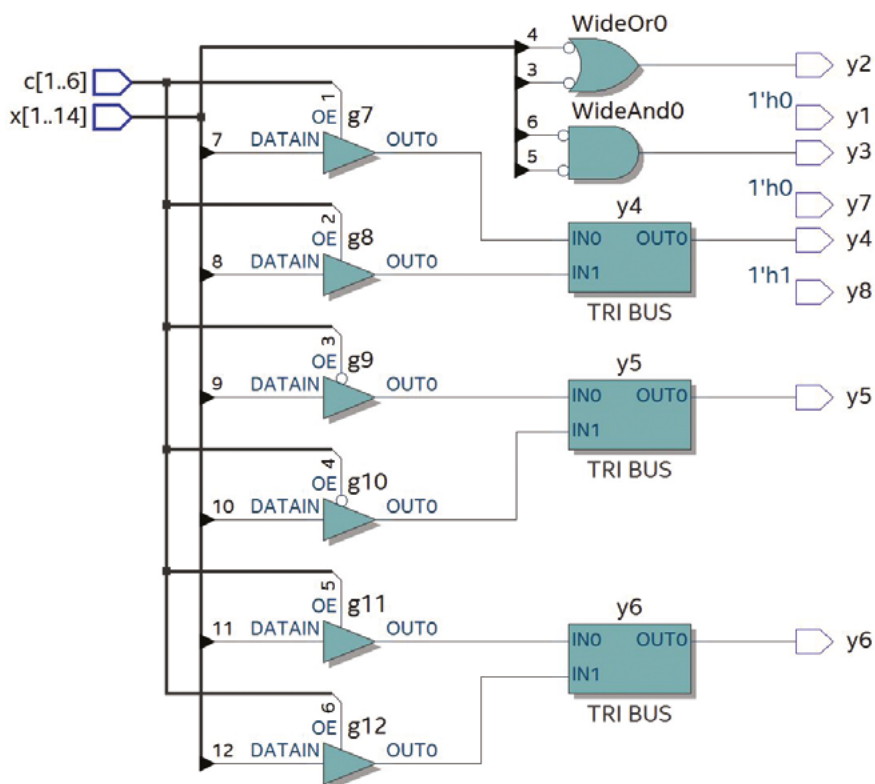
module net_connects(
    input [1:14] x,
    input [1:6] c,
    output wire y1,
    output wor y2,
    output wand y3,
    output tri0 y4,
    output tri1 y5,
    output trireg y6,
    output supply0 y7,
    output supply1 y8);

    not    //g1(y1,x[1]), // zabronione połączenie
          //g2(y1,x[2]), // wyjść typu wire
          g3(y2,x[3]),
          g4(y2,x[4]),
          g5(y3,x[5]),
          g6(y3,x[6]);

    bufif1 g7(y4,x[7],c[1]),
          g8(y4,x[8],c[2]);
    bufif0 g9(y5,x[9],c[3]),
          g10(y5,x[10],c[4]);
    bufif1 g11(y6,x[11],c[5]),
          g12(y6,x[12],c[6]);
endmodule

```

Realizacja modułu *net_connects* została przedstawiona na rysunku 2.8.



RYS. 2.8. Realizacja przykładu połączenia wyjść bramek za pomocą węzłów różnego typu: y1 – **wire**; y2 – **wor**; y3 – **wand**; y4 – **tri0**; y5 – **tri1**; y6 – **trireg**; y7 – **supply0**; y8 – **supply1**

2.3.4. Zmienne typy danych

Zmienne typy danych (**reg**) wykorzystuje się do przechowywania wartości w blokach proceduralnych. Określają one jedynie wartości logiczne i nie uwzględniają mocy sygnału. Gdy sygnał znajduje się po lewej stronie instrukcji proceduralnego przypisania **assign**, zawsze powinien być wykorzystywany typ **reg**. W starszych wersjach języka Verilog zmienne nazywano rejestrami (*registers*).

Deklaracje typów danych **reg** mają następującą składnię:

```
variable_type signed [range] variable_name, variable_name, ...;
variable_type signed [range] variable_name = initial_value, ...;
variable_type signed [range] variable_name [array], ...;
```

Pierwszy format składni służy do deklaracji listy zmiennych tego samego typu. Drugi pozwala na deklarację zmiennych wraz z ich początkowymi wartościami, trzeci zaś umożliwia deklarację tablic zmiennych. W przedstawionych formatach konstrukcje **signed**, [range], [array] i *initial_value* są opcjonalne.

Konstrukcja *variable_type* określa typ zmiennej i może przyjmować jedno z poniższych słów kluczowych:

- **reg** – zmienna zajmująca w pamięci określoną liczbę bitów, ze znakiem (jeśli dana konstrukcja jest opisana jako **signed**) bądź bez znaku;
- **integer** – 32-bitowa zmienna ze znakiem;
- **time** – 64-bitowa zmienna bez znaku;
- **real** – zmienna zmiennoprzecinkowa o podwójnej precyzji;
- **realtime** – zamiennik **real**.

Konstrukcja **signed** może występować jedynie ze zmiennymi typu **reg**; wskazuje ona, że wartość interpretowana jest jako liczba całkowita ze znakiem w kodzie uzupełnień do dwóch.

Konstrukcja *[range]* określa zakres bitów dla zmiennych typu **reg**. W przypadku braku konstrukcji *[range]* zmienna **reg** traktowana jest jako zmienna 1-bitowa. Format zakresu jest dokładnie taki sam, jak przy deklaracji zmiennych typu sieciowego. Maksymalny zakres dla zmiennych **reg** wynosi $2^{16} = 65\,536$ bitów. Niektóre kompilatory mają jednak ten zakres zwiększony do miliona bitów.

Konstrukcja do tworzenia tablicy *[array]* ma dokładnie taki sam format, jak ta służąca do tworzenia zmiennych sieciowych. Jednowymiarowa tablica zmiennych typu **reg** traktowana jest jak pamięć.

Konstrukcja *initial_value* określa początkowe wartości zmiennych. Początkowa wartość zmiennej jest ustalana w zerowym punkcie czasu symulacji, podobnie jak w przypadku nadawania wartości za pomocą procedury **initial**. Jeżeli wartość początkowa nie została określona, zmienne **reg**, **integer** i **time** przyjmują wartość **X**, natomiast zmienne typu **real** i **realtime** – 0.0.

Oprócz tego zaraz po wystąpieniu słowa kluczowego **reg** mogą być używane inne słowa kluczowe, jak **vectored** lub **scalared**, które mają dokładnie takie samo znaczenie jak dla sieciowych typów danych.

Przykłady deklaracji zmiennych:

- **reg** a, b, c; // trzy skalarnie zmienne jednobitowe;
- **reg signed** [7:0] v1, v2; // dwie 8-bitowe zmienne ze znakiem;
- **reg** [7:0] M[0:63][0:15]; // dwuwymiarowa tablica zmiennych 8-bitowych;
- **integer** i, k; // dwie zmienne całkowitoliczbowe ze znakiem;
- **real** r1, r2; // dwie rzeczywiste zmienne o podwójnej precyzji;
- **reg** clk2 = 0, rst2 = 1; // dwie zmienne typu **reg** z przypisaniem wartości.

Oprócz wspomnianych typów danych w języku Verilog występują także inne typy danych, takie jak: **parameter**, **localparam**, **specparam**, **genvar** i **event**. Nie są to podstawowe typy danych i pełnią funkcje pomocnicze.

2.3.5. Parametry

Typ danych **parameter** pozwala określać parametry dla modułów, takie jak wartości stałych, opóźnień (*delays*), oraz definiować ciągi znaków. Dla każdej instancji modułu wartości parametrów mogą być ustanowione na nowo.

Składnia typu danych **parameter**:

parameter signed [*range*] *constant_name* = *value*, ...

parameter *constant_type* *constant_name* = *value*, ...

W danych formatach konstrukcje **signed**, [*range*] i *constant_type* nie są obowiązkowe. Konstrukcje **signed** i [*range*] nie zmieniają swojego znaczenia, określają, czy liczba jest ze znakiem oraz jej zakres bitowy. Jeżeli przy deklaracji stałych nie był określony zakres [*range*], to pamięć zarezerwowana na daną stałą będzie obejmowała minimalną liczbę bitów, na których można zapisać przypisaną do stałej wartość.

Konstrukcja *constant_type* jest określana za pomocą następujących słów kluczowych: **integer**, **time**, **real** i **realtime**. Stała zadeklarowana z takim typem danych będzie miała te same właściwości, co zmienna danego typu. Jeżeli typ danych dla stałych nie jest określony, to domyślnie stała będzie miała taki typ danych, jak ostatnia przypisana do niej wartość.

Przykłady deklaracji parametrów:

parameter integer period = 10; // stała liczba całkowita

parameter [3:0] s1 = 4'b0001, // cztery 4-bitowe stałe

s2 = 4'b0010, // wykorzystuje się do kodowania

s3 = 4'b0100, // stanów wewnętrznych automatów

s4 = 4'b1000; // skończonych

2.3.6. Parametry lokalne

Typ danych **localparam** pozwala przypisywać wartości lokalnym parametrom działającym tylko w obrębie modułu. Parametry lokalne nie mogą zmieniać wartości przy tworzeniu nowej instancji modułu.

Składnia typu danych **localparam**:

localparam signed [*range*] *constant_name* = *value*, ...

localparam *constant_type* *constant_name* = *value*, ...

W tym typie danych konstrukcje **signed**, [*range*] i *constant_type* są nieobowiązkowe i oznaczają to samo co w poprzednich typach.

Przykład parametru lokalnego:

localparam signed st1 = -6; /* stała ze znakiem, bez rozmiaru, domyślnie przyjmie szerokość zainicjalizowanej wartości (6 = 110₂, czyli zajmuje 3bity + 1 bit znaku) */

2.3.7. Zmienne generacji

Typ danych **genvar** określa zmienną, która jest używana wyłącznie wewnątrz pętli generowania kodu (*generate loop*) w blokach generujących. Ta zmienna jest wykorzystywana przez kompilator do generowania powtarzających się fragmentów kodu i nie może być używana do symulacji.

Składnia typu danych **genvar**:

```
genvar event_name, ...
```

Przykład:

```
genvar i;      // deklaracja zmiennej generacji i
```

2.3.8. Łańcuchy znaków

Łańcuchy znaków w języku Verilog, w przeciwieństwie do języków programowania, nie są podstawowymi typami danych. Zgodnie z zasadą łańcuchy znaków nie są syntezowalne, a ich wykorzystanie ogranicza się do wyświetlania tekstów podczas symulacji. Łańcuchy znaków są także używane jako argumenty niektórych systemowych zadań i funkcji.

2.3.9. Skalary, wektory i tablice

Typy **net** i **reg** mogą być zadeklarowane jako skalarne lub wektorowe. Skalar ma zawsze szerokość jednego bitu, podczas gdy wektory mają szerokość większą niż jeden bit. Podczas deklaracji typów danych **net** lub **reg** ważne jest również rozróżnienie między wektorami a tablicami.

Pojęcie wektora jest przeciwieństwem skalara i służy do określenia szerokości zadeklarowanego elementu w bitach. Wektor deklaruje się za pomocą przedziału (*[range]*), który zapisuje się przed nazwą zmiennej. Zakres jest zawsze podawany przy użyciu pojedynczych nawiasów kwadratowych:

```
wire [7:0] bus_a;      // sieć bus_a ma szerokość 8 bitów  
reg [0:15] out;       // zmienna out ma szerokość 16 bitów
```

Tablica jest przeznaczona do definiowania liczby i organizacji strukturalnej zadeklarowanych elementów. Tablicę deklaruje się przy pomocy konstrukcji *[array]*, która reprezentuje jeden lub więcej nawiasów kwadratowych i jest zapisywana *po* nazwie zmiennej. Każdy nawias kwadratowy definiuje jeden wymiar tablicy. Liczba wymiarów tablicy w języku Verilog nie jest ograniczona. Przykłady tablic:

```
wire array_b [7:0] [3:0]; // dwuwymiarowa tablica 8 × 4 1-bitowych linii  
reg result_c [0:15];     // jednowymiarowa tablica 16 zmiennych 1-bitowych
```

Elementy tablicy, podobnie jak wszystkie zmienne, mogą być skalarne lub wektorowe. W powyższych przykładach elementami tablicy były skalary. Przykłady tablic wektorowych:

```
wire [7:0] bus_array [1:16]; // jednowymiarowa tablica 16 8-bitowych magistral  
reg [1:0] macierz [3:0] [0:2]; // macierz 4 × 3 zmiennych 2-bitowych
```

Tablica nie powinna być mylona z wektorem. Wektor jest pojedynczym elementem o określonej szerokości bitowej, natomiast tablica jest zbiorem elementów o szerokości jednego lub więcej bitów.

2.3.10. Wybór bitów i pól bitowych

Odwołanie do jednego bitu wektora przyjmuje następujący format:

```
vector_name [bit_number]
```

gdzie: *vector_name* – nazwa wektora, *bit_number* – numer wybranego bitu.

Numer wybranego bitu (*bit_number*) może być liczbą całkowitą, stałą, siecią, zmienną bądź wyrażeniem.

Polem bitowym nazywany jest nieprzerwany ciąg bitów pewnego wektora. Wybór pola bitowego może przyjmować następujące trzy formaty składni:

```
vector_name [bit_number : bit_number]
```

```
vector_name [starting_bit_number +: port_select_width]
```

```
vector_name [starting_bit_number -: port_select_width]
```

gdzie: *vector_name* i *bit_number* mają to samo znaczenie co w powyższym przykładzie, *starting_bit_number* – numer początkowego bitu, *port_select_width* – szerokość wybranej części bitów.

Pierwszy format składni jest praktycznie zgodny z notacją zakresów portu, określone zostają tu dwa krańcowe numery bitów tworzących pole bitowe. Drugi i trzeci format pozwalają wyznaczyć pole bitowe według numeru początkowego bitu i szerokości pola bitowego: w drugim formacie numery bitów pola w odniesieniu do bitu początkowego rosną, natomiast w trzecim maleją.

Numer bitu (*bit_number* lub *starting_bit_number*) w przedstawionych formatach składni powinien być liczbą bądź stałą. Szerokość wybieranej części wektora (*port_select_width*) może być liczbą, stałą lub odwołaniem się do funkcji o wartości stałej. Przy wyborze pola bitowego powinien być zachowany taki sam porządek numerów bitów, jaki był zadeklarowany w wektorze. Innymi słowy, jeżeli najmniejszy znaczący bit wektora ma mniejszy numer w deklaracji, to najmniej znaczący bit w wybranym polu także powinien mieć mniejszy numer.

Przykłady:

```
wire [7:0] data; // węzeł złożony z 8 bitów (magistrala z 8 linii)
```

```
data [7:3];      // przykłady prawidłowego wyboru pola bitowego  
data [7-:5];
```

```
data [3:7];      // przykłady nieprawidłowego wyboru pola bitowego  
data [3+:5];
```

2.3.11. Wybór elementów tablicy i pól bitowych elementów tablicy

Składnia wyboru elementów tablicy:

```
array_name [index][index] ...
```

gdzie: *array_name* – nazwa tablicy, *index* – numer indeksu określającego wybierany element.

Przykłady wyboru elementów tablicy:

```
wire [7:0] my_array [0:31] [0:63]; /* deklaracja dwuwymiarowej tablicy  
                                     8-bitowych węzłów */  
  
my_array [0] [0];                  // wybór pierwszego elementu tablicy my_array  
my_array [31] [63];                // wybór ostatniego elementu tablicy my_array
```

Składnia wyboru bitu elementu tablicy:

```
array_name [index][index]...[bit_number]
```

gdzie: *array_name* i *index* mają takie samo znaczenie jak poprzednio; *bit_number* – numer wybieranego bitu.

Przykłady wyboru bitów elementów tablicy:

```
my_array [0] [0] [0]; // wybór zerowego bitu w pierwszym  
                      // elemencie tablicy my_array  
my_array [31] [63] [7]; // wybór siódmego bitu w ostatnim  
                      // elemencie tablicy my_array
```

Składnia wyboru bitowego pola elementu tablicy:

```
array_name [index] [index]...[part_select]
```

gdzie: [*part_select*] – wybór części pola bitowego wektora, może przyjmować jeden z dopuszczalnych formatów składni wyboru pola.

Przykłady wyboru pola bitowego:

```
my_array [0] [0] [7:3] // wybór pięciu starszych bitów pierwszego  
                      // elementu tablicy my_array  
my_array [0] [0] [7-:5] // to samo
```

2.3.12. Deklaracja pamięci

Pamięć w języku Verilog jest reprezentowana za pomocą jednowymiarowych tablic zmiennych typu **reg**. Te tablice mogą być wykorzystywane do realizacji pamięci typu RAM, ROM lub rejestrów. Zasadniczo w języku Verilog tablice mogą mieć wiele wymiarów. Jednakże w przypadku tablic wielowymiarowych nie jest możliwe ich zastosowanie jako standardowych bloków pamięci w rzeczywistych urządzeniach.

Przykłady deklaracji pamięci:

```
reg [7:0] mem_1x8;           // jedno 8-bitowe słowo w pamięci
reg mem_8x1 [7:0];          // pamięć z ośmiu elementów 1-bitowych
reg [7:0] mem_1024x8 [0:1023]; // pamięć z 1024 słów 8-bitowych
reg [7:0] mem_5x8 [0:4];     // pamięć z pięciu 8-bitowych słów
```

Przykłady wykorzystania pamięci:

```
reg [7:0] mema [0:255];      // pamięć z 256 słów 8-bitowych
mema[1] = 0;                  // przypisanie elementowi pamięci mema
                               // z indeksem 1 wartości 8'b00000000
```

Należy jednak rozróżnić rejestry i pamięć złożoną z 1-bitowych słów, na przykład:

```
reg [7:0] regb;              // 8-bitowy rejestr (nie pamięć)
reg memb [7:0];              // pamięć z ośmiu słów 1-bitowych
```

W celu odczytu i zapisu elementów tablicy, która reprezentuje pamięć (jednowymiarowej tablicy zmiennych typu **reg**), można korzystać z następujących funkcji systemowych: **\$readmemb**, **\$readmemh**, **\$sreadmemb** i **\$sreadmemh**.

2.4. Wyrażenia

Wyrażenia są konstrukcjami języka Verilog, które łączą operacje i operandy w celu tworzenia wyniku. Operacje definiują działania wykonywane na operandach, które są argumentami operacji. Operandami większości operacji mogą być: liczby, sieci, zmienne, pojedyncze bity (sygnały), wektory, części wektorów (pola bitowe) oraz wywołania funkcji. Należy zauważyć, że niektóre operacje są niedozwolone dla liczb rzeczywistych.

W operacjach logicznych, które zwracają wartości *prawda* lub *fałsz*, bit o wartości 1 oznacza prawdę, a 0 i **X** oznaczają fałsz.

Wyrażenia języka Verilog są obliczane zgodnie z kolejnością ważności operacji. Nawiasy okrągłe można wykorzystać do zmiany kolejności obliczeń.

Przykłady wyrażeń:

```
a ^ b ^ cin;                  // opis funkcji sumy
a & b | (a ^ b) & cin;         // opis funkcji transferu
```

```
a & b | cin & a | cin & b // warunek ustawienia funkcji transferu
a + b + cin;             // opis funkcji dodawania
```

2.5. Operatory języka Verilog

Składnia większości operatorów języka Verilog odpowiada składni występującej w językach programowania C/C++. Operatory są wykorzystywane w wyrażeniach do wykonywania czynności na operandach.

Operatory języka Verilog zwykle dzieli się na następujące kategorie:

- operatory bitowe (*bitwise*);
- operatory redukcji (*reduction*);
- operatory logiczne (*logic*);
- operatory relacji (*relational*);
- operatory identyczności (*identity*);
- operatory arytmetyczne (*arithmetic*);
- operatory różne (*miscellaneous*).

2.5.1. Operatory bitowe

Operatory bitowe języka Verilog zostały przedstawione w tabeli 2.6. Są one często wykorzystywane do opisu projektów za pomocą funkcji i wyrażeń logicznych. Dodatkowo umożliwiają wykonywanie operacji logicznych na poszczególnych bitach wektorów i pól bitowych.

TABELA 2.6. Operatory bitowe

Symbol	Przykład	Opis
~	~m	inwersja każdego bitu wektora <i>m</i> , operator unarny
&	m & n	operacja logiczna AND każdego bitu wektora <i>m</i> z każdym bitem wektora <i>n</i>
	m n	operacja logiczna OR każdego bitu wektora <i>m</i> z każdym bitem wektora <i>n</i>
^	m ^ n	operacja logiczna alternatywy wykluczającej (XOR) każdego bitu wektora <i>m</i> z każdym bitem wektora <i>n</i>
~^ lub ^~	m ~^ n	negacja alternatywy wykluczającej (XNOR) każdego bitu wektora <i>m</i> z każdym bitem wektora <i>n</i>
<<	m << n	przesunięcie w lewo o <i>n</i> bitów (pozycji) zawartości wektora <i>m</i> , wartość bitów z lewej strony uzupełnia się zerami
>>	m >> n	przesunięcie w prawo o <i>n</i> bitów (pozycji) zawartości wektora <i>m</i> , wartość bitów z prawej strony uzupełnia się zerami

Wyniki działania operatorów bitowych zostały przedstawione w tabeli 2.7. Podczas obliczeń bitowych wartość **Z** jest traktowana tak samo jak wartość **X**.

TABELA 2.7. Wyniki operacji bitowych

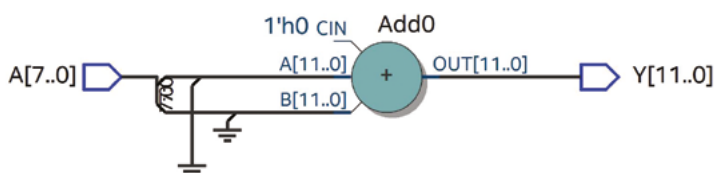
Operandy		Rezultaty wyrażeń				
a	b	$\sim a$	$a \& b$	$a b$	$a \wedge b$	$a \sim \wedge b$
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1
0	X	1	0	X	X	X
X	0	X	0	X	X	X
1	X	0	X	1	X	X
X	1	X	X	1	X	X
X	X	X	X	X	X	X

Jako przykład wykorzystania operatora bitowego przesunięcia pokażemy układ mnożący liczbę wejściową przez 10.

```
// Wykorzystana właściwość:  $A \cdot 10 = A \cdot (2 + 8) = A \cdot 2 + A \cdot 8$ 
module mult_on_10 (input [7:0] A,           // zmienna wejściowa
                   output [11:0] Y);       // wynik większy o 4 bity

    wire [8:0] Ax2 = A << 1;                //  $A \cdot 2$ 
    wire [10:0] Ax8 = A << 3;              //  $A \cdot 8$ 
    assign Y = Ax2 + Ax8;                  //  $Y = (A \ll 1) + (A \ll 3)$ ,
endmodule
```

Wynik syntezy przykładu *mult_on_10* został przedstawiony na rysunku 2.9.



RYŚ. 2.9. Realizacja modułu *mult_on_10*

2.5.2. Operatory redukcji

Operatory redukcji, nazywane także operatorami skracania, przypominają składnię operatory bitowe, ale ich działanie różni się znacząco. W przypadku operacji bitowych operatory są używane w każdej parze bitów w dwóch wektorach, co prowadzi do utworzenia nowego wektora. Natomiast operatory redukcji są stosowane

najpierw do pierwszych dwóch bitów wektora, a następnie operacja jest powtarzana na wyniku poprzedniej operacji i kolejnym bicie wektora. Ten proces jest kontynuowany dla wszystkich bitów wektora. Wynikiem operacji redukcji jest pojedynczy bit. Innymi słowy, operatory redukcji, stosując odpowiednie wyrażenie logiczne do każdego bitu wektora, zmniejszają wektor bitowy do jednego bitu.

Operatory redukcji języka Verilog zostały przedstawione w tabeli 2.8, a przykłady ich wykorzystania znajdują się w tabeli 2.9.

TABELA 2.8. Operatory redukcji

Symbol	Przykład	Opis
&	& m	redukuje bity wektora <i>m</i> za pomocą AND
~&	~& m	redukuje bity wektora <i>m</i> za pomocą NAND
	m	redukuje bity wektora <i>m</i> za pomocą OR
~	~ m	redukuje bity wektora <i>m</i> za pomocą NOR
^	^ m	redukuje bity wektora <i>m</i> za pomocą alternatywy wykluczającej (XOR)
~^ lub ^~	~^ m	redukuje bity wektora <i>m</i> za pomocą negacji alternatywy wykluczającej (XNOR)

TABELA 2.9. Przykłady wykorzystania operatorów redukcji

Operand	Rezultat instrukcji redukcji		
a	&a	a	^a
00000000	0	0	0
11111111	1	1	0
10101010	0	1	1
110011zz	0	1	x
1111111x	x	1	x

Poniższy kod ilustruje przykład wykorzystania operatorów redukcji do sprawdzenia, czy liczba jedynek w wektorze bitowym jest parzysta oraz czy wszystkie bity mają wartość 1:

```

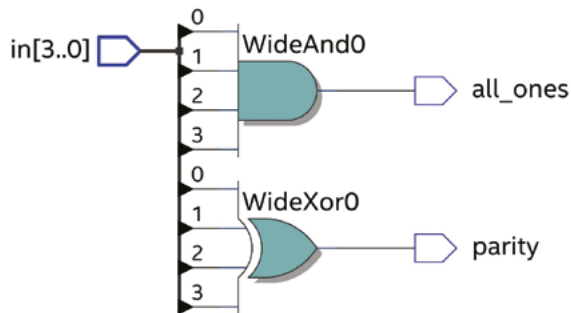
module inputs_test (
    input [3:0] in,           // wejściowy wektor bitów
    output parity,           // prawda, jeżeli liczba jedynek jest parzysta
    all_ones);              // prawda, jeżeli wszystkie bity mają wartość 1

    assign parity = ^ in;
    assign all_ones = & in;

endmodule

```

Wynik syntezy modułu *inputs_test* został przedstawiony na rysunku 2.10.



RYS. 2.10. Realizacja modułu *inputs_test*

Kolejny przykład prezentuje wykorzystanie operatorów bitowych i operatorów redukcji do realizacji funkcji równości w komparatorze:

```

module comp_eq (
    input [3:0] a, b,      // wektory wejściowe
    output eq);           // funkcja równości

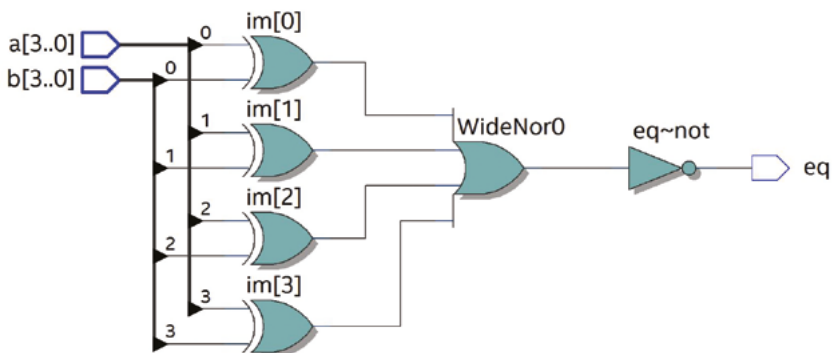
    wire [3:0] im;        // zmienna pomocnicza

    assign im = a ^ b;     // znalezienie bitów, gdzie a i b nie są równe
    assign eq = ~| im;     // jeżeli wektor im zawiera chociażby jedną jedynkę,
                           // wówczas a i b nie są równe

endmodule

```

Wynik syntezy modułu *comp_eq* został przedstawiony na rysunku 2.11.



RYS. 2.11. Realizacja modułu *comp_eq*

2.5.3. Operatory logiczne

Operatory logiczne są używane w wyrażeniach logicznych, na przykład w instrukcjach **if**, **for**, **while**. Wynikiem operatorów logicznych jest 1-bitowa wartość, która przyjmuje 1, jeśli wyrażenie jest prawdziwe, lub 0 (bądź **X**), jeśli jest fałszywe. Operatory logiczne zostały przedstawione w tabeli 2.10.

TABELA 2.10. Operatory logiczne

Symbol	Przykład	Opis
!	! <i>m</i>	prawda, jeżeli <i>m</i> to fałsz
&&	<i>m</i> && <i>n</i>	prawda, jeżeli <i>m</i> i <i>n</i> to prawda
	<i>m</i> <i>n</i>	prawda, jeżeli albo <i>m</i> , albo <i>n</i> to prawda

Poniższy kod przedstawia przykład układu sterującego mikrokontrolera, który sprawdza, czy podany rozkaz jest operacją odejmowania:

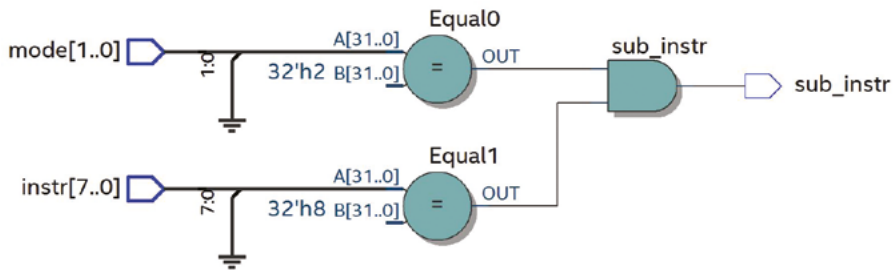
```
module if_subtraction
    (input [7:0] instr,
     input [1:0] mode,
     output sub_instr);

    parameter IMMEDIATE=2'b00, DIRECT=2'b01;           // lokalne parametry
                                                         // określające tryb pracy
                                                         // mikrokontrolera
    parameter SUBA_imm=8'h80, SUBA_dir=8'h90,           // lokalne parametry określające
    SUBB_imm=8'hc0, SUBB_dir=8'hd0;                     // kod operacji

    assign sub_instr = (((mode == IMMEDIATE) &&
                        ((instr == SUBA_imm) ||
                         (instr == SUBB_imm))) ||
                       ((mode == DIRECT) &&
                        ((instr == SUBA_dir) ||
                         (instr == SUBB_dir))));

endmodule
```

Wynik syntezy modułu *if_subtraction* został przedstawiony na rysunku 2.12.



RYS. 2.12. Realizacja modułu *if_subtraction*

2.5.4. Operatory relacji

Operatory relacji zazwyczaj znajdują zastosowanie w wyrażeniach logicznych, pozwalając na porównanie wartości operandów pod kątem relacji (na przykład większość, mniejszość, równość). Wynikiem operacji relacji jest 1-bitowa wartość wskazująca, czy dane wyrażenie jest prawdziwe, czy fałszywe. Operatory relacji zostały przedstawione w tabeli 2.11.

TABELA 2.11. Operatory relacji

Symbol	Przykład	Opis
==	$m == n$	prawda, jeżeli m jest równe n
!=	$m != n$	prawda, jeżeli m nie jest równe n
<	$m < n$	prawda, jeżeli m jest mniejsze od n
>	$m > n$	prawda, jeżeli m jest większe niż n
<=	$m <= n$	prawda, jeżeli m jest mniejsze bądź równe n
>=	$m >= n$	prawda, jeżeli m jest większe lub równe n

Wykorzystanie instrukcji relacji zaprezentowano poniżej na przykładzie syntezy parametryzowanego komparatora:

```

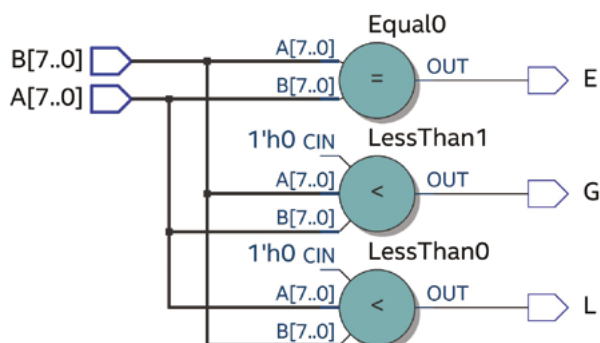
module comparator
    #(parameter SIZE = 8)
    (input [SIZE-1:0] A, B,
     output L, E, G);

    assign L = (A < B);
    assign E = (A == B);
    assign G = (A > B);

```

endmodule

Wynik syntezy układu *comparator* został przedstawiony na rysunku 2.13.



RYS. 2.13. Realizacja modułu *comparator*

Jeśli choć jeden z operandów relacji zawiera bit o wartości **X** lub **Z**, wynikiem operacji relacji będzie **X**. Mówiąc inaczej, operatory relacji uwzględniają jedynie istotne wartości operandów. W przypadku wystąpienia wartości logicznej **X** lub **Z** na dowolnym bicie w którymkolwiek z operandów wynik wyrażenia relacji również będzie **X**. Do porównywania operandów, gdzie bity mogą przyjmować wartości **X** lub **Z**, służą operatory identyczności.

2.5.5. Operatory identyczności

Sprawdzanie identyczności w języku Verilog jest wyrażane za pomocą dwóch instrukcji: `===` oraz `!==`. Różnica pomiędzy tymi operatorami a operatorami relacji równości (`==`) i nierówności (`!=`) polega na tym, że pozwalają one na uwzględnienie wszystkich czterech możliwych stanów logicznych każdego bity operandów: 0, 1, **X** i **Z**. Jednak nie wszystkie kompilatory obsługują operatory identyczności podczas syntezy projektu.

2.5.6. Operatory arytmetyczne

Operatory arytmetyczne języka Verilog zostały przedstawione w tabeli 2.12.

TABELA 2.12. Operatory arytmetyczne

Symbol	Przykład	Opis
+	$m + n$	dodawanie m do n
-	$m - n$	odejmowanie n od m
-	$-m$	liczba przeciwna do m (reprezentacja w kodzie uzupełnień do 2)
*	$m * n$	mnożenie m razy n

Symbol	Przykład	Opis
/	m / n	dzielenie m przez n
%	$m \% n$	reszta z dzielenia m przez n
**	$m ** n$	liczba m podniesiona do potęgi n
<<<	$m <<< n$	przesunięcie m w lewo o n bitów, zwolnione bity z prawej strony zostają wypełnione zerami
>>>	$m >>> n$	przesunięcie m w prawo o n bitów; jeżeli m jest liczbą ze znakiem, wówczas zwalniane bity z lewej strony wypełniane są wartością bitu znaku, w przypadku liczby bez znaku są wypełniane zerami

Przykład wykorzystania operatora arytmetycznego:

```
assign {cout, s} = a + b + cin;
```

Należy zauważyć, że nie wszystkie operatory arytmetyczne są wspierane przez kompilatory podczas syntezy projektu.

2.5.7. Operatory różne

Inne operacje w języku Verilog, które nie zostały sklasyfikowane w żadnej z powyższych grup, zostały nazwane operatorami różnymi (*miscellaneous*) i przedstawione w tabeli 2.13.

TABELA 2.13. Operatory różne

Symbol	Przykład	Opis
?:	$sel ? m : n$	operacja warunkowa, jeżeli sel to prawda, zwraca m , w przeciwnym wypadku zwraca n
{ }	$\{m, n\}$	operacja konkatencji, złączenie m z n , tworzy wektor
{ { } }	$\{n\{m\}\}$	operator powtórzenia, łączy ze sobą m , n -krotnie
->	$-> m$	operator wystąpienia zdarzenia, zmienia wartość m typu <i>event</i>

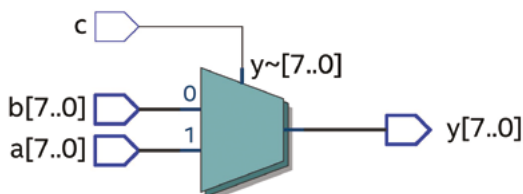
Za pomocą operatora warunkowego można w prosty sposób zrealizować multiplexer 2-1:

```
module mux_2_1(
    input [7:0] a, b,      // magistrale wejściowe
    input c,              // sygnał sterujący
    output [7:0] y);      // magistrala wyjściowa

    assign y = c ? a: b;

endmodule
```

Realizacja modułu *mux_2_1* została przedstawiona na rysunku 2.14.



RYS. 2.14. Realizacja modułu *mux_2_1*

Kilka przykładów wykorzystania operatora konkatencji:

```
wire [3:0] a, b;           // wektory 4-bitowe
wire [7:0] c, d;           // wektory 8-bitowe
wire [11:0] e, f;          // wektory 12-bitowe

assign c = {a,b};          // wynik 8-bitowy
assign e = {b,a,b};         // wynik 12-bitowy
assign f = {3{a}};          // wynik 12-bitowy: {a,a,a}
assign b = {4{e==f}};       // wynik 4-bitowy: {(e==f), (e==f), (e==f), (e==f)}
assign f = {a,d};           // wynik 12-bitowy
assign e = {2{1'b1,a,1,b0}}; // wynik 12-bitowy: {1,a,0,1,a,0}
assign {a,b} = d;            // wynik 8-bitowy
assign {a,b,f} = {e,d} +1;   // wynik 20-bitowy, może wystąpić przepełnienie!
```

2.5.8. Wykonywanie operacji

Przed wykonaniem operacji wszystkie operandy są rozszerzane do rozmiaru największego wektora w wyrażeniu (zarówno po lewej, jak i po prawej stronie operatora). Wartości bez znaku są uzupełniane zerami, a wartości ze znakiem są uzupełniane bitem znaku. Operatory konkatencji i powtórzenia są rozszerzane dla każdej pary operandów.

Wyrażenia arytmetyczne są obliczane zgodnie z poniższymi zasadami:

- jeżeli którykolwiek z operandów jest liczbą rzeczywistą, wynik wyrażenia jest liczbą zmiennoprzecinkową;
- jeżeli którykolwiek z operandów jest wartością bez znaku, wynik również jest wartością bez znaku;
- jeżeli wszystkie operandy są wartościami ze znakiem, wykonywana jest arytmetyka dla liczb ze znakiem.

Operand może być określony jako ze znakiem bądź bez znaku za pomocą funkcji systemowych **\$signed** oraz **\$unsigned**.

2.5.9. Priorytety operatorów

Operatory w języku Verilog są wykonywane zgodnie z porządkiem priorytetów instrukcji. Aby zmienić kolejność wykonywania działań, należy użyć nawiasów okrągłych. Priorytety operatorów języka Verilog przedstawiono w tabeli 2.14 – maleją w niej wraz z numeracją.

TABELA 2.14. Priorytety operatorów

Priorytet instrukcji	Symbole	Uwagi
1	!, ~, +, -	operatory unarne (logiczna negacja, inwersja bitowa, arytmetyczne operatory znaków liczb dodatnich i ujemnych)
2	{}, {{}}	operatory konkatencji i powtórzenia
3	()	nawiasy okrągłe określają kolejność wykonywania działań
4	**	arytmetyczny operator potęgowania
5	*, /, %	arytmetyczne operatory mnożenia, dzielenia, reszty z dzielenia (modulo)
6	+, -	arytmetyczne operatory dodawania i odejmowania
7	<<, >>, <<<, >>>	operatory przesunięcia logicznego i arytmetycznego
8	<, <=, >, >=	operatory relacji
9	==, !=, ===, !==	operatory równości i identyczności
10	&, ~&	operatory bitowe AND i NAND
11	^, ~^	operatory bitowe XOR i XNOR
12	~, ~	operatory bitowe OR i NOR
13	&&	operator logiczny AND
14		operator logiczny OR
15	? :	operator warunkowy

2.5.10. Rozmiar wyrażeń bitowych

Rozmiary wyrażeń bitowych, otrzymywane w zależności od wykorzystywanego operatora, zaprezentowane zostały w tabeli 2.15.

TABELA 2.15. Rozmiary wyrażeń bitowych

Wyrażenie	Rozmiar	Operator op
$i \text{ op } j$	$\max(L(i), L(j))$	operatory binarne + - / * % & ^ ~^
$\text{op } i$	$L(i)$	operatory unarne + - ~
$i \text{ op } j$	1 bit	=== !== == != && < <= > >= & ~& ~ ^ ~^ !
$i \text{ op } j$	$L(i)$	>> << ** >>> <<<

Wyrażenie	Rozmiar	Operator op
$i ? j : k$	$\max(L(j), L(k))$	operator warunkowy
$\{i, \dots, j\}$	$L(i) + \dots + L(j)$	operator konkatencji
$\{i\{j, \dots, k\}\}$	$i * (L(j) + \dots + L(k))$	operator powtórzenia

2.6. Instrukcja przypisania ciągłego assign

2.6.1. Przypisania w języku Verilog

W języku Verilog nie występują typowe instrukcje przypisania znane z innych języków programowania. Jest to spowodowane różnicami w przypisywaniu wartości takim typom danych, jak sieci *net* i zmienne **reg**. Przypisywanie wartości zmiennym (**reg**) jest podobne do przypisywania wartości w językach programowania. Jednak sieci (*net*) służą głównie do tworzenia połączeń między elementami projektu i mogą być porównane do przewodów łączących wyprowadzenia poszczególnych układów elektronicznych. Warto zaznaczyć, że w schematach elektronicznych te połączenia są ciągłe i niezmiennie w czasie działania urządzenia lub schematu. Przypisanie wartości do ścieżki oznacza nadanie wartości sygnałowi wewnątrz niej, co jest zwykle określane przez źródło sygnału, czyli sterownik. W przypadku jednej sieci może istnieć kilka źródeł sygnałów, które mogą działać niezależnie i równocześnie.

Instrukcja przypisania ciągłego **assign** określa wartość jednego ze źródeł sygnału sieci. Ponieważ ścieżki w układach cyfrowych są niezmiennie i połączone ze stałymi źródłami sygnałów, instrukcja **assign** jest nazywana *przypisaniem ciągłym* (*continuous*). Ostateczna wartość sygnału w sieci jest determinowana przez typ ścieżki, czyli sposób, w jaki źródło sygnału łączy się z siecią, na przykład jako OR, AND, z otwartym kolektorem, lub inny.

Ponieważ wszystkie sygnały w schemacie elektronicznym są przesyłane przez ścieżki równocześnie, instrukcje przypisania ciągłego **assign** są również wykonywane równolegle. Dodatkowo w języku Verilog jednocześnie wykonywane są:

- instancje modułów;
- instancje prymitywów;
- bloki proceduralne.

2.6.2. Formaty instrukcji przypisania ciągłego

Instrukcja przypisania ciągłego **assign** ma dwie składnie: jawną i niejawną. Jawny format instrukcji **assign** wygląda następująco:

```
assign # (delay) net_name = expression;
```

przy czym zakłada się, że ścieżka była wcześniej określona za pomocą następującej składni:

```
net_type [size] net_name;
```

Niejawny format instrukcji **assign** wykorzystywany jest przy deklaracji nazwy sieci. W praktyce jest to połączenie dwóch wcześniejszych formatów bez słowa kluczowego **assign**:

```
net_type [size] # (delay) net_name = expression;
```

W zaprezentowanych składniach konstrukcje `[size]` i `# (delay)` nie są obowiązkowe. Wykorzystywany tu `net_type` jest jednym z sieciowych typów danych, z wyjątkiem typu **trireg**, `[size]` oznacza rozmiar wektora lewej części w formacie `[msb:lsb]`, a `delay` to czas opóźnienia wykonania operacji przypisania (domyślnie przyjmuje wartość 0).

Nazwa sieci `net_name` w lewej części instrukcji **assign** powinna zostać wcześniej zadeklarowana jako nazwa ścieżki lub portu modułu. W ostatnim przypadku dodatkowa deklaracja nazwy portu nie jest wymagana. Jeżeli niezadeklarowana nazwa ścieżki `net_name` nie jest nazwą portu, wówczas domyślnie traktowana jest jako jednobitowa ścieżka typu **wire**.

Wyrażenie `expression` w prawej części instrukcji przypisania ciągłego **assign** może zawierać różne typy danych, instrukcje oraz wywołania funkcji.

Działanie instrukcji przypisania ciągłego **assign** jest następujące: po każdej zmianie wartości w wyrażeniu `expression` po prawej stronie wyrażenie to jest obliczane, a otrzymana wartość po upływie określonego czasu opóźnienia `delay` jest przypisywana wartości po lewej stronie znaku `=`.

Przykłady wykorzystania instrukcji **assign**:

```
// jawne przypisanie ciągłe
wire [31:0] mux_out;
assign mux_out = sel ? a : b;           // multiplexer szyn a i b na 32 bitach

// niejawne przypisanie ciągłe
tri [0:15] # 2.7 buf_out = en ? in : 16'bz; // 16-bitowy bufor z trzema stanami
// i opóźnieniem równym 2,7 jednostkom // czasu
```

2.6.3. Wykorzystanie instrukcji przypisania ciągłego

Na wstępie warto podkreślić dwie ważne właściwości instrukcji **assign**:

- 1) równoległość – wszystkie instrukcje **assign** w jednym module są wykonywane równolegle, niezależnie od ich lokalizacji;
- 2) wrażliwość na zdarzenia związane z sygnałami z prawej strony znaku równości – za każdym razem, gdy wartość ścieżki lub zmiennych po prawej stronie ulega zmianie, wyrażenie to jest obliczane od nowa, a uzyskaną wartość przypisuje się ścieżce po lewej stronie instrukcji **assign**.

Instrukcja **assign** znajduje szerokie zastosowanie w opisie układów kombinacyjnych za pomocą wyrażeń logicznych. Przedstawimy tutaj opis jednobitowego sumatora z pośrednią ścieżką *a_xor_b*, wykorzystując instrukcję przypisania ciągłego w sposób niejawny:

```
module sum_1_1
    (input a, b, cin,
     output s, cout);

    wire a_xor_b = a ^ b;      // niejawne wykorzystanie
                               // instrukcji przypisania ciągłego
                               // jednocześnie z inicjacją ścieżki
    assign s = a_xor_b ^ cin;   // jawne wykorzystanie instrukcji przypisania ciągłego
    assign cout = (a & b) | ((a_xor_b) & cin);
endmodule
```

Warto zaznaczyć, że stosowanie instrukcji **assign** umożliwia opisanie układów kombinacyjnych bez konieczności zgłębiania ich struktury. Wyeliminowana jest także potrzeba manualnego projektowania, takiego jak przekształcanie równań logicznych, minimalizacja, faktoryzacja czy dekompozycja. Wystarczy jedynie utworzyć wyrażenia logiczne opisujące działanie układu kombinacyjnego, a resztą zajmie się automatycznie kompilator. Taka praktyka uwalnia projektantów od wykonywania formalnych i czasem skomplikowanych operacji związanych z syntezą logiczną, umożliwiając im skupienie się na algorytmie działania układu cyfrowego. Dodatkowo taki sposób projektowania znacznie redukuje ryzyko błędów wynikających z ręcznego projektowania oraz zwiększa czytelność kodu projektu.

Po lewej stronie instrukcji **assign** może występować operacja konkatencji, co zostało zilustrowane na poniższym przykładzie:

```
module sum_1_2
    (input a, b, cin,
     output s, cout);

    assign {cout, s} = a + b + cin;

endmodule
```

Po obliczeniu wartości wyrażenia i wykonaniu wszystkich przekształceń po lewej stronie – w przypadku ogólnym zarówno po prawej, jak i po lewej stronie znaku = instrukcji przypisania **assign** – znajdować się będą wektory bitowe. Jeśli oba wektory będą miały jednakowy rozmiar, wartości bitów z prawego wektora zostaną przypisane bitom lewego wektora, zaczynając od najmniej znaczącego bitu. Jeśli liczba bitów na prawej stronie jest większa niż na lewej, starsze bity będą odrzucone.

Natomiast jeśli liczba bitów po lewej stronie jest większa, starsze bity zostaną wypełnione zerami.

W kolejnym przykładzie instrukcja **assign** jest wykorzystywana do opisu prostej jednostki ALU. W tym przykładzie ALU wykonuje operacje dodawania i odejmowania na argumentach *a* i *b*. Dodatkowo ustalane są wartość sygnału przeniesienia *c* oraz dwie flagi: *gt* – oznaczająca większość (gdy $a > b$) oraz *z* – oznaczająca wynik zerowy.

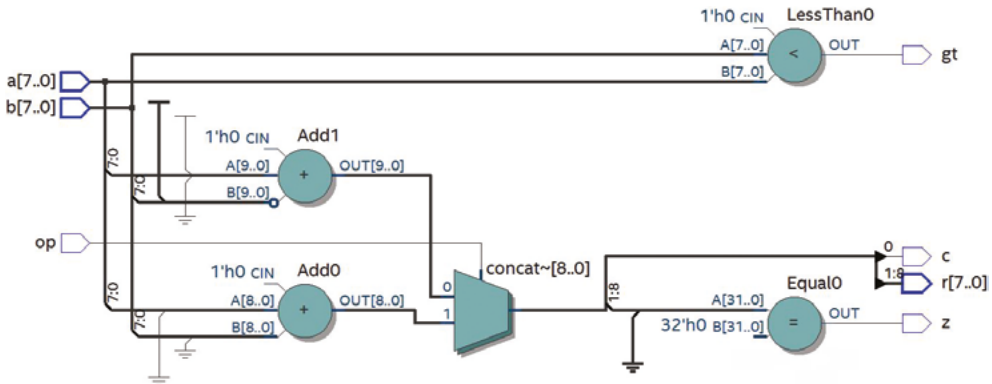
```

module simple_ALU (
    input [7:0] a, b,      // argumenty
    input op,             // kod operacji
    output [7:0] r,       // wynik
    output gt, c, z;      // flagi

    assign {c,r} = op ? (a + b) : (a - b);
    assign gt = (a > b);
    assign z = (r == 0);
endmodule

```

Wynik syntezy modułu *simple_ALU* został przedstawiony na rysunku 2.15.



RYŚ. 2.15. Realizacja modułu *simple_ALU*

W taki sposób za pomocą instrukcji przypisania ciągłego **assign** można opisywać dość skomplikowane układy kombinacyjne.

2.7. Instrukcje i bloki proceduralne

2.7.1. Instrukcje proceduralne **initial** i **always**, bloki proceduralne

W języku Verilog istnieją dwie *instrukcje proceduralne*, **initial** oraz **always**, które umożliwiają tworzenie *bloków proceduralnych*. Blok proceduralny reprezentuje grupę instrukcji zawartych w nawiasach instrukcji, takich jak **begin-end** lub **fork-join**.

Instrukcje wewnątrz bloku proceduralnego typu **initial** są wykonywane tylko raz. Bloki tego typu zazwyczaj używane są do inicjalizacji wartości zmiennych przed rozpoczęciem symulacji. Natomiast instrukcje wewnątrz bloku proceduralnego typu **always** są wykonywane ciągle w formie nieskończonej pętli: po wykonaniu ostatniego polecenia w bloku sterowanie jest przekazywane do pierwszej instrukcji tego bloku. Bloki typu **always** najczęściej służą do opisu zachowania układów cyfrowych.

Wartości przypisane zmiennym w blokach proceduralnych są przechowywane do momentu, gdy zostanie im ponownie przypisana nowa wartość. Za pomocą bloków proceduralnych można opisywać zarówno układy kombinacyjne, jak i sekwencyjne.

2.7.2. Instrukcje **begin-end** i **fork-join**

W języku Verilog występują dwa rodzaje nawiasów instrukcji, które są określane słowami kluczowymi **begin-end** oraz **fork-join**.

Przykłady bloków proceduralnych:

initial

```
begin // blok proceduralny initial z nawiasami instrukcji begin-end  
/* tutaj może znajdować się kod projektu */  
end
```

always

```
fork // blok proceduralny always z nawiasami instrukcji fork-join  
/* tutaj może znajdować się kod projektu */  
join
```

Linie kodu wewnątrz nawiasów instrukcji **begin-end** są wykonywane sekwencyjnie w takiej kolejności, w jakiej zostały zapisane w kodzie źródłowym projektu. Czas rozpoczęcia wykonania każdej instrukcji w grupie **begin-end** odpowiada czasowi zakończenia wykonywania poprzedniej instrukcji. Natomiast instrukcje wewnątrz grupy **fork-join** są wykonywane równolegle. Czas rozpoczęcia wykonania każdej operacji w grupie **fork-join** jest jednakowy i odpowiada czasowi rozpoczęcia wykonywania całej grupy. Jeżeli grupa zawiera tylko jedną instrukcję, konstrukcje **initial** oraz **always** mogą być zapisane bez nawiasów instrukcji **begin-end** czy **fork-join**.

Warto zauważyć, że nawiasy instrukcji **fork-join** nie są wspierane przez synteзаторы.

2.7.3. Nazwane bloki proceduralne

Blokom proceduralnym można przypisać nazwy. Wówczas taki blok jest określany *nazwanym* blokiem proceduralnym (*named block*).

Przykłady nazwanych bloków proceduralnych:

initial

```
begin : block_1      // nazwany blok proceduralny initial o nazwie block_1  
                        /* tutaj może znajdować się kod projektu */  
end
```

always

```
fork : block_2      // nazwany blok proceduralny always o nazwie block_2  
                        /* tutaj może znajdować się kod projektu */  
join
```

W porównaniu z blokami bez nazwy nazwane bloki proceduralne mają kilka dodatkowych cech:

- etykiety nazwanych bloków proceduralnych mogą być używane w hierarchicznej ścieżce przy wyszukiwaniu określonego elementu w strukturze projektu;
- nazwane bloki proceduralne mogą zawierać deklaracje własnych lokalnych zmiennych i parametrów;
- wykonywanie instrukcji wewnątrz nazwanego bloku proceduralnego może być przerwane za pomocą instrukcji **disable**.

2.7.4. Format bloków proceduralnych

Ogólnie bloki proceduralne mogą mieć następującą składnię:

```
type_of_block @ (sensitivity_list)  
statement_group : group_name  
local_variable_declarations  
time_control procedural_statements  
end_of_statement_group
```

przy czym konstrukcje *sensitivity_list*, *statement_group*, *group_name*, *time_control*, *local_variable_declarations* i *end_of_statement_group* nie są obowiązkowe.

Wartość *type_of_block* określa typ bloku proceduralnego i może przyjmować jedno z następujących słów kluczowych: **initial** lub **always**.

Konstrukcja @ (*sensitivity_list*) nazywana jest *listą czułości*. Pozwala ona kontrolować różne zdarzenia czasowe, takie jak zmiana wartości jednego lub kilku sygnałów, opadające lub narastające zbocze sygnału, oraz inne, służące do sterowania wykonywaniem instrukcji zawartych w bloku proceduralnym.

W nazwanych blokach proceduralnych po słowie kluczowym **begin** albo **fork** po dwukropku występuje nazwa (*group_name*) bloku proceduralnego. Bloki proceduralne mogą także zawierać deklaracje zmiennych i parametrów lokalnych (*local_variable_declarations*), których zakres działania jest ograniczony do obszaru bloku proceduralnego. Następnie, po deklaracji zmiennych lokalnych, występują instrukcje bloku proceduralnego.

Przykłady bloków proceduralnych:

/ przykład bloku proceduralnego **always**, który sterowany jest za pomocą zmiany dowolnego z sygnałów wejściowych*

**/*

```
always @ (a or b or cin) begin           // w danym przypadku nawiasy proceduralne nie są
                                           // wymagane
```

```
    s = a + b + cin;
```

```
end
```

/ przykład bloku proceduralnego **always**, sterowanego wzrastającym zboczem sygnału synchronizacji *clk**

**/*

```
always @ (posedge clk)
```

```
    q = data;
```

Jeszcze jednym przykładem jest układ kombinacyjny kontroli większościowej dla trzech wejść:

```
module maj_3 (
```

```
    input a, b, c,                      // opis wejść
```

```
    output reg y);                     // opis wyjścia
```

```
    always @ (a, b, c)                 // blok proceduralny z listą czułości
```

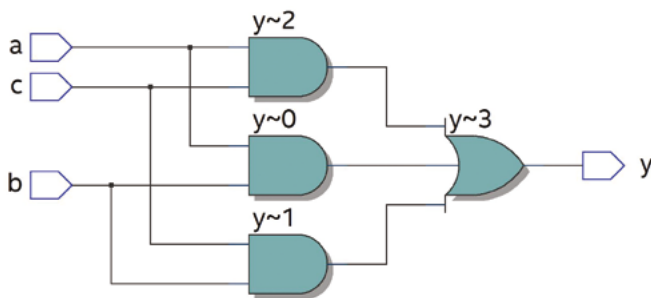
```
    begin                               // początek bloku proceduralnego
```

```
        y = (a & b) | (b & c) | (a & c); // instrukcja przypisania
```

```
    end                                 // koniec bloku proceduralnego
```

```
endmodule
```

Blok **always**, opisujący zachowanie danego układu, wykorzystuje instrukcję sterowania zdarzeniami @, która zawiera listę zmiennych (*a*, *b*, *c*) określaną jako listę czułości bloku **always**. Można zatem stwierdzić, że blok **always** reaguje na zmiany zmiennych *a*, *b* i *c*. Gdy wystąpi zdarzenie (zmiana wartości dowolnej z tych zmiennych), rozpoczyna się proces wykonywania bloku **always**. W wyniku tego obliczone zostanie wyrażenie po prawej stronie instrukcji przypisania, a jego wartość zostanie zapisana w zmiennej *y*. Przechowuje ona swoją wartość tak długo, dopóki nie wystąpi kolejne zdarzenie związane ze zmiennymi *a*, *b* lub *c*. W przedstawionym przykładzie blok proceduralny **always** zawiera tylko jedną instrukcję, dlatego nawiasy instrukcji **begin-end** można pominąć. Wynik syntezy przykładu został przedstawiony na rysunku 2.16.



RYS. 2.16. Realizacja modułu *maj_3*

2.8. Zarządzanie czasem

W języku Verilog opracowano trzy sposoby sterowania czasem wykonywania operacji: instrukcje opóźnienia **#**, czułości **@** oraz oczekiwania **wait**. Instrukcje opóźnienia **#** i oczekiwania **wait** są używane tylko w symulacji.

2.8.1. Instrukcja czułości **@**

Instrukcja czułości **@** pozwala na wstrzymanie wykonania kolejnej instrukcji (lub grupy instrukcji) do momentu wystąpienia przynajmniej jednego z określonych zdarzeń. Takimi zdarzeniami mogą być zmiany wartości sygnału bądź rodzaj zbocza sygnału. Instrukcja czułości przyjmuje następujące formaty składni:

@ (edge signal or edge signal or ...)

@ (edge signal, edge signal, ...)

@ (*)

gdzie w nawiasach okrągłych podawana jest lista czułości, *edge* – zbocze sygnału, które może być jednym z następujących słów kluczowych: **posedge** (zbocze narastające) lub **negedge** (zbocze opadające); **or** – słowo kluczowe; *signal* – sygnał, którego zmiana wartości jest monitorowana.

Konstrukcja *edge* przed każdym sygnałem nie jest obowiązkowa. Jeśli przed sygnałem występuje jedno ze słów kluczowych **posedge** lub **negedge**, to sprawdzane jest odpowiednie zbocze sygnału. Jeśli jednak przed sygnałem nie występuje ta konstrukcja, to sprawdzana jest dowolna zmiana wartości sygnału.

Na liście czułości mogą występować sygnały sieciowe lub zmienne o dowolnym rozmiarze. Są one oddzielane przecinkami, choć zamiast przecinka można użyć słowa kluczowego **or**. Jeśli lista czułości zawiera tylko jeden sygnał, nawiasy okrągłe można pominąć. Znak gwiazdki w nawiasach okrągłych (*) oznacza, że lista czułości zawiera wszystkie rejestrowane sygnały, które są przetwarzane w danej instrukcji.

Przykłady instrukcji czułości:

```
@ (a, b, c)           // sprawdzana jest zmiana wartości sygnałów a, b i c
@ (a or b or c)       // to samo co wyżej
@ (posedge clk1, negedge clk2) // sprawdza się zbocze rosnące sygnału clk1
                           // oraz zbocze opadające clk2
@ (*) // sprawdza się zmiany wartości wszystkich // sygnałów wejściowych
```

2.8.2. Lista czułości

Lista czułości służy do zawieszenia (opóźnienia) wykonania bloku proceduralnego do momentu wystąpienia zdarzenia związanego ze zmianą wartości sygnałów wejściowych.

Najczęściej listę czułości wykorzystuje się w instrukcji proceduralnej **always**. W takim przypadku ma ona następującą składnię:

```
always @ (signal, signal, ...) statement;
```

gdzie: **always** – instrukcja proceduralna, w której występuje lista czułości; @ – znak specjalny poprzedzający listę czułości; *signal, signal, ...* – lista sygnałów (w nawiasach okrągłych), których zmiana powoduje wznowienie wykonywania instrukcji proceduralnej; *statement* – instrukcja (lub grupa instrukcji), której wykonywanie jest wstrzymywane daną listą czułości.

Działanie zaprezentowanego przykładu listy czułości jest następujące: instrukcja lub grupa instrukcji *statement* będzie wstrzymana do momentu, gdy chociaż jeden z sygnałów wymienionych na liście czułości zmieni swój stan.

Zaprezentowany format składni zazwyczaj wykorzystywany jest do opisu działania układów reagujących na poziom sygnałów. Typowym przykładem takiego schematu jest zatrask (latch), czyli przerzutnik transparentny.

Poniższy przykład demonstruje wariant sumatora jednobitowego z listą czułości:

```
module adder_sen_list_1  (input a, b, cin,
                        output reg s, cout);

    always @ (a, b, cin)
        {cout,s} = a + b + cin;

endmodule
```

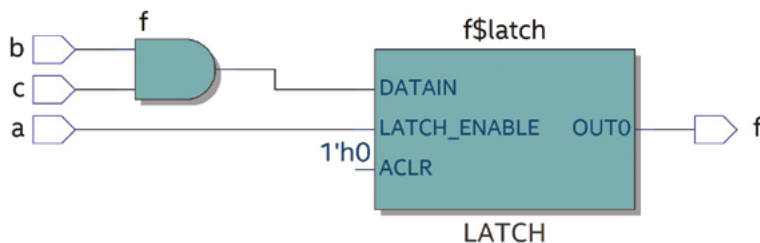
Kolejny przykład pokazuje powstawanie zatrasku na wyjściu układu kombinacyjnego w przypadku błędu opisu listy czułości:

```
module circ_latch  (input a, b, c,
                  output reg f);

    always @(a, b, c)
        if (a==1) f = b & c;

endmodule
```

Wynik syntezy modułu *circ_latch* został przedstawiony na rysunku 2.17.



RYS. 2.17. Realizacja modułu *circ_latch*

2.8.3. Lista czułości w układach kombinacyjnych

Jeśli w opisie układów kombinacyjnych na liście czułości brakuje chociaż jednego z sygnałów wejściowych, syntezyator automatycznie umieści przerzutnik na wyjściu schematu. Aby uniknąć takiej sytuacji, należy na liście czułości wymienić wszystkie sygnały wejściowe. Niemniej jednak w praktyce ta zasada często jest pomijana przez projektantów. Na przykład podczas nanoszenia zmian w projekcie dodatkowe sygnały mogą zostać dodane do wyrażenia, ale pominięte na liście czułości.

Aby zapobiec takim błędom, w języku Verilog-2001 została dodana następująca składnia:

@ * i @ (*).

Obydwa formaty są równoważne. Gwiazdka występująca po znaku @ (w nawiasach lub bez nawiasów) oznacza, że lista czułości zawiera wszystkie sygnały wejściowe danej instrukcji proceduralnej. Powyższe formaty listy czułości są wykorzystywane do opisu układów kombinacyjnych.

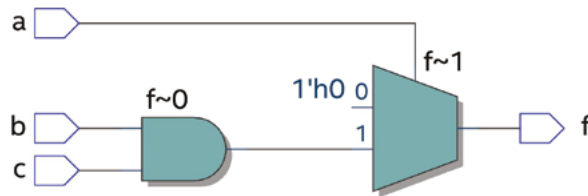
Przykład wykorzystania listy czułości ze składnią @ (*) przedstawia następujący kod:

```

module circ_no_latch      (input a, b, c,
                           output reg f);
    always @(*)
        if (a==1)    f = b & c;
        else        f = 0;
endmodule

```

Realizacja układu *circ_no_latch* została przedstawiona na rysunku 2.18.



RYS. 2.18. Realizacja modułu *circ_no_latch*

2.8.4. Lista czułości w układach sekwencyjnych

Układy sekwencyjne zazwyczaj są sterowane zboczem sygnału synchronizacji. Wśród tych układów znajdują się przerzutniki, rejestry, liczniki, automaty skończone, kontrolery itp. Do sterowania czasem przy modelowaniu takich urządzeń wykorzystuje się następujący format listy czułości:

always @ (posedge signal, negedge signal, ...)

Dla każdego sygnału na liście czułości można określić albo narastające (**posedge**), albo opadające (**negedge**) zbocze, jednak nie można wybrać obu zboczy dla tego samego sygnału. Ponadto typ zbocza powinien być określony dla każdego sygnału znajdującego się na liście. Innymi słowy, na tej samej liście sygnałów nie można mieszać sterowania poziomami i zboczami sygnałów.

Wykorzystanie listy czułości w systemach sekwencyjnych pokazane zostało na przykładzie realizacji przerzutnika typu D:

```

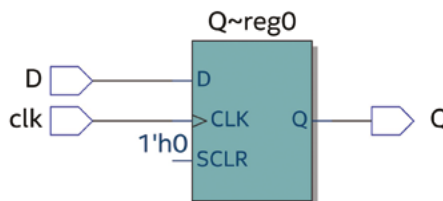
module my_DFF (input D, clk,
                output reg Q);

    always @ (posedge clk)
        Q <= D;                                // instrukcja przypisania nieblokującego

endmodule

```

Wynik syntezy modułu *my_DFF* został przedstawiony na rysunku 2.19.



RYS. 2.19. Realizacja modułu *my_DFF*

2.9. Instrukcje przypisania

W języku Verilog instrukcje przypisania dzielą się na instrukcje blokujące (oznaczone symbolem =) oraz nieblokujące (oznaczone symbolem <=). Typ instrukcji przypisania może mieć wpływ nie tylko na wynik symulacji, lecz także na wynik syntezy projektu.

2.9.1. Instrukcja przypisania blokującego „=”

Instrukcja przypisania blokującego ma następującą składnię:

variable = *expression*;

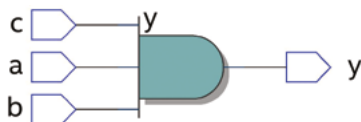
gdzie: *expression* – pewne wyrażenie; *variable* – zmienna typu **reg**, której zostanie nadana wartość obliczonego wyrażenia.

Po lewej stronie instrukcji przypisania można wykorzystać tylko zmienne typu **reg**, nawet w przypadku realizacji układów kombinacyjnych.

Instrukcja przypisania blokującego działa w następujący sposób: gdy symulator napotka daną instrukcję w kodzie, obliczane jest wyrażenie *expression*, a jego wynik zostaje przypisany zmiennej *variable*. Wewnątrz bloku **begin-end** instrukcje są wykonywane sekwencyjnie i zatrzymywane do momentu zakończenia operacji przypisania. Stąd też wywodzi się nazwa instrukcji przypisania blokującego.

W poniższym przykładzie modułu *blocking_AND* najpierw wartość zmiennej wejściowej *a* jest przypisywana do zmiennej *y*. Następnie obliczane jest wyrażenie *a & b*, a jego wartość jest ponownie przypisywana do zmiennej *y*. Ostatnia instrukcja oblicza wyrażenie *a & b & c*, którego wartość przypisuje do zmiennej wyjściowej *y*. Implementacja modułu *blocking_AND* została przedstawiona na rysunku 2.20. Zwróćmy uwagę, że w tym przykładzie zmienna wyjściowa *y* pełni funkcję zmiennej pośredniej.

```
module blocking_AND (input a, b, c, output reg y);  
    always @(*) begin  
        y = a;  
        y = y & b;  
        y = y & c;  
    end  
endmodule
```



rys. 2.20. Wyniki syntezy modułu *blocking_AND*

2.9.2. Instrukcja przypisania nieblokującego „<=”

Instrukcja przypisania nieblokującego ma następującą składnię:

variable <= *expression*;

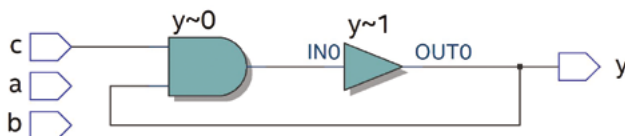
gdzie: *expression* to pewne wyrażenie, a *variable* to zmienna typu **reg**, której będzie nadana wartość wyliczona z wyrażenia.

Działanie instrukcji nieblokującego przypisania jest następujące: w momencie, gdy symulator napotka daną instrukcję, obliczane jest wyrażenie *expression*, jednak przypisanie otrzymanej wartości zmiennej z lewej strony równania zostaje odłożone do czasu zakończenia symulacji. W bloku **begin-end** sekwencyjne wykonywanie instrukcji nie jest zakłócone (nie jest blokowane), dlatego też instrukcja ta nosi nazwę przypisania nieblokującego. Innymi słowy, następna instrukcja będzie wykonywana bez oczekiwania na zakończenie przypisania.

Następujący fragment kodu przedstawia przykład wykorzystania instrukcji nieblokującego przypisania do sekwencyjnego obliczenia funkcji AND:

```
module non_blocking_AND (input a, b, c, output reg y);  
    always @(*) begin  
        y <= a;  
        y <= y & b;  
        y <= y & c;  
    end  
endmodule
```

W przedstawionym przykładzie trzy nieblokujące instrukcje przypisują wartości różnych wyrażeń do tej samej zmiennej wyjściowej *y*. Dodatkowo zmienna *y* jest uwzględniona w dwóch wyrażeniach po prawej stronie od znaku <=. Implementacja modułu *non_blocking_AND* została przedstawiona na rysunku 2.21.



RYS. 2.21. Wyniki syntezy modułu *non_blocking_AND*

Na rysunku 2.21 widać, że syntezyator zaimplementował tylko ostatnią instrukcję nieblokującą. W efekcie powstało sprzężenie kombinacyjne, co zazwyczaj jest niedopuszczalne w syntezie układów kombinacyjnych. W rezultacie wartość zmiennej *y* jest zawsze równa logicznemu 0. Aby wyjaśnić otrzymany wynik syntezy, konieczne jest zrozumienie sposobu, w jaki narzędzia syntezy interpretują przypisanie nieblokujące.

Niech x będzie pewną zmienną. Podczas syntezy nieblokujące przypisanie zmiennej x jest interpretowane w następujący sposób:

- 1) na początku bloku **always** wartość zmiennej x jest przypisywana do zmiennej tymczasowej x_entry ;
- 2) wewnątrz bloku **always** zmienna tymczasowa x_entry zastępuje zmienną x w wyrażeniach po prawej stronie od znaku $\leq$;
- 3) wprowadza się zmienną tymczasową x_exit ; w bloku **always** zmienna x_exit zastępuje zmienną x na lewo od znaku $\leq$;
- 4) na końcu bloku **always** wartość zmiennej x_exit jest przypisywana do zmiennej x .

Opisane czynności są wyjaśnione w komentarzach do poniższego kodu:

```
always @(*) begin           //  $x\_entry = x$ 
    ...
     $y \leq x \& \dots$ ;         //  $y = x\_entry \& \dots$ ;
    ...
     $x \leq \dots$ ;             //  $x\_exit = \dots$ ;
    ...
end                        //  $x = x\_exit$ ;
```

W przypadku wielu nieblokujących przypisań tej samej zmiennej syntezytor implementuje w kodzie źródłowym tylko ostatnią instrukcję.

W związku z powyższym nasz program *non_blocking_AND* jest interpretowany następująco:

```
module non_blocking_AND (input a, b, c, output reg y);
    always @(*) begin      //  $y\_entry = y$ ;
         $y \leq a$ ;           //  $y\_exit = a$ ;
         $y \leq y \& b$ ;       //  $y\_exit = y\_entry \& b$ ;
         $y \leq y \& c$ ;       //  $y\_exit = y\_entry \& c$ ;
    end                    //  $y = y\_exit$ 
endmodule
```

W rezultacie syntezytor ignoruje dwie pierwsze instrukcje nieblokujące i implementuje tylko trzecią instrukcję, co odpowiada następującemu kodowi:

```
always @(*)
     $y \leq y \& c$ ;
```

Schemat na rysunku 2.21 jest zgodny z przedstawionym kodem.

2.9.3. Specyfika syntezy instrukcji blokujących i nieblokujących

Przeanalizujemy poniższy przykład równoległej realizacji przerzutników przy użyciu instrukcji przypisania blokującego:

```
module blocking_assignments (  
    input data, clk,  
    output reg x, y, w, z);  
  
    always @(posedge clk)  
    begin  
        x = data;  
        y = x;  
        w = y;  
        z = w;  
    end  
endmodule
```

Zgodnie z przedstawionym kodem spróbujemy określić wynik syntezy. Na początku można założyć, że wartości zmiennych x , y , w i z będą formowane na wyjściach przerzutników, które są sterowane rosnącym zboczem sygnału synchronizacji clk , ponieważ cały blok jest sterowany za pomocą instrukcji czułości @ (posedge clk). Pierwsza instrukcja przypisania nada zmiennej x wartość wejścia $data$, przy czym zablokuje wykonywanie wszystkich następnych instrukcji przypisania. Druga instrukcja przypisania powinna określić wartość zmiennej x i przypisać ją do zmiennej y . Jednakże wartość zmiennej x została już ustalona jako wartość zmiennej $data$. Idąc podobnym tokiem rozumowania, można dojść do wniosku, że wszystkim zmiennym x , y , w i z powinna być przypisana wartość tej samej zmiennej wejściowej $data$. W ten sposób opisany schemat realizowany jest za pomocą czterech przerzutników, gdzie na wyjściach formowane są wartości zmiennych x , y , w i z , a na wejścia wszystkich przerzutników podawana jest wartość wejściowego sygnału $data$ (rysunek 2.22).

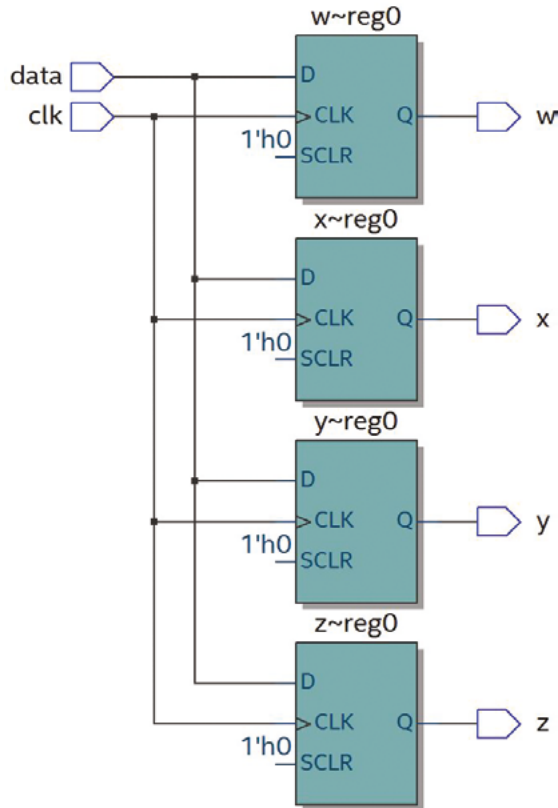
Następnie omówimy przykład sekwencyjnej realizacji przerzutników z wykorzystaniem instrukcji przypisania nieblokującego:

```
module non_blocking_assignments (  
    input data, clk,  
    output reg x, y, w, z);  
    always @(posedge clk)  
    begin  
        x <= data;  
        y <= x;  
        w <= y;
```

```

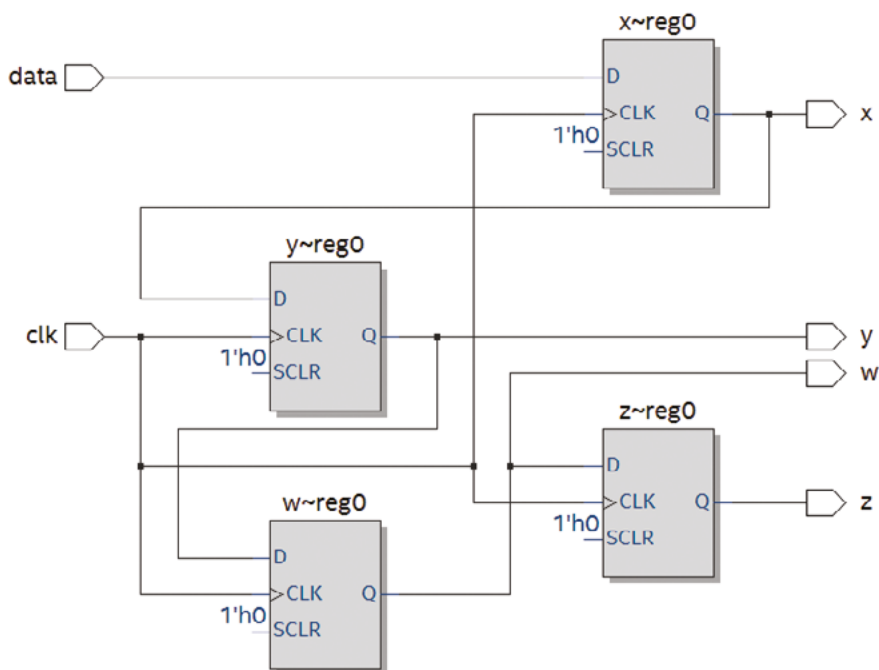
    z <= w;
end
endmodule

```



RYS. 2.22. Realizacja modułu *blocking_assignments*

Próbując przewidzieć rezultat syntezy przedstawionego kodu, można się spodziewać, że wartości zmiennych x , y , w i z będą ustawione na wyjściach przerzutników, które są sterowane narastającym zboczem sygnału synchronizacji clk , ponieważ cały blok sterowany jest za pomocą instrukcji czułości @ (**posedge** clk). Pierwsza instrukcja przypisania nadaje zmiennej x wartość wejściową $data$, przy czym pozostałe instrukcje nie są blokowane. Druga instrukcja przypisania powinna nadać zmiennej y wartość zmiennej x , jednak wartość zmiennej x nie jest jeszcze znana, gdyż dopiero w kolejnym takcie po nadejściu narastającego zbocza sygnału clk zostanie ona zaktualizowana. Dlatego wejście przerzutnika, na którym formowany jest sygnał zmiennej y , powinno być podłączone do wyjścia przerzutnika, który zwraca wartość zmiennej x . W podobny sposób można dojść do wniosku, że przerzutniki, na których wyjściach ustalane są wartości zmiennych x , y , w i z , powinny być połączone sekwencyjnie, zgodnie z tym, co zostało zaprezentowane na rysunku 2.23.



RYS. 2.23. Realizacja modułu *non_blocking_assignments*

Należy zauważyć, że syntezytor zazwyczaj ignoruje instrukcje zarządzania czasem w instrukcjach przypisania. Wyjątek stanowi instrukcja listy czułości @, która sprawdza zbocze sygnału synchronizacji. Gdy steruje ona blokiem zawierającym instrukcje przypisania blokującego, tworzony jest układ równoległych elementów pamięci, natomiast w przypadku instrukcji nieblokujących – schemat sekwencyjnie połączonych elementów pamięci.

Aby uniknąć potencjalnych wyścigów w syntezywanym układzie oraz w celu uzyskania wiarygodnych danych podczas symulacji z zerowym czasem opóźnienia, zaleca się stosowanie instrukcji przypisania blokującego (=) do opisu układów kombinacyjnych, natomiast instrukcji przypisania nieblokującego (<=) do opisu układów sekwencyjnych.

2.9.4. Wykorzystanie przypisania blokującego i nieblokującego w jednym bloku proceduralnym

Ogólnie nie zaleca się używania instrukcji blokujących i nieblokujących w tym samym bloku proceduralnym **always**. Istnieją jednak przypadki, kiedy jest to dopuszczalne.

Przyjrzyjmy się sytuacji, w której trzeba obliczyć pewną funkcję logiczną i zapisać otrzymaną wartość do rejestru. Tradycyjne podejście polega na użyciu instrukcji blokujących i nieblokujących w różnych blokach proceduralnych, co zademonstrowano

na poniższym przykładzie. Tutaj zastosowano dwa oddzielne bloki **always**, z których pierwszy implementuje układ kombinacyjny, a drugi realizuje przerzutnik. Kolejność instrukcji **always** w tym przypadku nie ma znaczenia. Implementacja modułu *nb_9_10* została przedstawiona na rysunku 2.24.

Przykład użycia instrukcji blokujących i nieblokujących w różnych blokach proceduralnych:

```

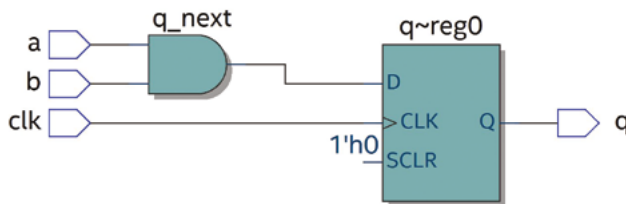
module nb_9_10 (input clk, a,b, output reg q);
    reg q_next;

    always @(*)
        q_next = a & b;

    always @(posedge clk)
        q <= q_next;

endmodule

```



RYŚ. 2.24. Wyniki syntezy modułu *nb_9_10*

Możliwe jest również umieszczenie instrukcji blokującego i nieblokującego przypisania razem w tym samym bloku **always**, co pokazano w następnym przykładzie. Jednakże w tym przypadku istotne są zarówno kolejność instrukcji w bloku **always**, jak i typ instrukcji przypisania.

```

module nb_9_11(input clk, a,b, output reg q);
    reg ab;
    always @(posedge clk)
    begin
        ab = a & b;
        q <= ab;
    end
endmodule

```

Wynik syntezy tego układu jest taki sam, jak na rysunku 2.24.

Przy stosowaniu instrukcji blokujących (=) i nieblokujących (<=) w jednym bloku proceduralnym należy przestrzegać następujących zasad:

- instrukcje blokującego przypisania (=) muszą poprzedzać w kodzie instrukcje nieblokującego przypisania (<=);
- do realizacji schematu kombinacyjnego należy stosować instrukcje blokujące (=);
- do implementacji elementów pamięci (przerzutników i rejestrów) należy używać instrukcji nieblokujących (<=);
- kolejność instrukcji blokujących (=) musi być taka, aby w momencie wykonywania każdej takiej instrukcji znane były wartości wszystkich zmiennych w wyrażeniu po prawej stronie znaku =.

2.10. Instrukcje programowania proceduralnego

Instrukcje programowania proceduralnego języka Verilog służą przede wszystkim do zmiany porządku ich wykonywania. Za pomocą tych instrukcji realizowane są rozgałęzienia przy ich sekwencyjnym wykonywaniu, tworzą się pętle, powtórzenia, umożliwia się wykonanie wybranych fragmentów kodu w zależności od określonych warunków itp. Większość instrukcji programowania strukturalnego języka Verilog została zaczerpnięta z języka programowania C, choć występują też pewne wyjątki.

W języku Verilog zdefiniowane są następujące instrukcje programowania proceduralnego: **if-else**, **case**, **casez**, **casex**, **for**, **while**, **repeat**, **forever**, **disable**.

Nie wszystkie instrukcje programowania strukturalnego są syntezywalne. W zaprezentowanych poniżej składniach instrukcji programowania proceduralnego zamiast instrukcji *statement* może występować blok instrukcji umieszczony wewnątrz nawiasów **begin-end**.

2.10.1. Instrukcja if-else

Instrukcja **if-else** umożliwia wykonanie określonych fragmentów kodu w zależności od spełnienia wyznaczonego warunku. Składnia tej instrukcji:

```
if (expression) statement  
else statement
```

gdzie: **if** i **else** – słowa kluczowe; *expression* – sprawdzany warunek; *statement* – instrukcje. Konstrukcja **else** i następująca za nią instrukcja są opcjonalne.

Działanie instrukcji **if-else** zaczyna się od obliczenia wyrażenia *expression*. Jeśli jego wartość jest prawdą (1'b1), zostaje wykonana instrukcja *statement* występująca po wyrażeniu *expression*. Natomiast jeśli wynikiem wyrażenia *expression* jest fałsz (1'b0) lub nieznana wartość (X), wykonuje się instrukcja *statement* występująca po słowie kluczowym **else**. Jeśli konstrukcja **else** nie jest obecna, wykonuje się kolejna instrukcja kodu.

Jako przykład wykorzystania instrukcji **if-else** przedstawimy parametryzowany moduł dwuwejściowego multipleksera:

```
module mux_N_if #(parameter N=8)           // N – szerokość słów
    (input [N-1:0] A, B,                     // słowa wejściowe
     output reg [N-1:0] Y,                  // wyjście
     input sel);                            // wejście sterujące

always @(*)                               // lista czułości
    if (sel==1)    Y = B;                  // instrukcja if
    else           Y = A;
endmodule
```

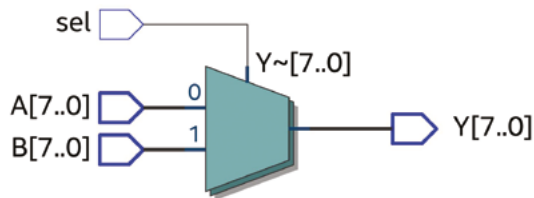
Zamiast instrukcji **if** może być użyta operacja warunkowa:

```
module mux_N_conditional #(parameter N=8) // N – szerokość słów
    (input [N-1:0] A, B,                   // słowa wejściowe
     output [N-1:0] Y,                    // wyjście
     input sel);                          // wejście sterujące

assign Y = (sel) ? B : A;

endmodule
```

Realizacja obu przykładów daje taki sam wynik przedstawiony na rysunku 2.25.

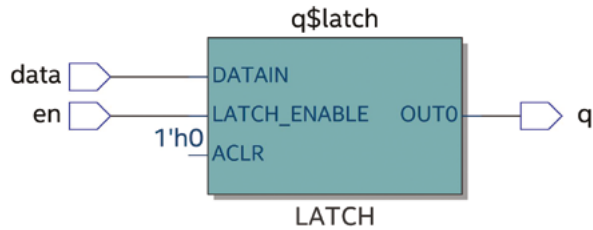


RYS. 2.25. Realizacja modułu dwuwejściowego multipleksera

Kolejny przykład demonstruje opis zatrzasku, którego realizację przedstawiono na rysunku 2.26.

```
module latch_1 (output reg q,
                input data, en);

    always @(*)
        if (en==1) q = data;
endmodule
```



RYS. 2.26. Realizacja modułu *latch_1* zatrzasku za pomocą instrukcji *if*

Zauważmy, że brak konstrukcji **else** przy operatorze **if** w opisie układu kombinacyjnego powoduje, że syntezytor na wyjściu schematu ustawia zatrząsk.

Należy zachować ostrożność podczas wykorzystywania instrukcji **if** do opisu układów kombinacyjnych. Jeżeli dla niektórych danych wejściowych nie są jawnie określone wartości wyjść układu kombinacyjnego, przyjmuje się, że dla danych wejściowych wartości wyjściowe pozostają niezmiennione, co oznacza, że wyjścia zachowują poprzednie wartości. Jednakże zachowanie wartości wyjść wymaga wykorzystania elementów pamięci. W rezultacie syntezytor tworzy na wyjściu schematu zatrząsk, a układ przestaje być kombinacyjny i przechodzi do klasy układów z pamięcią (czyli układów sekwencyjnych). Podobna sytuacja wystąpi, gdy w instrukcji **if** nie uwzględną się konstrukcji **else**.

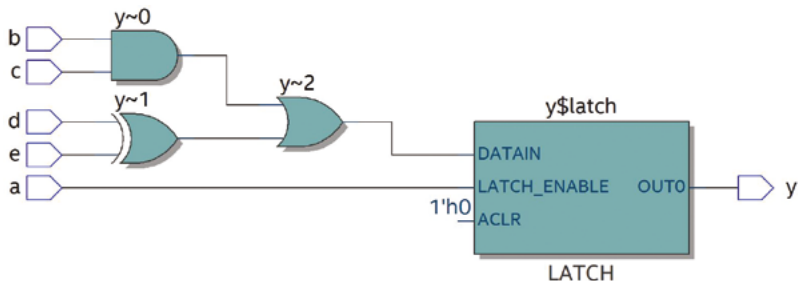
Poniżej zaprezentowany został błędny opis schematu kombinacyjnego:

```

module latch_2 (output reg y,
                input a, b, c, d, e);
    always @(*)
        if (a==1) y = b & c | d ^ e;
endmodule

```

Wynik syntezy układu został przedstawiony na rysunku 2.27.



RYS. 2.27. Realizacja modułu *latch_2*: wynik błędnego opisu układu kombinacyjnego – na wyjściu schematu zostaje ustawiony zatrząsk

Poniżej przedstawiono kilka potencjalnych sposobów wykorzystania instrukcji **if-else** do implementacji układów kombinacyjnych.

Przykład z użyciem konstrukcji **else** przy realizacji układu kombinacyjnego:

```
module comb_circuit_1 (output reg y,  
    input a, b, c, d, e);  
    always @(*)  
        if (a==1) y = b & c | d ^ e;  
        else y = 0;  
endmodule
```

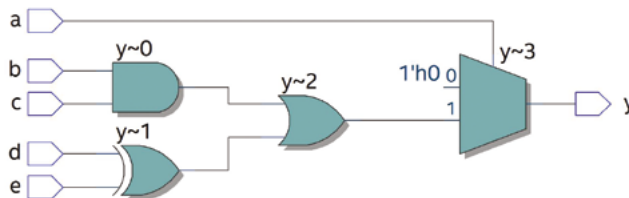
Opis układu kombinacyjnego z nadaniem początkowych wartości:

```
module comb_circuit_2 (output reg y,  
    input a, b, c, d, e);  
    always @(*) begin  
        y = 0;                // nadanie początkowej wartości  
        if (a==1) y = b & c | d ^ e;  
    end  
endmodule
```

Opis układu kombinacyjnego z nadaniem wartości sygnału sterującego:

```
module comb_circuit_3 (output reg y,  
    input a, b, c, d, e);  
    always @(*)  
        if (a==1) y = b & c | d ^ e;  
        else y = a;           // nadanie wartości sygnału sterującego  
endmodule
```

Wynik syntezy dla wszystkich przykładów jest identyczny i został przedstawiony na rysunku 2.28.



RYS. 2.28. Realizacja modułów *comb_circuit_1-3*

2.10.2. Instrukcja case

Instrukcja **case** umożliwia wykonywanie różnych fragmentów kodu w zależności od wartości wyliczonego wyrażenia. Poniżej przedstawiona jest składnia instrukcji **case**:

```
case (expression)  
    case_item:    statement;  
    ...  
    case_item, ..., case_item: statement;  
    default:      statement;  
endcase
```

gdzie: **case**, **endcase** i **default** – słowa kluczowe; *expression* – obliczane wyrażenie; *case_item* – element stały; *statement* – wykonywana instrukcja. Konstrukcja odpowiadająca słowu kluczowemu **default** nie jest obowiązkowa. Jako elementy *case_item* mogą występować stałe, wyrażenia lub wywołania funkcji stałych.

Działanie instrukcji **case** przebiega w następujący sposób: najpierw obliczane jest wyrażenie *expression*. Następnie uzyskana wartość jest porównywana kolejno z elementami stałymi *case_item*. Jeśli natrafimy na stały element, który odpowiada wartości wyrażenia *expression*, wykonywana jest instrukcja *statement*, która bezpośrednio następuje po danym elemencie stałym. Jeżeli żaden ze stałych elementów nie odpowiada wartości wyliczonego wyrażenia, to wykonuje się instrukcja znajdująca się po słowie kluczowym **default**. Jednej instrukcji *statement* może być przypisanych kilka wartości elementów stałych. Instrukcja ta będzie wykonywana, gdy wartość wyrażenia będzie odpowiadać dowolnemu z elementów stałych. Po wykonaniu instrukcji, która występuje bezpośrednio po odpowiednim elemencie stałym, pozostałe elementy nie są już sprawdzane, a wykonanie bloku **case** zostaje zakończone.

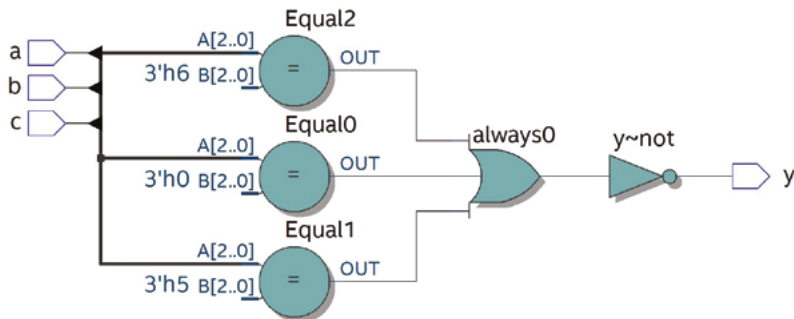
Przedstawimy przykłady wykorzystania instrukcji **case** do projektowania układów kombinacyjnych. Aby porównać możliwości instrukcji **case** oraz **if-else** w projektowaniu schematów kombinacyjnych, rozpatrzmy najpierw realizację następującej funkcji boolowskiej

$$y = f(a,b,c) = \Sigma m(a,b,c) = \Sigma(1,2,3,4,7) = \Pi(0,5,6)$$

za pomocą instrukcji **if-else**:

```
module sm_IF      (input a, b, c,  
                  output reg y);  
  
    wire [2:0] w={a,b,c}; // wektor pomocniczy, połączenie sygnałów wejściowych  
    always @(*) begin  
        if (w==3'd0 || w==3'd5 || w==3'd6) y = 1'b0;  
        else y = 1'b1;  
    end  
endmodule
```

Wynik syntezy schematu został przedstawiony na rysunku 2.29.



RYS. 2.29. Realizacja modułu *sm_IF*

Rozważmy teraz różne sposoby opisanie tej samej funkcji *y* za pomocą instrukcji **case**:

- Opis funkcji *y* za pomocą instrukcji **case** odpowiadającej tablicy prawdy:

```
module sm_Case1 (input a, b, c,  
                 output reg y);
```

```
    always @(*)
```

```
    case ({a, b, c})           // wykorzystanie operacji konkatencji
        3'b000: y = 1'b0;      // wszystkie możliwe kombinacje wejściowe
        3'b001: y = 1'b1;      // i odpowiadające im wartości wyjść
        3'b010: y = 1'b1;
        3'b011: y = 1'b1;
        3'b100: y = 1'b1;
        3'b101: y = 1'b0;
        3'b110: y = 1'b0;
        3'b111: y = 1'b1;
```

```
    endcase
```

```
endmodule
```

- Opis funkcji *y* z wykorzystaniem konstrukcji **default**:

```
module sm_Case2 (input a, b, c,  
                 output reg y);
```

```
    always @(*)
```

```
    case ({a, b, c})
        3'b000: y = 1'b0;      // sprawdzane są tylko zerowe wartości wyjścia
        3'b101: y = 1'b0;
        3'b110: y = 1'b0;
```



```

        default: y = 1'b1;    // wartość wyjścia równa 1 zostaje przyjęta domyślnie
    endcase
endmodule

```

- Wariant realizacji poprzedniego przykładu, gdy jednej instrukcji przypisanych jest kilka elementów stałych instrukcji **case**:

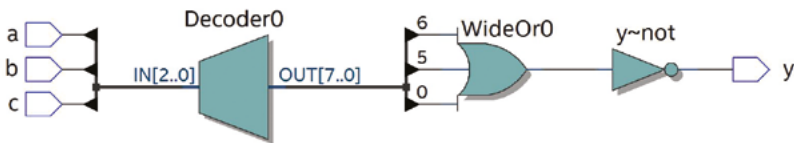
```

module sm_Case3 (input a, b, c,
                 output reg y);

    always @(*)
        case ({a, b, c})
            3'd0, 3'd5, 3'd6: y = 1'b0;    // wykorzystanie kilku elementów stałych
            default: y = 1'b1;
        endcase
endmodule

```

W wyniku syntezy powyższych przykładów otrzymany został taki sam schemat, zaprezentowany na rysunku 2.30.



RYS. 2.30. Realizacja modułów *sm_Case1-3*

Przy implementacji układów kombinacyjnych za pomocą instrukcji **case** pojawiają się te same problemy, co w przypadku użycia instrukcji **if-else**: jeśli dla pewnych kombinacji sygnałów wejściowych wartości wyjść nie są jednoznacznie określone, to syntezyator ustawi zatrzaski na wyjściach układu. Aby zapobiec wprowadzeniu niepożądanych zatrząsków na wyjściach układów kombinacyjnych, zaleca się używanie konstrukcji **default** w operatorze **case**. Możliwe jest również przypisanie wartości początkowych do wyjść:

```

always @(*)
begin
    out = 0;
    case (wyrażenie)
        ...
        case_item: out = ...;
        ...
    endcase
end

```

Istotną wadą instrukcji **case** ograniczającą jej wykorzystanie jest fakt, że poszczególne bity w wartości wyrażenia, jak również poszczególne bity elementów stałych mogą przyjmować jedynie wartości 0 albo 1. Aby rozszerzyć możliwości instrukcji **case**, do języka Verilog zostały dodane instrukcje **casez** i **casex**.

2.10.3. Instrukcje casez i casex

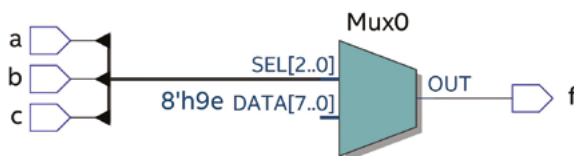
Składnie instrukcji **casez** i **casex** całkowicie odpowiadają składni instrukcji **case**, z wyjątkiem tego, że zamiast słowa kluczowego **case** wykorzystuje się słowo kluczowe **casez** lub **casex**.

Instrukcja **casez** pozwala na wykorzystanie logicznych wartości **Z** w rozpatrywanym wyrażeniu oraz w elementach stałych do oznaczania nieokreślonych (*don't care*) wartości bitów. Natomiast instrukcja **casex** umożliwia wykorzystanie dwóch wartości logicznych **Z** oraz **X** do oznaczenia nieokreślonych wartości bitów. Dodatkowo w elementach stałych instrukcji **casez** i **casex** można używać znaku zapytania (?) do oznaczenia nieokreślonych i nieznanymi wartości bitów.

Poniżej przedstawiono kilka przykładów wykorzystania instrukcji **casez** i **casex**. Pierwszy przykład ilustruje użycie wartości nieokreślonych w standardowej instrukcji **case**:

```
module sm_Case4 (input a, b, c,
                 output reg f);
    always @(*)
        case ({a, b, c})
            3'b001: f = 1'b1;
            3'b010: f = 1'b1;
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b110: f = 1'b0;
            3'b111: f = 1'b1;
            default: f = 1'bx;    // wyjście może przyjmować nieokreśloną wartość
        endcase
endmodule
```

Wynik syntezy układu został przedstawiony na rysunku 2.31.



RYS. 2.31. Realizacja modułu *sm_Case4*: przykład wykorzystania nieokreśloności do nadania wartości wyjścia w operatorze **case**

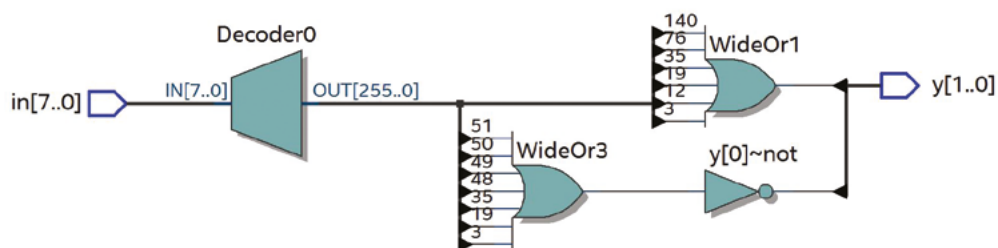
Kolejny przykład demonstruje wykorzystanie nieokreślonych wartości wejść w instrukcji **case**:

```

module decoder    (input [7:0] in,
                   output reg [1:0] y);
    always begin
        case (in)    // wektor wejściowy, którego wartość zostaje sprawdzona
            8'b001100xx : y=2'b00;
            8'b1100xx00 : y=2'b01;
            8'b00xx0011 : y=2'b10;
            8'bxx001100 : y=2'b11;
            default:    y=2'bxx;
        endcase
    end
endmodule

```

Wyniki syntezy dekodera zostały pokazane na rysunku 2.32.



RYS. 2.32. Realizacja modułu *decoder*: przykład wykorzystania nieokreślonej wartości wejść w operatorze **case**

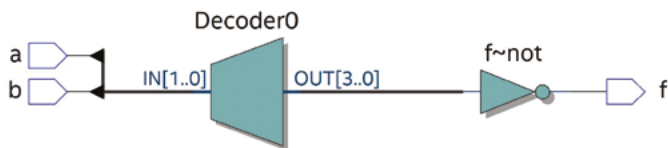
Następny przykład prezentuje wykorzystanie znaku zapytania (?) w instrukcji **case** przy opisie schematu kombinacyjnego:

```

module sm_Case5  (output reg f,
                  input a, b);
    always @(*)
        case ({a,b})
            2'b0? : f = 1'b1;
            2'b10 : f = 1'b0;
            2'b11 : f = 1'b1;
        endcase
endmodule

```

Wynik syntezy powyższego przykładu został przedstawiony na rysunku 2.33.



RYS. 2.33. Realizacja modułu *sm_Case5*: przykład wykorzystania znaku ? przy opisie wartości wejść w operatorze **case**

2.10.4. Atrybuty *full_case* i *parallel_case*

Wykorzystanie języka Verilog w praktyce projektowania pokazało, że często jego poszczególne elementy wymagają uszczegółowienia podczas ich realizacji w kompilatorach, synteźatorach, symulatorach itd. Do przekazania takich szczegółowych informacji wykorzystuje się atrybuty. Innymi słowy, atrybuty w języku Verilog określają specyficzne cechy instrukcji i konstrukcji realizowanych w konkretnych środowiskach programistycznych, na przykład w kompilatorze języka Verilog z pakietu Quartus. Składnia atrybutów w języku Verilog wygląda następująco:

(attribute, attribute,... *)*

gdzie: *(** – początek listy atrybutów; **)* – koniec listy atrybutów; *attribute* – definiowany atrybut.

Atrybuty w języku Verilog mogą być przypisywane jako prefiksy podczas deklaracji zmiennych, instancji modułów, instrukcji, portów itd. Mogą także być przypisywane jako przyrostki do instrukcji bądź wywołań funkcji. Atrybuty mogą mieć wartości. Jeżeli wartość atrybutu nie jest określona, domyślnie przyjmuje się wartość 1. Możliwe jest także jednocześnie określenie kilku atrybutów. W tym przypadku listę atrybutów oddziela się przecinkami. Standard języka Verilog nie definiuje konkretnych atrybutów, a lista atrybutów jest zwykle wprowadzana przez twórców kompilatorów.

Chociaż konkretne atrybuty w języku Verilog są ustalane przez projektantów pakietów oprogramowania, wszystkie współczesne kompilatory języka Verilog wspierają dwa atrybuty: *full_case* oraz *parallel_case*.

2.10.5. Atrybut *full_case*

Atrybut *full_case* jest używany do zapobiegania automatycznemu ustawieniu zatrząsków na wyjściach układów kombinacyjnych w przypadku, gdy w operatorze **case** nie występuje konstrukcja **default**. Atrybut *full_case* informuje synteźator, aby ustawił wyjścia schematu jako nieokreślone dla danych wejściowych, które nie zostały podane w operatorze **case**. Poniżej zaprezentowano wykorzystanie atrybutu *full_case* do opisu schematu kombinacyjnego:

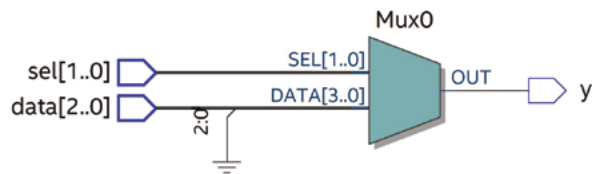
```

module use_full_case (input [2:0] data,
                     input [1:0] sel,
                     output reg y);

  always @ (*)
    (* full_case *)           // wykorzystanie atrybutu full_case
    case (sel)                // przy operatorze case
      2'b00 : y = data[0];
      2'b01 : y = data[1];
      2'b10 : y = data[2];
      // brak występowania sekwencji wejściowej 2'b11
    endcase
endmodule

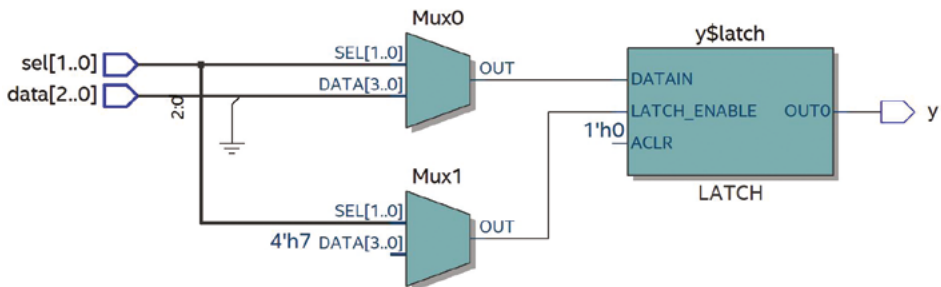
```

Wynik syntezy modułu *use_full_case* został przytoczony na rysunku 2.34.



RYS. 2.34. Realizacja modułu *use_full_case*

Realizacja tego przykładu bez atrybutu *full_case* wymaga większej liczby elementów logicznych. Została ona przedstawiona na rysunku 2.35.



RYS. 2.35. Realizacja przykładu bez atrybutu *full_case*

2.10.6. Atrybut *parallel_case*

Algorytm wykonywania instrukcji **case** wymaga zaimplementowania w schemacie kodera priorytetowego, który zakończy wykonanie instrukcji **case** w momencie znalezienia odpowiedniej wartości elementu stałego.

Istnieją dwa powody, dla których warto zmienić strategię implementacji instrukcji **case**. Po pierwsze, gdy jest pewność, że w przypadku znalezienia wartości elementu stałego, która odpowiada wartości wyrażenia, inne elementy stałe nie mogą być mu równe. Przykładem może być opis za pomocą instrukcji **case** pełnej tablicy prawdy funkcji logicznej. W takich przypadkach zwiększenie szybkości działania układu oraz czasami zmniejszenie kosztów realizacji mogą być uzasadnione przez eliminację kodera priorytetowego ze schematu implementacji instrukcji **case**.

Po drugie, w elementach stałych instrukcji **casex** oraz **casez** mogą być używane nieokreślone wartości bitów (oznaczane znakiem **?**). W takich przypadkach jedna wartość wyrażenia może odpowiadać kilku elementom stałym. Czasami wymaga się, aby instrukcje należące do takich elementów stałych były wykonywane równoległe. W podobnych sytuacjach istotne jest, aby sprawdzenie wartości wyrażenia ze wszystkimi elementami stałymi i wykonanie odpowiadających im instrukcji nie było przerywane po znalezieniu pierwszego elementu stałego, który pasuje do wartości wyrażenia.

Aby rozwiązać wspomniane problemy, można skorzystać z atrybutu *parallel_case*. Informuje on syntezy, że zamiast używania kodera priorytetowego należy zastosować równoległą realizację instrukcji dla wszystkich elementów stałych instrukcji **case**.

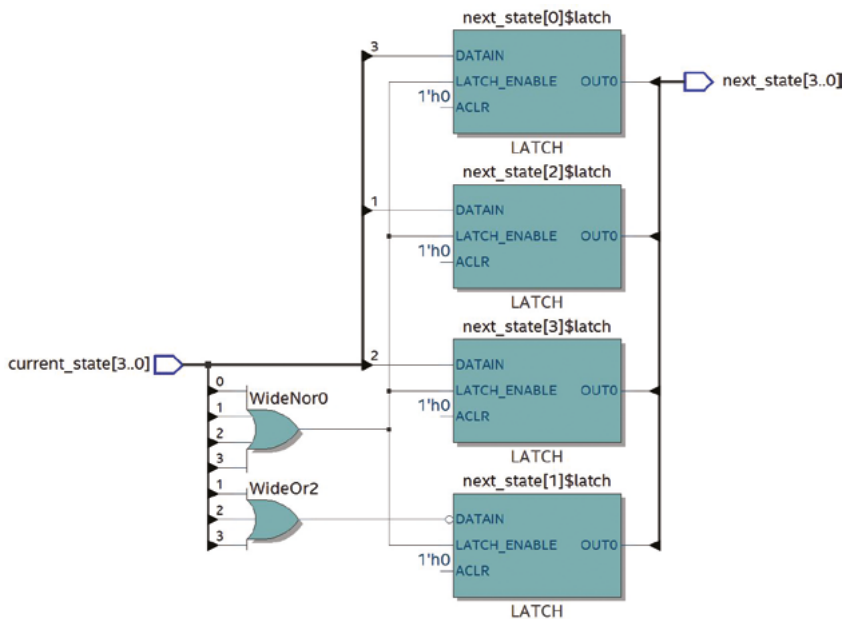
Jako przykład przedstawiono użycie atrybutu *parallel_case* do sprawdzania stanów wewnętrznych automatu skończonego zakodowanych w kodzie unarnym (*one-hot*):

```
module one_hot_assign1 (
    input [3:0] current_state, // bieżący stan automatu
    output reg [3:0] next_state); // następny stan automatu

    parameter s1 = 4'b0001, s2 = 4'b0010, s3 = 4'b0100, s4 = 4'b1000;
                                // kody sterujące

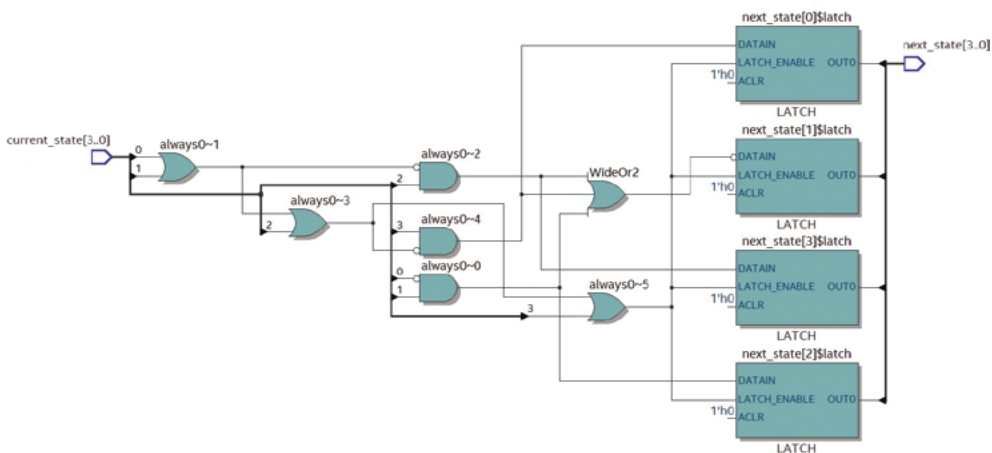
    always @(*)
    (* parallel_case *)         // wykorzystanie atrybutu parallel_case
    case (1'b1)                // poszukiwanie 1 w różnych kodach stanów
        current_state[0] : next_state = s2; // jako że jest to kod unarny, będzie
        current_state[1] : next_state = s3; // wykonana tylko jedna instrukcja
        current_state[2] : next_state = s4;
        current_state[3] : next_state = s1;
    endcase
endmodule
```

Realizacja przykładu wymaga 6 elementów logicznych i została przedstawiona na rysunku 2.36.



RYS. 2.36. Realizacja modułu *one_hot_assign1*

Realizacja tego przykładu bez wykorzystania atrybutu *parallel_case* wymaga 11 elementów logicznych i została przedstawiona na rysunku 2.37.



RYS. 2.37. Realizacja przykładu *one_hot_assign1* bez atrybutu *parallel_case*

Atrybuty *full_case* oraz *parallel_case* mogą być wykorzystywane razem:

```
module one_hot_assign2 (input [3:0] current_state,
                        output reg [3:0] next_state);
```

```
parameter s1 = 4'b0001, s2 = 4'b0010, s3 = 4'b0100, s4 = 4'b1000;
```

```
always @(*)
```

```
(* parallel_case, full_case *)
```

```
case (current_state)
```

```
  4'b???1 : next_state = s2;
```

```
  4'b??1? : next_state = s3;
```

```
  4'b?1?? : next_state = s4;
```

```
  4'b1??? : next_state = s1;
```

```
endcase
```

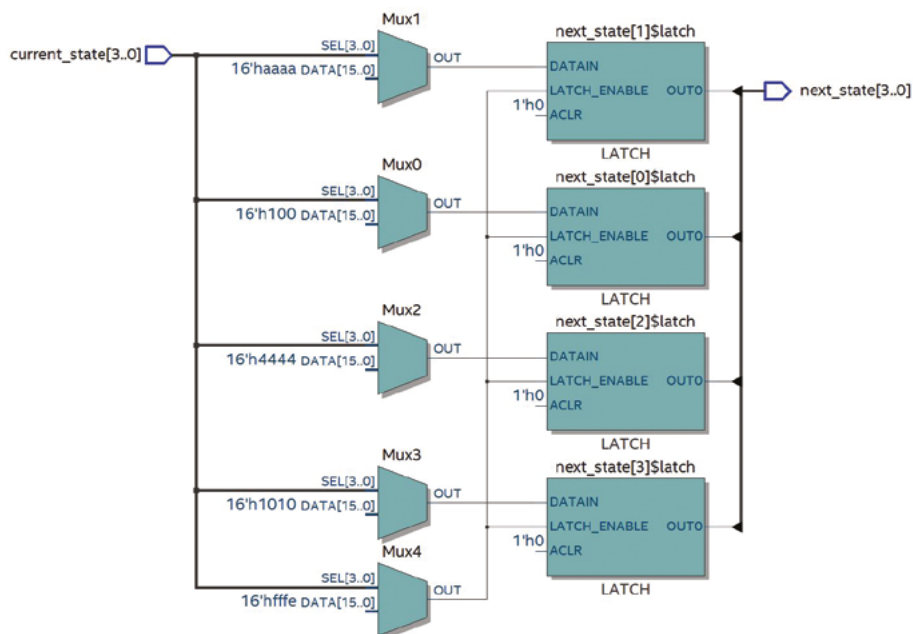
```
endmodule
```

Realizacja przykładu nie wymaga żadnych elementów logicznych i została przedstawiona na rysunku 2.38.



RYS. 2.38. Realizacja modułu *one_hot_assign2*

Realizacja tego samego przykładu z pominięciem atrybutów *full_case* oraz *parallel_case* wymaga zastosowania 9 elementów logicznych i została pokazana na rysunku 2.39.



RYS. 2.39. Realizacja przykładu z pominięciem atrybutów *full_case* i *parallel_case*

2.10.7. Instrukcja for

Instrukcja **for** służy do tworzenia pętli. Jej składnia jest analogiczna do składni odpowiedniej instrukcji w języku programowania C. Przedstawia się ona następująco:

for (*initial_assignment*; *expression*; *step_assignment*) *statement*

gdzie: *initial_assignment* – instrukcja przypisania, która wykonywana jest tylko jeden raz na początku instrukcji **for**; *expression* – wyrażenie, którego wartość jest sprawdzana przed każdą iteracją pętli; *step_assignment* – instrukcja przypisania, którą wykonuje się na końcu każdej iteracji.

Działanie instrukcji **for** przebiega w następujący sposób: jako pierwsza zostaje wykonana instrukcja *initial_assignment*, która przypisuje początkowe wartości zmiennej iteracyjnej (zazwyczaj zmienna ta jest typu **integer**). Następnie obliczana jest wartość wyrażenia *expression*. Jeśli obliczona wartość jest prawdą, wówczas wykonywana jest instrukcja *statement*; w przeciwnym razie wykonywana jest kolejna instrukcja występująca po operatorze **for**. Po zakończeniu każdej iteracji wykonywana jest instrukcja *step_assignment*, która zazwyczaj przypisuje nową wartość zmiennej iteracyjnej. Następnie ponownie sprawdza się wyrażenie *expression* i proces wykonywania instrukcji **for** jest powtarzany.

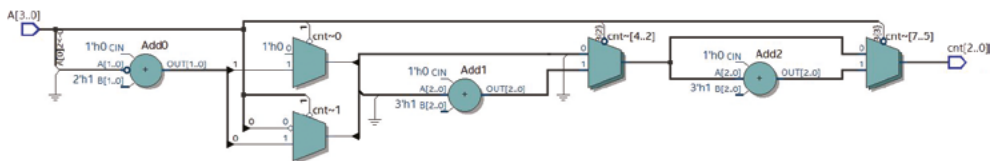
Do inkrementacji (dekrementacji) zmiennej w operatorze **for** nie wolno używać instrukcji `++i` lub `--i` znanych z języka C, ponieważ w języku Verilog instrukcje te nie występują. Dodatkowo konkretne kompilatory mogą nakładać także inne ograniczenia na zmienną iteracyjną oraz instrukcje przypisania używane w operatorze **for**. Poniżej znajduje się przykład zastosowania instrukcji **for** do określenia liczby zer w 4-bitowym słowie:

```
module count_zero1 (input [3:0] A,           // słowo wejściowe
                   output reg [2:0] cnt);    // wynik

integer i;                                  // zmienna iteracyjna

always @(*) begin
    cnt = 3'd0;                             // początkowa wartość licznika
    for (i=0; i<4; i=i+1)
        if (!A[i]) cnt = cnt + 1;           // zliczanie liczby zer
end
endmodule
```

Wynik syntezy przykładu został przytoczony na rysunku 2.40.



rys. 2.40. Realizacja modułu *count_zero1*

Analiza rysunku 2.40 wykazuje, że zastosowanie instrukcji **for** pozwala na skrócenie opisu, jednak prowadzi do złożonej realizacji sprzętowej oraz wprowadzenia wielopoziomowych układów, co może spowolnić działanie projektu.

2.10.8. Instrukcja while

Instrukcja **while** jest przeznaczona do tworzenia pętli, które są przerywane w przypadku niespełnienia określonych warunków. Składnia instrukcji **while** jest następująca: **while** (*expression*) *statement*

gdzie: **while** – słowo kluczowe; *expression* – sprawdzany warunek; *statement* – instrukcja (lub grupa instrukcji), która jest wykonywana.

Działanie instrukcji **while** polega na tym, że przed każdą iteracją pętli obliczana jest wartość wyrażenia *expression*. Jeśli jest ona prawdziwa, wykonywana jest instrukcja *statement*, w przeciwnym wypadku wykonywana jest instrukcja poza operatorem **while**. Innymi słowy, instrukcja *statement* jest wykonywana tak długo, jak długo wartość wyrażenia *expression* jest prawdziwa. Aby zapewnić wyjście z pętli **while**, w bloku instrukcji *statement* należy przewidzieć działania, które zmieniają wartość wyrażenia *expression*; w przeciwnym razie pętla będzie wykonywana bez końca.

Poniżej znajduje się przykład wykorzystania instrukcji **while** do określenia liczby zer w 4-bitowym słowie:

```
module count_zero2 (input [3:0] A,           // słowo wejściowe
                   output reg [2:0] cnt);    // wynik

integer i;           // zmienna iteracyjna
always @(*) begin
    i = 0;
    cnt = 3'd0;
    while (i < 4) begin
        if (!A[i]) cnt = cnt + 1;
        i = i + 1;
    end
end
endmodule
```

W wyniku syntezy zostanie zrealizowany dokładnie taki sam układ jak na rysunku 2.40.

2.10.9. Instrukcja repeat

Instrukcja **repeat** służy do wykonania określonej liczby powtórzeń pętli. W przeciwieństwie do instrukcji **while** wykonanie instrukcji **repeat** zawsze się kończy. Ponadto w pętli nie ma konieczności zawierania instrukcji zmieniającej wyrażenie *expression*. Jedyną wadą instrukcji **repeat** jest konieczność wcześniejszego określenia, ile razy pętla ma zostać wykonana. Składnia instrukcji **repeat** przedstawia się następująco:

repeat (*number*) *statement*

gdzie *number* – liczba powtórzeń pętli. Wartość *number* w operatorze **repeat** jest wyrażeniem, które jest obliczane tylko raz, na początku wykonania instrukcji **repeat**, i nie zmienia się w jej trakcie.

Działanie instrukcji **repeat** polega na tym, że instrukcja *statement* jest wykonywana tyle razy, ile wynosi liczba powtórzeń pętli określona przez wartość wyrażenia *number*.

Jako przykład wykorzystania instrukcji **repeat** zaprezentowano ponownie układ do wyznaczania liczby zer w 4-bitowym słowie:

```
module count_zero3 (input [3:0] A,  
                   output reg [2:0] cnt);
```

```
integer i;  
always @(*) begin  
    i = 0;  
    cnt = 3'd0;  
    repeat (4) begin  
        if (!A[i]) cnt = cnt + 1;  
        i = i + 1;  
    end  
end  
endmodule
```

Wynik syntezy odpowiada również układowi przedstawionemu na rysunku 2.40. Wykorzystanie instrukcji iteracyjnych (**for**, **while** i **repeat**) prowadzi do utworzenia wielopoziomowego, kaskadowego układu z sumatorem na każdym poziomie. Dlatego niektóre kompilatory ograniczają maksymalną liczbę iteracji na etapie syntezy projektów.

2.10.10. Instrukcja forever

Instrukcja **forever** służy do tworzenia nieskończonych pętli. Jej składnia jest następująca:

forever *statement*

gdzie *statement* to instrukcja wykonywana w nieskończonej pętli.

Instrukcja **forever** działa następująco: rozpoczyna cykliczne powtarzanie wykonania instrukcji *statement*, które trwa aż do zakończenia czasu symulacji. Instrukcja **forever** często jest używana do tworzenia nieskończonych sekwencji sygnałów, takich jak sygnał synchronizacji.

Przykład wykorzystania instrukcji **forever** prezentuje kod realizujący generator sygnału synchronizacji z okresem 50 jednostek czasu, który rozpoczyna swoją pracę po upływie 1000 jednostek czasu od początku symulacji:

```
module clock_oscillator (output reg clk);  
    initial begin  
        clk = 0;  
        #1000 forever #25 clk = ~clk;  
    end  
endmodule
```

Przedstawiony przykład nie może być zrealizowany przez syntezytor, ponieważ zabrania on implementacji nieskończonych cykli. Niemniej jednak taki opis może być przydatny podczas symulacji projektu.

2.10.11. Instrukcja disable

Instrukcja **disable** służy do przerywania wykonywania sekwencji instrukcji w nazwanym bloku (*named block*). Składnia instrukcji **disable** prezentuje się następująco:

disable *block_name*

gdzie *block_name* to nazwa bloku nazwanego, którego wykonywanie należy przerwać.

Działanie instrukcji **disable** polega na tym, że gdy symulator napotka w kodzie tę instrukcję, zostaje przerwane wykonywanie kodu w nazwanym bloku *block_name*. Sterowanie zaś jest przekazywane instrukcji po nim następującej. Instrukcja **disable** często jest używana do wymuszania wyjścia z pętli, co powoduje zakończenie pracy instrukcji **for**, **while** i innych.

Poniżej zaprezentowano przykłady wykorzystania instrukcji **disable**. Następujący fragment kodu realizuje funkcje instrukcji *continue* i *break*, znanych z języka C, za pomocą instrukcji **disable**:

begin: break

```
    for (i=0; i<n; i=i+1)
    begin: cont
        if (a==0) disable cont;    // przejście do następnej iteracji instrukcji for
                                   // substytut instrukcji continue w języku C

        // inne instrukcje

        if (a==b) disable break;    // przerwanie wykonywania bloku break
                                   // substytut instrukcji break w języku C

        // inne instrukcje

    end    // end cont
end    // end break
```

Kolejny przykład realizuje wyszukanie w wektorze bitowym indeksu pierwszej jedynki z lewej strony:

```
module index_of_left_one # (parameter N=8)
    (input [N-1:0] a,    // wektor wejściowy
     output integer y);  // zwracana wartość
    always begin: block_1    // nazwany blok block_1
        for (y=0; y<N; y=y+1)
            if (a[y]==1'b1)
                disable block_1;    // wyjście z bloku block_1
    end    // koniec bloku block_1
endmodule
```

2.11. Bloki generacyjne

Bloki generacyjne (generate blocks) w języku Verilog służą do generowania fragmentów kodu. Podobnie jak dyrektywy kompilatora w innych językach programowania, bloki generacyjne wykonują funkcje preprocesora przekształcające kod źródłowy przed jego ostateczną kompilacją. Bloki generacyjne języka Verilog przeprowadzają wstępną obróbkę kodu projektu przed jego przetwarzaniem przez symulator lub syntezyzator. Pozwalają na tworzenie sekwencji powtarzających się fragmentów kodu, które mogą się nieznacznie różnić między sobą, oraz na wybieranie fragmentów kodu do wykonania przez symulator w zależności od określonych warunków i inne zadania. Ponieważ bloki generacyjne przekształcają kod projektu przed jego wykonaniem, wszystkie używane w nich wyrażenia powinny być stałe.

W językach programowania często instrukcje preprocesora różnią się znacząco od instrukcji języka (na przykład w językach Assembler, C). W języku Verilog składnia instrukcji używanych w blokach generacyjnych odpowiada składni instrukcji samego języka Verilog.

W celu wyjaśnienia działania instrukcji generacyjnych wprowadzono pojęcia kodu źródłowego oraz kodu skompilowanego. Kod źródłowy to kod projektu w języku Verilog stworzony przez projektanta. Kod skompilowany to kod projektu po jego przetworzeniu za pomocą bloków generacyjnych. Jeśli kod źródłowy zawiera bloki generacyjne, wówczas kod skompilowany się od niego różni (może być zarówno większy, jak i mniejszy, ale zazwyczaj jest on większy).

Bloki generacyjne są definiowane wewnątrz modułu i wykorzystuje się je do generowania kodu dla tego modułu. Składnia bloku generacyjnego wygląda następująco:

```
genvar genvar_name, ...;  
generate  
    genvar genvar_name, ...;  
    generate_items  
endgenerate
```

gdzie: **genvar**, **generate** i **endgenerate** – słowa kluczowe; *genvar_name* – zmienna typu **genvar**; *generate_items* – elementy bloku generacji.

Zmienna typu **genvar** powinna być liczbą całkowitą dodatnią i służy do tworzenia fragmentów kodu kompilacji na podstawie pewnego fragmentu kodu źródłowego, działając na zasadzie zmiennej iteracyjnej wewnątrz instrukcji **for**. Zmienne typu **genvar** używane są wyłącznie w bloku generacyjnym, mogą mieć przypisaną wartość tylko w trakcie generacji kodu i nie istnieją w trakcie symulacji. Zmienne te mogą być deklarowane zarówno wewnątrz, jak i na zewnątrz bloku generacyjnego. Jeśli zmienna typu **genvar** jest deklarowana poza blokiem generacyjnym, może być również wykorzystywana w innych blokach generacyjnych.

Elementami bloku generacyjnego (*generate_items*) mogą być:

- instrukcje przypisania wartości zmiennym typu **genvar**;
- deklaracje sieci (*net*) i zmiennych (**reg**);
- instancje modułów i prymitywów;
- bloki proceduralne;
- deklaracje procedur i funkcji;
- instrukcje generacji **if-else**, **case** i **for**.

Przypisanie pewnej wartości do zmiennej typu **genvar** ma następujący format:

```
genvar_name = constant_expression;
```

gdzie *constant_expression* – wartość stała.

Należy odróżniać instrukcje bloku generacyjnego **if-else**, **case** i **for** od analogicznych instrukcji proceduralnych. Te pierwsze są wykonywane na początku kompilacji

w celu przekształcenia kodu wyjściowego, podczas gdy instrukcje proceduralne **if-else**, **case** i **for** są wykonywane w trakcie trwania symulacji.

Poniżej przedstawiono przykład wykorzystania instrukcji generacji **if-else** do wyboru funkcji realizującej różne warianty algorytmu mnożenia w zależności od rozmiaru argumentów:

```

module multiplier (a, b, product);
    parameter a_size = 8, b_size = 8;           // parametry modułu
    localparam product_size = a_size + b_size; // parametr lokalny
    input [a_size-1:0] a;                       //stary styl deklaracji portów
    input [b_size-1:0] b;
    output [product_size-1:0] product;

    generate
        if ((a_size < 8) || (b_size < 8)) // wykorzystanie algorytmu carry-look-ahead
            //ze schematem przyspieszonego przeniesienia
            CLA_mult #(a_size, b_size) m (a, b, product);
        else // wykorzystanie algorytmu Wallace-tree
            WALLACE_mult #(a_size, b_size) m (a, b, product);
    endgenerate
endmodule

```

Kolejny przykład ilustruje wykorzystanie instrukcji generacji **case** do stworzenia instancji sumatora za pomocą różnych sposobów realizacji w zależności od szerokości słowa:

```

generate
    case (WIDTH)
        1: adder_1bit x1(co, sum, a, b, ci); // 1-bitowa realizacja sumatora
        2: adder_2bit x1(co, sum, a, b, ci); // 2-bitowa realizacja sumatora
        default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
                                   // realizacja sumatora
    endcase // z przyspieszonym przeniesieniem
endgenerate

```

Przykład wykorzystania instrukcji generacji **for** do zamiany kodu Graya na kod binarny:

```

module gray_bin # (parameter N = 4)
    (input [N-1:0] gray,           // wartość wejściowa – kod Graya
     output [N-1:0] bin);         // wartość wyjściowa – kod binarny
    genvar i;                     // i – zmienna generacji

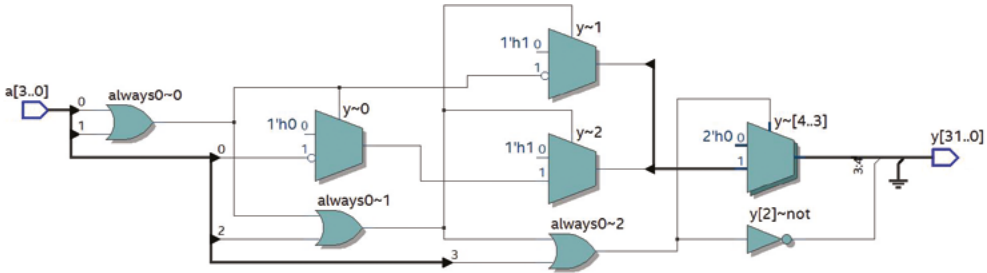
```

```

generate                                     // początek bloku generacji
  for (i=0; i<N; i=i+1) begin: bl // instrukcja generacji for
    assign bin[i] = ^gray[N-1:i]; // obliczenie kolejnej wartości
  end
endgenerate
endmodule

```

Wynik syntezy powyższego przykładu został przytoczony na rysunku 2.41.



RYŚ. 2.41. Realizacja modułu *gray_bin*

Wewnątrz pętli instrukcji **for** nie wolno umieszczać deklaracji zmiennych ani funkcji, ponieważ każde zadanie lub funkcja w operatorach generujących mogą być deklarowane tylko jeden raz.

2.12. Zadania i funkcje

Zadania i funkcje są istotnymi konstrukcjami języka Verilog, umożliwiającymi strukturalny opis kodu, co zwiększa jego czytelność i kompaktowość. Ponieważ mogą być one wielokrotnie wywoływane, ich użycie pozwala na skrócenie kodu źródłowego projektu. Dodatkowo zadania i funkcje umożliwiają wykorzystanie zmiennych lokalnych. Podobnie jak w przypadku języków programowania ogólnego przeznaczenia, zadania i funkcje są kluczowymi elementami programowania strukturalnego.

Zadania w języku Verilog przypominają procedury w innych językach programowania, natomiast funkcje w tym języku są analogiczne do funkcji w językach programowania. Co więcej, w języku Verilog istnieje koncepcja *funkcji stałych* (*constant function*).

Należy pamiętać, że w języku Verilog głównym elementem konstrukcyjnym projektu jest moduł, a procedury i funkcje są wykorzystywane jako pomocnicze konstrukcje przy opisie modułów. W poniższych przykładach zamiast pojedynczych instrukcji (*statement*) można użyć bloku instrukcji wewnątrz nawiasów operatorowych.

2.12.1. Automatyczne i statyczne zadania i funkcje

Pojęcie automatycznych i statycznych zadań i funkcji wywodzi się z języków programowania i odnosi się do sposobu, w jaki są one obsługiwane przez symulator. Wszystkie zadania i funkcje są domyślnie statyczne, co oznacza, że podczas symulacji przydzielany jest im stały obszar pamięci na wszystkie zmienne (wejściowe, wyjściowe i lokalne). Przy każdym wywołaniu zadania lub funkcji wartości zmiennych są zapisywane w tym obszarze.

W przypadku symulacji procesów równoległych mogą się pojawić problemy z pamięcią statyczną, zwłaszcza gdy istnieje możliwość równoczesnego wykonywania wielu instancji tego samego zadania lub funkcji. W takiej sytuacji wartości zmiennych z różnych instancji pracujących równolegle mogą być zapisywane w tym samym obszarze pamięci.

W dynamicznych zadaniach i funkcjach pamięć dla zmiennych jest przydzielana przy każdym wywołaniu funkcji lub zadania, a następnie zwalniana po zakończeniu jej pracy. Wykorzystanie dynamicznej alokacji pamięci pozwala na jednoczesne wykonywanie kilku instancji zadania bądź funkcji. W kontekście symulacji oznacza to, że można modelować procesy równoległe.

Jeśli w projekcie istnieje możliwość równoczesnego wywoływania różnych instancji zadań lub funkcji, zaleca się ich deklarowanie jako dynamicznych. W sytuacjach, gdy równoczesne wywoływanie nie jest wymagane, zadania i funkcje mogą być zadeklarowane jako statyczne.

2.12.2. Zadania

Jak wcześniej wspomniano, zadania w języku Verilog przypominają procedury w innych językach programowania. Istnieje jednak różnica: w tych innych językach wartości argumentów przekazywane są do procedury za pomocą listy parametrów, podczas gdy w języku Verilog korzysta się z listy portów do przekazywania wartości wejściowych i wyjściowych.

Zadania są deklarowane wewnątrz modułu i stanowią elementy lokalne w jego kontekście. Mogą być wywoływane z bloków proceduralnych, takich jak **initial** czy **always**, a także z innych zadań.

Zadania mogą mieć dowolną liczbę portów wejściowych, wyjściowych oraz dwukierunkowych, a także mogą być pozbawione portów (analogicznie do modułów). Wewnątrz zadań można używać instrukcji zarządzania czasem, takich jak **#**, **@** lub **wait**.

Deklaracja zadania ma następującą składnię:

```
task automatic task_name (  
    port_declaration port_name, port_name, ...,  
    port_declaration port_name, port_name, ...);  
    local_variable_declarations
```

procedural_statement

endtask

gdzie: **task**, **automatic** i **endtask** – słowa kluczowe; *task_name* – nazwa zadania; *port_declaration* – deklaracja portów; *port_name* – nazwa portu; *local_variable_declaration* – deklaracja zmiennych lokalnych; *procedural_statement* – instrukcja (blok instrukcji) tworząca ciało zadania.

Stara forma deklaracji zadania wygląda następująco:

task automatic *task_name*;

port_declaration port_name, port_name, ...;

port_declaration port_name, port_name, ...;

local_variable_declarations

procedural_statement or statement_group

endtask

Konstrukcja **automatic** nie jest obowiązkowa. Wskazuje jedynie, że procedura będzie korzystała z dynamicznie przydzielanej pamięci, co umożliwi równoległą pracę kilku instancji tego samego zadania.

Konstrukcja *port_declaration* może przyjmować następujące składnie:

port_direction **reg signed range** oraz

port_direction port_type

Występujące tu elementy **reg**, **signed** oraz *range* są opcjonalne. Konstrukcja *port_direction* określa kierunek przekazu danych, może więc przyjmować wartości: **input**, **output** albo **inout**. Kombinacja słów kluczowych **reg** oraz **signed** deklaruje zmienną typu **reg** ze znakiem. Konstrukcja *range* ma składnię [*msb:lsb*] i określa zakres bitowy zmiennej. Konstrukcja *port_type* może przyjmować wartości **integer**, **time**, **real** lub **realtime**.

Wywołanie zadania wygląda następująco:

task_name (*signal*, *signal*, ...);

gdzie: *task_name* – nazwa zadania; *signal*, *signal* – spis podłączanych sygnałów.

Poniżej znajduje się przykład deklaracji zadania, które czyta dane z pamięci:

```
module read_mem_PC (  
    input [15:0] PC,  
    output [31:0] IR,  
    input read_grant,  
    input clock,  
    input [31:0] data_bus,  
    output reg read_request,  
    output reg [15:0] addr_bus);
```

```
// definicja zadania read_mem, które czyta dane z pamięci
task read_mem (input [15:0] address,      // wartość adresu
               output [31:0] data);      // zwracana wartość

    begin
        read_request = 1;                // ustalenie flagi „prośba o odczyt”
        wait (read_grant)                // oczekiwanie na pozwolenie do odczytu
        addr_bus = address;              // przekazanie adresu na szynę adresową
        data = data_bus;                 // odczyt danych z szyny danych
        #5 addr_bus = 16'bz;             // przejście szyny adresowej w stan wysokiej
                                         // impedancji
        read_request = 0;                // zdjęcie flagi „prośba o odczyt”
    end
endtask

always @ (posedge clock)
    read_mem (PC, IR);                  // wywołanie zadania
endmodule
```

2.12.3. Funkcje

Funkcje w języku Verilog są bardzo podobne do funkcji w języku C, z tym że zamiast listy zmiennych, wykorzystuje się listę portów. Funkcja, podobnie jak zadanie, jest deklarowana wewnątrz modułu i jest lokalna w stosunku do danego modułu. W przeciwieństwie do zadań funkcje zawsze zwracają wartość, która jest utożsamiana z nazwą funkcji. Funkcję można wywołać w dowolnym miejscu, gdzie można używać wartości wyrażeń. Każda funkcja powinna mieć co najmniej jeden port wejściowy, nie może zaś mieć portów wyjściowych lub dwukierunkowych. Funkcje nie powinny też zawierać instrukcji zarządzania czasem (**#**, **@** i **wait**) ani instrukcji przypisania nieblokującego (**<=**). Zakłada się, że funkcje zawsze są wykonywane w zerowym czasie symulacji (0.0).

Deklaracja funkcji ma następującą składnię:

```
function automatic range_or_type function_name (
    input range_or_type port_name, port_name, ...,
    input range_or_type port_name, port_name, ...);
    local_variable_declarations
    procedural_statement
endfunction
```

gdzie **function**, **automatic** oraz **endfunction** – słowa kluczowe. Konstrukcje **automatic** i **range_or_type** są opcjonalne. Słowo kluczowe **automatic** określa dynamiczne przydzielenie pamięci dla funkcji. Umożliwia to symulatorowi jednoczesne, równoległe wykonywanie kilku instancji funkcji (wywołań) oraz wykorzystywanie rekurencji.

Stary styl deklaracji funkcji ma następującą składnię:

```
function automatic [range_or_type] function_name;  
    input range_or_type port_name, port_name, ...;  
    input range_or_type port_name, port_name, ...;  
    local_variable_declarations  
    procedural_statement or statement_group  
endfunction
```

Konstrukcja *range_or_type* określa typ zwracanej przez funkcję wartości oraz typ portów wejściowych. Gdy konstrukcja ta nie występuje, deklarowana jest 1-bitowa zmienna typu **reg**. Konstrukcja *range_or_type* może przyjmować następujące składnie:

```
signed [msb : lsb]  
reg signed [msb : lsb]  
integer, time, real or realtime
```

gdzie: **signed** określa zmienną ze znakiem w kodzie dopełnień do dwóch; **reg**, **integer**, **time**, **real** i **realtime** opisują typy zmiennych; *msb* oraz *lsb* mogą być liczbami, stałymi, wyrażeniami lub stałymi funkcjami.

Wywołanie funkcji ma następującą składnię:

```
function_name (signal, signal, ...)
```

gdzie: *function_name* – nazwa funkcji; *signal*, *signal* – lista podłączanych sygnałów.

Poniżej podany jest przykład deklaracji funkcji do obliczenia silni:

```
module fact_1 #(parameter LIMIT=32)  
    (input [31:0] data,  
    output reg [63:0] result);  
  
// opis funkcji do wyliczania silni  
function automatic [63:0] factorial (input reg [31:0] n);  
    if (n <= 1)    factorial = 1;  
    else            factorial = n * factorial (n-1);    // wywołanie rekurencyjne  
endfunction  
  
always  
    if (data <= LIMIT)    result = factorial(data);    // wywołanie funkcji factorial  
    else                    result = 0;  
endmodule
```

Należy zauważyć, że nie wszystkie kompilatory obsługują rekurencyjne wywołanie funkcji.

2.12.4. Funkcje stałe

Funkcje stałe w języku Verilog przypominają funkcje preprocesora stosowane w innych językach programowania. Innymi słowy, funkcja stała powinna zwrócić określoną wartość przed rozpoczęciem symulacji. W związku z tym istnieje kilka ograniczeń dotyczących ich deklaracji i wykorzystania w porównaniu ze standardowymi funkcjami w języku Verilog:

- w funkcjach stałych mogą być używane tylko zmienne lokalne;
- nie można stosować sieciowych (*net*) typów danych;
- wartości parametrów wykorzystywanych przez funkcję muszą być określone przed jej wywołaniem;
- parametry nie mogą być modyfikowane za pomocą instrukcji **defparam**;
- wywołania funkcji stałych nie są dozwolone podczas deklaracji rozmiaru portu;
- funkcje systemowe i procedury nie mogą być wywoływane w funkcjach stałych;
- w funkcjach stałych nie wolno także korzystać z hierarchicznych odnośników i nazw.

Natomiast funkcje stałe mogą wywoływać inne funkcje stałe.

Przyjrzyjmy się przykładowi wykorzystania funkcji stałej do określenia rozmiaru adresu pamięci:

```
module ram_model                                // moduł jednoportowej synchronicznej
                                                // pamięci
    #( parameter data_width = 8,                // szerokość słowa
      ram_depth = 256)                          // przestrzeń adresowa
    (input [adder_width-1:0] address,            // szyna adresu
     input r_w, clk,                             // sygnały sterowania
     inout reg [data_width-1:0] data);          // szyna danych

    localparam adder_width = clogb2(ram_depth)-1; // szerokość szyny
adresowej

    /* stała funkcja clogb2, która zgodnie z rozmiarem przestrzeni adresowej
    ram_depth oblicza wystarczającą szerokość szyny adresowej */

    function integer clogb2(input [31:0] value);
        for (clogb2=0; value>0; clogb2=clogb2+1)
            value = value>>1;
    endfunction

    reg [data_width-1:0] RAM_mem[0:ram_depth-1]; // deklaracja pamięci

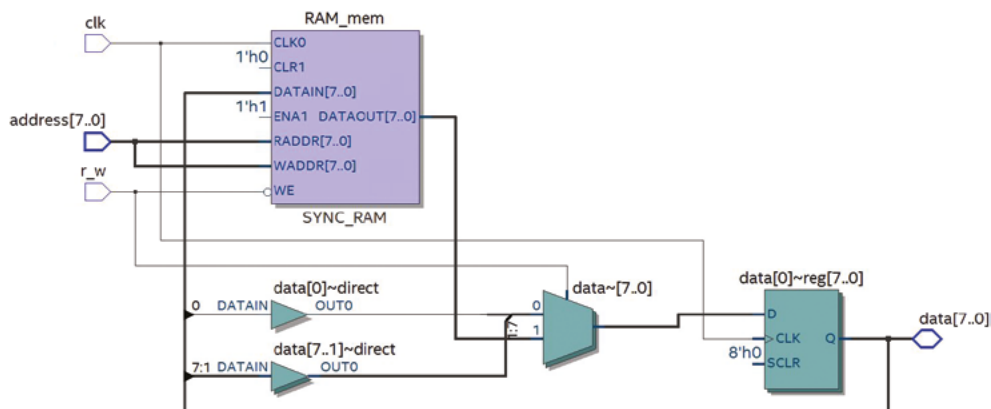
    always @(posedge clk) begin                  // synchronizacja pamięci
        if (r_w) data = RAM_mem[address];        // odczyt z pamięci
```

```

else RAM_mem[address] = data;      // zapis do pamięci
end
endmodule

```

Wynik syntezy przykładu został przytoczony na rysunku 2.42.



RYS. 2.42. Realizacja modułu *ram_model*

2.13. Zadania i funkcje systemowe

W języku Verilog, podobnie jak w języku programowania C, istnieje wiele wbudowanych zadań i funkcji, które nazywane są *systemowymi*. Systemowe zadania i funkcje w języku Verilog zazwyczaj rozpoczynają się od znaku dolara \$. Standard języka Verilog-2001 określa niewielką liczbę systemowych zadań i funkcji. Niemniej twórcy komputerowych systemów projektowania układów cyfrowych często definiują dodatkowe zadania i funkcje systemowe, na przykład do wyświetlenia na ekranie diagramów czasowych. Dodatkowo użytkownicy mogą sami tworzyć systemowe zadania i funkcje za pomocą *interfejsu języka programowania Verilog (Programming Language Interface – PLI)*.

Zadania i funkcje systemowe języka Verilog-2001 można znaleźć w [14]. Tutaj rozważymy jedynie funkcje ładowania pamięci **\$readmemb** i **\$readmemh** obsługiwane przez syntezytor Quartus. Funkcje te przeznaczone są do ładowania zawartości do pamięci projektu i mają następującą składnię:

\$readmemb ("file_name", variable_array, start_address, end_address);

\$readmemh ("file_name", variable_array, start_address, end_address);

gdzie: *file_name* – nazwa pliku, z którego dane są wczytywane i następnie zapisywane do pamięci; *variable_array* – dwuwymiarowa macierz przedstawiająca blok pamięci; *start_address* i *end_address* – początkowy i końcowy adres elementów pamięci, gdzie będą ładowane dane.

Adresy *start_address* oraz *end_address* są parametrami opcjonalnymi. Plik, z którego dokonywany jest odczyt danych, reprezentuje plik tekstowy, w którym dane powinny być zapisane w formacie dwójkowym (dla funkcji **\$readmemb**) lub szesnastkowym (dla funkcji **\$readmemh**).

Przykład wykorzystania funkcji **\$readmemh**:

```
reg [7:0] mem [1:256];           // określenie pamięci z 256 8-bitowych słów
initial $readmemh („mem.data”,mem); // zapis zawartości całej pamięci
initial $readmemh („mem.data”,mem,16); // zapis od adresu 16 do 256
initial $readmemh („mem.data”,mem,128,1); // zapis od adresu 128 do 1
```

2.14. Dyrektywy kompilatora

Dyrektywy kompilatora to polecenia wskazujące mu, jak powinien interpretować kod źródłowy projektu napisany w języku Verilog. Wszystkie dyrektywy kompilatora rozpoczynają się od znaku grawisa (`). Ponieważ dyrektywy języka Verilog nie są instrukcjami, na końcu linii z daną dyrektywą znak średnika (;) nie jest wymagany.

Zakres działania każdej dyrektywy kompilatora nie jest ograniczony do danego modułu czy pliku. Jeżeli kompilator spotyka w kodzie źródłowym dyrektywę, jej działanie obowiązuje tak długo, dopóki inna dyrektywa nie zmodyfikuje bądź anuluje działania poprzedniej. Dyrektywy kompilatora standardowego języka Verilog-2001 można znaleźć w [14].

2.15. Podsumowanie rozdziału

Język Verilog jest językiem opisu sprzętu służącym zarówno do syntezy systemów cyfrowych, jak i do symulacji ich działania.

Moduł jest podstawową konstrukcją w języku Verilog. Można go wykorzystać, tworząc *instancję* modułu. Jego elementami mogą być: deklaracje typów danych, deklaracje parametrów, instancje modułów, instancje prymitywów, bloki generacyjne, bloki proceduralne, przypisania ciągłe, zadania, funkcje oraz bloki specyfikacji.

W języku Verilog występują dwa podstawowe style opisu projektu – *behawioralny* (funkcjonalny) i *strukturalny*. Ten pierwszy służy do opisu funkcjonowania projektu, podczas gdy drugi opisuje projekt w postaci komponentów struktury (instancji modułów i prymitywów) oraz połączeń między nimi. Dopuszczalny jest również mieszany, *behawioralno-strukturalny* styl opisu projektu.

Dozwolone są dwie notacje opisu portów modułu: nowy i stary styl. Sygnały mogą być przypisane do portów modułu za pomocą lokalizacji lub nazwy portu. W przypadku, gdy niektóre porty instancji modułu nie będą podłączone, sygnały mogą być pominięte.

Parametry modułu pozwalają na znaczne zwiększenie elastyczności korzystania z modułu.

Język Verilog umożliwia równoległe wykonywanie swoich konstrukcji podczas modelowania. Do takich konstrukcji należą: instancje modułów, instancje prymitywów, instrukcje przypisania ciągłego (**assign**), bloki proceduralne (**initial**, **always**).

Sygnałom w języku Verilog odpowiadają sieci (*nets*). Sygnał może przyjąć cztery wartości: **0** – zero logiczne lub *fałsz*, **1** – jedynka logiczna lub *prawda*, **x** – wartość nieokreślona (*don't care*); **z** – stan wysokiej impedancji.

Liczby całkowite w języku Verilog są reprezentowane w systemie dwójkowym, ósemkowym, dziesiętnym lub szesnastkowym. Liczby rzeczywiste mogą być zapisywane z kropką dziesiętną lub w postaci wykładniczej.

Język Verilog ma dwie klasy podstawowych typów danych: sieci (*net*) i zmienne (**reg**). Sieciowe typy danych *net* są używane do tworzenia połączeń pomiędzy częściami projektu. Zmienny typ danych (**reg**) wykorzystuje się do ich tymczasowego przechowywania. Zasady stosowania typów danych *net* i **reg** są następujące: gdy sygnał sterowany jest wyjściem modułu, wyjściem prymitywu lub definiowany przez instrukcję przypisania ciągłego (**assign**), wykorzystywane są typy danych *net*; gdy wartość sygnału ustalana jest wewnątrz bloku proceduralnego (**initial** lub **always**), stosowany jest typ danych zmiennych (**reg**).

Wartość sygnału sieci jest definiowana za pomocą następujących typów obwodów: **wire (tri)**, **wand (triand)**, **wor (trior)**, **tri0** i **tri1**.

Typy danych **parameter**, **localparam**, **specparam**, **genvar** i **event** pełnią funkcje pomocnicze i nie należą do tych podstawowych. Typ danych **parameter** umożliwia ustawienie określonych wartości stałych dla modułów, **localparam** pozwala określić wartości parametrów lokalnych działających tylko w obrębie modułu, **genvar** zaś definiuje zmienną, która jest używana tylko wewnątrz pętli generowania kodu w blokach generujących. Łańcuchy znaków nie są obsługiwane przez syntezatory i służą do wysyłania wiadomości tekstowych podczas modelowania.

Sieciowe typy danych (*net*) i zmienne (**reg**) mogą być zadeklarowane jako *skalarne* lub *wektorowe*. Skalar ma szerokość jednego bitu, natomiast wektory mają szerokość większą niż jeden bit.

Tablica jest przeznaczona do definiowania liczby i organizacji strukturalnej zadeklarowanych elementów. Liczba wymiarów tablicy w języku Verilog nie jest ograniczona. *Pamięć* w języku Verilog jest reprezentowana za pomocą jednowymiarowych tablic zmiennych typu **reg**.

Wyrażenia łączą operatory i operandy, aby utworzyć wynik. W języku Verilog wyrażenia są obliczane w kolejności priorytetów operacji. Nawiasy okrągłe służą do zmiany kolejności obliczeń.

Składnia większości operatorów w języku Verilog odpowiada składni występującej w języku programowania C. Operatory w języku Verilog są podzielone na następujące grupy:

- operatory *bitowe* są stosowane w opisie projektów do manipulacji bitami i wykonują operacje logiczne na poszczególnych bitach wektorów oraz pól bitowych;

- operatory *redukcji* skracają wektor bitowy do jednego bitu, stosując odpowiednią operację logiczną do każdego bitu wektora;
- operatory *logiczne* są używane w wyrażeniach logicznych. Wynikiem tych operatorów jest 1-bitowa wartość, przyjmująca 1, gdy wyrażenie jest *prawdą*, lub 0 (bądź **X**) – gdy jest to *fałsz*;
- operatory *relacji* umożliwiają porównanie wartości operandów i określenie ich relacji (na przykład większość, mniejszość, równość);
- operatory *identyczności* w języku Verilog są reprezentowane przez dwa operatory: `===` i `!==`. Różnią się one od operatorów równości (`==`) i nierówności (`!=`) tym, że pozwalają na sprawdzenie wszystkich czterech możliwych wartości logicznych każdego bitu operandu: 0, 1, **X** i **Z**;
- do operatorów *różnych* w języku Verilog należą: operator warunkowy, operator konkatenacji, operator powtórzenia oraz operator wystąpienia zdarzenia.

Przed wykonaniem operacji wszystkie operandy zostają rozszerzone do rozmiaru większego wektora po obu stronach znaku operacji. Wartości bez znaku są uzupełniane zerami, natomiast wartości ze znakiem są rozszerzane bitem znaku. Operatory konkatenacji i powtórzeń są wykonywane przed wyrównaniem operandów. Instrukcje arytmetyczne wykonywane są zgodnie z następującymi zasadami: jeśli którykolwiek z operandów jest liczbą rzeczywistą, stosowana jest arytmetyka liczb zmiennoprzecinkowych; jeśli którykolwiek z operandów jest wartością bez znaku, stosowana jest arytmetyka bez znaku; jeśli wszystkie operandy są wartościami ze znakiem, stosowana jest arytmetyka liczb ze znakiem.

Przypisanie wartości zmiennym (**reg**) przypomina instrukcję przypisania w językach programowania. Natomiast o wartości sygnału sieci (*net*) decyduje rodzaj obwodu, czyli sposób podłączenia źródeł sygnału do sieci, takie jak OR, AND, otwarty kolektor itp.

Instrukcja *przypisania ciągłego* ma dwa formaty: jawny i niejawny. Gdy wartość sieci lub zmiennych w prawej części instrukcji przypisania ulega zmianie, wyrażenie jest przeliczane, a otrzymana wartość jest przypisywana do sieci w lewej części instrukcji **assign**. Instrukcja przypisania pozwala opisywać układy kombinacyjne w postaci wyrażen logicznych bez znajomości struktury układu.

Język Verilog ma dwie *instrukcje proceduralne*, **initial** i **always**, które służą do tworzenia bloku proceduralnego. Reprezentuje on grupę instrukcji ujętych w nawiasy operatorowe. Linie kodu wewnątrz nawiasów operatorowych **begin-end** są wykonywane sekwencyjnie, natomiast instrukcje wewnątrz grupy **fork-join** są wykonywane równolegle.

Blokom proceduralnym można przypisać nazwy, wówczas jest to blok *nazwany*. W przeciwieństwie do bloków bez nazwy mają one kilka dodatkowych cech: ich nazwy mogą być używane w ścieżce hierarchicznej przy wyszukiwaniu konkretnego elementu w hierarchii projektu; mogą zawierać deklaracje swoich zmiennych lokalnych i parametrów; wykonywanie instrukcji bloków nazwanych może zostać przerwane za pomocą instrukcji proceduralnej **disable**.

Instrukcja czułości @ pozwala na wstrzymanie wykonania kolejnej instrukcji (lub grupy instrukcji) do momentu wystąpienia przynajmniej jednego z określonych zdarzeń. Te zdarzenia mogą być spowodowane zmianą poziomu sygnału lub wystąpieniem zbocza sygnału.

W języku Verilog instrukcje przypisania dzielą się na *blokujące* (=) i *nieblokujące* (<=). Typ instrukcji przypisania może mieć wpływ na wyniki symulacji i syntezy. Zaleca się stosowanie blokującej instrukcji przypisania (=) do opisu układów kombinacyjnych, a nieblokującej instrukcji przypisania (<=) do opisu układów sekwencyjnych.

Język Verilog zawiera następujące instrukcje programowania proceduralnego: instrukcję warunkową **if-else**; instrukcje wyboru **case**, **casez** i **casex**; instrukcje pętli **for**, **while**, **repeat** oraz **forever**; instrukcję przerwania **disable**.

Instrukcja **if-else** umożliwia wykonanie określonych fragmentów kodu, gdy zostanie spełniony podany warunek. Każda instrukcja **if** może zostać zastąpiona operacją warunkową. Instrukcja **case** pozwala na wykonanie różnych fragmentów kodu w zależności od wartości obliczonego wyrażenia. Instrukcja **casez** umożliwia wykorzystanie wartości logicznej **Z** do oznaczenia nieokreślonych (*don't care*) wartości bitów w obliczanym wyrażeniu oraz w elementach stałych, a instrukcja **casex** pozwala dodatkowo na wykorzystanie wartości **X**.

Atrybuty języka Verilog określają specyficzne cechy instrukcji i konstrukcji, które są zaimplementowane w konkretnym kompilatorze. Atrybuty są definiowane przez projektantów narzędzi programistycznych, jednak wszystkie kompilatory Verilog wspierają dwa atrybuty: *full_case* i *parallel_case*. Atrybut *full_case* wskazuje syntezytorowi, by ustawiał wyjścia schematu jako nieokreślone (*don't care*) dla danych wejściowych, które nie zostały podane w instrukcjach **case**. Atrybut *parallel_case* nakazuje syntezytorowi, aby zamiast realizacji koda priorytetowego zapewnił równoległą implementację instrukcji dla wszystkich elementów stałych instrukcji **case**.

Instrukcja **for** jest przeznaczona do tworzenia pętli i ma składnię podobną do języka programowania C. Instrukcja **while** wykorzystywana jest do wykonania pętli w przypadku spełnienia warunku. Instrukcja **repeat** służy do powtarzania pętli określoną ilość razy. Instrukcja **forever** jest przeznaczona do organizacji nieskończonych pętli.

Polecenie **disable** jest wykorzystywane do zatrzymania wykonywania sekwencji instrukcji w nazwanym bloku.

Bloki generujące służą do generowania fragmentów kodu. Elementami bloku generującego (*generate_items*) mogą być: instrukcje przypisania wartości zmiennej typu **genvar**, deklaracje sieci i zmiennych (**reg**), instancje modułów i prymitywów, bloki proceduralne, deklaracje zadań i funkcji, instrukcje generacji **if-else**, **case** oraz **for**.

Zadania w języku Verilog przypominają procedury, natomiast funkcje są podobne do funkcji w innych językach programowania. Dodatkowo w języku Verilog istnieje pojęcie *funkcji stałych*, które zwracają wartość stałą. W języku Verilog występuje również wiele predefiniowanych zadań i funkcji, które są określane jako *systemowe*. Nazwy zadań i funkcji systemowych Verilog zaczynają się od znaku dolara (\$).

Dyrektywy kompilatora są poleceniami, które wskazują mu sposób interpretacji kodu źródłowego projektu.

3. System projektowania Quartus Prime

W tym rozdziale omówione zostaną podstawowe etapy realizacji projektów opisanych w języku Verilog w systemie projektowania Quartus Prime.

3.1. Podstawowe etapy realizacji projektów na układach FPGA

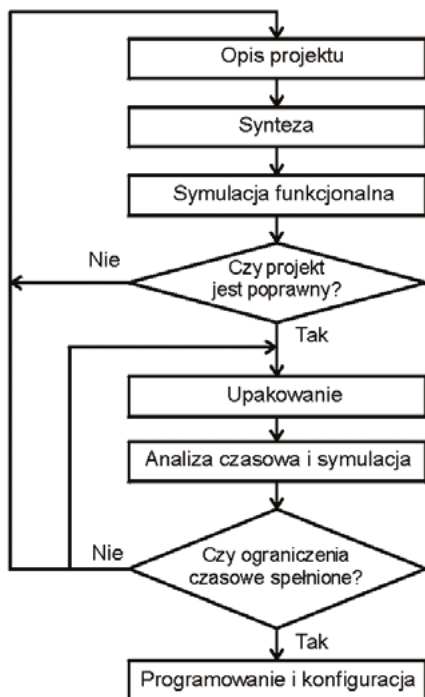
System zautomatyzowanego projektowania Quartus Prime stanowi kompleksowe narzędzie wspierające projektowanie systemów cyfrowych opartych na programowalnych układach logicznych FPGA (*Field Programmable Gate Arrays*) oraz systemów na układach SoC (*System-on-Chip*).

Etapy procesu projektowania systemów cyfrowych przy wykorzystaniu układów FPGA przedstawione zostały na rysunku 3.1.

Proces projektowania składa się z następujących podstawowych etapów:

- *wprowadzenie projektu (Design Entry)* – polega na opisie projektu za pomocą jednego z języków opisu sprzętu (Verilog, VHDL) lub zdefiniowaniu schematu projektu z wykorzystaniem edytora graficznego;
- *synteza (Synthesis)* – przekształcenie opisu projektu na sieć elementów logicznych (*logic elements* – LE) zawartych w strukturze FPGA;
- *symulacja funkcjonalna (Functional Simulation)* – modelowanie zsyntezowanego układu w celu weryfikacji poprawności funkcjonowania, bez uwzględnienia opóźnień czasowych sygnałów;
- *upakowanie (Fitting)* – odwzorowanie elementów logicznych projektu w strukturę docelowego (*target*) układu FPGA (konkretnego układu, na podstawie którego będzie realizowany projekt), zaprojektowanie połączeń pomiędzy poszczególnymi blokami logicznymi oraz wygenerowanie *listy połączeń (netlist)* projektu;
- *symulacja czasowa (Timing Simulation)* – modelowanie upakowanego schematu w celu sprawdzenia jego poprawności funkcjonalnej, z uwzględnieniem faktycznych opóźnień czasowych wprowadzanych przez elementy logiczne oraz połączenia;
- *analiza czasowa (Timing Analysis)* – w upakowanym schemacie analizowane są opóźnienia propagacji sygnałów w celu zapewnienia *ograniczeń czasowych (timing constraints)* projektu i określenia przewidywanej wydajności systemu;

- *programowanie i konfiguracja (Programming and Configuration)* – fizyczna implementacja projektu w docelowym układzie FPGA przez skonfigurowanie elementów logicznych i odpowiednich połączeń w strukturze układu FPGA.

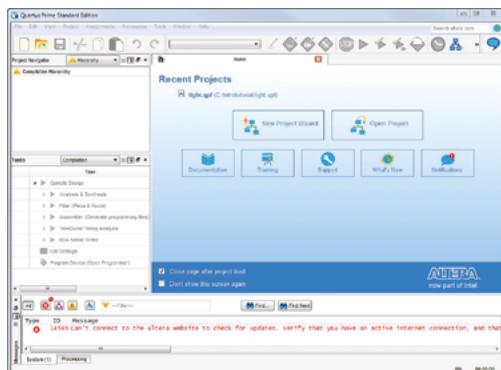


RYS. 3.1. Proces projektowania w systemie Quartus

Opis ogólny procesu projektowania nie obejmuje wszystkich istotnych szczegółów praktycznej realizacji projektu. W dalszej części rozdziału przedstawione zostaną praktyczne wskazówki projektowe. Poszczególne kroki realizacji projektu przy wykorzystaniu systemu Quartus zostaną zilustrowane na przykładzie projektowania pełnego jednobitowego sumatora, który będzie implementowany na płycie prototypowej DE1-SoC.

3.2. Utworzenie nowego projektu w systemie Quartus

Po uruchomieniu programu Quartus Prime na ekranie pojawi się okno zaprezentowane na rysunku 3.2. Użytkownik będzie miał możliwość utworzenia nowego projektu za pomocą *kreatora tworzenia nowych projektów (New Project Wizard)*, otwarcia istniejącego projektu (*Open Project*), a także przekierowania się do strony internetowej FPGA Intel w celu uzyskania dostępu do dokumentacji, zapoznania się z materiałami szkoleniowymi oraz informacjami technicznymi.



RYS. 3.2. Strona startowa systemu Quartus

Zazwyczaj praca nad nowym projektem rozpoczyna się od utworzenia na dysku katalogu projektu, w którym znajdować się będą wszystkie związane z nim pliki. Oprócz plików tworzonych przez użytkownika w tym katalogu umieszczone zostaną także pliki generowane przez zasoby programowe systemu Quartus. Ich liczba może być dość duża. Zbiór wszystkich plików związanych z danym projektem nazywany jest *bazą danych projektu*.

Mimo że system Quartus pozwala na umieszczenie kilku projektów w jednym katalogu, to jednak dobrym stylem projektowania jest przydzielenie każdemu projektowi osobnego katalogu na dysku.

Jako przykład zaprojektujemy prosty układ jednobitowego sumatora. Kod projektu w języku Verilog przedstawiony został poniżej:

```
module sum1 (input cin, a, b,           // porty wejściowe
              output s, cout);         // porty wyjściowe

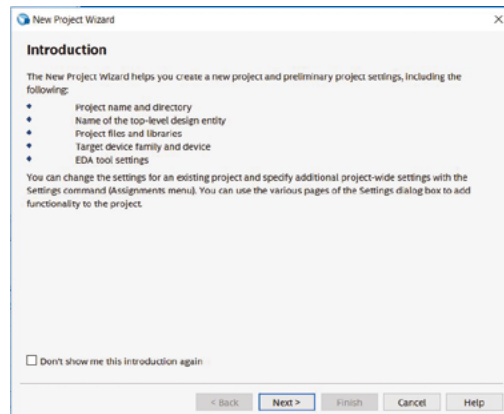
    wire g1_o, g2_o, g3_o;             // zmienne wewnętrzne

    and g1 (g1_o, a, b);                // instancja pierwszej bramki AND
    xor g2 (g2_o, a, b);                // instancja pierwszej bramki XOR
    and g3 (g3_o, g2_o, cin);           // instancja drugiej bramki AND
    or g4 (cout, g1_o, g3_o);           // instancja bramki OR
    xor g5 (s, g2_o, cin);              // instancja drugiej bramki XOR
endmodule
```

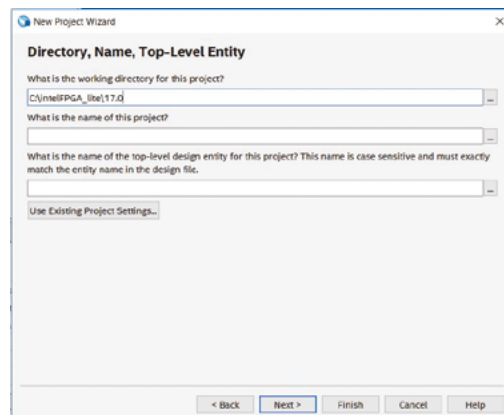
Rozpoczęcie pracy nad nowym projektem zazwyczaj polega na uruchomieniu kreatora tworzenia projektu. Jeśli wcześniej nie utworzono katalogu projektu, system Quartus stworzy go automatycznie. Poniższy scenariusz opisuje właśnie taki przypadek.

Algorytm 3.1. Scenariusz utworzenia nowego projektu

1. Wywołanie kreatora tworzenia projektu. Należy wybrać opcję **File > New Project Wizard** i kliknąć przycisk **Next**, w wyniku pojawi się okno, które zostało przedstawione na rysunku 3.3, zawierające listę kroków wykonywanych podczas tworzenia nowego projektu. Naciśnięcie przycisku **Next** przeniesie nas do kolejnego okna, widocznego na rysunku 3.4, gdzie zostaniemy poproszeni o podanie nazwy katalogu projektu (*What is the working directory for this project?*) oraz nazwy projektu (*What is the name of this project?*).



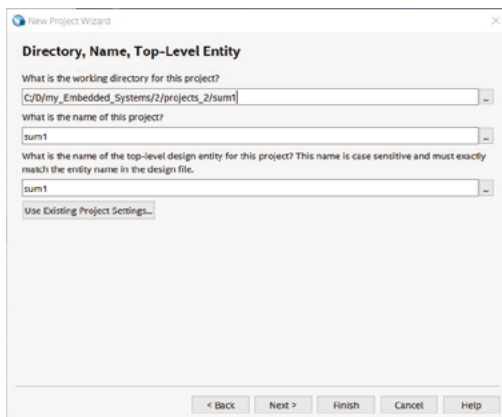
RYS. 3.3. Tworzenie nowego projektu za pomocą kreatora New Project Wizard



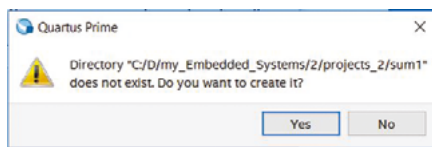
RYS. 3.4. Zapytanie o nazwę katalogu, projektu oraz modułu najwyższego poziomu

2. Podanie katalogu i nazwy projektu. W polu tekstowym należy wpisać nazwę katalogu projektu. Przykładowo możemy wpisać ścieżkę `D:/.../sum1`. Jeśli taki katalog nie istnieje, zostanie on utworzony automatycznie przez system Quartus. Następnie musimy określić nazwę projektu. Powinna być ona taka sama, jak nazwa modułu najwyższego poziomu, na przykład *sum1* (rysunek 3.5). Jeśli katalog

nie został wcześniej utworzony, pojawi się okno dialogowe, podobne do tego na rysunku 3.6, gdzie należy odpowiedzieć **Yes**.

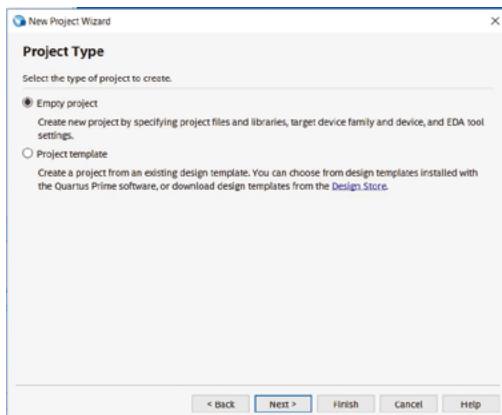


RYS. 3.5. Podanie nazwy katalogu, projektu oraz modułu najwyższego poziomu dla projektu *sum1*

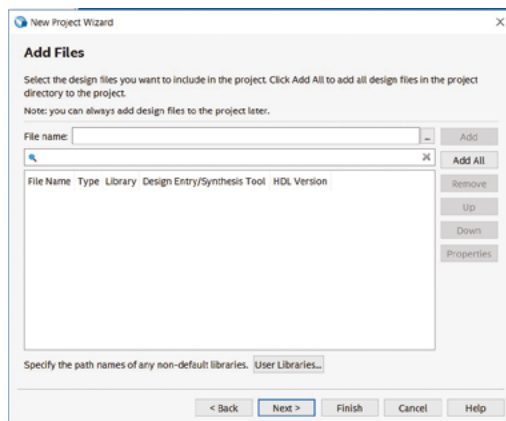


RYS. 3.6. Zgoda na utworzenie katalogu projektu

3. Wybór typu projektu. W kolejnym oknie kreatora (rysunek 3.7) należy wybrać typ projektu. Można go utworzyć od podstaw (*Empty*) lub wybrać z dostępnych szablonów projektów (*Project template*). Aby stworzyć projekt od podstaw, należy wybrać opcję **Empty** i kliknąć przycisk **Next**. Następnie pojawi się kolejne okno, podobne do tego na rysunku 3.8.

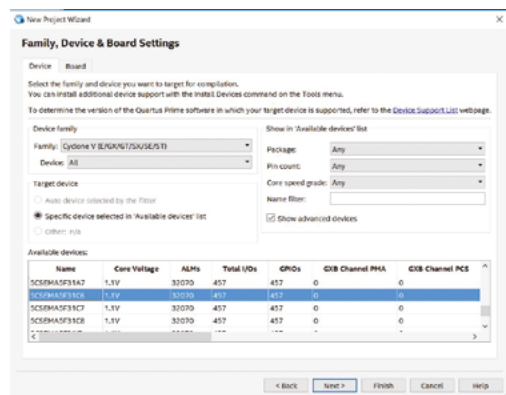


RYS. 3.7. Wybór typu projektu



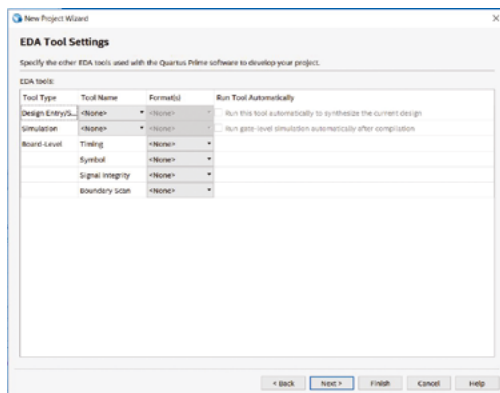
RYS. 3.8. Dodawanie plików projektu

4. Dodawanie plików projektu. W trakcie tworzenia projektu istnieje opcja dodania wstępnie przygotowanych plików. Można je również dodać później, w trakcie pracy nad projektem, korzystając z opcji **Project > Add/Remove Files in Project**. Jeśli nie mamy gotowych plików projektu, wystarczy kliknąć przycisk **Next** i przejść do kolejnego okna, jak to pokazano na rysunku 3.9.



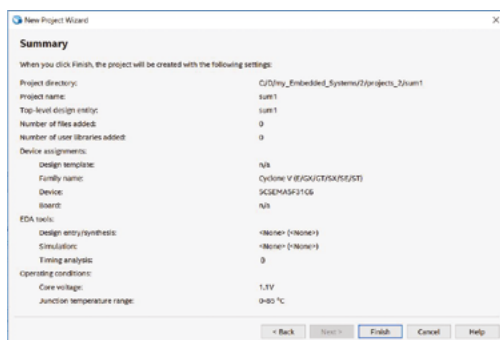
RYS. 3.9. Wybór rodziny FPGA oraz konkretnego układu

5. Wybór rodziny FPGA oraz docelowego układu. Z listy rozwijanej należy wybrać rodzinę FPGA (*Family*) oraz zaznaczyć układ docelowy (*Target Devices*), w którego strukturze będzie realizowany projekt. Jeśli konkretny układ FPGA nie zostanie wskazany, system Quartus dokona wyboru automatycznie. W przypadku projektu na płycie prototypowej DE1-SoC trzeba wybrać rodzinę **Family**: Cyclone V [2], a z listy dostępnych układów (*Available devices*) wybrać Cyclone V SoC 5CSEMA5F31C6. Po kliknięciu przycisku **Next** otworzy się kolejne okno, widoczne na rysunku 3.10.



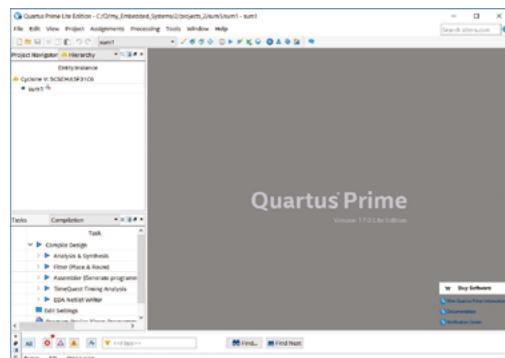
RYS. 3.10. Określenie narzędzi projektowania innych producentów

6. Określenie narzędzi zautomatyzowanego projektowania innych producentów (*EDA Tool Settings*). W systemie Quartus terminem *EDA Tools* określa się narzędzia projektowania opracowane przez inne firmy (nie Intel), które mogą być wykorzystane w procesie projektowania. Czasami korzystanie z tych narzędzi może prowadzić do uzyskania bardziej efektywnych wyników w wybranych etapach projektowania. Jeśli projektowanie odbywa się wyłącznie przy użyciu narzędzi dostarczonych przez Quartus, wystarczy kliknąć przycisk **Next**, co spowoduje pojawienie się okna, jak przedstawiono na rysunku 3.11.



RYS. 3.11. Podsumowanie utworzonego projektu

7. Podsumowanie ustawień projektu. Okno podsumowania zawiera podstawowe ustawienia projektu. Jeśli to konieczne, istnieje możliwość powrotu do poprzednich kroków i zmiany parametrów projektu. Jeśli wszystkie ustawienia są poprawne, należy kliknąć przycisk **Finish**. W rezultacie system Quartus utworzy nowy projekt z zadanymi parametrami i przejdzie do kolejnego okna, jak pokazano na rysunku 3.12.



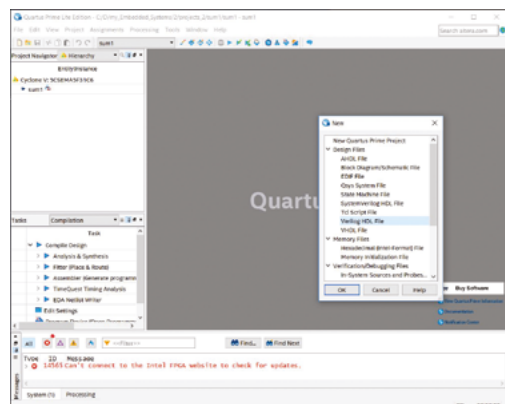
RYS. 3.12. Widok okna systemu Quartus z utworzonym projektem *sum1*

3.3. Wprowadzenie kodu projektu w języku Verilog

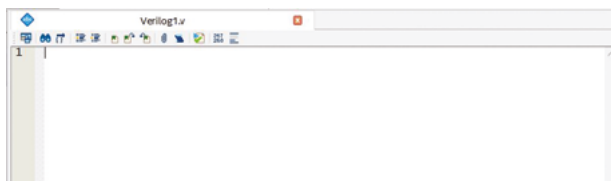
Kod projektu w języku Verilog jest zwykłym plikiem tekstowym w formacie ASCII z rozszerzeniem *.v*. Do jego przygotowania można użyć dowolnego edytora tekstu umożliwiającego tworzenie plików ASCII. Kod projektu w języku Verilog można również wprowadzić bezpośrednio w systemie Quartus, zgodnie z poniższym scenariuszem.

Algorytm 3.2. Scenariusz wprowadzenia kodu projektu w języku Verilog

1. Tworzenie nowego pliku z kodem w języku Verilog. Po wybraniu opcji **File > New**, na ekranie pojawi się okno, jak pokazano na rysunku 3.13. Należy w nim wybrać typ pliku – **Verilog HDL File** i kliknąć **OK**. Wtedy system przeniesie nas do okna edytora tekstu przedstawionego na rysunku 3.14.

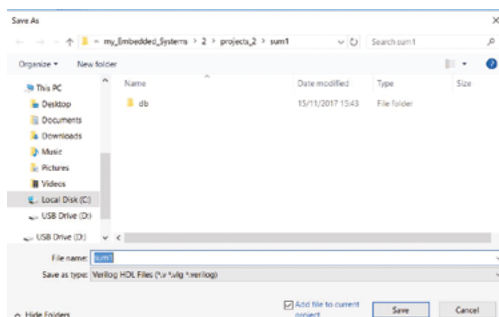


RYS. 3.13. Tworzenie nowego pliku z kodem projektu w języku Verilog



RYS. 3.14. Okno edytora tekstu

2. Zapis pliku do katalogu projektu. W tym celu należy wybrać opcję **File > Save As**. W odpowiedzi pojawi się okno, które zostało pokazane na rysunku 3.15. W polu tekstowym **File name** należy wpisać nazwę pliku (w naszym przykładzie to *sum1*), a z listy rozwijanej **Save as type** trzeba wybrać typ pliku **Verilog HDL Files**. Należy także zaznaczyć opcję **Add file to current project** (Dodać plik do bieżącego projektu). Po kliknięciu przycisku **Save** plik zostanie zapisany, a system ponownie przeniesie nas do okna edytora tekstu (rysunek 3.14).



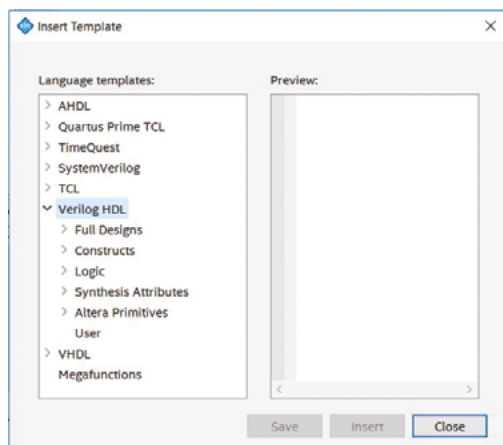
RYS. 3.15. Zapis pliku do katalogu projektu

3. Wprowadzenie kodu projektu. W edytorze tekstowym wpisujemy kod projektu z klawiatury. Można korzystać z szablonów języka Verilog, wybierając opcję **Edit > Insert Template > Verilog HDL**.
4. Zapis pliku. Plik można zapisać za pomocą opcji **File > Save** lub skrótu klawiszowego **Ctrl-s**.

3.4. Szablony języka Verilog

Edytor tekstowy systemu Quartus oferuje funkcję podpowiadania składni dla wybranych elementów i konstrukcji języka Verilog. W tym celu system udostępnia zestaw szablonów języka Verilog.

Aby skorzystać z odpowiedniego szablonu, wystarczy ustawić kursor w edytorze tekstowym w miejscu, w którym chcemy wstawić szablon, a następnie wybrać opcję **Edit > Insert Template > Verilog HDL**. Po wykonaniu tych kroków pojawi się okno, jak na rysunku 3.16.



RYS. 3.16. Szablony języka Verilog

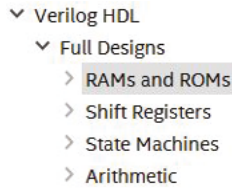
System Quartus udostępnia następujące szablony:

- gotowe projekty (*Full Designs*);
- konstrukcje języka Verilog (*Constructs*);
- elementy logiczne (*Logic*);
- atrybuty syntezy (*Synthesis Attributes*);
- prymitywy firmy Altera (*Altera Primitives*).

3.4.1. Gotowe projekty

System udostępnia następujące gotowe projekty (rysunek 3.17):

- pamięci typu RAM i ROM (jednoportowe, dwuportowe z rozdzielonymi portami zapisu i odczytu, dwuportowe z oddzielnymi portami dwukierunkowymi);
- rejestry przesuwne (zwykle, z asynchronicznym zerowaniem, wielopozycyjne, specjalne);
- automaty skończone (typu Moore'a i Mealy'ego z 4 stanami, niezawodne automaty skończone, a także automaty skończone z kodami stanów definiowanymi przez użytkownika);
- standardowe bloki arytmetyczne (sumatory, liczniki, układy mnożenia, układy mnożenia z akumulatorem, mnożenia z dodawaniem, a także bloki cyfrowego przetwarzania sygnałów).

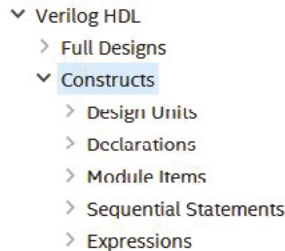


RYS. 3.17. Szablony gotowych projektów w języku Verilog

3.4.2. Konstrukcje języka Verilog

Szablony konstrukcji języka Verilog (*Constructs*) opisują następujące konstrukcje (rysunek 3.18):

- deklaracja modułów (*Design Unit*) w stylu 1 (nowy styl) oraz w stylu 2 (stary styl);
- deklaracje (*Declaration*) sieci, zmiennych, funkcji, zadań, zmiennych kodu generacji;
- elementy modułów (*Module Items*): instrukcja ciągłego przypisania **assign**, instrukcja **always** dla układów kombinacyjnych i sekwencyjnych, instrukcja instancji modułów, a także elementy i instrukcje generacji kodu;
- instrukcje proceduralne (*Sequential Statements*): zadań systemowych, blokującego i nieblokującego przypisania, instrukcje **if**, **case**, **casex**, **casez**, instrukcje pętli **for**, **while**, bloku proceduralnego;
- wyrażenia (*Expressions*): instrukcje unarne i binarne, instrukcja warunkowa.



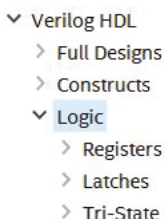
RYS. 3.18. Szablony konstrukcji języka Verilog

3.4.3. Elementy logiczne

Szablony elementów logicznych (*Logic*) zawierają następujące komponenty (rysunek 3.19):

- rejestry (*Register*) z różnymi wariantami sterowania: przełączane narastającym lub opadającym zboczem sygnału synchronizacji, z ustawieniem na 1 po włączeniu zasilania, z asynchronicznym zerowaniem, z wejściem blokowania synchronizacji;
- zatraski (*Latch*; przezroczyste przerzutniki typu D): podstawowe oraz z wejściem zerowania;

- trójstanowe elementy (*Tri-State*): szyna trójstanowa, rejestr z wyjściami trójstanowymi, dwukierunkowe wejście-wyjście, bufor z otwartym drenem (*open-drain buffer*).



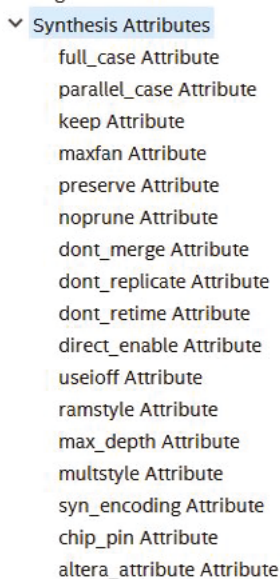
RYS. 3.19. Szablony elementów logicznych

3.4.4. Atrybuty syntezy

Atrybuty syntezy (*Synthesis Attributes*) zawierają następujące komponenty (rysunek 3.20):

- **full_case** – syntezyator ustawi wyjścia schematu jako nieokreślone (*don't care*) dla tych danych wejściowych, które nie zostały podane w operatorze **case**; pozwala to zapobiec automatycznemu umieszczaniu zatrząsków na wyjściach układów kombinacyjnych;
- **parallel_case** – syntezyator wykona równoległą realizację instrukcji dla wszystkich elementów stałych instrukcji **case** zamiast realizacji z wykorzystaniem kodera priorytetowego; pozwala to zwiększyć szybkość działania układu oraz zmniejszyć koszty realizacji instrukcji **case**, zwłaszcza gdy wiadomo, że wartości elementów stałych się nie powtarzają;
- **keep** – chroni zmienną przed usunięciem w procesie optymalizacji logicznej;
- **maxfan** – zapobiega przekroczeniu maksymalnej wartości obciążenia wyjść rejestru lub układu kombinacyjnego;
- **preserve** – chroni rejestr przed usunięciem w procesie optymalizacji;
- **noprune** – zapobiega usunięciu w procesie optymalizacji rejestru, który nie jest bezpośrednio połączony z wyjściem modułu najwyższego poziomu;
- **don't_merge** – zapobiega połączeniu rejestru z innymi rejestrami;
- **don't_replicate** – zapobiega duplikowaniu rejestrów;
- **don't_retime** – zapobiega wykonaniu *retimingu* układu; *retiming* jest jedną z metod optymalizacji, polegającą na przemieszczeniu przerzutników w schemacie w celu zwiększenia wydajności, zmniejszenia zajmowanego obszaru lub obniżenia poboru mocy przy zachowaniu niezmiennego funkcjonowania układu;
- **direct_enable** – pozwala skierować ścieżkę krytyczną bezpośrednio do portu zezwolenia sygnału synchronizacji (*clock enable*) rejestru, a pozostałe warunki synchronizacji realizować za pomocą regularnej logiki;

- **useioff** – pozwala na wykorzystanie przerzutników bloków wejścia-wyjścia do realizacji logiki projektu; umożliwia to zmniejszenie liczby elementów logicznych FPGA użytych do realizacji projektu oraz zwiększenie wydajności układu;
- **ramstyle** – określa sposób realizacji pamięci (RAM, ROM) lub rejestru przesuwanego;
- **max_depth** – określa maksymalną głębokość pamięci, pozwala zoptymalizować wykorzystanie bloku pamięci;
- **multstyle** – określa, gdzie ma być zrealizowany układ mnożenia: w bloku DSP czy w bloku logiki ogólnej (zrealizowany za pomocą elementów logicznych);
- **syn_encoding** – wybiera sposób kodowania stanów wewnętrznych automatu skończonego: kody stanów są definiowane przez użytkownika, stosowane jest kodowanie bezpieczne lub wykorzystywane są oba sposoby;
- **chip_pin** – przyporządkowuje wyprowadzenia portom modułu;
- **altera_attribute** – umożliwia wykonanie dowolnych przypisań obiektom w kodzie Verilog; przypisania realizowane są w formacie QSF.



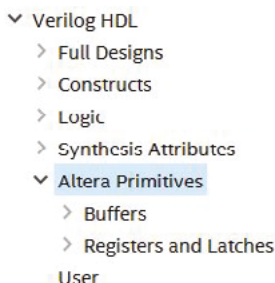
RYS. 3.20. Szablony atrybutów syntezy

3.4.5. Prymitywy firmy Altera

Do prymitywów opracowanych przez firmę Altera (*Altera Primitives*) należą następujące elementy (rysunek 3.21):

- bufor: wejściowe, wyjściowe, dwukierunkowe, CASCADE, CARRY_SUM, GLOBAL, LCELL, OPNDRN, TRI;

- przerzutniki i zatrzaski: DFF, DFFE, DFFEAS, DFFEAS, JKFF, JKFFE, LATCH, SRFF, SRFFE, TFF, TFFE.



RYS. 3.21. Szablony prymitywów

Oprócz przedstawionych standardowych elementów użytkownik może tworzyć własne szablony projektów w języku Verilog.

3.5. Dodawanie plików do projektu

Nawet proste projekty, składające się z jednego pliku z kodem źródłowym w języku Verilog, często wymagają dołączenia do projektu dodatkowych plików, takich jak plik ograniczeń czasowych lub plik przyporządkowania wyprowadzeń. W przypadku bardziej złożonych projektów, które składają się z wielu plików, procedura włączenia tych plików do projektu jest niezbędna.

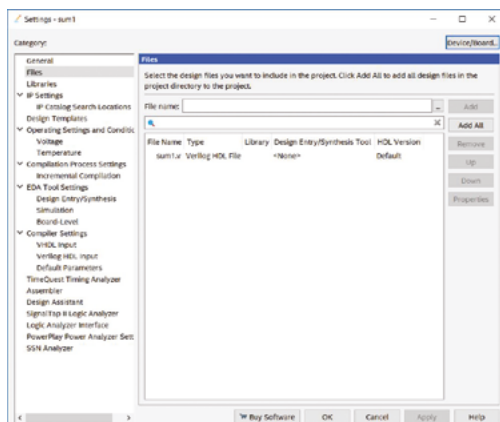
W naszym przykładzie plik **sum1.v** został dodany do projektu automatycznie, w wyniku zaznaczenia opcji **Add file to current project** przy zapisywaniu pliku na dysku.

W celu dołączenia dowolnego pliku do bieżącego projektu należy zrealizować następujący scenariusz.

Algorytm 3.3. Dodawanie pliku do projektu

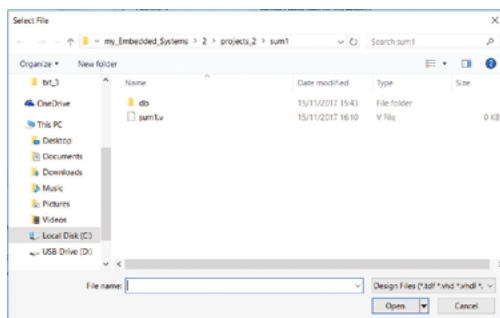
1. Otwieranie okna dodawania plików do projektu. Przejść do tego okna można na dwa sposoby:
 - przez opcję ustawień **Assignments > Settings** i wybranie elementu **Files**;
 - wybierając komendę **Project > Add/Remove Files in Project**.

W wyniku otworzy się okno przedstawione na rysunku 3.22.



RYS. 3.22. Okno dodawania plików do projektu

2. Wyszukiwanie pliku na dysku. Po wciśnięciu przycisku [...] po prawej stronie od pola **File name** na ekranie pojawi się okno umożliwiające przeglądanie katalogów (rysunek 3.23), w którym należy odnaleźć i zaznaczyć dodawany plik. Po kliknięciu przycisku **Open** nazwa wybranego pliku pojawi się w polu **File name** (rysunek 3.22).



RYS. 3.23. Okno wyboru pliku

3. Dodanie pliku do projektu. Aby dodać plik do projektu, należy kliknąć przycisk **Add** w oknie na rysunku 3.22. Jeśli przycisk **Add** jest niedostępny, może to oznaczać, że plik nie został wybrany lub nie udało się go odnaleźć.
4. Punkty 2–3 należy powtórzyć dla wszystkich plików, które trzeba dołączyć do projektu.

Warto zauważyć, że dodawanie plików do projektu można przeprowadzić w dowolnym momencie procesu projektowania, na przykład w odpowiedzi na komunikat o błędzie wynikający z braku wymaganego pliku.

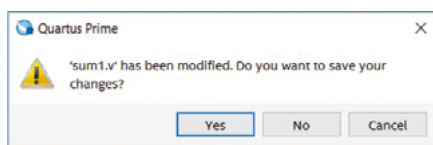
3.6. Kompilacja projektu

Projekt wprowadzony do systemu Quartus w postaci kodu w języku Verilog (lub VHDL), tak samo jak projekt przygotowany za pomocą edytora graficznego schematów, musi być najpierw skompilowany. W tym celu jest on przetwarzany przy użyciu różnych narzędzi systemu Quartus, które wykonują analizę kodu, syntezę układu, generowanie plików konfiguracji wykorzystywanych do implementacji projektu w strukturze docelowego układu FPGA. Procesami tymi zarządza aplikacja zwana *kompilatorem* (*Compiler*).

Kompilator można uruchomić na kilka sposobów:

- wybierając z menu polecenie **Processing > Start Compilation**;
- klikając na ikonę  w panelu narzędzi systemu;
- używając skrótu klawiszowego **Ctrl+L**.

Jeśli plik projektu został zmodyfikowany, a zmiany nie zostały zapisane, przed rozpoczęciem kompilacji pojawi się komunikat, jak na rysunku 3.24. Należy wtedy wyrazić zgodę na zapis pliku, klikając przycisk **Yes**.

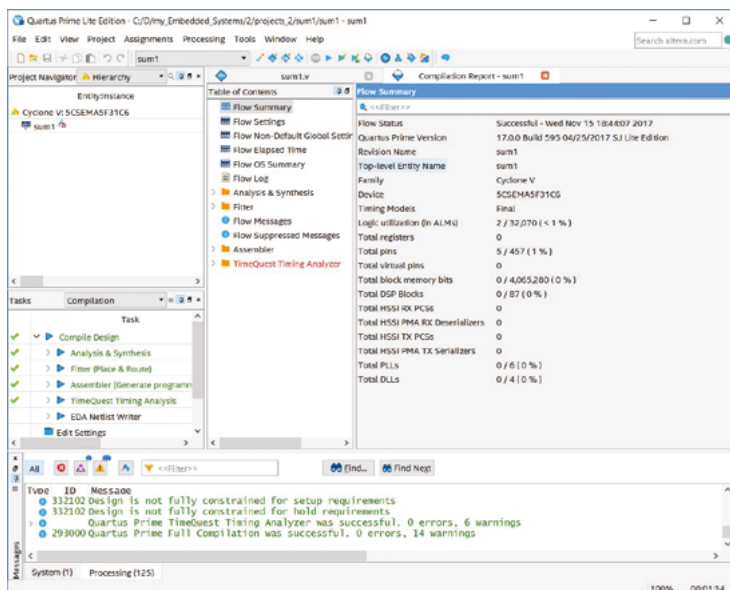


RYS. 3.24. Zgoda na zapis pliku przed kompilacją

W trakcie kompilacji w oknie **Tasks** (rysunek 3.25) pomyślnie zakończone etapy kompilacji będą oznaczane na zielono. Jednocześnie w oknie **Message** kompilator będzie wyświetlał wszystkie komunikaty generowane przez narzędzia uczestniczące w procesie kompilacji (rysunek 3.25).

Jeśli kod projektu nie zawiera błędów i kompilacja zakończyła się pomyślnie, na ekranie pojawi się raport o wynikach kompilacji, jak pokazano na rysunku 3.25. Zakładkę z oknem tego raportu (*Compilation Report*) można zamknąć zwykłym sposobem, czyli przez kliknięcie przycisku [x]. W każdym momencie można ją przywrócić, wybierając komendę **Processing > Compilation Report**.

Raport z wykonania kompilacji projektu zawiera szczegółowe informacje o wynikach jej poszczególnych etapów, które są dostępne po kliknięciu odpowiedniej pozycji w oknie **Table of Contents**. Po zakończeniu kompilacji automatycznie otwiera się okno z podsumowaniem (*Flow Summary*) widoczne na rysunku 3.25, gdzie przedstawione są podstawowe parametry projektu oraz wykorzystane zasoby układu FPGA, takie jak liczba użytych w projekcie elementów logicznych, przerzutników, wyprowadzeń i innych bloków funkcjonalnych. Na przykład do realizacji naszego przykładu wykorzystano 2 elementy logiczne typu ALM oraz 5 wyprowadzeń układu FPGA.



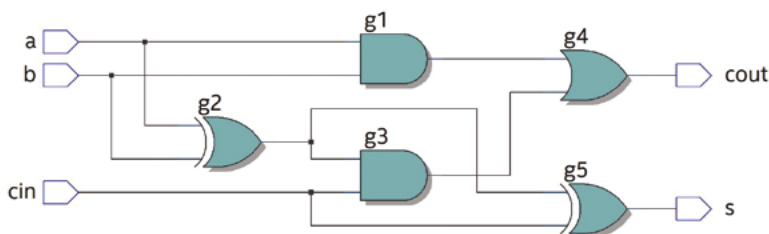
RYS. 3.25. Okno systemu Quartus po zakończeniu kompilacji projektu

3.7. Przegląd i analiza wyników syntezy

Po wykonaniu kompilacji projektu można przeglądać wyniki syntezy układu w postaci:

- schematu na *poziomie przesłach międzyrejestrowych* (Register Transition Level – RTL);
- mapy technologicznej projektu po odwzorowaniu na układ docelowy FPGA (*Technology Map Post-Mapping*);
- mapy technologicznej po upakowaniu projektu (*Technology Map Fitting*);
- automatu skończonego (za pomocą *State Machine Viewer*).

Schemat projektu na poziomie przesłach międzyrejestrowych można obejrzeć, wybierając w menu komendę **Tools > Netlist Viewers > RTL Viewer**, w wyniku czego zostanie otwarte okno podglądu schematu **RTL Viewer**. Schemat RTL naszego projektu jednobitowego sumatora przedstawiony został na rysunku 3.26.



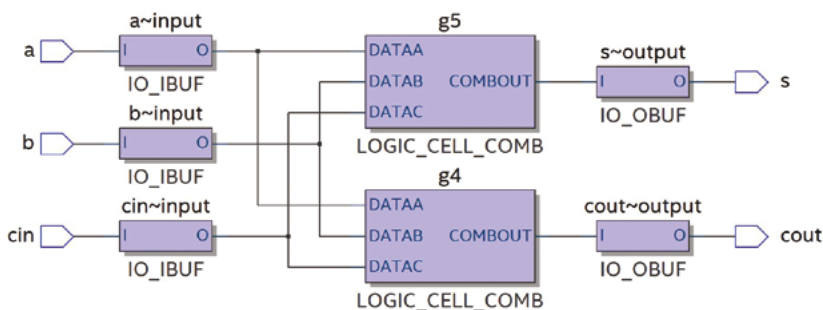
RYS. 3.26. Schemat projektu *sum1* na poziomie przesłach międzyrejestrowych

Schemat projektu na poziomie RTL zawiera komponenty, takie jak bramki, rejestry, przerzutniki i zatrzaski, które nie istnieją w rzeczywistych strukturach FPGA. Niemniej jednak analiza schematu na poziomie przesłań międzyrejestrowych może być bardzo przydatna, umożliwiając sprawdzenie, czy otrzymany w wyniku syntezy schemat odpowiada oczekiwaniom użytkownika. W naszym przykładzie widzimy, że schemat na rysunku 3.26 dokładnie odzwierciedla wprowadzony kod projektu w języku Verilog.

Często analiza schematu na poziomie przesłań międzyrejestrowych pozwala wykryć błędy w kodzie, którego składnia jest poprawna, na przykład:

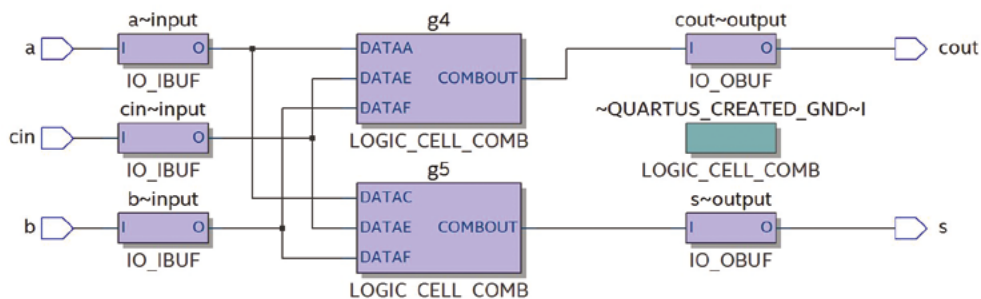
- brak połączenia między wejściem lub wyjściem a wewnętrzną logiką projektu;
- brak wymaganych elementów projektu, które mogą zostać usunięte w wyniku optymalizacji schematu;
- pojawienie się w schemacie zbędnych, niepożądanych elementów, na przykład ustawienie zatrzasków na wyjściach przy niepoprawnym zastosowaniu instrukcji **if** lub **case**.

Mapa technologiczna po odwzorowaniu projektu na strukturę docelowego układu FPGA dla naszego przykładu przedstawiona została na rysunku 3.27. Składa się ona z pięciu elementów wejścia-wyjścia oraz dwóch elementów logicznych układu FPGA, na których zrealizowane zostały układy kombinacyjne funkcji logicznych *s* i *cout*.



RYS. 3.27. Mapa technologiczna projektu *sum1* po odwzorowaniu na strukturę docelowego układu FPGA

Mapa technologiczna po upakowaniu projektu dla naszego układu (rysunek 3.28) praktycznie pokrywa się z poprzednim schematem.



RYS. 3.28. Mapa technologiczna projektu `sum1` po upakowaniu w docelowym układzie FPGA

Nasz projekt nie zawiera automatu skończonego, dlatego graf automatu dla niego nie jest generowany.

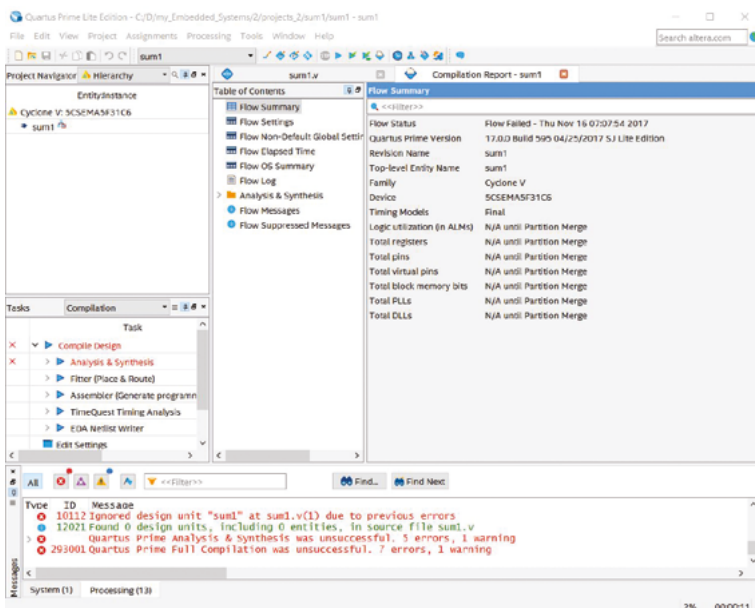
3.8. Komunikaty o błędach kompilacji

Jeśli po zakończeniu kompilacji w oknie **Message** pojawi się komunikat, taki jak na rysunku 3.25, czyli „*Full Compilation was successful. 0 errors*”, oznacza to, że została ona przeprowadzona pomyślnie. Jednakże czasami kompilacja projektu może zostać przerwana z powodu wykrycia błędów, głównie w kodzie projektu. Błędy te powinny być zawsze naprawione, ponieważ bez tego niemożliwe jest przejście do kolejnych etapów projektowania. Dlatego każdy użytkownik powinien umieć odnaleźć i naprawić zasygnalizowane błędy.

Informacje o błędach i ostrzeżeniach kompilatora są wyświetlane w oknie komunikatów **Message**. Aby zlokalizować miejsce błędu w kodzie projektu w języku Verilog, należy dwukrotnie kliknąć na wybrany komunikat o błędzie, co spowoduje podświetlenie odpowiedniej linii kodu w edytorze tekstowym. Dodatkowe informacje o błędzie można także uzyskać poprzez skorzystanie z klawisza funkcyjnego F1. Analizę błędów należy zacząć od pierwszego komunikatu, ponieważ kolejne zgłaszane komunikaty mogą wynikać z poprzednio wykrytych błędów.

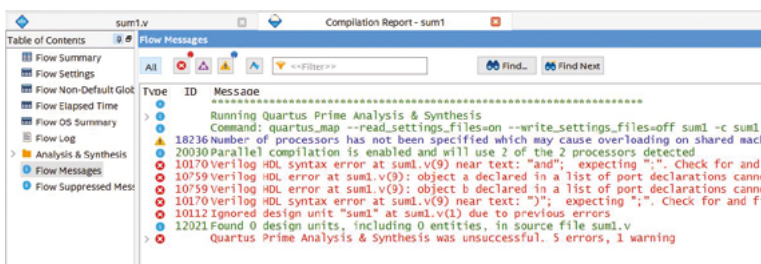
Oprócz zgłaszania błędów kompilator generuje wiele komunikatów informacyjnych, może również wyświetlać ostrzeżenia. Komunikaty informacyjne zawierają szczegóły dotyczące kolejnych etapów procesu kompilacji. Ostrzeżenia, które pojawiają się w oknie komunikatów (*Warning Messages* oraz *Critical Warning Messages*), nie przerywają procesu projektowania, jednak mogą zawierać istotne informacje, które zawsze należy przeanalizować, zwłaszcza komunikaty krytyczne (*Critical Warning Messages*).

Aby zasymulować błąd składni w kodzie projektu, otworzymy plik `sum1.v` oraz skasujemy średnik w pierwszym operatorze przypisania **assign**. Ponowna kompilacja spowoduje wygenerowanie komunikatu o błędzie, jak pokazano na rysunku 3.29.



RYS. 3.29. Komunikaty o błędach kompilacji

Na panelu **Tasks** widocznym na rysunku 3.29 możemy zauważyć, że kompilacja została przerwana na etapie **Analysis & Synthesis** (analizy i syntezy). Aby ułatwić analizę komunikatów o błędach, w panelu **Table of Contents** można otworzyć okno raportu **Flow Messages**, jak pokazano na rysunku 3.30.



RYS. 3.30. Okno komunikatów o błędach kompilacji

Raport kompilacji może zostać wyświetlony w oddzielnym oknie poprzez wybór opcji **Window > Detach Window** lub przywrócony z powrotem za pomocą **Window > Attach Window**. Podwójne kliknięcie na pierwszym komunikacie o błędzie przekieruje nas do pliku sum1.v z operatorem kodu podświetlonym w miejscu błędu.

3.9. Przyporządkowanie wyprowadzeń

Kolejnym etapem realizacji projektu jest przyporządkowanie wyprowadzeń układu docelowego FPGA do sygnałów wejściowych i wyjściowych projektu. Jeśli nie zostanie ono dokonane, kompilator może przypisywać dowolne wyprowadzenia układu FPGA do zewnętrznych sygnałów projektu. Jednak w niektórych zastosowaniach istotne jest określenie, do których fizycznych wyprowadzeń układu FPGA będą podłączone sygnały wejściowe i wyjściowe schematu. Ponieważ na płycie prototypowej DE1-SoC poszczególne komponenty (takie jak przełączniki lub diody LED) są związane z konkretnymi wyprowadzeniami układu FPGA, należy wskazać odpowiednie wyprowadzenia podczas realizacji projektu.

W naszym przykładzie wykorzystamy przełączniki SW0, SW1 i SW2 płytki prototypowej jako odpowiednio wejścia *a*, *b* i *cin* naszego układu, natomiast diody świecące LEDR0 i LEDR1 zostaną podłączone do wyjść *s* i *cout*. Przyporządkowanie wyprowadzeń układu FPGA dla tego przykładu przedstawiono w tabeli 3.1.

TABELA 3.1. Przyporządkowanie wyprowadzeń na płycie DE1-SoC

Komponent	Nazwa wyprowadzenia	Nazwa sygnału
SW0	PIN_AB12	<i>a</i>
SW1	PIN_AC12	<i>b</i>
SW2	PIN_AF9	<i>cin</i>
LEDR0	PIN_V16	<i>s</i>
LEDR1	PIN_W16	<i>cout</i>

W systemie Quartus istnieją dwa sposoby przyporządkowania wyprowadzeń FPGA zewnętrznym sygnałom projektu:

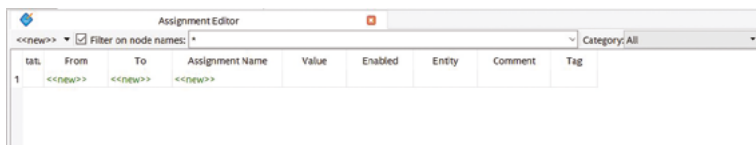
- za pomocą edytora przypisań (*Assignment Editor*);
- przez plik ustawień (*Quartus Setting File – QSF*).

3.9.1. Przyporządkowanie wyprowadzeń za pomocą edytora przypisań

Do przyporządkowania wyprowadzeń można wykorzystać następujący scenariusz.

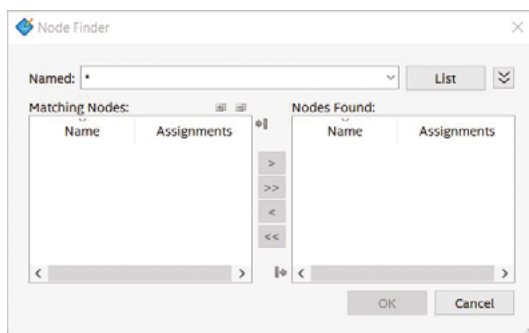
Algorytm 3.4. Przyporządkowanie wyprowadzeń FPGA zewnętrznym sygnałom projektu za pomocą edytora przypisań

1. Uruchomienie edytora przypisań. Moduł ten uruchomiany jest poleceniem **Assignments > Assignment Editor**, w wyniku czego pojawi się okno przedstawione na rysunku 3.31.

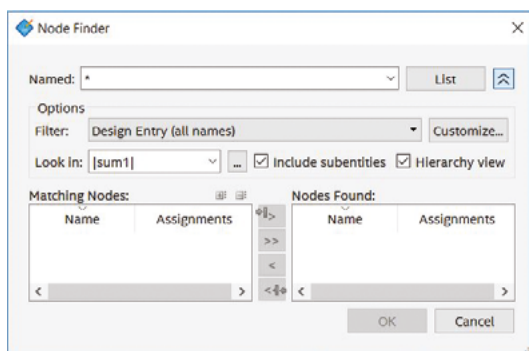


RYS. 3.31. Okno edytora przypisań

- Wyszukiwanie zewnętrznych sygnałów projektu. W polu **Category** z rozwijanej listy należy wybrać opcję **All**. Podwójne kliknięcie w pole <<new>> w kolumnie **To** spowoduje pojawienie się przycisku wyszukiwania węzła **Node Finder** . Po kliknięciu w ten przycisk na ekranie pojawi się okno zaprezentowane na rysunku 3.32. Przy użyciu przycisków  i  można rozwijać lub związać więcej parametrów wyszukiwania (rysunek 3.33).

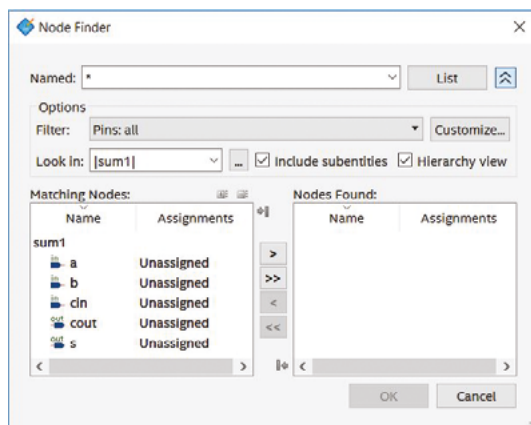


RYS. 3.32. Okno narzędzia wyszukiwania węzłów Node Finder



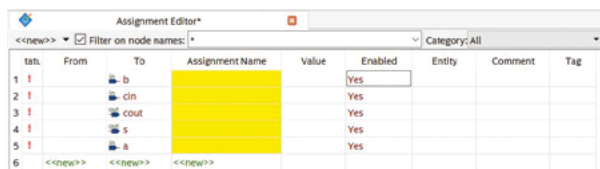
RYS. 3.33. Zwiększona liczba parametrów wyszukiwania

Z rozwijanej listy **Filter** należy wybrać opcję **Pins: all** (wszystkie wyprowadzenia), a następnie kliknąć przycisk **List**. W odpowiedzi w lewym oknie zostanie wyświetlona lista wszystkich zewnętrznych sygnałów projektu: *a*, *b*, *cin*, *s*, *cout* (rysunek 3.34).



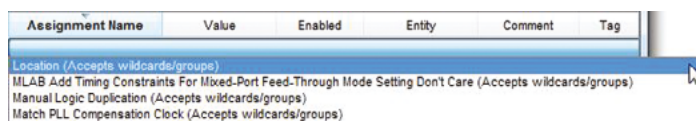
RYS. 3.34. Lista zewnętrznych sygnałów projektu w module Node Finder

- Wybór sygnałów. Wybrane sygnały należy przenieść do prawego okna modułu **Nodes Found** za pomocą przycisku [>]. Aby przenieść wszystkie sygnały naraz, można skorzystać z przycisku [>>]. Przyciski [<] i [<<] pozwalają z powrotem cofnąć wybór niepotrzebnych sygnałów. Po zatwierdzeniu utworzonej w prawym oknie listy przyciskiem **OK** wybrane sygnały przeniesione zostaną do kolumny **To** edytora przypisań (rysunek 3.35).



RYS. 3.35. Lista zewnętrznych sygnałów projektu w edytorze przypisań

- Przyporządkowanie wyprowadzeń sygnałom wejścia-wyjścia. Dla każdego sygnału na liście należy dwukrotnie kliknąć w pole kolumny **Assignment Name** i z rozwijanej listy (rysunek 3.36) wybrać pozycję **Location**. Następnie podwójnym kliknięciem w polu kolumny **Value** uruchomić okienko tekstowe do wpisywania nazwy wyprowadzenia układu FPGA. Te czynności należy powtórzyć dla każdego sygnału z listy. W rezultacie określimy przyporządkowanie wyprowadzeń, tak jak przedstawiono na rysunku 3.37.

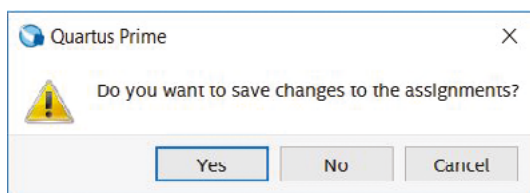


RYS. 3.36. Rozwijane menu Assignment Name

cat	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓	b	Location	PIN_A12	Yes			
2	✓	on	Location	PIN_A10	Yes			
3	✓	out	Location	PIN_W16	Yes			
4	✓	1	Location	PIN_V16	Yes			
5	✓	8	Location	PIN_A12	Yes			

RYS. 3.37. Wynik przyporządkowania wyprowadzeń

Po przyporządkowaniu wszystkich sygnałów należy ponownie skompilować projekt, aby ich nowe przypisanie zostały uwzględnione. Kompilator można uruchomić za pomocą ikony  na pasku narzędzi. W tym przypadku pojawi się komunikat, który przypomina o zapisaniu wprowadzonych zmian do pliku ustawień projektu (w naszym przykładzie jest to plik sum1.qsf), jak pokazano na rysunku 3.38.



RYS. 3.38. Przypomnienie o zapisie przyporządkowania wyprowadzeń do pliku

Po zakończeniu ponownej kompilacji można zauważyć, że liczba komunikatów ostrzeżeń generowanych przez kompilator się zmniejszyła. Aby sprawdzić poprawność przyporządkowania wyprowadzeń wejściom i wyjściom projektu, po zakończeniu kompilacji należy wybrać w polu **Table of Contents** opcję **Fitter > Resource Section > Input (Output) Pins**. W uzyskanym oknie (rysunek 3.39) możemy sprawdzić przypisanie sygnałów do wyprowadzeń układu FPGA.

Name	Pin #	IO Bank	X coordinate	Y coordinate	Z coordinate	Combinational Fan-Out
1 a	A12	3A	12	0	17	2
2 b	A12	3A	16	0	0	2
3 cin	AP9	3A	0	0	34	2

RYS. 3.39. Sprawdzanie przyporządkowania wyprowadzeń układu FPGA

3.9.2. Przyporządkowanie wyprowadzeń za pomocą pliku ustawień

Przyporządkowanie wyprowadzeń za pomocą edytora przypisań może być pracochłonne, zwłaszcza w przypadku bardzo dużych projektów. Alternatywnym podejściem do opisanego powyżej jest wykorzystanie pliku ustawień projektu QSF do przypisania wyprowadzeń.

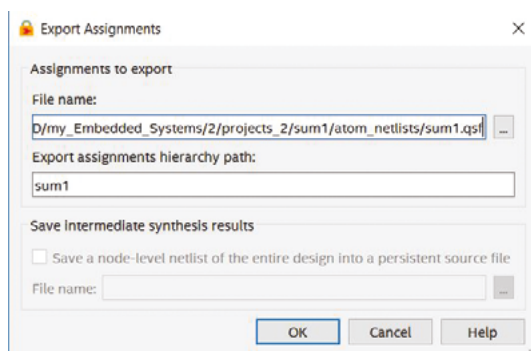
Składnia przyporządkowania wyprowadzeń w pliku QSF jest dość klarowna, co można zaobserwować na przykładzie opisu wyprowadzeń dla naszego projektu na rysunku 3.40.

```
set_location_assignment PIN_AC12 -to b
set_location_assignment PIN_AF9 -to cin
set_location_assignment PIN_W16 -to cout
set_location_assignment PIN_V16 -to s
set_location_assignment PIN_AB12 -to a
```

RYS. 3.40. Fragment pliku ustawień sum1.qsf określający przyporządkowanie wyprowadzeń

3.9.3. Import i eksport pliku ustawień

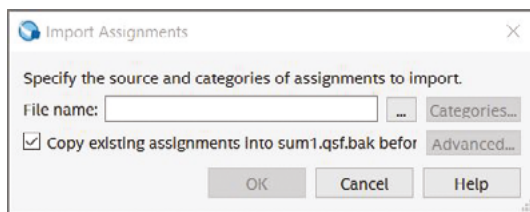
Plik ustawień, który został utworzony, może być eksportowany i importowany w celu wykorzystania w innych projektach. Eksport pliku ustawień można przeprowadzić z okna edytora przypisać (rysunek 3.37). Wybierając opcję **Assignments > Export Assignment**, uzyskamy okno pokazane na rysunku 3.41.



RYS. 3.41. Eksport pliku ustawień

W polu tekstowym **File name** należy wprowadzić nazwę katalogu, do którego plik ustawień zostanie wyeksportowany (domyślnie takim katalogiem jest *atom_netlist* w katalogu projektu).

Aby zaimportować plik ustawień z innego projektu, należy wybrać z menu polecenie **Assignments > Import Assignments**, co spowoduje otwarcie okna importu ustawień, jak pokazano na rysunku 3.42. W polu **File name** należy wprowadzić pełną ścieżkę i nazwę importowanego pliku wraz z rozszerzeniem **qsf**. Wymagany plik można także znaleźć na dysku za pomocą przycisku [...].



RYS. 3.42. Import pliku ustawień

3.9.4. Plik ustawień dla płytki prototypowej DE1-SoC

Wszystkie połączenia komponentów na płytce prototypowej DE1-SoC są stałe, dlatego wyprowadzenia FPGA mają niezmiennie przyporządkowania sygnałów, które zdefiniowane są w pliku ustawień DE1_SoC.qsf. Można go pobrać ze strony Intel FPGA University Program. W pliku tym stosowane są nazwy sygnałów określone w dokumentacji płytki DE1-SoC (*DE1-SoC User Manual*), dlatego też przy importowaniu pliku ustawień DE1_SoC.qsf do swojego projektu należy używać takich samych nazw sygnałów. Tak więc w naszym projekcie powinny to być nazwy SW[0], SW[1], SW[2], LEDR[0] i LEDR[1] zamiast sygnałów *a*, *b*, *cin*, *s* i *cout* odpowiednio.

Nazwy sygnałów i przyporządkowane im wyprowadzenia FPGA określone w pliku DE1_SoC.qsf dla przełączników, diod LED i przycisków umieszczonych na płytce prototypowej przedstawiono w tabelach 3.2, 3.3 i 3.4.

TABELA 3.2. Nazwy wyprowadzeń FPGA połączonych z przełącznikami SW0-SW9 na płycie DE1-SoC

Nazwa sygnału	Nazwa wyprowadzenia FPGA
SW[0]	PIN_AB12
SW[1]	PIN_AC12
SW[2]	PIN_AF9
SW[3]	PIN_AF10
SW[4]	PIN_AD11
SW[5]	PIN_AD12
SW[6]	PIN_AE11
SW[7]	PIN_AC9
SW[8]	PIN_AD10
SW[9]	PIN_AE12

TABELA 3.3. Nazwy wyprowadzeń FPGA połączonych diodami LED LEDR0-LEDR9 na płycie DE1-SoC

Nazwa sygnału	Nazwa wyprowadzenia FPGA
LEDR[0]	Pin_V16
LEDR[1]	Pin_W16
LEDR[2]	Pin_V17
LEDR[3]	Pin_V18
LEDR[4]	Pin_W17
LEDR[5]	Pin_W19
LEDR[6]	Pin_Y19
LEDR[7]	Pin_W20
LEDR[8]	Pin_W21
LEDR[9]	Pin_Y21

TABELA 3.4. Nazwy wyprowadzeń FPGA połączonych z przyciskami KEY0-KEY3 na płycie DE1-SoC

Nazwa sygnału	Nazwa wyprowadzenia FPGA
KEY[0]	PIN_AA14
KEY[1]	PIN_AA15
KEY[2]	PIN_W15
KEY[3]	PIN_Y16

Nazwy sygnałów SW, LEDR i KEY w tabelach 3.2–3.4 opisane są jako wektory, ponieważ w kodzie Verilog można się do nich odwoływać jak do wektorów. Na przykład przez SW[9.0] odwołujemy się do wszystkich przełączników SW na płycie DE1-SoC.

Zaimportowanie gotowego pliku **qsf** ustawień znacznie ułatwia pracę nad projektem. Jednakże należy pamiętać o odpowiednim nazewnictwie zewnętrznych sygnałów projektu.

3.9.5. Wykorzystanie nazw sygnałów predefiniowanych w pliku ustawień

Problem nazewnictwa zewnętrznych sygnałów można rozwiązać na kilka sposobów:

- 1) w kodzie projektu w języku Verilog można zastąpić nazwę każdego sygnału zewnętrznego odpowiednią nazwą zdefiniowaną w pliku QSF;
- 2) utworzyć nowy moduł najwyższego poziomu, w którym wszystkie porty mają nazwy odpowiadające nazwom w pliku QSF, a następnie połączyć porty tego modułu za pomocą instrukcji **assign** z oryginalnymi nazwami sygnałów;
- 3) stworzyć nowy moduł najwyższego poziomu, w którym umieszczona zostanie instancja oryginalnego modułu.

Kody projektów dla wszystkich trzech zaproponowanych rozwiązań, zapewniające zgodność nazw sygnałów w pliku ustawień DE1_SoC.qsf z nazwami zewnętrznymi sygnałów projektu, dla naszego przykładu przedstawione zostały w listingach 3.1, 3.2 i 3.3.

Listing 3.1. Zastąpienie nazw zewnętrznych sygnałów projektu nazwami z pliku QSF

```
module sum1_1      (input [0:2] SW,
                   output [0:1] LEDR);

    assign LEDR[0] = SW[0] ^ SW[1] ^ SW[2];
    assign LEDR[1] = SW[0] & SW[1] | (SW[0] ^ SW[1]) & SW[2];
endmodule
```

Listing 3.2. Przedstawienie projektu w postaci jednego modułu o nazwach portów zgodnych z definicjami w pliku QSF

```
module sum1_2 (SW, LEDR);
    input  [0:2]  SW;
    output[0:1]  LEDR;

    wire      a, b, cin, s, cout;

    // przypisanie nazw sygnałów zewnętrznych
    assign a = SW[0];
    assign b = SW[1];
    assign cin = SW[2];
    assign LEDR[0] = s;
    assign LEDR[1] = cout;

    // obliczenie wartości wyjść
    assign s = a ^ b ^ cin;
    assign cout = a & b | (a ^ b) & cin;
endmodule
```

Listing 3.3. Przedstawienie projektu w postaci dwóch modułów, gdzie moduł wyższego poziomu tworzy instancję modułu niższego poziomu

```
module sum1_3(                                // moduł najwyższego poziomu
    input  [0:2] SW,
    output[0:1] LEDR);
```

```

// tworzenie instancji sumatora
sum1_4 ex1(SW[0],SW[1],SW[2],LEDR[0],LEDR[1]);

endmodule

module sum1_4(                                // moduł sumatora
    input a,b,cin,
    output s,cout);

    assign s = a ^ b ^ cin;
    assign cout = a & b | (a ^ b) & cin;
endmodule

```

Każde z przedstawionych rozwiązań ma swoje wady i zalety. Pierwszy sposób pozwala na opisanie projektu w postaci jednego modułu, jednak zastąpienie oryginalnych nazw standardowymi może prowadzić do utraty semantycznego znaczenia nazw sygnałów. Ponadto dla dużych projektów zamiana nazw sygnałów w całym kodzie może być bardzo pracochłonna. Przy drugim rozwiązaniu semantyczne znaczenie nazw sygnałów zostaje zachowane, jednak kod projektu zostaje rozszerzony o dodatkowe definicje **wire** oraz instrukcje **assign**. Trzecie rozwiązanie zachowuje oryginalny moduł sumatora bez zmian, lecz wprowadzany jest dodatkowy moduł najwyższego poziomu, który zapewnia zgodność nazw sygnałów projektu z definicjami w pliku QSF.

Wybór sposobu rozwiązania problemu nazewnictwa sygnałów zależy od specyfiki projektu oraz wyboru projektantów. W naszej książce będziemy preferować trzecie rozwiązanie, które polega na zmianie nazw sygnałów poprzez tworzenie instancji oryginalnego modułu.

3.10. Symulacja projektu

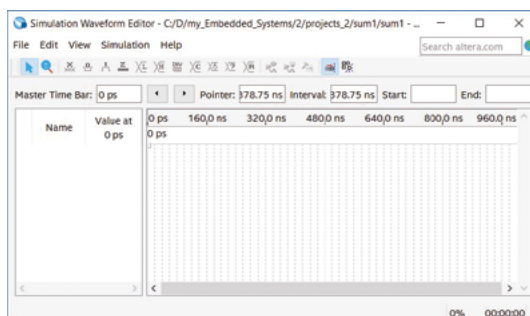
Przed implementacją zaprojektowanego układu w strukturze FPGA konieczne jest przeprowadzenie symulacji projektu w celu zweryfikowania poprawności jego działania. System Quartus oferuje dwa rodzaje symulacji: *funkcjonalną* (*functional simulation*) oraz *czasową* (*timing simulation*). Przed rozpoczęciem procesu symulacji konieczne jest wygenerowanie zestawu danych testowych, które będą podawane na wejściu układu. Taki zestaw danych nazywany jest zbiorem *wektorów testowych* (*test vectors*).

3.10.1. Generowanie wektorów testowych

Zbiór wektorów testowych w systemie Quartus generowany jest za pomocą edytora przebiegów czasowych (*Simulation Waveform Editor*). Proces tworzenia pliku zawierającego testowe przebiegi czasowe odbywa się według następującego scenariusza.

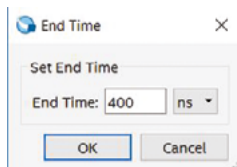
Algorytm 3.5. Generowanie wektorów testowych

1. Uruchomienie edytora przebiegów czasowych. Z menu systemu Quartus należy wybrać polecenie **File > New > Verification/Debugging Files > University Program VWF**, co spowoduje otwarcie okna edytora przebiegów czasowych, jak pokazano na rysunku 3.43. Utworzony plik zawierający przebiegi czasowe automatycznie zostanie zapisany w katalogu projektu pod nazwą *Waveform.vwf*.



RYS. 3.43. Okno edytora przebiegów czasowych

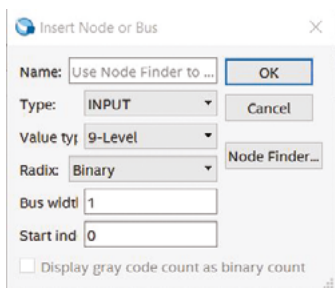
2. Określenie czasu trwania symulacji. Domyślnie przedział czasu symulacji wynosi 1000 ns. Aby zmienić końcowy czas symulacji, należy użyć komendy **Edit > Set End Time** i wpisać w wyświetlonym oknie nową wartość czasu symulacji, na przykład 400 ns (rysunek 3.44).



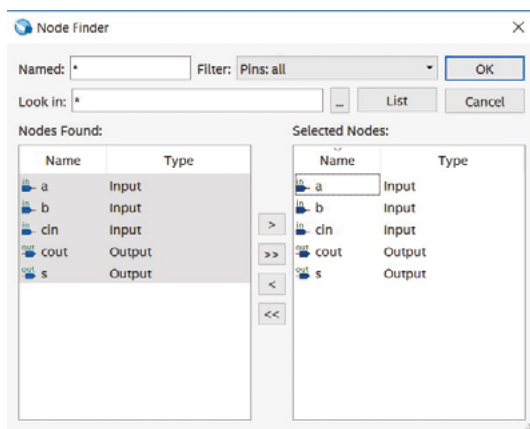
RYS. 3.44. Określenie czasu trwania symulacji

3. Wybór węzłów do obserwacji. Aby dodać węzły projektowanego układu do listy przebiegów, należy użyć komendy **Edit > Insert > Insert Node or Bus**, co spowoduje otwarcie okna przedstawionego na rysunku 3.45. W polu **Name** można wpisać nazwę dodawanego węzła lub skorzystać z przycisku **Node Finder...**, który umożliwia wyszukiwanie węzłów projektu. Po kliknięciu **Node Finder...** pojawi się okno wyszukiwarki, jak pokazano na rysunku 3.46. Należy ustawić wartość **Pins: all** w polu **Filter**, a następnie kliknąć przycisk **List**, aby wyświetlić listę wszystkich

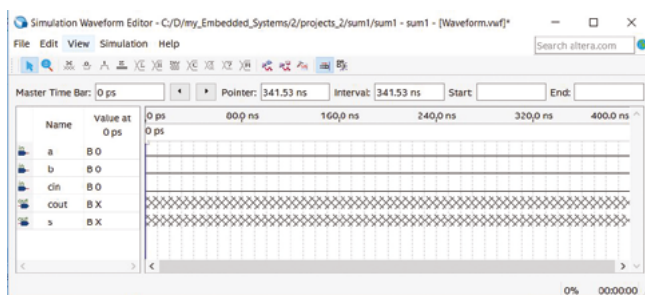
wyprowadzeń projektu w lewym oknie **Nodes Found** wyszukiwarki. Wybór wszystkich sygnałów do analizy dokonuje się przez kliknięcie przycisku [>>], co spowoduje przeniesienie ich do prawego okna **Selected Nodes** wyszukiwarki. Wybranie pojedynczego węzła lub grupy węzłów możliwe jest przez kliknięcie przycisku [>]. Przyciski [<<] i [<] służą do usunięcia sygnałów z listy wybranych do obserwacji. Po zatwierdzeniu wyboru przyciskiem **OK** wybrane węzły projektu zostaną dodane do okna edytora przebiegów czasowych (rysunek 3.47).



RYS. 3.45. Okno wyboru węzłów lub szyn



RYS. 3.46. Okno narzędzia **Node Finder** wyszukiwania węzłów



RYS. 3.47. Lista obserwowanych węzłów w edytorze przebiegów czasowych

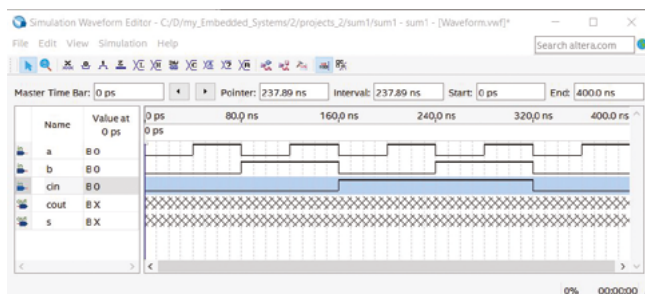
Kolejność wyświetlania sygnałów na wykresach można modyfikować, przesuwając wybrany sygnał w górę lub w dół. Edytor przebiegów czasowych umożliwia dodawanie do listy obserwowanych sygnałów nie tylko portów wejściowych i wyjściowych, lecz także dowolnych węzłów wewnętrznych projektu. Warto jednak zauważyć, że aby dany węzeł był dostępny do obserwacji, nie powinien zostać usunięty w procesie optymalizacji schematu. Aby temu zapobiec, należy zadeklarować węzeł z atrybutem *keep*.

4. Określenie wartości logicznych sygnałów (rysowanie przebiegów czasowych sygnałów wejściowych). Domyślnie na rysunku 3.47 wszystkie sygnały wejściowe projektu mają ustaloną wartość logiczną 0, natomiast wartości sygnałów wyjściowych są nieokreślone . Użytkownik powinien samodzielnie stworzyć przebiegi sygnałów wejściowych, podczas gdy ich wartości będą obliczane w procesie symulacji. Żądany kształt sygnałów wejściowych można bardzo sprawnie sformować za pomocą narzędzi edytora przebiegów (*Selection Tool*), które można aktywować, klikając ikonę .

Aby przypisać wartość sygnału wejściowego, należy najpierw zaznaczyć odpowiedni przedział czasowy, a następnie określić jego pożądaną wartość. Zaznaczanie przedziału czasowego odbywa się przez kliknięcie na początek wybranego przedziału i przeciągnięcie kursora myszy do jego końca. Po zaznaczeniu przedziału czasowego można ustawić wartość sygnału, wybierając jedną z dostępnych opcji z menu edytora przebiegów           , gdzie:

-  – nieokreślona wartość (*Unknown (x)*);
-  – zero logiczne (*Low (0)*);
-  – jedynka logiczna (*High (1)*);
-  – stan wysokiej impedancji (*High impedance (Z)*);
-  – słaby niski poziom (*Weak Low (L)*);
-  – słaby wysoki poziom (*Weak High (H)*);
-  – negacja aktualnej wartości sygnału (*Invert*);
-  – podanie wartości liczbowej (*Count Value*);
-  – ustawienie sygnału synchronizacji (*Overwrite Clock*);
-  – dowolna wartość (*Arbitrary Value*);
-  – losowa wartość (*Random Values*).

Dla naszego projektu ustawimy wszystkie możliwe kombinacje sygnałów wejściowych, tak jak pokazano na rysunku 3.48.

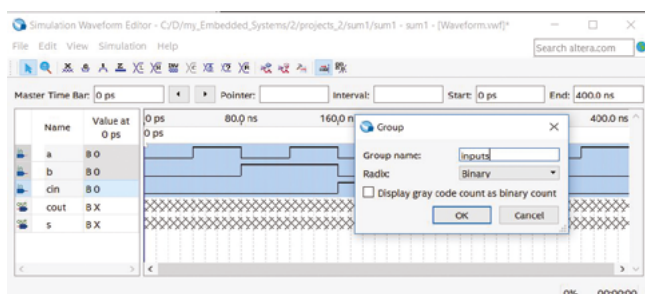


RYS. 3.48. Przebiegi czasowe sygnałów wejściowych

3.10.2. Grupowanie i rozgrupowanie sygnałów

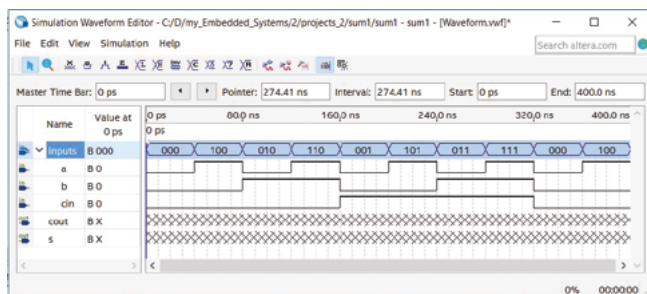
Operacja grupowania sygnałów umożliwia połączenie wybranych linii w jedną magistralę (szynę) i odwoływanie się do nich jako do wektora sygnałów. Aby połączyć sygnały w grupę, należy wykonać następujące kroki:

1. Zaznaczyć sygnały, które mają być połączone. Można to zrobić przez kliknięcie na każdym z grupowanych sygnałów, przytrzymując klawisz **Ctrl**.
2. Zgrupować zaznaczone sygnały, wybierając opcję **Grouping > Group...** z menu rozwijanego, dostępnego po kliknięciu prawym przyciskiem myszy. Spowoduje to otwarcie okna, jak pokazano na rysunku 3.49.



RYS. 3.49. Grupowanie sygnałów

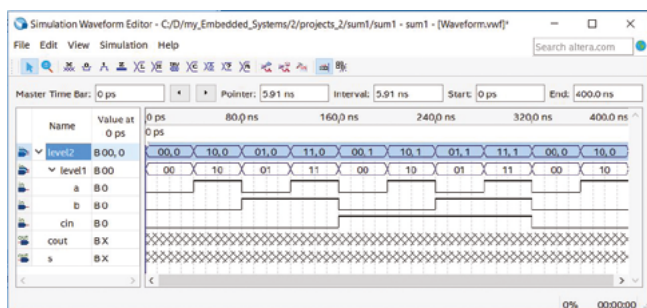
3. Nadać nazwę grupie sygnałów (*Group name*) oraz określić podstawę systemu liczbowego (*Radix*), w którym będą podawane wartości danych na magistrali. Utworzona grupa została pokazana na rysunku 3.50.



RYS. 3.50. Wynik grupowania sygnałów

Każdy sygnał wektorowy (szynę) można rozgrupować – podzielić na osobne linie. Aby to zrobić, wystarczy kliknąć prawym przyciskiem myszy na grupę sygnałów i wybrać opcję **Grouping > Ungroup...** z pojawiającego się menu.

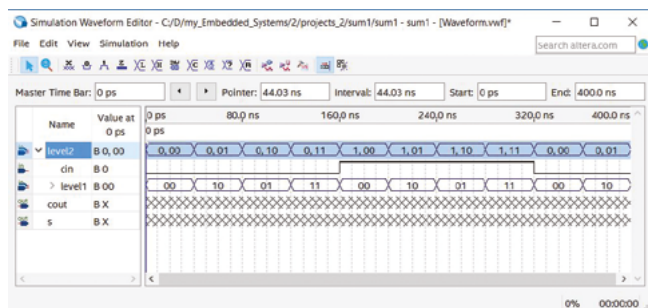
Można też tworzyć hierarchiczne grupy sygnałów, łącząc w grupę szynę z oddzielnymi sygnałami lub scalając razem kilka grup sygnałów. W przykładzie na rysunku 3.51 dwuliniowy sygnał *level1*, który zawiera sygnały *a* i *b*, został połączony z pojedynczym sygnałem *cin*. W rezultacie uzyskano grupę hierarchiczną *level2* składającą się z trzech sygnałów.



RYS. 3.51. Grupowanie szyny z pojedynczym sygnałem

3.10.3. Zmiana porządku bitowego sygnałów w szynie

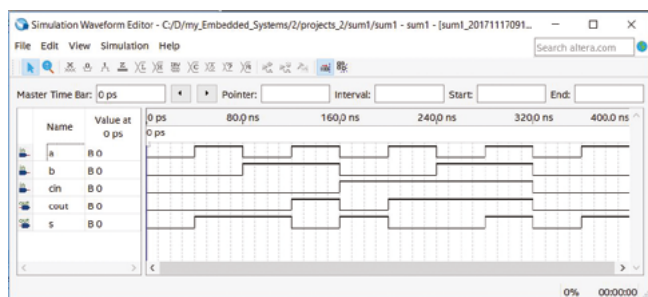
W edytorze przebiegów czasowych możliwa jest zmiana porządku bitowego linii w zgrupowanym sygnale (szynie). Aby to zrobić, wystarczy kliknąć prawym przyciskiem myszy na sygnał grupowy i z menu kontekstowego wybrać opcję **Reverse Group or Bus Bit Order**. Po wykonaniu tej operacji porządek bitów sygnałów w szynie zostanie odwrócony (rysunek 3.52).



RYS. 3.52. Zmiana porządku bitów w zgrupowanym hierarchicznym sygnale *level2*

3.10.4. Symulacja funkcjonalna

Symulacja funkcjonalna polega na założeniu, że wszystkie elementy logiczne i połączenia mają zerowe opóźnienia czasowe. Innymi słowy, ten rodzaj symulacji sprawdza jedynie poprawność funkcjonowania projektu, bez uwzględniania opóźnień propagacyjnych. Proces symulacji funkcjonalnej można uruchomić poprzez wybór opcji **Simulation > Run Functional Simulation** lub za pomocą ikony  w edytorze przebiegów czasowych. Po uruchomieniu symulacji pojawi się okno informujące o postępie procesu. Po zakończeniu symulacji wyniki zostaną wyświetlone w edytorze przebiegów, gdzie zostaną przedstawione przebiegi czasowe sygnałów wyjściowych (rysunek 3.53).



RYS. 3.53. Wynik symulacji funkcjonalnej

Wyniki symulacji funkcjonalnej wymagają dokładnej analizy, aby upewnić się co do poprawności funkcjonowania projektu. W tym celu przydatny może być kursor przebiegów czasowych, który jest wyświetlany na ekranie jako pogrubiona linia pionowa. Można go umieścić w dowolnym miejscu na wykresie i odczytać wartości sygnałów w kolumnie **Value at** dla ustalonej pozycji. W tej kolumnie można przeanalizować wektor sygnałów wejściowych i odpowiadające mu wartości generowane na wyjściach schematu. Możliwe jest również ustawienie przyciągania kursora do siatki lub do granic przedziałów czasowych za pomocą ikon  i  odpowiednio.

3.10.5. Symulacja czasowa

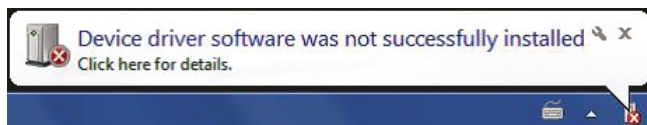
Symulację czasową uruchamiamy poleceniem **Simulation > Run Timing Simulation** lub klikając na ikonę  w oknie **Simulation** edytora przebiegów czasowych. Następnie w oknie tego edytora pojawiają się wyniki symulacji (rysunek 3.53). W naszym przypadku wynik symulacji jest identyczny z symulacją funkcjonalną. Wynik symulacji na rysunku 3.53 nie uwzględnia opóźnień sygnałów, ponieważ system Quartus Prime wykonuje symulację funkcjonalną jedynie dla układów FPGA z rodziny Cyclone IV i Stratix IV, podczas gdy nasz projekt jest przeznaczony dla układów FPGA z rodziny Cyclone V. Aby uzyskać symulację uwzględniającą opóźnienia, można wybrać inną rodzinę układów FPGA (na przykład Cyclone IV) w opcji **Assignment > Device...**. Jednak wyniki takiej symulacji czasowej nie będą odzwierciedlać naszego projektu, który docelowo będzie implementowany w strukturze układu FPGA z rodziny Cyclone V [2].

W takiej sytuacji zamiast przeprowadzania symulacji czasowej zaleca się wykonanie analizy czasowej projektu przy użyciu analizatora czasowego TimeQuest, który oferuje bardziej zaawansowane możliwości niż symulacja czasowa.

3.11. Programowanie i konfigurowanie układu FPGA

Plik zawierający dane niezbędne do zaprogramowania i skonfigurowania układu FPGA jest generowany w procesie kompilacji przez moduł **Assembler** kompilatora systemu Quartus. Jest to plik o rozszerzeniu SOF (*SRAM Object File*), który zawiera dane konfiguracyjne potrzebne do implementacji projektu w strukturze docelowego układu FPGA. W przypadku naszego przykładu będzie to układ FPGA umieszczony na płycie DE1-SoC, która jest podłączana do komputera za pomocą złącza USB. Aby umożliwić to połączenie, konieczne jest zainstalowanie na komputerze sterownika USB-Blaster dostarczanego wraz z oprogramowaniem Quartus.

Aby przygotować płytkę prototypową DE1-SoC do pracy, należy połączyć ją z komputerem kablem USB-Blaster oraz podłączyć do źródła zasilania. Jeśli na ekranie komputera pojawi się komunikat o braku sterownika (rysunek 3.54), trzeba poinformować system o lokalizacji sterownika USB-Blaster na dysku (zwykle znajduje się w katalogu oprogramowania Quartus \quartus\drivers\usb-blaster). Po jego zainstalowaniu można przystąpić do programowania układu FPGA.

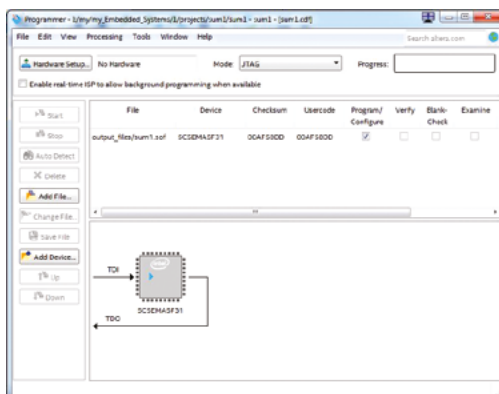


RYS. 3.54. Komunikat systemu operacyjnego o braku sterownika

W systemie projektowania Quartus programowanie układu FPGA lub płytki prototypowej wykonuje się za pomocą narzędzia *programatora* (*Programmer*). W celu zaprogramowania płytki DE1-SoC należy zrealizować następujący scenariusz.

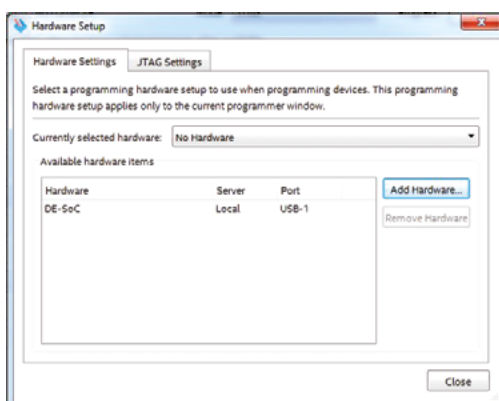
Algorytm 3.7. Programowanie płytki DE1-SoC

1. Uruchomienie programatora. Po wybraniu z menu systemu Quartus komendy **Tools > Programmer** na ekranie pojawi się okno, jak na rysunku 3.55.



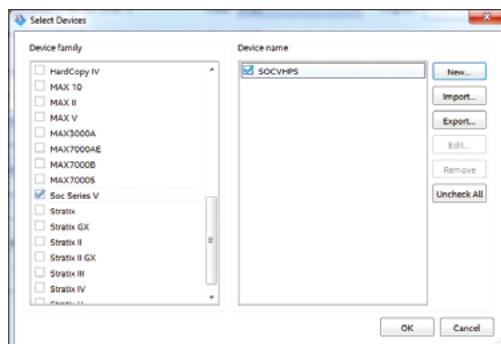
RYS. 3.55. Okno programatora

2. Wybór rodzaju sprzętu oraz trybu programowania. W polu **Mode** należy wybrać tryb programowania **JTAG** (jeśli nie jest on ustawiony domyślnie). Następnie, aby określić sprzęt docelowy, klikamy przycisk **Hardware Setup...** i w ukazującym się oknie (rysunek 3.56) wybieramy **DE-SoC**.



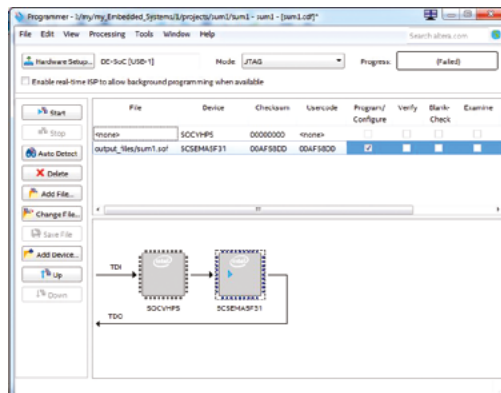
RYS. 3.56. Ustawienie rodzaju płytki prototypowej

3. Dołączenie pliku konfiguracji. Jeśli plik konfiguracyjny układu FPGA sum1.sof nie został automatycznie dołączony przez *programator* (nie pojawił się w kolumnie **File** na rysunku 3.55), trzeba dodać go ręcznie. W tym celu należy kliknąć przycisk **Add File...** i wybrać plik konfiguracji projektu sum1.sof z podkatalogu output_files, który znajduje się w katalogu bieżącego projektu.
4. Wybór rodziny układów FPGA. Aby wybrać układ docelowy, należy kliknąć przycisk **Add Device...** (rysunek 3.55). W otwartym oknie dialogowym (rysunek 3.57) w sekcji **Device family** należy wybrać **Soc Series V**. Następnie w prawej ramce **Device name** pojawi się nazwa układu SOCVHPS. Należy odhaczyć pole wyboru przy tej nazwie i nacisnąć przycisk **OK**.



RYS. 3.57. Ustawienie modułu SOCVHPS do programowania układu FPGA rodziny Cyclone V

5. Sprawdzenie kolejności występowania układów. Należy się upewnić, że kolejność występowania układów jest poprawna (SOCVHPS → 5CSEMA5F31), zgodnie z tym, co jest pokazane na rysunku 3.58. Aby dokonać zmiany kolejności, należy kliknąć na wybrany układ myszą, a następnie skorzystać z przycisków  **Up** lub  **Down**.



RYS. 3.58. Ustawienie modułu do programowania układu 5CSEMA5F31

6. Uruchomienie procesu programowania. Programowanie układu rozpoczynamy, naciskając przycisk **Start** (rysunek 3.58). W trakcie konfigurowania układu FPGA możemy obserwować świecenie diody LED na płycie prototypowej. W przypadku pojawienia się komunikatów o błędach generowanych przez programator należy odpowiednio zareagować, na przykład sprawdzając poprawność podłączenia płytki.

3.12. Testowanie projektu

Po skonfigurowaniu układu FPGA można przystąpić do testowania zaimplementowanego projektu. W celu sprawdzenia poprawności działania układu należy za pomocą przełączników SW0, SW1 i SW2 ustawić wartości sygnałów wejściowych a , b i cin odpowiednio. Następnie należy obserwować wyniki działania układu na diodach LED0 i LED1, które prezentują wartości sygnałów wyjściowych sumy s i przeniesienia $cout$.

3.13. Podsumowanie rozdziału

Typowy proces projektowania systemów wbudowanych w systemie Quartus obejmuje następujące podstawowe etapy:

- *wprowadzenie projektu (Design Entry)* – specyfikacja projektu w języku Verilog (VHDL) lub w postaci schematu projektu;
- *synteza (Synthesis)*;
- *symulacja funkcjonalna (Functional Simulation)* – weryfikacja poprawności funkcjonowania projektu;
- *upakowanie (Fitting)* – odwzorowanie elementów logicznych projektu w strukturę docelowego układu FPGA;
- *symulacja czasowa (Timing Simulation)* – sprawdzenie poprawności funkcjonalnej projektu z uwzględnieniem opóźnień czasowych sygnałów;
- *analiza czasowa (Timing Analysis)* – sprawdzenie warunków zapewnienia ograniczeń czasowych projektu i określenie przewidywanej wydajności systemu;
- *programowanie i konfiguracja (Programming and Configuration)* – fizyczna implementacja projektu w docelowym układzie FPGA.

Praca nad nowym projektem w systemie Quartus zaczyna się od utworzenia na dysku katalogu projektu, w którym będą przechowywane wszystkie jego pliki (baza danych projektu). Zaleca się przypisanie każdemu projektowi osobnego katalogu na dysku. Nowy projekt jest tworzony za pomocą *kreatora nowych projektów (New Project Wizard)*.

Plik programu w języku Verilog jest zwykłym plikiem tekstowym w formacie ASCII z rozszerzeniem *.v*. Można go utworzyć w dowolnym edytorze tekstu ASCII, jednak zaleca się korzystanie z edytora tekstowego systemu Quartus, który oferuje zestaw szablonów języka Verilog.

Do wykorzystania są następujące szablony:

- gotowe projekty (*Full Designs*);
- konstrukcje języka Verilog (*Constructs*);
- elementy logiczne (*Logic*);
- atrybuty syntezy (*Synthesis Attributes*);
- prymitywy firmy Altera (*Altera Primitives*).

W przypadku rozbudowanego projektu, który składa się z wielu plików, wszystkie wymagane pliki powinny być dołączone do projektu, a ich lokalizacja na dysku powinna być wskazana.

Każdy nowy lub zmodyfikowany projekt w systemie Quartus powinien być skompilowany. Kompilator systemu Quartus łączy różne narzędzia programistyczne, takie jak syntezyzator (*Analysis & Synthesis*), narzędzie upakowania projektu (*Fitter*) czy też analizator czasowy TimeQuest. Raport kompilacji (*Compilation Report*) zawiera wiele rozdziałów i podrozdziałów, które dostępne są w oknie **Table of Contents**.

Po pomyślnie zakończonym procesie kompilacji projektu wyniki syntezy można obejrzeć w postaci:

- schematu na poziomie przesłań międzyrejestrowych (*Register Transition Level – RTL*);
- mapy technologicznej projektu po odwzorowaniu na układ docelowy FPGA (*Technology Map Post-Mapping*);
- mapy technologicznej po upakowaniu projektu (*Technology Map Fitting*);
- automatu skończonego (za pomocą *State Machine Viewer*).

Analiza schematu na poziomie przesłań międzyrejestrowych pozwala sprawdzić, czy otrzymany w wyniku syntezy schemat odpowiada oczekiwaniom użytkownika. Wiele przydatnych do analizy projektu informacji można odczytać także z map technologicznych projektu oraz tablicy przejść i grafu automatu skończonego.

Wszystkie błędy kompilacji projektu powinny być poprawione, ponieważ bez tego niemożliwe jest przejście do kolejnych etapów projektowania. Ostrzeżenia kompilatora, zarówno te zwykłe (*Warning Messages*) jak i krytyczne (*Critical Warning Messages*), również należy uważnie przeanalizować.

Przyporządkowanie wyprowadzeń FPGA zewnętrznym sygnałom projektu należy wykonywać w przypadku, gdy planowana jest fizyczna realizacja projektu. System Quartus udostępnia dwa sposoby przypisywania wyprowadzeń FPGA do sygnałów wejściowych i wyjściowych projektu: za pomocą edytora przypisań (*Assignment Editor*) oraz z wykorzystaniem pliku ustawień (*Quartus Setting File – QSF*).

System Quartus umożliwia przeprowadzenie dwóch rodzajów symulacji projektu: *funkcjonalnej (functional simulation)* oraz *czasowej (timing simulation)*. Proces symulacji należy rozpocząć od utworzenia zbioru danych testowych, które będą podawane na wejścia układu. Taki zbiór nazywany jest zbiorem *wektorów testowych (test vectors)*. Wektory testowe w systemie Quartus przygotowywane są za pomocą *edytora przebiegów czasowych (Simulation Waveform Editor)*.

Informacja o konfigurowaniu układu FPGA zapisywana jest do pliku konfiguracyjnego z rozszerzeniem SOF (*SRAM Object File*), który jest generowany w procesie kompilacji przez moduł **Assembler** kompilatora systemu Quartus. Konfigurowanie układu FPGA lub płytki prototypowej wykonywane jest za pomocą narzędzia systemu Quartus nazywanego *programatorem (Programmer)*.

4. Projektowanie układów kombinacyjnych

Niezbędnymi składnikami każdego systemu wbudowanego są różnorodne oryginalne układy kombinacyjne, które muszą być zaprojektowane od podstaw. Chociaż czasami w procesie ich projektowania można skorzystać z gotowych bibliotek standardowych bloków funkcjonalnych, w wielu przypadkach stworzenie własnego bloku kombinacyjnego pozwala uzyskać rozwiązanie bardziej efektywne i lepiej dopasowane do wymagań projektowych.

W niniejszym rozdziale skoncentrujemy się na projektowaniu układów kombinacyjnych za pomocą języka Verilog. W rozdziale 2 omówiliśmy już wiele związanych z tym aspektów. Dlatego teraz skupimy się głównie na przedstawieniu krótkiego przeglądu elementów języka Verilog stosowanych do opisu układów kombinacyjnych, omówimy również metodologię projektowania oraz przeanalizujemy szczegóły implementacji bloków kombinacyjnych w strukturze układów programowalnych FPGA.

4.1. Elementy języka Verilog wykorzystywane w projektowaniu układów kombinacyjnych

Do opisu układów kombinacyjnych w języku Verilog mogą być wykorzystane następujące konstrukcje:

- prymitywy języka Verilog;
- prymitywy użytkownika;
- instrukcja ciągłego przypisania **assign**;
- instrukcje proceduralne;
- funkcje i zadania.

4.1.1. Prymitywy

W języku Verilog zdefiniowane są prymitywy dwóch rodzajów – bramki i bufory [14]. Ponadto dodatkowe prymitywy są wprowadzane do wykorzystania w języku Verilog przez kompilatory oraz systemy projektowania. Gdy korzysta się z prymitywów, projektant powinien ręcznie wykonać syntezę układu kombinacyjnego i przedstawić go w postaci schematu składającego się ze standardowych prymitywów bramkowych.

Takie podejście wiąże się z większym ryzykiem popełnienia błędów wynikających z ręcznego projektowania, zwłaszcza w przypadku bardzo złożonych schematów. Niemniej jednak przy projektowaniu niewielkich układów kombinacyjnych lub włączaniu pojedynczych bramek do projektu prymitywy języka Verilog mogą się okazać bardzo przydatne. Ponadto prymitywy często są wykorzystywane do implementacji *logiki klejowej (glue logic)* przy łączeniu komponentów projektu.

4.1.2. Prymitywy użytkownika

Bardziej przydatne do modelowania układów kombinacyjnych wydają się *prymitywy definiowane przez użytkownika (user defined primitives – UDP)*. Dzięki ich mechanizmowi można zdefiniować niewielkie bloki kombinacyjne, które często znajdują zastosowanie w projektach. W przypadku tych prymitywów blok kombinacyjny opisywany jest w postaci tablicy prawdy za pomocą instrukcji **table** [14] (w języku Verilog jest ona stosowana wyłącznie do definiowania prymitywów użytkownika).

Ograniczeniem prymitywów użytkownika jest ich limitowana liczba wejść (10 dla prymitywów kombinacyjnych i 9 dla sekwencyjnych) oraz posiadanie tylko jednego wyjścia. Dodatkowo nie wszystkie kompilatory wspierają mechanizm tworzenia prymitywów przez użytkownika. Kompilator języka Verilog w systemie Quartus umożliwia definiowanie zarówno kombinacyjnych, jak i sekwencyjnych prymitywów użytkownika.

4.1.3. Instrukcja ciągłego przypisania assign

Instrukcja ciągłego przypisania **assign** jest wyjątkowo użyteczna do modelowania logiki kombinacyjnej. Za jej pomocą można opisywać układy kombinacyjne, wykorzystując wyrażenia logiczne funkcji boolowskich, które reprezentują logikę danego układu. Instrukcja ta nie narzuca żadnych ograniczeń na liczbę wejść i wyjść układu kombinacyjnego (w granicach możliwości FPGA i kompilatora). W wyrażeniach zawartych w instrukcji **assign** mogą występować dowolne instrukcje Verilog, w tym operatory arytmetyczne (choć z pewnymi zastrzeżeniami), przesunięcia bitowe, operatory warunkowe, konkatenacja oraz operatory powtórzenia.

Podczas korzystania z instrukcji **assign** należy upewnić się, że rozmiary operandów po obu stronach przypisania są identyczne, ponieważ automatyczne dopasowywanie rozmiarów przez kompilator może prowadzić do powstania błędów trudnych do wykrycia.

4.1.4. Instrukcje proceduralne

Układy kombinacyjne w języku Verilog mogą być opisywane również za pomocą instrukcji proceduralnych wewnątrz bloku **always**. Na jego liście czułości powinny znaleźć się wszystkie sygnały wejściowe danego bloku, czyli wszystkie zmienne występujące po prawej stronie znaku równości instrukcji przypisania blokującego (w opisie układów kombinacyjnych w blokach **always** zawsze używamy instrukcji przypisania blokującego =).

Aby uniknąć pominięcia jakiegokolwiek sygnału wejściowego na liście czułości bloku **always**, należy zastosować poniższy format deklaracji bloku:

```
always @(*) ... ;
```

gdzie zamiast gwiazdki kompilator automatycznie podstawia listę wszystkich zmiennych wejściowych danego bloku **always**.

Instrukcje proceduralne umożliwiają opisywanie układów kombinacyjnych na podstawie algorytmu ich działania, zamiast opierania się na schemacie lub wyrażeniach logicznych. Najczęściej używane do modelowania układów kombinacyjnych są instrukcje **if** i **case** (oraz ich warianty **casez** i **casex**). Instrukcje pętli **for**, **while** i **repeat** mają ograniczone zastosowanie; konieczne jest, aby liczba iteracji była znana przed użyciem pętli w kodzie. Dodatkowo wiele kompilatorów narzuca ograniczenie na maksymalną liczbę iteracji pętli. Instrukcje proceduralne **forever** i **disable** nie są w ogóle uwzględniane przez syntezyzator i mogą być używane jedynie do symulacji projektu.

4.1.4.1. Instrukcja **if**

Przy wykorzystaniu instrukcji **if** do syntezy układów kombinacyjnych należy się upewnić, że wszystkie wyjścia układu są jednoznacznie określone dla wszystkich możliwych kombinacji zmiennych wejściowych. Brak spełnienia tego warunku dla którejkolwiek z wyjść wymagać będzie podtrzymywania poprzedniej wartości sygnału na tym wyjściu. W takim przypadku kompilator automatycznie umieści na wyjściu zatrask (przezroczysty przerzutnik typu D), co spowoduje, że układ kombinacyjny zostanie przekształcony w sekwencyjny, ponieważ będzie zawierał elementy pamięci. Taka sytuacja będzie sprzeczna z założeniami naszego zadania polegającego na syntezie układu kombinacyjnego.

Można zastosować następujące sposoby rozwiązania tego problemu:

- każdemu słowu kluczowemu **if** w opisie układu kombinacyjnego powinno odpowiadać słowo kluczowe **else**;
- przed wykorzystaniem instrukcji **if** w bloku **always** należy określić domyślne wartości wyjść układu kombinacyjnego.

Na przykładzie rozważmy realizację sprzętową funkcji logicznej f od trzech argumentów a , b i c , takiej że przy $a = 1$ funkcja przyjmuje wartość $f = b \& c$, w przeciwnym razie $f = 0$. Poniższy kod prezentuje niepoprawny opis funkcji f :

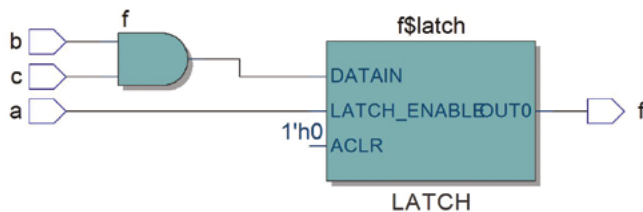
```

module f_with_latch (
    output reg f,
    input a, b, c);

    always @(*)
    if (a==1) f = b & c;
endmodule

```

W tym przykładzie instrukcja **if** oblicza wartość funkcji f tylko dla wartości argumentu $a = 1$, jednak pozostaje ona nieokreślona dla $a = 0$. W rezultacie podczas syntezy kompilator dodaje zatrząsk na wyjściu (rysunek 4.1).



RYS. 4.1. Niepoprawna realizacja funkcji f z umieszczeniem zatrząsku na wyjściu

Błąd w kodzie programu można naprawić, uzupełniając instrukcję **if** konstrukcją **else**:

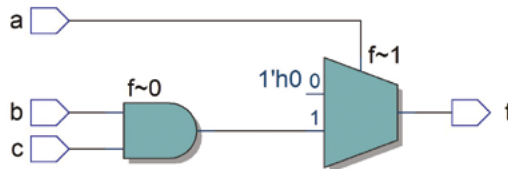
```

module f_with_else (
    output reg f,
    input a, b, c);

    always @(*)
    if (a==1)    f = b & c;
    else        f = 1'b0;
endmodule

```

W wyniku zostanie zsyntezowany poprawny układ kombinacyjny (rysunek 4.2).



RYS. 4.2. Realizacja funkcji f z wykorzystaniem konstrukcji **else** w operatorze **if**

Innym sposobem rozwiązania problemu jest określenie w bloku **always** przed operatorem **if** domyślnej wartości funkcji *f*:

```
module f_with_starting_value (  
    output reg f,  
    input a, b, c);  
  
    always @(*)  
    begin  
        f = 1'b0;  
        if (a==1)      f = b & c;  
    end  
endmodule
```

W wyniku syntezy powyższego przykładu otrzymamy identyczny schemat jak na rysunku 4.2. Kompilator systemu Quartus zawsze informuje o każdym automatycznie dodanym zatrasku do schematu. Dlatego szczególnie ważne jest, aby podczas syntezy układów kombinacyjnych dokładnie przeanalizować wszystkie ostrzeżenia generowane przez kompilator.

4.1.4.2. Instrukcja **case**

Instrukcja **case** pozwala opisywać układy kombinacyjne w innym stylu, bardziej przypominającym reprezentację tablicy prawdy funkcji logicznych. W takim opisie dla różnych wartości zmiennych wejściowych określone są wartości wyjść układu kombinacyjnego. Nie wymaga to zapisywania wyrażeń logicznych (jak w przypadku instrukcji **if**), co może znacznie ograniczyć ryzyko popełnienia błędów przy ręcznym wyprowadzaniu tych wyrażeń.

Przy wykorzystaniu do opisu układu kombinacyjnego instrukcji **case**, podobnie jak w przypadku instrukcji **if**, należy określić wartości sygnałów wyjściowych dla wszystkich możliwych kombinacji zmiennych wejściowych. W przeciwnym razie kompilator automatycznie umieści zatraski na wyjściach schematu. Aby tego uniknąć, należy w operatorze **case** użyć konstrukcję **default**, która pozwala określić wartości wyjść układu kombinacyjnego dla tych kombinacji sygnałów wejściowych, które nie zostały wymienione w operatorze **case**.

Funkcja *f* z poprzedniego przykładu może być niepoprawnie opisana za pomocą instrukcji **case**, jak przedstawiono w poniższym kodzie:

```
module f_with_case_latch (  
    output reg f,  
    input a, b, c);  
  
    always @(*)  
        case (a)
```

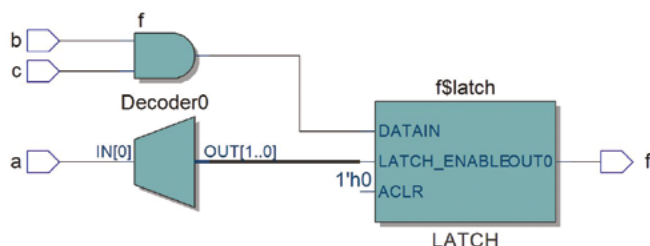


```

        1'b1: f=b & c;
    endcase
endmodule

```

Po wykonaniu kompilacji przedstawionego kodu uzyskany zostanie schemat z zatrzaskiem na wyjściu (rysunek 4.3).



rys. 4.3. Niepoprawna realizacja funkcji f za pomocą instrukcji **case**

Zastosowanie konstrukcji **default** pozwala określić domyślne wartości wyjść układu kombinacyjnego:

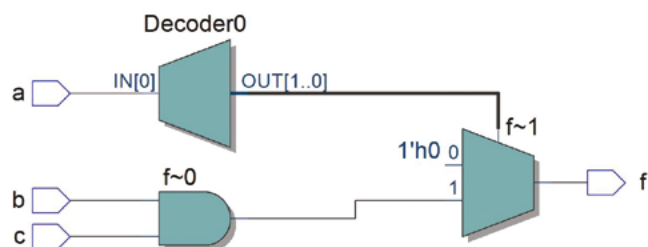
```

module f_with_case_default (
    output reg f,
    input a, b, c);

    always @(*)
        case (a)
            1'b1: f=b & c;
            default: f=1'b0;
        endcase
endmodule

```

W wyniku zostanie wygenerowany układ kombinacyjny bez zatrząskó (rysunek 4.4).



rys. 4.4. Realizacja funkcji f z wykorzystaniem konstrukcji **default** w operatorze **case**

Aby skrócić zapis instrukcji **case**, zazwyczaj w konstrukcji **default** podaje się wartość funkcji, która najczęściej występuje w tablicy prawdy.

Instrukcja **case** jest szczególnie przydatna do opisu bloków funkcjonalnych, takich jak dekodery, konwertery kodów i innych, których przykłady syntezy zostaną przedstawione w następnym rozdziale.

4.1.5. Funkcje i zadania

Opis układów kombinacyjnych może być również realizowany przy użyciu funkcji lub zadań. Funkcja zwraca wartość liczbową w miejscu jej wywołania, więc może być wykorzystana w wyrażeniach. Z kolei odwołanie do zadania wykonuje się przez oddzielną instrukcję, przypominającą tworzenie instancji modułu. Warto zauważyć, że funkcja zawsze powinna mieć argumenty i zwracać tylko jedną wartość, podczas gdy zadanie może zwracać wiele wartości, ale nie może przyjmować argumentów. Dodatkowo należy wcześniej zapoznać się z ograniczeniami narzucanymi przez kompilator dotyczącymi wykorzystania funkcji i zadań.

4.2. Metodologia projektowania układów kombinacyjnych

Metodologia projektowania układów kombinacyjnych w strukturach FPGA obejmuje następujące etapy:

- opis funkcjonowania układu kombinacyjnego;
- minimalizacja funkcji logicznych;
- przedstawienie układu kombinacyjnego w postaci układu funkcji logicznych;
- przedstawienie układu kombinacyjnego w postaci algorytmu;
- opis układu w języku HDL (na przykład w języku Verilog).

Poszczególne etapy nie zawsze są wymagane w procesie projektowania, ponieważ każdy układ kombinacyjny może być projektowany na różne sposoby.

4.2.1. Opis funkcjonowania układu kombinacyjnego

Ten etap projektowania jest bardzo istotny, ponieważ popełnione tu błędy mogą doprowadzić do zbudowania zupełnie innego schematu niż zakładany. Istnieje wiele metod reprezentacji układów kombinacyjnych:

- tablica prawdy;
- tablica Karnaugh (*Karnaugh maps*);
- kanoniczna postać sumy (KPS);
- kanoniczna postać iloczynu (KPI);

- skrócony zapis dziesiętny;
- binarne diagramy decyzyjne (*binary decision diagram* – BDD);
- opis słowny i inne.

Tablice prawdy są użyteczne przy projektowaniu układów kombinacyjnych za pomocą instrukcji **case**. Ograniczeniem może być ich bardzo duży rozmiar, gdyż stają się one mniej wygodne przy opisie funkcji z dużą liczbą argumentów.

Tablice Karnaugh często znajdują zastosowanie w przedstawianiu prostych funkcji logicznych o niewielkiej liczbie argumentów (zwykle do pięciu). Pozwalają one w wygodny sposób ręcznie zminimalizować funkcje logiczne. Ponieważ w wielu układach FPGA liczba wejść elementów logicznych LUT jest ograniczona do 4, tablice Karnaugh nadal są szeroko stosowane w praktyce inżynierskiej.

Kanoniczna postać sumy pozwala opisywać funkcje logiczne jako sumę pełnych iloczynów (minitermów), co ułatwia reprezentację układów kombinacyjnych za pomocą instrukcji przypisania ciągłego **assign**.

Alternatywnym sposobem zapisu funkcji logicznej jest kanoniczna postać iloczynu, gdzie funkcja jest przedstawiana jako iloczyn pełnych sum (makstermów). W praktyce taka reprezentacja funkcji jest wykorzystywana przy realizacji układu na bramkach NOR.

Skrócone zapisy dziesiętne stanowią kompaktowy sposób zapisu kanonicznych postaci funkcji logicznych KPS lub KPI. Opisują one funkcje logiczne za pomocą zbioru indeksów dziesiętnych minitermów lub makstermów.

Binarne diagramy decyzyjne znajdują szerokie zastosowanie w metodach syntezy i modelowania układów kombinacyjnych. Ten sposób opisu funkcji ułatwia komputerową reprezentację funkcji logicznych, jednak w praktyce inżynierskiej jest rzadko stosowany.

Opis słowny funkcji jest trudny do sformalizowania, dlatego w celu dalszej syntezy powinien być zastąpiony jednym z omówionych powyżej sposobów reprezentacji funkcji logicznych.

Przedstawimy sposoby opisu funkcji logicznych na przykładzie funkcji $f = f(a, b, c)$, której tablica prawdy została pokazana w tabeli 4.1.

TABELA 4.1. Tablica prawdy funkcji f

	<i>a b c</i>	<i>f</i>
0	0 0 0	0
1	0 0 1	1
2	0 1 0	1
3	0 1 1	1
4	1 0 0	1
5	1 0 1	0
6	1 1 0	0
7	1 1 1	1

Tablica Karnaugh dla funkcji f została przedstawiona na rysunku 4.5. Grupy sąsiednich kombinacji zaznaczono przerywanymi liniami, co umożliwia minimalizację funkcji.

		bc			
a		00	01	11	10
	0	0	1	1	1
	1	1	0	1	0

RYS. 4.5. Tablica Karnaugh dla funkcji f

Po przeprowadzeniu minimalizacji możemy przedstawić funkcję f w postaci minimalnej:

$$f = \bar{a}c + \bar{a}b + bc + a\bar{b}\bar{c}.$$

Kanoniczne postaci KPS i KPI funkcji f możemy otrzymać bezpośrednio z tablicy prawdy:

$$f = \bar{a}bc + \bar{a}b\bar{c} + abc + a\bar{b}\bar{c}.$$

$$f = (a + b + c)(\bar{a} + b + \bar{c})(\bar{a} + \bar{b} + c).$$

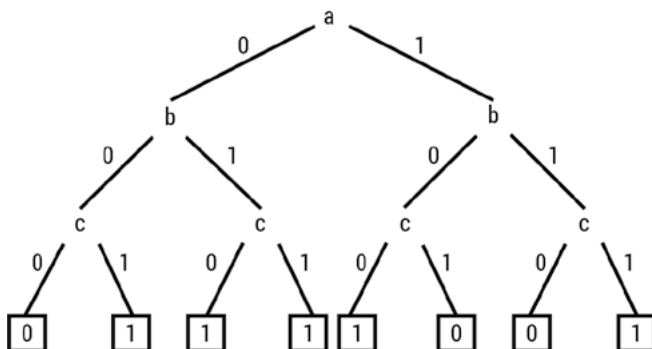
Skrócony zapis dziesiętny funkcji f w postaci sumy *minitermów* prezentuje się następująco:

$$f(a, b, c) = \Sigma(1, 2, 3, 4, 7).$$

Z kolei skrócony zapis dziesiętny funkcji f w postaci iloczynu *makstermów* przedstawimy jak niżej:

$$f(a, b, c) = \Pi(0, 5, 6).$$

Reprezentacja funkcji f w postaci binarnego diagramu decyzyjnego zaprezentowana została na rysunku 4.6.



RYS. 4.6. Binarny diagram decyzyjny dla funkcji f

4.2.2. Minimalizacja funkcji logicznych

Większość współczesnych systemów projektowania na układach FPGA, w tym również Quartus, automatycznie wykonuje minimalizację logiki projektu podczas syntezy. Nie wymaga to żadnych dodatkowych działań ze strony użytkownika. Czasami jednak może zaistnieć potrzeba dodatkowej minimalizacji funkcji logicznych, co może być uzasadnione optymalizacją wyników syntezy. W takich przypadkach można skorzystać z dedykowanych narzędzi, na przykład akademickiego programu komputerowej minimalizacji funkcji ESPRESSO [1]. Przy realizacji kilku funkcji logicznych w układzie FPGA zaleca się stosowanie *oddzielnej minimalizacji* funkcji logicznych, co zazwyczaj przynosi lepsze wyniki w porównaniu z *minimalizacją wspólną*.

Warto zauważyć, że automatyczna minimalizacja logiki projektu może prowadzić do sytuacji, gdy niektóre wewnętrzne węzły projektu, które chcielibyśmy monitorować podczas symulacji, zostają usunięte w wyniku optymalizacji. Aby zapobiec temu problemowi, zmienne wewnętrzne poddawane analizie w symulatorze należy opisywać w języku Verilog, korzystając z dyrektywy *keep*.

4.2.3. Opis układu kombinacyjnego za pomocą systemu funkcji logicznych

Powyższe metody opisu funkcji logicznych nie są odpowiednie do reprezentacji złożonych układów kombinacyjnych z dużą liczbą zmiennych wejściowych. Takie układy mogą obejmować szereg standardowych bloków funkcjonalnych, takich jak sumatory, układy mnożące, komparatory i inne. W skomplikowanych wyrażeniach logicznych mogą występować różnorodne funkcje pośrednie lub pomocnicze, takie jak generowanie i propagacja przeniesień, obliczanie iloczynów częściowych, obliczanie sum częściowych i inne. Poprzez wykorzystanie tych funkcji pośrednich można opisać złożone układy kombinacyjne za pomocą systemu funkcji logicznych.

4.2.4. Opis układu kombinacyjnego w postaci algorytmu

Instrukcje proceduralne języka Verilog umożliwiają opisanie algorytmu funkcjonowania projektu, co pozwala na reprezentację układów kombinacyjnych w formie algorytmu.

Przedstawienie układu kombinacyjnego jako algorytmu zostanie omówione na przykładzie projektu sumatora cyfr BCD (*binary-coded decimal*). Wykonuje on dodawanie dwóch cyfr *A* i *B* zakodowanych w kodzie BCD z uwzględnieniem bitu przeniesienia *cin*. Wynik dodawania *S* składa się z dwóch cyfr BCD *s1* i *s0*: $S = \{s1, s0\}$. Funkcjonowanie takiego sumatora można przedstawić w postaci pseudokodu:

```

S = A + B + cin;
if (S > 9) then
    s0 = S - 10;
    s1 = 1;
else
    s0 = S;
    s1 = 0;
endif

```

Opis sumatora BCD w języku Verilog przedstawia się następująco:

```

module adder_BCD_2_digits(
    input [3:0] A,B,
    input cin,
    output [7:0] S);

    reg [3:0] s0,s1;
    wire [4:0] T;           // dodatkowa zmienna

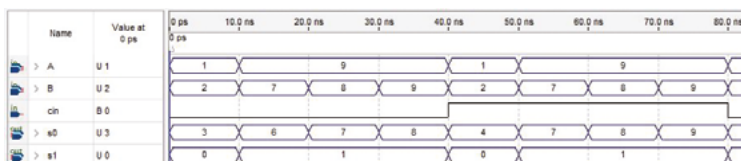
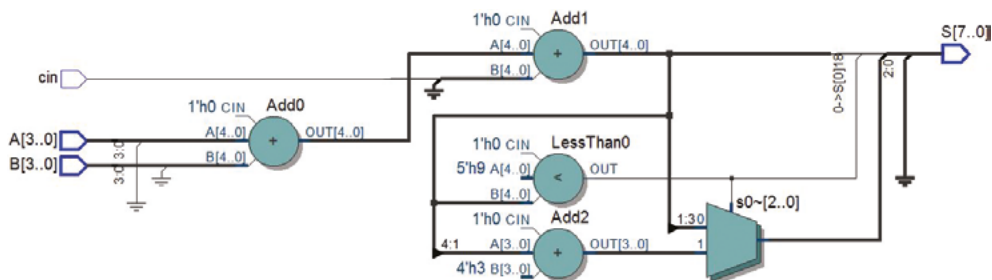
    assign T = A + B + cin; // obliczenie sumy w postaci dwójkowej

    always @(*) begin
        if (T > 9) begin
            s0 = T + 4'b0110;
            s1 = 4'b0001;
        end
        else begin
            s0 = T;
            s1 = 4'b0000;
        end
    end

    assign S = {s1,s0};     // generowanie wyniku
endmodule

```

Zaprojektowany schemat sumatora BCD został przedstawiony na rysunku 4.7, natomiast wyniki symulacji funkcjonalnej układu zostały zaprezentowane na rysunku 4.8.



4.2.5. Opis układu kombinacyjnego w języku Verilog

Układy kombinacyjne mogą być opisywane przy użyciu wspomnianych wcześniej konstrukcji języka Verilog, z zachowaniem określonych ograniczeń. Przy projektowaniu złożonych bloków kombinacyjnych z dużą liczbą wejść mogą być użyteczne bloki generacyjne. W tym samym celu można także wykorzystać funkcje i zadania.

4.3. Zapobieganie wprowadzeniu zatrząsków na wyjściach układów kombinacyjnych

Podczas projektowania układów kombinacyjnych istotne jest uwzględnienie rozwiązań, które zapobiegają automatycznemu umieszczeniu zatrząsków na wyjściach kombinacyjnych. Kompilator samoczynnie dodaje zatrząski na tych wyjściach układu, dla których wartości nie zostały określone dla wszystkich możliwych wartości sygnałów wejściowych. W rezultacie zamiast układu kombinacyjnego otrzymamy układ zawierający elementy pamięci.

Aby zapobiec automatycznemu umieszczaniu zatrząsków na wyjściach układów kombinacyjnych, ważne jest przestrzeganie następujących reguł:

- lista czułości bloku **always** powinna zawierać wszystkie zmienne wejściowe opisywanego bloku, dlatego też instrukcja powinna mieć format: **always** @(*) ...;
- przy wykorzystaniu instrukcji **if** każdej konstrukcji **if** powinna odpowiadać konstrukcja **else**;

- można również zapobiec umieszczaniu zatrząsków, określając domyślne wartości funkcji logicznych na samym początku bloku **always**;
- w instrukcji **case** należy stosować konstrukcję **default** do określenia domyślnych wartości funkcji.

Równie ważne jest dokładne przeanalizowanie komunikatów ostrzegawczych wygenerowanych podczas kompilacji, gdyż kompilator zawsze informuje o automatycznym dołączeniu do projektu każdego zatrząsku.

4.4. Podsumowanie rozdziału

Zaprojektowanie własnego modułu kombinacyjnego często pozwala uzyskać bardziej efektywne rozwiązanie niż gotowe prymitywy lub moduły biblioteczne.

Do opisu układów kombinacyjnych w języku Verilog mogą być wykorzystane następujące konstrukcje:

- prymitywy języka Verilog;
- prymitywy użytkownika;
- instrukcja ciągłego przypisania **assign**;
- instrukcje proceduralne;
- funkcje i zadania.

Wykorzystanie prymitywów bramkowych związane jest z większym prawdopodobieństwem popełnienia błędów w wyniku ręcznego projektowania, zwłaszcza w przypadku bardzo złożonych schematów. Ograniczeniem na wykorzystanie prymitywów użytkownika jest niewielka liczba wejść oraz fakt, że nie wszystkie kompilatory wspierają mechanizm prymitywów tworzonych przez użytkownika. Dlatego też do opisu układów kombinacyjnych przeważnie wykorzystywane są instrukcje proceduralne oraz instrukcja przypisania ciągłego **assign**.

Należy zadbać o to, aby rozmiary operandów po lewej i prawej stronie znaku równości w instrukcji **assign** były zgodne. Automatyczne wyrównywanie rozmiarów operandów przez kompilator może bowiem prowadzić do powstawania trudnych do wykrycia błędów.

Dla układów kombinacyjnych blok **always** powinien mieć następującą postać:

always @(*) ...

Najczęściej do opisu układów kombinacyjnych wykorzystuje się instrukcje proceduralne **if** i **case**. Instrukcje pętli **for**, **while** i **repeat** mają ograniczone zastosowanie, ponieważ wymagają, aby liczba powtórzeń pętli była znana przed ich użyciem w kodzie.

Bardzo ważne dla syntezy układów kombinacyjnych jest to, żeby w instrukcjach proceduralnych **if** i **case** zostały jednoznacznie określone wartości wszystkich wyjść układu dla wszystkich możliwych kombinacji zmiennych wejściowych. W tym celu w instrukcji **if** zalecane jest używanie konstrukcji **else**, a w instrukcji **case** – konstrukcji **default**.

Jeśli nawet dla jednej kombinacji zmiennych wejściowych nie zostanie określona wartość sygnału na przynajmniej jednym wyjściu, kompilator automatycznie umieści zatraski na wyjściach układu kombinacyjnego, co spowoduje jego przekształcenie w układ sekwencyjny. Dlatego istotne jest dokładne analizowanie komunikatów ostrzegawczych kompilatora dotyczących automatycznego dołączania zatrasków do projektu.

Metodologia projektowania układów kombinacyjnych w strukturach FPGA obejmuje następujące etapy:

- opis funkcjonowania układu kombinacyjnego;
- minimalizacja funkcji logicznych;
- przedstawienie układu kombinacyjnego w postaci układu funkcji logicznych;
- przedstawienie układu kombinacyjnego w postaci algorytmu;
- opis układu w języku HDL (na przykład, w języku Verilog).

Najczęściej w praktyce wykorzystywane są następujące sposoby reprezentacji układów kombinacyjnych:

- tablica prawdy;
- tablica Karnaugh (Karnaugh maps);
- kanoniczna postać sumy (KPS);
- kanoniczna postać iloczynu (KPI);
- skrócony zapis dziesiętny;
- binarne diagramy decyzyjne (*binary decision diagram* – BDD);
- opis słowny.

Większość współczesnych systemów projektowania na układach FPGA podczas syntezy automatycznie wykonuje minimalizację logiki projektu, bez konieczności dodatkowych działań ze strony użytkownika. Czasami jednak, w celu uzyskania optymalnych wyników syntezy, można dodatkowo skorzystać ze specjalnych programów do minimalizacji funkcji logicznych, takich jak ESPRESSO. W przypadku realizacji kilku funkcji logicznych w układzie FPGA zaleca się ich *osobną minimalizację*, co zazwyczaj prowadzi do lepszych wyników w porównaniu z *minimalizacją wspólną*.

Złożone wyrażenia logiczne mogą zawierać różne pośrednie lub pomocnicze funkcje logiczne, takie jak funkcje generowania i propagacji przeniesień, obliczenia iloczynów częściowych, obliczenia sum częściowych i inne.

Instrukcje proceduralne języka Verilog umożliwiają opisywanie układów kombinacyjnych w postaci algorytmów.

Przy projektowaniu złożonych układów kombinacyjnych z dużą liczbą wejść przydatne są bloki generacyjne, a także funkcje i zadania.

5. Projektowanie standardowych kombinacyjnych bloków funkcjonalnych

Nieodzownymi elementami systemów wbudowanych są różnego rodzaju standardowe bloki funkcjonalne, takie jak multipleksery, dekodery, komparatory i inne. Niniejszy rozdział poświęcony będzie zagadnieniom projektowania tych bloków z wykorzystaniem języka Verilog przy ich implementacji w strukturach programowalnych układów FPGA.

W systemach wbudowanych często pojawia się potrzeba sterowania przesyłaniem danych, takiego jak zmiana kierunku przesyłania, multipleksowanie linii transmisji w czasie, odbieranie danych od wielu nadajników lub przesyłanie danych z jednego źródła do wielu odbiorników. Do realizacji tych funkcji sterowania wykorzystuje się multipleksery, demultipleksery oraz komutatory.

W systemach wbudowanych często stosuje się różnego rodzaju konwertery kodów, które służą do przekształcania różnych sposobów kodowania używanych w systemie. Przykładem prostych konwerterów kodów są kodery i dekodery. Kodowanie unarne jest przekształcane na kod dwójkowy przez koder, podczas gdy dekodery wykonuje odwrotną operację, czyli zamienia kod dwójkowy na unarny. Kodery i dekodery są zazwyczaj realizowane jako bloki kombinacyjne. W przypadku bardziej skomplikowanych konwersji kodów, takich jak te stosowane w protokołach transmisji danych, używa się bardziej zaawansowanych układów synchronicznych.

5.1. Bufory trójstanowe

Bufory trójstanowe same w sobie nie są blokami kombinacyjnymi. Zazwyczaj są one wykorzystywane w projektach na wyjściach FPGA w celu umożliwienia odłączenia od zewnętrznej szyny. W przypadku wykorzystania w tym celu są one zazwyczaj zrealizowane w blokach wejścia-wyjścia układu FPGA. Bufory trójstanowe mogą również być umieszczone wewnątrz układu FPGA, gdzie są realizowane za pomocą wewnętrznej logiki. Niektóre układy FPGA mają w swojej strukturze wewnętrzne magistrale z buforami trójstanowymi.

Trzeci stan, również znany jako stan wysokiej impedancji, jest oznaczany w języku Verilog symbolem **Z**. Praktycznie oznacza on odłączenie wyjścia od magistrali,

co powoduje, że wysoka impedancja wyjścia nie ma wpływu na podłączone do niego komponenty.

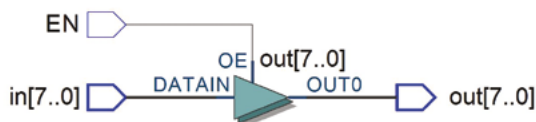
Bufory trójstanowe w języku Verilog można opisywać za pomocą prymitywów, instrukcji przypisania ciągłego **assign** oraz instrukcji proceduralnych. Poniżej przedstawiono przykład opisu sparametryzowanego bufora trójstanowego z wykorzystaniem instrukcji przypisania ciągłego **assign**:

```
module buf_tri_N
    # (parametr N=8)
    (input EN,
     input [N-1:0] in,
     output [N-1:0] out);

    assign out = EN ? in : {N {1'bz}};

endmodule
```

Wynik syntezy pokazany został na rysunku 5.1.



RYS. 5.1. Wynik syntezy bufora trójstanowego

Powyższy wynik dokładnie odpowiada naszym oczekiwaniom. Przytoczony przykład pokazuje, że czasami znacznie prostszym podejściem jest opisanie wybranego bloku funkcjonalnego niż korzystanie z prymitywów lub szablonów języka Verilog.

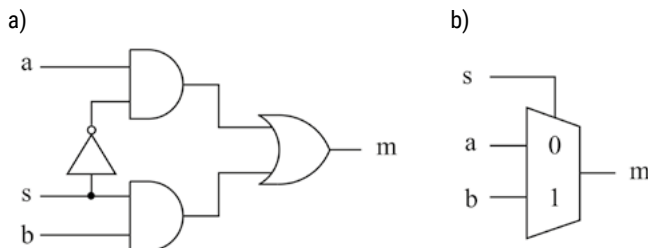
Chcielibyśmy teraz zaproponować czytelnikowi opisanie N -bitowego bufora trójstanowego za pomocą prymitywów oraz instrukcji proceduralnych i porównanie otrzymanych wyników z projektem *buf_tri_N*.

5.2. Multiplexery

Multiplexer (multiplexer) jest układem cyfrowym, który przeprowadza multipleksację danych, czyli ich przekierowywanie z kilku źródeł do jednego wyjścia. Wejścia multiplexera są podłączane do wyjścia w krótkich interwałach czasowych poprzez cykliczną zmianę kodu na wejściach sterujących układu. Ten proces nazywamy *multipleksacją czasową danych*. Multiplexer pełni również funkcję selektora, pozwalając na wybór źródła danych przez podanie odpowiedniego kodu na wejściach sterujących. Jest to układ komutacyjny, który może być wykorzystany do przełączania sygnałów, dlatego czasami nazywany jest komutatorem lub przełącznikiem.

Istnieją dwa podstawowe sposoby opisu projektów w języku Verilog: strukturalny i behawioralny. Opis strukturalny korzysta z prymitywów i instancji modułów niższego poziomu. Podejście behawioralne pozwala opisywać algorytm funkcjonowania multipleksera za pomocą instrukcji języka Verilog, takich jak instrukcja przypisania ciągłego **assign** lub instrukcje proceduralne **if** albo **case**.

Najprostszy multiplekser z dwoma wejściami danych *a* i *b*, wejściem sterującym *s* (nazywanym też *wejściem adresowym*) oraz wyjściem *m* przedstawiony został na rysunku 5.2. Taki multiplekser nazywany jest *multiplekserem 2-kanalowym* (lub *2-wejściowym*) o *1-bitowej szerokości danych*. Multiplekser z takimi parametrami będziemy oznaczać jako *MUX_2_1*.



RYŚ. 5.2. Multiplekser 2-kanalowy: a) schemat na poziomie bramek; b) symbol graficzny

5.2.1. Sposoby opisu multiplekserów

Na przykładzie projektu multipleksera *MUX_2_1* zaprezentujemy różnorodność sposobów opisu układów kombinacyjnych, jakie udostępnia język Verilog. Przy omówieniu innych bloków funkcjonalnych będziemy przytaczać tylko jeden z możliwych sposobów opisu, wiedząc, że może ich być znacznie więcej.

Opis strukturalny multipleksera *MUX_2_1* z wykorzystaniem prymitywów języka Verilog może być przedstawiony następująco:

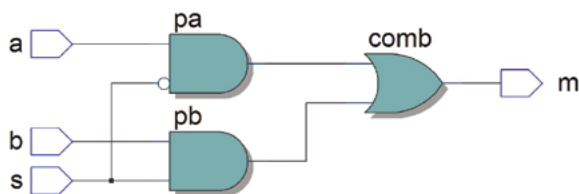
```
module MUX_2_1_struct(
    input a,b,
    input s,
    output m);

    wire pa,pb,ps;

    not (ps,s);
    and (pa,a,ps);
    and (pb,s,b);
    or (m,pa,pb);

endmodule
```

W powyższym kodzie wykorzystane zostały prymitywy bramkowe języka Verilog **not**, **and** i **or**. Wynik syntezy multipleksera *MUX_2_1_struct* przedstawiono rysunku 5.3.



RYS. 5.3. Schemat multipleksera *MUX_2_1_struct*

Funkcjonowanie multipleksera *MUX_2_1* można opisać wyrażeniem w instrukcji ciągłego przypisania **assign**:

```
module MUX_2_1_assign(
    input a,b,
    input s,
    output m);

    assign m = (~s & a) | (s & b);
```

endmodule

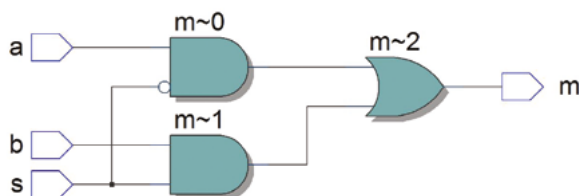
Innym sposobem jest wykorzystanie operacji warunkowej:

```
module MUX_2_1_assign_2(
    input a,b,
    input s,
    output m);

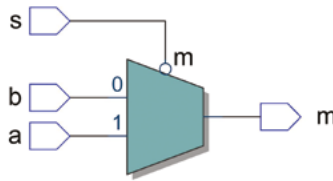
    assign m = (~s) ? a : b;
```

endmodule

Wyniki syntezy na podstawie przytoczonych opisów zaprezentowano na rysunkach 5.4 i 5.5.



RYS. 5.4. Wynik syntezy multipleksera *MUX_2_1_assign* przy opisie za pomocą wyrażenia



RYS. 5.5. Wynik syntezy multipleksera *MUX_2_1_assign_2* przy opisie za pomocą operacji warunkowej

Do opisu multiplexerów bardzo dobrze się nadaje instrukcja proceduralna **case**:

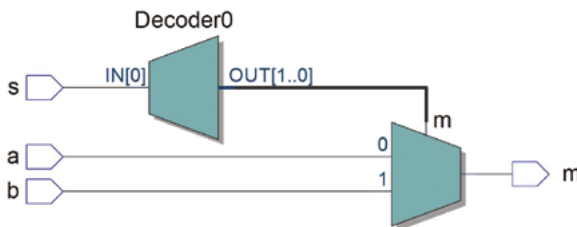
```

module MUX_2_1_case(
    input a,b,
    input s,
    output reg m);

    always @(*)
        case (s)
            1'b0: m = a;
            1'b1: m = b;
        endcase
endmodule

```

Należy zwrócić uwagę, że przy wykorzystaniu instrukcji proceduralnych wyjścia układu kombinacyjnego powinny być zadeklarowane jako zmienne typu **reg**. Wynik syntezy modułu *MUX_2_1_case* możemy zobaczyć na rysunku 5.6.



RYS. 5.6. Wynik syntezy multipleksera *MUX_2_1* przy opisie za pomocą instrukcji **case**

Opis multipleksera *MUX_2_1* z wykorzystaniem instrukcji **if** przedstawia się następująco:

```

module MUX_2_1_if(
    input a,b,
    input s,
    output reg m);

```

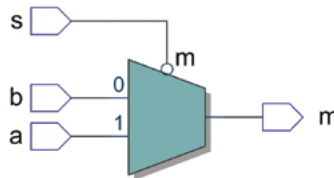
```

always @(*)
    if (!s)      m = a;
    else        m = b;

```

endmodule

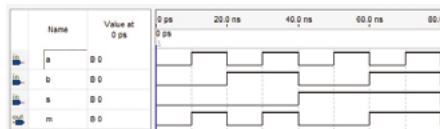
Wynik syntezy jest pokazany na rysunku 5.7.



RYŚ. 5.7. Wynik syntezy multiplexera *MUX_2_1* przy opisie za pomocą instrukcji **if**

Można zauważyć, że wyniki syntezy multiplexera *MUX_2_1* w przypadku opisu strukturalnego oraz opisu za pomocą wyrażenia w instrukcji **assign** są identyczne. Podobnie wyniki syntezy pokrywają się również w przypadku wykorzystania operacji warunkowej w instrukcji **assign** oraz instrukcji proceduralnej **if**.

Bez względu na sposób opisu funkcjonowanie wszystkich zaprojektowanych schematów multiplexera *MUX_2_1* jest tożsame (rysunek 5.8).



RYŚ. 5.8. Wynik symulacji funkcjonalnej multiplexera *MUX_2_1*

5.2.2. Multiplexery grupowe

W projektach systemów wbudowanych multiplexery mogą być wykorzystywane do multipleksowania szyn danych. W takim przypadku wejścia i wyjścia multiplexera będą sygnałami wektorowymi (grupowymi), czyli szynami danych. Multiplexer grupowy z dwoma wejściami wektorowymi o szerokości N bitów oznaczmy jako *MUX_2_1_N_bits*. Jego opis w języku Verilog można przedstawić następująco:

```

module MUX_2_1_N_bits
    #(parameter N=4)
    (input [N-1:0] A,B,
    input s,
    output reg [N-1:0] M);

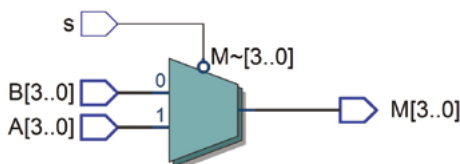
```

```

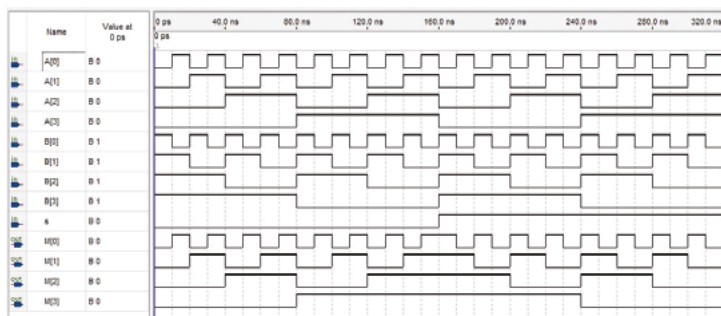
always @(*)
    if (!s) M = A;
    else M = B;
endmodule

```

Wyniki syntezy oraz symulacji modułu *MUX_2_1_N_bits* dla domyślnej wartości parametru *N=4* zaprezentowane zostały na rysunkach 5.9 oraz 5.10.



RYŚ. 5.9. Wynik syntezy modułu *MUX_2_1_N_bits*



RYŚ. 5.10. Wyniki symulacji modułu *MUX_2_1_N_bits*

5.2.3. Zwiększenie liczby wejść multipleksera

Przy projektowaniu systemów wbudowanych możemy potrzebować multiplekserów o większej liczbie wejść – 4, 8 lub więcej (liczba wejść informacyjnych multipleksa rośnie wykładniczo). Zwykle są one budowane przez kaskadowe połączenie mniejszych multiplekserów. Na przykład 4-kanalowy multiplekser może być zbudowany z trzech kaskadowo połączonych multiplekserów 2-kanalowych (rysunek 5.11). Otrzymany multiplekser oznaczmy jako *MUX_4_1*.

Opis strukturalny 4-kanalowego multipleksa w języku Verilog przedstawiony został w poniższym fragmencie kodu:

```

module MUX_4_1_struct(
    input a,b,c,d,
    input [1:0] S,
    output m);

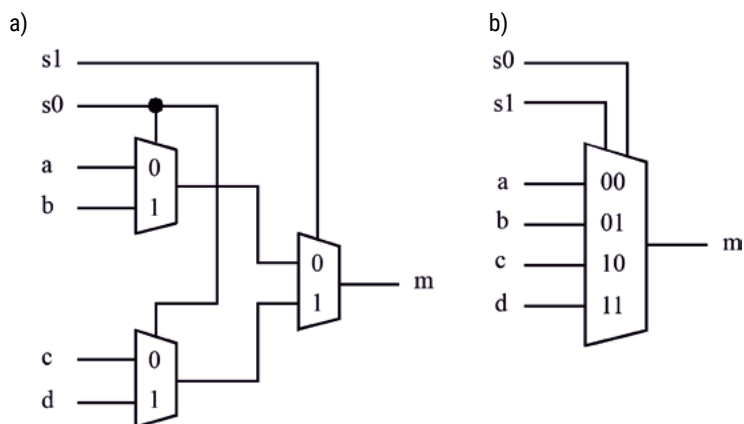
```



```
wire p1,p2;
```

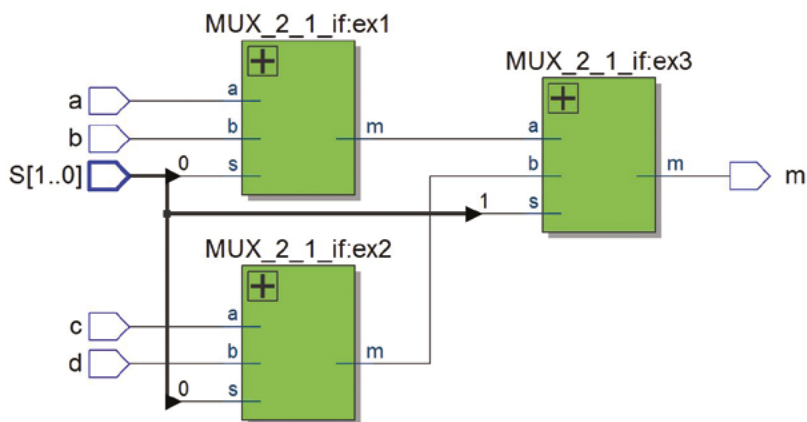
```
MUX_2_1_if ex1(a,b,S[0],p1);
MUX_2_1_if ex2(c,d,S[0],p2);
MUX_2_1_if ex3(p1,p2,S[1],m);
```

```
endmodule
```



RYS. 5.11. Multiplexer 4-kanalowy: a) struktura; b) symbol graficzny

Schemat 4-kanalowego multiplexera otrzymany w wyniku syntezy możemy zaobserwować na rysunku 5.12.



RYS. 5.12. Wynik syntezy modułu *MUX_4_1_struct*

Do opisu multiplexerów o dużej liczbie wejść bardzo dobrze nadaje się instrukcja **case**, na przykład 4-kanalowy multiplexer może być opisany następująco:

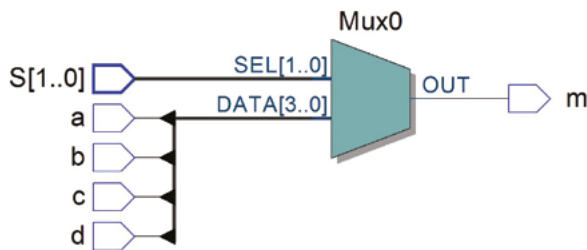
```

module MUX_4_1_case(
    input a,b,c,d,
    input [1:0] S,
    output reg m);

    always @(*)
        case (S)
            2'b00: m = a;
            2'b01: m = b;
            2'b10: m = c;
            2'b11: m = d;
        endcase
endmodule

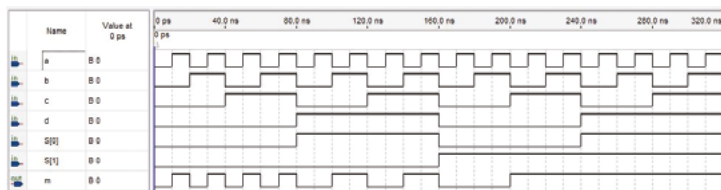
```

W wyniku syntezy wygenerowany zostanie schemat pokazany na rysunku 5.13.



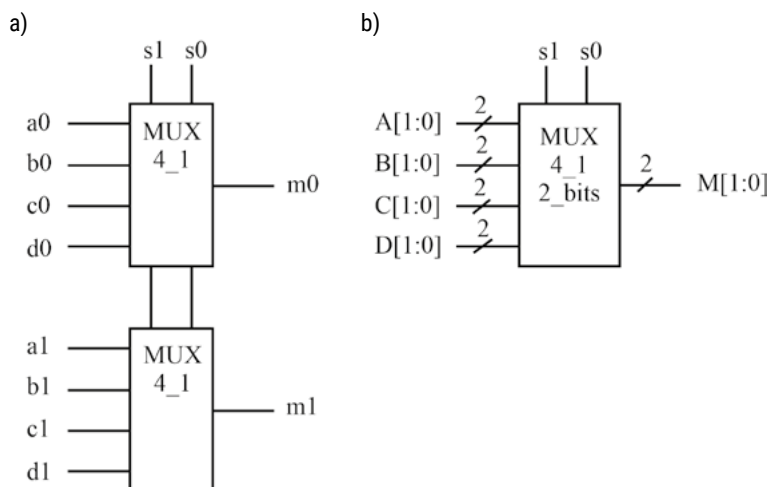
RYS. 5.13. Wynik syntezy modułu *MUX_4_1_case*

Przeprowadzenie symulacji obu przytoczonych przykładów opisu multiplexera *MUX_4_1* daje identyczne wyniki (rysunek 5.14).



RYS. 5.14. Wyniki symulacji multiplexera *MUX_4_1*

Aby zwiększyć szerokość danych 4-kanalowego multipleksera, można skonstruować strukturę z równoległe połączonymi multiplekserami, które są sterowane wspólnymi sygnałami adresowymi (rysunek 5.15).



RYS. 5.15. Grupowy multiplekser *MUX_4_1_2_bits* o 2-bitowej szerokości danych: a) struktura; b) symbol graficzny

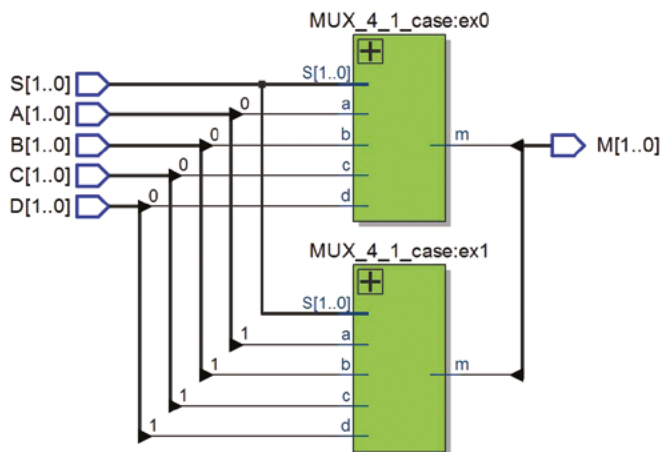
Otrzymany multiplekser grupowy oznaczymy jako *MUX_4_1_2_bits*, a jego opis w języku Verilog można przedstawić następująco:

```
module MUX_4_1_2_bits(
    input [1:0] A,B,C,D,
    input [1:0] S,
    output [1:0] M);

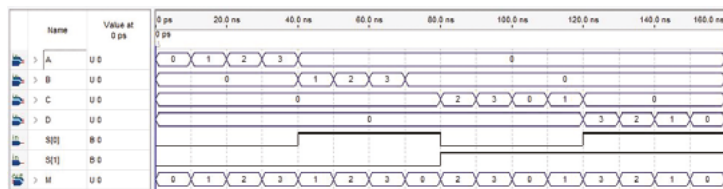
    MUX_4_1_case ex0(A[0],B[0],C[0],D[0],S,M[0]);
    MUX_4_1_case ex1(A[1],B[1],C[1],D[1],S,M[1]);

endmodule
```

Schemat modułu *MUX_4_1_2_bits* otrzymany w wyniku syntezy pokazany jest na rysunku 5.16, a wyniki symulacji projektu – na rysunku 5.17.

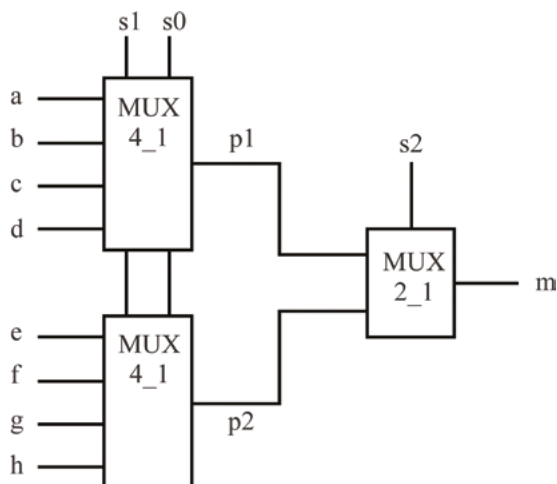


RYS. 5.16. Wynik syntezy modułu *MUX_4_1_2_bits*



RYS. 5.17. Wyniki symulacji modułu *MUX_4_1_2_bits*

Analogicznie można zaprojektować multipleksery o większej liczbie wejść. Na przykład 8-kanalowy multipleksery można uzyskać przez kaskadowe połączenie dwóch multipleksów 4-kanalowych i jednego 2-kanalowego (rysunek 5.18).



RYS. 5.18. Struktura multipleksa *MUX_8_1*

Multiplexer 8-kanałowy *MUX_8_1* z rysunku 5.18 opiszemy w języku Verilog na poziomie strukturalnym:

```

module MUX_8_1_struct(
    input a,b,c,d,e,f,g,h,
    input [2:0] S,
    output m);

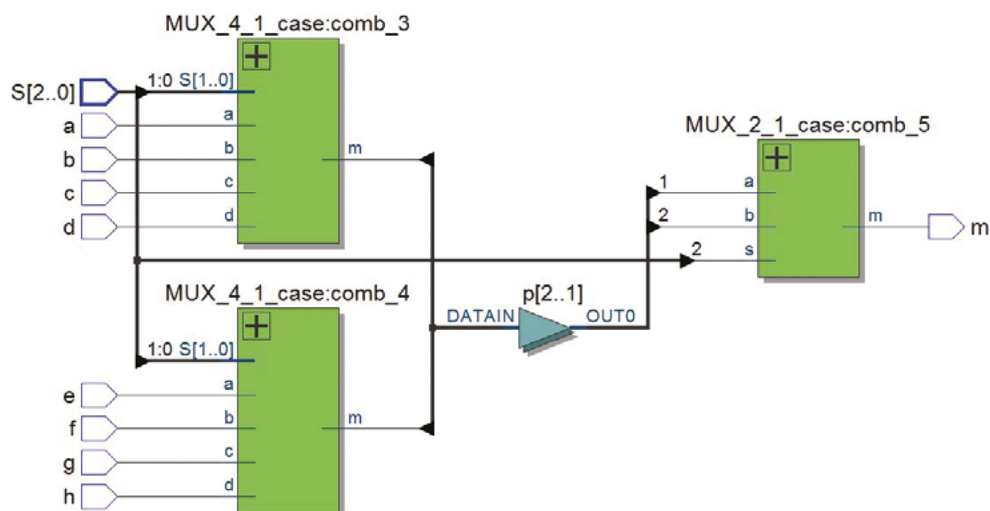
    (* keep *) wire p1,p2;

    MUX_4_1_case (a,b,c,d,S[1:0],p1);
    MUX_4_1_case (e,f,g,h,S[1:0],p2);
    MUX_2_1_case (p1,p2,S[2],m);

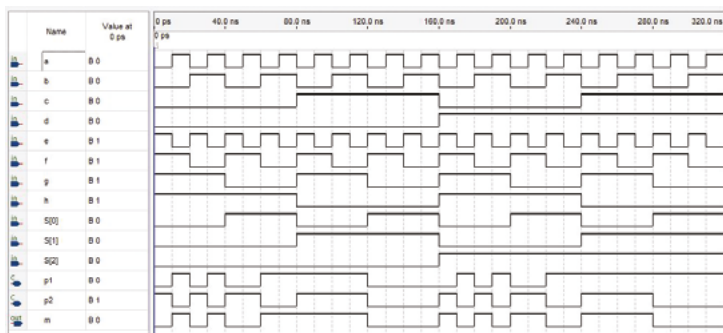
endmodule

```

W powyższym kodzie zmienne pomocnicze *p1* i *p2* zostały zadeklarowane z wykorzystaniem dyrektywy kompilatora *keep*, która zapobiega ich usunięciu w procesie optymalizacji układu. Zastosowaliśmy ten sposób deklaracji, aby móc monitorować wartości zmiennych *p1* i *p2* podczas symulacji projektu. Schemat modułu multiplexera *MUX_8_1_struct* został przedstawiony na rysunku 5.19, a wyniki symulacji można zaobserwować na rysunku 5.20.



RYS. 5.19. Wynik syntezy modułu *MUX_8_1_struct*



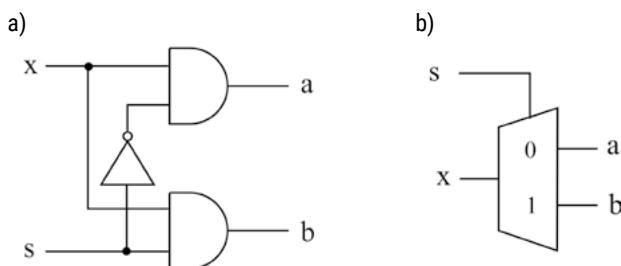
RYS. 5.20. Wyniki symulacji modułu *MUX_8_1_struct*

Proponujemy, aby czytelnik za pomocą systemu projektowania zbadał różne sposoby opisu multiplekserów i określił, który z nich jest najbardziej efektywny w przypadku realizacji w docelowym układzie FPGA.

5.3. Demultipleksery

Demultiplekser (demultiplexer) wykonuje funkcję odwrotną do multipleksera, przesyłając dane z pojedynczego wejścia na jedno spośród wielu wyjść. Numer wybranego kanału wyjściowego określany jest kodem podanym na wejścia adresowe demultipleksera. Kanały wyjściowe mogą być cyklicznie łączone z wejściem przez krótki okres czasu, co umożliwia przesyłanie informacji wejściowej przez różne linie, wybierane przez cykliczną zmianę kodu na wejściach adresowych. Układ *komutatora* w systemach transmisji danych może być zbudowany przez sekwencyjne połączenie multipleksera i demultipleksera. Wejścia adresowe multipleksera przy tym służą do wyboru źródła przesyłania informacji, natomiast wejścia adresowe demultipleksera wybierają docelowy odbiornik. Takie funkcje mogą być przydatne także w systemach wbudowanych.

Najprostszy schemat demultipleksera z dwoma wyjściami przedstawiony został na rysunku 5.21. Taki układ nazywany jest demultiplekserem *2-kanałowym* (lub *2-wyjściowym*) z *1-bitową szerokością danych*. Będziemy go oznaczać jako *DMUX_1_2*.



RYS. 5.21. Demultiplekser 2-kanałowy: a) schemat na poziomie bramek; b) symbol graficzny

Opis strukturalny demultipleksera *DMUX_1_2* z wykorzystaniem prymitywów języka Verilog można przedstawić następująco:

```

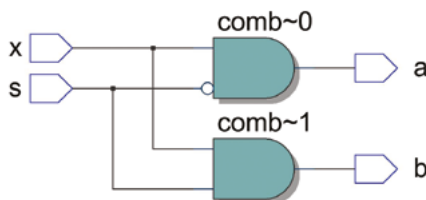
module DMUX_1_2_struct(
    input x,
    input s,
    output a,b);

    and (a,x,~s);
    and (b,x, s);

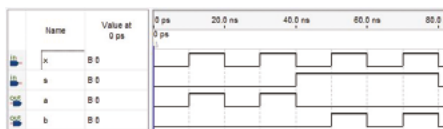
endmodule

```

Wynik syntezy demultipleksera *DMUX_1_2* na podstawie podanego opisu zaprezentowano na rysunku 5.22, a przebiegi symulacji – na rysunku 5.23.



RYS. 5.22. Wynik syntezy demultipleksera *DMUX_1_2*



RYS. 5.23. Wyniki symulacji demultipleksera *DMUX_1_2*

Podobnie jak w przypadku multiplexerów, zwiększenie liczby kanałów demultipleksera można osiągnąć przez kaskadowe połączenie demultiplekserów o mniejszej liczbie wyjść (rysunek 5.24).

Opis strukturalny 4-kanałowego demultipleksera z rysunku 5.24 w języku Verilog prezentuje się następująco:

```

module DMUX_1_4_struct(
    input x,
    input [1:0] S,
    output a,b,c,d);

    wire p1,p2;

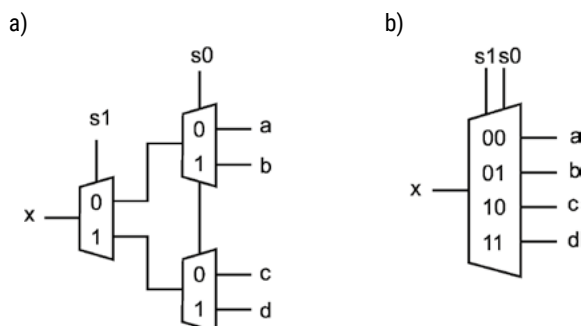
```

```

DMUX_1_2_struct (x,S[1],p1,p2);
DMUX_1_2_struct (p1,S[0],a,b);
DMUX_1_2_struct (p2,S[0],c,d);

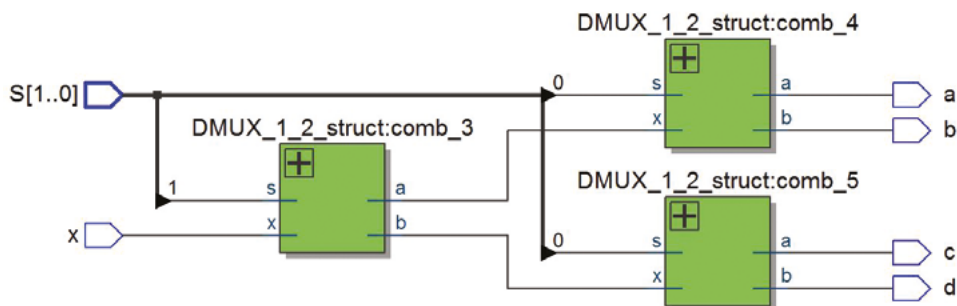
```

endmodule



RYS. 5.24. Demultiplexer 4-kanalowy zbudowany z trzech demultiplexerów 2-kanalowych: a) schemat; b) symbol graficzny

Wyniki syntezy demultiplexera 4-kanalowego *DMUX_1_4_struct* możemy zobaczyć na rysunku 5.25.



RYS. 5.25. Schemat modułu *DMUX_1_4_struct*

Na poniższym przykładzie przedstawiono opis modułu demultiplexera *DMUX_1_4* z wykorzystaniem instrukcji przypisania ciągłego **assign**:

```

module DMUX_1_4_assign(
    input x,
    input [1:0] S,
    output a,b,c,d);

```



```

assign a = ~S[0] & ~S[1] & x;
assign b = S[0] & ~S[1] & x;
assign c = ~S[0] & S[1] & x;
assign d = S[0] & S[1] & x;

```

endmodule

Innym sposobem opisu jest wykorzystanie instrukcji proceduralnej **case**:

```

module DMUX_1_4_case(
    input x,
    input [1:0] S,
    output reg a,b,c,d);

    always @(*) begin
        a = 1'b0; b = 1'b0; c = 1'b0; d = 1'b0;
        case (S)
            2'b00: a = x;
            2'b01: b = x;
            2'b10: c = x;
            2'b11: d = x;
        endcase
    end
endmodule

```

Kolejny przykład ilustruje sposób opisu 4-kanalowego demultipleksera *DMUX_1_4* za pomocą instrukcji proceduralnej **if**:

```

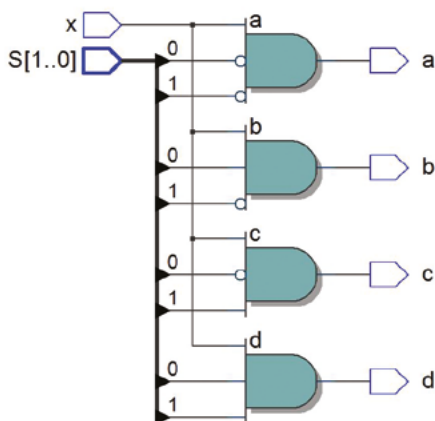
module DMUX_1_4_if(
    input x,
    input [1:0] S,
    output reg a,b,c,d);

    always @(*) begin
        a=0; b=0; c=0; d=0;
        if (!S[1] & !S[0]) a = x;
        else if (!S[1] & S[0]) b = x;
        else if ( S[1] & !S[0]) c = x;
        else d = x;
    end
endmodule

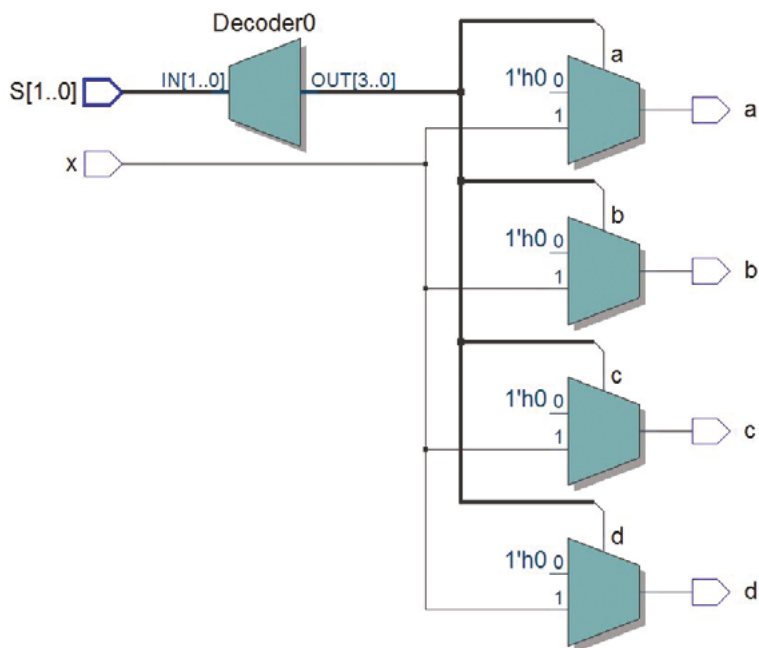
```

Należy zauważyć, że w kodzie zaprezentowanych modułów –z wykorzystaniem instrukcji proceduralnych **case** i **if** – określone zostały domyślne wartości sygnałów wyjściowych *a*, *b*, *c* i *d*. Dzięki temu zapobiega się automatycznemu ustawieniu przez kompilator zatrząsków na wyjściach demultipleksera.

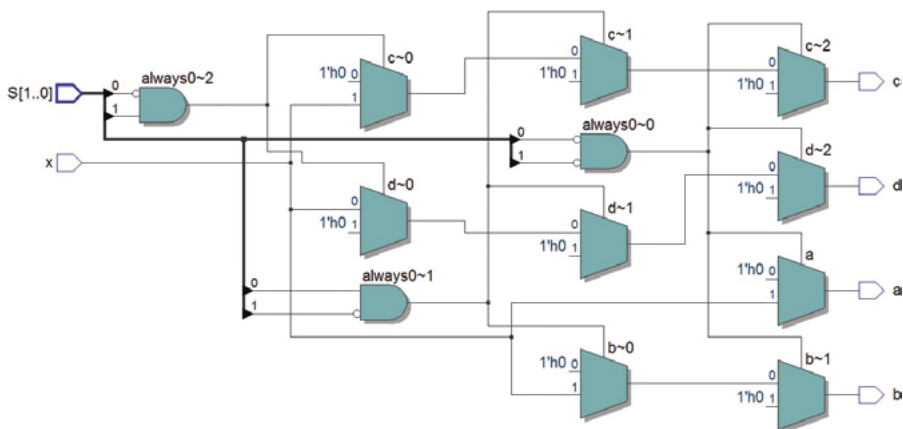
Wyniki syntezy modułów *DMUX_1_4_assign*, *DMUX_1_4_case* i *DMUX_1_4_if* przedstawiono odpowiednio na rysunkach 5.26, 5.27 i 5.28.



RYS. 5.26. Schemat demultipleksa *DMUX_1_4* przy wykorzystaniu instrukcji **assign**

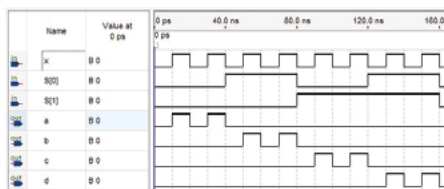


RYS. 5.27. Schemat demultipleksa *DMUX_1_4* przy wykorzystaniu instrukcji **case**



RYS. 5.28. Schemat demultipleksera *DMUX_1_4* przy wykorzystaniu instrukcji **if**

Dla wszystkich przytoczonych opisów demultipleksera *DMUX_1_4* otrzymane zostały takie same wyniki symulacji (rysunek 5.29).



RYS. 5.29. Wyniki symulacji demultipleksera *DMUX_1_4*

Demultiplexer grupowy można skonstruować z demultiplexerów jednobitowych, na które podawane są wspólne sygnały adresowe, tak jak to zostało zobrazone na rysunku 5.30.

W strukturze demultipleksera grupowego na rysunku 5.30 każdy ze składowych demultiplexerów obsługuje przesyłanie jednego z bitów danych. Opis 4-kanalowego demultipleksera z N -bitową szerokością wektora danych w języku Verilog może być przedstawiony w następujący sposób:

```

module DMUX_1_4_N_bits
    #(parameter N=2)
    (input [N-1:0] X,
     input [1:0] S,
     output reg [N-1:0] A,B,C,D);

    always @(*) begin
        A=0; B=0; C=0; D=0;
        case (S)

```

```

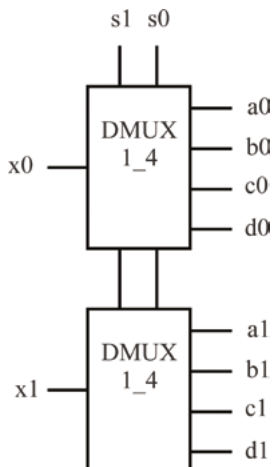
2'b00: A = X;
2'b01: B = X;
2'b10: C = X;
2'b11: D = X;

endcase

end

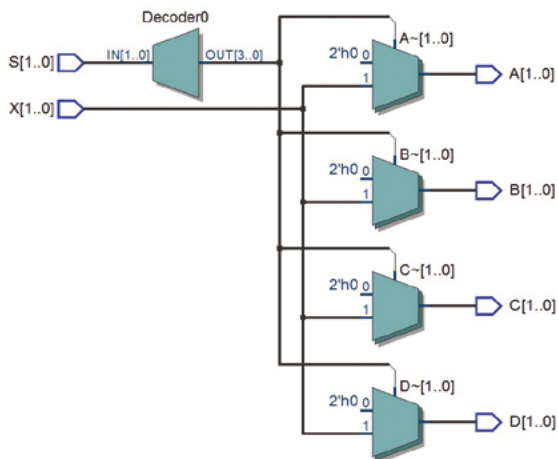
endmodule

```

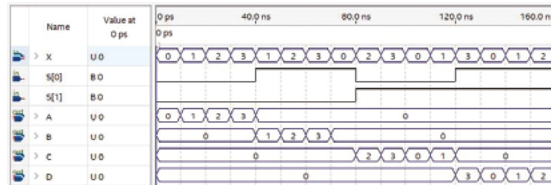


RYS. 5.30. 4-kanałowy demultiplexer grupowy z 2-bitową szerokością danych zbudowany z dwóch demultiplexerów jednobitowych

W naszym przykładzie szerokość wektora danych N przyjmuje domyślną wartość równą 2. Wynik syntezy modułu *DMUX_1_4_N_bits* można zobaczyć na rysunku 5.31, a wyniki modulacji projektu – na rysunku 5.32.



RYS. 5.31. Wynik syntezy modułu *DMUX_1_4_N_bits*



RYS. 5.32. Wyniki symulacji modułu *DMUX_1_4_N_bits*

Przy większej liczbie wyjść demultipleksera wygodnie jest opisać je w postaci wektora i odwoływać się do wyjść za pomocą indeksów. W poniższym przykładzie kodu wyjścia 8-kanalowego demultipleksera są adresowane za pośrednictwem indeksów – $D[0]$, ..., $D[7]$.

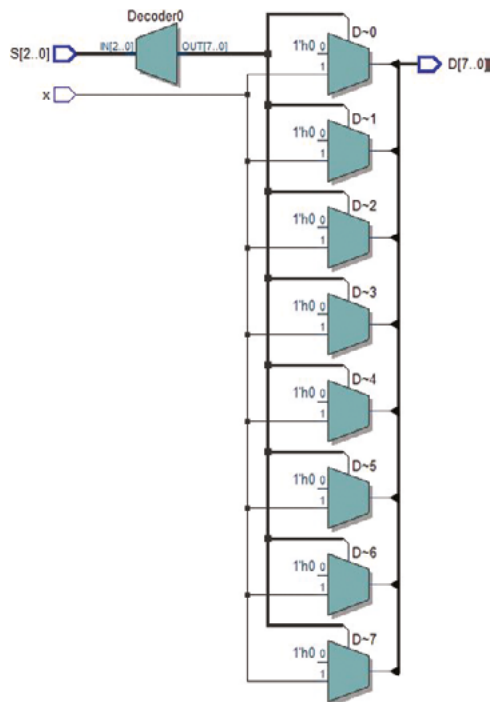
```

module DMUX_1_8_case(
    input x,
    input [2:0] S,
    output reg [7:0] D);

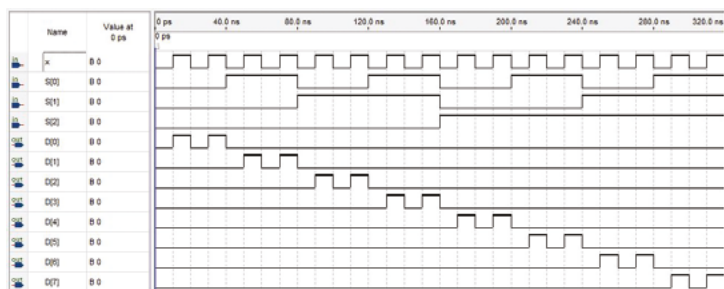
    always @(*) begin
        D = 0;
        case (S)
            3'b000: D[0] = x;
            3'b001: D[1] = x;
            3'b010: D[2] = x;
            3'b011: D[3] = x;
            3'b100: D[4] = x;
            3'b101: D[5] = x;
            3'b110: D[6] = x;
            3'b111: D[7] = x;
        endcase
    end
endmodule

```

Schemat otrzymany w wyniku syntezy modułu *DMUX_1_8_case* został pokazany na rysunku 5.33, a wyniki jego symulacji – na rysunku 5.34.



RYS. 5.33. Wynik syntezy modułu *DMUX_1_8_case*



RYS. 5.34. Wyniki symulacji modułu *DMUX_1_8_case*

5.4. Kodery i kodery priorytetowe

5.4.1. Kodery

Koder (*encoder*) jest układem cyfrowym, który przekształca *kod unarny* (*one-hot code*) na odpowiedni *kod binarny* (*binary code*). Kod unarny to sekwencja bitów, w której tylko jeden bit ma wartość 1, a pozostałe są równe 0. W kodowaniu unarnym liczba bitów w słowie kodowym odpowiada liczbie kodowanych elementów. Przy podaniu

kodu unarnego na wejścia kodera jedno z wejść zostanie wyróżnione (aktywowane), a na wyjściach układu wygenerowany zostanie binarny kod numeru aktywnego wejścia. W prostym koderze tylko jedno z wejść może być aktywne, a dodatkowo nie wolno podawać wszystkich zer na wejścia układu.

Kodery w systemach cyfrowych służą do generowania binarnego kodu, który aktywuje sygnał lub urządzenie w danym momencie. Na przykład często są wykorzystywane do odczytywania wartości cyfr dziesiętnych z klawiatury matrycowej, przekształcając wartości cyfr na format binarny, który jest bardziej praktyczny do przechowywania.

Tablica prawdy, która opisuje funkcjonowanie kodera 8:3, przedstawiona została w tabeli 5.1.

TABELA 5.1. Tablica prawdy kodera 8:3

Wejścia								Wyjścia		
a7	a6	a5	a4	a3	a2	a1	a0	b2	b1	b0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Wyrażenia logiczne opisujące funkcje wyjściowe kodera są bardzo złożone i nie poddają się minimalizacji. Dlatego instrukcja **assign** nie jest zazwyczaj odpowiednia do opisu koderów w języku Verilog. Opis strukturalny kodera przy użyciu prymitywów języka Verilog również byłby bardzo pracochłonny. Dlatego w praktyce kodery zazwyczaj opisuje się za pomocą instrukcji proceduralnych **if** lub **case**.

Najpierw przedstawimy przykłady opisu koderów przy użyciu instrukcji **case**. Poniższy kod można bezpośrednio uzyskać z tablicy prawdy kodera:

```

module encoder_8_3_case(
    input [7:0] a,
    output reg [2:0] b);

    always @(*)
        case(a)
            8'b00000001: b = 0;
            8'b00000010: b = 1;
            8'b00000100: b = 2;
        endcase

```

```

8'b00001000: b = 3;
8'b00010000: b = 4;
8'b00100000: b = 5;
8'b01000000: b = 6;
8'b10000000: b = 7;
default: b = 3'bxxx;

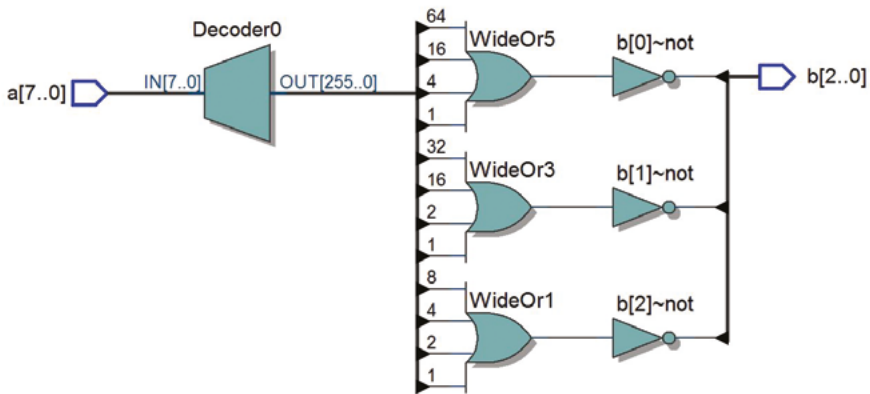
```

endcase

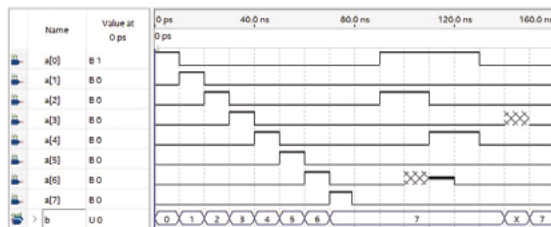
endmodule

W zaprezentowanym opisie zakładamy, że każdemu kodowi unarnemu zostanie przyporządkowany odpowiedni kod dwójkowy na wyjściach, natomiast dla wszystkich pozostałych kombinacji na wejściach kodera wartości wyjściowe są nieokreślone.

Wyniki syntezy i symulacji modułu *encoder_8_3_case* przedstawione zostały na rysunkach 5.35 i 5.36.



RYS. 5.35. Wynik syntezy modułu *encoder_8_3_case*



RYS. 5.36. Wyniki symulacji modułu *encoder_8_3_case*

Analiza wyników symulacji pokazuje, że schemat kodera działa poprawnie tylko w przypadku, gdy na wejścia kodera podawany jest kod unarny, czyli wektor wejściowy zawiera tylko jedną jedynkę. Natomiast jeśli na wejściu pojawi się wektor

zerowy lub kombinacja zawierająca kilka jedynek, koder błędnie generuje na wyjściu kod 3'b111, co odpowiada sytuacji, gdy na wejściu *a*[7] pojawi się jedynka. Nieokreślony stan na wyjściach kodera zostaje ustawiony jedynie w przypadku pojawienia się na wejściach stanu nieokreślonego **X** lub stanu wysokiej impedancji **Z**. Funkcjonowanie zaprojektowanego modułu *encoder_8_3_case* nie spełnia założeń, ponieważ koder reaguje na kombinację zerową oraz kombinacje z kilkoma jedynkami na wejściach tak samo, jak na pojawienie się jedynki na wejściu *a*[7].

Spróbujemy wyjaśnić, dlaczego otrzymaliśmy taki wynik syntezy. Okazuje się, że w sytuacji, gdy wyjściom przypisane są stany nieokreślone, syntezyator traktuje je jako nieistotne i zastępuje stany nieokreślone wyjść dowolnie wybranymi wartościami. Zazwyczaj te wartości są tak dobrane, aby jak najlepiej zminimalizować funkcje logiczne i uprościć schemat. W przypadku naszego schematu syntezyator zastąpił nieokreślone wartości jedynkami na wszystkich wyjściach.

Aby usprawnić projekt, dodamy do schematu jedno dodatkowe wyjście, które sygnalizuje (ustawieniem na 1) pojawienie się na wejściach kodera kombinacji innego rodzaju niż kod unary. Poniżej przedstawiono opis kodera z dodatkowym wyjściem informacyjnym *b*[3]:

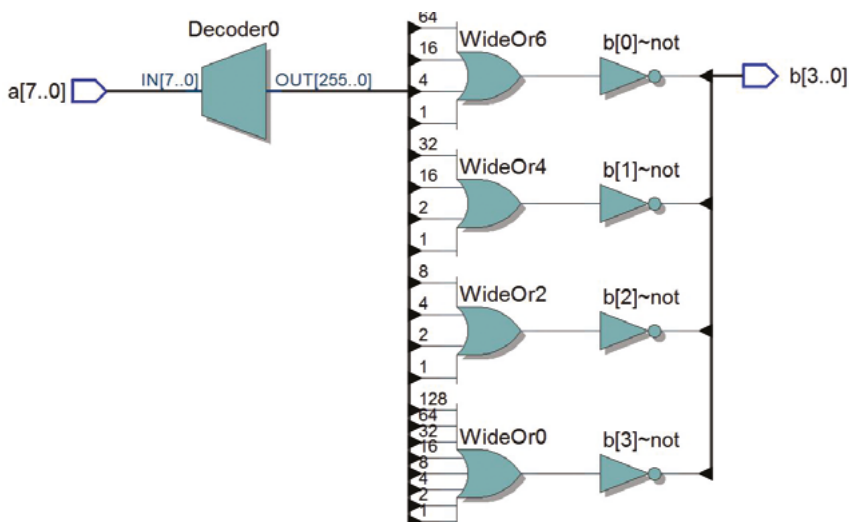
```
module encoder_8_3_case_v2(
    input [7:0] a,
    output reg [3:0] b);

    always @(*)
        case(a)
            8'b00000001: b = 0;
            8'b00000010: b = 1;
            8'b00000100: b = 2;
            8'b00001000: b = 3;
            8'b00010000: b = 4;
            8'b00100000: b = 5;
            8'b01000000: b = 6;
            8'b10000000: b = 7;
            default: b = 4'b1xxx;
        endcase
endmodule
```

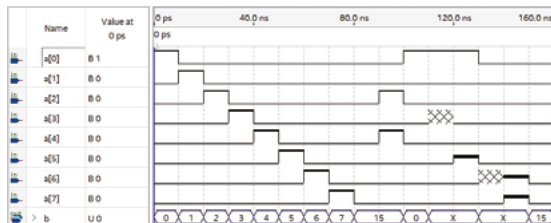
Wyniki syntezy i symulacji modułu *encoder_8_3_case_v2* można zaobserwować na rysunkach 5.37 i 5.38 odpowiednio.

Analizując przebiegi czasowe na rysunku 5.38, można zauważyć, że w przypadku zerowej kombinacji wejściowej lub podania na wejścia więcej niż jednej jedynki, koder generuje na wyjściach wartość 15 (dwójkowo 4'b1111). Natomiast przy pojawieniu się na jednym z wejść wartości **X** lub **Z**, układ ustawia na wyjściach stan

nieokreślony X. Moduł *encoder_8_3_case_v2* teraz już odróżnia zerowy wektor wejściowy (wartość 15 na wyjściu) od kodu unarnego z jedynką na wejściu *a[7]* (wartość 7 na wyjściu).



RYS. 5.37. Wynik syntezy modułu *encoder_8_3_case_v2*



RYS. 5.38. Wyniki symulacji modułu *encoder_8_3_case_v2*

Innym sposobem rozwiązania problemu może być dodanie dodatkowego wyjścia *valid*, które będzie przyjmować wartość jeden tylko w przypadku podania kodu unarnego na wejścia układu:

```
module encoder_8_3_case_valid(
    input [7:0] a,
    output reg [2:0] b,
    output reg valid);

    always @(*)
        case(a)
            8'b00000001: begin b=0; valid=1; end
            8'b00000010: begin b=1; valid=1; end
```

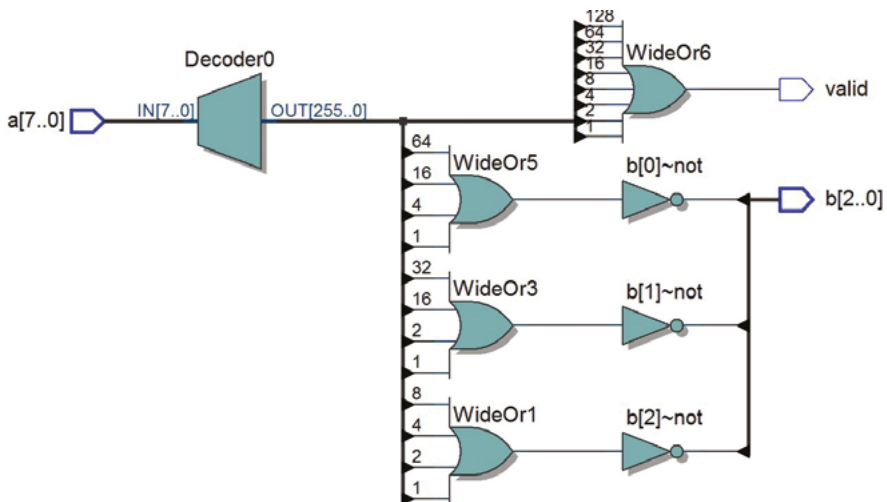
```

8'b00000100: begin b=2; valid=1; end
8'b00001000: begin b=3; valid=1; end
8'b00010000: begin b=4; valid=1; end
8'b00100000: begin b=5; valid=1; end
8'b01000000: begin b=6; valid=1; end
8'b10000000: begin b=7; valid=1; end
default: begin b = 3'bxxx; valid=0; end
endcase

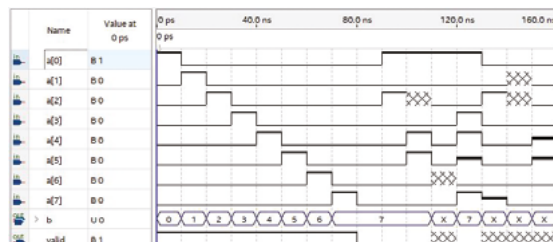
```

endmodule

Wyniki syntezy i symulacji *encoder_8_3_case_valid* przedstawiono na rysunkach 5.39 i 5.40 odpowiednio.



RYS. 5.39. Wynik syntezy modułu *encoder_8_3_case_valid*



RYS. 5.40. Wyniki symulacji modułu *encoder_8_3_case_valid*

Wyniki symulacji przedstawione na rysunku 5.40 pokazują, że wyjście *valid* zostanie ustawione na 1 dla wszystkich kodów unarnych na wejściu. Natomiast przyjmuje ono wartość 0 lub X, jeśli na wejściach pojawi się dowolna inna kombinacja, w tym

też wartości **X** lub **Z**. Moduł kodera *encoder_8_3_case_valid* może być wykorzystany w projektach, gdy wiadomo, że na wejścia układu mogą być podawane nie tylko unarne, lecz także inne kombinacje.

Do opisu kodera można zastosować także instrukcję proceduralną **if**:

```

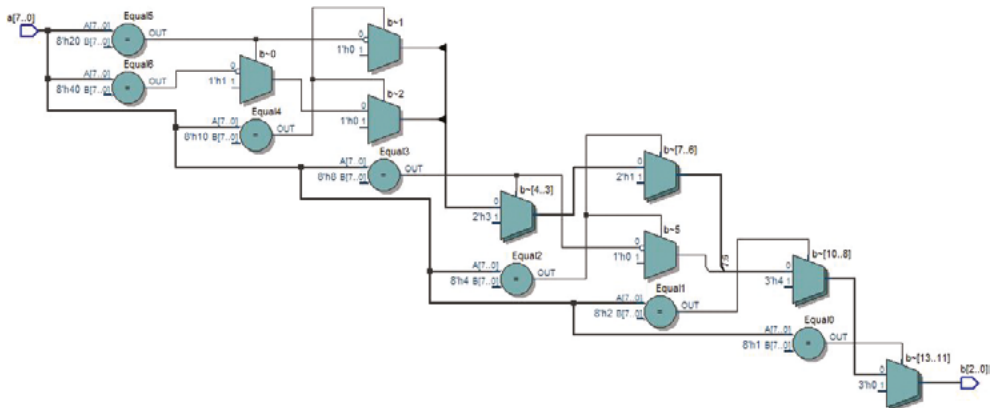
module encoder_8_3_if(
    input [7:0] a,
    output reg [2:0] b);

    always @(*)
        if (a == 8'b00000001) b = 0;
        else if (a == 8'b00000010) b = 1;
        else if (a == 8'b00000100) b = 2;
        else if (a == 8'b00001000) b = 3;
        else if (a == 8'b00010000) b = 4;
        else if (a == 8'b00100000) b = 5;
        else if (a == 8'b01000000) b = 6;
        else if (a == 8'b10000000) b = 7;
        else
            b = 3'bxxx;

endmodule

```

Schemat otrzymany w wyniku syntezy modułu *encoder_8_3_if* możemy zobaczyć na rysunku 5.41, a wyniki symulacji są takie same, jak te pokazane na rysunku 5.36.



RYS. 5.41. Wynik syntezy modułu *encoder_8_3_if*

Projekt kodera *encoder_8_3_if* ma te same niedociągnięcia, które były wykryte w przypadku projektu *encoder_8_3_case*: kombinacja zerowa oraz kilka jedynek w kombinacji wejściowej dają taki sam wynik na wyjściu, jak kombinacja unarna z jedynką na wejściu *a*[7].

Jednym z rozwiązań może być wprowadzenie dodatkowego bitu wyjściowego, który informuje o niedozwolonej kombinacji na wejściu kodera:

```
module encoder_8_3_if_v2(  
    input [7:0] a,  
    output reg [3:0] b);  
  
    always @(*)  
        if (a == 8'b00000001) b = 0;  
        else if (a == 8'b00000010) b = 1;  
        else if (a == 8'b00000100) b = 2;  
        else if (a == 8'b00001000) b = 3;  
        else if (a == 8'b00010000) b = 4;  
        else if (a == 8'b00100000) b = 5;  
        else if (a == 8'b01000000) b = 6;  
        else if (a == 8'b10000000) b = 7;  
        else b = 4'b1xxx;  
  
endmodule
```

Innym sposobem rozwiązania tego problemu jest dodanie sygnału *valid*, który jest ustawiany na 1 tylko w przypadku, gdy na wejściach kodera występują kombinacje unarne:

```
module encoder_8_3_if_valid(  
    input [7:0] a,  
    output reg [2:0] b,  
    output reg valid);  
  
    always @(*)  
        if (a == 8'b00000001) begin b=0; valid=1; end  
        else if (a == 8'b00000010) begin b=1; valid=1; end  
        else if (a == 8'b00000100) begin b=2; valid=1; end  
        else if (a == 8'b00001000) begin b=3; valid=1; end  
        else if (a == 8'b00010000) begin b=4; valid=1; end  
        else if (a == 8'b00100000) begin b=5; valid=1; end  
        else if (a == 8'b01000000) begin b=6; valid=1; end  
        else if (a == 8'b10000000) begin b=7; valid=1; end  
        else begin b = 3'bxxx; valid=0; end  
  
endmodule
```

5.4.2. Kodery priorytetowe

Koder priorytetowy (priority encoder) przyjmuje na wejściach kombinacje zawierające więcej niż jedną jedynkę. Funkcja rozstrzygania priorytetów w koderze aktywuje tylko jedno z wejść, które ma najwyższy priorytet, ignorując jedynki na pozostałych wejściach. Zazwyczaj przyjętym rozwiązaniem jest nadawanie wyższego priorytetu wejściom o wyższych numerach. Kodery priorytetowe są powszechnie stosowane w systemach wbudowanych do realizacji różnych schematów wyboru według priorytetów, na przykład do obsługi przerwań w mikrokontrolerach lub procesorach, systemach arbitrażu przydzielających dostęp do magistrali urządzeniu o najwyższym priorytecie oraz innych zastosowań. Tablica prawdy kodera priorytetowego 8:3 przedstawiona została w tabeli 5.2.

TABELA 5.2. Tablica prawdy kodera priorytetowego 8:3

Wejścia								Wyjścia		
a7	a6	a5	a4	a3	a2	a1	a0	b2	b1	b0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	–	0	0	1
0	0	0	0	0	1	–	–	0	1	0
0	0	0	0	1	–	–	–	0	1	1
0	0	0	1	–	–	–	–	1	0	0
0	0	1	–	–	–	–	–	1	0	1
0	1	–	–	–	–	–	–	1	1	0
1	–	–	–	–	–	–	–	1	1	1

Funkcjonowanie koderów priorytetowych w języku Verilog jest opisywane za pomocą instrukcji proceduralnych **case** lub **if**. Koder priorytetowy 8:3 można zdefiniować na podstawie tablicy prawdy przy użyciu instrukcji **case** w następujący sposób:

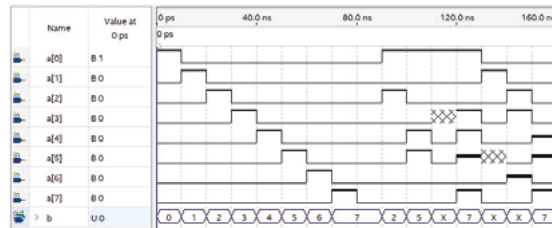
```
module pr_encoder_8_3_case(  
    input [7:0] a,  
    output reg [2:0] b);  
  
    always @(*)  
        casex(a)  
            8'b00000001: b = 0;  
            8'b0000001x: b = 1;  
            8'b0000001xx: b = 2;  
            8'b000001xxx: b = 3;  
            8'b0001xxxx: b = 4;  
            8'b001xxxxx: b = 5;
```

```

        8'b01xxxxxx: b = 6;
        8'b1xxxxxxx: b = 7;
        default: b = 3'bxxx;
    endcase
endmodule

```

Ponieważ elementy stałe instrukcji **case** w opisie kodera priorytetowego zawierają wartości logiczne **X**, zaleca się zamiast instrukcji **case** użyć instrukcji **casex**. Wyniki symulacji modułu *pr_encoder_8_3_case* zostały przedstawione na rysunku 5.42.



RYS. 5.42. Wyniki symulacji modułu *pr_encoder_8_3_case*

Stan nieokreślony **X** na wyjściach kodera priorytetowego pojawia się jedynie w przypadku, gdy wartości **X** lub **Z** znajdują się na wejściu o wyższym priorytecie. Z wykresów na rysunku 5.42 możemy zauważyć, że projekt *pr_encoder_8_3_case* nie działa poprawnie w przypadku zerowej kombinacji na wejściach, ponieważ na wyjściach jest generowana wartość 7. Aby wykryć zerową kombinację, można dodać dodatkowy bit na wyjściu kodera priorytetowego:

```

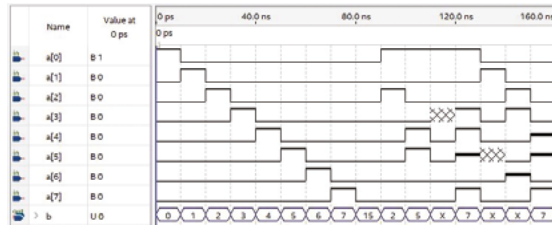
module pr_encoder_8_3_case_v2(
    input [7:0] a,
    output reg [3:0] b);

    always @(*)
        casex(a)
            8'b00000001: b = 0;
            8'b0000001x: b = 1;
            8'b000001xx: b = 2;
            8'b00001xxx: b = 3;
            8'b0001xxxx: b = 4;
            8'b001xxxxx: b = 5;
            8'b01xxxxxx: b = 6;
            8'b1xxxxxxx: b = 7;
            default: b = 4'b1xxx;
        endcase

endmodule

```

Wyniki symulacji modułu *pr_encoder_8_3_case_v2* zaprezentowano na rysunku 5.43.



RYS. 5.43. Wyniki symulacji modułu *pr_encoder_8_3_case_v2*

Teraz kombinacji zerowej odpowiada wartość 15 na wyjściu, czyli bit $b[3]=1$ informuje o tym, że na wejścia podane zostały same zera. W kolejnym przykładzie do wykrywania zerowego wektora na wejściach kodera priorytetowego wykorzystuje się dodatkowe wyjście *valid*:

```

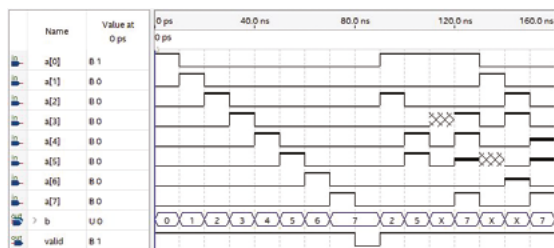
module pr_encoder_8_3_case_valid(
    input [7:0] a,
    output reg [2:0] b,
    output reg valid);

    always @(*)
        casex(a)
            8'b00000001: begin b=0; valid=1; end
            8'b0000001x: begin b=1; valid=1; end
            8'b000001xx: begin b=2; valid=1; end
            8'b00001xxx: begin b=3; valid=1; end
            8'b0001xxxx: begin b=4; valid=1; end
            8'b001xxxxx: begin b=5; valid=1; end
            8'b01xxxxxx: begin b=6; valid=1; end
            8'b1xxxxxxx: begin b=7; valid=1; end
            default: begin b = 3'bxxx; valid=0; end
        endcase

endmodule

```

Wyniki symulacji modułu *pr_encoder_8_3_case_valid* pokazane zostały na rysunku 5.44.



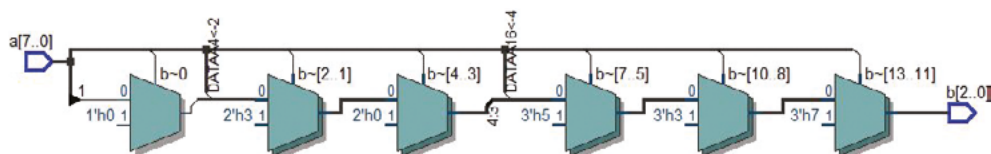
RYS. 5.44. Wyniki symulacji modułu *pr_encoder_8_3_case_valid*

Z wykresów przedstawionych na rysunku 5.44 wynika, że kombinacji zerowej na wejściach odpowiada wartość zerowa na wyjściu sygnału *valid*.

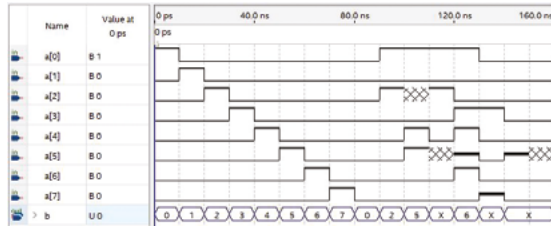
Opis kodera priorytetowego z wykorzystaniem instrukcji **if** można przedstawić następująco:

```
module pr_encoder_8_3_if(  
    input [7:0] a,  
    output reg [2:0] b);  
  
    always @(*)  
        if (a[7]) b = 7;  
        else if (a[6]) b = 6;  
        else if (a[5]) b = 5;  
        else if (a[4]) b = 4;  
        else if (a[3]) b = 3;  
        else if (a[2]) b = 2;  
        else if (a[1]) b = 1;  
        else if (a[0]) b = 0;  
        else b = 3'bxxx;  
  
endmodule
```

W przytoczonym kodzie istotna jest kolejność sprawdzania bitów wektora wejściowego a , sprawdzenie należy rozpocząć od najstarszego bitu. Wyniki syntezy i symulacji modułu *pr_encoder_8_3_if* zostały przedstawione na rysunkach 5.45 i 5.46.



RYS. 5.45. Wynik syntezy modułu *pr_encoder_8_3_if*



RYS. 5.46. Wyniki symulacji modułu *pr_encoder_8_3_if*

Niedokładnością w funkcjonowaniu zaprojektowanego modułu *pr_encoder_8_3_if* jest to, że koder generuje identyczne wyniki dla kombinacji zerowej oraz dla pojedynczej jedynki na wejściu *a[0]*. Aby naprawić ten błąd, w kolejnej wersji projektu do schematu koderu priorytetowego wprowadzono dodatkowe wyjście *valid*:

```

module pr_encoder_8_3_if_valid(
    input [7:0] a,
    output reg [2:0] b,
    output reg valid);

    always @(*)
        if (a[7]) begin b=7; valid=1; end
        else if (a[6]) begin b=6; valid=1; end
        else if (a[5]) begin b=5; valid=1; end
        else if (a[4]) begin b=4; valid=1; end
        else if (a[3]) begin b=3; valid=1; end
        else if (a[2]) begin b=2; valid=1; end
        else if (a[1]) begin b=1; valid=1; end
        else if (a[0]) begin b=0; valid=1; end
        else begin b = 3'bxxx; valid=0; end

endmodule

```

Przy projektowaniu koderów priorytetowych z dużą liczbą wejść wygodnie jest skorzystać z instrukcji pętli **for**:

```

module pr_encoder_8_3_for_valid(
    input [7:0] a,
    output reg [2:0] b,
    output reg valid);

    integer i;

    always @ (*)
        begin

```

```

        valid = 0;
        b = 3'bxxx;
        for (i=0; i<=7; i=i+1)
            if (a[i])
                begin
                    b = i;
                    valid = 1;
                end
        end
    end
endmodule

```

Na podstawie opisu modułu *pr_encoder_8_3_for_valid* można łatwo zbudować sparametryzowany moduł kodera priorytetowego. Proponujemy czytelnikowi zaprojektowanie modułu kodera priorytetowego z liczbą wejść określoną parametrem N .

5.5. Dekodery

Dekoder binarny (binary decoder) to układ cyfrowy, który przekształca *kod dwójkowy (binary code)* na *kod unarny (one-hot code)*. Kod dwójkowy podany na wejścia dekodera aktywuje tylko jedno wyjście, którego numer odpowiada liczbie dwójkowej na wejściach układu. W systemach wbudowanych dekodery mogą być używane do wyboru (aktywacji) urządzeń, które są adresowane kodem dwójkowym.

Tablica prawdy dekodera 3-bitowego kodu dwójkowego na 8-bitowy kod unarny (oznaczany jest jako 3:8) przedstawiona została w tabeli 5.3.

TABELA 5.3. Tablica prawdy pełnego binarnego dekodera 3:8

Wejścia			Wyjścia							
a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

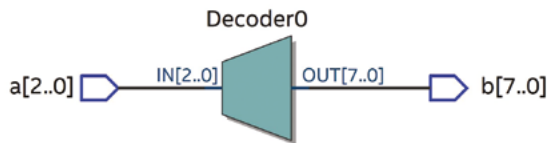
Dla pełnego kodera dwójkowego spełniony jest warunek: $N = 2^L$, gdzie L – liczba wejść, N – liczba wyjść dekodera. Jeśli zachodzi nierówność $N < 2^L$ (czyli nie wszystkie słowa kodowe są wykorzystywane), to dekodery nazywany jest *niepełnym*.

5.5.1. Sposoby opisu dekodерów

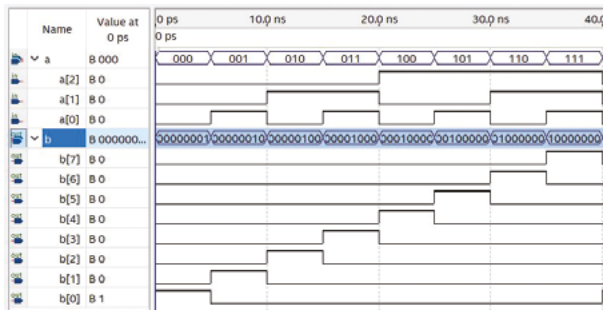
Najczęściej do opisu funkcjonowania dekodерów w języku Verilog wykorzystywana jest instrukcja **case**. Na przykład moduł pełnego dekodera binarnego 3:8 można przedstawić następująco:

```
module decoder_3_8(  
    input [2:0] a,  
    output reg [7:0] b);  
  
    always @(*)  
        case(a)  
            3'o0: b = 8'b00000001;  
            3'o1: b = 8'b00000010;  
            3'o2: b = 8'b00000100;  
            3'o3: b = 8'b00001000;  
            3'o4: b = 8'b00010000;  
            3'o5: b = 8'b00100000;  
            3'o6: b = 8'b01000000;  
            3'o7: b = 8'b10000000;  
        endcase  
endmodule
```

Wyniki syntezy i symulacji modułu *decoder_3_8* przedstawiono odpowiednio na rysunkach 5.47 i 5.48.



RYS. 5.47. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji **case**



RYS. 5.48. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji **case**

Ponieważ instrukcji **case** i **if** można używać zamiennie, pokażemy, jak dekodery 3:8 można opisać za pomocą instrukcji **if**:

```

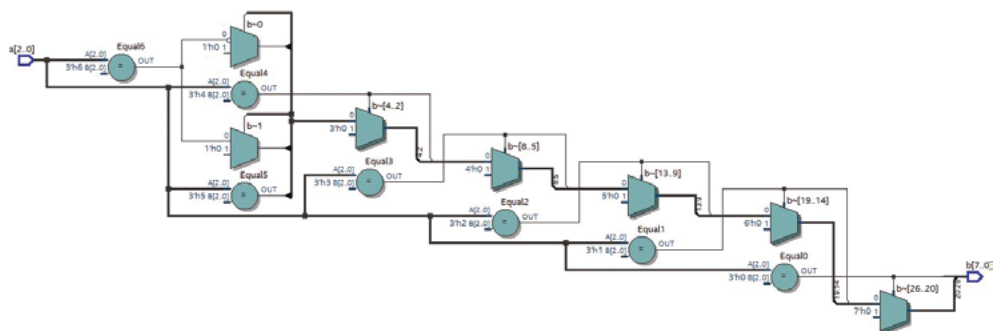
module decoder_3_8_if(
    input [2:0] a,
    output reg [7:0] b);

    always @(*)
        if (a == 3'b000) b = 8'b00000001;
        else if (a == 3'b001) b = 8'b00000010;
        else if (a == 3'b010) b = 8'b00000100;
        else if (a == 3'b011) b = 8'b00001000;
        else if (a == 3'b100) b = 8'b00010000;
        else if (a == 3'b101) b = 8'b00100000;
        else if (a == 3'b110) b = 8'b01000000;
        else
            b = 8'b10000000;

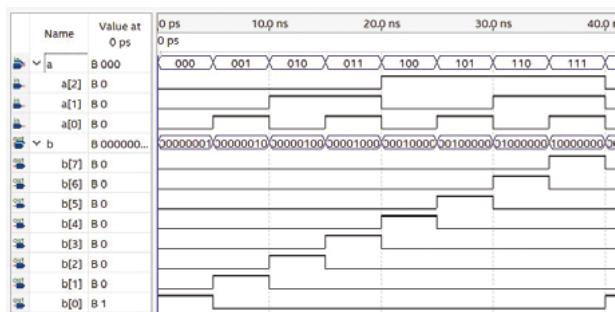
endmodule

```

Wyniki syntezy i symulacji modułu *decoder_3_8_if* możemy zobaczyć odpowiednio na rysunkach 5.49 i 5.50.



RYS. 5.49. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji **if**



RYS. 5.50. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji **if**

Dekoder 3:8 możemy opisać na podstawie wyrażeń logicznych zapisanych bezpośrednio z tablicy prawdy za pomocą instrukcji przypisania ciągłego **assign**:

```

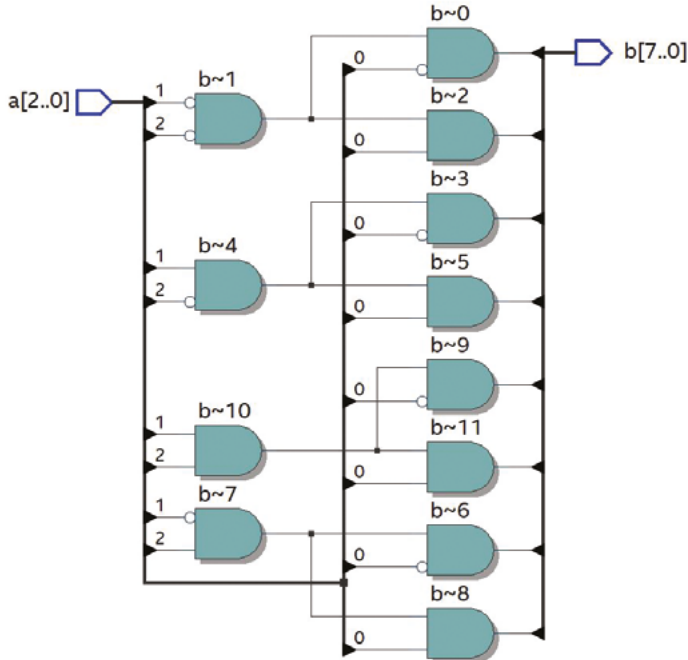
module decoder_3_8_assign(
    input [2:0] a,
    output [7:0] b);

    assign b[0] = ~a[2] & ~a[1] & ~a[0];
    assign b[1] = ~a[2] & ~a[1] & a[0];
    assign b[2] = ~a[2] & a[1] & ~a[0];
    assign b[3] = ~a[2] & a[1] & a[0];
    assign b[4] = a[2] & ~a[1] & ~a[0];
    assign b[5] = a[2] & ~a[1] & a[0];
    assign b[6] = a[2] & a[1] & ~a[0];
    assign b[7] = a[2] & a[1] & a[0];

endmodule

```

Wyniki syntezy i symulacji modułu *decoder_3_8_assign* przedstawione zostały na rysunkach 5.51 i 5.52 odpowiednio.



RYS. 5.51. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji **assign**



RYS. 5.52. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji **assign**

Przy projektowaniu dekoderek o dużej liczbie wyjść, w celu zmniejszenia rozmiaru kodu, można wykorzystać instrukcję **for**:

```

module decoder_3_8_for(
    input [2:0] a,
    output reg [7:0] b);

    integer i;

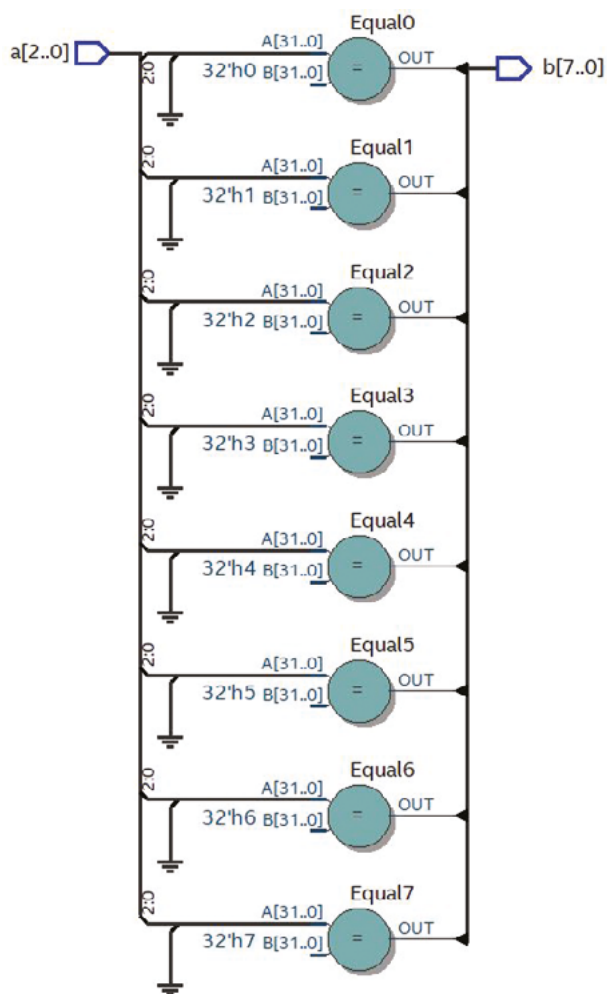
    always @(*)
        for (i=0; i<=7; i=i+1)
            if (a == i)    b[i] = 1;
            else         b[i] = 0;

endmodule

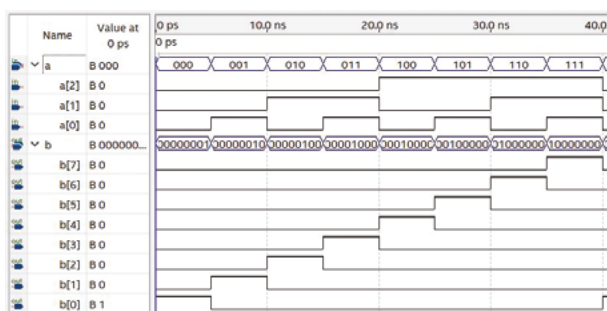
```

Wyniki syntezy i symulacji modułu *decoder_3_8_for* przedstawiono na rysunkach 5.53 i 5.54.

Na podstawie modułu *decoder_3_8_for* można utworzyć opis sparametryzowanego dekodera. Należy jednak pamiętać, że niektóre kompilatory narzucają ograniczenia na liczbę syntezyzowanych powtórzeń pętli. Proponujemy czytelnikowi zaprojektowanie pełnego dekodera binarnego z liczbą wejść określaną parametrem *N*.



RYS. 5.53. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji **for**



RYS. 5.54. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji **for**

5.5.2. Dekoder z wejściem zezwalającym

W systemach wbudowanych często realizowane są dekodery z dodatkowym wejściem *en* (*enable*) zezwalającym na pracę układu. Tablica prawdy dekodera 3:8 z wejściem zezwalającym została przedstawiona w tabeli 5.4.

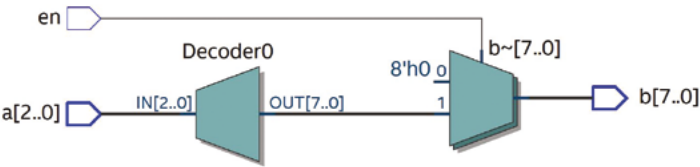
TABELA 5.4. Tablica prawdy dekodera 3:8 z wejściem zezwalającym

Wejścia				Wyjścia							
<i>en</i>	<i>a2</i>	<i>a1</i>	<i>a0</i>	<i>b7</i>	<i>b6</i>	<i>b5</i>	<i>b4</i>	<i>b3</i>	<i>b2</i>	<i>b1</i>	<i>b0</i>
0	–	–	–	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

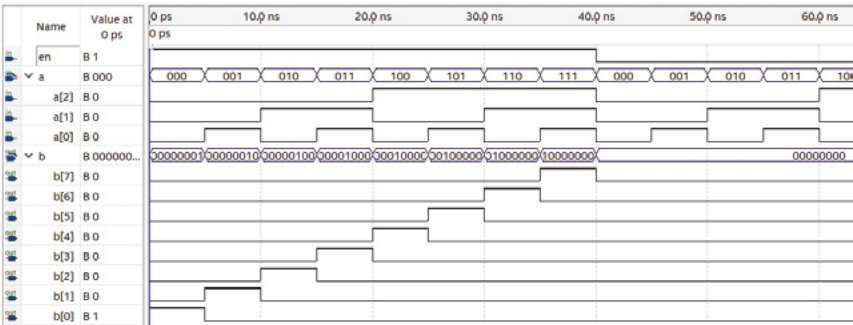
Dekoder 3:8 z wejściem zezwalającym można przedstawić w języku Verilog w następujący sposób:

```
module decoder_3_8_enable(  
    input [2:0] a,  
    input en,  
    output reg [7:0] b);  
  
    always @(*)  
        if (en)  
            case (a)  
                3'b000: b = 8'b00000001;  
                3'b001: b = 8'b00000010;  
                3'b010: b = 8'b00000100;  
                3'b011: b = 8'b00001000;  
                3'b100: b = 8'b00010000;  
                3'b101: b = 8'b00100000;  
                3'b110: b = 8'b01000000;  
                3'b111: b = 8'b10000000;  
            endcase  
        else  
            b = 8'b00000000;  
    endmodule
```

Wyniki syntezy i symulacji modułu *decoder_3_8_enable* przedstawiono na poniższych rysunkach 5.55 oraz 5.56.



RYS. 5.55. Wynik syntezy dekodera 3:8 z wejściem zezwolenia



RYS. 5.56. Wyniki symulacji dekodera 3:8 z wejściem zezwolenia

5.5.3. Niepełne dekodery

W niepełnym dekodерze liczba wyjść N jest mniejsza od liczby 2^L wszystkich możliwych kombinacji wejściowych. Tak więc w przypadku niepełnego dekodera 3:5 opisanego w tablicy prawdy (tabela 5.5), wartości wyjściowe określone zostały tylko dla pięciu z ośmiu możliwych kombinacji dwójkowych.

TABELA 5.5. Tablica prawdy niepełnego dekodera 3:5

Wejścia			Wyjścia				
a2	a1	a0	b4	b3	b2	b1	b0
0	0	0	0	0	0	0	1
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	0	1	0	0	0
1	0	0	1	0	0	0	0

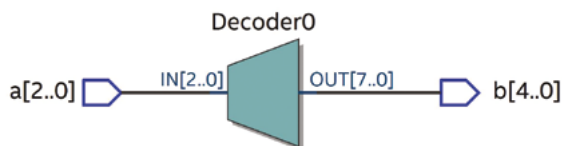
Przy projektowaniu niepełnego dekodera należy rozważyć, jakie wartości powinny być generowane na wyjściach dla niewykorzystywanych kombinacji wejściowych. Jeśli można założyć, że kombinacje wejściowe, które nie zostały ujęte w tablicy prawdy, nigdy nie pojawią się na wejściach układu, wówczas można dla nich ustawić nieokreślony stan wyjść **X**. Syntezator zastąpi nieokreślone stany wartościami logicznymi 0 lub 1, wybierając takie wartości, które pozwolą na lepszą optymalizację układu. Jednakże jeśli na wejściach dekodera mogą się pojawić dowolne kombinacje zmiennych, należy przypisać wyjściom dekodera konkretne wartości (na przykład zera) dla wszystkich niewykorzystywanych wektorów wejściowych.

Przedstawimy dwie wersje opisu projektu niepełnego dekodera 3:5. W pierwszej wersji zakładamy, że dla kombinacji wejściowych pominiętych w tablicy prawdy na wyjściach dekodera są generowane wartości zerowe:

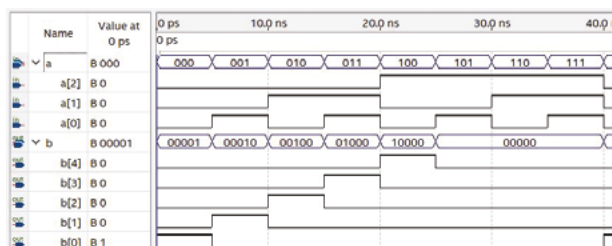
```
module decoder_3_5_v1(
    input [2:0] a,
    output reg [4:0] b);

    always @(*)
        case(a)
            3'o0: b = 5'b00001;
            3'o1: b = 5'b00010;
            3'o2: b = 5'b00100;
            3'o3: b = 5'b01000;
            3'o4: b = 5'b10000;
            default: b = 5'b00000;
        endcase
endmodule
```

Wyniki syntezy i symulacji modułu *decoder_3_5_v1* przedstawione zostały na rysunkach 5.57 oraz 5.58.



RYS. 5.57. Wynik syntezy niepełnego dekodera 3:5 z zerowymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych



RYS. 5.58. Wyniki symulacji niepełnego dekodera 3:5 z zerowymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych

W drugiej wersji projektu niepełnego dekodera 3:5 przyjmujemy, że dla niewykorzystywanych kombinacji wejściowych zostaną ustalone wartości nieokreślone na wyjściach:

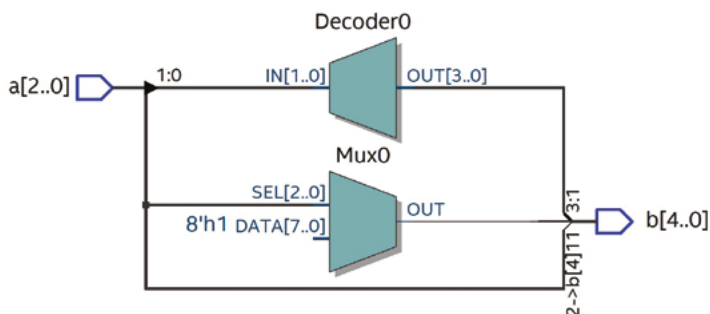
```

module decoder_3_5_v2(
    input [2:0] a,
    output reg [4:0] b);

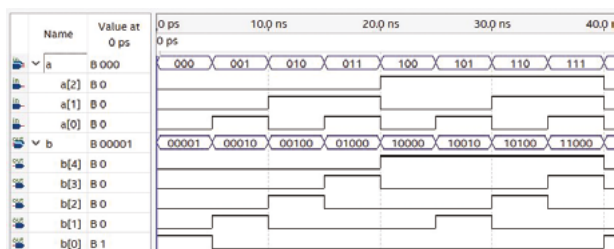
    always @(*)
        case(a)
            3'o0: b = 5'b00001;
            3'o1: b = 5'b00010;
            3'o2: b = 5'b00100;
            3'o3: b = 5'b01000;
            3'o4: b = 5'b10000;
            default: b = 5'bxxxxx;
        endcase
endmodule

```

Wyniki syntezy i symulacji modułu *decoder_3_5_v2* przedstawione zostały na rysunkach 5.59 i 5.60.



RYS. 5.59. Wynik syntezy niepełnego dekodera 3:5 z nieokreślonymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych



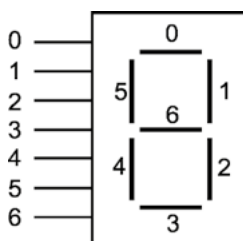
RYS. 5.60. Wyniki symulacji niepełnego dekodera 3:5 z nieokreślonymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych

Z przebiegów czasowych na rysunku 5.60 możemy zauważyć, że dla kombinacji wejściowych 3'o5, 3'o6 i 3'o7 na wyjściach dekodera pojawiają się dwie jedynki. Dzieje się tak, ponieważ syntezy zastępuje nieokreślone wartości na wyjściach zerami lub jedynkami w celu optymalizacji schematu. Z tego powodu na wyjściach dekodera mogą się pojawić nieoczekiwane wartości zakłócające funkcjonowanie układu. Dlatego też lepszym rozwiązaniem jest przypisanie wyjściom konkretnych wartości, jak to zostało wykonane w pierwszej wersji projektu.

5.5.4. Dekodery wyświetlaczy siedmiosegmentowych

Dekoder może być wykorzystany w systemach wbudowanych do przekształcania kodów dwójkowych na kody obsługujące wyświetlacz siedmiosegmentowy. W takim przypadku na wejściach dekodera podawany jest kod znaku do wyświetlania, a na wyjściach generowane są odpowiednie sygnały sterujące wyświetlaczem siedmiosegmentowym. W przeciwieństwie do zwykłego dekodera binarnego na wyjściach dekodera wyświetlacza siedmiosegmentowego jednocześnie może się pojawić kilka aktywnych sygnałów sterujących. Pozwala to na wyświetlanie nie tylko cyfr dziesiętnych, lecz także cyfr szesnastkowych oraz niektórych liter.

Na rysunku 5.61 przedstawiono symbol graficzny wyświetlacza siedmiosegmentowego z zaznaczonymi numerami segmentów wyświetlacza oraz wejść sterujących segmentami.



RYS. 5.61. Wyświetlacz siedmiosegmentowy

Dla wyświetlaczy siedmiosegmentowych umieszczonych na płytce prototypowej DE1-SoC aktywnym poziomem sygnałów sterowania segmentami jest poziom niski. Tablica prawdy dekodera siedmiosegmentowego wyświetlacza do wypisywania cyfr dziesiętnych zamieszczona została w tabeli 5.6.

TABELA 5.6. Tablica prawdy dekodera cyfr dziesiętnych do sterowania wyświetlaczem siedmiosegmentowym

Wejścia				Wyjścia						
a3	a2	a1	a0	h0	h1	h2	h3	h4	h5	h6
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0

Dekoder wyświetlacza siedmiosegmentowego do wypisywania liczb dziesiętnych można przedstawić w języku Verilog w następujący sposób:

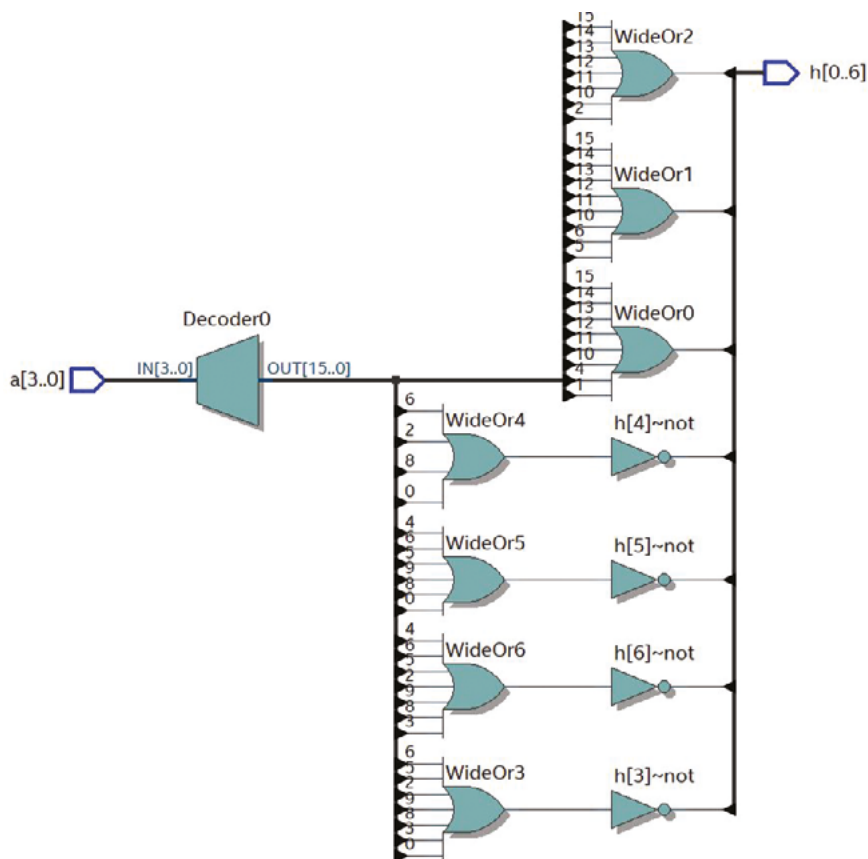
```

module decoder_hex_10(
    input [3:0] x,
    output reg [0:6] h);

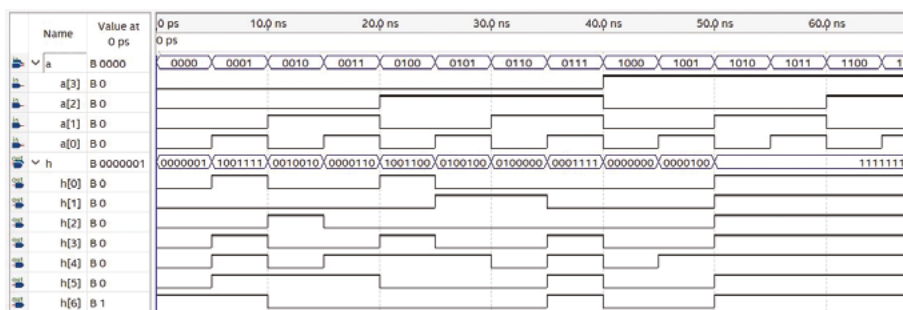
    always @(*)
        casex(x)
            4'd0: h=7'b0000001;
            4'd1: h=7'b1001111;
            4'd2: h=7'b0010010;
            4'd3: h=7'b0000110;
            4'd4: h=7'b1001100;
            4'd5: h=7'b0100100;
            4'd6: h=7'b0100000;
            4'd7: h=7'b0001111;
            4'd8: h=7'b0000000;
            4'd9: h=7'b0000100;
            default: h=7'b1111111;
        endcase
endmodule

```

W przedstawionym powyżej przykładzie przyjęto domyślną wartość wyjść $h=7'b1111111$, co opowiada wygaszeniu wszystkich segmentów wyświetlacza. Wyniki syntezy i symulacji modułu *decoder_hex_10* zostały przedstawione na rysunkach 5.62 i 5.63.



RYS. 5.62. Wynik syntezy dekodera cyfr dziesiętnych



RYS. 5.63. Wyniki symulacji dekodera cyfr dziesiętnych

Zaprojektowany dekodery nazywany jest także dekoderm kody BCD na kod wyświetlacza siedmiosegmentowego (*BCD to seven-segment decoder*).

Czytelnikowi proponujemy samodzielne zaprojektowanie modułu *decoder_hex_16* dekodera do wypisywania cyfr szesnastkowych na wyświetlaczu siedmiosegmentowym.

5.5.5. Dekodery do wyświetlania tekstu na wyświetlaczu siedmiosegmentowym

W celu wypisania tekstu na wyświetlaczach siedmiosegmentowych konieczne jest przypisanie kodów dwójkowych wszystkim literom, które chcemy wyświetlić. Jednakże nie wszystkie litery mogą być wyświetlone za pomocą siedmiu segmentów. Czasami konieczne jest również korzystanie z różnych wariantów liter, na przykład zamiast dużej litery używamy jej małej formy i odwrotnie. Przykładowo możemy zaprojektować dekodery do wyświetlania słowa „HELLO” na wyświetlaczach siedmiosegmentowych. W tym celu należy zakodować litery używane w tym słowie, jak przedstawiono w tabeli 5.7.

TABELA 5.7. Tablica prawdy dekodera do wyświetlania słowa „HELLO” na wyświetlaczu siedmiosegmentowym

Litera	Wejścia		Wyjścia						
	c1	c0	h6	h5	h4	h3	h2	h1	h0
H	0	0	0	0	0	1	0	0	1
E	0	1	0	0	0	0	1	1	0
L	1	0	1	0	0	0	1	1	1
O	1	1	1	0	0	0	0	0	0

Opis dekodera do wypisywania słowa „HELLO” na wyświetlaczach siedmiosegmentowych w języku Verilog można przedstawić następująco:

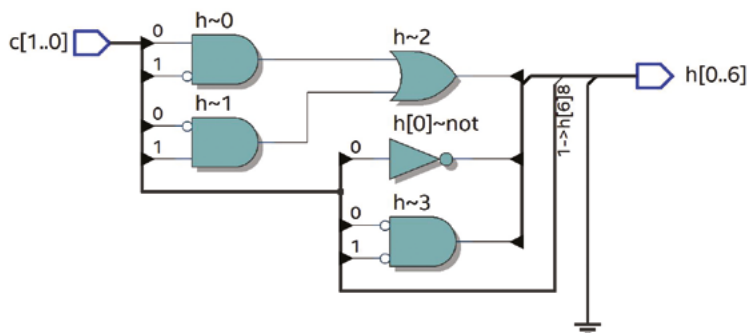
```

module decoder_hex_HELLO(
    input [1:0] c,
    output [0:6] h);

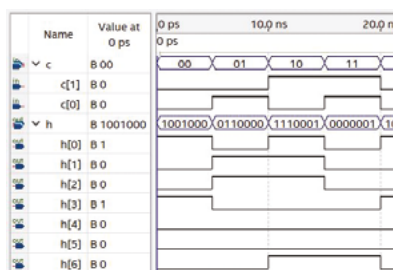
    assign h[0] = ~c[0];
    assign h[1] = ~c[1] & c[0] | c[1] & ~c[0];
    assign h[2] = h[1];
    assign h[3] = ~c[1] & ~c[0];
    assign h[4] = 1'b0;
    assign h[5] = 1'b0;
    assign h[6] = c[1];
endmodule

```


W przedstawionym przykładzie do opisu dekodera zostały wykorzystane wyłącznie instrukcje przypisania ciągłego **assign**. Wyniki syntezy i symulacji modułu *decoder_hex_HELLO* znajdują się na rysunkach 5.64 i 5.65.



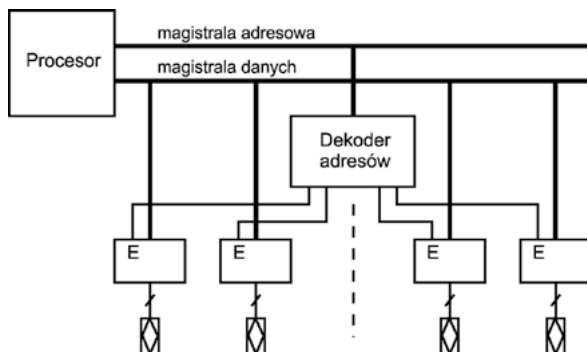
RYS. 5.64. Wynik syntezy dekodera do wypisywania słowa „HELLO” na wyświetlaczach siedmio-segmentowych



RYS. 5.65. Wyniki symulacji dekodera do wypisywania słowa „HELLO” na wyświetlaczach siedmio-segmentowych

5.5.6. Dekodery adresów

Dekoder binarny, który jest podłączony do magistrali adresowej systemu mikroprocesorowego, nazywany jest *dekoderem adresów* (*address decoder*). Jego wyjścia podłączone są do wejść zezwolenia pracy odpowiednich bloków peryferyjnych systemu, takich jak rejestry, pamięci ROM, RAM i inne. Dzięki temu możliwe jest aktywowanie wybranych urządzeń poprzez ich adres (rysunek 5.66).



RYS. 5.66. Dekoder adresów w systemie mikroprocesorowym

5.6. Podsumowanie rozdziału

Bufory trójstanowe w projektach opartych na układach FPGA zazwyczaj są umieszczane na wyjściach systemu w celu umożliwienia odłączenia go od zewnętrznej szyny. W tym przypadku bufory trójstanowe są realizowane w blokach wejścia-wyjścia układu FPGA. Mogą być także umieszczone wewnątrz układu FPGA, gdzie są realizowane za pomocą wewnętrznej logiki. Niektóre układy FPGA mają też wbudowane magistrale z buforami trójstanowymi.

Niektóre standardowe bloki funkcjonalne łatwiej opisać od podstaw w języku Verilog, niż wykorzystywać gotowe prymitywy lub jego szablony.

Multiplexer (multiplexer) jest układem cyfrowym, który przekazuje dane od kilku źródeł do jednego wyjścia. Proces ten nazywany jest multipleksacją. W przypadku *multipleksacji czasowej* wejścia multiplexera są podłączane kolejno do wyjścia przez krótki czas przez cykliczną zmianę kodu na wyprowadzeniach sterujących układem. Multiplexer czasem jest określany jako selektor lub przełącznik.

Opis kombinacyjnych bloków funkcjonalnych w języku Verilog może być wykonany w postaci:

- opisu strukturalnego z wykorzystaniem prymitywów języka Verilog lub instancji modułów niższego poziomu;
- opisu behawioralnego (funkcjonalnego):
 - w postaci wyrażeń (logicznych lub arytmetycznych) za pomocą instrukcji przypisania ciągłego **assign**;
 - za pomocą instrukcji proceduralnych (przypisania blokującego = , **if**, **case**, rzadziej **for**).

Przy wykorzystaniu instrukcji proceduralnych wyjścia układów kombinacyjnych powinny być zadeklarowane jako zmienne typu **reg**.

Demultiplexer (demultiplexer) wykonuje funkcję odwrotną do multiplexera: przesyła dane od jednego źródła do wielu odbiorników.

Układ *komutatora* w systemach transmisji danych może być zbudowany przez sekwencyjne połączenie multipleksera i demultipleksera. Wejścia adresowe multipleksera służą do wyboru źródła przesyłania informacji, natomiast wejścia adresowe demultipleksera wybierają docelowy odbiornik.

W przypadku wykorzystania instrukcji proceduralnych **case** i **if** do opisu demultipleksów należy określić domyślne wartości sygnałów wyjściowych w celu uniknięcia automatycznego umieszczenia zatrząsków na wyjściach demultipleksa.

Koder (encoder) jest układem cyfrowym, który przekształca *kod unarny (one-hot code)* na *kod binarny (binary code)*. W systemach cyfrowych kodery są wykorzystywane do generowania kodu dwójkowego, który wybiera aktywne w danym momencie urządzenie.

Koder priorytetowy (priority encoder) generuje kod dwójkowy odpowiadający aktywnemu wejściu o najwyższym priorytecie. Kodery priorytetowe znajdują zastosowanie w systemach wbudowanych do realizacji schematów wyboru na podstawie priorytetów, takich jak priorytetowa obsługa przerwań, przydzielanie magistrali urządzeniu o najwyższym priorytecie i inne.

Przy projektowaniu koderów w języku Verilog należy uwzględnić sytuacje, w których na wejściach mogą się pojawić różne kombinacje, takie jak kilka jedynek, wartości nieokreślone **X** lub stany wysokiej impedancji **Z**. Aby skutecznie identyfikować niedozwolone kombinacje wejściowe, można rozważyć dodanie dodatkowego wyjścia *valid*. Wartość tego wyjścia wynosiłaby 1 tylko w przypadku, gdy na wejściach kodera pojawiają się kody unarne, a dla pozostałych kombinacji wartość byłaby równa 0.

Przy projektowaniu koderów priorytetowych należy zwrócić uwagę na wykrywanie kombinacji zerowej na wejściach kodera. W tym celu także można wykonać dodatkowe wyjście *valid*, które będzie przyjmować wartość 1 przy pojawieniu się jedynki przynajmniej na jednym z wejść kodera priorytetowego, natomiast 0 na wyjściu *valid* będzie wskazywać na zerową kombinację wejściową.

Dekoder binarny (*binary decoder*) jest układem cyfrowym, który wykonuje zamianę kodu dwójkowego (*binary code*) na kod unarny (*one-hot code*). Dekodery w systemach wbudowanych są wykorzystywane do wyboru urządzeń zaadresowanych kodem dwójkowym.

Do opisu dekodekserów w języku Verilog najlepiej nadaje się instrukcja **case**, ale czasami mogą być też wykorzystane instrukcje **if**, **for** i **assign**.

W niepełnym dekodekserze liczba wyjść N jest mniejsza od liczby 2^L wszystkich możliwych kombinacji wejściowych. Jeśli jest pewność, że kombinacje wejściowe, które nie zostały ujęte w tablicy prawdy, nigdy nie pojawią się na wejściach układu, można dla nich ustawić nieokreślony stan wyjść **X**. Syntezator zastąpi nieokreślone stany **X** wartościami logicznymi 0 lub 1, tak aby zoptymalizować schemat. Jednakże jeśli na wejściach dekodeksera mogą się pojawić dowolne kombinacje zmiennych, wtedy należy przypisać wyjściom dekodeksera konkretne wartości (na przykład zera) dla wszystkich niewykorzystywanych wektorów wejściowych.

Dekoder binarny podłączony do magistrali adresowej systemu mikroprocesorowego nazywany jest *dekoderem adresów* (*address decoder*). Wyjścia dekodera adresów podłączone są do wejść zezwolenia pracy odpowiednich bloków peryferyjnych systemu.

Podczas projektowania dekoderek dla wyświetlaczy siedmiosegmentowych, dla kombinacji sygnałów wejściowych, które nie są wykorzystywane, należy ustawić takie wartości na wyjściach dekodera, które wygaszą segmenty wyświetlacza.

6. Projektowanie układów sekwencyjnych

W dziedzinie techniki cyfrowej istnieje powszechnie przyjęty podział układów na dwa podstawowe rodzaje: kombinacyjne oraz sekwencyjne. Układy kombinacyjne charakteryzują się brakiem pamięci, a stan ich wyjść zależy jedynie od aktualnego stanu wejść. Do tego rodzaju układów zaliczane są między innymi omówione wcześniej multipleksery, demultipleksery, kodery i dekodery. Układy sekwencyjne cechują się posiadaniem pamięci, co umożliwia im zapamiętywanie całej sekwencji kombinacji sygnałów, które zostały wcześniej podane na wejście schematu (stąd też nazwa *układ sekwencyjny*). Wyjścia tego układu zależą nie tylko od bieżącego stanu wejść, lecz także od stanu pamięci.

Najprostszymi elementami pamięciowymi są zatrzaski i przerzutniki, które pozwalają przechowywać jeden bit informacji. Do przechowywania większych ilości informacji wykorzystywane są rejestry.

Zatrzask (latch) to element pamięciowy, który jest aktywowany *poziomem sygnałem* na wejściu zegarowym. Inny sposób sterowania ma *przerzutnik (flip-flop)*, który wyzwalany jest *zbozcem sygnału* zegarowego – *narastającym (dodatnim)* lub *opadającym (ujemnym)*, w zależności od konkretnego rozwiązania zastosowanego w układzie. W przypadku układów FPGA przerzutniki zazwyczaj są przełączane za pomocą narastającego zbocza sygnału.

Specyficzną cechą projektowania zatrzasków i przerzutników przy użyciu języka Verilog jest fakt, że nie można ich opisać bezpośrednio w kodzie projektu. Zamiast tego zatrzaski i przerzutniki są automatycznie wprowadzane do schematu przez kompilator na podstawie kodu projektu. W niektórych sytuacjach pojawienie się zatrzasków w projekcie może być niespodzianką dla projektanta, dlatego ważne jest zawsze śledzenie komunikatów kompilatora, które informują o automatycznym dodaniu zatrzasków do projektu. Podobnie należy zachować ostrożność przy projektowaniu przerzutników, ponieważ ich implementacja w strukturach FPGA firmy Intel również ma własne specyficzne cechy.

W tym rozdziale omówione zostaną aspekty projektowania na układach FPGA przy użyciu języka Verilog elementów pamięciowych, takich jak zatrzaski, przerzutniki, rejestry, rejestry przesuwne oraz bloki rejestrów.

6.1. Zatrzaski

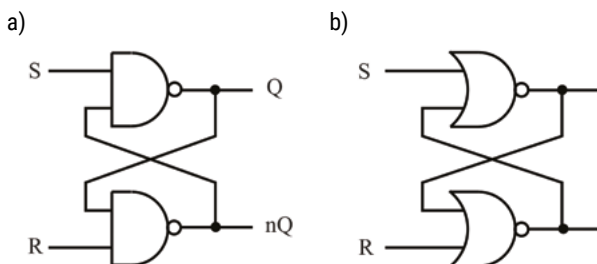
Producenci układów FPGA nie zalecają stosowania zatrząsków w projektach. Wynika to z faktu, że nie mają one swoich odpowiedników sprzętowych w strukturach FPGA, więc muszą być implementowane za pomocą elementów logicznych (w strukturach FPGA sprzętowo zrealizowane zostały tylko przerzutniki). Z kolei to może prowadzić do nieprawidłowego działania projektu. Zaleca się korzystanie z zatrząsków jedynie w przypadkach, gdy nie da się ich zastąpić przerzutnikami lub rejestrami.

Kompilator może automatycznie dodawać zatrzaski na wyjściach układów kombinacyjnych w sytuacji, gdy w kodzie projektu nie zostały uwzględnione wszystkie możliwe kombinacje wejściowe sygnałów oraz odpowiadające im wartości zmieniających na wyjściach. Na przykład zatrzaski będą automatycznie wprowadzane do schematu, gdy w instrukcji **if** nie zostanie uwzględniona konstrukcja **else**, a także jeśli w instrukcji **case** brakuje konstrukcji **default** i nie są wymienione wszystkie możliwe wartości stałych elementów. Kompilator systemu Quartus zawsze dostarcza informacji o automatycznie dodanych zatrząskach do schematu, dlatego istotne jest dokładne czytanie komunikatów kompilatora oraz analiza wyników syntezy projektu na poziomie RTL.

Należy zauważyć, że w niektórych przypadkach zatrzaski mogą zostać automatycznie dodane przez kompilator do projektu, nawet jeśli dla każdej instrukcji **if** przyporządkowano konstrukcję **else**.

6.1.1. Zatrzaski typu SR

Zatrzaski typu SR są najprostszymi układami, które pozwalają przechowywać informacje. Można je zbudować, wykorzystując bramki NAND (rysunek 6.1a) lub bramki NOR (rysunek 6.1b).



RYS. 6.1. Schemat zatrzązku typu SR: a) na elementach NAND; b) na elementach NOR

Sposób funkcjonowania zatrzązku typu SR opisany został w tabeli 6.1.

TABELA 6.1. Tablica prawdy zatrzasku typu SR

S	R	Q	nQ
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	x	x

Przy podaniu wartości 1 na wejście *R* (*reset*) wyjście *Q* zatrzasku przyjmuje wartość 0 (przy czym wyjście *nQ* otrzymuje wartość przeciwną); wystąpienie wartości 1 na wejściu *S* (*set*) powoduje przełączenie wyjścia *Q* zatrzasku w stan 1. Kombinacja 00 na wejściach *S* i *R* powoduje zachowanie poprzednich wartości wyjść. Jednak równoczesne podanie wartości 1 na oba wejścia *S* i *R* jest kombinacją niedozwoloną. Niemniej jednak w praktyce syntezyator może przypisać określone wartości dla tej kombinacji, na przykład wartości zerowe lub przeciwności do poprzednich.

Poniżej przedstawiono opis zatrzasku typu SR na podstawie rysunku 6.1a, wykorzystując prymitywy języka Verilog:

```

module latch_SR_NAND(
    input S,R,
    output Q,nQ);

    nand g1(Q,S,nQ);
    nand g2(nQ,R,Q);
endmodule

```

Opis zatrzasku SR z wykorzystaniem wyrażeń logicznych i instrukcji **assign** można przedstawić następująco:

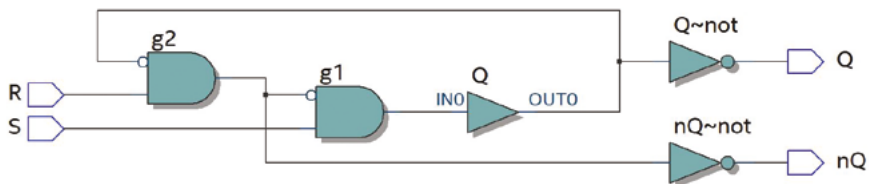
```

module latch_SR_assign_NAND(
    input S,R,
    output Q,nQ);

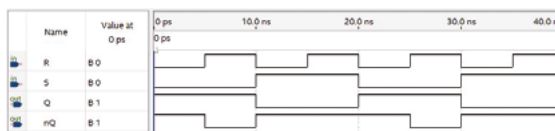
    assign Q = ~(S & nQ);
    assign nQ = ~(R & Q);
endmodule

```

Wyniki syntezy i symulacji dla obu wersji opisu są identyczne, możemy je zaobserwować na 6.2 i 6.3.



RYS. 6.2. Wynik syntezy zatrzaśku SR realizowanego za pomocą modułów *latch_SR_NAND* i *latch_SR_assign_NAND*



RYS. 6.3. Wyniki symulacji zatrzaśku SR realizowanego za pomocą modułów *latch_SR_NAND* i *latch_SR_assign_NAND*

Analiza przebiegów na rysunku 6.3 pokazuje, że zaprojektowany układ nie zachowuje się zgodnie z opisem. Na przykład przy $R = 1$ otrzymujemy na wyjściu $Q = 1$, podczas gdy przy $S = 1$ wyjście $Q = 0$. Te wartości są zupełnie odwrotne w porównaniu z tymi podanymi w tabeli 6.1.

Projekt zatrzaśku SR wykonany na podstawie rysunku 6.1b z wykorzystaniem prymitywów języka Verilog można przedstawić następująco:

```

module latch_SR_NOR(
    input S,R,
    output Q,nQ);

    nor g1(nQ,S,Q);
    nor g2(Q,R,nQ);
endmodule

```

Ten sam projekt, lecz opisany z wykorzystaniem wyrażeń logicznych i instrukcji **assign**, przedstawiono poniżej:

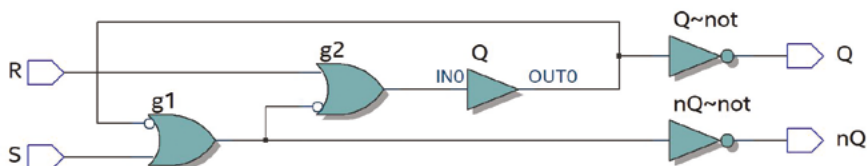
```

module latch_SR_assign_NOR(
    input S,R,
    output Q,nQ);

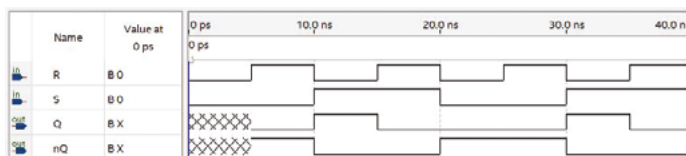
    assign Q = ~(R | nQ);
    assign nQ = ~(S | Q);
endmodule

```


Otrzymane wyniki syntezy i symulacji są identyczne dla obu projektów, możemy je zobaczyć odpowiednio na rysunkach 6.4 i 6.5.



RYS. 6.4. Wynik syntezy zatrasku SR realizowanego na podstawie modułów *latch_SR_NOR* i *latch_SR_assign_NOR*



RYS. 6.5. Wyniki symulacji zatrasku SR realizowanego na podstawie modułów *latch_SR_NOR* i *latch_SR_assign_NOR*

Analizując wykresy na rysunku 6.5, można zauważyć, że działanie układu ponownie nie odpowiada oczekiwaniom. Chociaż w tym przypadku reakcja układu na sygnały zerowania ($R = 1$) oraz ustawienia ($S = 1$) jest poprawna, to jednak przy podaniu kombinacji 00 na wejścia S i R wyjścia nie zachowują poprzednich wartości.

Wniosek: realizacja zatrasku SR na podstawie schematu z wykorzystaniem prymitywów języka Verilog lub odpowiednich wyrażeń logicznych prowadzi do niepoprawnego działania układu.

Powyższe przykłady pokazały, że nie da się opisać zatrasku w kodzie projektu jawnie. Spróbujemy przedstawić układ w inny sposób, wykorzystując w tym celu instrukcję proceduralną `if`:

```

module latch_SR_if(
    input S,R,
    output reg Q,
    output nQ);

    always @(*)
        if (S)          Q=1'b1;
        else if (R)    Q=1'b0;

    assign nQ=~Q;
endmodule

```

Opis zatrzasku SR wykonamy także za pomocą instrukcji **assign** i operacji warunkowej:

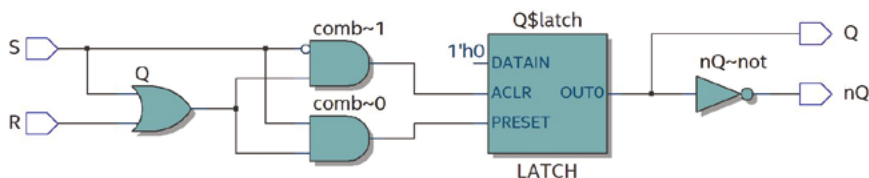
```

module latch_SR_assign(
    input S,R,
    output Q,nQ);

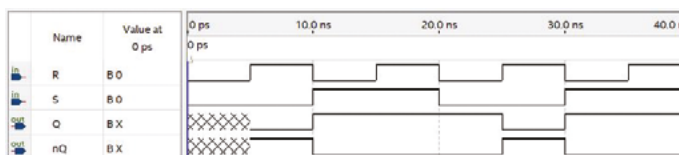
    assign Q = (S) ? 1'b1 : (R) ? 1'b0 : Q;
    assign nQ = ~Q;
endmodule

```

Obie wersje opisu zatrzasku SR dają takie same wyniki syntezy i symulacji, przedstawione na rysunkach 6.6 i 6.7.



RYS. 6.6. Wynik syntezy zatrzasku SR realizowanego na podstawie modułów *latch_SR_if* i *latch_SR_assign*



RYS. 6.7. Wyniki symulacji zatrzasku SR realizowanego na podstawie modułów *latch_SR_if* i *latch_SR_assign*

Analizując schemat na rysunku 6.6, można zauważyć, że zatrzask SR został zbudowany przy użyciu prymitywu LATCH systemu Quartus, który ma wejścia asynchronicznego zerowania (ACLR) i ustawienia (PRESET). Aby sterować tymi wejściami, zbudowano dodatkowy schemat z elementów logicznych. W rezultacie cały układ zatrzasku SR jest realizowany przy użyciu dwóch elementów LUT.

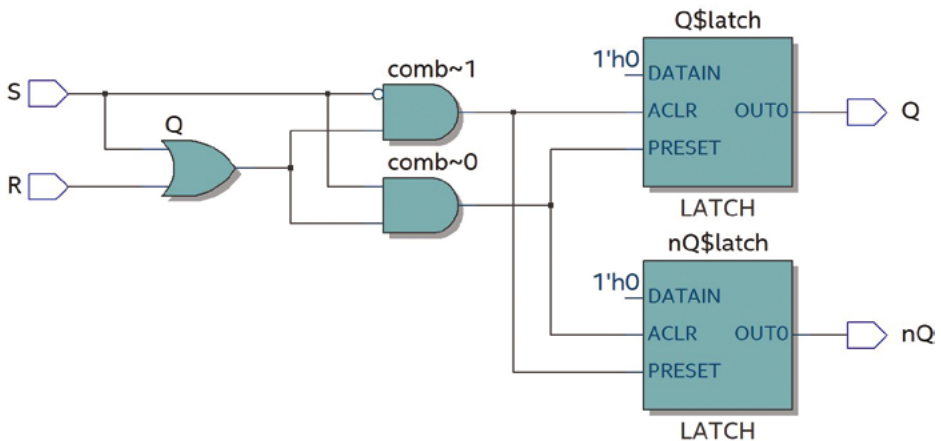
Na przebiegach czasowych prezentujących wyniki symulacji możemy zaobserwować, że zaprojektowany zatrzask SR przez pierwsze 5 ns dla zerowych wartości na wejściach S i R generuje nieokreślone stany X na wyjściach. Następnie jednak zaczyna funkcjonować zgodnie z tabelą 6.1. Warto również zauważyć, że przy podaniu na wejścia zabronionej kombinacji 11, wartości na wyjściach zmieniają się na przeciwne.

Zaproponujemy jeszcze jeden sposób opisu zatrzasku SR – za pomocą instrukcji **if**:

```
module latch_SR_if_v2(
    input S,R,
    output reg Q,nQ);

    always @(*)
        if (S)
            begin
                Q=1'b1;
                nQ=1'b0;
            end
        else if (R)
            begin
                Q=1'b0;
                nQ=1'b1;
            end
    end
endmodule
```

W tym przypadku (rysunek 6.8) każde z wyjść (Q oraz nQ) generowane jest za pomocą oddzielnego prymitywu LATCH, a do realizacji układu potrzebne są trzy elementy LUT. Wyniki symulacji są dokładnie takie same jak na rysunku 6.7.

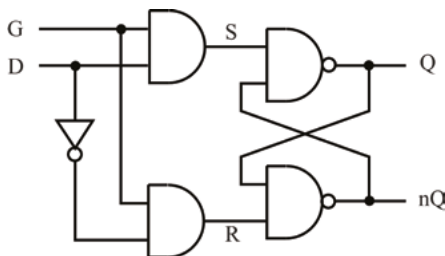


RYS. 6.8. Wynik syntezy zatrzasku SR na podstawie modułu *latch_SR_if_v2*

Wniosek. Projekt zatrzasku SR na FPGA należy opisywać w języku Verilog za pomocą instrukcji **if** lub za pomocą odpowiedniej operacji warunkowej w instrukcji przypisania ciągłego **assign**.

6.1.2. Transparentny przerzutnik typu D

Zatraski typu SR są najprostszymi elementami pamięciowymi, które mogą przechowywać pojedynczy bit informacji. W praktyce są one stosowane bardzo rzadko, zwykle służą jako podstawa do budowy zatrasków innych typów, bardziej nadających się do wykorzystania w projektach. Jednym z takich elementów jest *zatrask typu D*, inaczej nazywany *transparentnym przerzutnikiem typu D* (rysunek 6.9).



RYS. 6.9. Schemat zatrasku typu D

Zatrask typu D wyposażony jest w jedno wejście danych *D*, wyjście *Q*, wejście bramkujące *G* (w niektórych układach nazywane *LE* – *latch enable*). Dodatkowo może mieć wejścia asynchronicznego zerowania i ustawienia. Gdy na wejściu bramkującym *G* występuje wysoki poziom, sygnał z wejścia danych *D* jest przekazywany bez zmian (z pewnym opóźnieniem) na wyjście *Q*. Przełączenie sygnału na wejściu bramkującym *G* z wysokiego na niski powoduje zapamiętanie na wyjściu *Q* ostatniej wartości z wejścia *D*. Warto zauważyć, że przy wysokim stanie na wejściu bramkującym *G* wyjście zatrasku *Q* jest funkcją kombinacyjną względem wejścia.

W języku Verilog nie jest możliwe bezpośrednie opisanie zatrasków; są one automatycznie wprowadzane do projektu na podstawie kodu projektu. Najczęściej zatraski są dodawane za pomocą instrukcji proceduralnej **always**. Aby zatrask został automatycznie dodany do schematu, muszą być spełnione dwa warunki:

- 1) istnieje co najmniej jedna kombinacja zmiennych wejściowych, dla której nie zostały określone wartości wyjściowe;
- 2) lista czułości instrukcji **always** nie zawiera zdarzeń wyzwalanych zboczem sygnału (czyli słów kluczowych **posedge** i **negedge**).

Proponujemy czytelnikowi opisanie zatrasku typu D na podstawie schematu przedstawionego na rysunku 6.9, wykorzystując prymitywy języka Verilog. Następnie sugerujemy przeprowadzenie syntezy i symulacji projektu w celu sprawdzenia, czy wyniki symulacji odpowiadają oczekiwanemu funkcjonowaniu zatrasku typu D.

Transparentny przerzutnik typu D można opisać w języku Verilog w prosty sposób za pomocą instrukcji **if**:

```
module latch_if(
    input D,G,
    output reg Q);

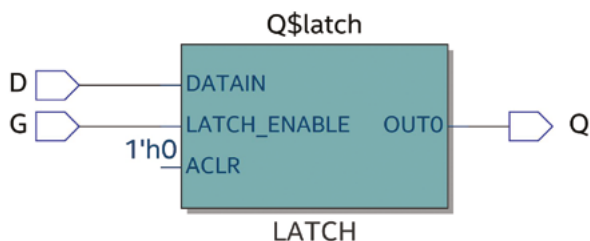
    always @(*)
        if (G) Q=D;
endmodule
```

Można też do opisu wykorzystać operację warunkową w instrukcji **assign**:

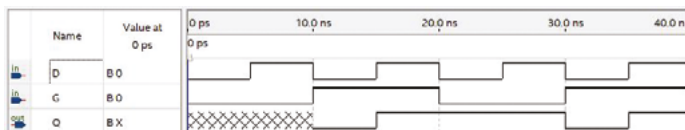
```
module latch_assign(
    input D,G,
    output Q);

    assign Q = (G) ? D : Q;
endmodule
```

Oba opisy projektu prowadzą do tych samych wyników syntezy (rysunek 6.10) i symulacji (rysunek 6.11).



RYS. 6.10. Wynik syntezy modułów *latch_if* i *latch_assign*



RYS. 6.11. Wyniki symulacji modułów *latch_if* i *latch_assign*

Z rysunku 6.10 wynika, że zatrzaszk typu D został zrealizowany przy użyciu jednego prymitywu LATCH systemu Quartus. Sygnał *D* jest połączony z wejściem DATAIN, natomiast sygnał *G* – z wejściem LATCH_ENABLE prymitywu LATCH.

Wyniki symulacji funkcjonalnej wykazują, że na początku, gdy sygnał na wejściu bramkującym *G* jest niski, wyjście *Q* przyjmuje stan nieokreślony **X**. Po podaniu wysokiego poziomu na wejście *G* wyjście *Q* powtarza zmiany sygnału na wejściu *D*, a każde kolejne przełączenie sygnału bramkującego *G* z wysokiego na niski powoduje zapamiętanie na wyjściu *Q* poprzedniej wartości. Przebiegi czasowe na rysunku 6.11 potwierdzają, że zaprojektowany zatrzask funkcjonuje poprawnie.

Przedstawimy dwie wersje opisu zatrzasku z sygnałem zerowania *reset*. Pierwsza wersja projektu wykorzystuje instrukcję **if**:

```
module latch_if_reset(
    input D,G,reset,
    output reg Q);

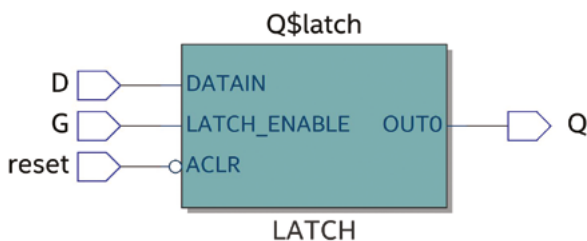
    always @(*)
        if (~reset)    Q=1'b0;
        else if (G)    Q=D;
endmodule
```

W drugiej wersji specyfikacji projektu zatrzask opisywany jest za pomocą operacji warunkowej i instrukcji **assign**:

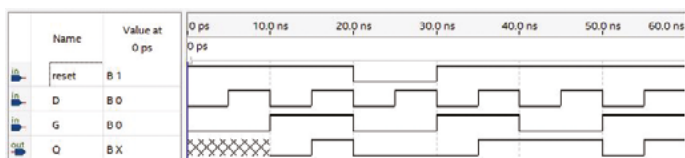
```
module latch_assign_reset(
    input D,G,reset,
    output Q);

    assign Q = (~reset) ? 1'b0 : (G) ? D : Q;
endmodule
```

Oba opisy prowadzą do otrzymania identycznych wyników syntezy i symulacji, które pokazane zostały na rysunkach 6.12 i 6.13.



RYS. 6.12. Wynik syntezy modułów *latch_if_reset* i *latch_assign_reset*



RYS. 6.13. Wyniki symulacji modułów *latch_if_reset* i *latch_assign_reset*

Na rysunku 6.12 zauważamy, że do zerowania zatrzasku za pomocą sygnału *reset* wykorzystywane jest wejście *ACLR* prymitywu LATCH. Wyniki symulacji (rysunek 6.13) pokazują, że oba projekty funkcjonują poprawnie.

Można zaproponować kilka funkcjonalnie równoważnych opisów zatrzasku z zanegowanym wyjściem *nQ*, z których najprostszy można zapisać następująco:

```

module latch_if_with_nQ(
    input D,G,
    output reg Q,
    output nQ);

    always @(*)
        if (G) Q = D;
    assign      nQ = ~Q;
endmodule

```

Czytelnikowi proponujemy wykonać syntezę i przeanalizować wyniki symulacji projektu *latch_if_with_nQ*.

6.1.3. Niepoprawny opis zatrzasku D

Zatrask typu D można opisać w języku Verilog bez wykorzystania instrukcji *if*:

```

module not_latch_always(
    input D,G,
    output reg Q);

    always @(*)
        Q = (G & D) | (~G & Q);
endmodule

```

Inny przykład opisu wykorzystuje wyrażenie logiczne zamiast operacji warunkowej:

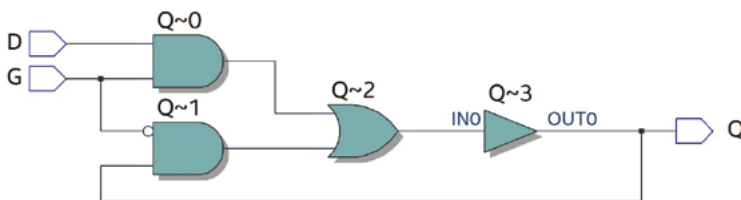
```

module not_latch_assign(
    input D,G,
    output Q);

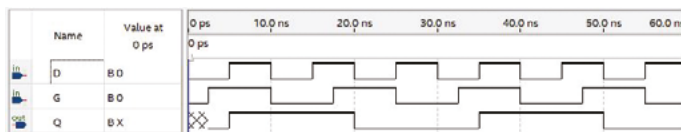
    assign Q = (G & D) | (~G & Q);
endmodule

```

Wyniki syntezy i symulacji modułów z dwóch powyższych opisów pokazano na rysunkach 6.14 i 6.15.



rys. 6.14. Wynik syntezy modułów *not_latch_always* i *not_latch_assign*



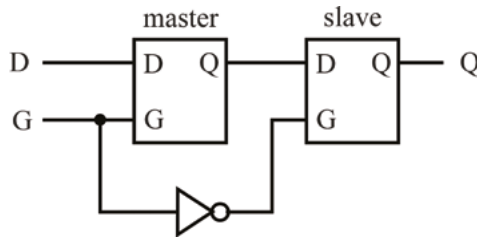
rys. 6.15. Wyniki symulacji modułów *not_latch_always* i *not_latch_assign*

Na rysunku 6.14 można zauważyć, że zaprojektowany schemat nie zawiera prymitywu LATCH, lecz zamiast tego składa się z układu kombinacyjnego ze sprzężeniem zwrotnym. Wyniki symulacji wskazują, że układ działa poprawnie. Jednak taki sposób opisu projektu nie jest zalecany, ponieważ schematy ze sprzężeniem zwrotnym często stanowią potencjalne źródło trudności w odnajdywaniu błędów w projektach na układach FPGA.

Poprawnym opisem zatrasku jest taki, który prowadzi do umieszczenia w schemacie prymitywu LATCH. Oznacza to, że kompilator jest w stanie rozpoznać zatrask w kodzie projektu i wprowadzić odpowiedni prymityw do schematu. Taka praktyka gwarantuje stabilną pracę układu.

6.1.4. Zatrask master-slave

W celu zapewnienia bardziej niezawodnego zapamiętywania sygnałów wejściowych w projektach na FPGA czasami stosuje się zatraski typu *master-slave* (rysunek 6.16).



RYS. 6.16. Struktura zatrzasku typu *master-slave*

Opiszemy zatrzask *master-slave* w języku Verilog z wykorzystaniem instrukcji **if**:

```

module latch_master_slave_if(
    input D,G,
    output reg Q);

    reg c;

    always @(*)
        if (G)          c = D;

    always @(*)
        if (~G)        Q = c;
endmodule

```

W następnym przykładzie opiszemy zatrzask z wykorzystaniem instrukcji **assign**:

```

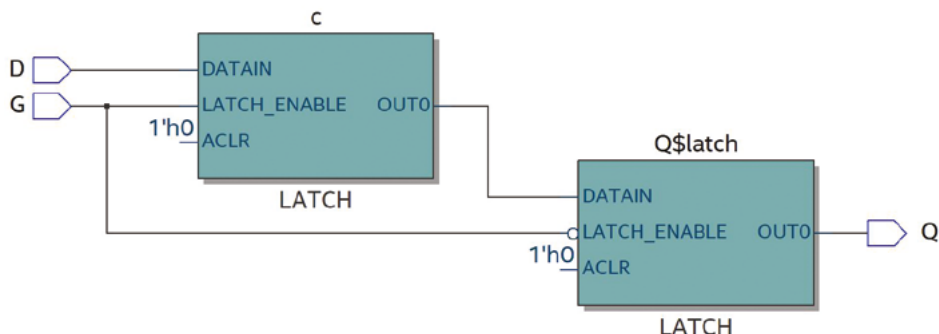
module latch_master_slave_assign(
    input D,G,
    output Q);

    wire c;

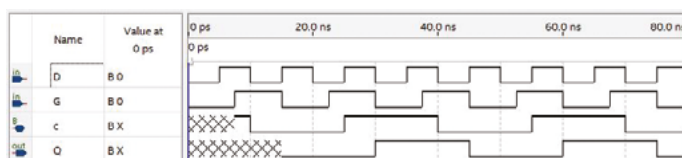
    assign c = (G) ? D : c;
    assign Q = (~G) ? c : Q;
endmodule

```

Oba przedstawione moduły dają identyczne wyniki syntezy i symulacji, które możemy zaobserwować na rysunkach odpowiednio 6.17 i 6.18.



RYS. 6.17. Wynik syntezy zatrasku *master-slave*



RYS. 6.18. Wyniki symulacji zatrasku *master-slave*

6.1.5. Błędne umieszczenie zatrasków na wyjściach układów kombinacyjnych

Jak już wcześniej wspomniano, kompilator może automatycznie umieszczać zatraski na wyjściach układów kombinacyjnych. Przyczyną wprowadzenia niechcianych zatrasków w układach kombinacyjnych mogą być błędy kodowania, takie jak brak konstrukcji **else** w instrukcji **if**, brak konstrukcji **default** w operatorze **case**, gdy w elementach stałych instrukcji **case** nie zostały wymienione wszystkie możliwe wartości zmiennych wejściowych, oraz inne.

Jednakże w niektórych przypadkach, nawet przy zachowaniu reguł prawidłowego kodowania, kompilator może automatycznie umieścić zatraski na wyjściach układu kombinacyjnego. Pokażemy teraz jeden z takich przykładów:

```

module mistaken_latch(
    input s,
    input a1,a2,b1,b2,
    output reg z1,z2,z3);

    always @(*)
        if (s)
            begin
                z1 <= a1;
            end

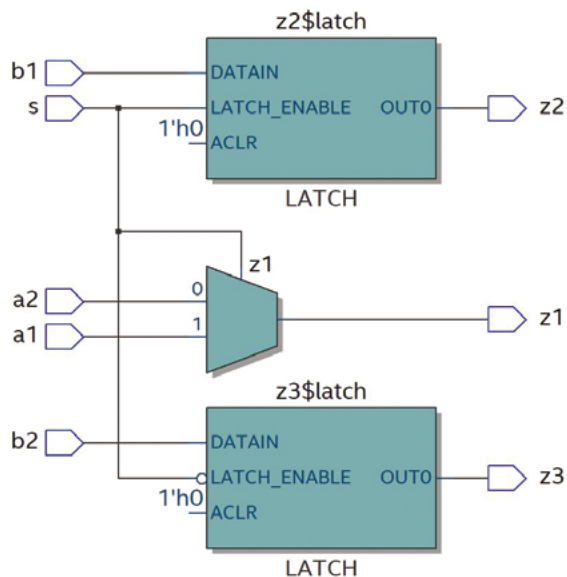
```

```

        z2 <= b1;
    end
    else
    begin
        z1 <= a2;
        z3 <= b2;
    end
endmodule

```

W zaprezentowanym kodzie instrukcja **if** zawiera konstrukcję **else**, co sugeruje poprawność opisu projektu. Jednak wynik syntezy (rysunek 6.19) wykazuje, że na wyjściach *z2* i *z3* zostały automatycznie umieszczone zatrzaski



RYS. 6.19. Umieszczenie zatrzasków na wyjściach układu kombinacyjnego w wyniku błędnie sporządzonego kodu projektu

Jeśli dokładnie przeanalizujemy kod modułu, zauważymy, że wartość wyjścia *z2* została określona tylko dla sygnału *s*=1, natomiast brakuje wskazania wartości na wyjściu dla *s*=0. Analogiczna sytuacja występuje dla wyjścia *z3*, które nie zostało zdefiniowane dla sygnału wejściowego *s*=1. W rezultacie, ponieważ wartości wyjść *z2* i *z3* nie zostały określone dla wszystkich możliwych kombinacji sygnałów wejściowych, na tych wyjściach automatycznie zostały umieszczone zatrzaski.

Wniosek. Zawsze ważne jest dokładne przejrzanie wyników syntezy oraz analiza komunikatów kompilatora, które informują o automatycznym wprowadzeniu do schematu zatrzasków. W każdym przypadku należy dokładnie sprawdzić, czy zatrzask nie został dodany do schematu w wyniku błędnego opisu układu kombinacyjnego.

6.2. Przerzutniki

Przerzutnik (flip-flop) to element pamięciowy, który zmienia swój stan w odpowiedzi na narastające lub opadające zbocze sygnału sterującego. Ten sygnał sterujący nazywany jest *sygnałem zegarowym* lub *sygnałem synchronizacji*, a odpowiednie wejście przerzutnika nazywane jest *wejściem zegarowym* lub *wejściem taktującym*. Przerzutniki mogą być dodatkowo wyposażone w sygnały sterujące, takie jak sygnały ustawiania i zerowania układu, oraz sygnał blokowania sygnału zegarowego. Te sygnały mogą działać synchronicznie lub asynchronicznie względem sygnału zegarowego.

6.2.1. Przerzutnik typu D

Przerzutnik typu D ma jedno wejście danych *D* oraz wyjście *Q*, a także wejście synchronizacji i ewentualnie inne sygnały sterowania. Stan logiczny wyjścia *Q* jest traktowany jako stan przerzutnika. Zmiana sygnału na wejściu synchronizacji powoduje zapamiętanie w przerzutniku wartości z wejścia *D*, co skutkuje ustawieniem tej wartości na wyjściu *Q*.

Przerzutniki w języku Verilog, tak jak zatrzaski, nie są opisywane bezpośrednio, ale są automatycznie generowane na podstawie opisu funkcjonalnego projektu. Ponieważ nie wszystkie opisy przerzutników są syntezywalne, systemy projektowania dostarczają różnych szablonów i prymitywów umożliwiających poprawne dodanie przerzutników do projektu.

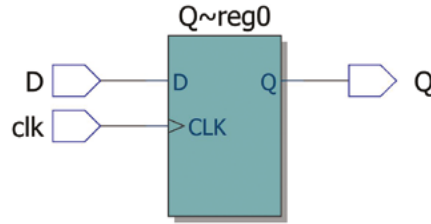
Poniżej przedstawiamy sposób opisu prostego przerzutnika typu D:

```
module my_DFF (output reg Q,      // wyjście
               input clk, D);     // wejścia

    always @(posedge clk)         // lista czułości
        Q <= D;                  // określenie stanu wyjścia
endmodule
```

W przedstawionym przykładzie wykorzystano instrukcję **always**, której lista czułości zawiera słowo kluczowe **posedge**, oznaczające narastające zbocze sygnału synchronizacji *clk*. Przy każdym wystąpieniu tego zdarzenia, czyli pojawienia się narastającego zbocza sygnału *clk*, następuje wykonanie instrukcji przypisania, w którym wyjście *Q* przyjmuje wartość wejścia *D*. Warto zauważyć, że w układach wrażliwych na zbocze sygnału powinno się stosować instrukcję nieblokującego przypisania *<=* języka Verilog.

Wynik syntezy projektu *my_DFF* został pokazany na rysunku 6.20.



RYS. 6.20. Schemat projektu *my_DFF* na poziomie przesłań międzyrejestrowych

Ponieważ instrukcja czułości @ może być stosowana jedynie w instrukcjach proceduralnych, nie jest możliwe opisanie działania przerzutnika za pomocą zwykłej instrukcji przypisania **assign**.

W kolejnym przykładzie pokażemy opis przerzutnika D z dodatkowymi sygnałami zerowania *reset* i ustawienia *set*:

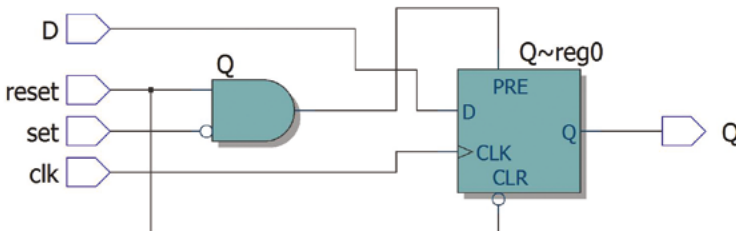
```

module my_DFF_with_control_signals
  (output reg Q,           // wyjście
   input D,               // wejście
   input clk, reset, set); // sygnały sterowania

  always @(posedge clk, negedge reset, negedge set) // lista czułości
  begin // blok always
    if (~reset) Q <= 1'b0; // zerowanie przy niskim poziomie reset
    else if (~set) Q <= 1'b1; // ustawienie przy niskim poziomie set
    else Q <= D;
  end
endmodule

```

Aby uwzględnić reakcję schematu na opadające zbocza sygnałów *reset* i *set*, w liście czułości występuje słowo kluczowe **negedge**, a w wyrażeniach warunkowych sprawdzane są zanegowane wartości sygnałów. Wynik syntezy projektu *my_DFF_with_control_signals* przedstawiono na rysunku 6.21.



RYS. 6.21. Schemat projektu *my_DFF_with_control_signals* na poziomie RTL

Aby zagwarantować poprawność wyników syntezy, należy przestrzegać następujących zasad przy opisie przerzutników w projekcie:

- na liście czułości instrukcji **always** należy wymienić wszystkie sygnały sterowania przerzutnikiem (takie jak sygnał synchronizacji, zerowania i ustawienia);
- pierwszą instrukcją w bloku **always** powinna być instrukcja **if**;
- w wyrażeniu warunkowym instrukcji **if** należy sprawdzić warunki ustawienia i zerowania przerzutnika;
- kolejność sprawdzania warunków ustawienia i zerowania (występowania odpowiednich instrukcji **if**) powinna odpowiadać kolejności, w której sygnały zostały wymienione na liście czułości (w przykładzie *my_DFF_with_control_signals* najpierw sprawdzany jest sygnał *reset*, a następnie *set*);
- wyrażenia warunkowe w operatorach **if** powinny być proste, a zmienne nie powinny mieć indeksów, na przykład:
 if (~reset) lub **if** (reset == 1'b0),
 if (set) lub **if** (set == 1'b1);
- sygnał synchronizacji *clk* jest umieszczany na liście czułości, lecz nie jest sprawdzany w wyrażeniach warunkowych instrukcji **if** w bloku **always**;
- ostatnia instrukcja w bloku **always** powinna określać reakcję (na przykład, $Q \leq D$) na odpowiednie zbocze sygnału synchronizacji (w naszym przykładzie **posedge** *clk*);
- wejścia przerzutnika mogą być sterowane wysokim lub niskim poziomem sygnału, przy aktywnym wysokim poziomie sygnał powinien być opisany na liście czułości ze słowem kluczowym **posedge** (w naszym przykładzie *clk*), przy aktywnym poziomie niskim – ze słowem kluczowym **negedge** (w naszym przykładzie sygnały *reset* i *set*);
- w bloku **always** powinny być wykorzystywane tylko instrukcje nieblokującego przypisania $\leq$.

Sygnał synchronizacji jest automatycznie określany przez syntezytor. Należy uwzględnić ten sygnał na liście czułości, ale nie powinno się go sprawdzać w wyrażeniach warunkowych instrukcji **if** w bloku **always**.

Aby uczynić sygnał sterujący *synchronicznym*, wystarczy usunąć go z listy czułości instrukcji **always**. W takim przypadku wszystkie czynności opisane w bloku **always** będą wykonywane w odpowiedzi na pojawienie się aktywnego zbocza sygnału synchronizacji. Poniższy przykład demonstruje opis przerzutnika D z synchronicznymi sygnałami sterowania:

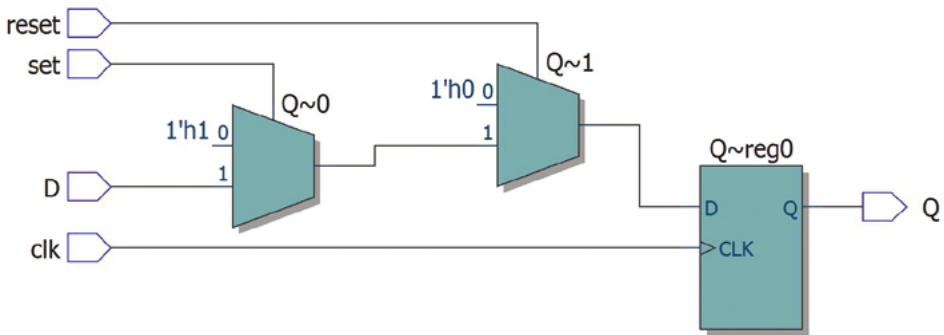
```
module my_DFF_with_sync_control_signals
    (output reg Q,           // wyjście
     input D,               // wejście
     input clk, reset, set); // sygnały sterowania
```

```

always @(posedge clk)           // lista czułości
begin                           // blok always
    if (~reset)   Q <= 1'b0;      // zerowanie przy niskim poziomie reset
    else if (~set) Q <= 1'b1;     // ustawienie przy niskim poziomie set
    else          Q <= D;
end
endmodule

```

Wynik syntezy projektu *my_DFF_with_sync_control_signals* został podany na rysunku 6.22.



RYS. 6.22. Schemat projektu *my_DFF_with_sync_control_signals* na poziomie RTL

Zanegowane wyjście przerzutnika można opisać za pomocą oddzielnej instrukcji przypisania **assign**:

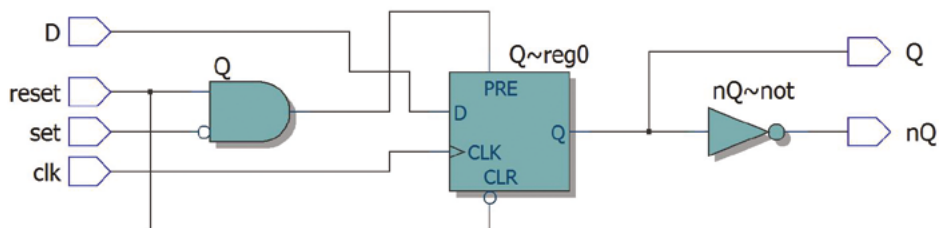
```

module my_DFF_with_nQ
    (output reg Q, output nQ,           // wyjścia
     input D,                          // wejście
     input clk, reset, set);           // sygnały sterowania

    always @(posedge clk, negedge reset, negedge set) // lista czułości
    begin                               // blok always
        if (~reset)   Q <= 1'b0;      // zerowanie przy niskim poziomie reset
        else if (~set) Q <= 1'b1;     // ustawienie przy niskim poziomie set
        else          Q <= D;         // przełączenie stanu
    end
    assign nQ = ~Q;                   // opis zanegowanego wyjścia
endmodule

```

Wynik syntezy projektu *my_DFF_with_nQ* przedstawiono na rysunku 6.23.



RYS. 6.23. Schemat projektu *my_DFF_with_nQ* na poziomie RTL

W przypadku wykorzystania dodatkowego sygnału zezwolenia sygnału zegarowego *ce* (*clock enable*) w przerzutniku należy go opisać jako synchroniczny sygnał sterowania, zgodnie z przedstawionym poniżej kodem:

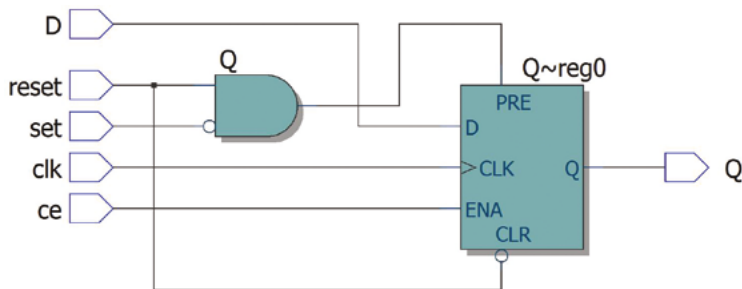
```

module my_DFF_with_ce
    (output reg Q,           // wyjście
    input D,                 // wejście
    input clk, reset, set, ce); // sygnały sterowania

    always @(posedge clk, negedge reset, negedge set) // lista czułości
    begin // blok always
        if (~reset) Q <= 1'b0; // zerowanie niskim poziomem reset
        else if (~set) Q <= 1'b1; // ustawienie niskim poziomem set
        else if (ce) Q <= D; // sprawdzenie stanu sygnału ce
    end
endmodule

```

Wynik syntezy projektu *my_DFF_with_ce* możemy zobaczyć na rysunku 6.24.



RYS. 6.24. Schemat projektu *my_DFF_with_ce* na poziomie RTL

Proponujemy czytelnikowi przeanalizowanie schematów na rysunkach 6.20–6.24 i wyjaśnienie wyników syntezy wykonanej kompilatorem.

6.2.2. Inne typy przerzutników (SR, JK i T)

Oprócz przerzutników typu D, które zostały dokładnie omówione w poprzednim podrozdziale, w projektach systemów wbudowanych wykorzystywane są także przerzutniki innych typów – SR, JK lub T.

Przerzutnik typu SR ma dwa wejścia danych – S i R. Wejście S (*set*) służy do ustawienia przerzutnika w stan 1, natomiast wejście R (*reset*) – do jego zerowania. Funkcjonowanie przerzutnika SR w dużym stopniu przypomina pracę zatrzasku typu SR. Różnica polega na tym, że zmiana stanu zatrzasku następuje w odpowiedzi na zmianę poziomów sygnałów na wejściach S i R, podczas gdy przerzutnik SR jest przełączany zboczem sygnału synchronizacji.

Największym problemem podczas projektowania przerzutnika SR w języku Verilog jest niedozwolona kombinacja sygnałów na wejściach przerzutnika, tj. S=1 i R=1. Po wykryciu takiej kombinacji na wejściach można wygenerować komunikat, jak pokazano w poniższym przykładzie. Jednakże taki komunikat zostanie wyświetlony tylko podczas symulacji, gdyż syntezygnator ignoruje wszystkie komunikaty systemowe dotyczące wyświetlania tekstu na ekranie. W przypadku zabronionej kombinacji wejściowej SR=11 układ z przykładu zachowuje poprzedni stan sygnałów na wyjściach.

```
module my_SRE (output reg Q,  
               input clk, R, S, ce);  
  
    always @(posedge clk)  
    if (ce) begin  
        if (R)  
            if (S) $display(„zabroniona kombinacja na wejściach SR”);  
            else Q <= 1'b0;  
        else  
            if (S) Q <= 1'b1;  
            else Q <= Q;  
        end  
    endmodule
```

Wynik syntezy projektu *my_SRE* został przedstawiony na rysunku 6.25.

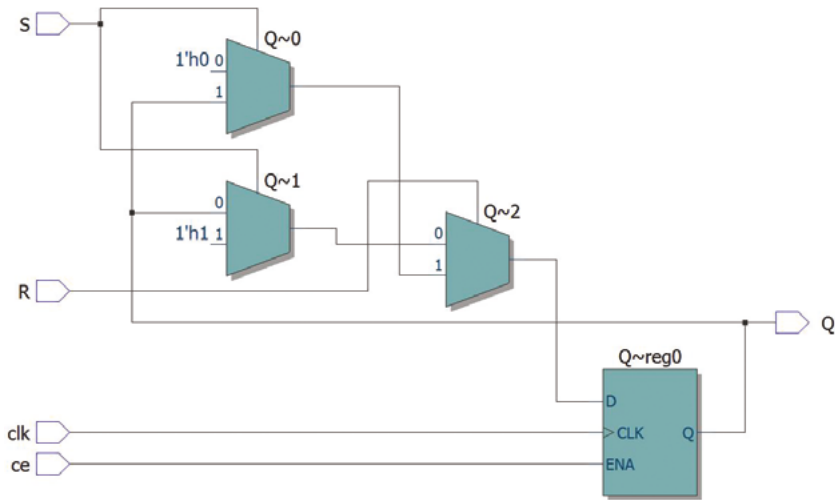
Przerzutnik typu JK jest bardzo podobny do przerzutnika SR, przy czym wejście J odpowiada wejściu S, a wejście K – wejściu R. Istotną różnicą w funkcjonowaniu przerzutnika JK jest to, że w przypadku kombinacji sygnałów wejściowych J=1 i K=1 przerzutnik JK zmienia swój stan na przeciwny. Przedstawiono to w poniższym kodzie:

```
module my_JKE (output reg Q,  
               input clk, K, J, ce);  
  
    always @(posedge clk)
```

```

if (ce) begin
  if (K)
    if (J)  Q <= ~Q;
    else  Q <= 1'b0;
  else
    if (J)  Q <= 1'b1;
    else  Q <= Q;
end
endmodule

```



RYS. 6.25. Schemat projektu *my_SRE* na poziomie przesłań międzyrejestrowych

Wynik syntezy projektu *my_JKE* przedstawiono na rysunku 6.26.

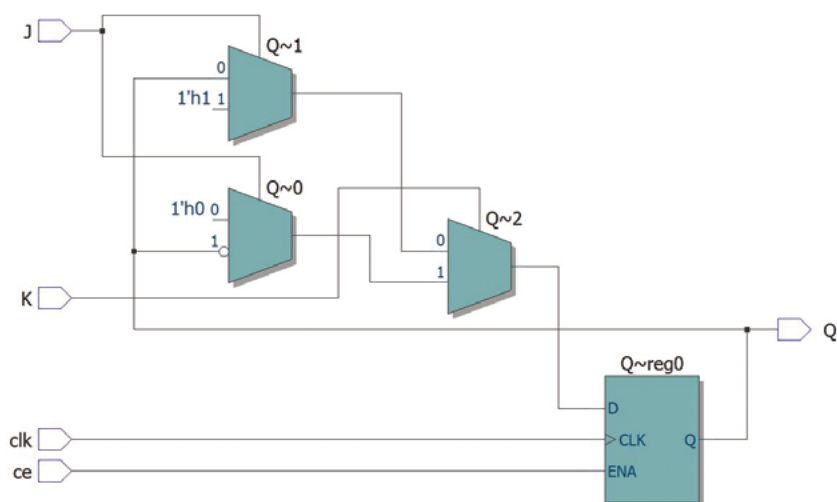
Przerzutnik typu T (*toggle flip-flop*) ma jedno wejście danych *T* oraz wejście synchronizacji. W momencie wystąpienia aktywnego zbocza sygnału zegarowego przy wysokim poziomie sygnału na wejściu *T* przerzutnik zmienia stan wyjść na przeciwny. Natomiast przy niskim poziomie na wejściu *T* wyjście zachowuje poprzedni stan. Poniżej przedstawiono opis przerzutnika typu T:

```

module my_TE (output reg Q,
              input clk, T, ce);

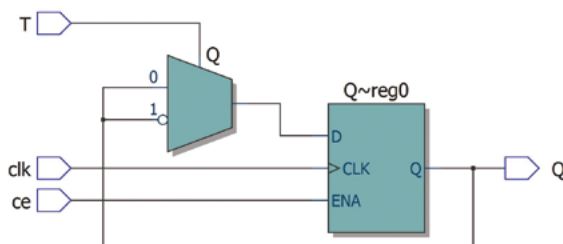
  always @(posedge clk)
    if (ce) begin
      if (T)  Q <= ~Q;
      else  Q <= Q;
    end
endmodule

```



RYS. 6.26. Schemat projektu *my_JKE* na poziomie przesłań międzyrejestrowych

Wynik syntezy modułu *my_TE* pokazany został na rysunku 6.27.



RYS. 6.27. Schemat projektu *my_TE* na poziomie RTL

Proponujemy czytelnikowi przeprowadzenie symulacji funkcjonalnej i czasowej wszystkich omówionych projektów w celu sprawdzenia ich poprawności działania oraz ewentualnego wykrycia błędów.

6.2.3. Określenie stanów początkowych przerzutników

Domyślnie, po włączeniu zasilania, wszystkie przerzutniki i rejestry w układach FPGA firmy Intel są zerowane. Istnieje możliwość ustawienia przerzutników po włączeniu zasilania w stan 1, jednak dla układów, które nie wspierają asynchronicznego ustawienia (*preset*) lub asynchronicznego ładowania (*aload*), wykorzystywana jest dodatkowa logika *NOT gate push-back* (przesunięcia bramki NOT).

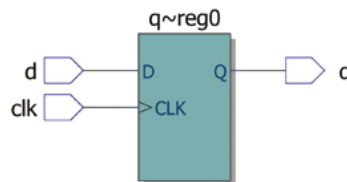
W języku Verilog dla układów FPGA firmy Intel istnieją dwa sposoby ustawienia stanu przerzutników po włączeniu zasilania:

- jawne określenie w kodzie projektu stanu przerzutnika;
- wykorzystanie opcji logicznej *Power-Up Level* systemu Quartus za pomocą atrybutu **altera_attribute**.

Kolejny przykład kodu projektu demonstruje opis przerzutnika typu D, który po włączeniu zasilania ustawiany jest w stan 1:

```
module dff_power_up_1(d,clk,q);  
    input d,clk;  
    output reg q=1'b1;    // określenie początkowej wartości 1 wyjścia q  
  
    always @(posedge clk)  
        q <= d;  
endmodule
```

W podanym przykładzie użyto starego stylu opisu portów modułu, który umożliwia nadanie wartości początkowej wyjściu przerzutnika q . Na rysunku 6.28 pokazany został wygenerowany schemat projektu na poziomie przesłań międzyrejestrów, natomiast na rysunku 6.29 przedstawiono mapę technologiczną projektu.



RYŚ. 6.28. Schemat projektu *dff_power_up_1* na poziomie RTL



RYŚ. 6.29. Mapa technologiczna projektu *dff_power_up_1*

Metoda *NOT gate push-back* polega na umieszczeniu inwerterów na wejściu i wyjściu przerzutnika. Zazwyczaj po włączeniu zasilania przerzutnik jest zerowany, ale z uwagi na inwerter na wyjściu zostanie ustawiony w stan 1.

Kolejny przykład przedstawia opis przerzutnika z ustawieniem w stan 1 po włączeniu zasilania za pomocą atrybutu **altera_attribute**. Schemat projektu na poziomie przesłań międzyrejestrów i mapa technologiczna projektu są identyczne jak dla projektu *dff_power_up_1*.

```

module dff_altera_attribute(d,clk,q);
    input d,clk;
    (* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) output reg q;

    always @(posedge clk)
        q <= d;
endmodule

```

Czytelnikowi proponujemy samodzielne przeprowadzenie symulacji projektów *dff_power_up_1* i *dff_altera_attribute*.

6.3. Rejestry

Rejestry (registers) znajdują szerokie zastosowanie w systemach cyfrowych, często pełniąc funkcję buforów do przechowywania danych wejściowych i wyjściowych systemów wbudowanych. Dodatkowo tworzą one wewnętrzne bufory do przekazywania informacji pomiędzy różnymi blokami systemu. Rejestry stanowią podstawę do budowy wielu układów cyfrowych, takich jak akumulatory, liczniki oraz bloki pamięci. Wszystkie urządzenia realizujące potokowe przetwarzanie danych zazwyczaj zawierają rejestry, które przechowują wyniki realizacji kolejnych etapów potoku.

W języku Verilog [14] rejestry często są kojarzone ze zmiennymi typu **reg**. Jednak nie zawsze zmienna typu **reg** będzie realizowana w schemacie w postaci rejestru. Na przykład gdy układ kombinacyjny jest prawidłowo opisany za pomocą instrukcji proceduralnych z wykorzystaniem zmiennych typu **reg**, do schematu projektu nie będą wprowadzane rejestry.

Tradycyjnie rejestr zbudowany jest z przerzutników typu D, które mają wspólny sygnał synchronizacji oraz sygnały sterowania. Rejestry mogą być także złożone z przerzutników innych typów (SR, JK i T) lub zatrzasków.

Rejestry w języku Verilog są opisywane za pomocą instrukcji **always**, w której obowiązkowo musi być zaznaczony sygnał synchronizacji. Jest on określany automatycznie (jest to sygnał, który musi być umieszczony na liście czułości instrukcji **always**, ale nie występuje w wyrażeniach warunkowych instrukcji **if** w bloku **always**). Przy opisywaniu rejestrów w języku Verilog obowiązują te same zasady, co w przypadku przerzutników.

Do określenia wartości sygnałów na wyjściach rejestru w bloku **always** wykorzystywana jest instrukcja nieblokującego przypisania **<=**. Funkcje sygnałów sterowania opisywane są za pomocą instrukcji **if**. Synchroniczne sygnały sterowania rejestrem nie powinny występować na liście czułości instrukcji **always**. Natomiast asynchroniczne sygnały sterowania wręcz przeciwnie, powinny się znajdować na liście czułości instrukcji **always**. Poniższy kod przedstawia opis w języku Verilog prostego 8-bitowego rejestru:

```

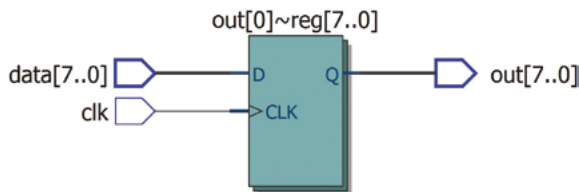
module my_register (input clk,      // wejście synchronizacji
                    input wire [7:0] data,      // wejście danych
                    output reg [7:0] out);      // wyjście

    always @(posedge clk)
        out <= data;

endmodule

```

W przytoczonym kodzie dane z wejścia *data* są przypisywane wyjściu *out* w momencie pojawienia się narastającego zbocza na wejściu synchronizacji *clk*. Wynik syntezy projektu *my_register* pokazany został na rysunku 6.30.



RYS. 6.30. Schemat projektu *my_register* na poziomie przesłań międzyrejestrowych

Jeśli istnieje ryzyko niejednoczesnego przełączenia poszczególnych przerzutników rejestru i powstania wyścigów w systemie cyfrowym, można wykorzystać rejestry zbudowane na *dwutaktowych* (*master-slave*) przerzutnikach. Poniższy kod prezentuje opis dwutaktowego rejestru w języku Verilog:

```

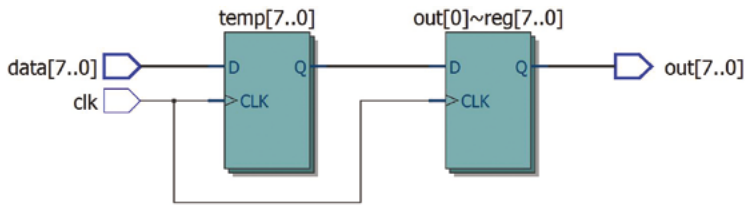
module my_master_slave_register (
    input clk,      // wejście synchronizacji
    input wire [7:0] data, // wejście danych
    output reg [7:0] out); // wyjście
    reg [7:0] temp;      // zmienna pomocnicza

    always @(posedge clk) begin
        temp <= data;
        out <= temp;
    end

endmodule

```

W tym przykładzie dane z wejścia *data* najpierw są zapamiętywane w zmiennej pomocniczej (rejestrze) *temp*, po czym kolejną instrukcją są przypisywane zmiennej *out*. Ponieważ wykorzystywana jest instrukcja nieblokującego przypisania *<=*, rejestry *temp* i *out* łączone są sekwencyjnie [14]. Wynik syntezy projektu *my_master_slave_register* został przedstawiony na rysunku 6.31.



RYS. 6.31. Schemat projektu *my_master_slave_register* na poziomie RTL

W zaprojektowanym module *my_master_slave_register* oba rejestry są sterowane narastającym zboczem tego samego sygnału synchronizacji *clk*. Chcielibyśmy zaproponować czytelnikowi, by zmodyfikował podany przykład w taki sposób, żeby przerzutniki drugiego poziomu były przełączane zanegowanym sygnałem synchronizacji *clk*.

W systemach wbudowanych często występuje specjalny sygnał globalnego resetowania systemu, którego aktywacja powoduje wyzerowanie wszystkich rejestrów systemu. Poniżej zamieszczono opis rejestru z asynchronicznym sygnałem zerowania:

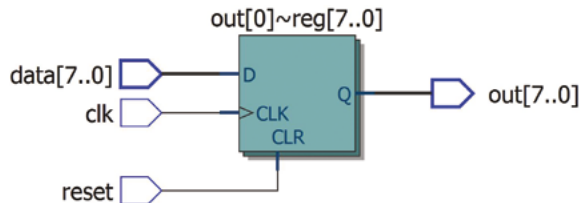
```

module my_register_with_areset (
    input clk, reset,           // sygnały sterowania
    input wire [7:0] data,     // wejście danych
    output reg [7:0] out);     // wyjście

    always @(posedge clk, posedge reset)
        if (reset) out <= 0;
        else out <= data;
endmodule

```

Wynik syntezy projektu *my_register_with_areset* przedstawiono na rysunku 6.32.



RYS. 6.32. Schemat projektu *my_register_with_areset* na poziomie RTL

Czasami konieczne jest wyzerowanie wszystkich przerzutników rejestru (bez wykorzystania globalnego resetowania) lub ustawienie ich w stan 1. W takich przypadkach można skorzystać z synchronicznych sygnałów sterowania rejestrem. Poniższy przykład zawiera opis rejestru z synchronicznymi sygnałami sterowania:

```

module my_register_with_set_and_reset (
    input clk, set, reset,     // sygnały sterowania

```

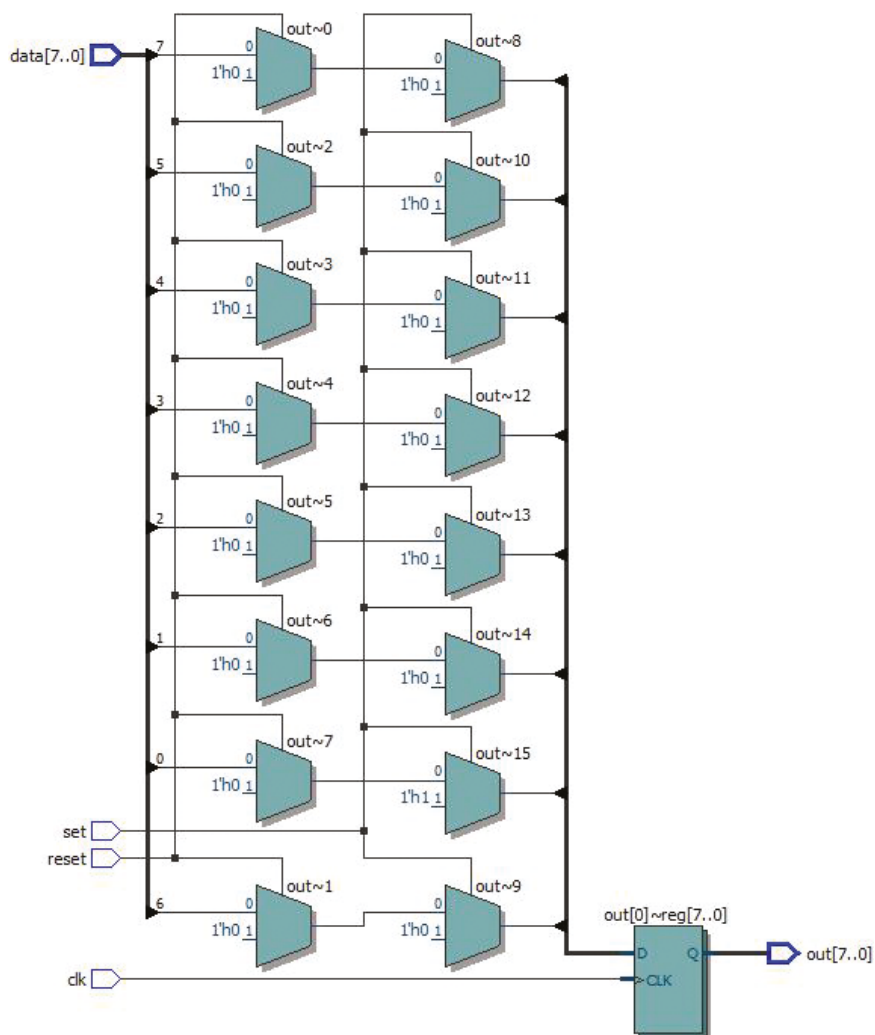
```

input wire [7:0] data,      // wejście danych
output reg [7:0] out;      // wyjście

always @(posedge clk)
  if (set)      out <= 8'b1;
  else if (reset) out <= 8'b0;
  else         out <= data;
endmodule

```

Wynik syntezy modułu *my_register_with_set_and_reset* został pokazany na rysunku 6.33.



RYS. 6.33. Schemat projektu *my_register_with_set_and_reset* na poziomie RTL

Analizując rysunki 6.32 i 6.33, można zauważyć, że asynchroniczne zerowanie rejestru jest w pełni zaimplementowane za pomocą prymitywu systemu Quartus. Natomiast w przypadku synchronicznego zerowania lub ustawiania konieczne jest skorzystanie z dodatkowej logiki FPGA.

Proponujemy czytelnikowi opisanie w języku Verilog projektu rejestru, który będzie zawierał sygnał globalnego resetowania systemowego oraz synchroniczne sygnały zerowania i ustawienia. Dodatkowo sugerujemy modyfikację przedstawionych przykładów w taki sposób, aby rozmiar rejestru można było określić za pomocą parametru.

Zaprezentujemy jeszcze jedną metodę opisu przerzutników (rejestrów) w formie automatu skończonego Moore'a. W języku Verilog taki automat można opisać za pomocą trzech procesów reprezentowanych przez instrukcję **always**. Pierwszy proces odpowiada za przełączenie pamięci automatu skończonego, drugi opisuje logikę kombinacyjną generującą funkcję przejścia do następnego stanu, trzeci zaś implementuje funkcje wyjściowe automatu skończonego. Poniżej przedstawimy przykład projektu *slow_impulses*, który generuje impulsy na wyjściu tylko wtedy, gdy na wejściu sterującym *en* znajduje się sygnał zezwolenia:

```

module slow_impulses (output reg Q,      // wyjście
                      input D, clk, reset, en); // wejścia

    reg    state,      // stan przerzutnika
          next;       // następny stan przerzutnika

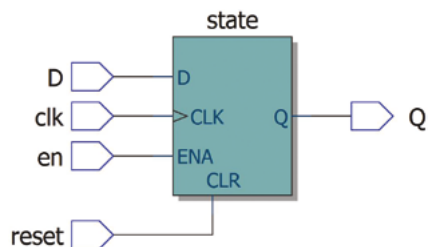
    always @(posedge clk, posedge reset) // pierwszy proces
        if (reset)    state <= 1'b0;
        else         state <= next;

    always @(*)          // drugi proces
        if (en)          next = D;      // określenie
        else            next = state;   // następnego stanu

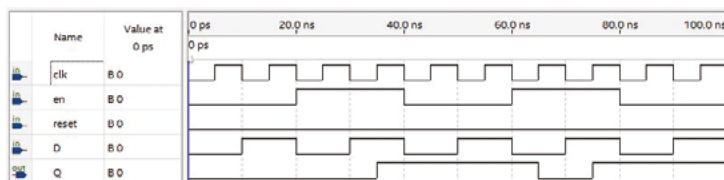
    always @(*)          // trzeci proces
        Q = state;       // stan przerzutnika określa wartość na wyjściu
endmodule

```

Wyniki syntezy i symulacji projektu *slow_impulses* przedstawione zostały na rysunkach 6.34 i 6.35.

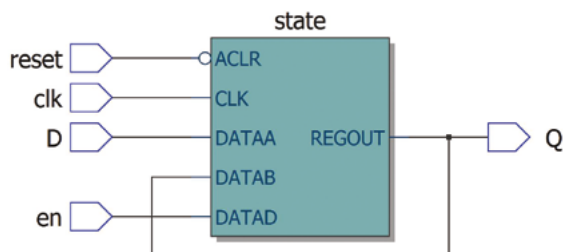


RYS. 6.34. Schemat projektu *slow_impulses* na poziomie RTL



RYS. 6.35. Wyniki symulacji projektu *slow_impulses*

Na rysunku 6.34 widzimy, że do realizacji projektu został wykorzystany jeden przerzutnik, wyposażony w wejście zezwolenia pracy ENA oraz wejście asynchronicznego zerowania CLR. Prezentowany styl opisu w formie automatu skończonego oferuje szeroki zakres możliwości projektowania różnorodnych schematów przerzutników oraz rejestrów. Dlatego też może nas zainteresować mapa technologiczna naszego projektu (rysunek 6.36).



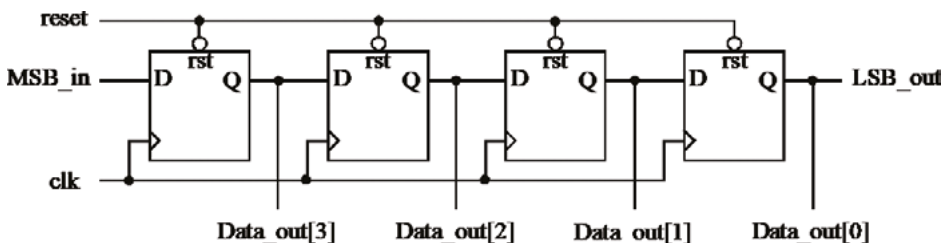
RYS. 6.36. Mapa technologiczna projektu *slow_impulses*

Możemy zauważyć, że w schemacie projektu zostało zrealizowane sprzężenie zwrotne pomiędzy wyjściem prymitywu a wejściem *DATAB*, co umożliwia przekazywanie stanu przerzutnika na wejście schematu. Pamięć wewnętrzna oraz sprzężenie zwrotne pomiędzy wyjściem pamięci a częścią kombinacyjną układu są charakterystycznymi cechami automatów skończonych. Zatem schemat przedstawiony na rysunku 6.36 rzeczywiście odpowiada strukturze automatu skończonego. Różne metody opisu automatów skończonych zostaną szczegółowo omówione w rozdziale 10.

6.4. Rejestry przesuwne

Rejestry przesuwne (shift registers) są wykorzystywane do przechowywania danych oraz do wykonywania operacji przesunięcia na tych danych. Typowym zastosowaniem rejestrów przesuwanych jest ich obecność w układach mnożenia liczb dwójkowych. Ponadto rejestry te doskonale sprawdzają się w konwersji danych z postaci równoległej na szeregową oraz odwrotnie. Dlatego też są one niezastąpionym elementem w systemach transmisji danych, mikrokontrolerach, systemach mikroprocesorowych i innych. Rejestry przesuwne znajdują również zastosowanie w synchronizacji fazowej oraz w systemach testowania układów programowalnych.

Rejestr przesuwany łatwo zbudować na przerzutnikach typu D. Na rysunku 6.37 przedstawiono schemat 4-bitowego rejestru przesuwanego, który wykonuje przesunięcie w prawo. Dane wprowadzane do niego są dostarczane na wejście *MSB_in*, a przesunięte bity pojawiają się na wyjściu *LSB_out*. Aktualną zawartość rejestru można odczytywać z szyny *Data_out*. Przesunięcie danych inicjowane jest narastającym zboczem sygnału synchronizacji *clk*, natomiast wyzerowanie rejestru realizuje się niskim poziomem sygnału na wejściu *reset*.



RYS. 6.37. Schemat 4-bitowego rejestru przesuwanego wykonującego przesunięcie w prawo

Poniżej przedstawiony został opis w języku Verilog sparametryzowanego modułu rejestru przesuwanego realizującego przesunięcie w prawo:

```

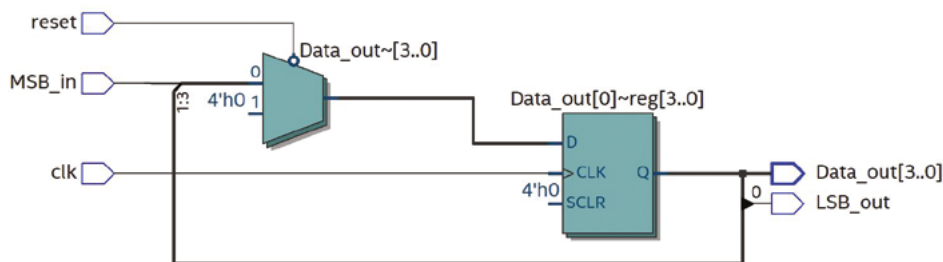
module right_shift_register #(parameter N=4)
    (input MSB_in,
     input clk,reset,
     output reg [N-1:0] Data_out,
     output LSB_out);

    assign LSB_out=Data_out[0];

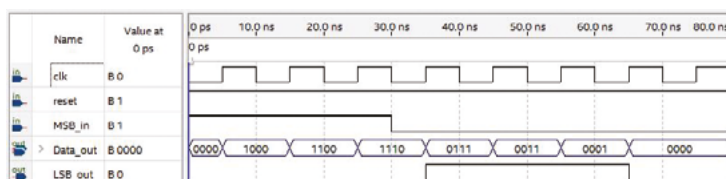
    always @(posedge clk)
        if (~reset) Data_out <= 0;
        else Data_out <= {MSB_in,Data_out[N-1:1]};
endmodule

```

Zwróćmy uwagę, że w podanym kodzie nie jest wykorzystywana operacja przesunięcia języka Verilog. Wyniki syntezy i symulacji modułu *right_shift_register* zostały przedstawione na rysunkach 6.38 i 6.39.



RYS. 6.38. Schemat projektu *right_shift_register* na poziomie RTL



RYS. 6.39. Wyniki symulacji projektu *right_shift_register*

Analogicznie rejestr przesuwany wykonujący operację przesunięcia w lewo można opisać w sposób następujący:

```

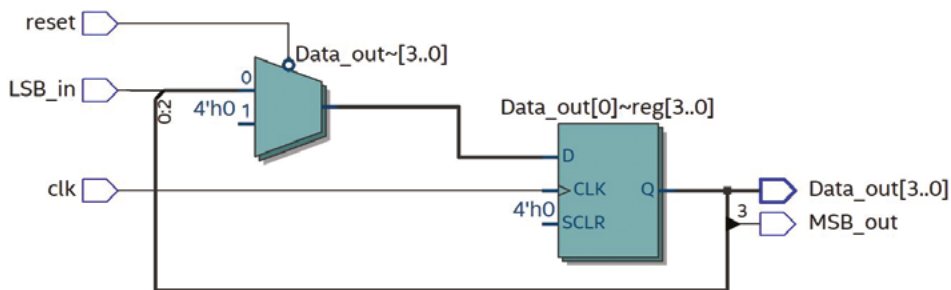
module left_shift_register #(parameter N=4)
    (input LSB_in,
    input clk,reset,
    output reg [N-1:0] Data_out,
    output MSB_out);

    assign MSB_out=Data_out[N-1];

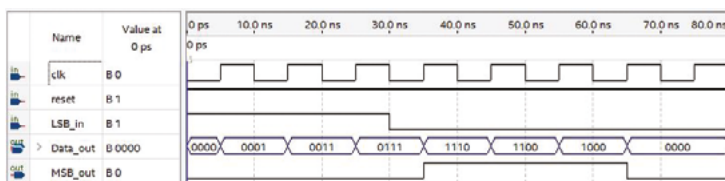
    always @(posedge clk)
        if (~reset)    Data_out <= 0;
        else          Data_out <= {Data_out[N-2:0],LSB_in};
endmodule

```

Wyniki syntezy i symulacji modułu *left_shift_register* pokazane zostały na rysunkach 6.40 i 6.41.



RYS. 6.40. Schemat projektu *left_shift_register* na poziomie RTL



RYS. 6.41. Wyniki symulacji projektu *left_shift_register*

Często konieczne jest wcześniejsze załadowanie danych do rejestru przesuwanego przed wykonaniem operacji przesunięcia. W celu umożliwienia tej funkcji do schematu dodajemy wejście danych *Data_in* oraz sygnał sterujący *load*, który umożliwia równoległe załadowanie danych z szyny *Data_in*. Kod rejestru przesuwanego z możliwością równoległego zapisu danych do rejestru można przedstawić następująco:

```

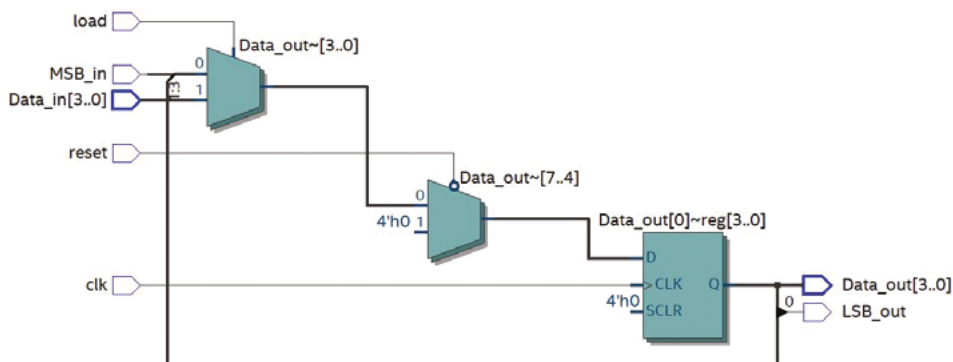
module right_shift_register_load #(parameter N=4)
    (input [N-1:0] Data_in,
     input MSB_in,
     input clk,reset,load,
     output reg [N-1:0] Data_out,
     output LSB_out);

    assign LSB_out=Data_out[0];

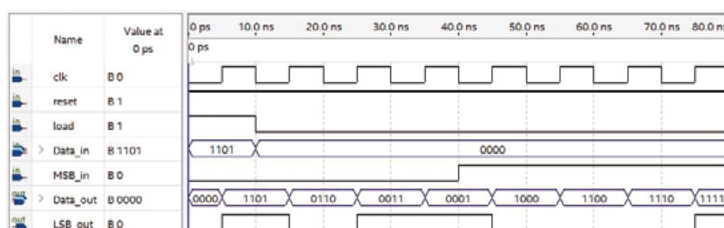
    always @(posedge clk)
        if (~reset)    Data_out <= 0;
        else if (load) Data_out <= Data_in;
        else           Data_out <= {MSB_in,Data_out[N-1:1]};
endmodule

```

Wyniki syntezy i symulacji modułu *right_shift_register_load* przedstawione zostały na rysunkach 6.42 i 6.43.



RYS. 6.42. Schemat projektu *right_shift_register_load* na poziomie RTL



RYS. 6.43. Wyniki symulacji projektu *right_shift_register_load*

Przedstawimy w języku Verilog uniwersalny rejestr przesuwany, który umożliwia przesuwanie danych zarówno w prawo, jak i w lewo. Dodatkowo rejestr ma funkcje załadowania oraz przechowywania danych. W celu wyboru funkcji wykonywanej przez rejestr dodamy do schematu sygnały sterujące *c1* i *c0* (tabela 6.2).

TABELA 6.2. Funkcje uniwersalnego rejestru przesuwanego

<i>c1 c0</i>	Funkcja	Opis
00	hold	przechowywanie danych
01	→	przesunięcie w prawo
10	←	przesunięcie w lewo
11	load	załadowanie danych

Uniwersalny rejestr przesuwany można opisać w języku Verilog za pomocą następującego kodu:

```

module universal_shift_register #(parameter N=4)
    (input [N-1:0] Data_in,
     input MSB_in,LSB_in,
     input clk,reset,
     input [1:0] c,

```

```

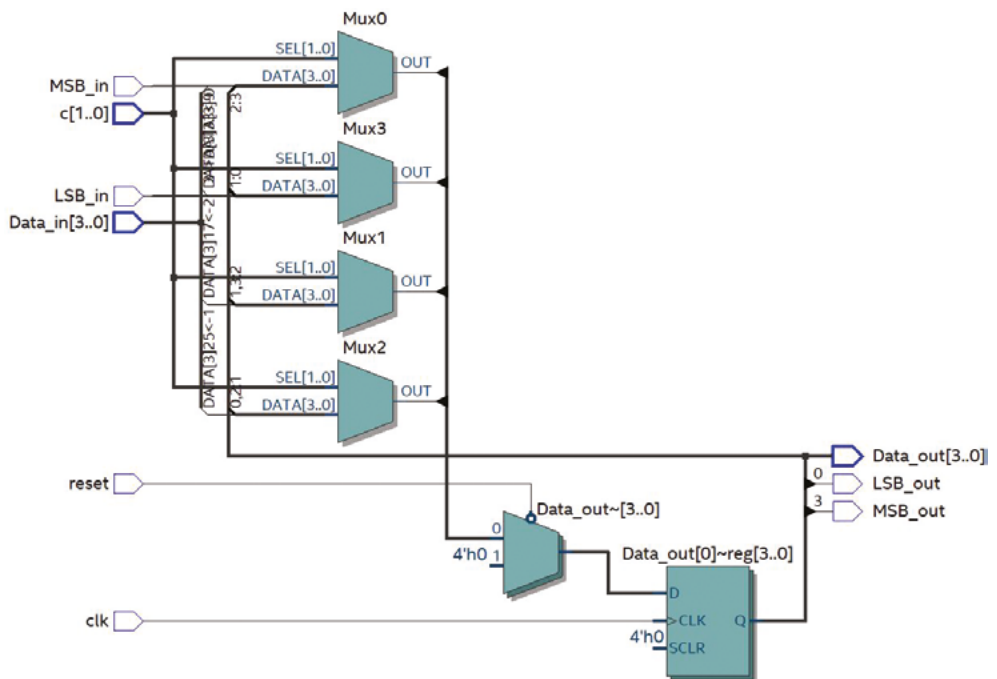
output reg [N-1:0] Data_out,
output MSB_out,LSB_out);

assign LSB_out=Data_out[0];
assign MSB_out=Data_out[N-1];

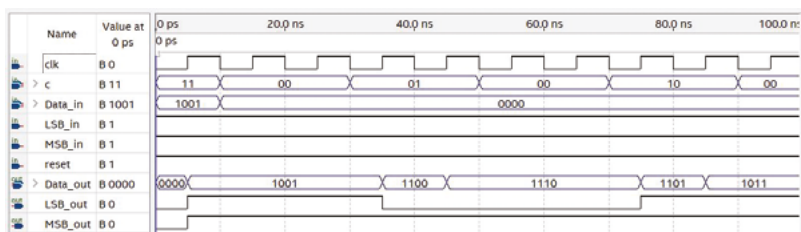
always @(posedge clk)
    if (~reset)    Data_out <= 0;
    else case (c)
        2'b00: Data_out <= Data_out;
        2'b01: Data_out <= {MSB_in,Data_out[N-1:1]};
        2'b10: Data_out <= {Data_out[N-2:0],LSB_in};
        2'b11: Data_out <= Data_in;
    endcase
endmodule

```

Wyniki syntezy i symulacji modułu *universal_shift_register* przedstawiono na rysunkach 6.44 i 6.45.



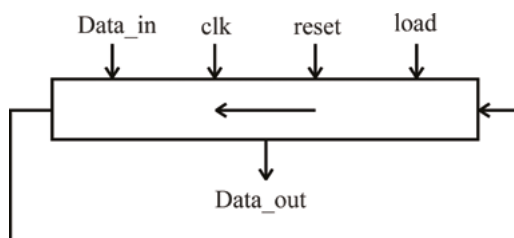
RYS. 6.44. Schemat projektu *universal_shift_register* na poziomie RTL



RYS. 6.45. Wyniki symulacji projektu *universal_shift_register*

6.5. Cykliczne rejestry przesuwne

Jedną z odmian rejestrów przesuwnych są *cykliczne rejestry przesuwne* (*barrel shifters*). Na rysunku 6.46 przedstawiono symbol graficzny stosowany do ich oznaczania na schematach.



RYS. 6.46. Symbol graficzny cyklicznego rejestru przesuwne

Na słowie binarnym zapisanym w cyklicznym rejestrze przesuwne wielokrotnie wykonuje się operację przesunięcia cyklicznego, co prowadzi do generacji powtarzającej się sekwencji kodów dwójkowych. Taka sekwencja może być przydatna do celów testowania lub samotestowania systemów wbudowanych. Opis cyklicznego rejestru przesuwne w języku Verilog może być przedstawiony w następujący sposób:

```

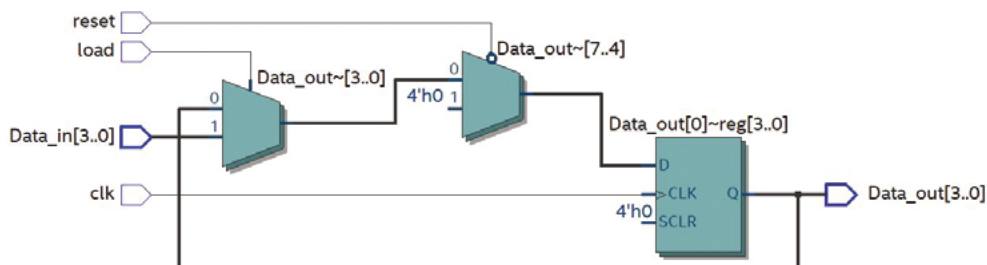
module barrel_shift_register #(parameter N=4)
  (input [N-1:0] Data_in,
   input clk,reset,load,
   output reg [N-1:0] Data_out);

  always @(posedge clk)
    if (~reset) Data_out <= 0;
    else if(load) Data_out <= Data_in;
    else
      Data_out <= {Data_out[N-2:0],Data_out[N-1]};

endmodule

```


Wyniki syntezy i symulacji modułu *barrel_shift_register* możemy zaobserwować na rysunkach odpowiednio 6.47 i 6.48.



RYS. 6.47. Schemat projektu *barrel_shift_register* na poziomie przesłań międzyrejestrowych



RYS. 6.48. Wyniki symulacji modułu *barrel_shift_register*

6.6. Rejestry przesuwne z liniowym sprzężeniem zwrotnym (LFSR)

Przedstawimy jeszcze jedną odmianę rejestrów znanych jako *rejestr przesuwany z liniowym sprzężeniem zwrotnym* (*linear-feedback shift register – LFSR*). Jest on rodzajem rejestru przesuwanego, w którym stan bitu wejściowego jest określany przez funkcję liniową poprzedniego stanu rejestru. Zwykle w postaci funkcji liniowej wykorzystywana jest operacja logiczna XOR. W związku z tym rejestr LFSR często opiera się na operacji XOR stosowanej do wybranych bitów wewnątrz tego rejestru.

Wartości generowane przez rejestr LFSR bezpośrednio zależą od jego poprzedniego stanu. Aby zachować determinizm w jego działaniu, konieczne jest określenie jego stanu początkowego. Początkowa wartość rejestru LFSR nazywana jest *ziarnem* (*seed*). Ponieważ liczba możliwych stanów rejestru jest ograniczona, rejestr LFSR generuje okresowo powtarzającą się sekwencję wartości. W przypadku odpowiednio dobranego wielomianu opisującego funkcję sprzężenia zwrotnego może on generować sekwencję kodów, która jest pseudolosowa i ma długi okres powtarzania.

Rejestry LFSR znajdują zastosowanie w generowaniu liczb pseudolosowych, szumu białego oraz w szybkich licznikach cyfrowych. Ponadto są one wykorzystywane do generowania sekwencji testowych podczas analizy sygnatur oraz do obliczania *sumy kontrolnej CRC* (*cyclic-redundancy check*). Poniższy przykład zawiera kod opisujący 8-bitowy rejestr LFSR:

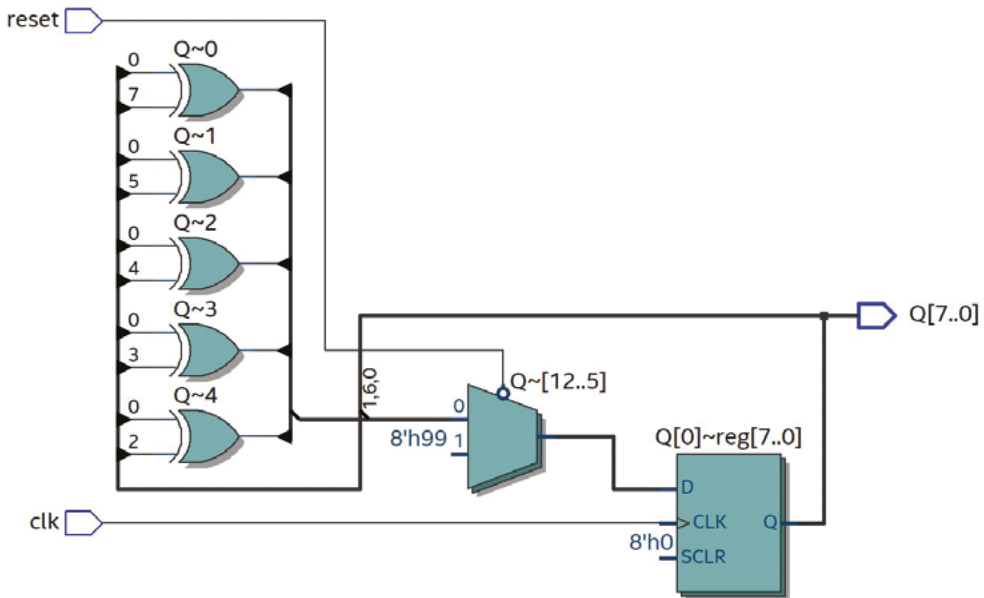
```

module linear_feedback_shift_register
  #(parameter N=8, initial_state=8'b10010001, c=8'b11001111)
  (input clk,reset,
  output reg [N-1:0] Q);

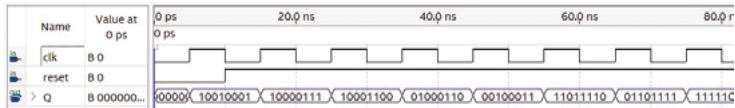
  always @(posedge clk)
    if (~reset) Q <= initial_state;
    else begin
      Q[7] <= Q[0];
      Q[6] <= c[6] ? Q[7]^Q[0]:Q[7];
      Q[5] <= c[5] ? Q[6]^Q[0]:Q[6];
      Q[4] <= c[4] ? Q[5]^Q[0]:Q[5];
      Q[3] <= c[3] ? Q[4]^Q[0]:Q[4];
      Q[2] <= c[2] ? Q[3]^Q[0]:Q[3];
      Q[1] <= c[1] ? Q[2]^Q[0]:Q[2];
      Q[0] <= c[0] ? Q[1]^Q[0]:Q[1];
    end
endmodule

```

W przedstawionym kodzie moduł licznika ma następujące parametry: rozmiar rejestru N , wartość początkowa rejestru *initial_state* oraz współczynniki sprzężenia zwrotnego $c=8'b11001111$. Wynik syntezy rejestru LFSR możemy zobaczyć na rysunku 6.49, a wynik symulacji na rysunku 6.50.



RYS. 6.49. Schemat projektu *linear_feedback_shift_register* na poziomie RTL



RYS. 6.50. Wyniki symulacji projektu *linear_feedback_shift_register*

6.7. Bloki rejestrów

W projektach systemów cyfrowych często wykorzystuje się *bloki rejestrów* (*register file*), które są realizowane w postaci pamięci statycznej RAM. Poniżej przedstawiono kod opisujący blok rejestrów, który składa się z czterech 8-bitowych rejestrów:

```

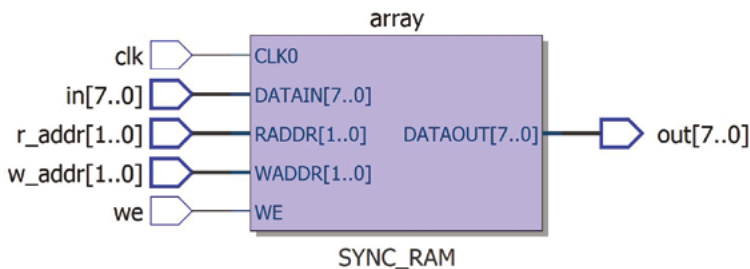
module register_file
    #(parameter B=8,           // rozmiar rejestru
      W=2)                    // liczba bitów adresowych
    (input clk,                // sygnał synchronizacji
      we,                     // zezwolenie zapisu
      input wire [W-1:0] w_addr, // adres do zapisu
      r_addr,                 // adres do odczytu
      input wire [B-1:0] in,   // wejście danych
      output wire [B-1:0] out); // wyjście danych

    reg [B-1:0] array [2**W-1:0]; // tablica rejestrów

    assign out = array[r_addr];    // odczyt z bloku rejestrów
    always @(posedge clk)         // zapis do bloku rejestrów
        if (we) array[w_addr] <= in;
endmodule

```

Wynik syntezy modułu pokazano na rysunku 6.51.



RYS. 6.51. Schemat projektu bloku rejestrów *register_file* na poziomie RTL

6.8. Podsumowanie rozdziału

Układy kombinacyjne charakteryzują się brakiem pamięci, co oznacza, że każda kombinacja zmiennych wejściowych jednoznacznie określa wartości wyjść układu. W układach sekwencyjnych występują elementy pamięci, które służą do zapamiętywania sekwencji kombinacji sygnałów poprzednio podawanych na wejścia schematu. Wartości sygnałów na wyjściach układu sekwencyjnego zależą nie tylko od aktualnego stanu wejść, lecz także od stanu pamięci.

Najprostszymi elementami pamięciowymi są zatrzaski i przerzutniki, które umożliwiają przechowywanie pojedynczego bitu informacji. *Zatrząsek (latch)* to element pamięciowy, który jest sterowany *poziomym sygnałem* na wejściu zegarowym. *Przerzutnik (flip-flop)* jest przełączany za pomocą *zbocza sygnału* zegarowego – może to być zarówno *narastające (dodatnie)*, jak i *opadające (ujemne)* zbocze.

Specyficzną cechą projektowania zatrząsek i przerzutników w języku Verilog jest to, że nie można ich opisywać jawnie w kodzie projektu. Są one automatycznie wprowadzane do schematu przez kompilator na podstawie kodu projektu.

Producenci układów FPGA nie zalecają stosowania zatrząsek w projektach na FPGA. Wykorzystywanie zatrząsek powinno być ograniczone do sytuacji, gdy nie można ich zastąpić przerzutnikami lub rejestrami.

Kompilator automatycznie dodaje zatrzaski na wyjściach układów kombinacyjnych w przypadku, gdy w kodzie projektu nie są opisane wszystkie możliwe kombinacje wejściowe sygnałów oraz odpowiadające im wartości zmiennych na wyjściach. Zatrzaski zawsze są dodawane do schematu w przypadku braku konstrukcji **else** w instrukcji **if** oraz gdy w instrukcji **case** brakuje konstrukcji **default** i nie zostały wymienione wszystkie możliwe wartości stałych elementów. W niektórych sytuacjach kompilator może automatycznie dodać zatrzaski do projektu nawet wtedy, gdy każdemu operatorowi **if** przypisana jest konstrukcja **else**.

Aby kompilator automatycznie dodał zatrząsek do schematu, muszą być spełnione dwa warunki: musi istnieć przynajmniej jedna kombinacja zmiennych wejściowych, dla której nie zostały określone wartości wyjść, oraz lista wrażliwości instrukcji **always** nie powinna zawierać zdarzeń wywoływanych zboczem sygnału (czyli słów kluczowych **posedge** i **negedge**).

Poprawnym opisem zatrząsku jest taki, który powoduje umieszczenie w schemacie prymitywu LATCH. Oznacza to, że kompilator identyfikuje zatrząsek w kodzie projektu i dodaje odpowiedni prymityw do schematu.

W celu zapewnienia bardziej niezawodnego zapamiętywania sygnałów wejściowych stosuje się zatrzaski typu *master-slave*.

Wejście przerzutnika, na które podawany jest sygnał synchronizacji, jest nazywane *wejściem zegarowym* lub *wejściem taktowania*. Przerzutniki mogą również zawierać dodatkowe sygnały sterujące, które kontrolują ustawienie i zerowanie układu, oraz sygnał blokowania sygnału synchronizacji. Te sygnały mogą działać synchronicznie lub asynchronicznie względem sygnału zegarowego.

Rejestr (register) składa się z przerzutników typu D, które mają wspólny sygnał synchronizacji oraz sygnały sterowania. Rejestry mogą być także zbudowane z przerzutników innych typów (SR, JK i T) lub zatrzasków.

Rejestry w języku Verilog są opisywane za pomocą instrukcji **always**, w której obowiązkowo musi być określony sygnał synchronizacji. Ma on być umieszczony na liście czułości instrukcji **always**, ale nie powinien występować w wyrażeniach warunkowych instrukcji **if** w bloku **always**.

Asynchroniczne sygnały sterowania przerzutnikiem lub rejestrem powinny być uwzględnione na liście czułości instrukcji **always**. Aby sygnał sterowania zrobić synchronicznym, wystarczy usunąć go z listy czułości instrukcji **always**.

W układach wrażliwych na zbocze sygnału należy stosować instrukcję nieblokującego przypisania $\leq$ języka Verilog. Do określenia na liście czułości narastającego zbocza sygnału używane jest słowo kluczowe **posedge**, natomiast dla opadającego zbocza – **negedge**.

Aby zapewnić poprawne wyniki syntezy, przy opisie przerzutników i rejestrów w projekcie należy przestrzegać określonych zasad. W języku Verilog dla układów FPGA firmy Intel istnieją dwa sposoby ustawienia stanu przerzutników po włączeniu zasilania: jawne określenie w kodzie projektu stanu przerzutnika oraz wykorzystanie opcji logicznej *Power-Up Level* systemu Quartus za pomocą atrybutu **altera_attribute**.

W przypadku, gdy istnieje ryzyko niejednoczesnego przełączania poszczególnych przerzutników w rejestrze, co może prowadzić do wyścigów w systemie cyfrowym, stosuje się rejestry zbudowane na *dwutaktowych (master-slave)* przerzutnikach.

Przerzutniki i rejestry mogą być opisywane jako automaty skończone typu Moore'a.

Rejestry przesuwne (shift registers) służą do przechowywania danych oraz do przeprowadzania operacji przesunięcia na nich. Jedną z odmian rejestrów przesuwnych są *cykliczne rejestry przesuwne (barrel shifters)*.

Rejestr przesuwny z liniowym sprzężeniem zwrotnym (linear-feedback shift register – LFSR) jest rejestrem przesuwным, w którym bit wejściowy jest funkcją liniową jego poprzedniego stanu. Zwykle w postaci funkcji liniowej wykorzystywana jest operacja logiczna XOR. Dlatego też w większości przypadków LFSR wykorzystuje operację XOR do przetwarzania wybranych bitów rejestru. Początkowa wartość rejestru LFSR nazywana jest *ziarnem (seed)*. Rejestr LFSR z właściwie dobranym wielomianem opisującym funkcję sprzężenia zwrotnego jest zdolny do generowania sekwencji kodów, które są pseudolosowe i mogą mieć długi okres powtarzania.

Blok rejestrów (register file) to zbiór rejestrów zorganizowanych w postaci niewielkiej pamięci statycznej RAM.

7. Liczniki, schematy opóźnień, dzielniki częstotliwości, czasomierze, zegary czasu rzeczywistego

W niniejszym rozdziale omawiane są zagadnienia związane z projektowaniem liczników na układach FPGA przy użyciu języka Verilog. Oprócz licznych przykładów projektów przedstawione zostaną również wyniki badań efektywności różnych opisów liczników realizowanych na układach FPGA. Omówione zostaną różne rodzaje liczników: dwójkowe, modulo n , rewersyjne, Graya i Johnsona. Dodatkowo liczniki mogą stanowić podstawę do tworzenia schematów opóźnień.

Typowe systemy cyfrowe mogą być sterowane dowolnym sygnałem zegarowym, który niekoniecznie musi być zsynchronizowany z czasem rzeczywistym. Jednakże w przypadku systemów wbudowanych zwykle sterują one urządzeniami pracującymi w czasie rzeczywistym. Dlatego też istotne staje się dokładne mierzenie czasu w skali rzeczywistej, a nie tylko abstrakcyjnej. W tym celu wykorzystuje się *czasomierze (timery)*. Dodatkowo, w systemach wbudowanych często konieczne jest wprowadzenie opóźnień czasowych dla niektórych sygnałów. Wspomóc w tym mogą specjalne *schematy opóźnień (delay circuits)*.

Zazwyczaj systemy wbudowane operują przy wysokich częstotliwościach zegara systemowego. Na przykład na płycie DE1-SoC generowany jest sygnał zegarowy o częstotliwości 50 MHz. Niemniej jednak większość urządzeń peryferyjnych działa na znacznie niższych częstotliwościach. Aby skutecznie sterować tymi wolniejszymi układami peryferyjnymi, niezbędne są dzielniki częstotliwości, generatory wolnych sygnałów zegarowych oraz schematy wprowadzające opóźnienia czasowe, mierzone w mikrosekundach (us), milisekundach (ms) i sekundach (s).

W tym rozdziale omówione zostaną także bloki funkcjonalne, które bezpośrednio współpracują z systemem synchronizacji projektu oraz mają szerokie zastosowanie w systemach wbudowanych: schematy opóźnień, dzielniki częstotliwości, generatory niskich częstotliwości, powolne liczniki, czasomierze oraz zegary czasu rzeczywistego.

7.1. Liczniki

Licznik (counter) to układ sekwencyjny, który cyklicznie przechodzi przez określoną niezmienną sekwencję swoich stanów. Liczniki, w których kody kolejnych stanów odpowiadają liczbom dwójkowym, nazywane są *licznikami dwójkowymi*. Długość M sekwencji stanów licznika określana jest jako *moduł* lub *podstawa zliczania*. Dla liczników dwójkowych maksymalna wartość modułu wynosi 2^N , gdzie N to liczba bitów licznika.

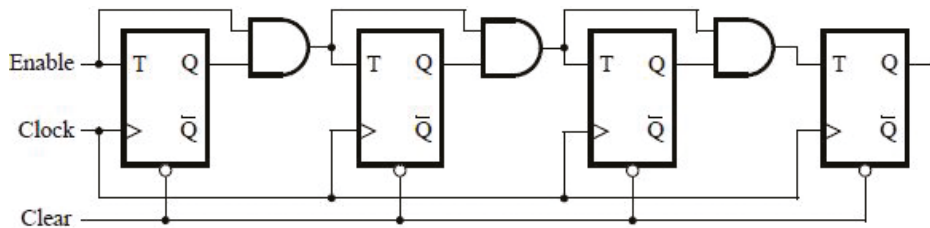
Liczniki są niezwykle wszechstronne w systemach wbudowanych i należą do jednych z najbardziej powszechnych bloków funkcjonalnych. Istnieje wiele różnych rodzajów liczników, co wynika z ich szerokiego zakresu zastosowań. Ich podstawowym zadaniem jest zliczanie, obejmujące między innymi liczenie impulsów zegarowych lub innych zdarzeń, oraz rejestrowanie impulsów odebranych od innych urządzeń. Liczniki odgrywają także istotną rolę w systemach sterowania czasem rzeczywistym. Dodatkowo często stanowią bazę do implementacji dzielników częstotliwości sygnałów zegarowych.

Zazwyczaj wejściem licznika dwójkowego jest wejście sygnału zegarowego, nazywane także *wejściem zliczającym*. W niektórych projektach liczników wejście zliczające i wejście synchronizacji mogą być oddzielone. Wyjściem licznika jest N -bitowa szyna, na której generowane są kolejno zliczane wartości w postaci liczb dwójkowych. Dodatkowo liczniki zazwyczaj mają różne sygnały sterujące, takie jak asynchroniczne (*aclr*) lub synchroniczne (*sclr*) zerowanie, sygnał zezwolenia pracy (*enable*), asynchroniczne (*aload*) bądź synchroniczne (*sload*) załadowanie oraz inne. Liczniki mogą również dysponować szyną danych wejściowych (*data*), za pomocą której przekazywane są dane ładowane do licznika.

Liczniki mogą prowadzić zliczanie w górę (*up*) albo w dół (*down*), a istnieją także liczniki *rewersyjne*, które umożliwiają wybór kierunku zliczania. Często liczniki wyposażone są w wyjście *rollover*, znane też jako wyjście *przepełnienia licznika*. Jest ono ustawiane na 1, gdy licznik osiąga największą możliwą wartość do zliczania.

7.1.1. Strukturalny opis licznika

Za tradycyjną uważana jest struktura licznika zbudowana z przerzutników typu T (rysunek 7.1).



RYS. 7.1. Struktura 4-bitowego licznika na przerzutnikach typu T

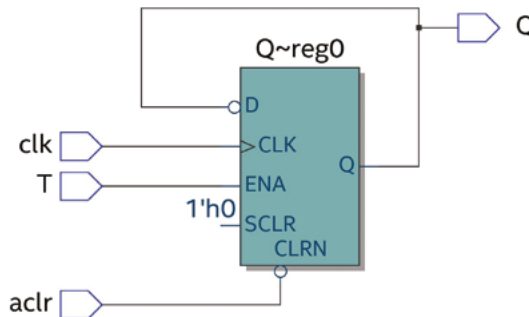
Do zaprojektowania licznika przedstawionego na rysunku 7.1 będzie nam potrzebny moduł przerzutnika typu T, którego opis w języku Verilog można przedstawić następująco:

```
module FFT_aset(
    input T,clk,aclr,
    output reg Q);

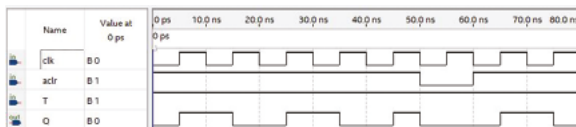
    always @(posedge clk, negedge aclr)
        if (!aclr)    Q <= 1'b0;
        else if (T)   Q <= ~Q;
        else         Q <= Q;

endmodule
```

Projekt przedstawia przerzutnik z wejściem danych *T*, wejściem sygnału zegarowego *clk*, wejściem asynchronicznego zerowania *aclr*, a także wyjściem *Q*. Wyniki syntezy i symulacji modułu *FFT_aset* pokazano na rysunkach 7.2 i 7.3.



RYS. 7.2. Wynik syntezy przerzutnika T z asynchronicznym zerowaniem



RYS. 7.3. Wyniki symulacji przerzutnika T z asynchronicznym zerowaniem

Zauważmy, że wynik syntezy jest zgodny z tradycyjną realizacją przerzutnika T na podstawie przerzutnika typu D.

Mając gotowy projekt przerzutnika T, możemy opisać strukturę 4-bitowego licznika z asynchronicznym zerowaniem *aclr* i sygnałem zezwalającym pracę *enable*:

```
module counter_T_4_bits(
    input clk,aclr,enable,
```



```
output [3:0] q;
```

```
wire [3:1] c;
```

```
assign c[1] = q[0] & enable;
```

```
assign c[2] = q[1] & c[1];
```

```
assign c[3] = q[2] & c[2];
```

```
FFT_areset ex0(enable,clk,aclr,q[0]);
```

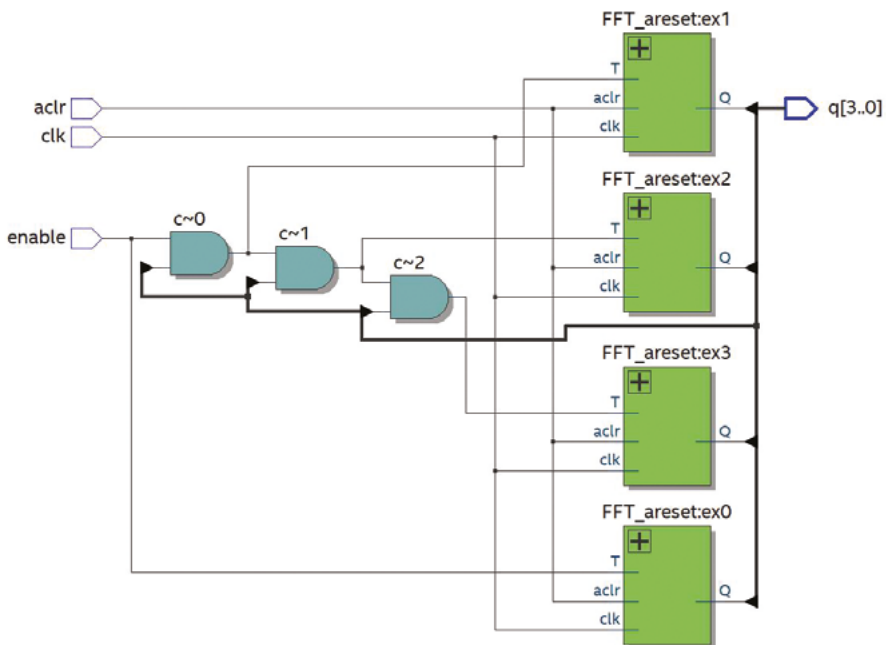
```
FFT_areset ex1(c[1],clk,aclr,q[1]);
```

```
FFT_areset ex2(c[2],clk,aclr,q[2]);
```

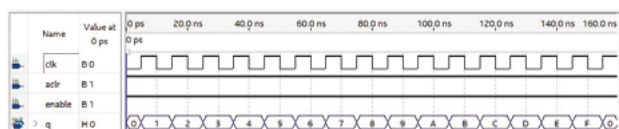
```
FFT_areset ex3(c[3],clk,aclr,q[3]);
```

```
endmodule
```

Wyniki syntezy i symulacji funkcjonalnej modułu licznika *counter_T_4_bits* zostały przedstawione na rysunkach odpowiednio 7.4 i 7.5.



RYS. 7.4. Wynik syntezy 4-bitowego licznika zbudowanego z przerzutników T



RYS. 7.5. Wyniki symulacji 4-bitowego licznika zbudowanego z przerzutników T

7.1.2. Behawioralny opis licznika

Licznik można także opisać za pomocą instrukcji proceduralnych, bazując na algorytmie jego działania. W języku Verilog liczniki zazwyczaj są opisywane z wykorzystaniem instrukcji **if** oraz konstrukcji **else if**, w których kolejno analizowane są stany sygnałów sterujących. Czynności wykonywane przez licznik są zazwyczaj opisane na końcu sekwencji instrukcji **else if** i **else**.

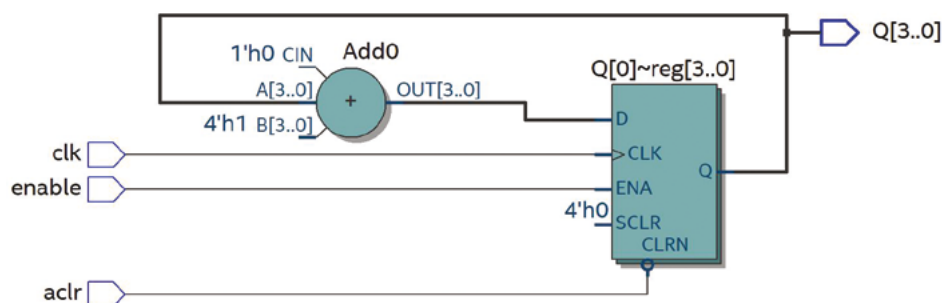
W poniższym przykładzie kodu przedstawiono behawioralny opis sparametryzowanego N -bitowego licznika z asynchronicznym zerowaniem *aclr* oraz wejściem sygnału zezwolenia na pracę *enable*.

```
module counter_N_bits
  #(parameter N=4)
  (input clk,aclr,enable,
  output reg [N-1:0] Q);

  always @(posedge clk, negedge aclr)
    if (!aclr)      Q <= {N{1'b0}};
    else if (enable) Q <= Q + 1'b1;
    else           Q <= Q;

endmodule
```

Wynik syntezy modułu *counter_N_bits* pokazano na rysunku 7.6, a wyniki symulacji są zgodne z przedstawionymi na rysunku 7.5.



RYS. 7.6. Wynik syntezy sparametryzowanego N -bitowego licznika dla $N=4$

Schemat licznika na rysunku 7.6 składa się z rejestru oraz sumatora. Sumator zwiększa zawartości rejestru o 1, a wynik tej operacji jest zapisywany do rejestru po kolejnym impulsie sygnału synchronizacji. Sygnały sterujące licznika zostały połączone z odpowiednimi sygnałami sterowania rejestrem.

7.1.3. Liczniki modulo M

Wcześniej zaprojektowane liczniki nazywane są *pełnymi*, ponieważ generują na swoich wyjściach wszystkie możliwe 2^N wartości – od 0 do 2^N-1 . Jednakże często konieczne jest zaprojektowanie *licznika modulo M* , gdzie $M < 2^N$. W poniższym kodzie przedstawiono sparametryzowany licznik modulo M z domyślną wartością parametru $M = 16$, z sygnałem asynchronicznego zerowania *aclr* oraz wejściem zezwalającym na zliczanie *enable*.

```
module counter_mod_M
    #(parameter M=16)
    (input clk,aclr,enable,
     output reg [N-1:0] Q);

    localparam N=clogb2(M-1);

    function integer clogb2(input [31:0] v);
        for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
            v = v >> 1;
    endfunction

    always @(posedge clk, negedge aclr)
        if (!aclr)           Q <= {N{1'b0}};
        else if (Q == M-1)   Q <= {N{1'b0}};
        else if (enable)     Q <= Q + 1'b1;
        else                 Q <= Q;
endmodule
```

W module *counter_mod_M* wykorzystano *funkcję stałą* [14] *clogb2* do obliczenia liczby bitów licznika N na podstawie wartości modułu M . Funkcja stała *clogb2* odpowiada następującej funkcji matematycznej:

$$N = \text{int}(\log_2 M), \quad (7.1)$$

gdzie $\text{int}(X)$ – najmniejsza liczba całkowita większa lub równa X .

Wynik syntezy modułu *counter_mod_M* przedstawiono na rysunku 7.7, a wyniki symulacji są takie same jak na rysunku 7.5.

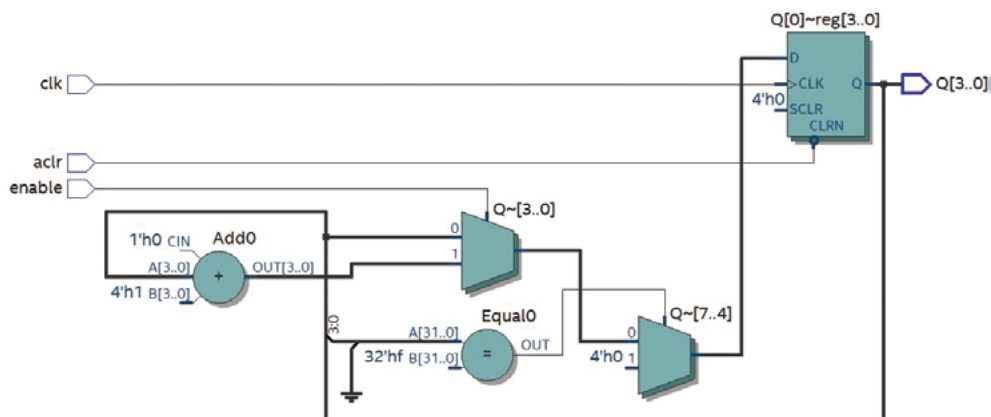
Na podstawie sparametryzowanego licznika modulo M łatwo jest zaprojektować licznik o dowolnej wartości M . Na przykład licznik modulo 10 można opisać następująco:

```
module counter_mod_10(
    input clk,aclr,enable,
```

```
output [3:0] Q);
```

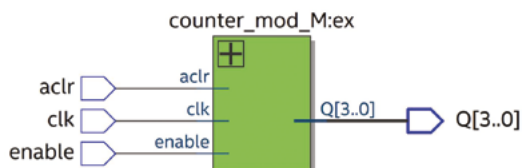
```
counter_mod_M #(10) ex(clk,aclr,enable,Q);
```

```
endmodule
```

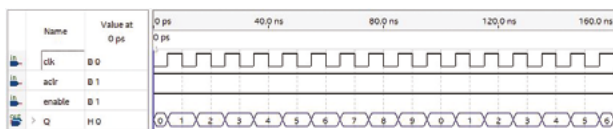


RYS. 7.7. Wynik syntezy licznika modulo M dla $M=16$

Wyniki syntezy oraz symulacji projektu *counter_mod_10* pokazane zostały na rysunkach 7.8 oraz 7.9.



RYS. 7.8. Wynik syntezy licznika modulo 10 zaprojektowanego na podstawie sparametryzowanego modułu *counter_mod_M*



RYS. 7.9. Wyniki symulacji licznika modulo 10

Licznik może mieć dodatkowe wyjście *rollover*. Na wyjściu zostaje ustawiony stan wysoki *rollover*=1, gdy licznik osiągnie wartość $M-1$. Poniżej zaprezentowano opis takiego licznika:

```

module counter_modulo_M_rollover
    #(parameter M=16)
    (input clk,aclr,enable,
    output reg [N-1:0] Q,
    output rollover);

    localparam N=clogb2(M-1);

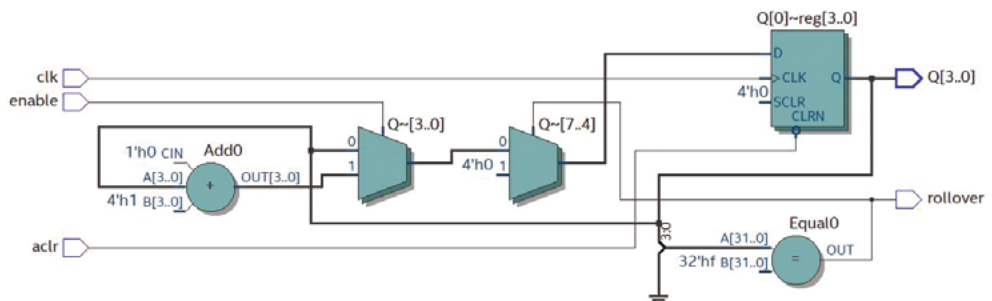
    function integer clogb2(input [31:0] v);
        for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
            v = v >> 1;
    endfunction

    always @(posedge clk, negedge aclr)
        if (!aclr)      Q <= {N{1'b0}};
        else if (Q == M-1)  Q <= {N{1'b0}};
        else if (enable)    Q <= Q + 1'b1;
        else                Q <= Q;

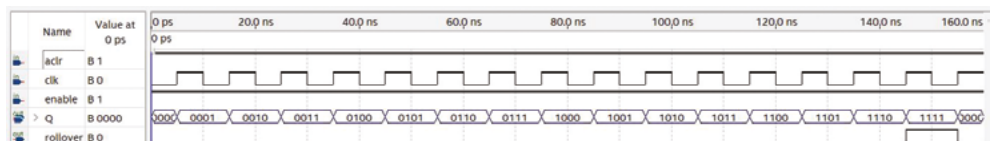
    assign rollover = (Q == M-1);
endmodule

```

Wyniki syntezy i symulacji projektu *counter_modulo_M_rollover* można zobaczyć na rysunkach 7.10 i 7.11.



RYS. 7.10. Wynik syntezy licznika modulo M z wyjściem *rollover* dla $M=16$



RYS. 7.11. Wyniki symulacji licznika z wyjściem *rollover*

7.1.4. Ładowanie danych do licznika

Łaďadowanie wartości początkowej do licznika umoŹliwia zrealizowanie zliczania określonej liczby zdarzeń K , co jest często wymagane w systemach wbudowanych. MoŹe to być osiągnięte przez ładowanie wartości K do licznika odejmującego i zliczania do zera lub przez ładowanie wartości $M-K$ do licznika dodającego i zliczania do wartości maksymalnej.

PoniŹszy kod zawiera opis licznika modulo M z wyjściem *rollover* i asynchronicznym ładowaniem danych.

```
module counter_mod_M_rollover_aload
    #(parameter M=16)
    (input clk,aclr,enable,aload,
    input [N-1:0] data,
    output reg [N-1:0] Q={N{1'b0}},
    output rollover);

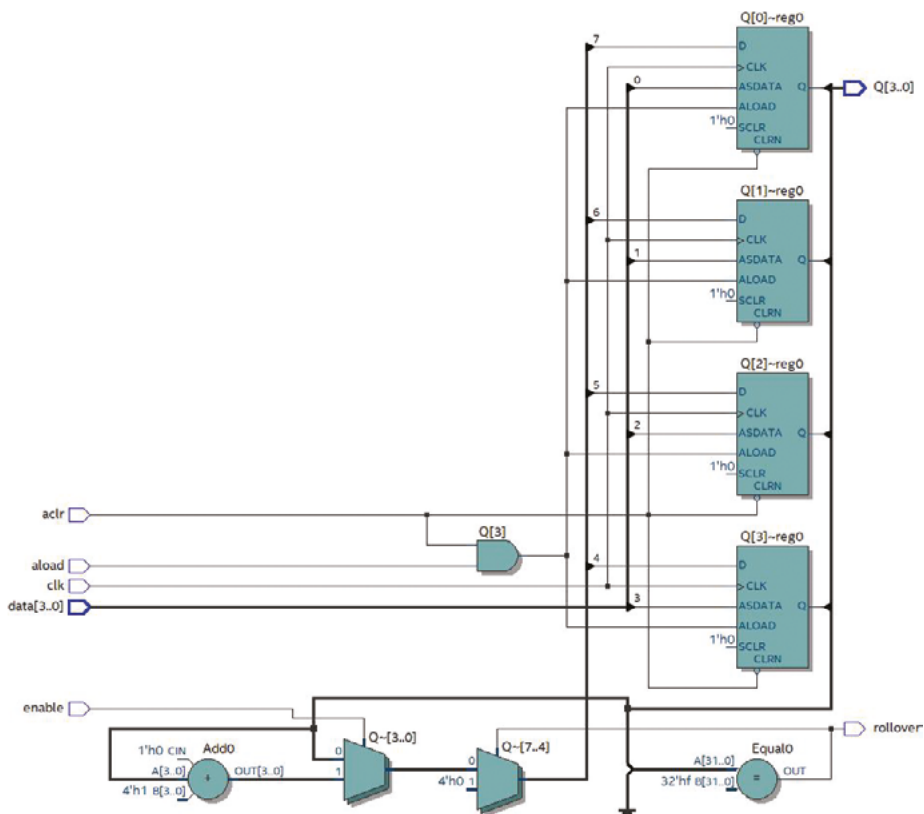
    localparam N=clogb2(M-1);

    function integer clogb2(input [31:0] v);
        for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
            v = v >> 1;
    endfunction

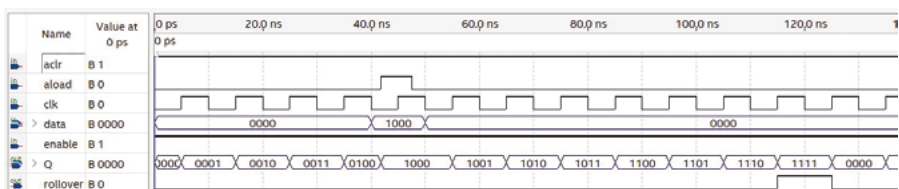
    always @(posedge clk, negedge aclr, posedge aload)
        if (!aclr)      Q <= {N{1'b0}};
        else if (aload)  Q <= data;
        else if (Q == M-1)  Q <= {N{1'b0}};
        else if (enable)  Q <= Q + 1'b1;
        else            Q <= Q;

    assign rollover = (Q == M-1);
endmodule
```

W module *counter_mod_M_rollover_aload* dodano wejście danych *data* oraz sygnał *aload* zezwalający na asynchroniczne ładowanie danych do licznika. Wyniki syntezy i symulacji projektu *counter_mod_M_rollover_aload* przedstawiono na rysunkach 7.12 i 7.13.



RYS. 7.12. Wynik syntezy licznika z asynchronicznym ładowaniem danych



RYS. 7.13. Wyniki symulacji licznika z asynchronicznym ładowaniem danych

W schemacie przedstawionym na rysunku 7.12 do asynchronicznego ładowania danych są wykorzystywane wejścia przerzutników: *ASDATA* – dla danych i *ALOAD* – do sterowania ładowaniem.

Ładowanie danych do liczników może także być wykonywane synchronicznie. W tym celu wystarczy usunąć z listy czułości instrukcji **always** sygnał sterowania *aload*. Poniższy kod opisuje licznik z synchronicznym ładowaniem danych:

```
module counter_mod_M_rollover_sload
    #(parameter M=16)
```

```

(input clk,aclr,enable,sload,
input [N-1:0] data,
output reg [N-1:0] Q={N{1'b0}},
output rollover);

localparam N=clogb2(M-1);

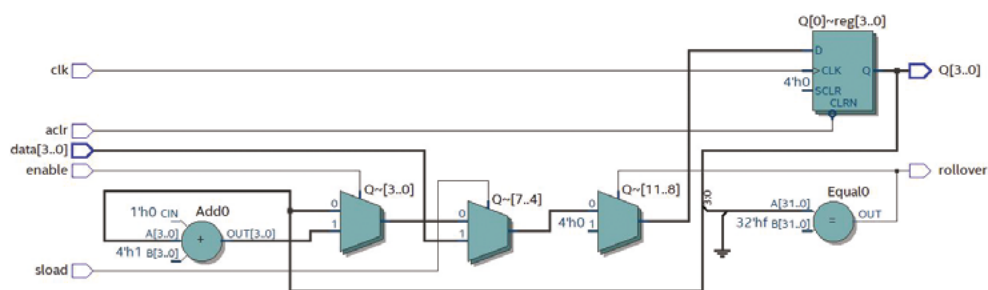
function integer clogb2(input [31:0] v);
    for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
        v = v >> 1;
endfunction

always @(posedge clk, negedge aclr)
    if (!aclr)      Q <= {N{1'b0}};
    else if (Q == M-1) Q <= {N{1'b0}};
    else if (sload)  Q <= data;
    else if (enable) Q <= Q + 1'b1;
    else            Q <= Q;

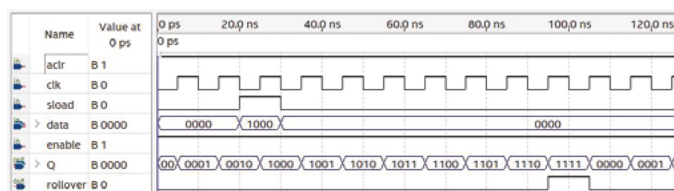
    assign rollover = (Q == M-1);
endmodule

```

Wyniki syntezy i symulacji projektu *counter_mod_M_rollover_sload* zaprezentowano na rysunkach 7.14 i 7.15.



RYS. 7.14. Wynik syntezy licznika z synchronicznym ładowaniem danych



RYS. 7.15. Wyniki symulacji licznika z synchronicznym ładowaniem danych

W projekcie na rysunku 7.14 do synchronicznego ładowania danych do licznika wykorzystuje się wejścia *D* przerzutników, a sterowanie ładowaniem danych realizuje się za pomocą układu logicznego zbudowanego z multiplexerów.

W kolejnym przykładzie zostanie zaprojektowany licznik z sygnałem synchronicznego zerowania. Aby zmienić asynchroniczny sposób zerowania licznika na synchroniczny, wystarczy usunąć sygnał sterujący *aclr* z listy czułości instrukcji **always**:

```
module counter_mod_M_rollover_sload_sclr
    #(parameter M=16)
    (input clk,sclr,enable,sload,
     input [N-1:0] data,
     output reg [N-1:0] Q={N{1'b0}},
     output rollover);

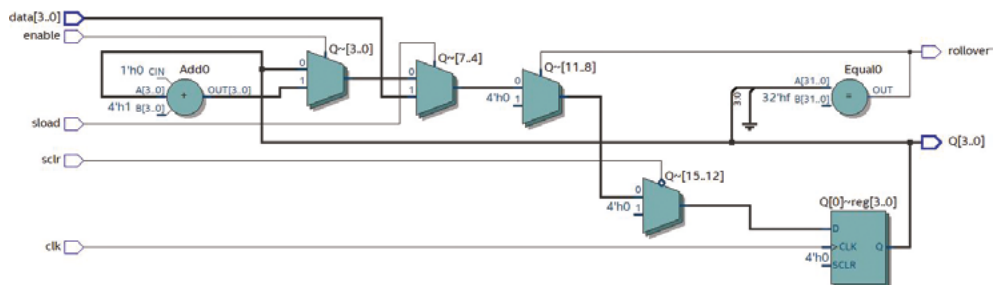
    localparam N=clogb2(M-1);

    function integer clogb2(input [31:0] v);
        for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
            v = v >> 1;
    endfunction

    always @(posedge clk)
        if (!sclr)      Q <= {N{1'b0}};
        else if (Q == M-1)  Q <= {N{1'b0}};
        else if (sload)    Q <= data;
        else if (enable)  Q <= Q + 1'b1;
        else              Q <= Q;

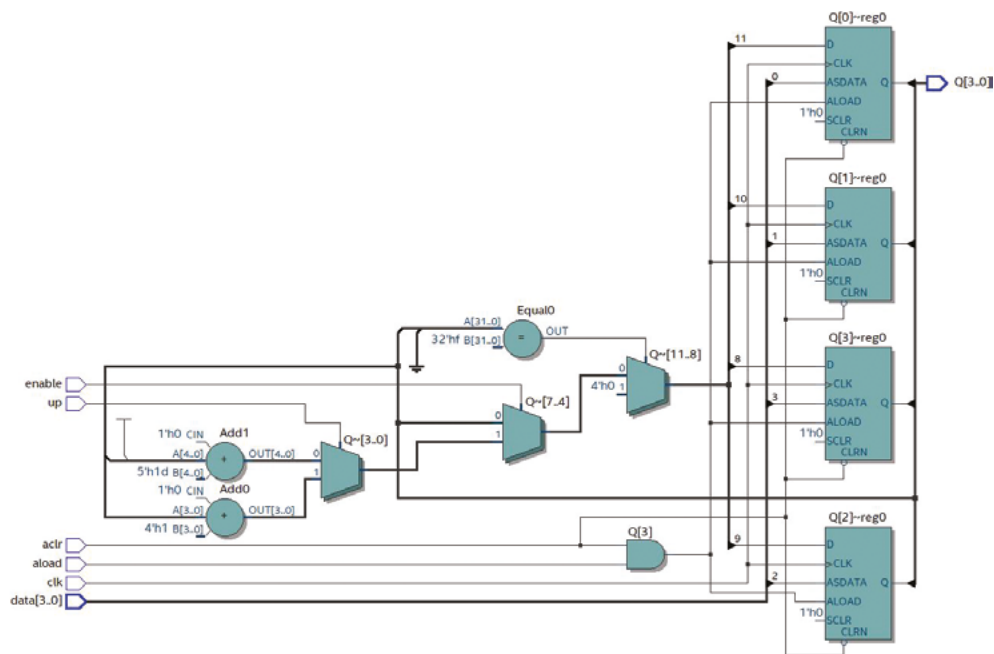
    assign rollover = (Q == M-1);
endmodule
```

Wyniki syntezy i symulacji projektu *counter_mod_M_rollover_sload* możemy zaobserwować na rysunkach 7.16 i 7.17.

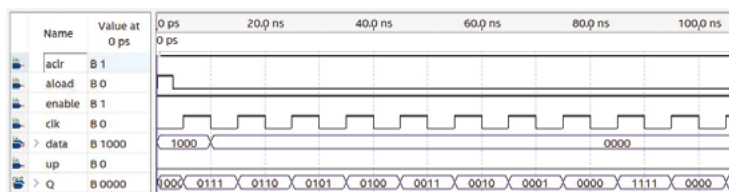


RYS. 7.16. Wynik syntezy licznika z synchronicznym ładowaniem danych i synchronicznym zerowaniem

W projekcie wybór kierunku zliczania dokonuje się za pomocą sygnału sterującego *up*. Gdy *up*=1, licznik zlicza w górę, a gdy *up*=0, zlicza w dół. Jednakże w danym opisie nie uwzględniono sprawdzania, czy licznik osiągnął wartość *M*–1 lub zero. Dlatego sprawdzanie to powinno być wykonane poza licznikiem. Wyniki syntezy i symulacji modułu *counter_up_down_aload* zostały przedstawione na zamieszczonych poniżej rysunkach 7.18 oraz 7.19.



RYS. 7.18. Wynik syntezy licznika rewersyjnego z asynchronicznym ładowaniem danych



RYS. 7.19. Wyniki symulacji licznika rewersyjnego z asynchronicznym ładowaniem danych

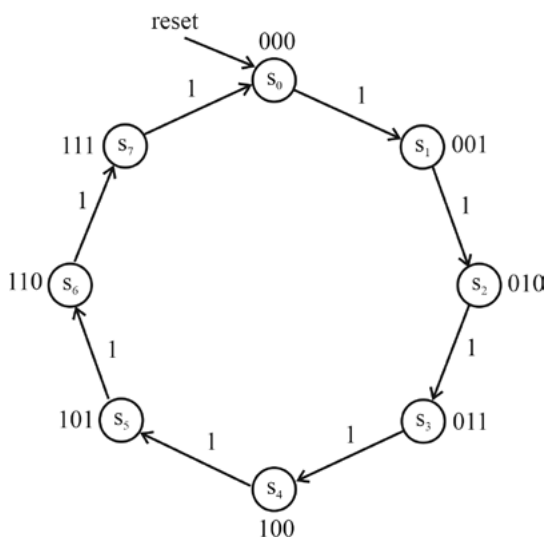
Na wykresach czasowych wyników symulacji projektu na rysunku 7.19 obserwujemy, że najpierw licznik zostaje załadowany wartością 1000, po czym rozpoczyna zliczanie w dół.

W schemacie licznika na rysunku 7.18 znajdują się dwa sumatory. Postaramy się opisać licznik rewersyjny w taki sposób, aby w schemacie występował tylko jeden sumator. W tym celu przy określaniu wartości wyjścia *Q* zamiast instrukcji *if* wykorzystamy instrukcję warunkową, jak w poniższym kodzie:

Wyniki symulacji modułu *counter_up_down_aload_2* są dokładnie takie same, jak dla projektu *counter_up_down_aload* (rysunek 7.19).

7.1.6. Opis licznika w postaci automatu skończonego

Ponieważ automat skończony jest modelem matematycznym dowolnego układu sekwencyjnego, licznik również może być opisany w postaci automatu skończonego. W tym celu licznik można przedstawić jako zamkniętą sekwencję stanów automatu skończonego Moore'a, gdzie następuje bezwarunkowe przejście pomiędzy stanami. Liczba stanów automatu jest równa wartości modułu M licznika, a wartości zmiennych wyjściowych przypisane poszczególnym stanom odpowiadają wartościom sygnałów generowanych na wyjściach licznika. Na rysunku 7.21 przedstawiono graf automatu skończonego licznika binarnego modulo 8.



RYS. 7.21. Opis licznika w postaci grafu automatu skończonego

Poniższy kod zawiera opis 4-bitowego licznika modulo $M=16$ w postaci automatu skończonego klasy C (klasy automatów skończonych opisane zostały w rozdziale 10).

```

module counter_4_bits_FSM (
    input clk, reset,           // sygnał zegarowy i sygnał zerowania
    output [3:0] Q);           // wyjście licznika

    reg [3:0] state, next;      // bieżący i następny stan

    localparam [3:0]

```

```

s0=4'b0000, s1=4'b0001, s2=4'b0010, s3=4'b0011,
s4=4'b0100, s5=4'b0101, s6=4'b0110, s7=4'b0111,
s8=4'b1000, s9=4'b1001, s10=4'b1010, s11=4'b1011,
s12=4'b1100, s13=4'b1101, s14=4'b1110, s15=4'b1111;

```

```

always @(*)           // określenie przejść

```

```

    case (state)

```

```

        s0: next=s1;
        s1: next=s2;
        s2: next=s3;
        s3: next=s4;
        s4: next=s5;
        s5: next=s6;
        s6: next=s7;
        s7: next=s8;
        s8: next=s9;
        s9: next=s10;
        s10: next=s11;
        s11: next=s12;
        s12: next=s13;
        s13: next=s14;
        s14: next=s15;
        s15: next=s0;

```

```

    endcase

```

```

always @(posedge clk) // przełączenie stanów

```

```

    if (!reset) state <= s0;
    else         state <= next;

```

```

assign Q=state;           // określenie wartości wyjść

```

```

endmodule

```

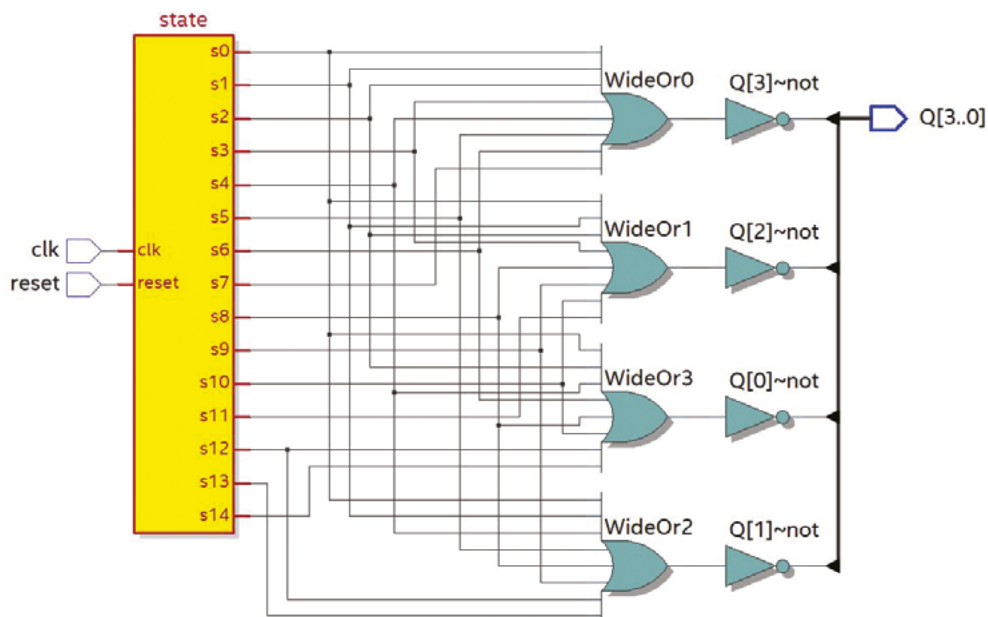
W podanym kodzie zadeklarowane zostały dwie zmienne pomocnicze – *state* i *next*. Zmienna *state* odpowiada bieżącemu stanowi automatu skończonego, natomiast zmienna *next* wskazuje na następny stan, do którego ma miejsce przejście. Kody stanów są określane za pomocą parametrów lokalnych *s0–s15*. Opis automatu skończonego zawiera dwa procesy wyznaczone za pomocą instrukcji **always** oraz jeden instrukcją przypisania ciągłego **assign**.

Pierwszy proces definiuje przejścia automatu skończonego. Przejścia pomiędzy stanami są opisywane za pomocą instrukcji **case**, gdzie sprawdzana jest wartość zmiennej *state*, a elementy instrukcji **case** określają stany, do których ma miejsce przejście. Drugi proces steruje przełączaniem między stanami, co następuje w momencie narastającego zbocza sygnału zegarowego *clk*.

Ponieważ w automacie klasy C kody stanów wewnętrznych dokładnie odpowiadają wartościom wyjść automatu skończonego (co jest cechą charakterystyczną wszystkich liczników), nie jest potrzebna realizacja schematu kombinacyjnego do generowania sygnałów wyjściowych na podstawie kodu stanu automatu. Dlatego zamiast tradycyjnego trzeciego procesu w opisie automatu Moore'a w naszym przykładzie umieszczamy zwykłą instrukcję przypisania ciągłego **assign**, która generuje wartości wyjść zgodne z kodem bieżącego wewnętrznego stanu automatu.

Aby umożliwić kodowanie stanów automatu określone przez użytkownika w kodzie projektu, należy w parametrach syntezy systemu Quartus: **Assignments > Settings... > Advanced Settings (Synthesis)... > State Machine Processing** wybrać opcję **User-Encoded**.

Wynik syntezy projektu *counter_4_bits_FSM* został przedstawiony na rysunku 7.22.

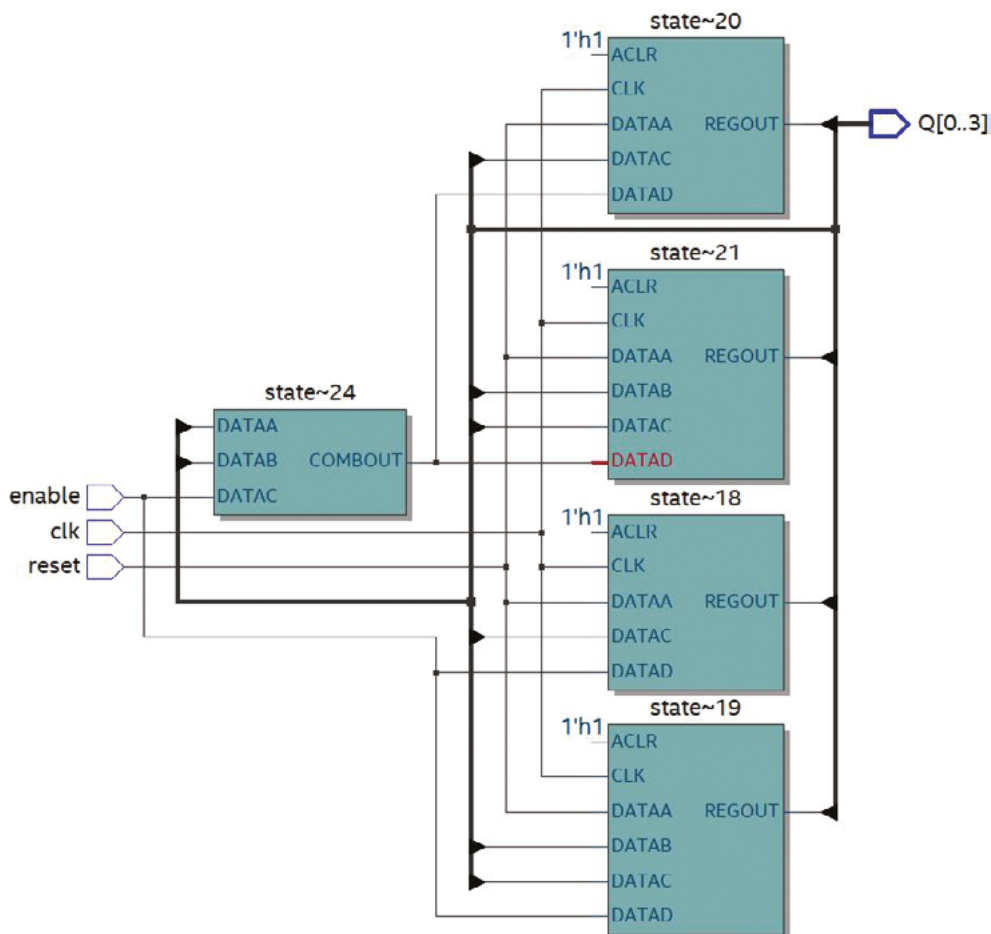


RYS. 7.22. Wynik syntezy 4-bitowego licznika opisanego w postaci automatu skończonego klasy C

Na rysunku 7.22 widzimy, że schemat oprócz automatu skończonego *state* zawiera także logikę generowania sygnałów wyjściowych na podstawie kodu stanu automatu, chociaż taki schemat logiczny nie powinien występować w automacie klasy C.

Można to wyjaśnić tym, że przy syntezie automatów skończonych na układach FPGA domyślnie wykorzystywane jest kodowanie unarne (*one-hot*), gdzie każdy stan jest identyfikowany ustawieniem na 1 tylko jednego przerzutnika. Schemat na rysunku 7.22 właśnie odpowiada takiemu kodowaniu.

Aby sprawdzić, jaki schemat został rzeczywiście wygenerowany, należy obejrzeć mapę technologiczną projektu (rysunek 7.23).

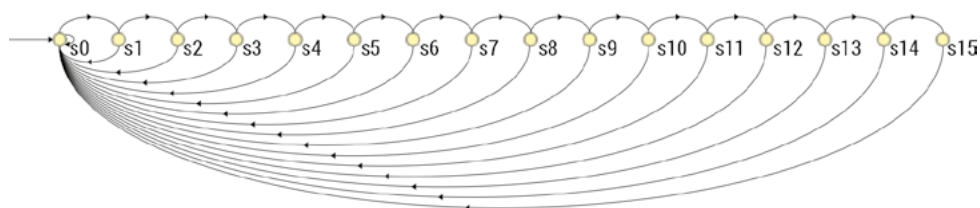


RYS. 7.23. Mapa technologiczna 4-bitowego licznika opisanego w postaci automatu skończonego klasy C

Na rysunku 7.23 widzimy, że automat zawiera cztery przerzutniki (*state~18* – *state~21*), które stanowią elementy pamięci automatu skończonego, oraz układ kombinacyjny (*state~24*), który realizuje funkcje przejść pomiędzy stanami. Sygnały wyjściowe Q_0 – Q_3 automatu skończonego pobierane są bezpośrednio z wyjść elementów pamięci, bez dodatkowego układu kombinacyjnego do realizacji funkcji wyjść. Tym samym schemat na mapie technologicznej projektu dokładnie odpowiada strukturze automatu skończonego klasy C, zgodnie z opisem w kodzie projektu.

Należy zauważyć, że do realizacji automatu klasy C na układach FPGA rodziny MAX II wykorzystano jedynie 5 elementów logicznych, podczas gdy schemat licznika w postaci zwykłego automatu Moore’a z unarnym kodowaniem stanów wymaga aż 26 elementów logicznych.

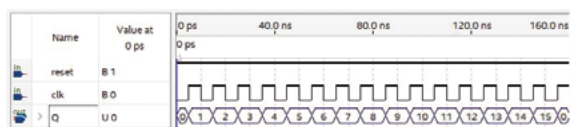
Graf zaprojektowanego automatu skończonego przedstawiono na rysunku 7.24.



RYS. 7.24. Graf zaprojektowanego automatu skończonego

Na rysunku 7.24 można zauważyć, że uzyskany graf automatu skończonego niecałkowicie odpowiada oczekiwanemu grafowi z rysunku 7.21. Wszystkie przejścia na rysunku 7.24 są warunkowe, podczas gdy w grafie licznika zostały zdefiniowane jako bezwarunkowe. Warunkiem, który pojawił się na rysunku 7.24, jest analiza sygnału zerowania *reset*. Przy zerowej wartości tego sygnału automat wraca do stanu początkowego *s0*, natomiast przy wartości równej 1 wykonuje kolejne przejścia pomiędzy stanami.

Wyniki symulacji projektu *counter_4_bits_FSM* pokazano na rysunku 7.25.



RYS. 7.25. Wyniki symulacji 4-bitowego licznika opisanego w postaci automatu skończonego

Analizując wykresy na rysunku 7.25, można potwierdzić, że działanie automatu skończonego dokładnie odpowiada funkcjonowaniu 4-bitowego licznika.

7.1.7. Wyniki badań realizacji liczników na układach FPGA

W celu oceny, który z omówionych opisów licznika jest najkorzystniejszy, dokonano porównania takich parametrów syntezy, jak złożoność realizacji, wyrażana liczbą LE (elementów logicznych FPGA), oraz szybkość działania, określana maksymalną częstotliwością działania licznika F_{max} . Wyniki badań dla różnych rodzin układów FPGA zostały zamieszczone w tabeli 7.1, gdzie maksymalna częstotliwość funkcjonowania liczników podana jest w MHz. W ostatnim wierszu tabeli 7.1 przedstawiono najlepsze wyniki dla każdej rodziny FPGA według złożoności min(LE) oraz według szybkości max(F_{max}).

Analiza wyników przedstawionych w tabeli 7.1 wykazuje, że licznik oparty na przerzutnikach T (projekt *counter_T_4_bits*) był najlepszym wyborem pod względem kosztów realizacji i szybkości działania dla większości rodzin układów FPGA. Jedynym wyjątkiem był wynik dotyczący szybkości dla rodzin MAX 10 i Arria II. Dla tych dwóch rodzin najszybszy okazał się licznik w postaci automatu skończonego (projekt *counter_4_bits_FSM*). Wyniki syntezy dla projektu *counter_N_bits* były prawie

identyczne z projektem *counter_T_4_bits*, z wyjątkiem kosztów realizacji dla układów rodziny MAX 10. Bardzo zbliżone wyniki do projektów *counter_T_4_bits* i *counter_N_bits* uzyskał licznik *counter_4_bits_FSM*, który jest opisywany w postaci automatu skończonego. Choć ten projekt nieco ustępuje projektom *counter_T_4_bits* i *counter_N_bits* pod względem szybkości działania, dla rodzin MAX 10 i Arria II osiąga on największą częstotliwość. Z tego można wywnioskować, że przy projektowaniu *N*-bitowego licznika z asynchronicznym zerowaniem i sygnałem zezwolenia zliczania najlepiej wybierać pomiędzy opisem strukturalnym na podstawie przerzutników T, opisem sparametryzowanym *N*-bitowego licznika oraz opisem licznika w postaci automatu skończonego.

TABELA 7.1. Wyniki realizacji liczników na układach FPGA różnych rodzin

Projekt counter_	MAX II		MAX V		MAX 10		Arria II		Cyclone IV		Cyclone V		Cyclone 10	
	LE	F_{max}	LE	F_{max}	LE	F_{max}	LE	F_{max}	LE	F_{max}	LE	F_{max}	LE	F_{max}
T_4_bits	5	558	5	279	5	600	4	1085	5	912	3	676	5	912
N_bits	5	558	5	279	6	600	4	1085	5	912	3	676	5	912
4_bits_FSM	5	450	5	253	6	643	4	1176	5	809	4	658	5	808
mod_M	6	417	6	225	7	517	4	1134	6	666	3	651	6	670
mod_M_rollover	6	390	6	223	7	518	5	1056	6	645	4	641	6	646
mod_M_roll_aload	7	304	7	204	25	227	18	560	25	312	10	268	25	324
up_down_aload	10	335	10	183	28	211	19	493	27	295	12	264	27	292
mod_M_roll_sload	12	340	12	164	13	422	6	1011	12	549	5	551	12	550
mod_M_roll_sclr	14	350	14	177	15	466	7	930	14	593	6	473	14	641
Best	5	558	5	279	5	643	4	1176	5	912	3	676	5	912

Porównując projekty liczników modulo *M* *counter_mod_M* i *counter_mod_M_rollover* do wcześniej omawianych liczników, można zauważyć niewielki wzrost złożoności realizacji oraz obniżenie szybkości działania dla wszystkich rodzin FPGA. Warto zaznaczyć, że największy spadek szybkości został zarejestrowany dla rodzin MAX II, MAX 10, Cyclone IV i Cyclone 10.

Wprowadzenie możliwości asynchronicznego ładowania danych do licznika (projekt *counter_mod_M_roll_aload*) skutkuje znacznym zwiększeniem złożoności realizacji oraz zauważalnym obniżeniem szybkości działania prawie dla wszystkich rodzin układów FPGA. Na przykład dla rodzin MAX 10, Arria II, Cyclone IV, Cyclone V i Cyclone 10 złożoność realizacji wzrosła ponad trzykrotnie, a szybkość się obniżyła prawie dwukrotnie.

Rozbudowanie licznika o możliwość zliczania zarówno w górę, jak i w dół (projekt *counter_up_down_aload*) jeszcze bardziej zwiększyło koszty realizacji i obniżyło szybkość. Zwiększenie złożoności realizacji jest szczególnie zauważalne dla rodzin układów MAX II i MAX V.

Zamiana asynchronicznego ładowania danych do licznika (projekt *counter_mod_M_roll_aload*) na ładowanie synchroniczne (projekt *counter_mod_M_roll_sload*) spowodowała następujące zmiany: dla rodzin MAX II i MAX V złożoność wzrosła z 7 do 12 elementów logicznych, natomiast dla pozostałych rodzin (MAX 10, Arria II, Cyclone IV, Cyclone V i Cyclone 10) sytuacja była odwrotna – złożoność realizacji zmniejszyła się prawie dwukrotnie, a jednocześnie zwiększyła się szybkość działania.

Zamiana asynchronicznego zerowania licznika (projekt *counter_mod_M_roll_sload*) na zerowanie synchroniczne (projekt *counter_mod_M_roll_sload_sclr*) prowadzi do zwiększenia złożoności realizacji dla wszystkich rodzin układów. W tym przypadku szybkość działania licznika dla niektórych rodzin nieznacznie się zwiększa (MAX II, MAX V, MAX 10, Cyclone IV i Cyclone 10), a dla innych obniża (Arria II i Cyclone V).

Wnioski:

- przy projektowaniu liczników z załadowaniem danych asynchroniczne ładowanie lepiej sprawdza się dla rodzin układów MAX II i MAX V, natomiast synchroniczne ładowanie jest bardziej odpowiednie dla rodzin MAX 10, Arria II, Cyclone IV, Cyclone V i Cyclone 10;
- synchroniczne zerowanie liczników nie daje żadnej przewagi w porównaniu z asynchronicznym zerowaniem, z tego powodu w projektach liczników zaleca się realizować zerowanie asynchroniczne.

7.2. Liczniki Graya i Johnsona

Liczniki Graya i Johnsona są powszechnie wykorzystywane w systemach wbudowanych. Generują one specyficzne sekwencje kodów, inne niż w licznikach dwójkowych.

7.2.1. Licznik Graya

Liczniki Graya (Gray counters) generują na wyjściach sekwencje kodów Graya – każde dwa sąsiednie słowa kodowe różnią się dokładnie na jednym bicie. Ta cecha kodu Graya eliminuje wyścigi przy przełączaniu rejestrów, co czyni go przydatnym w szybkich systemach przetwarzania danych, gdzie ryzyko niejednoczesnego przełączania przerzutników jest znaczące. Liczniki Graya znajdują zastosowanie również w przetwornikach kątowych oraz różnego rodzaju czujnikach położenia i obrotu.

Wartości bitów kodu Graya i kodu binarnego powiązane są następującą zależnością:

$$g[i] = b[i] \wedge b[i+1], \quad (7.1)$$

gdzie: $g[i]$ – i -ty bit kodu Graya; $b[i]$ – i -ty bit kodu dwójkowego; $\wedge$ – operacja XOR.

Odwrotna konwersja bitów kodu Graya na kod dwójkowy wykonuje się następująco:

$$b[i] = b[i+1] \wedge g[i]. \quad (7.2)$$

Licznik Graya można skonstruować, bazując na standardowym liczniku dwójkowym poprzez przekształcenie kodu dwójkowego na kod Graya zgodnie z wyrażeniem (7.1). Poniżej znajduje się przykładowy opis licznika Graya w języku Verilog:

```

module gray_counter
  #(parameter N=3)
  (input clk,reset,
   output reg [N-1:0] gray);

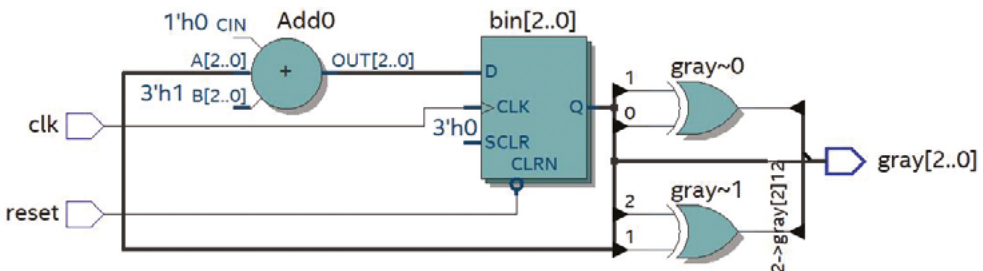
  reg [N-1:0] bin;
  integer i;

  always @(posedge clk, negedge reset)
    if(~reset)    bin <= 0;
    else          bin <= bin + 1'b1;

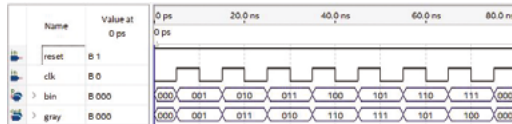
  always @(*)
    for (i=0; i<N; i=i+1)
      gray = (bin>>1) ^ bin;
endmodule

```

W przywołanym przykładzie pierwszy blok instrukcji **always** implementuje licznik dwójkowy, podczas gdy drugi blok instrukcji **always** przekształca kod dwójkowy na kod Graya. Wyniki syntezy i symulacji projektu *gray_counter* przedstawiono na rysunkach 7.26 i 7.27.



RYS. 7.26. Schemat projektu *gray_counter* na poziomie przesłań międzyrejestrowych



RYS. 7.27. Wyniki symulacji funkcjonalnej projektu *gray_counter*

Na rysunku 7.27 można zauważyć, że licznik *gray_counter* generuje na wyjściach sekwencję 3-bitowych kodów Graya.

7.2.2. Licznik Johnsona

Licznik Johnsona (Johnson counter) jest licznikiem pierścieniowym generującym symetryczne kody. Kolejno ustawia on jedynki w rejestrze, zaczynając od prawej strony aż do wypełnienia nimi całego słowa, po czym zaczyna wpisywać kolejne zera w miejsce jedynek, prowadząc do całkowitego wyzerowania rejestru. Wartość modułu M licznika Johnsona wynosi $2N$, gdzie N – liczba bitów licznika. W liczniku Johnsona na przejściach pomiędzy kolejnymi stanami, podobnie jak w liczniku Graya, zmienia się wartość tylko jednego bitu.

W systemach wbudowanych liczniki Johnsona są wykorzystywane do tworzenia dzielników częstotliwości impulsów, generatorów kodów unarnych oraz elementów systemów szybkiego przetwarzania danych.

Licznik Johnsona można łatwo zbudować na bazie rejestru przesuwanego w lewo. Aby to osiągnąć, wystarczy połączyć wyjście *MSB_out* rejestru przesuwanego z jego wejściem *LSB_in* za pośrednictwem inwertera. Poniżej przedstawiono kod projektu licznika Johnsona utworzonego w opisany sposób:

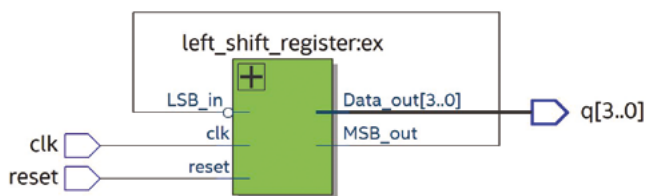
```
module johnson_counter
    #(parameter N=4)
    (input clk, reset,
     output [N-1:0] q);

    wire c;

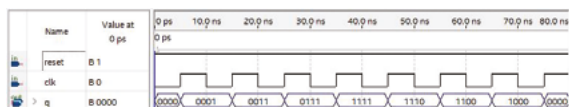
    left_shift_register #(4)
    ex(.LSB_in(~c), .clk(clk), .reset(reset), .Data_out(q), .MSB_out(c));

endmodule
```

Wyniki syntezy i symulacji projektu *johnson_counter* możemy zobaczyć na rysunkach 7.28 i 7.29.



RYS. 7.28. Schemat projektu *johnson_counter* na poziomie RTL



RYS. 7.29. Wyniki symulacji funkcjonalnej projektu *johnson_counter*

7.3. Schematy opóźnień

Przy tradycyjnym projektowaniu na poziomie bramek często wprowadza się parzystą liczbę inwerterów w celu uzgodnienia opóźnień sygnałów. Jednakże przy projektowaniu na układach FPGA takie podejście nie jest praktyczne, ponieważ architektury FPGA nie dysponują oddzielnymi bramkami inwerterów. Dodatkowo każda parzysta liczba inwerterów zostanie usunięta ze schematu w procesie optymalizacji projektu. Czasami projektanci wykorzystują sekwencyjnie połączone prymitywy AND, OR, BUF oraz LCELL (w przypadku układów FPGA firmy Intel) ze wskazaniem kompilatorowi, aby nie optymalizował określonych fragmentów projektu. Niemniej jednak jest to podejście niezalecane, gdyż wprowadzone opóźnienie sygnału w takiej sytuacji znacząco zależy od parametrów konkretnego układu FPGA, a także od takich czynników, jak napięcie zasilania, temperatura, technologia wykonania układu FPGA i inne parametry.

W celu wprowadzenia opóźnień sygnałów w projektach systemów wbudowanych na układach FPGA można wykorzystać kilka rozwiązań. Najprostszym i najbardziej niezawodnym podejściem jest buforowanie sygnałów wyjściowych rejestru. Jednakże wymaga to dodatkowych zasobów do sterowania wyjściami rejestrów.

Kolejnym rozwiązaniem jest wprowadzenie do schematu ograniczeń czasowych zapisywanych w pliku SDC, które regulują czasy generowania określonych sygnałów. Przy takim podejściu problem jednoczesnego generowania sygnałów w wyznaczonym miejscu projektu jest rozwiązywany automatycznie przez oprogramowanie narzędziowe systemu. Niemniej zastosowanie tego podejścia wymaga znajomości podstaw analizy czasowej oraz języka opisu ograniczeń czasowych SDC lub Tcl.

Czasami w projekcie wymagane są dłuższe opóźnienia sygnałów. W takim przypadku zaleca się wykorzystanie liczników oraz rejestrów przesuwanych, a w przypadku FPGA firmy Xilinx – bloków DCM (*Digital Clock Manager*).

Następny przykład kodu opisuje sposób realizacji opóźnienia sygnału wejściowego o M cykli zegarowych sygnału clk , opierając się na rejestrze przesuwным *right_shift_register*. Aby to osiągnąć, do modułu jako parametr przekazywana jest wartość M , na podstawie której obliczana jest liczba bitów rejestru przesuwного. Uwzględniając czas cyklu T zegara systemowego, łatwo jest obliczyć rzeczywisty czas wprowadzanego opóźnienia $t = M \cdot T$. Na przykład, dla $T = 10$ ns i $M = 5$ opóźnienie sygnału wynosi $t = 50$ ns.

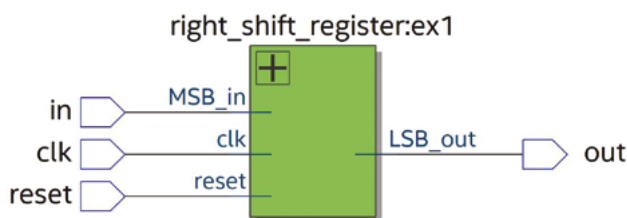
```

module delay_M_clock_shift_reg
    #(parameter M=5)
    (input in,clk,reset,
    output out);

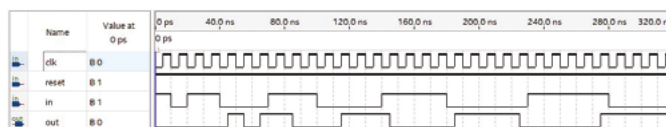
    right_shift_register #(M) ex1(in,clk,reset,,out);
endmodule

```

Wyniki syntezy i symulacji projektu *delay_M_clock_shift_reg* zaprezentowano na rysunkach 7.30 i 7.31.



RYS. 7.30. Wynik syntezy schematu opóźnienia sygnału o M cykli zegarowych clk zbudowanego na podstawie rejestru przesuwного w prawo



RYS. 7.31. Wyniki symulacji schematu opóźnienia sygnału o M cykli zegarowych dla $M=5$

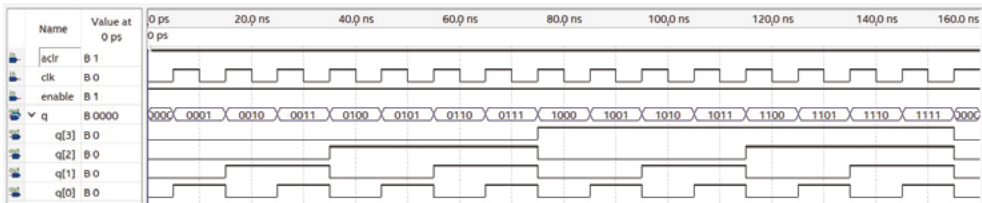
W analogiczny sposób można skonstruować schemat wprowadzający opóźnienie o dowolny przedział czasowy. Zaletą tego podejścia jest to, że przy stabilnym sygnale zegarowym (opartym na rezonatorze kwarcowym) czas opóźnienia jest bardzo dokładnie określany, ponadto nie zależy on od układu docelowego FPGA, napięcia zasilania, temperatury, wilgotności czy innych parametrów. Jedynym ograniczeniem jest rozmiar rejestru przesuwного przy wysokiej częstotliwości sygnału zegarowego i dłuższych czasach opóźnień. Aby zaradzić temu problemowi, wykorzystuje się liczniki działające

na znacznie niższych częstotliwościach niż częstotliwość systemowego sygnału synchronizacji. Do obniżenia częstotliwości systemowego lub bazowego sygnału synchronizacji wykorzystywane są układy dzielników częstotliwości.

7.4. Dzielniki częstotliwości

W systemach wbudowanych często występuje potrzeba zmniejszenia częstotliwości sygnału synchronizacji w celu uzyskania sygnałów o niższych częstotliwościach. Układ, który wykonuje to zadanie, nosi nazwę *dzielnika częstotliwości* (*frequency divider*). W literaturze można również spotkać terminy takie jak *dzielnik częstotliwości zegara* (*clock divider*), *skaler* (*scaler*), *preskaler* (*prescaler*) lub *postskaler* (*postscaler*). Najprostszym dzielnikiem częstotliwości jest przerzutnik typu T, który dwukrotnie zmniejsza częstotliwość sygnału wejściowego.

Wszystkie liczniki również pełnią funkcję dzielników częstotliwości. Na przykład wyjście *q0* 4-bitowego licznika dzieli częstotliwość sygnału synchronizacji licznika przez 2, *q1* dzieli przez 4, *q2* przez 8, a *q3* przez 16. Możemy to zobaczyć na przebiegach czasowych wyników symulacji 4-bitowego licznika (rysunek 7.32).



RYS. 7.32. Czterobitowy licznik jako dzielnik częstotliwości

Zaprojektujemy moduł dzielnika częstotliwości, który na wejściu przyjmuje sygnał synchronizacji *clk_in*, a na wyjściu będzie generował sygnał *clk_out*, którego częstotliwość zostanie podzielona przez określoną liczbę *M*. Poniżej został przedstawiony kod takiego układu:

```
module clock_divider
    #(parameter M=2)
    (input clk_in,
     output clk_out);

    reg [M-1:0] counter=0;

    always @(posedge clk_in)
    begin
        counter <= counter + 1'b1;
    end
endmodule
```



```

        if (counter >= (M - 1))
            counter <= 0;

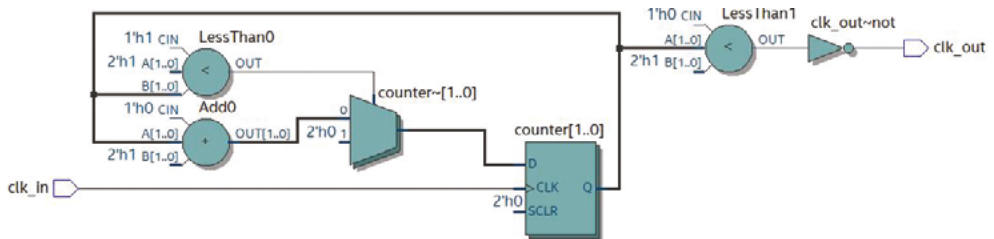
    end

    assign clk_out = (counter < M/2) ? 1'b0 : 1'b1;

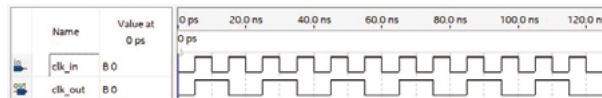
endmodule

```

W kodzie projektu *clock_divider* wartość dzielnika M jest określana parametrem. Projekt ten zawiera wewnętrzny licznik zdefiniowany za pomocą zmiennej *counter*. Stan sygnału wyjściowego *clk_out* jest określany na podstawie wartości zmiennej *counter* i instrukcji **assign**. Przez połowę okresu zliczania licznika sygnał *clk_out* przyjmuje stan niski, a przez drugą połowę – stan wysoki. Wyniki syntezy i symulacji modułu *clock_divider* pokazano na rysunkach 7.33 i 7.34.



RYS. 7.33. Wynik syntezy dzielnika częstotliwości na poziomie RTL



RYS. 7.34. Wynik symulacji dzielnika częstotliwości dla $M=2$

Warto zauważyć, że zaprojektowany dzielnik częstotliwości pozwala na dzielenie częstotliwości sygnału wejściowego przez dowolną liczbę całkowitą, a nie tylko przez liczbę będącą potęgą dwójki, jak to ma miejsce w przypadku wykorzystania do dzielenia częstotliwości licznika dwójkowego. Zachęcamy czytelnika do przeprowadzenia symulacji funkcjonalnej dzielnika częstotliwości *clock_divider* dla zarówno przyrostych, jak i nieparzystych wartości parametru M .

7.5. Generatory impulsów sygnałów długookresowych

Przy projektowaniu zegarów czasu rzeczywistego należy na podstawie sygnału zegarowego o wysokiej częstotliwości wygenerować pojedyncze impulsy z dłuższym okresem powtarzania, na przykład

- 1s,
- 0,1 s = 100 ms,
- 0,01 s = 10 ms lub
- 0,001 s = 1 ms.

Warto zaznaczyć, że im większa jest częstotliwość bazowego sygnału zegarowego, tym dokładniejsze mogą być obliczenia wyznaczonego przedziału czasu.

Płytką prototypową DE1-SoC wyposażoną jest w rezonator kwarcowy, który generuje systemowy sygnał zegarowy o częstotliwości 50 MHz, przekazywany na wejście synchronizacji układu FPGA. W dalszej części rozdziału przedstawimy obliczenia, które ilustrują, że czas cyklu sygnału synchronizacji o częstotliwości 50 MHz wynosi 20 ns.

Do generowania impulsów występujących w dłuższych okresach czasu idealnie nadaje się licznik z wyjściem przepełnienia *rollover* (projekt *counter_modulo_M_rollover*). Na przykład, aby wygenerować impuls o okresie 1s, należy ustawić wartość modułu licznika równą 50 000 000, a pożądany sygnał będzie dostępny na wyjściu *rollover*.

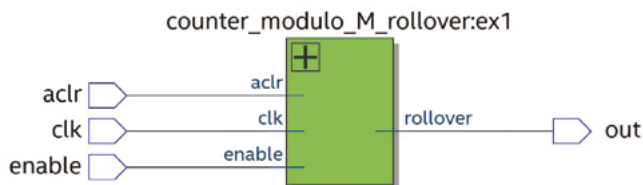
Kod generatora impulsów o okresie powtarzania równym pięciu cyklom sygnału zegarowego, zbudowany na podstawie licznika z wyjściem *rollover*, może zostać przedstawiony w następujący sposób:

```
module pulse_generator_M_clock_period
    #(M=5)
    (input clk,aclr,enable,
     output out);

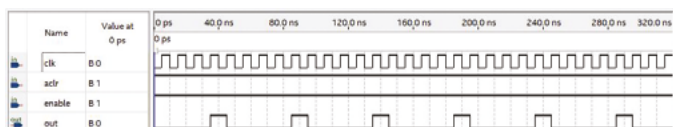
    counter_modulo_M_rollover #(M) ex1(clk,aclr,enable,,out);
```

endmodule

Wyniki syntezy i symulacji projektu *pulse_generator_M_clock_period* pokazano na rysunkach 7.35 oraz 7.36.



RYS. 7.35. Wynik syntezy generatora impulsów



RYS. 7.36. Wyniki symulacji generatora impulsów z okresem równym pięciu cyklom sygnału zegarowego

Proponujemy czytelnikowi zaprojektowanie generatora impulsów o długim okresie na podstawie modułu *counter_modulo_M_zero*. Ten moduł powinien być zaprojektowany jako licznik dwójkowy, który generuje na wyjściu *zero* sygnał o wartości 1, gdy zawartość licznika wynosi zero. Warto zauważyć, że moduł *counter_modulo_M_rollover* ustawia jedynkę na wyjściu *rollover* w momencie, gdy wartość licznika osiągnie $M-1$. Zalecamy porównanie projektów *counter_modulo_M_zero* i *counter_modulo_M_rollover* (analizę wyników syntezy na poziomie RTL, złożoności realizacji oraz wyników symulacji funkcjonalnej), wyjaśnienie istniejących różnic i rozważenie, który z projektów najlepiej sprawdzi się do realizacji zegara czasu rzeczywistego.

Sugerujemy czytelnikowi także przygotowanie projektu *delay_10ms*, który na bazie sygnału zegarowego o częstotliwości 50 MHz generuje impulsy o okresie równym 10 ms. Taki projekt będzie przydatny w implementacji zegara czasu rzeczywistego.

7.6. Powolne liczniki

W systemach wbudowanych może pojawić się potrzeba stosowania liczników o znacznie niższej częstotliwości niż zegar systemowy. Takie liczniki, nazywane *powolnymi* (*slow counter*), mogą być zbudowane na bazie licznika modulo M z wyjściem przepełnienia *rollover* oraz standardowego licznika z wejściem zezwolenia zliczania *enable*. Wyjście *rollover* pierwszego licznika należy połączyć z wejściem zezwolenia zliczania drugiego licznika. Wartość modułu $M1$ pierwszego licznika określa, o ile razy mniejsza będzie częstotliwość funkcjonowania drugiego licznika w porównaniu z częstotliwością przełączania pierwszego licznika. Poniżej znajduje się kod projektu powolnego licznika, w którym domyślnie zakłada się pięciokrotne zmniejszenie częstotliwości pracy drugiego licznika w odniesieniu do częstotliwości sygnału *clk*:

```
module slow_counter
#(parameter M1=5, N2=2)
(input clk,aclr,enable,
output reg[N2-1:0] Q2);

localparam N1=clogb2(M1-1);

function integer clogb2(input [31:0] v);
```

```

    for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
        v = v >> 1;
endfunction

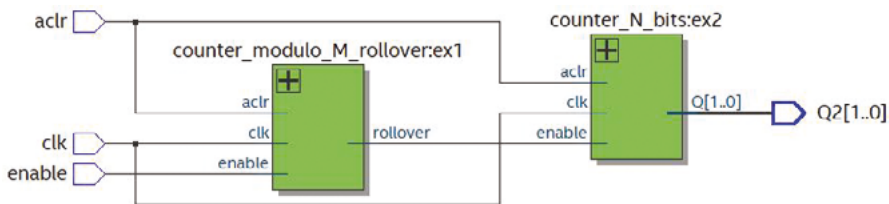
reg [N1-1:0] Q1;
wire rollover;

counter_modulo_M_rollover #(M1) ex1(clk,aclr,enable,Q1,rollover);

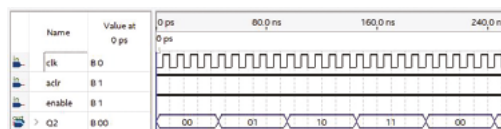
counter_N_bits #(N2) ex2(clk,aclr,rollover,Q2);
endmodule

```

Moduł *slow_counter* ma dwa parametry: *M1*, który określa wartość modułu pierwszego licznika, oraz *N2*, który wyznacza liczbę bitów drugiego licznika. W kodzie projektu do obliczenia liczby bitów *N1* pierwszego licznika wykorzystana została stała funkcja *clogb2* [14], która zwraca najmniejszą liczbę całkowitą równą lub większą od wartości $\log_2 M$, gdzie *M* jest podaną liczbą. Wyniki syntezy i symulacji projektu *slow_counter* przedstawiono na rysunkach 7.37 i 7.38.



RYS. 7.37. Schemat projektu modułu *slow_counter* na poziomie RTL



RYS. 7.38. Wyniki symulacji modułu *slow_counter*

Na rysunku 7.38 zauważamy, że sygnał na wyjściu *Q2* powolnego licznika zmienia swoją wartość co 5 cykli sygnału zegarowego *clk*.

W kolejnym przykładzie zaprojektujemy licznik sterowany sygnałem o okresie 1 s przy częstotliwości systemowego sygnału zegarowego równej 50 MHz. Najpierw na podstawie częstotliwości zegara systemowego 50 MHz = 50 000 000 Hz obliczymy długość cyklu pierwszego licznika:

$$T_{50\text{MHz}} = 1/50\,000\,000 = 0,000\,000\,02\text{ s} = 0,000\,02\text{ ms} = 0,02\text{ us} = 20\text{ ns}.$$

Z obliczeń wynika, że długość cyklu sygnału taktowania pierwszego licznika przy częstotliwości systemowego sygnału zegarowego równej 50 MHz wynosi 20 ns.

Następnie obliczymy wartość modułu pierwszego licznika, który pozwoli otrzymać sygnał sterujący drugim licznikiem o okresie 1 s:

$$M = 1 \text{ s} / 20 \text{ ns} = 1\,000\,000\,000 \text{ ns} / 20 \text{ ns} = 50\,000\,000.$$

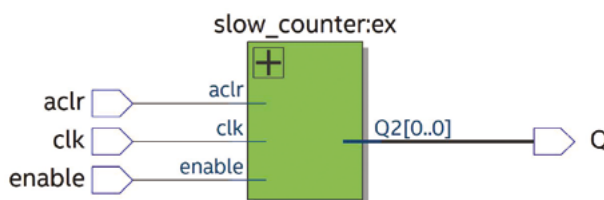
Wynik obliczeń pokazuje, że przy ustawieniu wartości modułu pierwszego licznika na $M = 50\,000\,000$ drugi licznik będzie pracować z częstotliwością 1 Hz, co odpowiada okresowi równemu 1 s.

Kod licznika pracującego z częstotliwością 1 Hz (o okresie 1 s) zawiera tylko jedną instrukcję tworzenia instancji powolnego licznika *slow_counter* z odpowiednimi parametrami:

```
module counter_1s(
    input clk,aclr,enable,
    output reg Q);

    slow_counter #(50000000,1) ex(clk,aclr,enable,Q);
endmodule
```

Wynik syntezy licznika o okresie sygnału synchronizacji równym 1 s pokazany został na rysunku 7.39.



RYS. 7.39. Schemat powolnego licznika o okresie 1s na poziomie RTL

Proponujemy czytelnikowi zaprojektowanie liczników o okresach 100 ns, 1 ms, 10 ms oraz 100 ms, zakładając, że częstotliwość zegara systemowego wynosi 50 MHz.

7.7. Czasomierze

Czasomierz (*timer*) to układ, który mierzy określony przedział czasu i sygnalizuje to ustawieniem jedynki na wyjściu. Zazwyczaj czasomierze są projektowane na bazie liczników odejmujących. Do licznika ładowana jest wartość, która na podstawie okresu systemowego sygnału zegarowego (na przykład 20 ns) określa wymaganą długość przedziału czasowego. Na wyjściu czasomierza generowana jest jedynka, gdy licznik osiąga wartość zerową. Poniżej znajduje się kod projektu zwykłego czasomierza:

```

module timer_N_bits
    #(parameter N=4)
    (input clk,load,enable,
    input [N-1:0] data,
    output out);

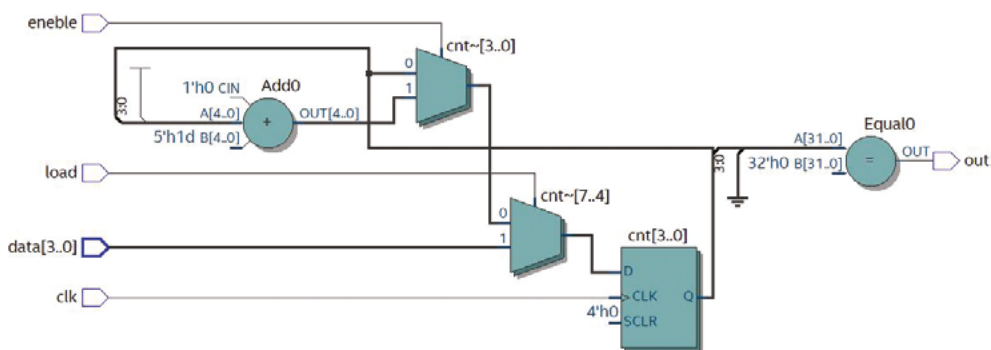
    reg [N-1:0] cnt;

    always @(posedge clk)
        if (load)           cnt <= data;
        else if (enable)    cnt <= cnt - 1;
        else                cnt <= cnt;

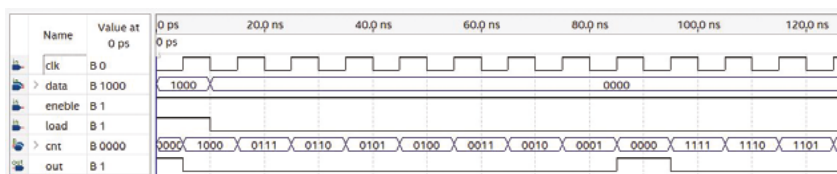
    assign out = (cnt==0);
endmodule

```

Wyniki syntezy i symulacji modułu *timer_N_bits* pokazane zostały na rysunkach 7.40 oraz 7.41.



RYS. 7.40. Wynik syntezy czasomierza



RYS. 7.41. Wyniki symulacji czasomierza

W przykładzie przeprowadzonej symulacji najpierw do licznika ładowana jest wartość $1000_2 = 8_{10}$, po czym licznik zaczyna odliczać w dół, a przy osiągnięciu wartości 0000_2 generowany jest impuls na wyjściu *out*. W ten sposób czasomierz odmierza ustalony przedział czasowy:

$$T = 8 \cdot T_{sys}$$

gdzie T_{sys} – długość cyklu systemowego sygnału zegarowego *clk*.

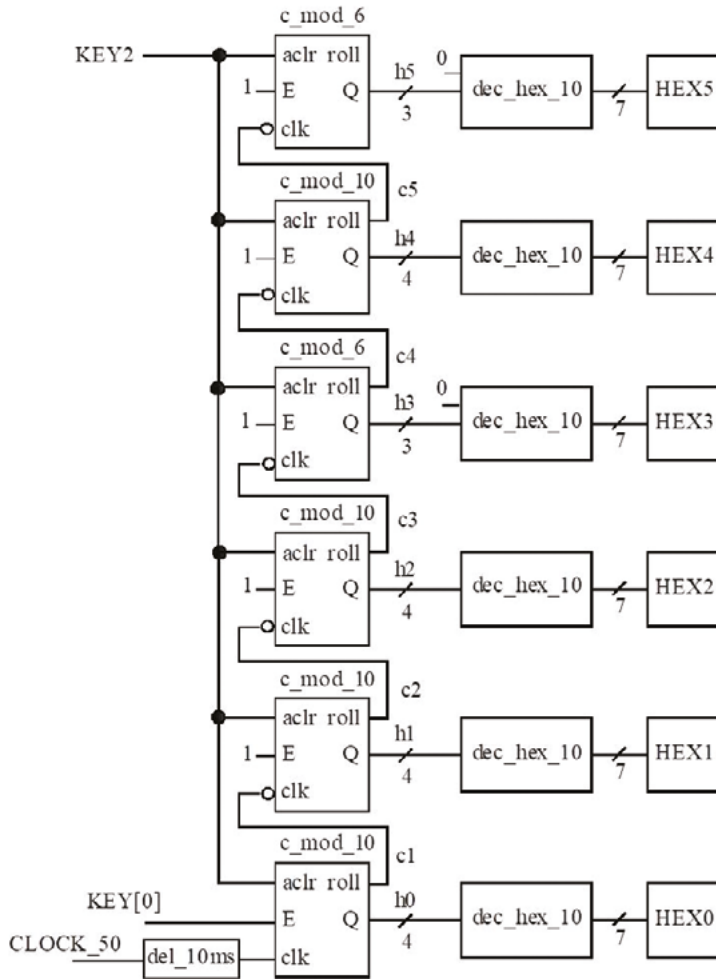
Zaprojektowany układ *timer_N_bits* pozwala zmierzyć przedział czasowy $M \cdot T_{sys}$, gdzie M jest liczbą, którą można zapisać na N bitach licznika *cnt*, co oznacza, że powinna być spełniona nierówność: $M \leq 2^N - 1$. Proponujemy czytelnikowi zaprojektowanie uniwersalnego układu czasomierza *timer_mod_M*, w którym liczba cykli zliczanych przez licznik w celu zmierzenia przedziału czasowego określana jest parametrem M . (Wskazówka: do obliczenia liczby N bitów licznika *cnt* można wykorzystać funkcję stałą *clogb2*).

Uwaga. Czasomierz może być także zaprojektowany na podstawie licznika z wyjściem *rollover*.

7.8. Zegar czasu rzeczywistego

Zegar czasu rzeczywistego (real time clock) jest powszechnie stosowany w systemach wbudowanych większości urządzeń AGD i RTV. W celu demonstracji zaprojektujemy układ zegara czasu rzeczywistego, który będzie realizowany na płycie DE1-SoC. Na wyświetlaczach HEX5-4 będą wypisywane minuty, na HEX3-2 – sekundy, a na HEX1-0 – setne części sekund. Strukturę projektu przedstawiono schematycznie na rysunku 7.42.

Schemat projektu obejmuje sześć liczników z wyjściem *rollover*, z asynchronicznym zerowaniem *aclr* i wejściem zezwolenia zliczania *E*. Wyjścia liczników są podłączone do dekodów *dec_hex_10*, które z kolei są połączone z wyświetlaczami HEX5-0. Wyjście *rollover* każdego licznika połączone jest przez inwertor do wejścia synchronizacji następnego licznika, co oznacza, że każdy z liczników pełni funkcję dzielnika częstotliwości. Wszystkie liczniki mają wspólny sygnał resetowania wysyłany przyciskiem KEY2. Przycisk KEY0 służy do zatrzymania odliczania czasu.



RYS. 7.42. Struktura zegara czasu rzeczywistego

Na wejście synchronizacji najmłodszego licznika związanego z wyświetlaczem HEX0 podawany jest sygnał zegarowy o okresie 10 ms. W tym celu pomiędzy systemowym sygnałem zegarowym *CLOCK_50* a wejściem synchronizacji licznika umieszczony został schemat opóźnienia *del_10ms* o 10 milisekund. Uruchomienie zegara czasu rzeczywistego sterowane jest przyciskiem KEY0, który jest podłączony do wejścia *E* najmłodszego licznika. Opis projektu zegara czasu rzeczywistego w języku Verilog można przedstawić następująco:

```

module real_time(
    input [2:0] KEY,
    input CLOCK_50,
    output [0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);

```



```

wire c0,c1,c2,c3,c4,c5;
wire [3:0] h0,h1,h2,h4;
wire [2:0] h3,h5;

delay_10ms del(CLOCK_50,c0);

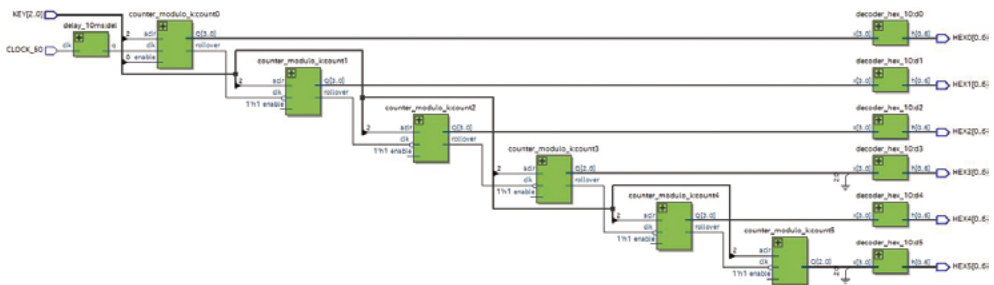
counter_modulo_M_rollover #(10) count0(c0,KEY[2],KEY[0],h0,c1);
counter_modulo_M_rollover #(10) count1(~c1,KEY[2],1'b1,h1,c2);
counter_modulo_M_rollover #(10) count2(~c2,KEY[2],1'b1,h2,c3);
counter_modulo_M_rollover #(6) count3(~c3,KEY[2],1'b1,h3,c4);
counter_modulo_M_rollover #(10) count4(~c4,KEY[2],1'b1,h4,c5);
counter_modulo_M_rollover #(6) count5(~c5,KEY[2],1'b1,h5);

decoder_hex_10 d0(h0,HEX0);
decoder_hex_10 d1(h1,HEX1);
decoder_hex_10 d2(h2,HEX2);
decoder_hex_10 d3({1'b0,h3},HEX3);
decoder_hex_10 d4(h4,HEX4);
decoder_hex_10 d5({1'b0,h5},HEX5);

```

endmodule

Na płytce DE1-SoC zamontowany jest układ FPGA 5CSEMA5F31C6N z rodziny Cyclone V SE. Aby przyporządkować wyprowadzenia układu FPGA zgodnie z połączeniami na płytce DE1-SoC, do projektu należy dołączyć plik DE1_SoC.qsf, który można znaleźć na stronie producenta płytki oraz na stronie **www.intel.com** (University Program). Wynik syntezy modułu *real_time* przedstawiony został na rysunku 7.43.



RYS. 7.43. Wynik syntezy zegara czasu rzeczywistego

Testowanie zaprojektowanego układu zegara czasu rzeczywistego można wykonać bezpośrednio na płytce DE1-SoC, ładując projekt do układu FPGA.

7.9. Podsumowanie rozdziału

Licznik (counter) to układ sekwencyjny, który cyklicznie przechodzi przez określoną niezmienną sekwencję swoich stanów.

Wejściem licznika dwójkowego jest wejście sygnału zegarowego, nazywane także *wejściem zliczającym*. Wyjściem licznika jest N -bitowa szyna, na której generowane są kolejne liczby dwójkowe. Długość M cyklu stanów licznika nazywana jest *modułem licznika*. Liczniki mogą mieć także szynę danych wejściowych (*data*), na którą podawane są dane ładowane do licznika, oraz dodatkowe sygnały sterujące: asynchronicznego (*aclr*) lub synchronicznego (*sclr*) zerowania, zezwolenia pracy (*enable*), asynchronicznego (*aload*) bądź synchronicznego (*sload*) załadowania i inne.

Liczniki, które zliczają w górę, nazywane są *licznikami dodającymi*, podczas gdy te, które zliczają w dół, określane są jako *liczniki odejmujące*. *Liczniki rewersyjne* mają zdolność do zliczania zarówno w górę, jak i w dół.

Dla liczników dwójkowych maksymalna wartość modułu wynosi 2^N , gdzie N – liczba bitów licznika. *Licznikiem modulo M* nazywany jest licznik, dla którego $M \leq 2^N$.

Podczas projektowania liczników w języku Verilog można korzystać z opisu strukturalnego lub behawioralnego, a także reprezentować je jako automaty skończone. Ładowanie danych do licznika może być realizowane synchronicznie albo asynchronicznie.

Licznik oparty na przerzutnikach T (projekt *counter_T_4_bits*) okazał się najlepszym wyborem pod względem kosztów realizacji i szybkości działania dla większości rodzin układów FPGA. Wyjątkiem były tylko wyniki dotyczące szybkości dla rodzin MAX 10 i Arria II, gdzie najszybszym rozwiązaniem okazał się licznik w postaci automatu skończonego.

Przy projektowaniu N -bitowego licznika z asynchronicznym zerowaniem i sygnałem zezwolenia zliczania najlepiej wybierać pomiędzy opisem strukturalnym na podstawie przerzutników T, opisem sparаметryzowanym N -bitowego licznika oraz opisem licznika w postaci automatu skończonego.

Projekty liczników modulo M *counter_mod_M* i *counter_mod_M_rollover*, w porównaniu z pełnymi licznikami, prowadzą do niewielkiego wzrostu złożoności realizacji oraz obniżenia szybkości działania dla wszystkich rodzin FPGA.

Wprowadzenie możliwości asynchronicznego ładowania danych do licznika (projekt *counter_mod_M_roll_aload*) prowadzi do znaczącego wzrostu złożoności realizacji oraz zauważalnego obniżenia szybkości działania prawie dla wszystkich rodzin układów FPGA.

Rozbudowanie licznika o możliwość zliczania zarówno w górę, jak i w dół (projekt *counter_up_down_aload*) jeszcze bardziej zwiększyło koszty realizacji i obniżyło szybkość dla wszystkich rodzin układów FPGA.

Przy projektowaniu liczników z funkcją ładowania danych asynchroniczne ładowanie najlepiej sprawdza się w przypadku rodzin układów MAX II i MAX V, natomiast synchroniczne ładowanie – dla rodzin MAX 10, Arria II, Cyclone IV, Cyclone V i Cyclone 10.

Synchroniczne zerowanie liczników nie daje żadnej przewagi w porównaniu z zerowaniem asynchronicznym. Z tego powodu w projektach liczników zaleca się raczej stosowanie zerowania asynchronicznego.

Licznik Graya (*Gray countes*) generuje na swoich wyjściach sekwencje kodów Graya. *Licznik Johnsona* (*Johnson counter*) jest licznikiem pierścieniowym, który generuje symetryczne kody. W sekwencjach kodów Graya i Johnsona każde dwa sąsiednie słowa kodowe różnią się dokładnie na jednym bicie. Wartość modułu licznika Graya wynosi 2^N , podczas gdy licznika Johnsona – $2N$. Licznik Johnsona można łatwo zbudować na bazie rejestru przesuwanego w lewo.

Przy projektowaniu na FPGA nie zaleca się stosowania sekwencyjnego połączenia parzystej liczby inwertorów, bramek AND, OR, a także prymitywów buforowych BUF i LCELL do wprowadzania opóźnień sygnałów.

Opóźnienia o kilka cykli zegarowych zazwyczaj są realizowane za pomocą rejestrów przesuwanych. W tym przypadku czas wprowadzonego opóźnienia można obliczyć jako iloczyn czasu cyklu sygnału zegarowego oraz liczby bitów rejestru przesuwanego.

Aby zmniejszyć częstotliwość sygnału zegarowego, w systemach wbudowanych często stosuje się dzielniki częstotliwości, generatory impulsów o długich okresach oraz powolne liczniki.

Najprostszym dzielnikiem częstotliwości jest przerzutnik typu T, który dzieli częstotliwość przez 2. Dodatkowo wyjścia dowolnego licznika dwójkowego mogą być wykorzystane do dzielenia częstotliwości przez potęgę dwójki.

Aby podzielić częstotliwość sygnału zegarowego przez dowolną liczbę, niekoniecznie będącą potęgą dwójki, należy skorzystać z projektu *clock_divider*.

Generatory impulsów długookresowych często są wykorzystywane w zegarach czasu rzeczywistego. Przykładem takiego generatora jest licznik modulo M z wyjściem przepełnienia *rollover*.

Powolne liczniki w systemach wbudowanych operują na znacznie mniejszych częstotliwościach niż sygnał zegara systemowego. Powolny licznik można skonstruować poprzez sekwencyjne połączenie dwóch liczników: licznika modulo M z wyjściem przepełnienia oraz licznika dwójkowego z wejściem zezwolenia *enable*.

Czasomierz (*timer*) jest układem, który odmierza określony przedział czasu i informuje o tym przez ustawienie jedynki na wyjściu. Zazwyczaj czasomierze są projektowane na podstawie liczników odejmujących. Również licznik modulo M z wyjściem *rollover* może pełnić funkcję czasomierza.

Zegar czasu rzeczywistego może być skonstruowany poprzez sekwencyjne połączenie liczników z wyjściem *rollover*. Do generowania sygnału synchronizacji o określonej częstotliwości wykorzystuje się często generatory impulsów długookresowych. Im wyższa jest częstotliwość bazowego sygnału synchronizacji, tym większa jest dokładność zegara czasu rzeczywistego.

8. Projektowanie bloków arytmetycznych

Podczas projektowania systemów wbudowanych często występuje konieczność implementacji bloków umożliwiających wykonywanie operacji arytmetycznych, takich jak dodawanie i mnożenie. Operacja odejmowania zazwyczaj sprowadzana jest do dodawania uzupełnienia dwójkowego. Natomiast operacja dzielenia praktycznie nie jest powszechnie stosowana w systemach wbudowanych opartych na układach FPGA. Jedynym wyjątkiem jest dzielenie przez potęgę dwójki, które realizowane jest przez przesunięcie w prawo o odpowiednią liczbę bitów.

W systemach wbudowanych opartych na układach FPGA bloki arytmetyczne zazwyczaj obsługują instrukcje na liczbach całkowitych. Jednakże do bardziej skomplikowanych obliczeń oraz przetwarzania liczb rzeczywistych zaleca się wykorzystanie wbudowanego (dla układów SoC) lub zewnętrznego procesora.

W niniejszym rozdziale omówiono podstawowe bloki arytmetyczne stosowane w systemach wbudowanych, takie jak sumatory, subtraktory, akumulatory, sumatory liczb BCD oraz układy inkrementujące i dekrementujące. Natomiast bardziej złożone układy mnożenia zostaną przedstawione w kolejnym rozdziale.

8.1. Sumatory

Podstawowym blokiem arytmetycznym w systemach obliczeniowych jest *sumator* (*adder*), który wykonuje operację dodawania. Na jego bazie tworzone są inne bloki arytmetyczne, takie jak subtraktory, akumulatory, układy inkrementujące i dekrementujące. Sumatory stanowią podstawę jednostek arytmetyczno-logicznych (ALU), układów mnożenia oraz są wykorzystywane do sprzętowej realizacji większości funkcji cyfrowego przetwarzania sygnałów (DSP) i kryptografii. Istnieje wiele metod projektowania sumatorów, które nadal są rozwijane.

Istotne znaczenie sumatorów w technice cyfrowej potwierdza fakt, że w wielu architekturach układów FPGA wprowadzono specjalne komponenty wspomagające efektywną realizację sumatorów. Na przykład w układach FPGA firmy Intel zastosowano *bloki szybkiego przeniesienia* (*Carry Chain*) oraz *tryb arytmetyczny* pracy elementów logicznych [2], natomiast w układach FPGA firmy Xilinx wykorzystano *logikę przeniesień* (*Carry Logic*).

8.1.1. Sumatory jednobitowe

Praktycznie wszystkie układy sumatorów budowane są na podstawie sumatora jednobitowego. Jego działanie można opisać przy użyciu tablicy prawdy zamieszczonej w tabeli 8.1, gdzie a i b to pojedyncze bity słów A i B , cin to przeniesienie z poprzedniego bitu, s to bit sumy, a $cout$ to przeniesienie do bitu następnego.

TABELA 8.1. Tablica prawdy jednobitowego sumatora

Wejścia			Wyjścia	
cin	a	b	$cout$	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Na podstawie tabeli 8.1 można utworzyć następujące równania logiczne funkcji wyjściowych:

$$s = \overline{cin} \cdot \overline{a} \cdot b + \overline{cin} \cdot a \cdot \overline{b} + cin \cdot \overline{a} \cdot \overline{b} + cin \cdot a \cdot b; \quad (8.1)$$

$$cout = \overline{cin} \cdot a \cdot b + cin \cdot \overline{a} \cdot b + cin \cdot a \cdot \overline{b} + cin \cdot a \cdot b,$$

które w wyniku minimalizacji i przekształceń logicznych można zaprezentować w następującej postaci:

$$s = a \oplus b \oplus cin; \quad (8.2)$$

$$cout = a \cdot b + (a \oplus b) \cdot cin,$$

gdzie znak $\oplus$ oznacza operację logiczną XOR.

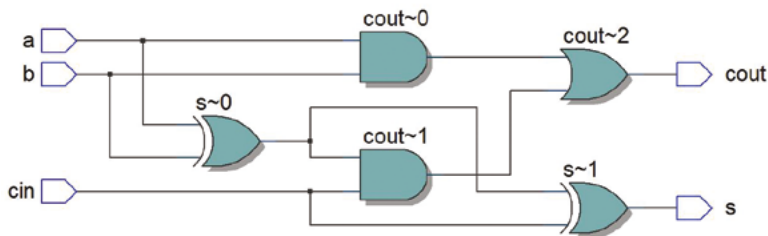
Przedstawiony sumator jednobitowy nazywany jest *sumatorem pełnym* (Full Adder – FA), jego opis w języku Verilog prezentuje się następująco:

```

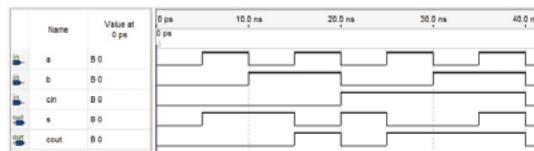
module full_adder(
    input a,b,cin,
    output s,cout);

    assign s = a ^ b ^ cin;
    assign cout = a & b | (a ^ b) & cin;
endmodule
```

Wyniki syntezy i symulacji modułu *full_adder* zostały pokazane na rysunkach 8.1 oraz 8.2.



RYS. 8.1. Wynik syntezy pełnego sumatora jednobitowego



RYS. 8.2. Wyniki symulacji pełnego sumatora jednobitowego

Poza sumatorami pełnymi często w strukturach sumatorów wielobitowych występują także *półsumatory* jednobitowe (*Half Adder* – HA) jako komponenty składowe. Półsumator, w przeciwieństwie do pełnego sumatora jednobitowego, nie ma sygnału wejściowego przeniesienia z poprzedniego bitu. Funkcjonowanie półsumatora można opisać za pomocą prostych równań logicznych:

$$\begin{aligned} s &= a \oplus b; \\ cout &= a \cdot b. \end{aligned} \quad (8.3)$$

Kod projektu półsumatora w języku Verilog można przedstawić następująco:

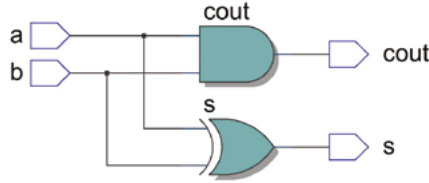
```

module half_adder(
    input a,b,
    output s,cout);

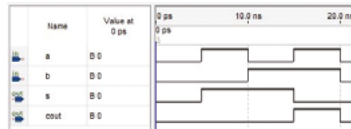
    assign s = a ^ b;
    assign cout = a & b;
endmodule

```

Wyniki syntezy i symulacji modułu *half_adder* przedstawiono na poniższych rysunkach 8.3 i 8.4.



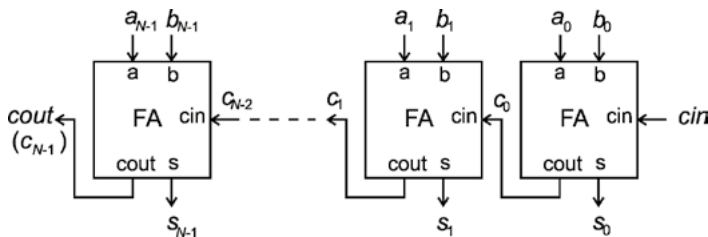
RYS. 8.3. Wynik syntezy półsumatora



RYS. 8.4. Wyniki symulacji półsumatora

8.1.2. Sumator RCA z przeniesieniami szeregowymi

Sumator z *przeniesieniami szeregowymi* (*Ripple-Carry Adder* – *RCA*) służy do dodawania dwóch słów binarnych $A = \{a_{N-1}, \dots, a_0\}$ i $B = \{b_{N-1}, \dots, b_0\}$ i składa się z N pełnych sumatorów jednobitowych (rysunek 8.5). Każdy pojedynczy pełny sumator realizuje funkcje sumy s_i i przeniesienia c_i dla jednego bitu sumatora na podstawie wartości bitów a_i i b_i oraz przeniesienia c_{i-1} z poprzedniej pozycji. Wyjście przeniesienia *cout* każdego pełnego sumatora jest podłączone do wejścia *cin* następnego sumatora, co tworzy długi łańcuch przeniesień szeregowych przez całą strukturę sumatora. Sygnał zewnętrzny z wejścia przeniesienia *cin* jest podawany na wejście *cin* najmłodszego bitu sumatora, a na wyjściu *cout* najstarszego bitu sumatora generowany jest zewnętrzny sygnał wyjściowy przeniesienia *cout*.



RYS. 8.5. Struktura N -bitowego sumatora z przeniesieniami szeregowymi

Sparametryzowany opis w języku Verilog sumatora z przeniesieniami szeregowymi można przedstawić następująco:

```
module ripple_carry_adder
    #(parameter N=4)
    (input [N-1:0] A,B,
```

```

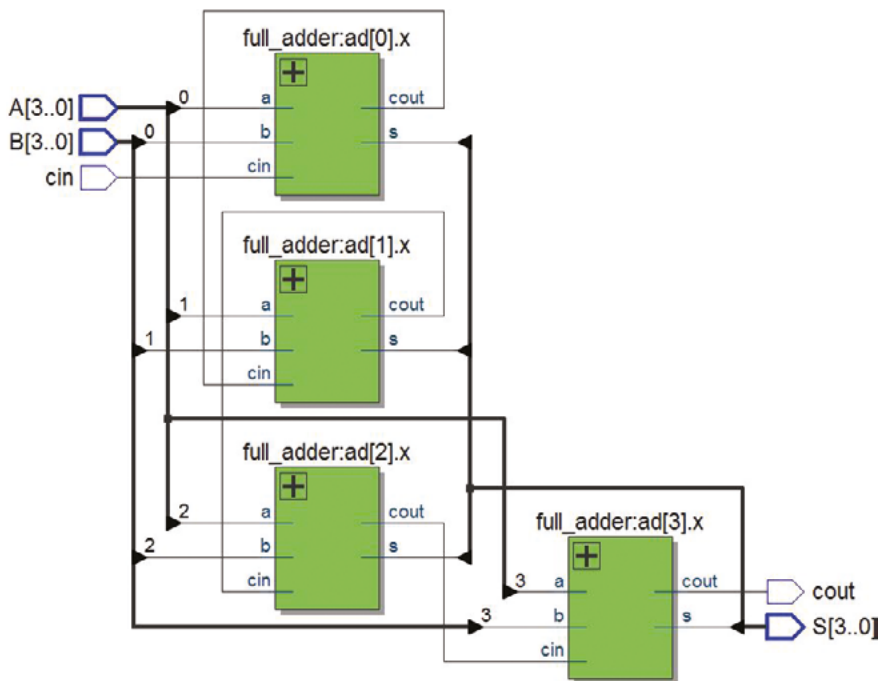
input cin,
output [N-1:0] S,
output cout);

wire [N-2:0] c;

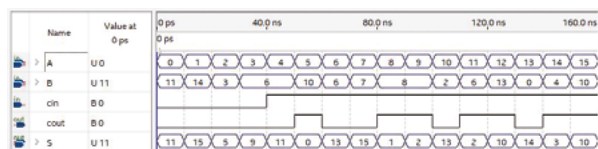
generate
genvar i;
for (i=0; i<N; i=i+1)
begin: ad
    case(i)
        0:          full_adder x(A[i],B[i],cin,S[i],c[i]);
        N-1:        full_adder x(A[i],B[i],c[i-1],S[i],cout);
        default:    full_adder x(A[i],B[i],c[i-1],S[i],c[i]);
    endcase
end
endgenerate
endmodule

```

Wyniki syntezy i symulacji modułu *ripple_carry_adder* pokazane zostały na rysunkach odpowiednio 8.6 i 8.7.



RYS. 8.6. Wynik syntezy sumatora z przeniesieniami szeregowymi dla $N=4$



RYS. 8.7. Wyniki symulacji sumatora z przeniesieniami szeregowymi

Sumator z przeniesieniami szeregowymi stanowi proste i ekonomiczne rozwiązanie, jednak jego główną wadą jest ograniczona szybkość działania. Czas uzyskania wyniku zależy od prędkości propagacji sygnału przeniesienia przez wszystkie bloki sumatora. W najbardziej niekorzystnym przypadku sygnał przeniesienia musi przejść od zewnętrznego wejścia *cin* przez wszystkie komórki sumatora do wyjścia ostatniego bloku s_{N-1} oraz zewnętrznego sygnału wyjściowego *cout*. Dodatkowo tego typu propagacja przeniesień uniemożliwia obliczenie wartości przeniesienia z *i*-tego bitu, dopóki nie zostanie wygenerowane przeniesienie z pozycji *i*–1.

Wielu naukowców zajmowało się opracowaniem struktur sumatorów, które mogą zwiększyć szybkość działania układu. W wyniku tych badań zaproponowano różne sposoby skrócenia czasu generowania przeniesień. Spośród nich najbardziej znane są:

- sumator z przeniesieniami równoległymi (*Carry-Lookahead Adder* – CLA);
- sumator z wyborem przeniesienia (*Carry-Select Adder* – CSEA);
- sumator z ominięciem przeniesienia (*Carry-Skip Adder* – CSA);
- sumator z warunkowym dodawaniem (*Conditional Sum Adder* – CSA).

8.1.3. Sumator z przeniesieniami równoległymi

8.1.3.1. Zredukowany pełny sumator

W sumatorze z przeniesieniami równoległymi, nazywanym także *sumatorem z przewidywaniem przeniesień*, wartości przeniesień ze wszystkich bitów $c_0, \dots, c_{N-1}$ generowane są jednocześnie z wyznaczeniem sum bitowych $s_0, \dots, s_{N-1}$. Innymi słowy, wszystkie obliczenia (poszczególne sumy oraz przeniesienia) wykonuje się równolegle. Aby to umożliwić, dla każdego bitu sumatora określone zostały dwie funkcje: *propagacji przeniesienia* p_i oraz *generacji przeniesienia* g_i :

$$p_i = a_i \oplus b_i; \quad (8.4)$$

$$g_i = a_i \cdot b_i.$$

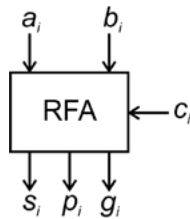
Poprawność wzorów (8.4) jest oczywista. Sygnał przeniesienia z poprzedniego bitu *i*–1 będzie przechodził przez *i*-ty sumator, jeśli tylko jeden z bitów, albo a_i , albo b_i , ma wartość 1 (funkcja propagacji przeniesienia). Z kolei sygnał przeniesienia zostanie wygenerowany w *i*-tej pozycji, jeśli jednocześnie oba bity a_i i b_i są ustawione na 1 (funkcja generacji przeniesienia).

Z uwzględnieniem wzorów (8.4) wyrażenia logiczne (8.2) można przekształcić do postaci:

$$s_i = p_i \oplus c_i; \quad (8.5)$$

$$c_{i+1} = g_i + p_i \cdot c_i.$$

Jeden bit sumatora z przeniesieniami równoległymi jest realizowany w postaci *zredukowanego pełnego sumatora* (*Reduced Full Adder – RFA*), który wykonuje funkcje (8.4) oraz pierwsze równanie z (8.5). Zredukowany pełny sumator (rysunek 8.8) na podstawie wartości bitów a_i i b_i oraz przeniesienia z poprzedniej pozycji c_i generuje dla i -go bloku trzy funkcje: sumy s_i , propagacji przeniesienia p_i i generacji przeniesienia g_i . Warto zauważyć, że zredukowany pełny sumator nie przekazuje bezpośrednio przeniesienia do następnego bitu.



RYS. 8.8. Zredukowany pełny sumator

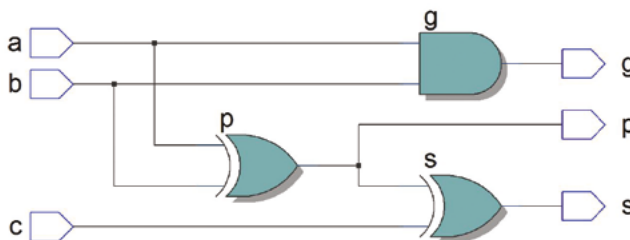
Poniżej przytoczono kod projektu w języku Verilog dla zredukowanego pełnego sumatora. Wyniki syntezy i symulacji projektu przedstawione zostały na rysunkach 8.9 i 8.10.

```

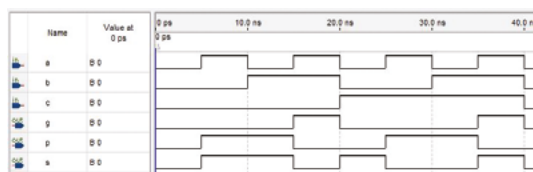
module reduced_full_adder(
    input a,b,c,
    output s,p,g);

    assign p = a ^ b;
    assign g = a & b;
    assign s = p ^ c;
endmodule

```



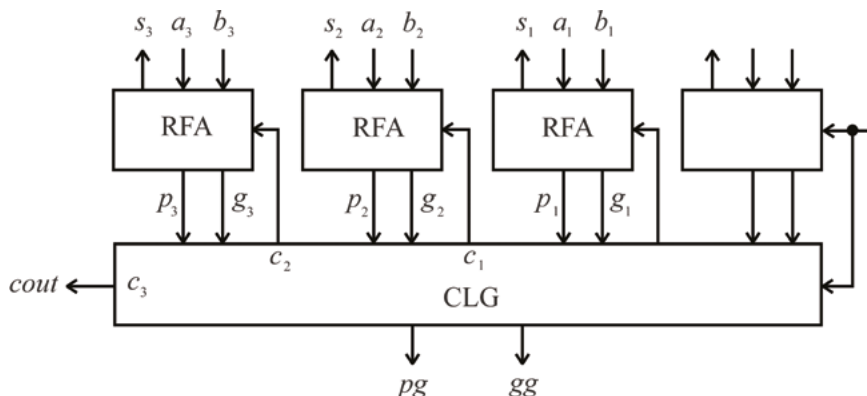
RYS. 8.9. Wynik syntezy zredukowanego pełnego sumatora RFA



RYS. 8.10. Wyniki symulacji zredukowanego pełnego sumatora RFA

8.1.3.2. Generator przeniesień

Do skonstruowania sumatora z przeniesieniami równoległymi oprócz sumatorów zredukowanych wymagana jest implementacja dodatkowego bloku *generatora przeniesień* (Carry Lookahead Generator – CLG). Na podstawie sygnału przeniesienia cin oraz sygnałów propagacji p_i i generacji g_i przeniesień blok ten generuje sygnały przeniesień $c_0, \dots, c_{N-1}$ do poszczególnych bitów sumatora. Schemat strukturalny 4-bitowego sumatora z przeniesieniami równoległymi został przedstawiony na rysunku 8.11.



RYS. 8.11. Struktura 4-bitowego sumatora z przeniesieniami równoległymi

Na rysunku 8.11 możemy zauważyć, że schemat generatora CLG generuje przeniesienia do kolejnych bitów $c_0, \dots, c_{N-1}$, a także przeniesienie $cout$ (c_3) do następnej 4-bitowej grupy sumatorów oraz grupowe sygnały propagacji przeniesienia pg oraz generacji przeniesienia gg . Wyrażenia logiczne funkcji przeniesień generowanych przez blok CLG dla 4-bitowej grupy przedstawiają się następująco:

$$\begin{aligned}
 c_0 &= g_0 + p_0 \cdot cin; \\
 c_1 &= g_1 + p_1 \cdot c_0; \\
 c_2 &= g_2 + p_2 \cdot c_1; \\
 c_3 &= g_3 + p_3 \cdot c_2.
 \end{aligned}
 \tag{8.6}$$

Wykonując podstawienia c_0 do wyrażenia funkcji c_1 , c_1 – do c_2 oraz c_2 – do c_3 , otrzymujemy następujące wyrażenia logiczne funkcji przeniesień:

$$\begin{aligned}c_0 &= g_0 + p_0 \cdot cin; \\c_1 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot cin; \\c_2 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot cin; \\c_3 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot cin.\end{aligned}\tag{8.7}$$

W równaniach (8.7) wszystkie funkcje przeniesień $c_0, \dots, c_{N-1}$ zależą tylko od przeniesienia wejściowego cin oraz funkcji propagacji i generacji przeniesień produkowanych w poszczególnych sumatorach zredukowanych. W związku z tym nie ma potrzeby przekazywania sygnałów przeniesień pomiędzy poszczególnymi sumatorami bitowymi, co oznacza, że wszystkie przeniesienia są obliczane równocześnie.

Oprócz sygnałów przeniesień układ generatora CLG wytwarza także grupowe sygnały propagacji przeniesienia pg i generacji przeniesienia gg . Sygnał grupowy propagacji przeniesienia pg przyjmuje wartość jedynki logicznej, gdy sygnał wejściowy przeniesienia cin przechodzi przez wszystkie bity grupy. Natomiast sygnał grupowy generacji przeniesienia gg ustawiany jest na 1, jeśli w danej grupie bitów generowany jest sygnał przeniesienia $cout$ (c_3). W związku z tym funkcje te można wyrazić za pomocą następujących równań logicznych:

$$\begin{aligned}pg &= p_0 \cdot p_1 \cdot p_2 \cdot p_3; \\gg &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0.\end{aligned}\tag{8.8}$$

Blok generatora przeniesień CLG dla 4-bitowej grupy sumatorów realizuje funkcje logiczne opisane wyrażeniami (8.7) i (8.8), a jego opis w języku Verilog można przedstawić następująco:

```
module carry_lookahead_generator_4_bits(
    input [3:0] p,g,
    input cin,
    output [3:0] c,
    output pg, gg);    // sygnały grupowe propagacji i generacji przeniesień

    assign c[0] = g[0] | cin & p[0];
    assign c[1] = g[1] | g[0] & p[1] | cin & p[0] & p[1];
    assign c[2] = g[2] | g[1] & p[2] | g[0] & p[1] & p[2] | cin & p[0] & p[1] & p[2];
    assign c[3] = g[3] | g[2] & p[3] | g[1] & p[2] & p[3] | g[0] & p[1] & p[2] & p[3] |
                cin & p[0] & p[1] & p[2] & p[3];

    assign pg = p[0]&p[1]&p[2]&p[3];
    assign gg =  g[3] |
```

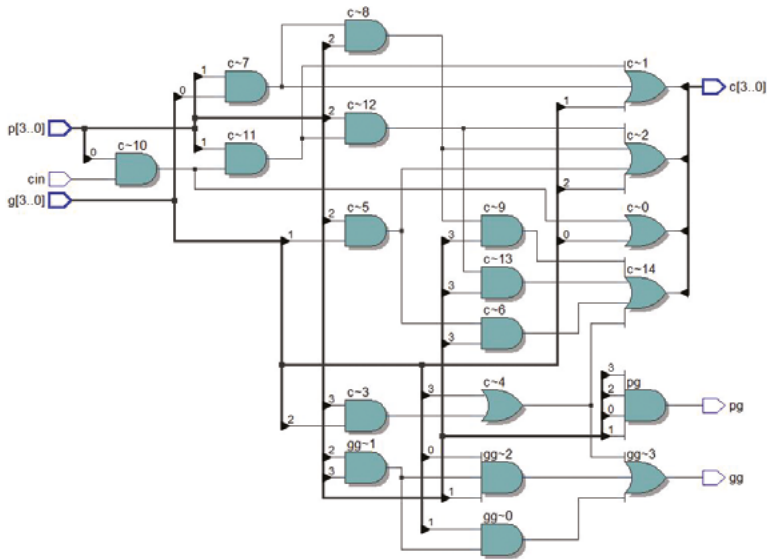
```

p[3]&g[2] |
p[3]&p[2]&g[1] |
p[3]&p[2]&p[1]&g[0];

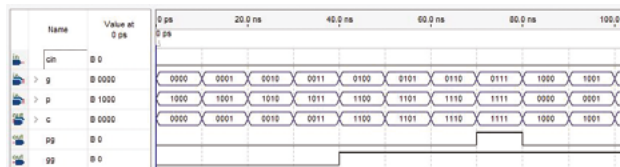
```

endmodule

Wyniki syntezy i symulacji bloku generatora przeniesień przedstawione zostały na rysunkach 8.12 i 8.13.



RYS. 8.12. Wynik syntezy generatora przeniesień dla 4-bitowej grupy sumatorów



RYS. 8.13. Wyniki symulacji generatora przeniesień dla 4-bitowej grupy sumatorów

8.1.3.3. Czterobitowy sumator z przeniesieniami równoległymi

Kod 4-bitowego sumatora z przeniesieniami równoległymi, którego schemat pokazany został na rysunku 8.11, można przedstawić następująco:

```

module carry_lookahead_adder_4_bits(
    input [3:0] a,b,
    input cin,
    output [3:0] s,
    output cout,

```

```
output pg, gg);
```

```
wire [3:0] p,g;
```

```
wire [3:0] c;
```

```
reduced_full_adder ex0(a[0],b[0],cin,s[0],p[0],g[0]);
```

```
reduced_full_adder ex1(a[1],b[1],c[0],s[1],p[1],g[1]);
```

```
reduced_full_adder ex2(a[2],b[2],c[1],s[2],p[2],g[2]);
```

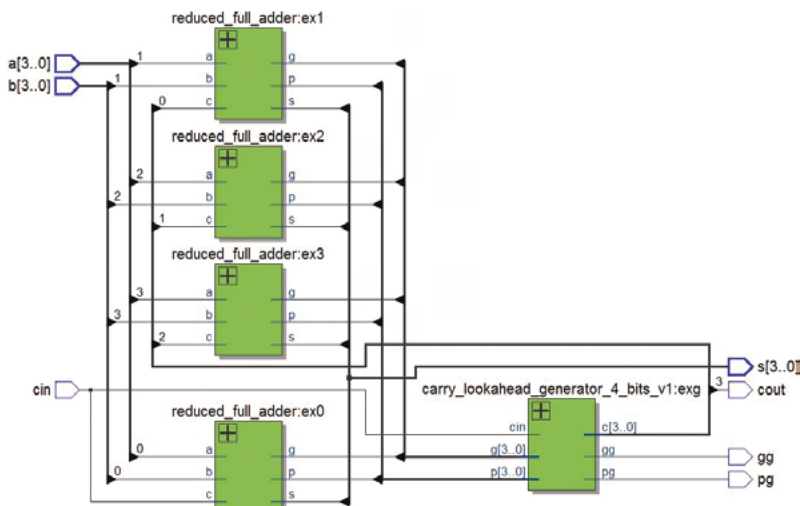
```
reduced_full_adder ex3(a[3],b[3],c[2],s[3],p[3],g[3]);
```

```
carry_lookahead_generator_4_bits exg(p,g,cin,c,pg,gg);
```

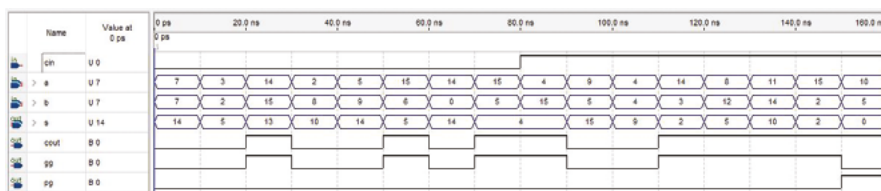
```
assign cout = c[3];
```

```
endmodule
```

Wyniki syntezy i symulacji 4-bitowego sumatora z przeniesieniami równoległymi zaprezentowano na rysunkach 8.14 oraz 8.15.



RYS. 8.14. Wynik syntezy 4-bitowego sumatora z przeniesieniami równoległymi

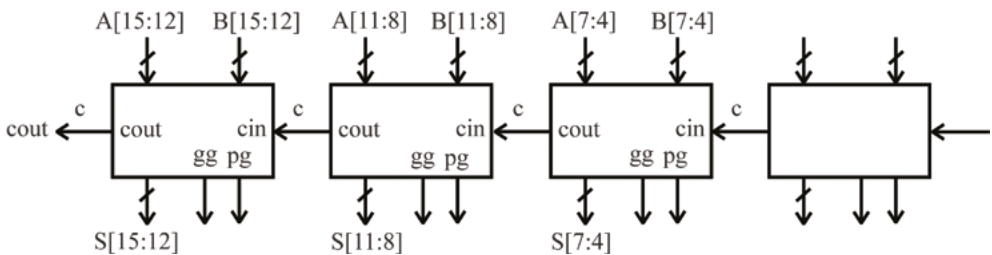


RYS. 8.15. Wyniki symulacji 4-bitowego sumatora z przeniesieniami równoległymi

8.1.3.4. Szesnastobitowy sumator z przeniesieniami równoległymi

Wraz ze wzrostem rozmiaru sumatora rośnie złożoność funkcji realizowanych generatorem przeniesień CLG, które są określane na podstawie wyrażeń logicznych (8.7) i (8.8). Syntezator implementuje te złożone funkcje w postaci wielopoziomowych układów kombinacyjnych, co prowadzi do wzrostu opóźnień propagacji sygnałów. To z kolei skutkuje utratą korzyści wynikających z zastosowania schematu szybkich równoległych przeniesień. Z tego powodu wielobitowe sumatory z przeniesieniami równoległymi są budowane z bloków o mniejszym rozmiarze, z realizacją schematów przeniesień pomiędzy nimi. W dalszej części przedstawione zostaną dwa projekty 16-bitowego sumatora zestawionego z 4-bitowych bloków z przeniesieniami równoległymi: jeden z realizacją szeregowych, a drugi równoległych przeniesień pomiędzy blokami.

Schemat strukturalny 16-bitowego sumatora zbudowanego z bloków z przeniesieniami równoległymi, z realizacją przeniesień szeregowych pomiędzy blokami, został przedstawiony na rysunku 8.16.



RYS. 8.16. Struktura 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień szeregowych pomiędzy blokami

W schemacie na rysunku 8.16 każdy z bloków składowych reprezentuje 4-bitowy sumator z przeniesieniami równoległymi CLA. Sygnały przeniesienia pomiędzy blokami przekazywane są szeregowo, zgodnie z modelem sumatora RCA. Poniżej znajduje się kod odpowiadający temu schematowi.

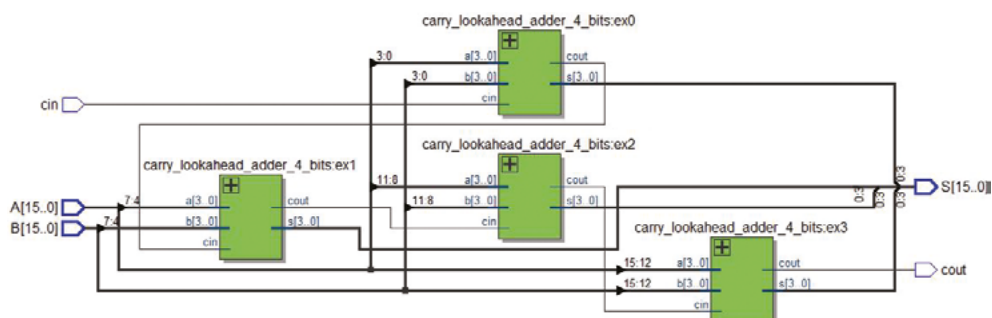
```
module carry_lookahead_adder_16_bits_v1(
    input [15:0] A,B,
    input cin,
    output [15:0] S,
    output cout);

    wire [3:1] c;

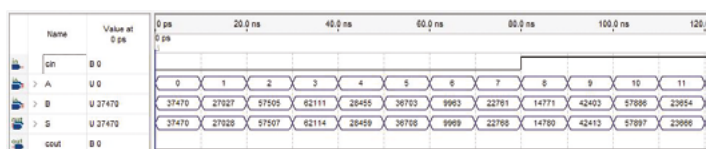
    carry_lookahead_adder_4_bits ex0(A[3:0],B[3:0],cin,S[3:0],c[1]);
    carry_lookahead_adder_4_bits ex1(A[7:4],B[7:4],c[1],S[7:4],c[2]);
    carry_lookahead_adder_4_bits ex2(A[11:8],B[11:8],c[2],S[11:8],c[3]);
    carry_lookahead_adder_4_bits ex3(A[15:12],B[15:12],c[3],S[15:12],cout);

endmodule
```

Wyniki syntezy i symulacji 16-bitowego sumatora zbudowanego z 4-bitowych sumatorów z przeniesieniami równoległymi i realizacją przeniesień szeregowych między blokami zostały zaprezentowane na rysunkach 8.17 i 8.18.

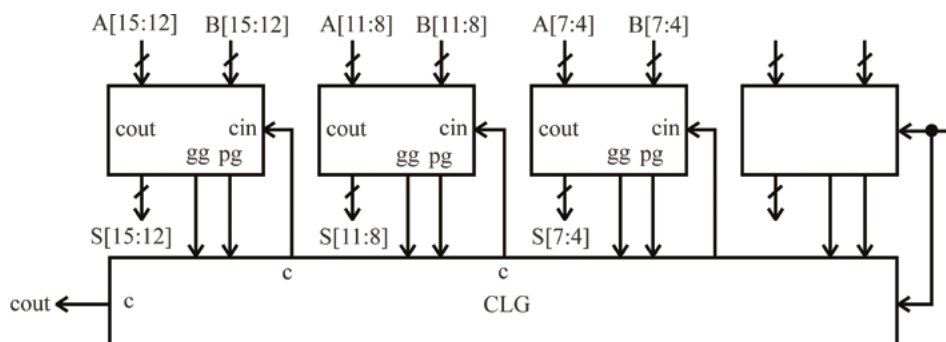


RYS. 8.17. Wynik syntezy 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień szeregowych pomiędzy blokami



RYS. 8.18. Wyniki symulacji 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień szeregowych pomiędzy blokami

Na rysunku 8.19 przedstawiony został schemat strukturalny 16-bitowego sumatora zbudowanego z 4-bitowych bloków z przeniesieniami równoległymi, gdzie realizowane są przeniesienia równoległe pomiędzy tymi blokami.



RYS. 8.19. Struktura 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami

W układzie na rysunku 8.19 do generowania sygnałów przeniesień pomiędzy blokami wykorzystywany jest taki sam schemat generatora przeniesień jednoczesnych CLG, który występuje w 4-bitowych blokach sumatorów. Generator CLG pokazany na rysunku 8.19 oblicza przeniesienia $c_0, \dots, c_3$ na podstawie grupowych sygnałów propagacji i generacji przeniesień, które są formowane w blokach generatorów CLG poszczególnych 4-bitowych sumatorów CLA. Ponieważ 16-bitowy sumator składa się z czterech bloków, do realizacji generatora CLG w układzie z rysunku 8.19 można bez zmian wykorzystać blok generatora przeniesień równoległych przedstawiony na rysunku 8.11.

Kod 16-bitowego sumatora zbudowanego z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami można przedstawić następująco:

```
module carry_lookahead_adder_16_bits_v2(
    input [15:0] A,B,
    input cin,
    output [15:0] S,
    output cout,
    output pg,gg);

    wire [3:0] p,g,c;

    carry_lookahead_adder_4_bits ex0(A[3:0],B[3:0],cin,S[3:0],,p[0],g[0]);
    carry_lookahead_adder_4_bits ex1(A[7:4],B[7:4],c[0],S[7:4],,p[1],g[1]);
    carry_lookahead_adder_4_bits ex2(A[11:8],B[11:8],c[1],S[11:8],,p[2],g[2]);
    carry_lookahead_adder_4_bits ex3(A[15:12],B[15:12],c[2],S[15:12],,p[3],g[3]);

    carry_lookahead_generator_4_bits exg(p,g,cin,c,pg,gg);

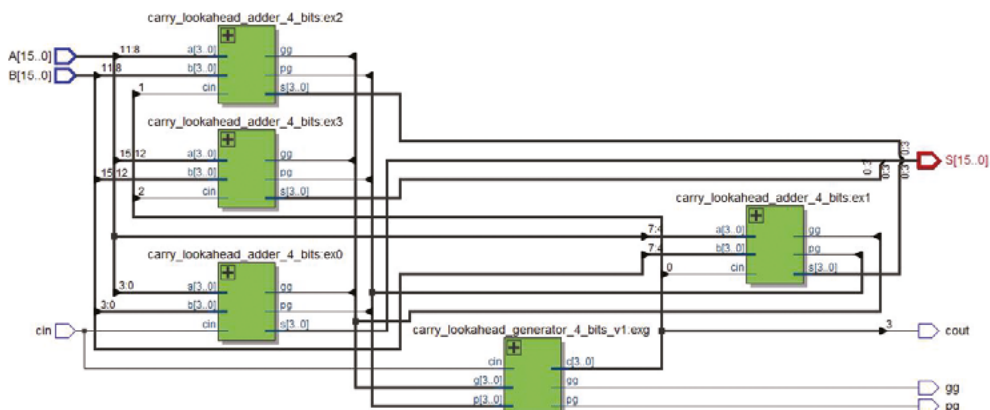
    assign cout = c[3];

endmodule
```

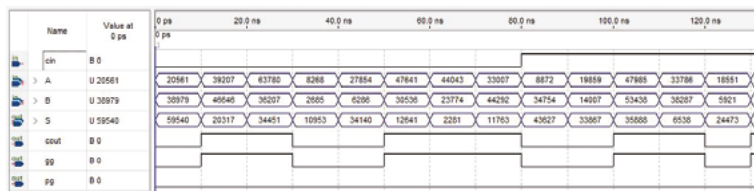
Wyniki syntezy i symulacji modułu *carry_lookahead_adder_16_bits_v2* zaprezentowane zostały na rysunkach 8.20 oraz 8.21.

Przedstawione wyżej struktury sumatorów z przeniesieniami szeregowymi RCA oraz z przeniesieniami równoległymi CLA są dwoma skrajnymi rozwiązaniami pod względem szybkości działania układu. Sumator z przeniesieniami szeregowymi RCA cechuje się najwolniejszym działaniem, ale jednocześnie ma najniższy koszt realizacji sprzętowej. Z kolei sumator z przeniesieniami równoległymi CLA działa najszybciej, ale wymaga największych kosztów realizacji. Oprócz tego przy projektowaniu sumatorów CLA dużego rozmiaru pojawia się problem związany z wyborem optymalnego rozmiaru bloku. Dla przykładu proponujemy czytelnikowi wyznaczenie

optymalnego rozmiaru bloku przy realizacji 64-bitowego sumatora z przeniesieniami równoległymi.



RYS. 8.20. Wynik syntezy 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami



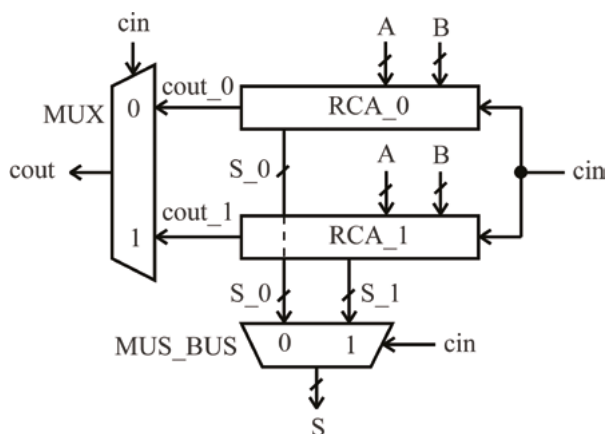
RYS. 8.21. Wyniki symulacji 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami

Wszystkie inne znane sposoby realizacji sumatorów dają wyniki pośrednie pomiędzy tymi dwoma rozwiązaniami. W kolejnych podrozdziałach krótko omówimy inne wybrane struktury sumatorów.

8.1.4. Sumator z wyborem przeniesienia

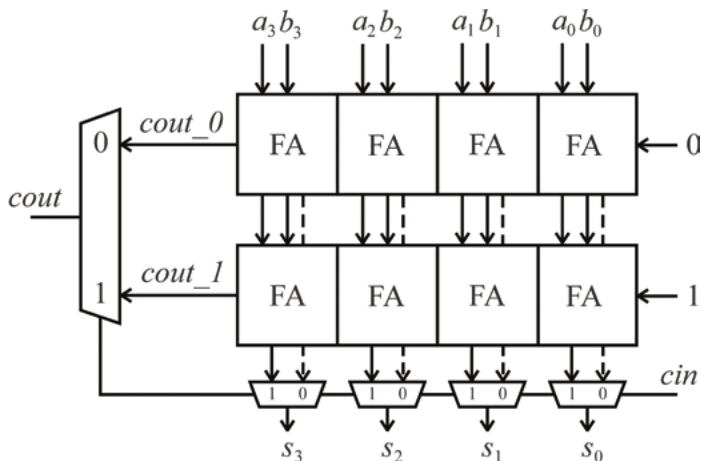
Sumator z wyborem przeniesienia (Carry-Select Adder – CSEA) składa się z sekwencyjnie połączonych bloków, co przedstawiono na rysunku 8.22. Każdy blok zawiera dwa identyczne sumatory z przeniesieniami szeregowymi RCA. Pierwszy sumator RCA_0 oblicza sumę S_0 i przeniesienie $cout_0$, zakładając, że przeniesienie wejściowe cin wynosi 0, natomiast drugi sumator RCA_1 oblicza sumę S_1 i przeniesienie $cout_1$, przyjmując, że wartość przeniesienia na wejściu cin to 1. Do wyjść każdego z bloków sumatora są podłączone dwa multiplexery 2:1 – grupowy oraz jednobi-towy. Multiplexer grupowy MUX_BUS wybiera odpowiednią sumę S , w zależności

od wartości przeniesienia na wejściu *cin*, natomiast jednobitowy multiplekser MUX wybiera właściwy sygnał przeniesienia *cout*, który jest przekazywany do następnego bloku.



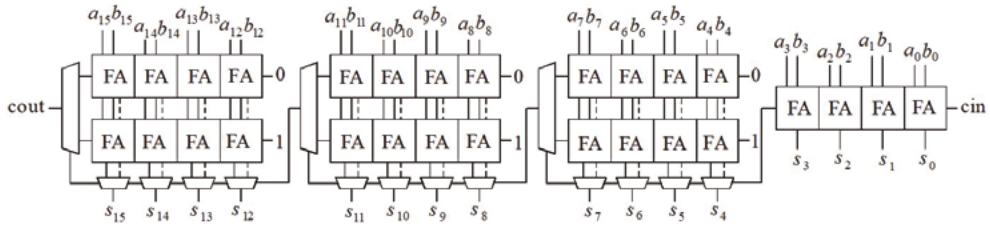
RYS. 8.22. Blok sumatora z wyborem przeniesienia CSEA

Struktura 4-bitowego bloku sumatora z wyborem przeniesienia CSEA pokazana została na rysunku 8.23.



RYS. 8.23. Struktura 4-bitowego bloku sumatora z wyborem przeniesienia CSEA

W strukturze 16-bitowego sumatora z wyborem przeniesienia CSEA (rysunek 8.24) jako najmłodszy blok występuje zwykły 4-bitowy sumator z przeniesieniami szeregowymi RCA, ponieważ wartość przeniesienia na wejściu *cin* dla niego jest znana.



RYS. 8.24. Struktura 16-bitowego sumatora z wyborem przeniesienia CSEA zestawionego z 4-bitowych bloków

W języku Verilog jeden 4-bitowy blok sumatora z wyborem przeniesienia CSEA można opisać następująco:

```

module carry_select_adder_4_bits(
    input [3:0] A,B,
    input cin,
    output [3:0] S,
    output cout);

    wire [3:0] S_0,S_1;
    wire cout_0,cout_1;

    ripple_carry_adder #(4) ex0(A,B,1'b0,S_0,cout_0);
    ripple_carry_adder #(4) ex1(A,B,1'b1,S_1,cout_1);

    MUX_2_1_if exm(cout_0,cout_1,cin,cout);
    MUX_2_1_N_bits #(4) exm4(S_0,S_1,cin,S);
endmodule

```

Moduł *carry_select_adder_4_bits* wykorzystamy do opisu 16-bitowego sumatora CSEA przedstawionego na rysunku 8.24:

```

module carry_select_adder_16_bits(
    input [15:0] A,B,
    input cin,
    output [15:0] S,
    output cout);

    wire c1,c2,c3;

    ripple_carry_adder #(4) ex0(A[3:0],B[3:0],cin,S[3:0],c1);
    carry_select_adder_4_bits ex1(A[7:4],B[7:4],c1,S[7:4],c2);

```

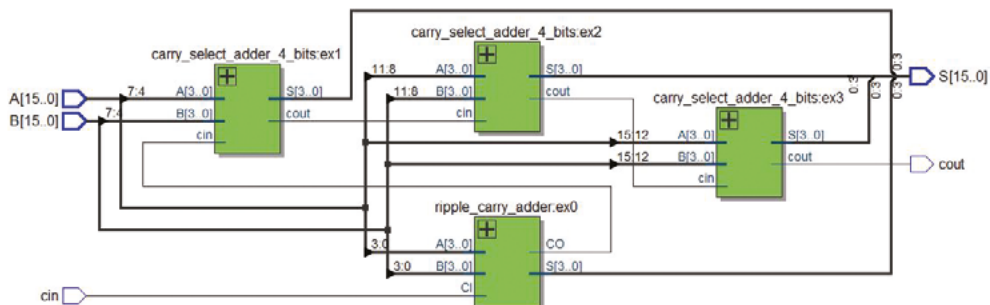
```

carry_select_adder_4_bits ex2(A[11:8],B[11:8],c2,S[11:8],c3);
carry_select_adder_4_bits ex3(A[15:12],B[15:12],c3,S[15:12],cout);

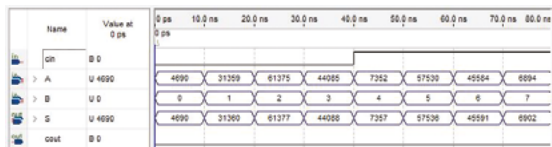
```

endmodule

Wyniki syntezy i symulacji projektu *carry_select_adder_16_bits* pokazane zostały na rysunkach 8.25 i 8.26.



RYS. 8.25. Wynik syntezy 16-bitowego sumatora z wyborem przeniesienia CSEA zbudowanego z 4-bitowych bloków



RYS. 8.26. Wyniki symulacji 16-bitowego sumatora z wyborem przeniesienia CSEA zbudowanego z 4-bitowych bloków

Analiza schematu na rysunku 8.24 sugeruje, że koszt realizacji sumatora z wyborem przeniesienia będzie około dwukrotnie wyższy niż koszt realizacji analogicznego sumatora z przeniesieniami szeregowymi RCA. W tym schemacie każdy blok (oprócz pierwszego) składa się z dwóch sumatorów RCA. Dodatkowo każdy blok (poza pierwszym) zawiera grupowy oraz jednobitowy multiplekser. Przyspieszenie obliczeń w tym przypadku wynika z faktu, że szeregowe przekazywanie przeniesień występuje tylko wewnątrz bloków, a sumy i sygnały wyjściowe przeniesień są generowane równocześnie we wszystkich blokach.

Dodatkowe opóźnienie w schemacie sumatora CSEA wynika z przekazywania przeniesień pomiędzy blokami: wartość przeniesienia na wejściu ostatniego bloku nie może być obliczona wcześniej, niż zostaną przekazane przeniesienia z poprzednich bloków. Stąd pojawiła się koncepcja budowy sumatorów CSEA z bloków zmiennego rozmiaru (*variable-sized adder*). Starsze bloki w takiej strukturze mają więcej czasu na propagację przeniesień wewnątrz bloku, dlatego ich rozmiary można zwiększyć. Na przykład 16-bitowy sumator z wyborem przeniesienia CSEA można zbudować

z bloków zmiennego rozmiaru według schematu: 2-2-3-4-5. W takim układzie pierwszy (najmłodszy) blok jest 2-bitowy, drugi również, trzeci – 3-bitowy, czwarty – 4-bitowy, a piąty – 5-bitowy.

Kod 16-bitowego sumatora z wyborem przeniesień CSEA zbudowanego z bloków zmiennego rozmiaru 2-2-3-4-5 można przedstawić następująco:

```
module CSEA_16_var_blocks(
    input [15:0] A,B,
    input cin,
    output [15:0] S,
    output cout);

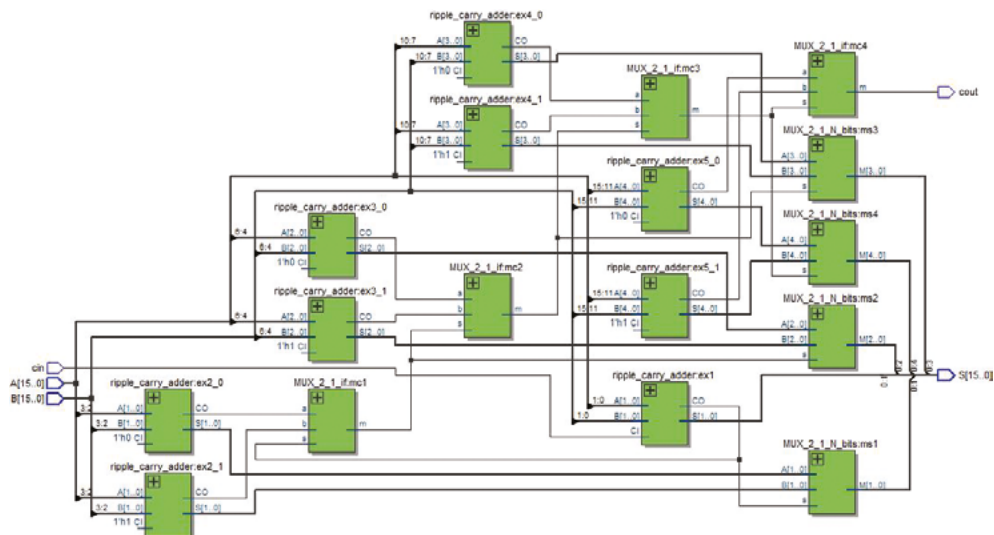
    wire c1,c2,c3,c4;
    wire c2_0,c2_1,c3_0,c3_1,c4_0,c4_1,c5_0,c5_1;
    wire [1:0] S2_0,S2_1;
    wire [2:0] S3_0,S3_1;
    wire [3:0] S4_0,S4_1;
    wire [4:0] S5_0,S5_1;

    ripple_carry_adder #(2) ex1(A[1:0],B[1:0],cin,S[1:0],c1);
    ripple_carry_adder #(2) ex2_0(A[3:2],B[3:2],1'b0,S2_0,c2_0);
    ripple_carry_adder #(2) ex2_1(A[3:2],B[3:2],1'b1,S2_1,c2_1);
    ripple_carry_adder #(3) ex3_0(A[6:4],B[6:4],1'b0,S3_0,c3_0);
    ripple_carry_adder #(3) ex3_1(A[6:4],B[6:4],1'b1,S3_1,c3_1);
    ripple_carry_adder #(4) ex4_0(A[10:7],B[10:7],1'b0,S4_0,c4_0);
    ripple_carry_adder #(4) ex4_1(A[10:7],B[10:7],1'b1,S4_1,c4_1);
    ripple_carry_adder #(5) ex5_0(A[15:11],B[15:11],1'b0,S5_0,c5_0);
    ripple_carry_adder #(5) ex5_1(A[15:11],B[15:11],1'b1,S5_1,c5_1);

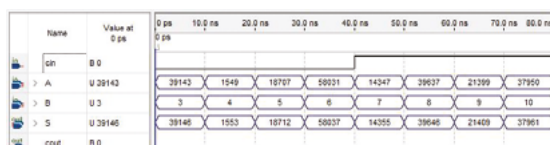
    MUX_2_1_if mc1(c2_0,c2_1,c1,c2);
    MUX_2_1_if mc2(c3_0,c3_1,c2,c3);
    MUX_2_1_if mc3(c4_0,c4_1,c3,c4);
    MUX_2_1_if mc4(c5_0,c5_1,c4,cout);

    MUX_2_1_N_bits #(2) ms1(S2_0,S2_1,c1,S[3:2]);
    MUX_2_1_N_bits #(3) ms2(S3_0,S3_1,c2,S[6:4]);
    MUX_2_1_N_bits #(4) ms3(S4_0,S4_1,c3,S[10:7]);
    MUX_2_1_N_bits #(5) ms4(S5_0,S5_1,c4,S[15:11]);
endmodule
```

Wyniki syntezy i symulacji projektu *CSEA_16_var_blocks* pokazane zostały na rysunkach 8.27 oraz 8.28.



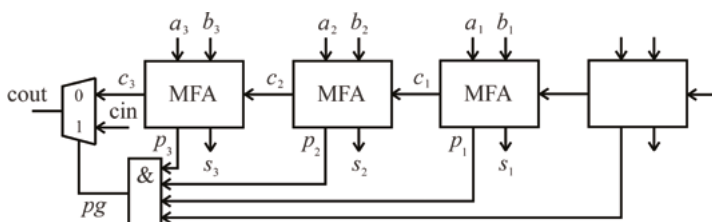
RYS. 8.27. Wynik syntezy 16-bitowego sumatora z wyborem przeniesienia CSEA utworzonego z bloków zmiennego rozmiaru



RYS. 8.28. Wyniki symulacji 16-bitowego sumatora z wyborem przeniesienia CSEA utworzonego z bloków zmiennego rozmiaru

8.1.5. Sumator z przeskokiem przeniesień

Sumator z przeskokiem przeniesień (Carry-Skip Adder – CSA), nazywany także sumatorem z ominięciem przeniesień (carry-bypass adder) umożliwia zwiększenie szybkości działania sumatora przy mniejszej złożoności realizacji sprzętowej w porównaniu do sumatora z wyborem przeniesienia CSEA. Czterobitowy blok sumatora z przeskokiem przeniesień CSA przedstawiony został na rysunku 8.29.



RYS. 8.29. 4-bitowy blok sumatora z przeskokiem przeniesień CSA

W strukturze przedstawionej na rysunku 8.29 każdy *zmodyfikowany pełny sumator* (*Modified Full Adder – MFA*) dla jednego bitu sumatora CSEA generuje funkcje sumy s_i , przeniesienia c_i oraz propagacji przeniesienia p_i . Sygnał grupowy propagacji przeniesienia pg obliczany jest na podstawie sygnałów propagacji przeniesień z poszczególnych bitów: $pg = p_0 \cdot p_1 \cdot p_2 \cdot p_3$. Sygnał pg steruje dwuwejściowym jednobitowym multiplekserem, który przekazuje na wyjście przeniesienia $cout$ jeden z sygnałów cin lub c_3 . W porównaniu do bloku sumatora z wyborem przeniesienia CSEA (rysunek 8.22), w schemacie sumatora z przeskokiem przeniesień CSKA nie są podwajane zasoby sprzętowe do realizacji funkcji sumy.

Funkcjonowanie zmodyfikowanego pełnego sumatora MFA można opisać za pomocą następujących równań logicznych:

$$\begin{aligned} s &= p \oplus cin; \\ cout &= a \cdot b + p \cdot cin; \\ p &= a \oplus b. \end{aligned} \tag{8.9}$$

Kod bloku MFA w języku Verilog można przedstawić następująco:

```
module modified_full_adder(
    input a,b,cin,
    output s,cout,p);

    assign p = a ^ b;
    assign s = p ^ cin;
    assign cout = a & b | p & cin;
endmodule
```

Kod 4-bitowego bloku sumatora z przeskokiem przeniesień CSKA został przedstawiony poniżej:

```
module carry_skip_adder_4_bits(
    input [3:0] A,B,
    input cin,
    output [3:0] S,
    output cout);

    wire [3:0] c;
    wire sel;
    wire [3:0] P;

    modified_full_adder ex0(A[0],B[0],cin,S[0],c[0],P[0]);
    modified_full_adder ex1(A[1],B[1],c1,S[1],c[1],P[1]);
    modified_full_adder ex2(A[2],B[2],c2,S[2],c[2],P[2]);
    modified_full_adder ex3(A[3],B[3],c3,S[3],c[3],P[3]);
```



```
assign sel = P[0]&P[1]&P[2]&P[3];
```

```
MUX_2_1_if mux(c4,cin,sel,cout);
```

```
endmodule
```

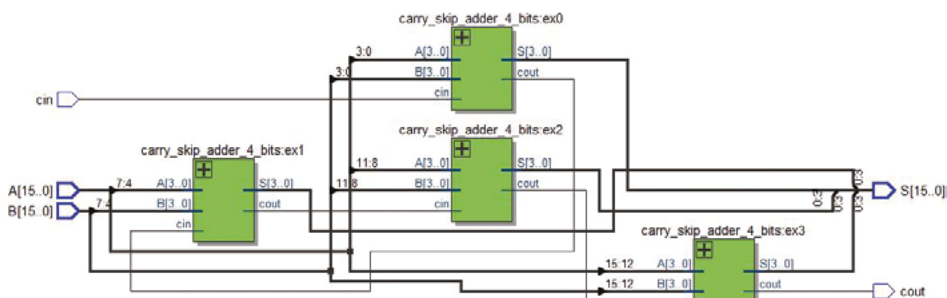
Kod 16-bitowego sumatora z przeskokiem przeniesień CSKA wykorzystuje instancje opisanych wyżej modułów:

```
module carry_skip_adder_16_bits(
    input [15:0] A,B,
    input cin,
    output [15:0] S,
    output cout);

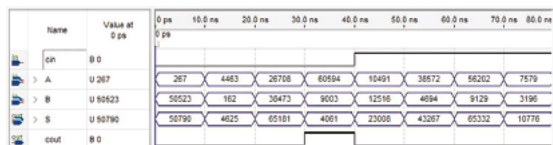
    wire c1,c2,c3;

    carry_skip_adder_4_bits ex0(A[3:0],B[3:0],cin,S[3:0],c1);
    carry_skip_adder_4_bits ex1(A[7:4],B[7:4],c1,S[7:4],c2);
    carry_skip_adder_4_bits ex2(A[11:8],B[11:8],c2,S[11:8],c3);
    carry_skip_adder_4_bits ex3(A[15:12],B[15:12],c3,S[15:12],cout);
endmodule
```

Wyniki syntezy i symulacji projektu *carry_skip_adder_16_bits* przedstawiono na rysunkach 8.30 i 8.31.



RYS. 8.30. Wynik syntezy 16-bitowego sumatora z przeskokiem przeniesień CSKA



RYS. 8.31. Wyniki symulacji 16-bitowego sumatora z przeskokiem przeniesień CSKA

8.1.6. Sumator z sumami warunkowymi

Sumator z sumami warunkowymi (Conditional sum adder – CSA) jest strukturą rekurencyjną, która opiera się na sumatorze z wyborem przeniesienia CSEA. W sumatorze CSA również obliczane są dwie sumy i dwa przeniesienia dla założonych wartości przeniesienia początkowego 0 i 1. Następnie za pomocą multiplexerów wybierany jest poprawny wynik na podstawie obliczonej wartości przeniesienia z poprzedniego bloku. Cały proces budowania struktury sumatora rozpoczyna się od dwóch bitów. Dalej wszystko jest powtarzane dla czterech bitów, ośmiu bitów itd. Dla N -bitowego sumatora, gdzie N nie jest potęgą dwójki, cały sumator zostaje podzielony na dwie grupy rozmiarem $N/2$ bitów, które z kolei są dzielone na mniejsze grupy.

Projektowanie 16-bitowego sumatora z sumami warunkowymi CSA zaczynamy od opisu dwubitowego sumatora CSA:

```
module CSA_2(  
    input [1:0] A,B,  
    input cin,  
    output [1:0] S,  
    output cout);  
  
    wire S_0,S_1;  
    wire c,c_0,c_1;  
  
    full_adder ex_0(A[0],B[0],cin,S[0],c);  
    full_adder ex_1_0(A[1],B[1],1'b0,S_0,c_0);  
    full_adder ex_1_1(A[1],B[1],1'b1,S_1,c_1);  
  
    MUX_2_1_if muxs(S_0,S_1,c,S[1]);  
    MUX_2_1_if muxc(c_0,c_1,c,cout);  
endmodule
```

Przy użyciu instancji modułu CSA_2 można opisać 4-bitowy sumator CSA:

```
module CSA_4(  
    input [3:0] A,B,  
    input cin,  
    output [3:0] S,  
    output cout);  
  
    wire [1:0] S_0,S_1;  
    wire c,c_0,c_1;  
  
    CSA_2 ex_1_0(A[1:0],B[1:0],cin,S[1:0],c);
```

```
CSA_2 ex_3_2_0(A[3:2],B[3:2],1'b0,S_0,c_0);
CSA_2 ex_3_2_1(A[3:2],B[3:2],1'b1,S_1,c_1);
```

```
MUX_2_1_N_bits #(2) muxs(S_0,S_1,c,S[3:2]);
MUX_2_1_if muxc(c_0,c_1,c,cout);
```

endmodule

W podobny sposób opisywany jest 8-bitowy sumator CSA:

```
module CSA_8(
    input [7:0] A,B,
    input cin,
    output [7:0] S,
    output cout);

    wire [3:0] S_0,S_1;
    wire c,c_0,c_1;

    CSA_4 ex_3_0(A[3:0],B[3:0],cin,S[3:0],c);
    CSA_4 ex_7_4_0(A[7:4],B[7:4],1'b0,S_0,c_0);
    CSA_4 ex_7_4_1(A[7:4],B[7:4],1'b1,S_1,c_1);

    MUX_2_1_N_bits #(4) muxs(S_0,S_1,c,S[7:4]);
    MUX_2_1_if muxc(c_0,c_1,c,cout);
```

endmodule

oraz 16-bitowy sumator:

```
module CSA_16(
    input [15:0] A,B,
    input cin,
    output [15:0] S,
    output cout);

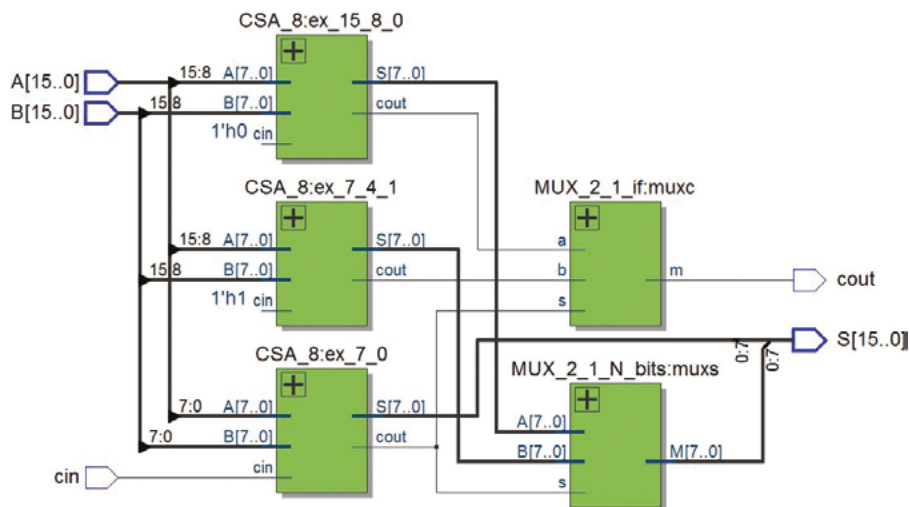
    wire [7:0] S_0,S_1;
    wire c,c_0,c_1;

    CSA_8 ex_7_0(A[7:0],B[7:0],cin,S[7:0],c);
    CSA_8 ex_15_8_0(A[15:8],B[15:8],1'b0,S_0,c_0);
    CSA_8 ex_7_4_1(A[15:8],B[15:8],1'b1,S_1,c_1);

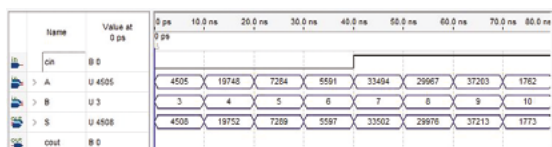
    MUX_2_1_N_bits #(8) muxs(S_0,S_1,c,S[15:8]);
    MUX_2_1_if muxc(c_0,c_1,c,cout);
```

endmodule

Wyniki syntezy i symulacji projektu CSA_16 możemy zobaczyć na rysunkach 8.32 i 8.33.



RYS. 8.32. Wynik syntezy 16-bitowego sumatora z sumami warunkowymi CSA



RYS. 8.33. Wyniki symulacji 16-bitowego sumatora z sumami warunkowymi CSA

Należy zauważyć, że wszystkie przedstawione struktury sumatorów są w różnym stopniu wspierane przez zasoby sprzętowe współczesnych układów FPGA. Zachęcamy czytelnika do zapoznania się z architekturą i zasobami docelowego układu FPGA, który zostanie użyty do implementacji sumatora, i wyznaczenia struktury sumatora, która będzie najbardziej efektywna dla danego układu FPGA.

8.1.7. Badania efektywności realizacji sumatorów na układach FPGA

Wszystkie opisane wcześniej projekty 16-bitowych sumatorów zostały zaimplementowane przy użyciu systemu Quartus Prime na układach FPGA różnych rodzin, aby ocenić ich efektywność pod kątem szybkości działania i kosztów realizacji. Metody projektowania sumatorów zostały porównane również z domyślną metodą realizacji dostępną w systemie Quartus.

Aby przeprowadzić badania nad metodą realizacji sumatorów w systemie Quartus, opiszemy sumator za pomocą standardowej operacji arytmetycznej dodawania:

```

module adder
    #(parameter N=16)
    (input [N-1:0] A,B,
    input cin,
    output [N-1:0] S,
    output cout);

    assign {cout,S} = A + B + cin;

endmodule

```

Każdy system zautomatyzowanego projektowania, w tym także Quartus, ma swoje własne metody realizacji operacji dodawania, które zazwyczaj wykorzystują specyficzne cechy architektury konkretnych rodzin układów FPGA.

Złożoność realizacji sumatorów została oceniona przez liczbę L elementów logicznych, jakie zostały wykorzystane w układzie FPGA. Aby obliczyć szybkość działania, do wejść i wyjść sumatorów zostały podłączone rejestry FPGA. Wówczas szybkość została określona jako maksymalna częstotliwość F przełączania rejestrów. Wyniki badań zestawione zostały w tabeli 8.2, gdzie:

- RCA – sumator z przeniesieniami szeregowymi;
- CLA(R) – sumator z przeniesieniami równoległymi oraz przeniesieniami szeregowymi pomiędzy blokami;
- CLA(L) – sumator z przeniesieniami równoległymi oraz przeniesieniami równoległymi pomiędzy blokami;
- CSEA(F) – sumator z wyborem przeniesienia z bloków jednakowego rozmiaru;
- CSEA(V) – sumator z wyborem przeniesienia z bloków zmiennego rozmiaru;
- CSKA – sumator z przeskokiem przeniesień;
- CSA – sumator z sumami warunkowymi;
- Quartus – realizacja sumatora za pomocą operacji arytmetycznej dodawania (metoda syntezy systemu Quartus).

Zgodnie z oczekiwaniami najszybszy okazał się sumator z przeniesieniami równoległymi oraz równoległym przekazywaniem przeniesień pomiędzy blokami CLA(L). Był on najszybszy dla wszystkich rodzin układów FPGA, z wyjątkiem rodziny Arria II, gdzie szybszy okazał się sumator CLA(R). Kolejnym pod względem szybkości działania jest sumator z przeniesieniami równoległymi, ale z szeregowym przekazywaniem przeniesień pomiędzy blokami CLA(R). Jednak oba te sumatory zajmują ostatnie miejsca, jeśli chodzi o złożoność realizacji.

Trzecie miejsce pod względem szybkości działania (a także złożoności realizacji) zajmuje sumator z wyborem przeniesienia zbudowany z bloków zmiennego rozmiaru CSEA(V). Następne w kolejności malejącej szybkości działania są sumatory CSA, CSEA(F), CSKA oraz RCA.

TABELA 8.2. Wyniki badań projektów 16-bitowych sumatorów

Sumator	MAX II		MAX V		MAX 10		Arria II		Cyclone IV		Cyclone V	
	L	F	L	F	L	F	L	F	L	F	L	F
RCA	32	124	32	80	42	145	37	193	41	144	25	146
CLA(R)	65	250	65	140	66	335	61	1075	65	397	35	454
CLA(L)	73	293	72	148	73	396	74	1030	72	576	38	510
CSEA(F)	47	157	47	93	48	157	36	590	47	207	25	294
CSEA(V)	50	196	50	115	52	184	45	665	51	246	25	296
CSKA	47	132	47	72	56	134	59	502	55	175	37	222
CSA	52	145	52	94	60	198	63	640	59	257	28	342
Best	32	293	32	148	42	396	37	1075	41	576	25	576
Quartus	19	331	19	191	19	409	17	767	18	494	10	403

Najlepszy według kosztów realizacji okazał się sumator z przeniesieniami szeregowymi RCA, natomiast dalej, w kolejności zwiększenia kosztów realizacji, były sumatory CSEA(V), CSKA, CSA, CLA(R) i CLA(L).

Porównując przedstawione metody realizacji sumatorów z metodą syntezy używaną w systemie Quartus, zauważalne jest, że dla wszystkich rodzin układów FPGA metoda systemu Quartus zapewnia lepsze wyniki pod względem kosztów realizacji (1,68–4,35-krotnie niższe koszty). Jeśli chodzi o szybkość działania, metoda Quartus osiąga lepsze wyniki dla wszystkich rodzin MAX, ale ustępuje sumatorowi z przeniesieniami równoległymi CLA w przypadku rodzin Arria II, Cyclone IV i Cyclone V. Przy czym sumator z przeniesieniami równoległymi CLA osiąga lepszą wydajność o 1,40 razy dla rodziny Arria II, 1,17 razy dla rodziny Cyclone IV oraz 1,27 razy dla rodziny Cyclone V.

Przewagę metody syntezy sumatorów systemu Quartus pod względem kosztów realizacji można wytłumaczyć wykorzystaniem specyficznych cech architektury układów FPGA, które są niedostępne dla użytkownika końcowego, na przykład arytmetycznego trybu pracy elementów logicznych FPGA. Do realizacji przeniesień w sumatorach są wykorzystywane szybkie *ścieżki przeniesień* (*Carry Chain*) między elementami logicznymi FPGA. Niemniej jednak wszystkie przeniesienia są przekazywane szeregowo. Dlatego też dla niektórych rodzin układów FPGA sumator CLA osiąga lepsze wyniki pod względem wydajności. Można przypuszczać, że wraz ze zwiększeniem rozmiaru sumatora przewaga sumatora CLA nad metodą systemu Quartus będzie narastać.

Proponujemy, aby czytelnik zaprojektował sumatory z przeniesieniami równoległymi CLA o różnych rozmiarach – 32, 64, 128 i 256 bitów, a następnie porównał szybkość ich działania z metodą projektowania systemu Quartus.

Wnioski. Jeśli celem jest zaprojektowanie sumatora o najniższych kosztach realizacji oraz wystarczająco dobrej szybkości, zaleca się skorzystanie z metody projektowania systemu Quartus.

Jeśli jednak potrzebny jest sumator bardzo szybki, warto rozważyć sumatory z przeniesieniami równoległymi CLA(L) lub CLA(R).

8.2. Sumator z przechowywaniem przeniesień

Sumator z *przechowywaniem przeniesień* (Carry-Save Adder) stosowany jest do obliczenia sumy trzech lub więcej N -bitowych liczb dwójkowych. W wyniku dodawania kilku liczb dwójkowych otrzymuje się dwie N -bitowe liczby: jedna to suma bitów, a druga to sekwencja bitów przeniesienia.

Cechą charakterystyczną sumatora z przechowywaniem przeniesień jest fakt, że przeniesienia podczas dodawania przekazywane są tylko między sąsiednimi bitami. Dzięki temu dodawanie kilku liczb jest wykonywane bardzo szybko.

Podczas końcowej fazy obliczeń dwie otrzymane sekwencje bitów są zamieniane na wynik operacji poprzez zezwolenie na przekazywanie zapamiętanych przeniesień przez wszystkie bity liczby. W ten sposób przekazywanie przeniesień wykonuje się tylko raz dla wszystkich dodawanych liczb. Przewaga sumatora z przechowywaniem przeniesień staje się zauważalna przy dodawaniu trzech lub więcej liczb.

Sumator z przechowywaniem przeniesień nie jest typowym sumatorem, lecz stosowany jest tam, gdzie konieczne jest jednoczesne dodawanie wielu liczb, na przykład w systemach kryptograficznych z kluczem publicznym.

8.3. Subtraktory

W technice cyfrowej układ wykonujący operację odejmowania nazywany jest *subtraktorem* (subtractor). Istnieją dwa podstawowe sposoby realizacji subtraktorów: w zwykłym kodzie dwójkowym oraz w kodzie dopełnienia do dwóch. W tym pierwszym budowany jest jednobitowy *pełny subtraktor* (Full Subtractor – FS) na podstawie tablicy prawdy przedstawionej w tabeli 8.3, gdzie a – bit odjemnej; b – bit odjemnika; d – bit wyniku; bin – wejście zapożyczenia; $bout$ – wyjście zapożyczenia.

Na podstawie jednobitowego pełnego subtraktora, analogicznie jak w przypadku sumatorów, można budować różne struktury wielobitowych subtraktorów, na przykład, z przeniesieniem szeregowym.

W systemach wbudowanych subtraktory najczęściej są realizowane w kodach uzupełnień do dwóch, bazując na sumatorach. Wykorzystywana jest przy tym następująca zależność:

$$A - B = A + (-B) = A + \bar{B} + 1, \quad (8.10)$$

gdzie $\bar{B}$ – inwersja bitowa liczby dwójkowej B .

TABELA 8.3. Tablica prawdy pełnego subtraktora

<i>a</i>	<i>b</i>	<i>bin</i>	<i>d</i>	<i>bout</i>
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Aby wykonać odejmowanie liczb *A* i *B* reprezentowanych w kodzie uzupełnienia do dwóch przy użyciu sumatora, należy podać liczbę *A* na wejście pierwszego operandu sumatora, inwersję bitową liczby dwójkowej *B* na wejście drugiego operandu, a jedynekę na wejście przeniesienia *cin*.

Poniżej przedstawiono kod *sumatora-subtraktora z przeniesieniami szeregowymi* (*Ripple-Carry Adder/Subtractor – RCAS*), który wykorzystuje zależność (8.10). Zamiast wejścia *cin* ma on wejście *sub*. Gdy *sub* = 0, moduł *ripple_carry_adder_subtractor* wykonuje operację dodawania $A + B$, a gdy *sub* = 1, operację odejmowania $A - B$.

```

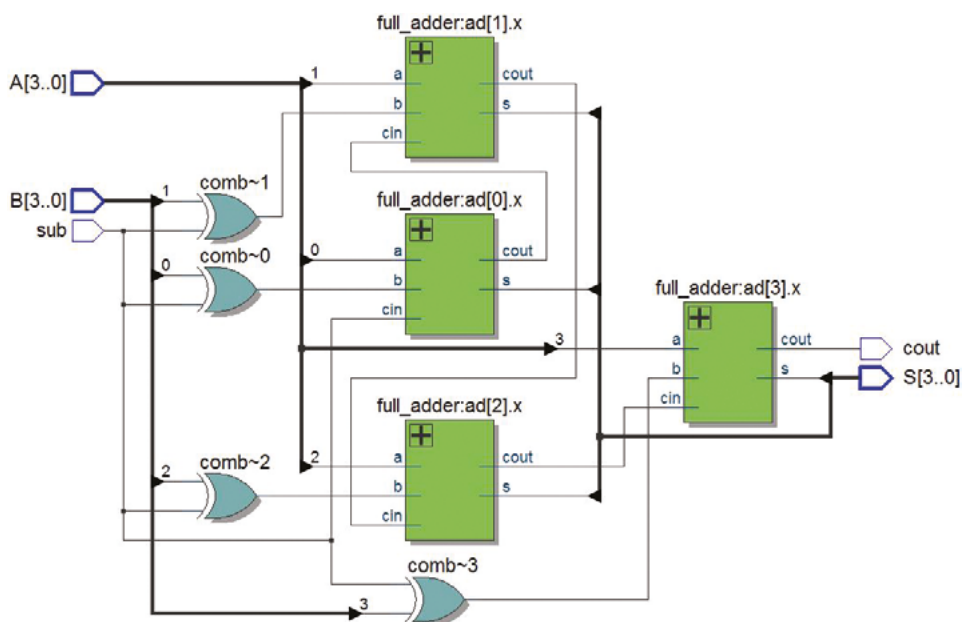
module ripple_carry_adder_subtractor
    #(parameter N=4)
    (input [N-1:0] A,B,
    input sub,
    output [N-1:0] S,
    output cout);

    wire [N-1:0] c;

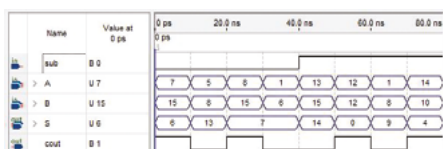
    generate
    genvar i;
    for (i=0; i<N; i=i+1)
    begin: ad
    case(i)
        0:          full_adder x(A[i],sub^B[i],sub,S[i],c[i]);
        N-1:        full_adder x(A[i],sub^B[i],c[i-1],S[i],cout);
        default:    full_adder x(A[i],sub^B[i],c[i-1],S[i],c[i]);
    endcase
    end
    endgenerate
endmodule

```


Wyniki syntezy i symulacji projektu *ripple_carry_adder_subtractor* przedstawione zostały na rysunkach 8.34 i 8.35.



RYS. 8.34. Wynik syntezy sumatora-subtraktora dla $N=4$

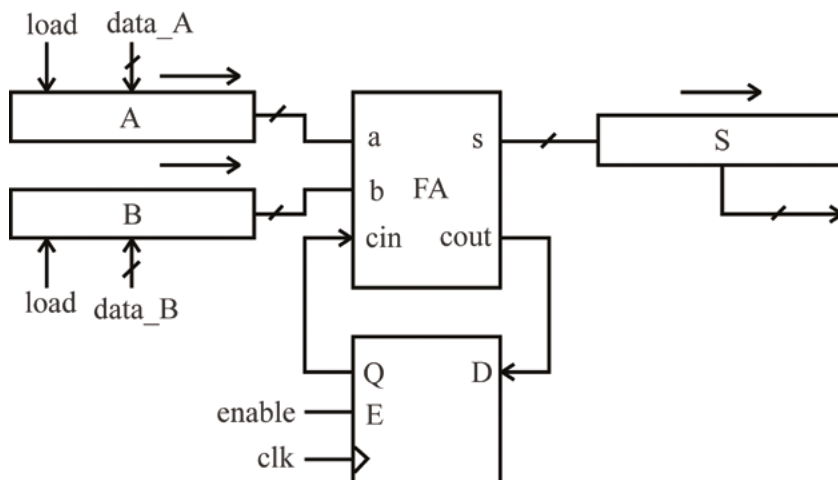


RYS. 8.35. Wyniki symulacji sumatora-subtraktora dla $N=4$

Zauważmy, że przy operacji odejmowania ($sub = 1$), jeśli występuje zapożyczenie do starszego bitu, wyjście *cout* wynosi 0, natomiast jeśli brak jest zapożyczenia, wyjście *cout* wynosi 1.

8.4. Sumator szeregowy na podstawie pełnego sumatora i rejestrów przesuwnych

Wielobitowy sumator można zbudować, wykorzystując tylko jeden jednobitowy pełny sumator jako podstawowy element. Taki schemat oprócz pełnego sumatora będzie zawierał dodatkowo trzy rejestry przesuwne dla liczb A, B oraz sumy S, a także przerzutnik w obwodzie przeniesień, co przedstawiono na rysunku 8.36.



RYS. 8.36. Struktura sumatora szeregowego na podstawie jednego pełnego sumatora i rejestrów przesuwanych

Schemat przedstawiony na rysunku 8.36 nie należy do klasy układów kombinacyjnych, tak jak większość sumatorów, lecz jest reprezentantem układów sekwencyjnych, które wymagają sygnału synchronizacji. Dodatkowo w tym schemacie występują sygnały sterowania rejestrami przesuwanymi oraz szyny wejściowe *data_A* i *data_B*, służące do załadowania wartości liczb *A* i *B*. Aby ten schemat działał poprawnie, konieczne jest także zrealizowanie specjalnego bloku sterowania, który generuje sekwencje sygnałów sterujących odpowiedzialnych za załadowanie liczb *A* i *B* oraz wykonanie operacji dodawania. Szeregowo synchroniczne sumatory oparte na pojedynczym pełnym sumatorze znajdują zastosowanie między innymi w systemach szeregowej transmisji danych czy przy obliczaniu sum kontrolnych.

Poniżej zaprezentowano kod Verilog projektu szeregowego synchronicznego sumatora, który odpowiada strukturze przedstawionej na rysunku 8.36.

```
module sequential_adder
  #(parameter N=8)
  (input [N-1:0] data_A,data_B,
  input clk,aclr,enable,load,
  output [N-1:0] S);

  wire a,b,cin,s_1bit,cout;

  right_shift_reg #(N) sr_A(data_A,1'b0,clk,aclr,enable,load,,a);
  right_shift_reg #(N) sr_B(data_B,1'b0,clk,aclr,enable,load,,b);

  full_adder ad(a,b,cin,s_1bit,cout);
```

```

    register_N_bits_areset_enable #(1) r(cout,clk,aclr,enable,cin);

    right_shift_reg #(N) sr_S(s_1bit,clk,aclr,enable,1'b0,S,);
endmodule

```

Poniższy kod opisuje rejestr przesuwny w prawo z asynchronicznym resetowaniem, sygnałem zezwolenia przesunięcia oraz synchronicznym ładowaniem danych:

```

module right_shift_reg
    #(parameter N=4)
    (input [N-1:0] data_in,
    input MSB_in,
    input clk,aclr,enable,load,
    output reg [N-1:0] Data_out,
    output LSB_out);

    assign LSB_out=Data_out[0];

    always @(posedge clk, negedge aclr)
        if(~aclr)           Data_out <= 0;
        else if(load)       Data_out <= data_in;
        else if(enable)     Data_out <= {MSB_in,Data_out[N-1:1]};
        else               Data_out <= Data_out;

endmodule

```

Realizację rejestru z asynchronicznym zerowaniem oraz sygnałem zezwolenia pracy ilustruje poniższy fragment kodu:

```

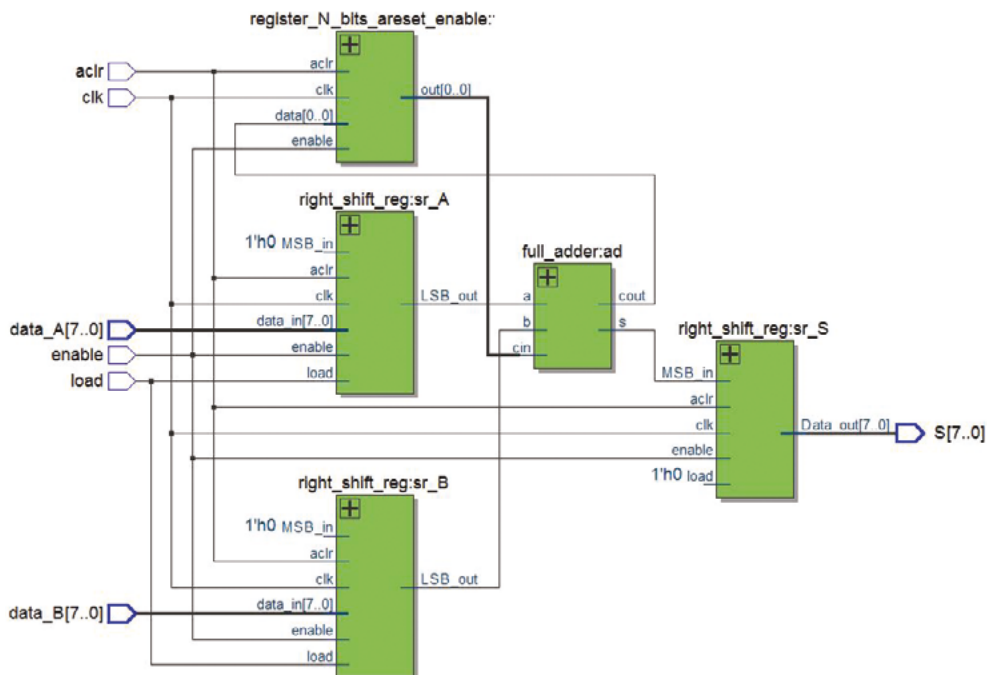
module register_N_bits_areset_enable
    #(parameter N=8)
    (input [N-1:0] data,
    input clk, aclr, enable,
    output reg [N-1:0] out);

    always @(posedge clk, negedge aclr)
        if (~aclr)           out <= {N{1'b0}};
        else if(enable)       out <= data;
        else                 out <= out;

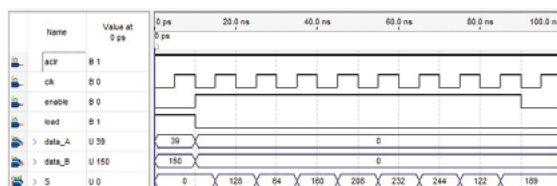
endmodule

```

Wyniki syntezy i symulacji projektu *sequential_adder* pokazane są na rysunkach 8.37 i 8.38.



RYS. 8.37. Wynik syntezy szeregowego sumatora na podstawie jednego pełnego sumatora



RYS. 8.38. Wyniki symulacji szeregowego sumatora na podstawie jednego pełnego sumatora

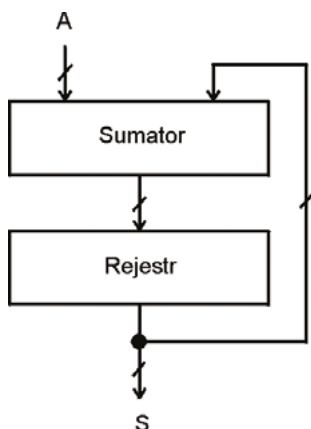
8.5. Sumator akumulujący

Sumator akumulujący, znany także jako *akumulator* (*accumulator*), jest układem sekwencyjnym, który dokonuje sukcesywnego dodawania kolejnych liczb, a wynik jest gromadzony w rejestrze wyjściowym. Funkcjonowanie akumulatora może być opisane za pomocą wyrażenia:

$$S := S + A,$$

gdzie: A – dodawana liczba; S – przechowywana wartość.

Struktura akumulatora przedstawiona została na rysunku 8.39.



RYS. 8.39. Struktura akumulatora

Akumulatory znajdują szerokie zastosowanie w układach arytmetyczno-logicznych (ALU). W sumatorze akumulującym jeden z operandów (oznaczony jako *S*) zawsze przyjmuje przechowywaną wartość sumy, podczas gdy drugi operand (oznaczony jako *A*) jest kolejną dodawaną liczbą.

Opis sumatora akumulującego w języku Verilog może być przedstawiony w następujący sposób:

```
module accumulator
    #(parameter N=8)
    (input [N-1:0] A,
     input clk,aclr,
     output [N-1:0] S);

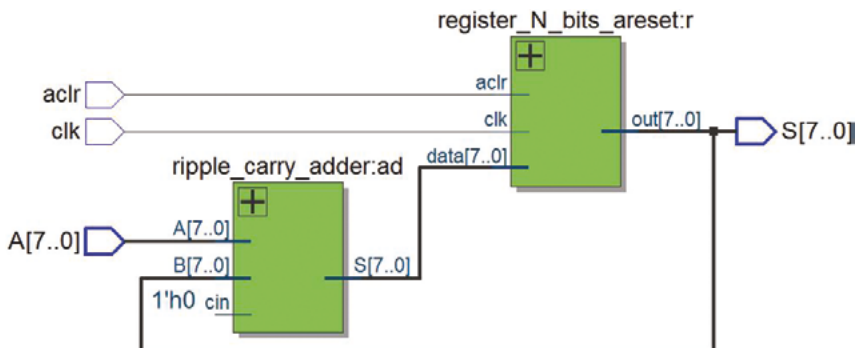
    wire [N-1:0] A_plus_B;

    ripple_carry_adder #(N) ad(A,S,1'b0,A_plus_B);

    register_N_bits_areset #(N) r(A_plus_B,clk,aclr,S);

endmodule
```

Wyniki syntezy i symulacji projektu *sequential_adder* ukazane zostały na rysunkach 8.40 i 8.41.



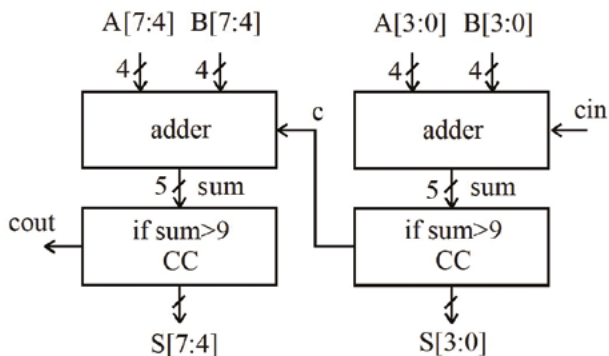
RYS. 8.40. Wynik syntezy sumatora akumulującego



RYS. 8.41. Wyniki symulacji sumatora akumulującego

8.6. Sumator liczb dwójkowo-dziesiętnych BCD

Sumatory liczb w kodzie *dwójkowo-dziesiętnym BCD* (*Binary Coded Decimal*) wykorzystywane są w kalkulatorach, co pozwala zaoszczędzić czas na konwersji liczb z kodu BCD na postać dziesiętną i odwrotnie. Struktura sumatora, który wykonuje operację dodawania dwóch cyfr BCD, została zaprezentowana na rysunku 8.42.



RYS. 8.42. Struktura sumatora dwóch cyfr BCD

W powyższym schemacie każda cyfra w kodzie BCD jest zapisana na czterech bitach. Wynik dodawania sum ma 5 bitów, uwzględniając przeniesienie, które może się pojawić przy dodawaniu cyfr. Ten wynik jest przekazywany do *schematu korekcji dziesiętnej* (*correct circuit – CC*). Sprawdza on wynik dodawania, a jeśli przekroczy on wartość 9, generuje sygnał przeniesienia c i dodaje korektę do wyniku sum . Wynik dodawania S jest określany w następujący sposób:

$$\begin{aligned} S &= sum - 10_{10} = sum + (-10_{10}) = sum + (\overline{10}_{10} + 1) = sum + (\overline{1010}_2 + 1) = \\ &= sum + (0101_2 + 1) = sum + 0110_2 = sum + 6_{10}. \end{aligned}$$

Poniżej zaprezentowany został kod sumatora dwóch cyfr BCD:

```
module BCD_adder_2_digits(
    input [7:0] A,B,
    input cin,
    output [7:0] S,
    output cout);

    wire c;
    wire [4:0] sum0,sum1;

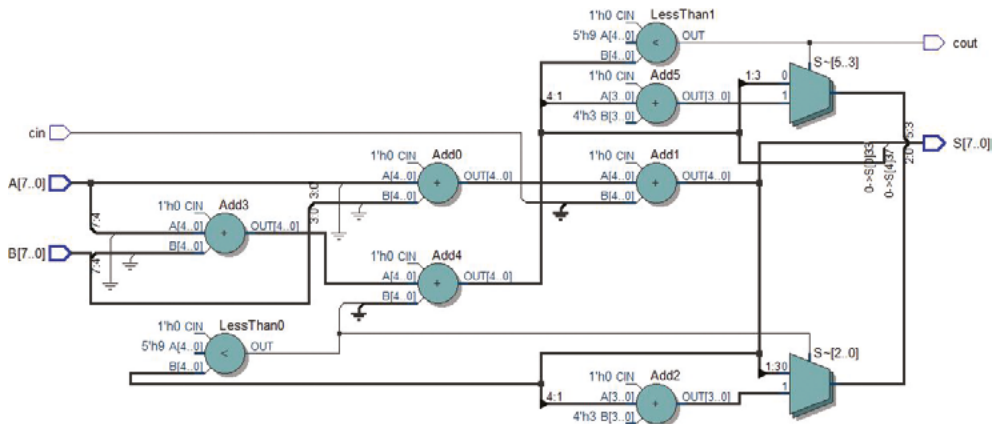
    assign sum0 = A[3:0] + B[3:0] + cin;
    assign S[3:0] = (sum0 > 9) ? sum0 + 4'b0110 : sum0;
    assign c = (sum0 > 9) ? 1'b1 : 1'b0;

    assign sum1 = A[7:4] + B[7:4] + c;
    assign S[7:4] = (sum1 > 9) ? sum1 + 4'b0110 : sum1;
    assign cout = (sum1 > 9) ? 1'b1 : 1'b0;

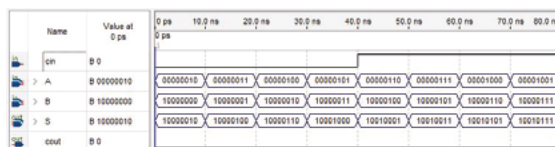
endmodule
```

Łącząc szeregowo moduły *BCD_adder_2_digits*, można rozszerzyć sumator liczb w kodzie BCD do dowolnego rozmiaru.

Wyniki syntezy i symulacji projektu *BCD_adder_2_digits* zostały pokazane na rysunkach 8.43 i 8.44.



RYS. 8.43. Wynik syntezy sumatora dwóch cyfr BCD



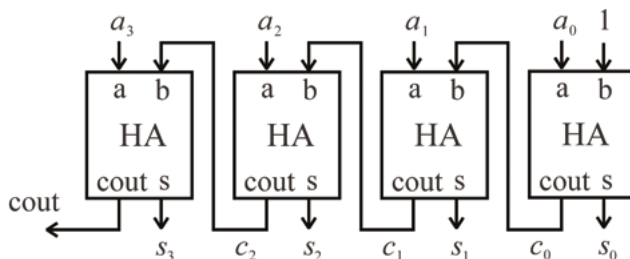
RYS. 8.44. Wyniki symulacji sumatora dwóch cyfr BCD

8.7. Inkrementatory i dekrementatory

8.7.1. Inkrementator

Inkrementator (incrementer) jest układem cyfrowym, który wykonuje operację zwiększenia wartości liczby o jeden: $S = A + 1$. Inkrementatory często wykorzystywane są w systemach wbudowanych do automatycznego zwiększenia adresu przy sekwencyjnym odczytywaniu zawartości pamięci lub w procesorach do obliczenia adresu następnej wykonywanej instrukcji.

Inkrementator może być zrealizowany na podstawie zwykłego sumatora. W tym przypadku wystarczy użyć stałej wartości 1 jako drugiego operandu; można także przyjąć, że drugi operand jest zerem, a na wejście *cin* podawana będzie jedynka. Aby zmniejszyć złożoność schematu, inkrementatory często są budowane na półsumatorach (rysunek. 8.45).



RYS. 8.45. Struktura 4-bitowego inkrementatora

W powyższym schemacie wyjście *cout* każdego półsumatora połączone jest z wejściem *b* następnego półsumatora. Jeśli na wejście *b* najmłodszego półsumatora podamy jedynkę, schemat wykonuje operację $S = A + 1$; w przypadku podania zera na to wejście na wyjściu otrzymamy liczbę A . Dlatego wejście *b* najmłodszego sumatora pełni funkcję sygnału zezwolenia zliczania (*enable*).

8.7.1.1. Inkrementator z przeniesieniem szeregowym

Inkrementatory wielobitowe, analogicznie do sumatorów, mogą być realizowane z różnymi schematami przeniesienia: szeregowym, równoległym i innymi. Poniżej zaprezentowano kod N -bitowego inkrementatora z przeniesieniem szeregowym.

```

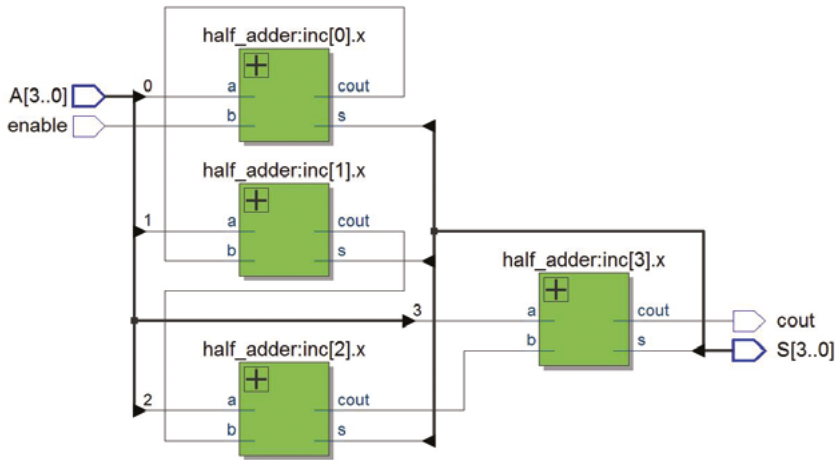
module ripple_carry_incrementer
    #(parameter N=4)
    (input [N-1:0] A,
    input enable,
    output [N-1:0] S,
    output cout);

    wire [N-1:0] c;

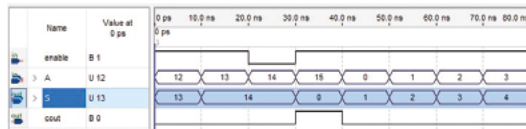
    generate
    genvar i;
    for (i=0; i<N; i=i+1)
    begin: inc
    case(i)
        0:          half_adder x(A[i],enable,S[i],c[i]);
        N-1:        half_adder x(A[i],c[i-1],S[i],cout);
        default:    half_adder x(A[i],c[i-1],S[i],c[i]);
    endcase
    end
    endgenerate
endmodule

```

Wyniki syntezy i symulacji projektu *ripple_carry_incrementer* widoczne są na rysunkach 8.46 i 8.47.



RYS. 8.46. Wynik syntezy inkrementatora z przeniesieniem szeregowym dla $N=4$



RYS. 8.47. Wyniki syntezy inkrementatora z przeniesieniem szeregowym dla $N=4$

8.7.1.2. Inkrementator z przeniesieniem równoległym

Aby zbudować inkrementator z przeniesieniem równoległym, przyjmujemy, że $B = 0$, czyli $b_i = 0$ dla wszystkich bitów $i = 0, N-1$. Po podstawieniu tych wartości do wyrażeń (8.4)–(8.8) możemy zauważyć, że podlegają one znacznemu uproszczeniu. W wyniku przekształcenia wyrażenia opisujące zredukowany pełny sumator (*reduced full adder – RFA*) przyjmują następującą postać:

$$\begin{aligned} p_i &= a_i; \\ s_i &= a_i \oplus c_{i-1}. \end{aligned} \quad (8.11)$$

Generator przyspieszonego przeniesienia dla 4-bitowego inkrementatora realizowany jest na podstawie następujących wyrażeń:

$$\begin{aligned} c_0 &= p_0 \cdot cin; \\ c_1 &= p_1 \cdot p_0 \cdot cin; \\ c_2 &= p_2 \cdot p_1 \cdot p_0 \cdot cin; \\ c_3 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot cin; \\ pg &= p_3 \cdot p_2 \cdot p_1 \cdot p_0. \end{aligned} \quad (8.12)$$

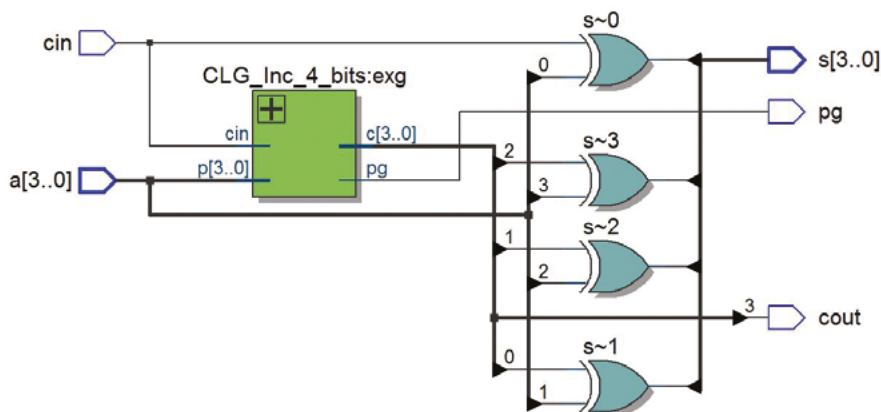
Poniżej przedstawiony został kod generatora przyspieszonego przeniesienia dla 4-bitowego inkrementatora:

```
module CLG_Inc_4_bits(  
    input [3:0] p,  
    input cin,  
    output [3:0] c,  
    output pg);  
  
    assign c[0]=p[0]&cin;  
    assign c[1]=p[1]&p[0]&cin;  
    assign c[2]=p[2]&p[1]&p[0]&cin;  
    assign c[3]=p[3]&p[2]&p[1]&p[0]&cin;  
  
    assign pg=p[3]&p[2]&p[1]&p[0];  
  
endmodule
```

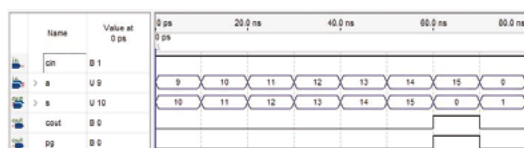
Kolejny przykład prezentuje kod 4-bitowego inkrementatora z przeniesieniem równoległym:

```
module CLI_4_bits(  
    input [3:0] a,  
    input cin,  
    output [3:0] s,  
    output cout, pg);  
  
    wire [3:0] c;  
  
    CLG_Inc_4_bits exg(a,cin,c,pg);  
  
    assign s[0]=a[0]^cin;  
    assign s[1]=a[1]^c[0];  
    assign s[2]=a[2]^c[1];  
    assign s[3]=a[3]^c[2];  
  
    assign cout=c[3];  
  
endmodule
```

Wyniki syntezy i symulacji projektu *CLI_4_bits* możemy zaobserwować na rysunkach 8.48 i 8.49.



RYS. 8.48. Wynik syntezy inkrementatora z przeniesieniem równoległym dla $N=4$



RYS. 8.49. Wyniki symulacji inkrementatora z przeniesieniem równoległym dla $N=4$

Inkrementator z przeniesieniem równoległym o rozmiarze 16 bitów, analogicznie jak w przypadku sumatora, można zbudować z czterech 4-bitowych inkrementatorów oraz jednego 4-bitowego generatora przyspieszonego przeniesienia:

```

module CLI_16_bits(
    input [15:0] a,
    input cin,
    output [15:0] s,
    output cout, pg);

    wire [3:0] p,c;

    CLI_4_bits ex0(a[3:0],cin,s[3:0],p[0]);
    CLI_4_bits ex1(a[7:4],c[0],s[7:4],p[1]);
    CLI_4_bits ex2(a[11:8],c[1],s[11:8],p[2]);
    CLI_4_bits ex3(a[15:12],c[2],s[15:12],p[3]);

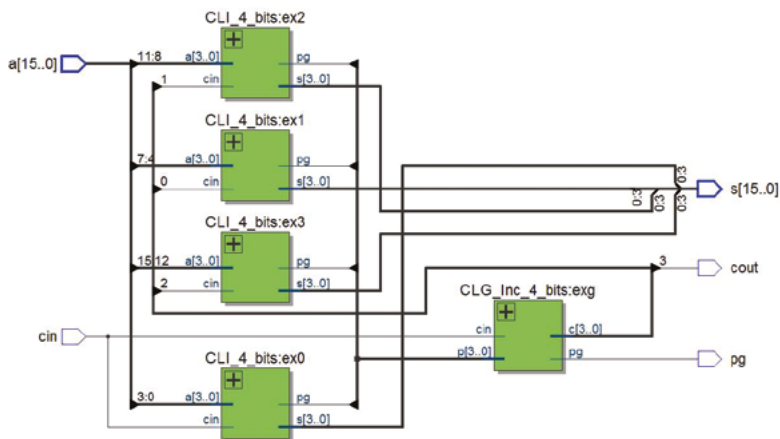
    CLG_Inc_4_bits exg(p,cin,c,pg);

    assign cout=c[3];

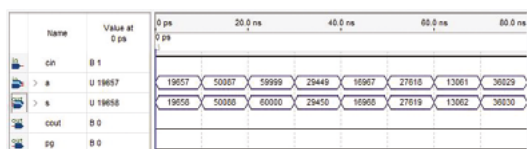
endmodule

```

Wyniki syntezy i symulacji projektu *CLI_16_bits* przedstawione zostały na rysunkach 8.50 i 8.51.



RYS. 8.50. Wynik syntezy 16-bitowego inkrementatora z przeniesieniem równoległym



RYS. 8.51. Wyniki symulacji 16-bitowego inkrementatora z przeniesieniem równoległym

8.7.2. Dekrementator

Dekrementatorem (decrementer) nazywany jest układ cyfrowy, który zmniejsza wartość operandu o jeden, czyli wykonuje operację: $S = A - 1$. Ponieważ wartość -1 w kodzie uzupełnień do dwóch reprezentowana jest jedynekami na wszystkich bitach, wyrażenia opisujące funkcje sumy i przeniesienia dla dekrementatora można otrzymać z równań (8.2), podstawiając jedynkę zamiast operandu b :

$$s = \overline{a \oplus \text{cin}}; \quad (8.13)$$

$$\text{cout} = a + \text{cin}.$$

Sygnal przeniesienia dekrementatora nazywany jest *zapożyczeniem*. Schemat realizujący równania logiczne (8.13) zostanie oznaczony jako *half_sub*. Poniżej pokazany jest kod schematu *half_sub*, który implementuje jeden bit dekrementatora:

```
module half_sub(
    input a,cin,
    output s,cout);
```

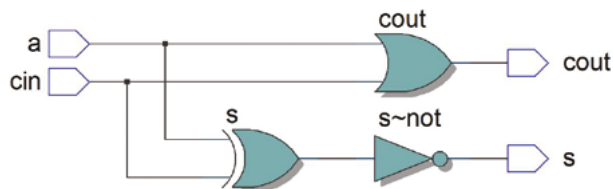
```

assign s = a ^ cin;
assign cout = a | cin;

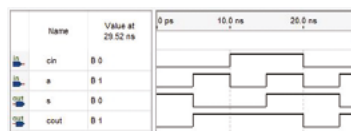
```

endmodule

Wyniki syntezy i symulacji projektu *half_sub* zobrazowane zostały na rysunkach 8.52 i 8.53.



RYS. 8.52. Wynik syntezy modułu *half_sub*



RYS. 8.53. Wyniki symulacji modułu *half_sub*

Dekrementatory wielobitowe mogą być konstruowane przy użyciu schematu przeniesienia szeregowego lub równoległego. Kod *N*-bitowego dekrementatora z przeniesieniem szeregowym może zostać przedstawiony w następujący sposób:

```

module ripple_carry_decrementer
    #(parameter N=4)
    (input [N-1:0] A,
    input enable,
    output [N-1:0] S,
    output cout);

    wire [N-1:0] c;

    generate
    genvar i;
    for (i=0; i<N; i=i+1)
    begin: dec
        case(i)
        0: half_sub x(A[i],~enable,S[i],c[i]);
        default: half_sub x(A[i],c[i-1],S[i],c[i]);
    end

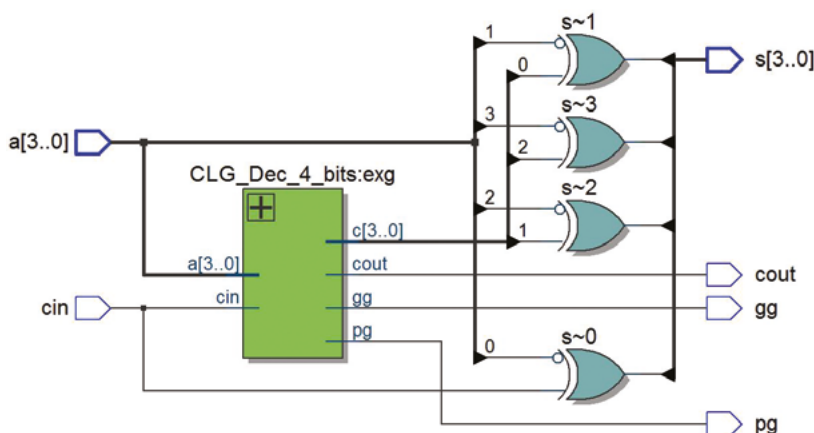
```

```

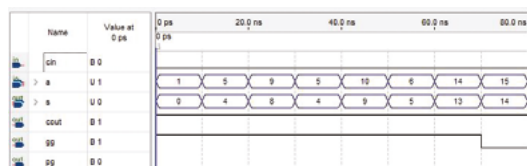
        endcase
    end
    endgenerate
    assign cout = ~c[N-1];
endmodule

```

Na rysunkach 8.54 i 8.55 możemy zaobserwować wyniki syntezy i symulacji projektu *ripple_carry_decrementer*.



RYS. 8.54. Wynik syntezy 4-bitowego dekrementatora z przeniesieniem szeregowym



RYS. 8.55. Wyniki symulacji 4-bitowego dekrementatora z przeniesieniem szeregowym

Specjalnie zaprojektowane inkrementatory i dekrementatory powinny charakteryzować się znacznie wyższą prędkością działania oraz niższym kosztem w porównaniu z podobnymi układami opartymi na sumatorze. Zachęcamy czytelnika do zrealizowania projektów 16-bitowych inkrementatorów i dekrementatorów z przeniesieniami szeregowym i równoległym, a następnie porównania szybkości działania oraz kosztów realizacji tych projektów z analogicznymi układami opartymi na sumatorze. W przypadku sumatora na wejście podawane są wartości 1 lub -1 jako drugi operand.

8.8. Realizacja licznika dodającego na podstawie inkrementatora

Na podstawie inkrementatora (lub dekrementatora) oraz rejestru można zbudować licznik dodający (albo odejmujący). W tym celu inkrementator (dekrementator) umieszczony jest w obwodzie pomiędzy wyjściem a wejściem rejestru. Poniżej znajduje się kod licznika dodającego, zbudowanego na bazie rejestru i inkrementatora.

```

module counter_inc
  #(parameter N=4)
  (input clk,aclr,enable,
   output [N-1:0] Q,
   output CR);

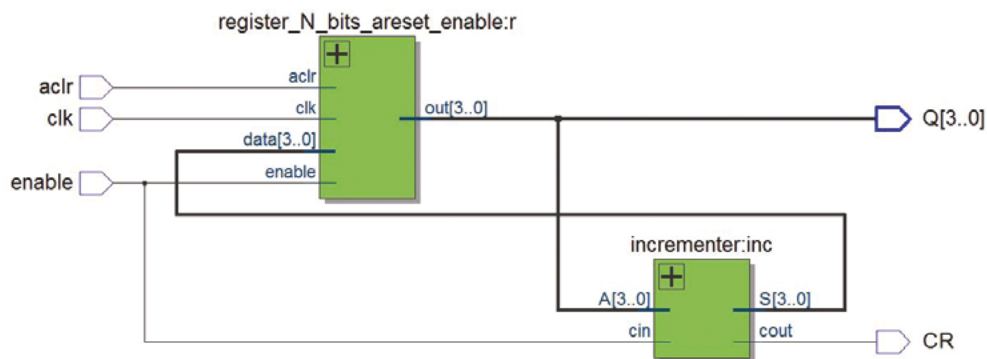
  wire [N-1:0] S;

  incrementer #(N) inc(Q,enable,S,CR);

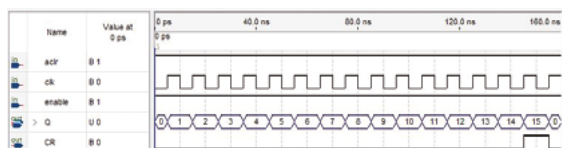
  register_N_bits_areset_enable #(N) r(S,clk,aclr,enable,Q);
endmodule

```

Wyniki syntezy i symulacji projektu *counter_inc* zaprezentowane zostały na rysunkach 8.56 i 8.57.



RYS. 8.56. Wynik syntezy licznika zbudowanego na podstawie rejestru i inkrementatora



RYS. 8.57. Wyniki symulacji licznika zbudowanego na podstawie rejestru i inkrementatora

8.9. Podsumowanie rozdziału

Sumator z przeniesieniami szeregowymi RCA jest najwolniejszy ze wszystkich sumatorów, jednakże jest najlepszy z punktu widzenia kosztów realizacji. Sumator z przyspieszonym przeniesieniem CLA jest najszybszy (przynajmniej do określonej liczby bitów), ale charakteryzuje się największym kosztem realizacji. Wszystkie pozostałe metody realizacji sumatorów (takie jak sumator z wyborem przeniesienia CSEA, sumator z przeskokiem przeniesień CSKA czy sumator z sumami warunkowymi CSA) zajmują pozycje pomiędzy tymi dwoma głównymi rozwiązaniami.

Jeśli celem jest zaprojektowanie sumatora o minimalnych kosztach realizacji oraz wystarczająco dobrej szybkości, zaleca się wybór metody projektowania zastosowanej w systemie Quartus. Gdy jednak potrzebujemy bardzo szybkiego sumatora, należy rozważyć rozwiązania oparte na sumatorach z przeniesieniami równoległymi CLA(L) lub CLA(R).

Sumator z *przechowywaniem przeniesień* (*Carry-Save Adder*) jest wykorzystywany do obliczania sumy trzech lub więcej N -bitowych liczb dwójkowych. Zazwyczaj znajduje on zastosowanie w systemach, gdzie równocześnie dodawanych jest wiele liczb, na przykład w systemach kryptograficznych opartych na kluczu publicznym.

Istnieją dwa podstawowe sposoby realizacji subtraktorów: w standardowym kodzie dwójkowym oraz w kodzie uzupełnień do dwóch. W praktyce najczęściej subtraktory są budowane na podstawie kodu uzupełnień do dwóch z wykorzystaniem sumatorów. W takim przypadku na wejściu pierwszego operandu sumatora podawana jest liczba A , na wejściu drugiego operandu – inwersja bitów liczby dwójkowej B , a na wejściu przeniesienia cin – jedynka.

Sumator szeregowy (synchroniczny) można skonstruować, wykorzystując pojedynczy pełny sumator oraz rejestry przesuwne. Takie sumatory są powszechnie używane w systemach szeregowej transmisji danych, podczas obliczania sum kontrolnych oraz w innych zastosowaniach.

Sumator *akumulujący* (*accumulator*) to układ sekwencyjny, który wykonuje kolejne instrukcje dodawania i przechowuje wynik w rejestrze wyjściowym. Akumulatory są powszechnie wykorzystywane w jednostkach arytmetyczno-logicznych (ALU).

Sumatory liczb w *dwójkowo-dziesiętnym kodzie BCD* (*Binary Coded Decimal*) znajdują zastosowanie w kalkulatorach, co przyspiesza konwersję liczb z kodu BCD na postać dziesiętną i odwrotnie.

Inkrementator (*incrementer*) umożliwia zwiększenie wartości operandu o jeden. Są one stosowane w systemach wbudowanych do automatycznego zwiększenia adresu przy sekwencyjnym odczycie zawartości pamięci lub w procesorach do obliczenia adresu następnej wykonywanej instrukcji. Aby uprościć realizację, inkrementatory często są budowane na półsumatorach. Wielobitowe inkrementatory, podobnie jak sumatory, mogą być zrealizowane przy użyciu różnych schematów przeniesienia: szeregowego, równoległego i innych.

Dekrementator (decrementer) służy do zmniejszania wartości operandu o jeden. Wielobitowe dekrementatory mogą być również zbudowane z wykorzystaniem schematu szeregowego lub równoległego przeniesienia.

Na podstawie inkrementatorów i dekrementatorów można zrealizować liczniki dodające lub odejmujące.

9. Projektowanie układów mnożących

Układy mnożące stanowią integralną część wielu systemów wbudowanych, szczególnie tych, które obsługują funkcje *cyfrowego przetwarzania sygnałów* (*Digital Signal Processing – DSP*). Istnieje znacząca różnorodność układów mnożących, które obsługują różne typy danych, takie jak liczby całkowite ze znakiem i bez znaku, liczby w kodzie uzupełnień do dwóch oraz liczby zmiennoprzecinkowe. Dodatkowo opracowano szereg algorytmów mnożenia i metod implementacji tych układów. Można wyodrębnić dwa główne typy układów mnożących: *szeregowy (serial)* oraz *równoległy (parallel)*.

Szeregowy układy mnożące zazwyczaj działają synchronicznie, generując cyklicznie iloczyny częściowe i dodając je do wcześniej obliczonego wyniku. W ich konstrukcji często wykorzystuje się rejestry przesuwne oraz sumator akumulujący lub zwykły. Ponadto układy te wymagają zastosowania automatu skończonego, który pełni funkcję sterowania w procesie mnożenia.

Równoległe układy mnożące są zazwyczaj implementowane jako układy kombinacyjne, które przyjmują liczby A i B na swoich wejściach, a na wyjściach generują wynik iloczynu P . W przeciwieństwie do układów szeregowych równoległe układy mnożące nie potrzebują rejestrów, sygnałów synchronizacji ani układów sterujących. Jednakże wraz ze wzrostem rozmiaru bitowego operandów złożoność tych układów znacznie wzrasta. Dodatkowo większe układy równoległe charakteryzują się dłuższymi opóźnieniami generowanych sygnałów, co prowadzi do obniżenia szybkości działania układu.

Aby zniwelować te problemy, równoległe układy mnożące często są projektowane w architekturze *potokowej (pipelined)*, co pozwala na przyspieszenie cyklu operacji mnożenia i zwiększenie wydajności całego systemu. W takim podejściu do schematu równoległego układu mnożącego dodaje się rejestry, które służą do przechowywania wyników pośrednich.

W tym rozdziale omówimy realizację na układach FPGA równoległych układów mnożących, które wykonują mnożenie liczb całkowitych bez znaku. Układy mnożące o większej złożoności są zazwyczaj implementowane na układach FPGA za pomocą specjalnych bloków DSP lub przy wykorzystaniu wbudowanych procesorów.

9.1. Matrycowy układ mnożący z przechowywaniem przeniesień

Układem mnożącym (multiplier) nazywamy układ, który wykonuje operację mnożenia:

$$P = A \cdot B = (a_{m-1} \cdot 2^{m-1} + \dots + a_0 \cdot 2^0) \cdot (a_{n-1} \cdot 2^{n-1} + \dots + b_0 \cdot 2^0). \quad (9.1)$$

W równaniu (9.1) liczba podlegająca mnożeniu (A) nazywana jest *mnożną* (*multiplicand*), liczba, przez którą wykonuje się mnożenie (B), nazywana jest *mnożnikiem* (*multiplier*), a wynik operacji mnożenia (liczba P) jest zwany *iloczynem* (*product*).

Najprostszym układem mnożącym jest *matrycowy układ mnożący z przechowywaniem przeniesień* (Carry-Save Array Multiplier – CSAM). Idea działania układu CSAM opiera się na klasycznym algorytmie mnożenia pisemnego (*paper and pencil*). Rysunek 9.1 ilustruje zastosowanie tego algorytmu do mnożenia dwóch liczb dziesiętnych: 11 i 12.

$$\begin{array}{r} \\ x \end{array} \begin{array}{rr} 1 & 1 \\ 1 & 2 \\ \hline 2 & 2 \\ 1 & 1 \\ \hline 1 & 3 & 2 \end{array}$$

RYS. 9.1. Mnożenie liczb w systemie dziesiętnym

Mnożenie tych samych liczb, ale w systemie dwójkowym, zostało zaprezentowane na rysunku 9.2.

$$\begin{array}{r} 1011 \\ \times 1100 \\ \hline 0000 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10000100 \end{array}$$

RYS. 9.2. Mnożenie liczb w dwójkowym systemie liczbowym

Niech zostaną przeprowadzone operacje mnożenia poszczególnych bitów liczb A i B, czyli wyznaczone zostaną *iloczyny częściowe* (*partial products*):

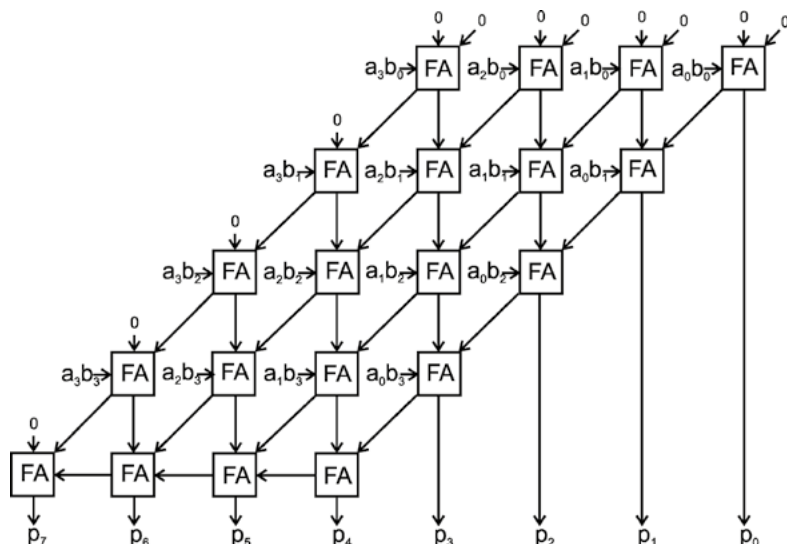
$$a_i \cdot b_j \quad \text{dla wszystkich } i = 0, \dots, N-1, j = 0, \dots, N-1. \quad (9.2)$$

Wówczas proces mnożenia przedstawiony na rysunku 9.2 można zobrazować w sposób pokazany na rysunku 9.3.

				x	a_3 b_3	a_2 b_2	a_1 b_1	a_0 b_0
					a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1		a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2		a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3		a_0b_3			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

RYS. 9.3. Mnożenie 4-bitowych liczb dwójkowych

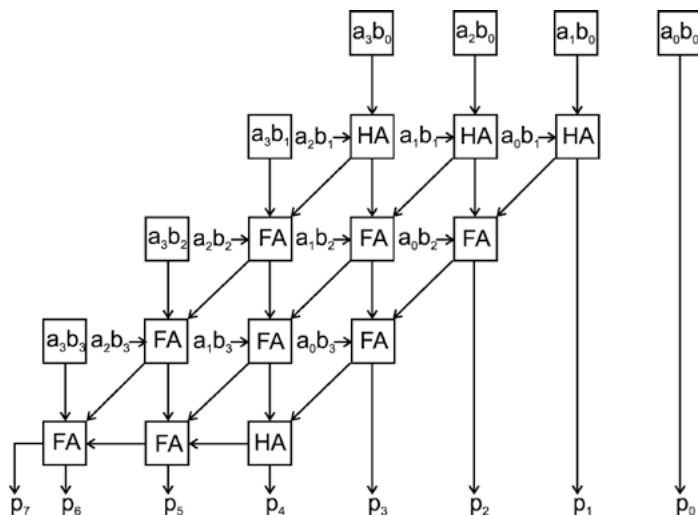
Struktura matrycowego układu mnożącego z przechowywaniem przeniesień wykonującego operację mnożenia 4-bitowych liczb dwójkowych (rysunek 9.3) została przedstawiona na rysunku 9.4.



RYS. 9.4. Struktura matrycowego układu mnożącego z przechowywaniem przeniesień dla 4-bitowych liczb dwójkowych

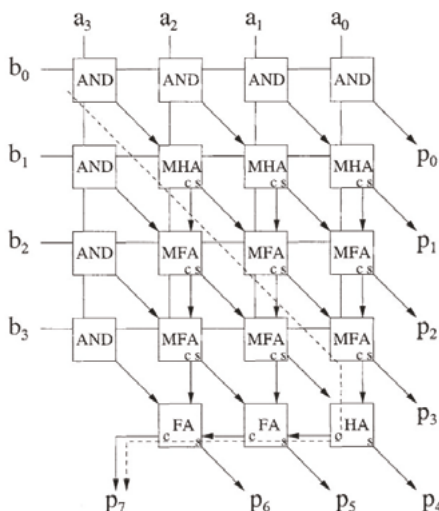
W każdym węźle (i, j) struktury matrycowej znajduje się pełny sumator, który dodaje do iloczynu częściowego $a_j \cdot b_i$ sumę i przeniesienie przekazywane z komórek umieszczonych powyżej. Najmłodsze cztery bity iloczynu P generowane są bezpośrednio na wyjściach odpowiednich sumatorów. Do uzyskania starszych czterech bitów iloczynu P mogą zostać wykorzystane sumatory z przeniesieniem szeregowym (Ripple-Carry Adder – RCA). Sumatory RCA na rysunku 9.4 umieszczone są w dolnym wierszu macierzy.

Schemat przedstawiony na rysunku 9.4 można znacznie uprościć. Ponieważ iloczyny częściowe zostały już obliczone zgodnie z wyrażeniem (9.2), można usunąć górny wiersz sumatorów ze schematu. Następnie każdy pełny sumator, do którego podawane są już tylko dwie wartości argumentów, można zastąpić półsumatorem. W wyniku tych przekształceń otrzymamy schemat przedstawiony na rysunku 9.5.



RYS. 9.5. Struktura matrycowego układu mnożącego 4×4 bity po uproszczeniu

Struktury matrycowych układów mnożących często są przedstawiane w postaci macierzy, gdzie dodawanie jest realizowane w kolumnach, jak pokazano na rysunku 9.6.



RYS. 9.6. Matrycowy układ mnożący 4×4 bitów z dodawaniem iloczynów częściowych w kolumnach

Na rysunku 9.6 wyraźnie widać strukturę matrycową układu mnożącego, gdzie poszczególnym kolumnom odpowiadają bity liczby A , wierszom zaś – bity liczby B . Dodatkowo obserwujemy, że dodawanie iloczynów częściowych jest wykonywane w kolumnach z przeniesieniem szeregowym między bitami (kolumnami). Młodsze bity iloczynu $p_3, \dots, p_0$ są generowane w prawej kolumnie macierzy układu mnożącego. Natomiast starsze bity $p_7, \dots, p_4$ są formowane na wyjściach sumatora z przeniesieniem szeregowym RCA, który składa się z jednego półsumatora i dwóch pełnych sumatorów (umieszczonych w dolnym rzędzie na rysunku 9.6). Na schemacie za pomocą skrótów MHA oraz MFA oznaczone zostały jednobitowe półsumatory oraz pełne sumatory, dla których jednym z argumentów jest iloczyn logiczny $a_i \cdot b_j$.

Szybkość działania matrycowego układu mnożącego pokazanego na rysunku 9.6 jest określana przez opóźnienie przesyłania sygnałów przez najdłuższą ścieżkę: od iloczynu częściowego a_3b_0 do bitu p_3 iloczynu.

Kod modułu generatora iloczynów częściowych obliczanych zgodnie z wyrażeniem (9.2) dla matrycowego układu mnożącego 4×4 można przedstawić następująco:

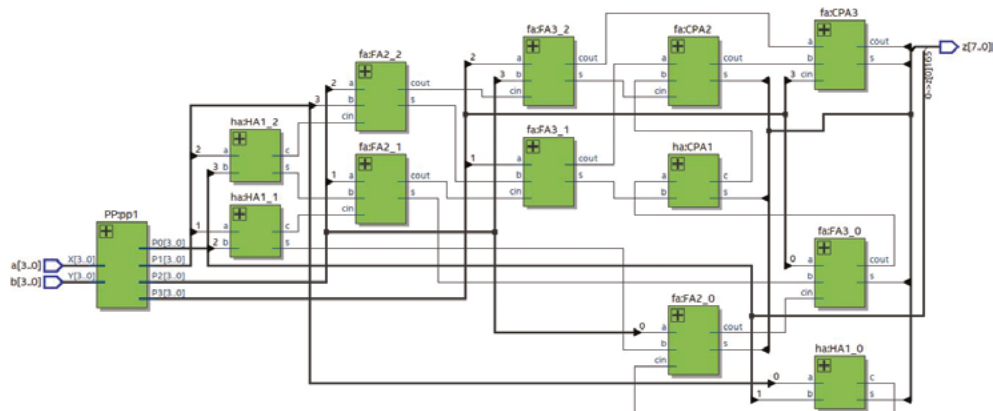
```
module PP ( // generator iloczynów częściowych (partial products)
    output [3:0] P3, P2, P1, P0,
    input [3:0] a,b);

    and pp1(P0[3], a[3], b[0]);
    and pp2(P0[2], a[2], b[0]);
    and pp3(P0[1], a[1], b[0]);
    and pp4(P0[0], a[0], b[0]);
    and pp5(P1[3], a[3], b[1]);
    and pp6(P1[2], a[2], b[1]);
    and pp7(P1[1], a[1], b[1]);
    and pp8(P1[0], a[0], b[1]);
    and pp9(P2[3], a[3], b[2]);
    and pp10(P2[2], a[2], b[2]);
    and pp11(P2[1], a[1], b[2]);
    and pp12(P2[0], a[0], b[2]);
    and pp13(P3[3], a[3], b[3]);
    and pp14(P3[2], a[2], b[3]);
    and pp15(P3[1], a[1], b[3]);
    and pp16(P3[0], a[0], b[3]);
endmodule
```

Kod w języku Verilog, który opisuje cały schemat matrycowego układu mnożącego 4×4 , został zamieszczony poniżej:

```
module array_multiplier_4x4 (  
    output [7:0] z,  
    input [3:0] a, b);  
  
    wire [2:0] c1,c2,c3,c4;  
    wire [2:0] s1,s2,s3;  
    wire [3:0] P0,P1,P2,P3;  
  
    // generacja iloczynów częściowych  
    PP pp1 (P3, P2, P1, P0, a, b);  
  
    // macierz układu mnożenia  
    ha HA1_2 (c1[2],s1[2],P1[2],P0[3]);  
    ha HA1_1 (c1[1],s1[1],P1[1],P0[2]);  
    ha HA1_0 (c1[0],s1[0],P1[0],P0[1]);  
    fa FA2_2 (c2[2],s2[2],P2[2],P1[3],c1[2]);  
    fa FA2_1 (c2[1],s2[1],P2[1],s1[2],c1[1]);  
    fa FA2_0 (c2[0],s2[0],P2[0],s1[1],c1[0]);  
    fa FA3_2 (c3[2],s3[2],P3[2],P2[3],c2[2]);  
    fa FA3_1 (c3[1],s3[1],P3[1],s2[2],c2[1]);  
    fa FA3_0 (c3[0],s3[0],P3[0],s2[1],c2[0]);  
  
    // generowanie młodszych bitów iloczynu  
    assign z[0] = P0[0];  
    assign z[1] = s1[0];  
    assign z[2] = s2[0];  
    assign z[3] = s3[0];  
  
    // generowanie starszych bitów iloczynu  
    // za pomocą jednobitowych sumatorów  
    ha RCA1 (c4[0],z[4],c3[0],s3[1]);  
    fa RCA2 (c4[1],z[5],c3[1],c4[0],s3[2]);  
    fa RCA3 (z[7], z[6], c3[2], c4[1], P3[3]);  
endmodule
```


Wyniki syntezy i symulacji projektu *array_multiplier_4x4* pokazane są na rysunkach 9.7 i 9.8.



RYS. 9.7. Wynik syntezy modułu matrycowego układu mnożącego z przechowywaniem przeniesień dwóch 4-bitowych liczb dwójkowych

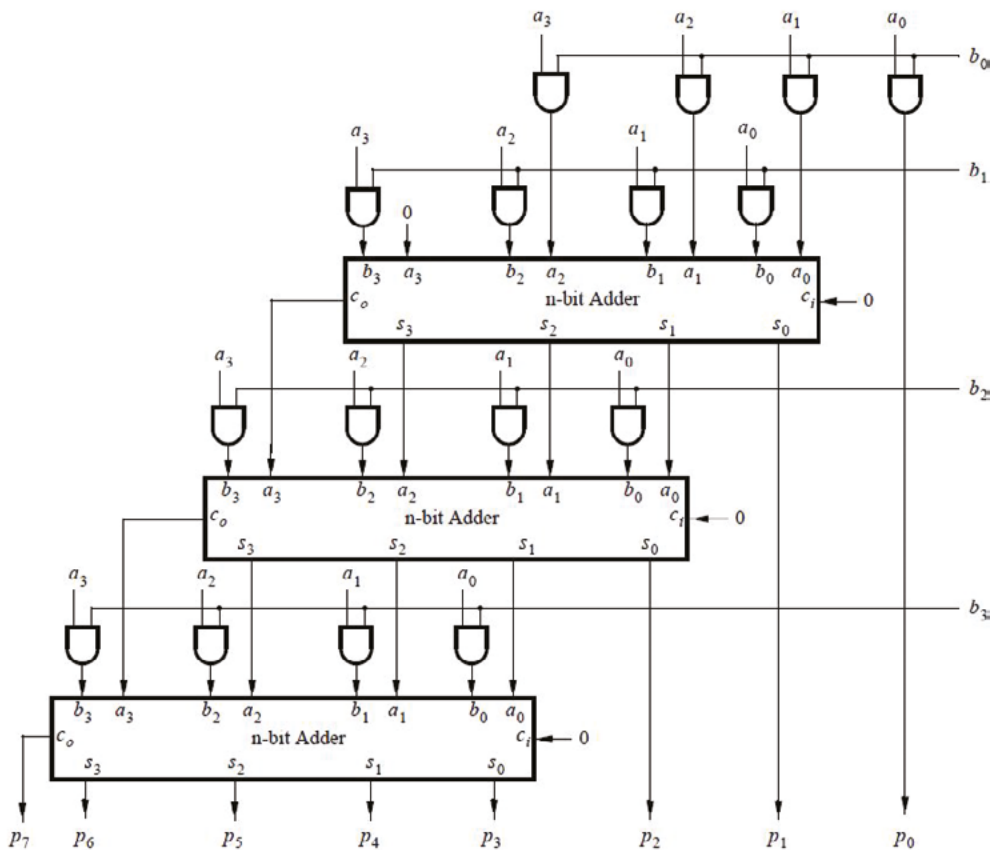
	Name	Value at 0 ps	0 ps	20.0 ns	40.0 ns	60.0 ns	80.0 ns			
	> a	U 2	2	10	1	11	14	3	14	0
	> b	U 5	5	15	14	10	11	12	2	8
	> z	U 10	10	150	14	110	154	36	28	0

RYS. 9.8. Wyniki symulacji modułu matrycowego układu mnożącego z przechowywaniem przeniesień dwóch 4-bitowych liczb dwójkowych

9.2. Układ mnożący z wykorzystaniem sumatorów z przeniesieniem szeregowym

Przedstawiony powyżej matrycowy układ mnożący z przechowywaniem przeniesień CSAM realizuje dodawanie iloczynów bitowych w każdej kolumnie (rysunek 9.6). Innym podejściem do realizacji matrycowych układów mnożących jest dodawanie iloczynów częściowych w wierszach macierzy. Struktura takiego układu została zaprezentowana na rysunku 9.9.

W strukturze matrycowego układu mnożącego na rysunku 9.9 wykorzystane są n -bitowe sumatory do dodawania iloczynów częściowych wiersza i z wartościami sum z poprzedniego wiersza $i-1$. Wejścia przeniesienia c_i sumatorów ustawione są na 0. Sumator wiersza i (gdzie $i = 0, \dots, n-1$) generuje bity sumy oraz przeniesienie c_o . Najmniej znaczący bit sumy s_0 sumatora wiersza i odpowiada bitowi iloczynu p_i , a przeniesienie c_o oraz bity sumy $s_{n-1}, \dots, s_1$ są przekazywane na wejścia sumatora następnego wiersza $i+1$. Na wejścia sumatora górnego wiersza 1 podawane są iloczyny bitowe odpowiadające wierszom 0 i 1 układu mnożącego na rysunku 9.6.



RYS. 9.9. Struktura matrycowego układu mnożącego z wykorzystaniem n-bitowych sumatorów

Opis w języku Verilog matrycowego układu mnożącego z wykorzystaniem sumatorów dla $n = 4$ można przedstawić następująco:

```

module multiplier_4_bits(
    input [3:0] a,b,
    output [7:0] p);

    wire [3:0] PP[3:0];    // jednowymiarowa tablica 4-bitowych elementów,
                          // iloczynzy częściowe  $a * b$ 
    wire [3:0] s[1:3];    // sumy częściowe

    wire cout[1:3];       // przeniesienia z szeregowych 4-bitowych sumatorów

    assign PP[0] = a & {4{b[0]}};
    assign PP[1] = a & {4{b[1]}};

```

```

assign PP[2] = a & {4{b[2]}};
assign PP[3] = a & {4{b[3]}};

```

```

adder_N_bits #(4) ex1({1'b0,PP[0][3:1]},PP[1],1'b0,s[1],cout[1]);
adder_N_bits #(4) ex2({cout[1],s[1][3:1]},PP[2],1'b0,s[2],cout[2]);
adder_N_bits #(4) ex3({cout[2],s[2][3:1]},PP[3],1'b0,s[3],cout[3]);

```

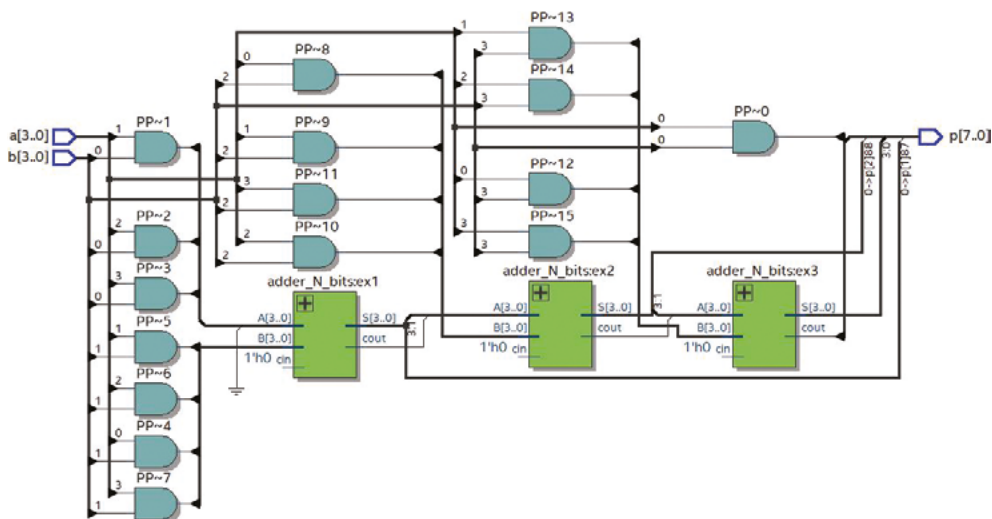
```

assign p[0] = PP[0][0];
assign p[1] = s[1][0];
assign p[2] = s[2][0];
assign p[6:3] = s[3];
assign p[7] = cout[3];

```

endmodule

Wyniki syntezy i symulacji projektu *multiplier_4_bits* pokazane są na rysunkach 9.10 i 9.11.



RYS. 9.10. Wynik syntezy modułu matrycowego układu mnożenia z wykorzystaniem sumatorów dwóch 4-bitowych liczb dwójkowych

Name	Value at	0 ps	40.0 ns	80.0 ns	120.0 ns	160.0 ns
a	U0	0	1	2	3	4
b	U7	7	8	9	10	11
p	U0	0	1	2	3	4

RYS. 9.11. Wyniki symulacji modułu matrycowego układu mnożenia z wykorzystaniem sumatorów dwóch 4-bitowych liczb dwójkowych

Główną zaletą układu mnożącego z rysunku 9.9 jest jego regularna struktura, co umożliwia łatwe opisywanie układów mnożących dla dowolnej liczby bitów. Kod sparametryzowanego modułu matrycowego układu mnożącego można zapisać w następujący sposób:

```
module multiplier_N_bits
    #(parameter N=4)
    (input [N-1:0] a,b,
    output [2*N-1:0] p);

    wire [N-1:0] PP[N-1:0];    // iloczyny częściowe a * b
    wire [N-1:0] s[1:N-1];    // sumy częściowe
    wire cout[1:N-1];    // przeniesienia z szeregowych N-bitowych sumatorów
    genvar i;

    // generowanie iloczynów częściowych
    generate
        for(i=0;i<N;i=i+1) begin:bl1
            assign PP[i] = a & {N{b[i]}};
        end
    endgenerate

    // generowanie sum częściowych
    adder_N_bits #(N) ex1({1'b0,PP[0][N-1:1]},PP[1],1'b0,s[1],cout[1]);
    generate
        for(i=2;i<N;i=i+1) begin:bl2
            adder_N_bits #(N) exi({cout[i-1],s[i-1][N-1:1]},PP[i],1'b0,s[i],cout[i]);
        end
    endgenerate

    // generowanie wyniku
    assign p[0] = PP[0][0];
    generate
        for(i=1;i<N-1;i=i+1) begin:bl3
            assign p[i] = s[i][0];
        end
    endgenerate

    assign p[2*N-2:N-1] = s[N-1];
    assign p[2*N-1] = cout[N-1];
endmodule
```

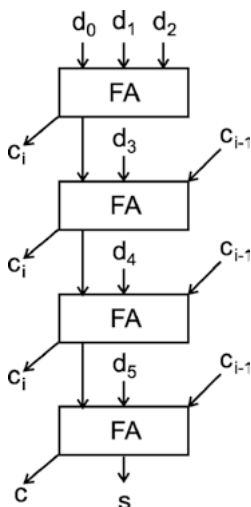
Wyniki syntezy i symulacji projektu *multiplier_N_bits* przy $N=4$ dokładnie odpowiadają wynikom przedstawionym na rysunkach 9.10 i 9.11.

Warto zauważyć, że szybkość działania układu mnożącego można zwiększyć, zastępując sumatory z przeniesieniem szeregowym sumatorami z przeniesieniem równoległym.

9.3. Drzewiasty układ mnożący Wallace'a

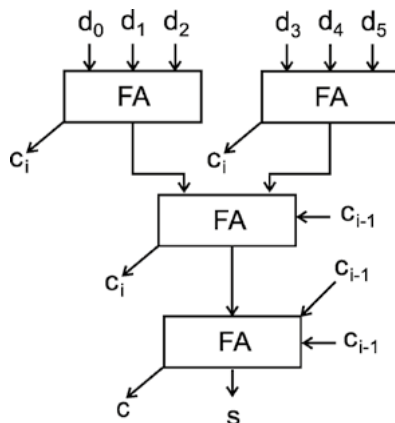
Głównym celem projektowania układów mnożących jest przyspieszenie operacji mnożenia. Jedną z metod skierowanych na zwiększenie szybkości działania jest wykorzystanie *drzewiastego układu mnożącego Wallace'a* (*Wallace Tree Multiplier*). Podstawowym założeniem tej metody syntezy jest takie przypisanie sygnałów wejściowych sumatorów, które wykonują dodawanie iloczynów częściowych w obrębie jednej kolumny macierzy układu mnożącego, aby zminimalizować liczbę poziomów sumatorów.

Założmy, że w pewnym matrycowym układzie mnożącym dodawanie iloczynów częściowych jest wykonywane standardowym sposobem, jak pokazano na rysunku 9.12.



RYS. 9.12. Dodawanie iloczynów częściowych w jednej kolumnie matrycowego układu mnożącego

Taki sam wynik można otrzymać, zmniejszając liczbę poziomów sumatorów (rysunek 9.13).



RYS. 9.13. Zmniejszenie liczby poziomów sumatorów przy dodawaniu iloczynów częściowych w jednej kolumnie matrycowego układu mnożącego

W metodzie syntezy układów mnożących Wallace’a istotne znaczenie mają wagi iloczynów częściowych, czasem nazywane również *ścieżkami*, przez które przekazywane są sygnały między sumatorami. Waga $w(a_n, b_m)$ iloczynu częściowego $a_n \cdot b_m$ jest określana w następujący sposób:

$$w(a_n, b_m) = 2^n \cdot 2^m = 2^{n+m}. \quad (9.3)$$

Proces projektowania układów mnożących Wallace’a składa się z trzech etapów:

- 1) generowanie iloczynów bitowych oraz łączenie ich w grupy, uwzględniając ich wagi;
- 2) zmniejszenie liczby iloczynów częściowych do nie więcej niż dwóch w każdej grupie poprzez wprowadzenie pełnych sumatorów i półsumatorów;
- 3) realizację dodawania otrzymanych ścieżek za pomocą sumatora z przeniesieniem szeregowym.

W pierwszym etapie generowane są wszystkie iloczyny częściowe liczb A i B , które następnie są grupowane zgodnie z ich wagą, określaną według równania (9.3). Dla 4-bitowego układu mnożącego wynik grupowania iloczynów częściowych został przedstawiony w tabeli 9.1.

TABELA 9.1. Grupowanie iloczynów częściowych 4-bitowego układu mnożącego według wag

Numer grupy	Waga ścieżki	Iloczyny częściowe (ścieżki)
0	1	$a_0 \cdot b_0$
1	2	$a_0 \cdot b_1, a_1 \cdot b_0$
2	4	$a_0 \cdot b_2, a_1 \cdot b_1, a_2 \cdot b_0$
3	8	$a_0 \cdot b_3, a_1 \cdot b_2, a_2 \cdot b_1, a_3 \cdot b_0$

Numer grupy	Waga ścieżki	Iloczyny częściowe (ścieżki)
4	16	$a_1 \cdot b_3, a_2 \cdot b_2, a_3 \cdot b_1$
5	32	$a_2 \cdot b_2, a_3 \cdot b_2$
6	64	$a_3 \cdot b_3$

Na drugim etapie następuje redukcja liczby iloczynów częściowych (ścieżek) w poszczególnych grupach. W tym celu rozważane są kolejne grupy ścieżek, zaczynając od grupy 0, i przetwarzane są zgodnie z następującym algorytmem.

Algorytm 9.1. Syntezy układu mnożącego Wallace’a

1. Jeśli grupa zawiera tylko jedną ścieżkę, jest ona przekazywana do kolejnego hierarchicznego poziomu układu mnożącego.
2. Jeśli grupa składa się z dwóch ścieżek, obie są podawane na wejścia półsumatora (czyli do schematu układu mnożącego wprowadzany jest półsumator). Wyjście sumy półsumatora tworzy ścieżkę kolejnego poziomu o tej samej wadze, a wyjście przeniesienia też formuje ścieżkę następnego poziomu, lecz w kolejnej grupie wagowej.
3. Jeśli grupa zawiera trzy ścieżki lub więcej ścieżek, to trzy z nich podawane są na wejścia pełnego sumatora (czyli do schematu układu mnożącego wprowadzany jest pełny sumator jednobitowy). Wyjścia pełnego sumatora stanowią ścieżki kolejnego poziomu podobnie jak w przypadku półsumatora. Pozostałe ścieżki danej grupy są przetwarzane w podobny sposób, zaczynając od punktu 1.

Zauważmy, że proces redukcji liczby ścieżek może skutkować utworzeniem nowych grup o większej wadze.

Następnie proces skracania ścieżek (poprzez zastosowanie pełnych sumatorów i półsumatorów) jest powtarzany dla kolejnych poziomów hierarchii aż do momentu, gdy w każdej grupie pozostaną nie więcej niż dwie ścieżki.

Na trzecim etapie iloczyny częściowe grup zawierających dwie ścieżki oraz iloczyny częściowe z najstarszej grupy są dodawane za pomocą sumatora z przeniesieniem szeregowym.

W procesie syntezy układu mnożącego Wallace’a pełne sumatory oraz półsumatory tworzą hierarchiczną strukturę w formie macierzy. Kolumny takiej macierzy odpowiadają wagom iloczynów częściowych (grupom ścieżek), natomiast wiersze macierzy poziomom hierarchii, na których umieszczane są pełne sumatory i półsumatory w strukturze układu mnożącego. Jeden etap skrócenia ścieżek układu mnożącego formuje jeden wiersz macierzy elementów. Zauważmy, że każdy węzeł takiej macierzy może zawierać kilka ścieżek, a także kilka pełnych sumatorów oraz półsumatorów.

Aby uprościć proces syntezy układu mnożącego Wallace'a, wprowadzimy następujące oznaczenia ścieżek układu mnożącego:

$$N_{i_jk}, \quad (9.4)$$

gdzie: N – skrót od wyrazu *net*; i – numer wiersza macierzy lub numer etapu skrócenia ścieżek układu mnożącego; j – numer kolumny macierzy lub numer grupy ścieżek; k – numer ścieżki w grupie dla tego samego węzła macierzy. Zgodnie z zastosowanymi oznaczeniami iloczynów częściowe z tabeli 9.1 tworzą zerowy poziom hierarchii, czyli dla wszystkich tych ścieżek $i = 0$.

Wprowadzając pełne sumatory i półsumatory do schematu układu mnożącego w procesie syntezy, również stosujemy oznaczenia, które określają ich miejsce w macierzy układu mnożącego w następujący sposób:

$$FA(HA)_{i_j}, \quad (9.5)$$

gdzie: FA – oznacza pełny sumator; HA – półsumator; i – numer wiersza macierzy układu mnożącego; j – numer kolumny macierzy układu mnożącego.

W tabeli 9.2 przedstawione zostały iloczyny częściowe zerowego poziomu oraz ich nowe oznaczenia.

TABELA 9.2. Oznaczenia iloczynów częściowych zerowego poziomu

Iloczyn częściowe	Oznaczenia
$a_0 \cdot b_0$	N_{0_00}
$a_1 \cdot b_0$	N_{0_10}
$a_0 \cdot b_1$	N_{0_11}
$a_2 \cdot b_0$	N_{0_20}
$a_1 \cdot b_1$	N_{0_21}
$a_0 \cdot b_2$	N_{0_22}
$a_3 \cdot b_0$	N_{0_30}
$a_2 \cdot b_1$	N_{0_31}
$a_1 \cdot b_2$	N_{0_32}
$a_0 \cdot b_3$	N_{0_33}
$a_3 \cdot b_1$	N_{0_40}
$a_2 \cdot b_2$	N_{0_41}
$a_1 \cdot b_3$	N_{0_42}
$a_3 \cdot b_2$	N_{0_50}
$a_2 \cdot b_3$	N_{0_51}
$a_3 \cdot b_3$	N_{0_60}

Przeprowadzimy syntezę układu mnożącego Wallace'a dla dwóch 4-bitowych liczb. Wyniki zastosowania algorytmu 9.1 do ścieżek zerowego poziomu pokazane są w tabeli 9.3. Jej pierwsze trzy kolumny odpowiadają tabeli 9.1, jednak iloczyny częściowe zostały zastąpione ich oznaczeniami. W czwartej kolumnie tabeli 9.3 znajdują się sumatory i półsumatory wprowadzone do schematu układu mnożącego w procesie realizacji algorytmu 9.1.

TABELA 9.3. Rozmieszczenie ścieżek w grupach zerowego poziomu

Numer grupy	Waga	Ilozyny częściowe (ścieżki)	Sumatory
0	1	N_0_0_0	
1	2	N_0_1_1, N_0_1_0	HA_1_1
2	4	N_0_2_2, N_0_2_1, N_0_2_0	FA_1_2
3	8	N_0_3_3, N_0_3_2, N_0_3_1, N_0_3_0	FA_1_3
4	16	N_0_4_2, N_0_4_1, N_0_4_0	FA_1_4
5	32	N_0_5_1, N_0_5_0	HA_1_5
6	64	N_0_6_0	

Grupa 0 ścieżek o wadze 1 zawiera tylko jedną ścieżkę N_0_0_0, dlatego jest ona przekazywana do następnego poziomu układu mnożącego bez żadnych zmian. Grupa 1 ścieżek o wadze 2 ma dwie ścieżki N_0_1_1 i N_0_1_0, które zgodnie z algorytmem 9.1 są podawane na wejścia półsumatora HA_1_1. Wyjście sumy półsumatora HA_1_1 tworzy nową ścieżkę N_1_1_0 w grupie 1, a wyjście przeniesienia tworzy ścieżkę N_1_2_0 w grupie 2. Grupa 2 ścieżek o wadze 4 składa się z trzech ścieżek N_0_2_2, N_0_2_1 i N_0_2_0, które są przekazywane na wejścia pełnego sumatora bitowego FA_1_2. Wyjście sumy pełnego sumatora FA_1_2 stanowi ścieżkę N_1_2_1 w grupie 2, natomiast wyjście przeniesienia – ścieżkę N_1_3_0 w grupie 3. Grupa 3 ścieżek o wadze 8 zawiera cztery ścieżki N_0_3_3, N_0_3_2, N_0_3_1 i N_0_3_0. Pierwsze trzy ścieżki, tj. N_0_3_3, N_0_3_2, N_0_3_1 podawane są na wejścia pełnego sumatora FA_1_3, a ścieżka N_0_3_0 jest przekazywana do następnego poziomu układu mnożącego. Wyjście sumy pełnego sumatora FA_1_3 tworzy ścieżkę N_1_3_1 w grupie 3, a wyjście przeniesienia stanowi ścieżkę N_1_4_0 w grupie 4. Analogicznie ścieżki N_0_4_2, N_0_4_1 i N_0_4_0 grupy 4 o wadze 16 są łączone za pomocą pełnego sumatora FA_1_4. Wyjście sumy pełnego sumatora FA_1_4 stanowi ścieżkę N_1_4_1 w grupie 4, a wyjście przeniesienia – ścieżkę N_1_5_0 w grupie 5. W grupie 5 na poziomie zerowym znajdują się tylko dwie ścieżki N_0_5_1 i N_0_5_0, które łączone są półsumatorem HA_1_5. Wyjście sumy półsumatora HA_1_5 tworzy ścieżkę N_1_5_1, a wyjście przeniesienia – ścieżkę N_1_6_0. Grupa 6 na zerowym poziomie ma tylko jedną ścieżkę N_0_6_0, która przesyłana zostaje do następnego poziomu hierarchii. W tabeli 9.3 ścieżki połączone sumatorami zostały pogrubione.

Rozmieszczenie ścieżek układu mnożącego w grupach, po wykonaniu skracania na poziomie zerowym, pokazane jest w tabeli 9.4.

TABELA 9.4. Rozmieszczenie ścieżek w grupach pierwszego poziomu

Numer grupy	Waga	Iloczynny częściowe (ścieżki)	Sumatory
0	1	N_0_0_0	
1	2	N_1_1_0	
2	4	N_1_2_0, N_1_2_1	HA_2_2
3	8	N_1_3_0, N_1_3_1, N_0_3_0	FA_2_3
4	16	N_1_4_0, N_1_4_1	HA_2_4
5	32	N_1_5_0, N_1_5_1	HA_2_5
6	64	N_1_6_0, N_0_6_0	HA_2_6

W analogiczny sposób redukuje się liczbę ścieżek na pierwszym poziomie. Do układu mnożącego wprowadzane są pełne sumatory oraz półsumatory drugiego poziomu, jak zaznaczono w ostatniej kolumnie tabeli 9.4. Nowo wprowadzone sumatory tworzą ścieżki drugiego poziomu. Wyjścia sum formują ścieżki kolejnego poziomu w tej samej grupie, a wyjścia przeniesienia – w następnej grupie. Rozmieszczenie ścieżek w grupach drugiego poziomu zostało zaprezentowane w tabeli 9.5.

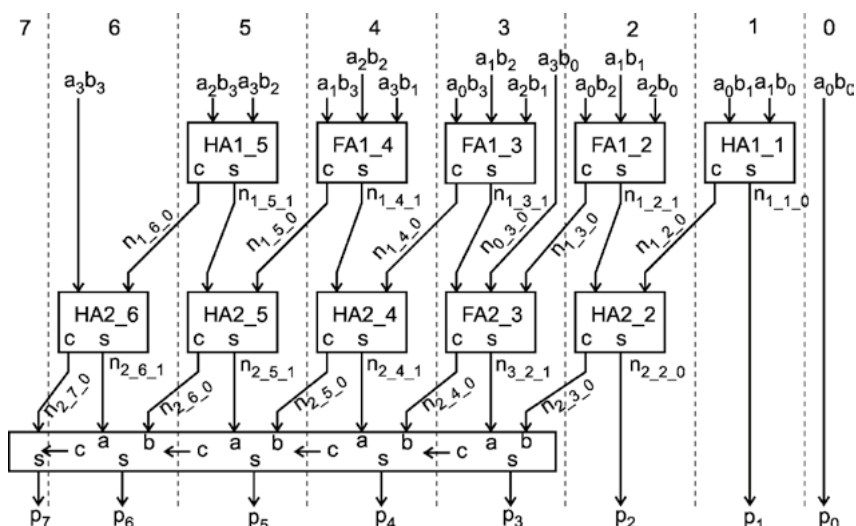
TABELA 9.5. Rozmieszczenie ścieżek w grupach drugiego poziomu hierarchii

Numer grupy	Waga	Iloczynny częściowe (ścieżki)
0	1	N_0_0_0
1	2	N_1_1_0
2	4	N_2_2_0
3	8	N_2_3_1, N_2_3_0
4	16	N_2_4_1, N_2_4_0
5	32	N_2_5_1, N_2_5_0
6	64	N_2_6_1, N_2_6_0
7	128	N_2_7_0

W tabeli 9.5 zauważalne jest, że redukcja liczby ścieżek na pierwszym poziomie spowodowała powstanie nowej grupy o numerze 7 dla ścieżek o wadze 128. W każdej grupie tabeli 9.5 znajdują się nie więcej niż dwie ścieżki, co oznacza, że proces redukcji iloczynów częściowych zostaje zakończony.

W tabeli 9.5 każda z grup 0, 1 i 2 zawiera tylko jedną ścieżkę, więc mogą być one przekazane na wyjścia układu mnożącego jako bity iloczynu, odpowiednio, p_0 , p_1 i p_2 . Ścieżki grup 3, 4, 5, 6 i 7 podawane są na wejścia sumatora szeregowego, który generuje bity iloczynu p_3 , p_4 , p_5 , p_6 i p_7 . W celu zwiększenia szybkości działania sumator szeregowy może zostać zastąpiony sumatorem z przeniesieniem równoległym.

Ostateczna realizacja drzewiastego matrycowego układu mnożącego Wallace'a została przedstawiona na rysunku 9.14.



RYS. 9.14. Struktura układu mnożącego Wallace'a 4×4

Kod w języku Verilog układu mnożącego Wallace'a można przedstawić następująco:

```

module wallace(
    input [3:0] a,b,
    output [7:0] p);

    wire n0_0_0,n0_1_0,n0_1_1,n0_2_0,n0_2_1,n0_2_2,n0_3_0,n0_3_1,n0_3_2,n0_3_3;
    wire n0_4_0,n0_4_1,n0_4_2,n0_5_0,n0_5_1,n0_6_0;
    wire n1_1_0,n1_2_0,n1_2_1,n1_3_0,n1_3_1,n1_4_0,n1_4_1,n1_5_0,n1_5_1,n1_6_0;
    wire n2_2_0,n2_3_0,n2_3_1,n2_4_0,n2_4_1,n2_5_0,n2_5_1,n2_6_0,n2_6_1,n2_7_0;
    wire [3:0] c;

    // generowanie iloczynów częściowych (ścieżek) pierwszego poziomu
    assign {n0_3_0,n0_2_0,n0_1_0,n0_0_0} = a & {4{b[0]}};
    assign {n0_4_0,n0_3_1,n0_2_1,n0_1_1} = a & {4{b[1]}};
    assign {n0_5_0,n0_4_1,n0_3_2,n0_2_2} = a & {4{b[2]}};
    assign {n0_6_0,n0_5_1,n0_4_2,n0_3_3} = a & {4{b[3]}};

    // sumatory pierwszego poziomu
    half_adder ha1_1(n0_1_0,n0_1_1,n1_1_0,n1_2_0);
    full_adder fa1_2(n0_2_0,n0_2_1,n0_2_2,n1_2_1,n1_3_0);
    full_adder fa1_3(n0_3_1,n0_3_2,n0_3_3,n1_3_1,n1_4_0);
    full_adder fa1_4(n0_4_0,n0_4_1,n0_4_2,n1_4_1,n1_5_0);
    half_adder ha1_5(n0_5_0,n0_5_1,n1_5_1,n1_6_0);

```

```

// sumatory drugiego poziomu
half_adder ha2_2(n1_2_0,n1_2_1,n2_2_0,n2_3_0);
full_adder fa2_3(n1_3_0,n1_3_1,n0_3_0,n2_3_1,n2_4_0);
half_adder ha2_4(n1_4_0,n1_4_1,n2_4_1,n2_5_0);
half_adder ha2_5(n1_5_0,n1_5_1,n2_5_1,n2_6_0);
half_adder ha2_6(n1_6_0,n0_6_0,n2_6_1,n2_7_0);

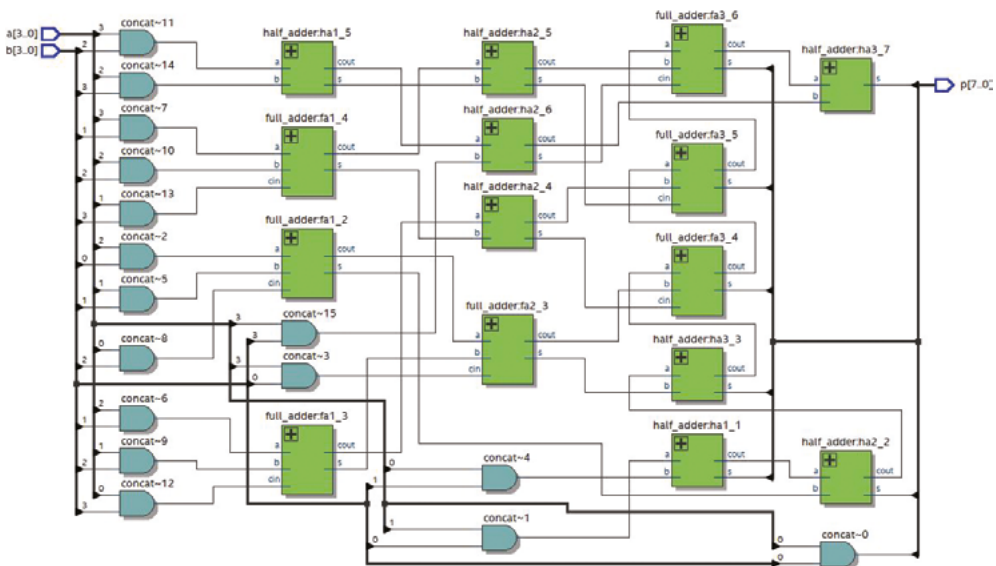
// generowanie młodszych bitów iloczynu
assign p[0] = n0_0_0;
assign p[1] = n1_1_0;
assign p[2] = n2_2_0;

// generowanie starszych bitów iloczynu
half_adder ha3_3(n2_3_0,n2_3_1,p[3],c[0]);
full_adder fa3_4(c[0],n2_4_0,n2_4_1,p[4],c[1]);
full_adder fa3_5(c[1],n2_5_0,n2_5_1,p[5],c[2]);
full_adder fa3_6(c[2],n2_6_0,n2_6_1,p[6],c[3]);
half_adder ha3_7(c[3],n2_7_0,p[7],);

endmodule

```

W przytoczonym kodzie sumator z przeniesieniem szeregowym najniższego poziomu został zrealizowany za pomocą jednobitowych sumatorów: HA3_3, FA3_4, FA3_5, FA3_6 i HA3_7. Wyniki syntezy i symulacji układu mnożącego Wallace'a o rozmiarze 4×4 bity zamieszczono na rysunkach 9.15 i 9.16.



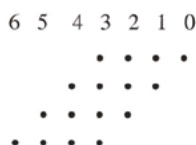
RYS. 9.15. Wynik syntezy układu mnożącego Wallace'a 4×4 bity

Name	Value at 0 ps	40.0 ns	80.0 ns	120.0 ns	160.0 ns
a	U0	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b	U9	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
p	U0	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

RYS. 9.16. Wyniki symulacji układu mnożącego Wallace'a 4 × 4 bity

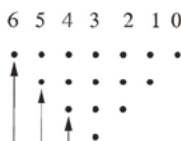
Analizując strukturę matrycowego układu mnożącego Wallace'a przedstawioną na rysunku 9.14, można zauważyć, że kluczowym aspektem procesu syntezy jest przekształcanie iloczynów częściowych każdego poziomu na iloczyny częściowe następnego poziomu przy użyciu sumatorów i półsumatorów. Warto zauważyć, że każde przeniesienie generuje iloczyn częściowy na tym samym poziomie, lecz w kolejnej kolumnie (w następnej grupie).

Iloczyny częściowe matrycowego układu mnożącego mogą być zobrazowane za pomocą diagramu kropkowego, co przedstawiono na rysunku 9.17.



RYS. 9.17. Diagram kropkowy oryginalnej macierzy układu mnożącego 4 × 4 bity

Mając na uwadze przemienność i łączność operacji dodawania, macierz z rysunku 9.17 może być przedstawiona w formie ukazanej na rysunku 9.18. Diagram na rysunku 9.18 potwierdza, że kolejność dodawania iloczynów częściowych w każdej kolumnie nie ma znaczenia.

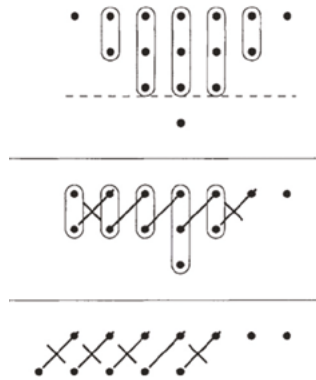


RYS. 9.18. Diagram kropkowy zreorganizowanej macierzy mnożenia 4 × 4 bity

Z uwagi na podobieństwo macierzy na rysunku 9.18 do drzewa układ mnożący Wallace'a jest określany jako *drzewiasty układ mnożący*. Korzystając z wprowadzonych oznaczeń, etapy realizacji algorytmu syntezy tego układu można przedstawić za pomocą sekwencji diagramów kropkowych, jak pokazano na rysunku 9.19.

W diagramach na rysunku 9.19 zakreślone kropki reprezentują iloczyny częściowe, które są dodawane za pomocą sumatorów (trzy kropki) lub półsumatorów (dwie kropki). Należy zauważyć, że każdy dodany sumator bądź półsumator generuje w diagramie na kolejnym etapie nową kropkę w danej kolumnie, reprezentującą sumę, oraz jedną kropkę w następnej kolumnie, symbolizującą przeniesienie. Linie

ukośne na rysunku 9.19 oznaczają wyjścia sumatorów, a przekreślone linie ukośne wskazują wyjścia półsumatorów.



RYS. 9.19. Diagramy kropkowe procesu syntezy układu mnożącego Wallace'a 4×4 bity

Proces wprowadzania sumatorów i półsumatorów do schematu nazywany jest *kompresją kolumn* (*column compression*), ponieważ zmniejsza liczbę kropek w kolumnach. Proces kompresji zostanie zakończony, gdy w każdej kolumnie pozostaną nie więcej niż dwie kropki (iloczyny częściowe). Na ostatnim etapie syntezy iloczyny częściowe są sumowane za pomocą standardowego sumatora z przeniesieniem szeregowym.

Różne strategie wprowadzania sumatorów i półsumatorów w procesie kompresji kolumn matrycowego układu mnożącego doprowadziły do powstania różnorodnych sposobów realizacji układów mnożących. Jednym z najbardziej znanych jest układ mnożący Dadda.

9.4. Układ mnożący Dadda

Metoda syntezy układów mnożących Dadda skupia się na minimalizacji liczby wykorzystywanych sumatorów i półsumatorów. Jej podstawę stanowi drzewiasty diagram kropkowy Wallace'a (rysunek 9.18), który należy zrealizować, stosując minimalną liczbę sumatorów i półsumatorów. W tym celu najpierw określana jest maksymalna wysokość kolumn w diagramach kropkowych na każdym etapie syntezy za pomocą następującego wzoru:

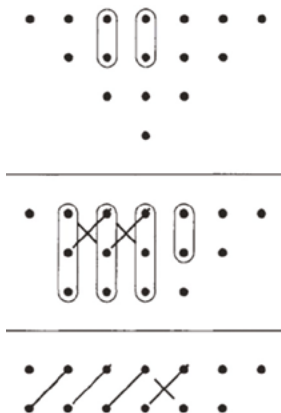
$$d_1 = 2, d_{i+1} = \text{int}(1,5 \times d_i), \quad (9.6)$$

gdzie $\text{int}(A)$ – najmniejsza liczba całkowita, większa lub równa A .

Wzór (9.6) pozwala otrzymać następujący ciąg wartości wysokości kolumn:

$$2, 3, 4, 6, 9, 13, 19, 29, 42, 63, 94, \dots$$

Charakterystyczną cechą metody syntezy Dadda jest numeracja etapów od końca. Na przykład dla układu mnożącego Dadda na pierwszym etapie syntezy $j = 3$ i $d_j = 4$, na drugim $j = 2$ i $d_j = 3$, a na trzecim etapie $j = 1$ i $d_j = 2$. Proces syntezy układu mnożącego Dadda 4×4 bity przedstawiono na rysunku 9.20.



RYS. 9.20. Diagramy kropkowe syntezy układu mnożącego Dadda 4×4 bity

Na pierwszym etapie syntezy, gdzie $j = 3$ i $d_j = 4$, ograniczenie na liczbę kropek jest spełnione dla każdej kolumny. Jednak aby również na drugim etapie, przy $j = 2$ i $d_2 = 3$, zapewnić spełnienie tych ograniczeń, konieczne jest zmniejszenie liczby kropek w kolumnie 3 o 1. W tym celu wprowadzony zostaje do schematu układu mnożącego półsumator HA1_3. Przeniesienie z półsumatora HA1_3 utworzy kropkę w kolumnie 4 na drugim etapie, co przekracza wartość $d_2 = 3$. W rezultacie konieczne jest dodanie do schematu półsumatora HA1_4. Analogicznie na trzecim etapie syntezy dodawane są sumatory i półsumatory. Proces syntezy układu mnożącego Dadda 4×4 bity został również przedstawiony w tabelach 9.6–9.8. Na rysunku 9.21 zobrażona została struktura układu mnożącego Dadda 4×4 bity.

TABELA 9.6. Podział ścieżek zerowego poziomu na grupy przy syntezie układu mnożącego Dadda 4×4 bity

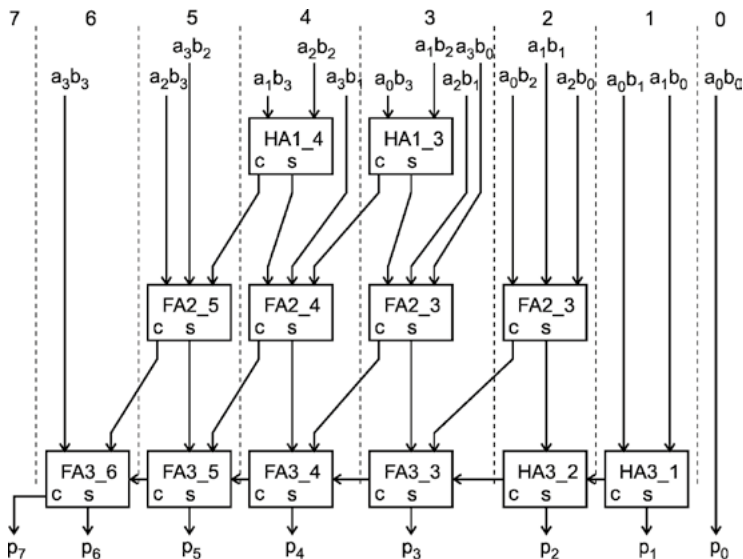
Numer grupy	Ścieżki	Sumatory
0	N_0_0_0	
1	N_0_1_1, N_0_1_0	
2	N_0_2_2, N_0_2_1, N_0_2_0	
3	N_0_3_3, N_0_3_2, N_0_3_1, N_0_3_0	HA1_3
4	N_0_4_2, N_0_4_1, N_0_4_0	HA_1_4
5	N_0_5_1, N_0_5_0	
6	N_0_6_0	

TABELA 9.7. Rozmieszczenie ścieżek pierwszego poziomu w grupach przy syntezie układu mnożącego Dadda 4×4 bity

Numer grupy	Ścieżki	Sumatory
0	N_0_0_0	
1	N_0_1_1, N_0_1_0	
2	N_0_2_2, N_0_2_1, N_0_2_0	FA2_2
3	N_1_3_0, N_0_3_1, N_0_3_0	FA2_3
4	N_1_4_1, N_0_4_1, N_0_4_0	FA2_4
5	N_0_5_1, N_0_5_0, N_1_5_0	FA2_5
6	N_0_6_0	

TABELA 9.8. Rozmieszczenie ścieżek ostatniego poziomu w grupach przy syntezie układu mnożącego Dadda 4×4 bity

Numer grupy	Ścieżki	Sumatory
0	N_0_0_0	
1	N_0_1_1, N_0_1_0	HA3_1
2	N_2_2_1, c1	HA3_2
3	N_2_3_1, N_2_3_0, c2	FA3_3
4	N_2_4_1, N_2_4_0, c3	FA3_4
5	N_2_5_1, N_2_5_0, c4	FA3_5
6	N_0_6_0, N_2_6_0, c5	FA3_6



RYS. 9.21. Struktura matrycowego układu mnożącego Dadda 4×4 bity

Poniżej zamieszczony został kod w języku Verilog układu mnożącego Dadda.

```
module dadda(
    input [3:0] a,b,
    output [7:0] p);

    wire n0_0_0,n0_1_0,n0_1_1,n0_2_0,n0_2_1,n0_2_2,n0_3_0,n0_3_1,n0_3_2,n0_3_3;
    wire n0_4_0,n0_4_1,n0_4_2,n0_5_0,n0_5_1,n0_6_0;
    wire n1_3_0,n1_4_0,n1_4_1,n1_5_0;
    wire n2_2_0,n2_3_0,n2_3_1,n2_4_0,n2_4_1,n2_5_0,n2_5_1,n2_6_0;
    wire [5:1] c;

    // generowanie iloczynów częściowych (ścieżek) pierwszego poziomu
    assign {n0_3_0,n0_2_0,n0_1_0,n0_0_0} = a & {4{b[0]}};
    assign {n0_4_0,n0_3_1,n0_2_1,n0_1_1} = a & {4{b[1]}};
    assign {n0_5_0,n0_4_1,n0_3_2,n0_2_2} = a & {4{b[2]}};
    assign {n0_6_0,n0_5_1,n0_4_2,n0_3_3} = a & {4{b[3]}};

    // sumatory pierwszego poziomu
    half_adder ha1_3(n0_3_2,n0_3_3,n1_3_0,n1_4_0);
    half_adder fa1_4(n0_4_1,n0_4_2,n1_4_1,n1_5_0);

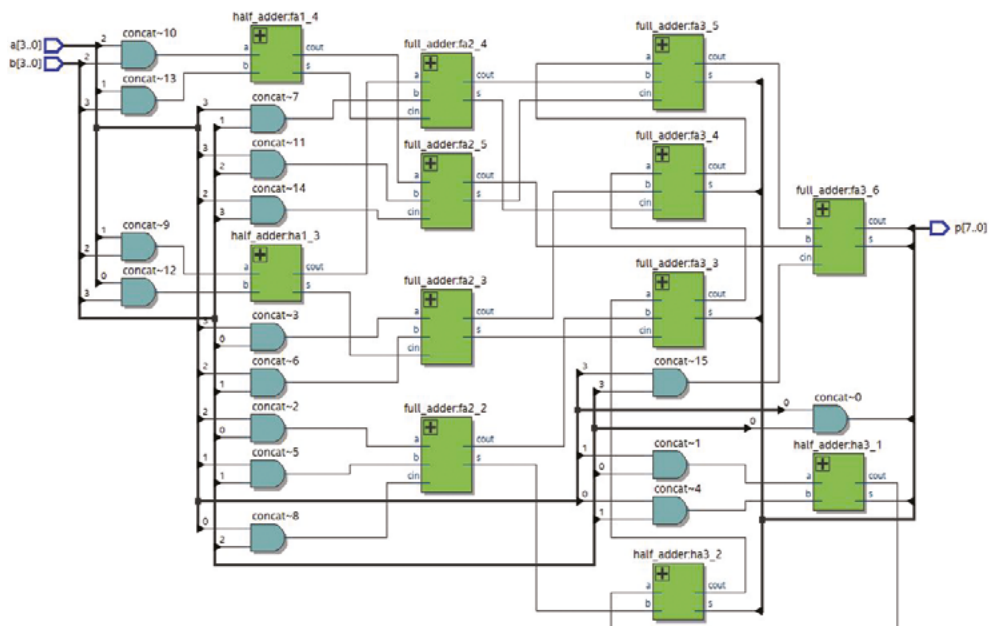
    // sumatory drugiego poziomu
    full_adder fa2_2(n0_2_0,n0_2_1,n0_2_2,n2_2_0,n2_3_0);
    full_adder fa2_3(n0_3_0,n0_3_1,n1_3_0,n2_3_1,n2_4_0);
    full_adder fa2_4(n1_4_0,n0_4_0,n1_4_1,n2_4_1,n2_5_0);
    full_adder fa2_5(n1_5_0,n0_5_0,n0_5_1,n2_5_1,n2_6_0);

    // generowanie młodszych bitów iloczynu
    assign p[0] = n0_0_0;

    // generowanie starszych bitów iloczynu
    half_adder ha3_1(n0_1_0,n0_1_1,p[1],c[1]);
    half_adder ha3_2(c[1],n2_2_0,p[2],c[2]);
    full_adder fa3_3(c[2],n2_3_0,n2_3_1,p[3],c[3]);
    full_adder fa3_4(c[3],n2_4_0,n2_4_1,p[4],c[4]);
    full_adder fa3_5(c[4],n2_5_0,n2_5_1,p[5],c[5]);
    full_adder fa3_6(c[5],n2_6_0,n0_6_0,p[6],p[7]);

endmodule
```

Wyniki syntezy i symulacji układu mnożącego Dadda 4×4 bity pokazane zostały na rysunkach 9.22 i 9.23.



RYS. 9.22. Wynik syntezy układu mnożącego Dadda 4×4 bity na poziomie przesłań międzyrejestrowych

Name	Value at 0 ps	0 ps	40.0 ns	80.0 ns	120.0 ns	160.0 ns
a	U:0	0	1	2	3	4
b	U:4	4	14	8	13	9
p	U:0	0	4	28	24	52

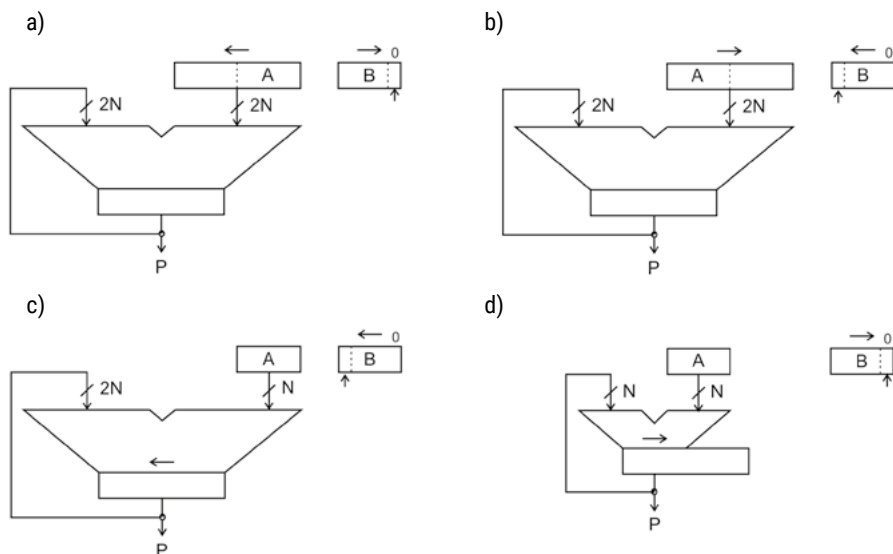
RYS. 9.23. Wyniki symulacji układu mnożącego Dadda 4×4 bity

Porównanie metod syntezy układów mnożących Wallace'a i Dadda wykazuje, że metoda Wallace'a jest skoncentrowana na projektowaniu bardzo szybkich układów mnożących. Dlatego największa liczba sumatorów i półsumatorów jest wprowadzana na wczesnych etapach syntezy w celu zminimalizowania długości ścieżki przeniesienia sumatora szeregowego na ostatnim poziomie. Z kolei metoda Dadda dąży do minimalizacji liczby wykorzystywanych sumatorów i półsumatorów. W tym przypadku główne obciążenie obliczeniowe przeniesione jest na sumator szeregowy ostatniego poziomu. Warto zauważyć, że w obu układach, Wallace'a i Dadda, w celu zwiększenia szybkości działania na ostatnim poziomie zamiast szeregowego sumatora można zastosować sumator równoległy.

9.5. Algorytmiczne metody realizacji układów mnożących

Omówione wyżej podejścia do projektowania układów mnożących należą do strukturalnych metod projektowania w języku Verilog – użytkownik samodzielnie projektuje schemat układu, a język Verilog służy jedynie do opisu komponentów schematu oraz ich połączeń ze sobą. W innym podejściu do projektowania układów mnożących w języku Verilog algorytm działania układu jest opisywany za pomocą instrukcji proceduralnych.

Zaprojektujemy układ mnożący, który realizuje operację mnożenia $P = A \cdot B$, gdzie operandy A i B mają jednakowy rozmiar N bitów. W literaturze wyodrębnia się cztery podstawowe algorytmiczne metody realizacji układów mnożących (rysunek 9.24). Pierwsza z nich (rysunek 9.24a) polega na wykonaniu na każdej iteracji dodawania z akumulacją $P := P + A'$ lub $P := P + 0$, w zależności od wartości młodszego bitu mnożnika, gdzie A' oznacza iloczyn częściowy uzyskany przez przesunięcie liczby A o jeden bit w lewo.



RYS. 9.24. Struktury układów mnożących: a) z przesunięciem mnożnika w prawo, mnożnej w lewo, analizowany jest młodszy bit mnożnika; b) z przesunięciem mnożnika w lewo, mnożnej w prawo, analizowany jest starszy bit mnożnika; c) z przesunięciem mnożnika w lewo, akumulatora w lewo, analizowany jest starszy bit mnożnika; d) z przesunięciem mnożnika w prawo, akumulatora w prawo, analizowany jest młodszy bit mnożnika

Funkcjonowanie układu mnożącego przedstawionego na rysunku 9.24a można opisać za pomocą następującego algorytmu.

Algorytm 9.2. Przesunięcie mnożnika w prawo, mnożnej w lewo, analizowany jest młodszy bit mnożnika:

1. Przyjąć $i = 0, P = 0$.
2. Sprawdzić bit b_0 . Jeśli $b_0 = 1$, wykonać operację $P := P + A$, inaczej ($b_0 = 0$) dodać wartość zerową $P := P + 0$.
3. Wykonać instrukcje $i := i + 1, B := B \gg 1, A := A \ll 1$.
4. Punkty 2 i 3 powtórzyć N razy.
5. Koniec.

Algorytm ten implementuje klasyczny sposób mnożenia pisemnego. Opis algorytmu 9.2 w języku Verilog może zostać zdefiniowany w następujący sposób:

```

module algorithm_mult
    #(parameter N=4)
    (input [N-1:0] a,b,
     output [2*N-1:0] p);

    reg [N-1:0] rb;
    reg [2*N-1:0] ra,rp;

    integer i;

    always @(*)
    begin
        ra = a;
        rb = b;
        rp = 0;
        for (i=0; i<N; i=i+1)
        begin
            if (rb[0])    rp = rp + ra;
            else        rp = rp + {2*N{1'b0}};
            rb = rb >> 1;
            ra = ra << 1;
        end
    end
    assign p = rp;
endmodule

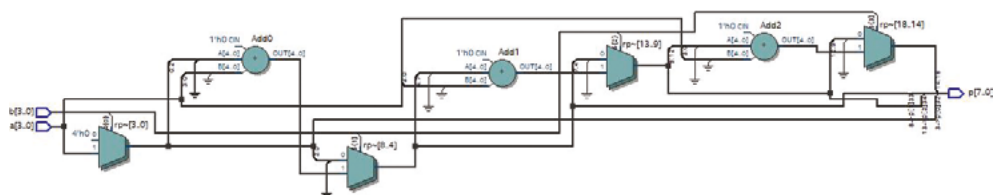
```

Instrukcje proceduralne języka Verilog operują jedynie na zmiennych typu **reg**. W związku z tym w programie zdefiniowane zostały zmienne pomocnicze ra , rb i rp typu **reg**. Należy zauważyć, że zmienna ra ma rozmiar $2N$, ponieważ podczas mnożenia N razy będzie wykonywane przesunięcie mnożnej A o jeden bit w lewo. W bloku **always** na początku następuje inicjalizacja zmiennych ra , rb i rp . Następnie

algorytm mnożenia jest realizowany poprzez N iteracji za pomocą instrukcji pętli **for**. Na każdej iteracji, w zależności od wartości $rb[0]$, do akumulatora rp dodawana jest liczba ra lub wartość zerowa. Nie można pominąć konstrukcji **else**, ponieważ mogłoby to spowodować automatyczne wprowadzenie zatrząsków do schematu. Można także zapisać:

else $rp = rp$;

jednak w tym przypadku kompilator mógłby dodać pętlę sprzężenia zwrotnego do układu kombinacyjnego. Na końcu każdej iteracji następuje przesunięcie logiczne mnożnika rb o jeden bit w prawo oraz mnożnej ra o jeden bit w lewo. Wyniki syntezy i symulacji projektu *algorithm_mult* zamieszczone zostały na rysunkach 9.25 i 9.26.



RYS. 9.25. Wynik syntezy modułu *algorithm_mult* na poziomie przesłań międzyrejestrowych

Name	Value at 0 ps	40.0 ns	80.0 ns	120.0 ns	160.0 ns
a	U 0	0	1	2	3
b	U 2	2	14	9	8
p	U 0	0	14	18	24

RYS. 9.26. Wyniki symulacji modułu *algorithm_mult*

Funkcjonowanie układu mnożącego, którego struktura przedstawiona została na rysunku 9.24b, można opisać za pomocą następującego algorytmu.

Algorytm 9.3. Przesunięcie mnożnika w lewo, mnożnej w prawo, analizowany jest starszy bit mnożnika:

1. Przyjąć $i = 0$, $P = 0$.
2. Jeśli $i = N$, przejść do punktu 6.
3. Wykonać przesunięcie mnożnej $A := A \gg 1$.
4. Jeśli $B[N-1] = 1$, wykonać operację dodawania $P := P + A$, inaczej dodać do akumulatora wartość zerowa $P := P + 0$.
5. Wykonać instrukcje $B := B \ll 1$, $i := i + 1$. Przejść do punktu 2.
6. Koniec.

Zauważmy, że w przedstawionym algorytmie przesunięcie mnożnej A o jeden bit w prawo jest wykonywane przed jej dodaniem do zawartości akumulatora P . Opis algorytmu 2 w języku Verilog można przedstawić następująco:

Funkcjonowanie układu mnożącego przedstawionego na rysunku 9.24c można opisać za pomocą następującego algorytmu.

Algorytm 9.4. Przesunięcie mnożnika w lewo, akumulatora w lewo, analizowany jest starszy bit mnożnika:

1. Przyjąć $i = 0, P = 0$.
2. Jeśli $i = N$, przejść do punktu 6.
3. Wykonać przesunięcie akumulatora $P := P \ll 1$.
4. Jeśli $B[N-1] = 1$, wykonać operację $P := P + A$, inaczej $P := P + 0$.
5. Wykonać instrukcje $B := B \ll 1, i := i + 1$. Przejść do punktu 2.
6. Koniec.

W algorytmie 9.4 przesunięcie iloczynu P w lewo wykonywane jest przed dodaniem do niego mnożnej A . Opis algorytmu 9.4 w języku Verilog zamieszczony został poniżej:

```

module algorithm_mult_c
    #(parameter N=4)
    (input [N-1:0] a,b,
     output [2*N-1:0] p);

    reg [N-1:0] ra,rb;
    reg [2*N-1:0] rp;
    integer i;

    always @(*)
        begin
            ra = a;
            rb = b;
            rp = 0;
            for (i=0; i<N; i=i+1)
                begin
                    rp = rp << 1;           // przesunięcie iloczynu w lewo
                    if (rb[N-1])    rp = rp + {{N{1'b0}},ra};
                    else           rp = rp + {2*N{1'b0}};
                    rb = rb << 1;
                end
            end
            assign p = rp;
        endmodule

```

Wynik syntezy modułu *algorithm_mult_c* na poziomie przesłań międzyrejestrów całkowicie się pokrywa z rysunkiem 9.27.

Funkcjonowanie układu mnożącego zamieszczonego na rysunku 9.24d opisywane jest za pomocą następującego algorytmu.

Algorytm 9.5. Przesunięcie mnożnika w prawo, akumulatora w prawo, analizowany jest młodszy bit mnożnika:

1. Przyjąć $i = 0$, $P = 0$.
2. Jeśli $i = N$, przejść do punktu 5.
3. Jeśli $B[0] = 1$, wykonać operację $P[2*N:N] := P[2*N-1:N] + A$, inaczej $P := P + 0$.
4. Wykonać instrukcje $B := B \gg 1$, $P := P \gg 1$, $i := i + 1$. Przejść do punktu 2.
5. Koniec.

Cechą szczególną struktury przedstawionej na rysunku 9.24c jest to, że użyto w niej N -bitowego sumatora (w pozostałych strukturach na rysunku 9.24 sumatory są $2*N$ -bitowe). Ponadto zauważmy, że w algorytmie 9.5 w wyniku dodawania części iloczynu $P[2*N-1:N]$ i mnożnej A może powstać przeniesienie. Dlatego też wektor $P[2*N:N]$ po lewej stronie znaku równości ma długość o jeden większą. Opis algorytmu 9.5 w języku Verilog przedstawia się następująco:

```

module algorithm_mult_d
    #(parameter N=4)
    (input [N-1:0] a,b,
    output [2*N-1:0] p);

    reg [N-1:0] ra,rb;
    reg [2*N:0] rp;
    integer i;

    always @(*)
    begin
        ra = a;
        rb = b;
        rp = 0;
        for (i=0; i<N; i=i+1)
        begin
            if (rb[0])    rp[2*N:N] = rp[2*N-1:N] + ra;
            else        rp = rp + {2*N{1'b0}};
            rb = rb >> 1;
            rp = rp >> 1;
        end
    end
    assign p = rp[2*N-1:0];
endmodule

```


Wynik syntezy modułu *algorithm_mult_d* na poziomie przesłań międzyrejestrów dokładnie odpowiada schematowi na rysunku 9.25.

9.6. Układ mnożący Bootha

Układ mnożący Bootha stanowi jedną z klasycznych metod przyspieszania operacji mnożenia. Metoda ta opiera się na następującym wyrażeniu:

$$2^m + 2^{m-1} + \dots + 2^k = 2^{m+1} - 2^k. \quad (9.7)$$

Poprawność równania (9.7) można sprawdzić, podstawiając zamiast m i k (gdzie $k < m$) dowolne liczby całkowite. Na przykład niech $m = 2$ i $k = 0$, wtedy lewa strona wyrażenia (9.7) będzie równa $2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7$, a prawa strona też przyjmuje taką samą wartość $2^3 - 2^0 = 8 - 1 = 7$.

Równanie (9.7) pozwala na zmniejszenie liczby generowanych iloczynów częściowych w przypadku, gdy mnożnik zawiera dłuższe sekwencje jedynek (o długości większej niż dwie jedynek). Zamiast sekwencyjnego dodawania do wartości P iloczynów częściowych odpowiadających kolejnym jedynkom sekwencji można wykonać tylko dwie operacje dodawania, które odpowiadają potęgom 2^{m+1} oraz 2^k .

W algorytmie Bootha analizowane są pary bitów (b_i, b_{i-1}) mnożnika, zaczynając od bitów najmłodszych, $i = 0, \dots, N-1$. Na początku nieistniejący bit b_{-1} przyjmuje wartość 0. Jeśli $(b_i, b_{i-1}) = 10$, oznacza to, że wykryty został początek sekwencji jedynek i do wyniku P dodawana jest wartość $(-A)$, która odpowiada składnikowi -2^k w wyrażeniu (9.7). Jeśli natomiast $(b_i, b_{i-1}) = 01$, oznacza to wykrycie końca sekwencji jedynek i do P dodawana jest wartość A , która (z uwzględnieniem wykonywanych przesunięć) odpowiada potędze 2^{m+1} w wyrażeniu (9.7). W przypadku, gdy (b_i, b_{i-1}) przyjmuje wartość 00 lub 11, nie jest wykonywana żadna operacja dodawania. Poniżej przedstawiony został formalny opis algorytmu Bootha.

Algorytm 9.6. Algorytm mnożenia Bootha

1. Przyjąć $i = 0, P = 0$.
2. Sprawdzić wartość pary bitów (b_i, b_{i-1}) .
3. Jeśli $(b_i, b_{i-1}) = 10$, wykonać operację $P := P + (-A)$.
4. Jeśli $(b_i, b_{i-1}) = 01$, wykonać operację $P := P + A$.
5. Jeśli $(b_i, b_{i-1}) = 00$ lub $(b_i, b_{i-1}) = 11$, wartość P się nie zmienia.
6. Wykonać instrukcje $i := i + 1, A := A \ll 1$.
7. Punkty 2 i 3 powtórzyć N razy.
8. Koniec.

Działanie algorytmu Bootha zademonstrujemy na przykładzie mnożenia liczb $4 \times 3 = 12$, gdzie mnożna $A = 4$, mnożnik $B = 3$, iloczyn $P = 12$. Niech operandy A i B mają taki sam rozmiar – 4 bity. Kolejne iteracje wykonania algorytmu Bootha zaprezentowane zostały w tabeli 9.9.

TABELA 9.9. Realizacja algorytmu Bootha przy obliczeniu iloczynu $4_{10} \times 3_{10} = 12_{10}$ lub $0100_2 \times 0011_2 = 00001100_2$

Iteracja	1	2	3	4
B	0011.0	0011.0	0011.0	0011.0
Działanie	$P := P + (-A)$	$P := P$	$P := P + A$	$P := P$
A	0100	01000	010000	0100000
P	00000000	11111100	11111100	00001100
Dodawana wartość	11111100		00010000	
Wynik	11111100	11111100	00001100	00001100

W tabeli 9.9 analizowane bity mnożnika B zostały pogrubione, a dodatkowy bit mnożnika b_{-1} oddzielony został od pozostałych kropką. Na pierwszej iteracji $(b_0, b_{-1}) = 10$, dlatego do P dodawana jest wartość $(-A)$, która w kodzie uzupełnień do dwóch reprezentowana jest w postaci 11111100. Na drugiej iteracji $(b_1, b_0) = 11$, więc operacja dodawania nie jest wykonywana. Na kolejnej iteracji $(b_2, b_1) = 01$, więc do P dodawana jest mnożna A przesunięta o dwa bity (przesunięcie A w lewo o jeden bit wykonywane jest pod koniec każdej iteracji). Na ostatniej iteracji $(b_3, b_2) = 00$, wobec tego iloczyn częściowy nie jest dodawany do P . Otrzymany wynik to $P = 00001100_2 = 12_{10}$.

Algorytm Bootha pozwala mnożyć liczby ze znakiem przedstawione w kodzie uzupełnień do dwóch, generując przy tym poprawny znak wyniku. Można się o tym przekonać na przykładzie mnożenia liczb 4 i -3 przedstawionym w tabeli 9.10.

TABELA 9.10. Wykonanie algorytmu Bootha przy obliczeniu $4_{10} \times (-3_{10}) = -12_{10}$ lub $0100_2 \times 1101_2 = 11110100_2$

Iteracje	1	2	3	4
B	1101.0	1101.0	1101.0	1101.0
Działanie	$P := P + (-A)$	$P := P + A$	$P := P + (-A)$	$P := P$
A	0100	01000	010000	0100000
P	00000000	11111100	00000100	11110100
Dodawana wartość	11111100	00001000	11110000	
Wynik	11111100	00000100	11110100	11110100

Proponujemy czytelnikowi samodzielne zastosowanie algorytmu Bootha do wykonania operacji mnożenia $(-4) \times 3 = -12$ i $(-4) \times (-3) = 12$.

Poniżej przedstawiony został kod w języku Verilog sparametryzowanego układu mnożącego, gdzie mnożna A oraz mnożnik B są jednakowej długości N (domyślnie $N = 4$).

```

module booth
    #(parameter N=4)
    (input signed [N-1:0] a,b,
    output signed [2*N-1:0] p);

    reg signed [2*N-1:0] ra, rp;
    reg signed [N:0] rb;

    integer i;

    always @(*)
    begin
        ra = a;
        rb = b <<< 1;
        rp = 0;

        for (i=0; i<N; i=i+1)
        begin
            case ({rb[i+1],rb[i]})
                2'b01: rp = rp + ra;
                2'b10: rp = rp - ra;
                default: ;
            endcase
            ra = (ra <<< 1);
        end
    end
    assign p = rp;

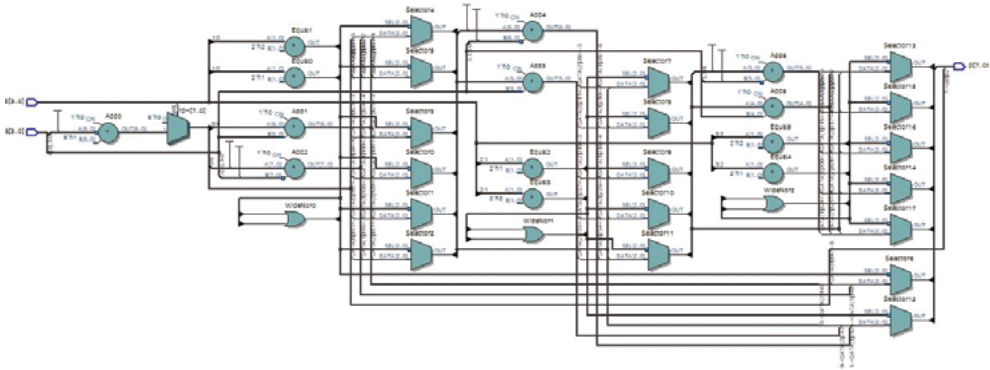
endmodule

```

Ponieważ algorytm Booth'a pozwala mnożyć liczby ze znakiem w kodzie uzupełnień do dwóch, operandy a i b , a także wynik p zdefiniowane są w programie jako liczby ze znakiem. W podanym kodzie zastosowany został algorytmiczny styl opisu projektu za pomocą instrukcji proceduralnych. Z tego powodu zmienne pomocnicze ra , rb i rp zadeklarowane zostały z wykorzystaniem typu **reg**. Rozmiar zmiennej rb zwiększony został o jeden bit, aby umożliwić przechowywanie dodatkowego bitu mnożnika b_{-1} . Przed wykonaniem pętli zainicjowane zostały zmienne pomocnicze. Zwróćmy uwagę na to, że zmienna rb inicjowana jest przesuniętą o jeden bit w lewo wartością mnożnika b . Dalsza część kodu dokładnie realizuje

algorytm 9.6. Warto też zwrócić uwagę na zastosowanie w programie pustej konstrukcji **default**.

Wyniki syntezy i symulacji projektu *booth* pokazane są na rysunkach 9.29 i 9.30.



RYS. 9.29. Wynik syntezy układu mnożącego Booth'a na poziomie przesłań międzyrejestrowych

Name	Value at	0 ps	40.0 ns	80.0 ns	120.0 ns	160.0 ns
a	5.0	0	0	0	0	0
b	5.-1	0	0	0	0	0
p	5.0	0	0	0	0	0

RYS. 9.30. Wyniki modelowania układu mnożącego Booth'a

Analiza rysunku 9.29 pokazuje, że algorytmiczna realizacja układu mnożącego Booth'a wymaga dość dużo zasobów logicznych. Czytelnikowi proponujemy wykonanie strukturalnego opisu w języku Verilog układu mnożącego Booth'a 4 × 4 bity.

9.7. Zmodyfikowany układ mnożący Booth'a

Algorytm Booth'a ma szereg wad, które utrudniają jego implementację sprzętową. Należą do nich: konieczność wykonania N iteracji, aby sprawdzić wartości par bitów (b_i, b_{i-1}) , wykorzystanie dwóch reprezentacji iloczynu częściowego A i $-A$ oraz spowolnienie działania operacji w przypadku wystąpienia w mnożniku zer i jedynek na zmianę. W praktyce często stosuje się zmodyfikowany algorytm Booth'a do sprzętowej realizacji operacji mnożenia, który piera się na następującej zależności:

$$b_i \cdot 2^i = b_i \cdot 2^{i+1} - 2b_i \cdot 2^{i-1}. \quad (9.8)$$

Poprawność wyrażenia (9.8) łatwo jest sprawdzić, wykonując przekształcenie prawej części równania.

Niech

$$P = A \cdot B = A \cdot (b_{N-1} \cdot 2^{N-1} + \dots + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0).$$

Zastosujemy równanie (9.8) do składników o parzystej potęgze dwójki:

$$P = A \cdot (b_{N-1} \cdot 2^{N-1} + \dots + b_4 \cdot 2^5 - 2b_4 \cdot 2^3 + b_3 \cdot 2^3 + b_2 \cdot 2^3 - \\ - 2b_2 \cdot 2^1 + b_1 \cdot 2^1 + b_0 \cdot 2^1 - 2b_0 \cdot 2^{-1}).$$

Teraz mnożnik jest zapisany za pomocą składników z nieparzystymi potęgami dwójki. Przekształcimy powyższy wzór w następujący sposób:

$$P = A \cdot (b_{N-1} \cdot 2^{N-1} + \dots + b_4 \cdot 2^5 + (-2b_4 \cdot 2^3 + b_3 \cdot 2^3 + b_2 \cdot 2^3) + \\ + (-2b_2 \cdot 2^1 + b_1 \cdot 2^1 + b_0 \cdot 2^1) + \\ + (-2b_0 \cdot 2^{-1} + b_{-1} \cdot 2^{-1} + b_{-2} \cdot 2^{-1})).$$

W celu zapewnienia regularności zapisu w ostatnim nawiasie dodane zostały składniki $b_{-1} \cdot 2^{-1}$ i $b_{-2} \cdot 2^{-1}$. Nie zmieniają one wartości wyrażenia, ponieważ $b_{-1} = b_{-2} = 0$.

Wprowadzimy oznaczenia dla iloczynów częściowych:

$$R_i = A \cdot (-2b_{i+1} + b_i + b_{i-1}) \cdot 2^i. \quad (9.9)$$

Z uwzględnieniem oznaczeń (9.9) operację mnożenia można opisać następująco:

$$P = A \cdot B = R_{N-1} + R_{N-3} + \dots + R_3 + R_1 + R_{-1}. \quad (9.10)$$

Zauważmy, że w wyrażeniu (9.10) występuje dwukrotnie mniej iloczynów częściowych. Jeśli mnożnik ma nieparzystą liczbę bitów, należy zastosować równanie (9.8) do nieparzystych potęg dwójki.

W zmodyfikowanym algorytmie mnożenia Booth'a iloczyny częściowe są określane wyrażeniami (9.9). Podczas implementacji tego algorytmu w każdej iteracji należy analizować trójki bitów mnożnika (b_{i+1}, b_i, b_{i-1}), a następnie – w zależności od ich wartości – generować odpowiednie iloczyny częściowe oraz wykonywać instrukcje opisane w tabeli 9.11.

TABELA 9.11. Instrukcje wykonywane w zmodyfikowanym algorytmie Booth'a

b_{i+1}	b_i	b_{i-1}	Wykonywana operacja
0	0	0	$P := P$
0	0	1	$P := P + A$
0	1	0	$P := P + A$
0	1	1	$P := P + 2A$
1	0	0	$P := P - 2A$
1	0	1	$P := P - A$
1	1	0	$P := P - A$
1	1	1	$P := P$

Biorąc pod uwagę powyższe rozważania, zmodyfikowany algorytm Booth'a może zostać opisany w następujący sposób:

Algorytm 9.7. Zmodyfikowany algorytm Bootha

1. Przyjąć $i = 0, P = 0$.
2. Sprawdzić wartość trójki bitów (b_{i+1}, b_i, b_{i-1}) . W zależności od tej wartości wykonać odpowiednią operację podaną w tabeli 9.11.
3. Wykonać instrukcje $i := i + 2, A := A \ll 2$.
4. Punkty 2 i 3 wykonywać dopóki $i < N$.
5. Koniec.

Działanie zmodyfikowanego algorytmu Bootha wyjaśnimy na przykładzie mnożenia liczb $4 \times 6 = 24$. Kolejne wykonywane instrukcje oraz ich wyniki przedstawione zostały w tabeli 9.12.

TABELA 9.12. Działanie zmodyfikowanego algorytmu Bootha przy obliczeniu $4_{10} \times 6_{10} = 24_{10}$ lub $0100_2 \times 0110_2 = 11000_2$

Iteracje	1	2
B	0110.0	0110.0
Operacja	$P := P - 2A$	$P := P + 2A$
A	0100	010000
P	00000000	11111000
Dodawana wartość	11111000	00100000
Wynik	11111000	00011000

Z tabeli 9.12 wynika, że operacja mnożenia liczb zrealizowana została w dwóch iteracjach.

Kod w języku Verilog zmodyfikowanego układu mnożącego Bootha można zapisać następująco:

```
module modified_booth
#(parameter N=4)
(input signed [N-1:0] a,b,
output signed [2*N-1:0] p);

reg signed [2*N-1:0] ra, rp;
reg signed [N:0] rb;

integer i;

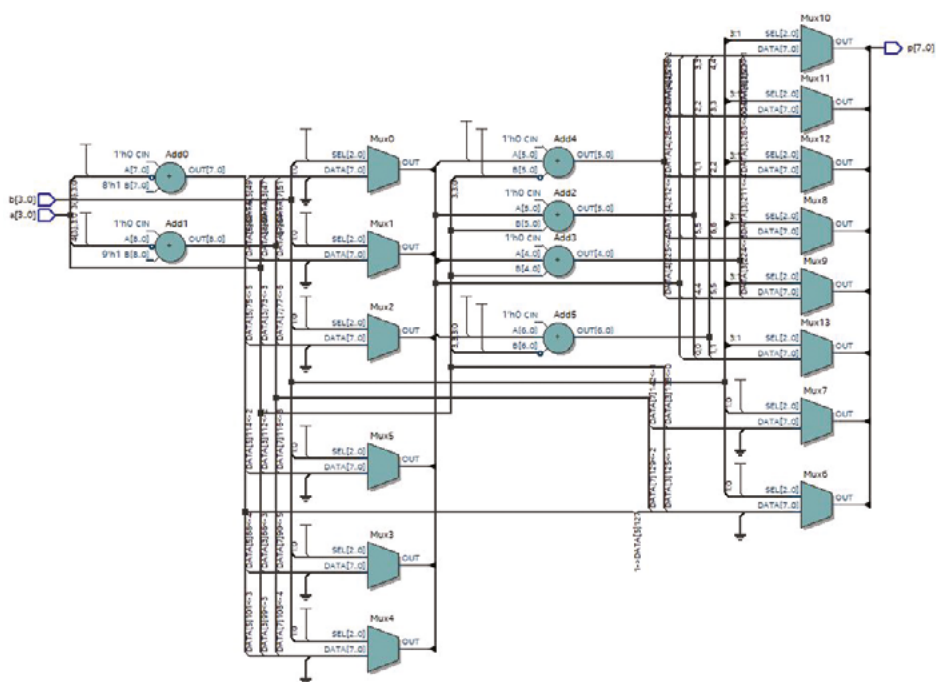
always @(*)
begin
    ra = a;
    rb = b <<< 1;
    rp = 0;
```

```

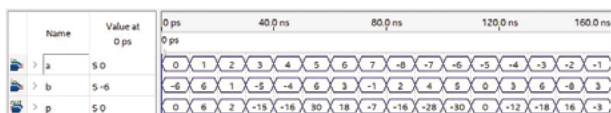
for (i=0; i<N; i=i+2)
begin
    case ({rb[i+2],rb[i+1],rb[i]})
        3'b001: rp = rp + ra;
        3'b010: rp = rp + ra;
        3'b011: rp = rp + (ra <<< 1);
        3'b100: rp = rp - (ra <<< 1);
        3'b101: rp = rp - ra;
        3'b110: rp = rp - ra;
        default: ;
    endcase
    ra = (ra <<< 2);
end
end
assign p = rp;
endmodule

```

Sugerujemy czytelnikowi dokonanie porównania przedstawionego kodu z omawianym wcześniej kodem standardowego układu mnożącego Booth'a oraz wyjaśnienie istotnych różnic między nimi. Wyniki syntezy i symulacji projektu *modified_booth* zamieszczono na rysunkach 9.31 i 9.32.



RYS. 9.31. Wynik syntezy zmodyfikowanego układu mnożącego Booth'a



RYS. 9.32. Wyniki symulacji zmodyfikowanego układu mnożącego Bootha

Zauważalne jest, że w porównaniu ze standardowym układem mnożącym Booth'a złożoność schematu zmodyfikowanego układu mnożącego Booth'a jest mniejsza. Poprosimy czytelnika o wyjaśnienie, dlaczego tak się stało.

9.8. Badania realizacji układów mnożących na różnych rodzinach układów FPGA

Efektywność realizacji operacji mnożenia odgrywa istotną rolę przy projektowaniu systemów wbudowanych, zwłaszcza w takich zastosowaniach jak cyfrowe przetwarzanie sygnałów oraz kryptografia. W celu odpowiedzi na pytanie, która z omówionych metod mnożenia jest najlepsza do zastosowania na układach FPGA, przeprowadzono badania efektywności różnych układów mnożących. Wszystkie wcześniej omawiane projekty układów mnożących 4×4 bity zostały zaimplementowane w środowisku Quartus Prime wersji 18.1 na różnych rodzinach układów FPGA, aby zbadać ich efektywność pod względem szybkości działania i złożoności realizacji. Aby móc ocenić szybkość działania, na wejściach i wyjściach układów mnożących zastosowano rejestry, a szybkość wyznaczono poprzez maksymalną częstotliwość F (wyróżoną w MHz) sygnału synchronizacji sterującego przełączeniem rejestrów. Koszt realizacji układów mnożących został określony liczbą L wykorzystanych elementów logicznych FPGA. Wyniki badań przedstawione zostały w tabeli 9.13, w której przyjęto następujące oznaczenia:

- CSAM – matrycowy układ mnożący z przechowywaniem przeniesień (*Carry-Save Array Multiplier*);
- MWA – układ mnożący z wykorzystaniem sumatorów z przeniesieniem szeregowym (*Multiplier With Adders*);
- Wallace – układ mnożący Wallace’a;
- Dadda – układ mnożący Dadda;
- AM_a, AM_b, AM_c i AM_d – algorytmiczny opis układów mnożących (*Algorithm Multiplier*) ze strukturami przedstawionymi na rysunku 9.24;
- Booth – układ mnożący Booth’a;
- MBooth – zmodyfikowany układ mnożący Booth’a;
- Quartus – metoda systemu Quartus, która realizuje operację arytmetyczną mnożenia (*);
- Best – najlepsze rozwiązanie dla danej rodziny układów: $L = \min$, $F = \max$.

TABELA 9.13. Wyniki badań projektów układów mnożących 4 x 4 bity

Układ mnożący	MAX II		MAX V		MAX 10		Arria II		Cyclone IV		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
CSAM	32	111	32	66	34	145	24	277	35	159	15	178	34	161
MWA	31	103	31	67	33	117	16	180	32	134	9	164	32	164
Wallace	32	104	32	61	34	104	25	224	32	137	16	185	32	143
Dadda	35	120	35	60	36	135	28	247	37	164	13	154	36	174
AM_a	32	96	32	51	34	122	28	177	37	154	15	100	34	160
AM_b	36	103	36	55	38	126	37	185	36	142	19	118	35	154
AM_c	36	103	36	55	38	126	37	185	36	141	19	118	35	154
AM_d	32	96	32	51	34	122	28	177	37	154	15	100	34	160
Booth	59	67	59	42	63	82	48	181	62	111	26	103	64	106
MBooth	52	98	52	50	55	127	41	190	53	130	25	115	56	141
Quartus	37	113	37	60	32	117	18	270	32	171	10	170	32	117
Best	31	120	31	67	32	145	16	277	32	171	9	185	32	174

Analiza tabeli 9.13 wskazuje, że lepszym według złożoności realizacji okazał się układ mnożący z wykorzystaniem sumatorów z przeniesieniem szeregowym MWA. Ponadto układ mnożący Wallace'a także pokazuje najlepsze wyniki dla układów rodzin Cyclone IV i Cyclone 10.

Według szybkości działania układ mnożący Dadda jest najlepszy dla rodzin MAX II i Cyclone 10. Matrycowy układ mnożący z przechowywaniem przeniesień CSAM pokazuje najlepsze wyniki dla rodzin MAX 10 i Arria II. Układ z wykorzystaniem sumatorów z przeniesieniem szeregowym MWA jest najszybszy dla rodziny MAX V. Natomiast układ mnożący Wallace'a uzyskał najlepszy wynik dla rodziny Cyclone V.

Metoda syntezy systemu Quartus okazała się lepszą pod względem złożoności realizacji dla rodziny układów MAX 10, a także pod względem szybkości działania dla rodziny Cyclone IV.

Ogólnie matrycowe układy mnożące zademonstrowały lepsze wyniki zarówno pod względem złożoności, jak i szybkości działania w porównaniu z metodami algorytmicznymi.

Porównanie najlepszych wyników przedstawionych układów mnożących z metodą systemu Quartus pokazuje, że pod względem kosztów realizacji metoda systemu Quartus prezentuje nieco lepsze rozwiązanie tylko dla rodziny MAX 10, natomiast pod względem szybkości działania jedynie dla rodziny Cyclone IV. Z tego wnioskujemy, że metoda systemu Quartus nie jest najlepszym wyborem dla większości rodzin FPGA w przypadku realizacji układów mnożących.

9.9. Podsumowanie rozdziału

Układy mnożące są powszechnie wykorzystywane w systemach wbudowanych, szczególnie do realizacji funkcji *cyfrowego przetwarzania sygnałów*. Istnieją dwa podstawowe rodzaje układów mnożących: *szeregowe (serial)* oraz *równoległe (parallel)*. Te pierwsze są układami synchronicznymi, które cyklicznie generują iloczyny częściowe i dodają je do wcześniej obliczonego wyniku. Natomiast równoległe układy mnożące są zazwyczaj realizowane jako układy kombinacyjne, gdzie wartości dwóch liczb są podawane na wejścia, a wyjścia generują wartość iloczynu.

Złożoność realizacji równoległych układów mnożących jest wysoka i rośnie znacznie wraz ze zwiększaniem rozmiaru bitowego operandów. Ponadto zwiększenie rozmiaru operandów prowadzi do spadku szybkości działania równoległych układów mnożących.

Aby przyspieszyć cykl wykonania operacji mnożenia i zwiększyć wydajność całego systemu, często równoległe układy mnożące są projektowane w architekturze *potokowej (pipelined)*. W tym celu do schematu takiego układu dodaje się rejestry, które służą do przechowywania wyników pośrednich.

Najprostszym układem mnożącym jest *matrycowy układ mnożący z przechowywaniem przeniesień (Carry-Save Array Multiplier – CSAM)*, który realizuje klasyczny algorytm mnożenia pisemnego.

Jedną z odmian matrycowego układu mnożącego z przechowywaniem przeniesień jest układ mnożący z wykorzystaniem sumatorów z przeniesieniem szeregowym.

Głównym celem metody *drzewiastego układu mnożącego Wallace’a (Wallace Tree Multiplier)* jest przyspieszenie operacji mnożenia poprzez takie przypisanie sygnałów wejściowych do sumatorów, które prowadzi do zmniejszenia liczby ich poziomów w każdej kolumnie matrycowego układu mnożącego.

Metoda syntezy układów mnożących *Dadda* podobna jest do metody Wallace’a, jednak skoncentrowana jest na minimalizacji liczby wykorzystywanych sumatorów i półsumatorów.

W celu zwiększenia szybkości działania w obu układach, Wallace’a i Dadda, szeregowo sumatory na ostatnim poziomie można zastąpić równoległymi sumatorami.

Znane są cztery podstawowe algorytmiczne metody realizacji układów mnożących: a) z przesunięciem mnożnika w prawo, mnożnej w lewo, analizowany jest młodszy bit mnożnika; b) z przesunięciem mnożnika w lewo, mnożnej w prawo, analizowany jest starszy bit mnożnika; c) z przesunięciem mnożnika w lewo, akumulatora w lewo, analizowany jest starszy bit mnożnika; d) z przesunięciem mnożnika w prawo, akumulatora w prawo, analizowany jest młodszy bit mnożnika.

W układzie mnożącym Bootha analizowane są pary bitów mnożnika, co pozwala na redukcję liczby iloczynów częściowych, szczególnie gdy mnożnik zawiera długie sekwencje jedynek (o długości większej niż dwie jedynek). Algorytm Bootha dla liczb ze znakiem w kodzie uzupełnień do dwóch automatycznie uwzględnia znaki operandów i generuje poprawny znak wyniku.

W zmodyfikowanym algorytmie Bootha na każdej iteracji analizowane są trzy bity mnożnika, co prowadzi do dwukrotnego zmniejszenia liczby generowanych iloczynów częściowych.

Przy projektowaniu układów mnożących o najmniejszych kosztach realizacji najlepszego rozwiązania należy poszukiwać między układem wykorzystującym sumatory szeregowo MWA a układem mnożącym Wallace'a. Natomiast w kontekście wyboru najszybszego układu należy rozważyć układy mnożące Dadda, CSAM, MWA oraz Wallace'a.

Wszystkie metody algorytmiczne pokazują znacznie gorsze wyniki w porównaniu z matrycowymi układami mnożącymi, zarówno pod względem kosztów realizacji, jak i szybkości działania. Szczególnie wysoką złożoność realizacji wykazują układy mnożące Bootha oraz zmodyfikowane układy mnożące Bootha. Z tego powodu zaleca się projektowanie tych układów ręcznie i opisywanie ich w formie strukturalnej w języku Verilog.

Zauważmy również, że matrycowe układy mnożące mogą zostać zmodyfikowane w celu zwiększenia szybkości ich działania poprzez zastąpienie sumatorów z przeniesieniem szeregowym sumatorami w przeniesieniu równoległym.

W przypadku, gdy znaczący wzrost szybkości działania nie jest wymagany ani nie ma potrzeby znacznego obniżenia kosztów realizacji, do implementacji operacji mnożenia może zostać wykorzystana metoda systemu Quartus, czyli zwykła instrukcja mnożenia (*) języka Verilog.

10. Projektowanie automatów skończonych

Poza układami kombinacyjnymi istotną rolę w systemach cyfrowych odgrywają także układy wyposażone w elementy pamięci, takie jak rejestry lub przerzutniki. Te rodzaje układów są nazywane *układami sekwencyjnymi*. Ich modelem matematycznym jest *automat skończony* (*finite state machine – FSM*). Ogólnie rzecz biorąc, każdy system cyfrowy można postrzegać jako automat skończony z bardzo dużą liczbą stanów wewnętrznych. W systemach wbudowanych automaty skończone zazwyczaj pełnią funkcję układów sterowania (kontrolerów) dla systemów, podsystemów, bloków lub schematów. Podczas projektowania systemów wbudowanych standardowe bloki funkcjonalne często się powtarzają, ale układy sterowania są zazwyczaj unikatowe i wymagają indywidualnego projektowania za każdym razem. Dlatego też dużo czasu poświęca się kwestiom projektowania automatów skończonych.

W niniejszym rozdziale zostaną omówione zagadnienia związane z projektowaniem automatów skończonych w kontekście ich implementacji w układach programalnych FPGA.

10.1. Modele strukturalne automatów skończonych przy ich realizacji na układach FPGA

W praktyce najbardziej rozpowszechnione są dwa rodzaje automatów skończonych: *automat Mealy’ego* oraz *automat Moore’a*. Zachowanie automatu Mealy’ego można opisać za pomocą następujących równań:

$$a_{t+1} = \varphi(z_t, a_t); \quad (10.1)$$

$$w_t = \psi(z_t, a_t);$$

gdzie: φ – funkcja przejść; ψ – funkcja wyjść; z_t – symbol wejściowy w momencie czasu automatu t ($t = 1, 2, 3, \dots$); w_t – generowany symbol wyjściowy; a_t – aktualny stan automatu (*stan automatu*); a_{t+1} – następny stan automatu (*stan przejścia*).

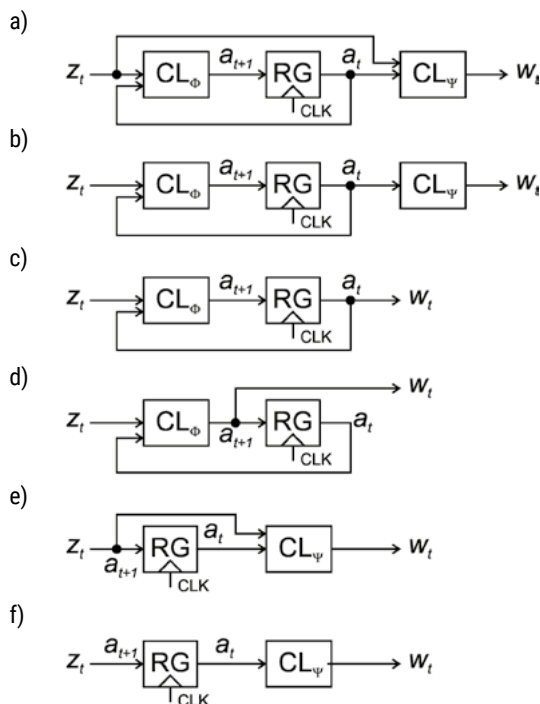
Funkcjonowanie automatu Moore'a można przedstawić w następujący sposób:

$$a_{t+1} = \varphi(z_t, a_t); \quad (10.2)$$

$$w_t = \psi(a_t).$$

Cechą charakterystyczną automatu Mealy'ego jest to, że wyjście automatu w_t zależy zarówno od stanu wejścia z_t , jak i od stanu wewnętrznego a_t . Natomiast w automacie Moore'a generowany symbol wyjściowy w_t zależy wyłącznie od stanu automatu a_t .

W strukturze automatu skończonego *funkcje przejść* φ i *wyjść* ψ realizowane są za pomocą układów kombinacyjnych CL_φ i CL_ψ (rysunek 10.1), z kolei *pamięć automatu* przedstawiona jest w postaci rejestru RG, który w każdym momencie czasu automatu t przechowuje kod wewnętrznego stanu a_t . Struktury automatów Mealy'ego i Moore'a pokazane są odpowiednio na rysunkach 10.1a i 10.1b.



RYS. 10.1. Struktury automatów skończonych: a) – automat klasy A (Mealy'ego); b) – automat klasy B (Moore'a); c) – automat klasy C (Moore'a); d) – automat klasy D (Mealy'ego); e) – automat klasy E (Mealy'ego); f) – automat klasy F (Moore'a)

Cechą charakterystyczną struktury automatu Moore'a jest brak bezpośredniego połączenia układu kombinacyjnego CL_ψ z wejściami automatu. W związku z tym w automacie Moore'a zmiana wartości sygnałów wejściowych w dowolnych

momentach czasu nie powoduje zmiany wartości sygnałów na wyjściach. Wyjścia automatu Moore'a zostaną zmienione dopiero po ustawieniu nowego stanu w rejestrze RG, który jest sterowany sygnałem synchronizacji CLK.

W strukturze automatu Mealy'ego układ kombinacyjny CL_ψ jest bezpośrednio połączony z wejściami automatu. W rezultacie zmiana sygnałów wejściowych automatu Mealy'ego spowoduje także zmianę sygnałów na wyjściach (przy tych samych wartościach zmiennych wewnętrznych). W związku z tym można stwierdzić, że schemat automatu Mealy'ego jest *asynchroniczny* względem wejść automatu.

Struktura automatu Mealy'ego (rysunek 10.1a) nazywana jest *automatem klasy A*, natomiast struktura automatu Moore'a (rysunek 10.1b) – *automatem klasy B* [6].

Jeżeli każdy zbiór sygnałów wyjściowych w_t automatu Moore'a będzie odpowiadać kodowi jego stanu wewnętrznego a_t , to jego zachowanie można opisać w następujący sposób:

$$a_{t+1} = \varphi(z_t, a_t); \quad (10.3)$$

$$w_t = a_t.$$

Ten rodzaj automatu jest nazywany *automatem klasy C*. Struktura automatu Moore'a klasy C została przedstawiona na rysunku 10.1c. Jej charakterystyczną cechą jest brak układu kombinacyjnego CL_ψ , a także fakt, że wszystkie wyjścia automatu są wyjściami rejestrowymi, ponieważ są one generowane na wyjściach rejestru RG. Należy zauważyć, że taka struktura jest łatwa do zrealizowania na układach FPGA, gdyż wszystkie układy FPGA pozwalają na konfigurowanie wyjść elementów logicznych jako wyjść rejestrowych.

W przypadku, gdy każdy zbiór sygnałów wyjściowych w_t automatu Mealy'ego odpowiada kodowi jego stanu przejścia a_{t+1} , otrzymujemy wtedy *automat klasy D* [6], którego funkcjonowanie opisywane jest następującymi równaniami:

$$a_{t+1} = \varphi(z_t, a_t); \quad (10.4)$$

$$w_t = a_{t+1}.$$

Struktura automatu Mealy'ego klasy D przedstawiona została na rysunku 10.1d. Jej cechą charakterystyczną jest także brak układu kombinacyjnego CL_ψ . Jednakże w odróżnieniu od automatu klasy C wyjścia automatu są typu kombinacyjnego, ponieważ są generowane na wyjściach układu kombinacyjnego CL_ϕ . Automat klasy D też jest łatwo implementowany na układach FPGA, gdyż wyjścia elementów logicznych mogą być również konfigurowane jako wyjścia kombinacyjne.

Podstawową różnicą pomiędzy automatami klas C i D jest to, że w automacie klasy C zbiór sygnałów wyjściowych w_t pokrywa się z kodem *aktualnego stanu automatu* a_t , natomiast w automacie klasy D zbiór sygnałów wyjściowych w_t odpowiada kodowi *stanu przejścia* a_{t+1} .

Zalety automatów klas C i D, w porównaniu z automatami klas A i B, są następujące:

- prostsza logika kombinacyjna automatu, ponieważ nie trzeba realizować układu kombinacyjnego CL_{ψ} ;
- mniejsza liczba wykorzystywanych elementów logicznych układu FPGA, gdyż rejestr pamięci automatu oraz funkcje wyjściowe realizowane są na tych samych elementach logicznych.

Kody stanów wewnętrznych automatu skończonego mogą odpowiadać nie tylko zbiorom zmiennych wyjściowych, lecz także zbiorom wartości zmiennych wejściowych automatu [6]. Jeśli każdy zbiór wejściowy z_t automatu Mealy'ego pokrywa się z kodem stanu przejścia a_{t+1} , wówczas otrzymujemy *automat klasy E*, którego funkcjonowanie można opisać następującymi równaniami:

$$\begin{aligned} a_{t+1} &= z_t; \\ w_t &= \psi(z_t, a_t). \end{aligned} \quad (10.5)$$

W analogiczny sposób jeśli każdy zbiór sygnałów wejściowych z_t automatu Moore'a pokrywa się z kodem stanu przejścia a_{t+1} , otrzymujemy *automat klasy F*, którego zachowanie można opisać następująco:

$$\begin{aligned} a_{t+1} &= z_t; \\ w_t &= \psi(a_t). \end{aligned} \quad (10.6)$$

Cechą charakterystyczną schematów automatów klas E (rysunek 10.1e) i F (rysunek 10.1f) jest brak układu kombinacyjnego CL_{ϕ} . W automacie klasy E układ kombinacyjny CL_{ψ} ma dwa rodzaje wejść: kombinacyjne oraz rejestrowe, natomiast w automacie klasy F – tylko wejścia rejestrowe.

Warunkiem koniecznym do realizacji automatów klas E i F na układach FPGA jest możliwość buforowania sygnałów wejściowych za pomocą rejestrów. Ponadto w przypadku automatu klasy E bufory wejściowe powinny obsługiwać dwa rodzaje połączeń z logiką wewnętrzną FPGA: kombinacyjne i rejestrowe. Warto zaznaczyć, że większość współczesnych układów FPGA ma takie funkcjonalności.

Zalety automatów klas E i F, w porównaniu z automatami klas A i B, są następujące:

- prostsza logika kombinacyjna automatu, ponieważ nie jest realizowany układ kombinacyjny CL_{ϕ} ;
- mniejsza liczba wykorzystywanych elementów logicznych układu, gdyż pamięć automatu realizowana jest za pomocą buforów wejściowych FPGA.

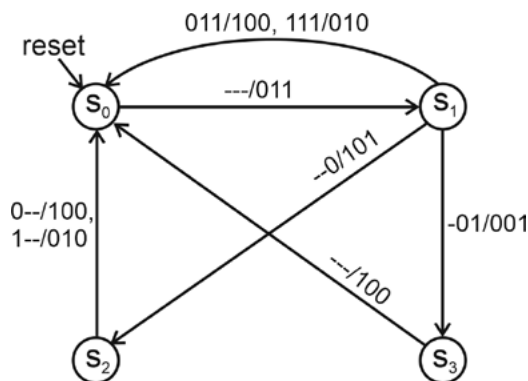
Istotną różnicą pomiędzy automatami Moore'a klas C i F jest to, że w automacie klasy C wejścia zawsze są rejestrowe, a w automacie klasy F są one zawsze kombinacyjne.

W praktyce rzadko udaje się bezpośrednio realizować automaty klas C–F. Można to wyjaśnić tym, że nie zawsze zbiory sygnałów wejściowych lub wyjściowych pozwalają na pełne opisanie wszystkich kodów stanów wewnętrznych automatu skończonego. Dlatego podczas projektowania automatów klas C–F konieczne jest wprowadzenie dodatkowych elementów pamięci oraz sprzężeń zwrotnych, co wymaga wykorzystania dodatkowych elementów logicznych w układach FPGA. W takich przypadkach tworzone są połączone modele automatów skończonych. Szczegółowe badania na ich temat można znaleźć w [12].

10.2. Sposoby opisu automatów skończonych

Automaty skończone zazwyczaj są opisywane za pomocą *grafu automatu* lub *tablicy przejść*. Grafy automatów dzięki swojej wysokiej czytelności i klarowności często stanowią preferowany sposób przedstawiania funkcjonowania automatów skończonych. Jednakże w miarę wzrostu liczby stanów i przejść informacje zawarte na grafie mogą stać się mniej czytelne, dlatego złożone automaty skończone częściej są reprezentowane w postaci tablic przejść. Warto zwrócić uwagę na różnice występujące w opisach automatów Mealy’ego oraz Moore’a.

Na rysunku 10.2 zaprezentowany został graf automatu Mealy’ego.



RYŚ. 10.2. Przykładowy graf automatu Mealy’ego

Wierzchołki grafu na rysunku 10.2 reprezentują stany, krawędzie zaś odpowiadają przejściom pomiędzy stanami automatu skończonego. W analizowanym przykładzie graf zawiera cztery wierzchołki, które odnoszą się do stanów $s_0, \dots, s_3$, trzy zmienne wejściowe $x_0, \dots, x_2$ oraz trzy zmienne wyjściowe $y_0, \dots, y_2$. Każdej krawędzi grafu automatu przypisany jest zbiór wartości zmiennych wejściowych (*wektor wejściowy*), który inicjuje dane przejście automatu skończonego. W grafie automatu Mealy’ego, poza zbiorem sygnałów wejściowych, dodatkowo zapisywany jest zbiór

sygnałów wyjściowych (wektor wyjściowy) generowanych przez automat w danym przejściu. Stan początkowy automatu skończonego jest zaznaczany strzałką z napisem *reset*. W analizowanym przykładzie stanem początkowym jest stan s_0 .

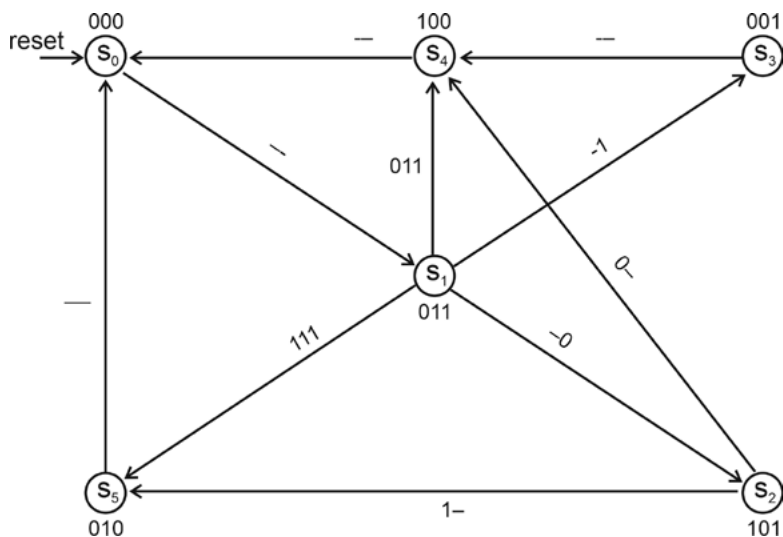
Tablica przejść automatu skończonego zawiera cztery kolumny i reprezentuje listę przejść automatu. Każdy wiersz tablicy przejść opisuje jedno przejście automatu. Kolumny tablicy przejść mają następujące oznaczenia: a_m – początkowy stan przejścia (*present state*); $X(a_m, a_s)$ – wektor zmiennych wejściowych (lub funkcja logiczna), który inicjuje dane przejście; $Y(a_m, a_s)$ – wektor wyjściowy generowany przez automat na danym przejściu; a_s – końcowy stan przejścia (*next state*). Kreska (-) w wektorze wejściowym oznacza, że odpowiednia zmienna wejściowa nie wpływa na dane przejście, czyli może przyjmować dowolną wartość. Tablica przejść automatu Mealy’ego dla naszego przykładu przedstawiona została w tabeli 10.1.

TABELA 10.1. Tablica przejść automatu Mealy’ego

a_m	$X(a_m, a_s)$	$Y(a_m, a_s)$	a_s
s_0	---	011	s_1
s_1	--0	101	s_2
s_1	-01	001	s_3
s_1	011	100	s_0
s_1	111	010	s_0
s_2	0--	100	s_0
s_2	1--	010	s_0
s_3	---	100	s_0

Zauważmy, że przedstawiona tablica przejść opisuje automat w formie listy przejść i faktycznie różni się od klasycznej tablicy przejść automatu skończonego, z jaką możemy się spotkać w literaturze dotyczącej teorii automatów.

Graf automatu Moore’a, który jest równoważny wcześniej opisanemu automatu Mealy’ego, czyli reprezentujący ten sam algorytm sterowania, został przedstawiony na rysunku 10.3, a jego tablicę przejść pokazano w tabeli 10.2. Zauważmy, że liczba stanów automatu Moore’a zwiększyła się z 4 do 6 w porównaniu z automatem Mealy’ego. Z teorii projektowania układów cyfrowych dobrze znane są metody konwersji automatu Mealy’ego na automat Moore’a i odwrotnie. Zachęcamy czytelnika do zastosowania jednej z tych metod w celu przekształcenia automatu Mealy’ego, przedstawionego na rysunku 10.2, na równoważny automat Moore’a i porównania wyniku przekształcenia z rysunkiem 10.3.



RYS. 10.3. Przykład grafu automatu Moore'a

TABELA 10.2. Tablica przejść automatu Moore'a

a_m	$X(a_m, a_s)$	$Y(a_m)$	a_s
s_0	---	000	s_1
s_1	--0	011	s_2
s_1	-01	011	s_3
s_1	011	011	s_4
s_1	111	011	s_5
s_2	0--	101	s_4
s_2	1--	101	s_5
s_3	---	001	s_4
s_4	---	100	s_0
s_5	---	010	s_0

Krawędziom grafu automatu Moore'a (rysunek 10.3) przypisane są jedynie wektory wejściowe, które inicjują przejścia, natomiast wartości zmiennych wyjściowych podane są przy wierzchołkach grafu. Oznacza to, że zbiór sygnałów wyjściowych jest formowany tylko w momencie przejścia automatu do danego stanu, czyli zmienne wyjściowe nie zależą od zmiennych wejściowych, lecz wyłącznie od stanu automatu. Z tego też powodu w tablicy przejść automatu Moore'a (tabela 10.2) wszystkim przejściom z tego samego stanu przypisywany jest ten sam wektor wyjściowy, a odpowiednia kolumna oznaczona jest $Y(a_m)$.

10.3. Projektowanie automatu Mealy'ego

Przedstawimy proces projektowania automatu Mealy'ego z naszego przykładu w systemie Quartus z wykorzystaniem języka Verilog.

10.3.1. Opis automatu Mealy'ego

W języku Verilog nie ma specjalnych konstrukcji przeznaczonych do opisu automatów skończonych. Automaty skończone są rozpoznawane w kodzie projektu automatycznie przez narzędzia syntezy systemu Quartus. Poniżej przedstawimy jeden z możliwych sposobów opisu automatu Mealy'ego z naszego przykładu przedstawionego na rysunku 10.2:

```
module Mealy_style_1(
    input clk, reset,           // sygnały synchronizacji i zerowania
    input [2:0] x,             // zmienne wejściowe
    output reg [2:0] y);       // zmienne wyjściowe

    (* keep *) reg [1:0] state, // stany automatu
    next;           // stan przejścia

    localparam [1:0]           // deklaracja stanów wewnętrznych
        s0=2'b00,              // (stan początkowy)
        s1=2'b01,              // z podaniem ich kodów
        s2=2'b10,
        s3=2'b11;

    always @(posedge clk, negedge reset) // opis pamięci automatu
        if (~reset) state <= s0;
        else state <= next;

    always @(*) // opis funkcji przejść
        case (state)
            s0: next = s1;
            s1: if (~x[0]) next = s2;
                else if (~x[1] && x[0]) next = s3;
                else if (~x[2] && x[1] && x[0]) next = s0;
                else next = s0;
            s2: if (~x[2]) next = s0;
                else next = s0;
            s3: next = s0;
        endcase
```

```

always @(*)                                // opis funkcji wyjść
case (state)
    s0:  y = 3'b011;
    s1:  if (~x[0])                                y = 3'b101;
        else if (~x[1] && x[0])                y = 3'b001;
        else if (~x[2] && x[1] && x[0])        y = 3'b100;
        else                                    y = 3'b010;
    s2:  if (~x[2])                y = 3'b100;
        else                        y = 3'b010;
    s3:  y = 3'b100;
endcase

endmodule

```

Niezbędnymi sygnałami sterującymi automatem skończonym są sygnał synchronizacji (w naszym projekcie *clk*) sterujący przełączeniem pamięci automatu oraz sygnał zerowania (*reset*), który przywraca automat do stanu początkowego. Opis funkcji przejść i wyjść automatu może być przedstawiony w różny sposób. W naszym przypadku automat ma trzy wejścia *x*[2:0] oraz trzy wyjścia *y*[2:0].

Do operowania na stanach wewnętrznych automatu zostały zadeklarowane dwie zmienne pomocnicze *state* i *next*, gdzie *state* odpowiada aktualnemu stanowi automatu skończonego (*present state*), natomiast *next* – stanowi przejścia (*next state*). Rozmiar bitowy kodów stanów *state* i *next* powinien być wystarczający do zakodowania wszystkich stanów wewnętrznych automatu. W naszym przykładzie automat ma cztery stany, do których zakodowania wystarczą dwa bity. Atrybut (* *keep* *) podany został przy deklaracji zmiennych *state* i *next*, aby umożliwić obserwację ich wartości podczas symulacji projektu.

Stany automatu oraz ich kody są definiowane jako parametry lokalne (lub mogą być wykorzystane jako zwykłe parametry), przy czym rozmiar kodów powinien być taki sam, jak w przypadku zmiennych *state* i *next*. Zauważmy, że podczas deklaracji zmiennych lokalnych zawsze powinny być podane wartości początkowe, na przykład

```
localparam [1:0] s0 = 0, s1 = 1, s2 = 2, s3 = 3;
```

nawet w tych przypadkach, gdy będzie wykorzystywany inny sposób kodowania. Stan początkowy automatu skończonego powinien być umieszczony jako pierwszy na liście parametrów lokalnych.

W przytoczonym przykładzie kodu funkcjonowanie automatu skończonego opisane zostało za pomocą trzech procesów, czyli trzech bloków **always**.

Pierwszy proces opisuje pamięć automatu skończonego, to znaczy rejestr RG przedstawiony na rysunku 10.1. Rejestr jest reprezentowany przez zmienną *state*, a po wystąpieniu narastającego zbocza sygnału synchronizacji *clk* do rejestru zostanie zapisana wartość zmiennej *next*. Asynchroniczne zerowanie rejestru wykonywane jest w momencie wystąpienia opadającego zbocza sygnału *reset*, przy czym do rejestru ładowany jest kod stanu początkowego *s0* automatu.

Drugi proces opisuje przejścia pomiędzy stanami automatu skończonego, czyli funkcje przejść oraz schemat kombinacyjny CL_ϕ przedstawiony na rysunku 10.1. W tym celu wykorzystana została instrukcja **case** – sprawdza ona, w którym stanie znajduje się automat. Dla każdego stanu badane są warunki przejścia do następnego stanu, co pozwala określić wartość zmiennej *next*. W tym przypadku używane są instrukcja **if** oraz konstrukcja **else if**. Ostatnia konstrukcja **else** w instrukcji **if** odpowiada wszystkim pozostałym warunkom, które nie zostały wcześniej opisane. Na przykład przy sprawdzaniu warunków przejścia ze stanu *s1* ostatnia konstrukcja **else** będzie odpowiadać warunkowi przejścia ($x[2] \&\& x[1] \&\& x[0] == 1$).

Poprawny opis automatu powinien uwzględniać wszystkie możliwe scenariusze jego działania, więc przejścia z każdego stanu powinny być określone dla wszystkich możliwych kombinacji zmiennych wejściowych. Oznacza to, że każda instrukcja **if** zawsze powinna mieć odpowiadającą mu konstrukcję **else**. Przejścia z każdego stanu można także opisać za pomocą instrukcji **case**. Przykład takiego opisu zostanie omówiony w dalszej części rozdziału.

Trzeci proces opisuje logikę wyjściową automatu skończonego, czyli funkcje wyjść oraz układ kombinacyjny CL_ψ przedstawiony na rysunku 10.1. W tym bloku także wykorzystana została instrukcja **case** do sprawdzenia stanu, w którym znajduje się automat, oraz instrukcja **if** do wykrywania warunków przejścia i określenia wartości zmiennych wyjściowych.

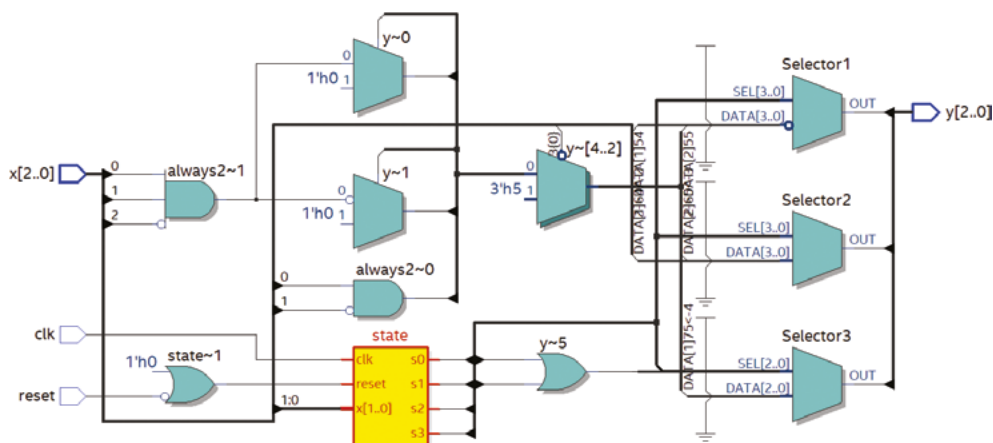
10.3.2. Synteza automatu Mealy’ego

Aby wykonać syntezę automatu opisanego w języku Verilog w systemie Quartus, należy ustawić odpowiednie parametry syntezy oraz przeprowadzić kompilację projektu.

W naszym przykładzie wybierzemy rodzinę Cyclone IV E układów FPGA (w tym celu w menu systemu Quartus przechodzimy do **Assignment > Device... > Device family**). Taki wybór jest niezbędny do obserwacji wyników symulacji czasowej, ponieważ w uproszczonej wersji systemu Quartus (University Program) możliwość analizy wyników w edytorze przebiegów czasowych (Simulation Waveform Editor) zachowana została tylko dla rodziny układów Cyclone IV.

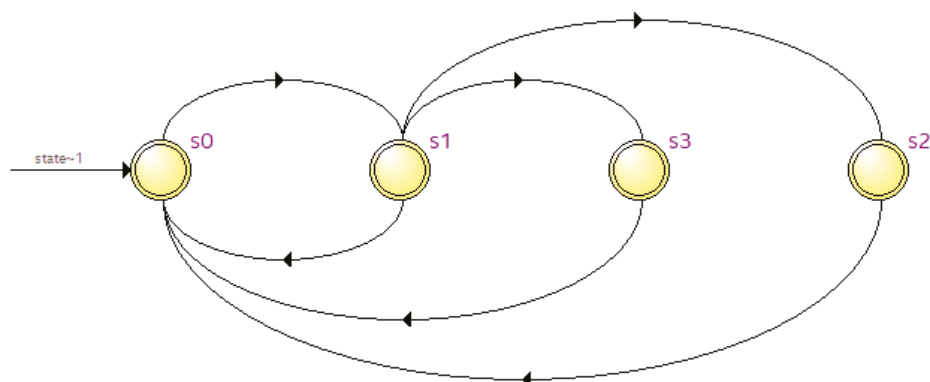
W naszym przykładzie ustawimy domyślny sposób kodowania stanów wewnętrznych automatu skończonego (**Assignment > Settings... > Computer Settings > Advanced Setting (Synthesis)... > State Machine Processing > Auto**).

Wynik syntezy automatu Mealy’ego dla naszego przykładu zaprezentowany został na rysunku 10.4. Można na nim zauważyć, że system rozpoznał w kodzie projektu automat skończony i zaznaczył go blokiem *state*. Pozostałe elementy schematu służą do formowania sygnałów wyjściowych na podstawie kodu stanu i wartości sygnałów na wejściach.



RYS. 10.4. Wynik syntezy projektu *Mealy_style_1* na poziomie przesłań międzyrejestrowych

Podwójnym kliknięciem na bloku *state* można otworzyć okno podglądu wyników syntezy automatu skończonego (**State Machine Viewer**), gdzie zwizualizowany został graf automatu (rysunek 10.5). W tym samym oknie w zakładce **Transitions** można obejrzeć zrealizowane w automacie przejścia pomiędzy stanami (rysunek 10.6), a w zakładce **Encoding** – kody przypisane stanom automatu (rysunek 10.7).



RYS. 10.5. Graf zaprojektowanego automatu Mealy'ego

	Source State	Destination State	Condition
1	s0	s1	
2	s1	s0	(x[0])!(x[1])
3	s1	s2	!(x[0])
4	s1	s3	(x[0])!(!(x[1]))
5	s2	s0	
6	s3	s0	

RYS. 10.6. Tablica przejść zaprojektowanego automatu Mealy'ego

	Name	s3	s2	s1	s0
1	s0	0	0	0	0
2	s1	0	0	1	1
3	s2	0	1	0	1
4	s3	1	0	0	1

RYS. 10.7. Kody stanów wewnętrznych zaprojektowanego automatu Mealy'ego

Tablica przejść na rysunku 10.6 składa się z trzech kolumn: *Source State* – początkowy stan przejścia; *Destination State* – końcowy stan przejścia; *Condition* – warunek, który umożliwia dane przejście. Jeśli pole w kolumnie *Condition* jest puste, oznacza to *przejście bezwarunkowe*, które nie zależy od wartości zmiennych wejściowych.

Zauważmy, że tablica przejść zaprojektowanego automatu Mealy'ego na rysunku 10.6 różni się od tablicy przejść oryginalnego automatu Mealy'ego przedstawionej w tabeli 10.1. Zmiana ta nastąpiła, ponieważ syntezytor systemu Quartus automatycznie dokonał minimalizacji automatu, co spowodowało otrzymanie tablicy przejść widocznej na rysunku 10.6.

Na rysunku 10.7 zauważamy, że do kodowania stanów automatu z naszego przykładu został wykorzystany 4-bitowy kod zamiast przypuszczanego 2-bitowego. Stanom automatu zostały przypisane następujące kody: s0 – 0000, s1 – 0011, s2 – 0101, s3 – 1001, co jest zupełnie innym zestawem niż ten podany w opisie projektu. Stało się tak, ponieważ w parametrach syntezy wybraliśmy domyślny sposób kodowania stanów (opcja **Auto**), a przy realizacji automatu na układach FPGA takim sposobem jest *kodowanie unarne (one-hot)*. Sposoby kodowania stanów wewnętrznych automatów zostaną omówione w dalszej części rozdziału.

10.3.3. Analiza funkcjonowania automatu Mealy'ego

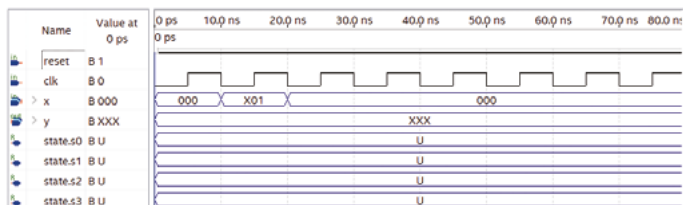
Po zakończeniu syntezy projektu zawsze warto sprawdzić poprawność jego funkcjonowania oraz określić momenty czasu, w których generowane są sygnały wyjściowe. Jest to szczególnie istotne w przypadku projektowania automatów skończonych, gdyż prawidłowe działanie układu sterowania ma kluczowe znaczenie dla całego systemu.

Przeanalizujemy działanie zaprojektowanego automatu Mealy'ego, korzystając z narzędzi do funkcjonalnej i czasowej symulacji systemu Quartus. Do tego celu posłużymy się edytorem przebiegów czasowych (**Simulation Waveform Editor**).

Aby przeprowadzić symulację działania zaprojektowanego automatu skończonego, należy najpierw utworzyć plik przebiegów czasowych (rysunek 10.8).

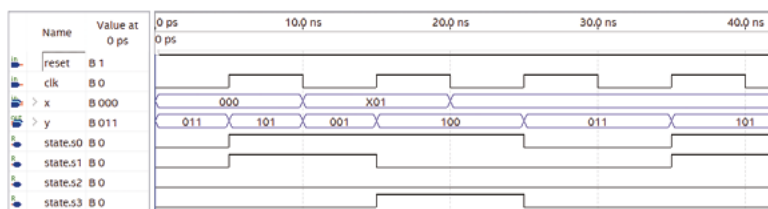
Ustawimy wartości zmiennych wejściowych projektu tak, jak pokazano na rysunku 10.8. W powyższym przykładzie czas cyklu sygnału synchronizacji wynosi 10 ns. Pierwszy sygnał wejściowy podawany na wejście automatu to wektor zerowy 000. Nie ma on wpływu na zmianę stanu automatu, ponieważ przejście ze stanu początkowego s0 do stanu s1 jest bezwarunkowe. W chwili 10 ns na wejście automatu podawany jest wektor x01, co powinno przełączyć automat ze stanu s1 do stanu s3. W kolejnym

momencie czasowym, tj. 20 ns, na wejściach automatu pojawiają się ponownie wartości zerowe, które znów nie mają wpływu na przejście ze stanu s_3 do stanu s_0 , gdyż to przejście w naszym automacie również jest bezwarunkowe. Sugerujemy czytelnikowi, aby rozważył, jakie wartości zmiennych wyjściowych automatu będą generowane w różnych momentach czasu, a następnie porównał je z otrzymanymi wynikami modelowania.



RYS. 10.8. Plik przebiegów czasowych

Przeprowadzimy funkcjonalną i czasową symulację automatu skończonego. Wyniki symulacji funkcjonalnej automatu Mealy'ego zamieszczone zostały na rysunku 10.9.



RYS. 10.9. Wyniki symulacji funkcjonalnej automatu Mealy'ego

Analizując wyniki symulacji przedstawione na rysunku 10.9, widzimy, że wartości zmiennych wewnętrznych $state.s_0$, ..., $state.s_3$, zgodnie z rysunkiem 10.7, reprezentują kod stanu, w którym znajduje się automat. W przedziale czasowym od 0 ns do 5 ns automat pozostaje w stanie początkowym s_0 , przy czym na wejściach podany jest wektor 000, a na wyjściach wygenerowane zostają sygnały wyjściowe 011, co jest zgodne z oczekiwanym bezwarunkowym przejściem ze stanu s_0 do stanu s_1 .

W chwili 5 ns narastające zbocze sygnału synchronizacji powoduje przejście automatu do stanu s_1 (kod 0011). Pomimo że na wejściach automatu pozostają wartości zerowe 000, zmienne na jego wyjściach przyjmują wartości 101, zgodnie z tym, co zachodzi podczas przejścia ze stanu s_1 do stanu s_2 . Należy zauważyć, że w tym momencie automat nie przechodzi bezpośrednio ze stanu s_1 w stan s_2 .

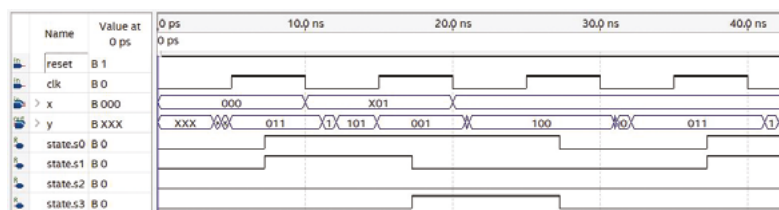
W chwili 10 ns na wejściach automatu pojawia się nowy wektor wejściowy x01, co skutkuje wygenerowaniem na wyjściach automatu wartości zmiennych wyjściowych 001. Te wartości są formowane na przejściu ze stanu s_1 do stanu s_3 , jednakże warto zauważyć, że mimo to automat nadal pozostaje w stanie s_1 i jeszcze nie przeszedł do stanu s_3 .

W chwili 15 ns kolejne narastające zbocze sygnału synchronizacji powoduje przejście automatu do stanu s_3 (kod 1001). Przejście to jest inicjowane przez wektor wejściowy $x01$. Na wyjściach automatu generowane są wartości 100, które są przypisane bezwarunkowemu przejściu ze stanu s_3 do stanu s_0 .

W momencie czasu 25 ns automat powraca do stanu początkowego s_0 (kod 0000), a na wyjściach ponownie pojawiają się wartości 011.

Warto zauważyć, że wyjścia automatu Mealy'ego są zależne zarówno od wartości zmiennych wejściowych, jak i od jego wewnętrznego stanu. W przykładzie, który omawiamy, gdy na wejście podany zostaje wektor $x01$ o czasie 10 ns, na wyjściach generowane są sygnały 001. Jednakże po przejściu automatu w stan s_3 o czasie 15 ns, przy tych samych wartościach na wejściach, stany sygnałów wyjściowych zmieniają się na 100.

Symulacja funkcjonalna prezentuje przejścia stanów automatu skończonego oraz wartości generowane na wyjściach, nie biorąc pod uwagę opóźnień czasowych sygnałów i zakładając, że wszystkie opóźnienia są zerowe. Dokładniejsze dane na temat działania automatu skończonego można uzyskać z wyników symulacji czasowej (rysunek 10.10).



RYS. 10.10. Wyniki symulacji czasowej automatu Mealy'ego

Przeprowadzamy analizę wyników symulacji czasowej z rysunku 10.10. W przedziale czasu od 0 ns do 5 ns automat znajduje się w stanie początkowym s_0 , co wynika z wyzerowania wszystkich przerzutników FPGA po uruchomieniu zasilania. Na wejściach podany jest wektor zerowy 000, a na wyjściach występują nieokreślone wartości.

W chwili czasu 5 ns po wystąpieniu narastającego zbocza sygnału synchronizacji zmienne wyjściowe przyjmują wartości 011 (z opóźnieniem 5 ns w porównaniu z symulacją funkcjonalną). Narastające zbocze powoduje przejście automatu ze stanu s_0 do stanu s_1 , które odbywa się z opóźnieniem równym 2 ns, czyli w momencie czasu 7 ns.

W momencie czasu 10 ns na wejściach automatu pojawia się wektor $x01$. Po upływie 2 ns, tj. o czasie 12 ns, na wyjściach generowane są wartości 101. Jest to reakcja automatu na przejście (z opóźnieniem 5 ns) do stanu s_1 , które występuje o czasie 7 ns, oraz wystąpienia wektora zerowego na wejściach.

W chwili 15 ns pojawia się kolejne narastające zbocze sygnału synchronizacji, które powoduje przełączenie automatu do stanu s_3 z opóźnieniem 2 ns, czyli w chwili czasu 17 ns. Równocześnie na wyjściach automatu pojawiają się wartości 001 – jest to rezultat (z opóźnieniem 5 ns) podania wektora $x01$ na wejścia automatu znajdującego się w stanie s_1 . Następnie o czasie 22 ns na wyjściach generowane są sygnały

o wartościach 100, co jest konsekwencją przejścia automatu do stanu s_3 , które miało miejsce w chwili czasu 17 ns.

W chwili 25 ns następuje kolejne narastające zbocze sygnału synchronizacji, co skutkuje przejściem automatu do stanu s_0 z opóźnieniem 2 ns, tj. o czasie 27 ns. W rezultacie tego przejścia na wyjściach automatu pojawiają się wartości 011 (z opóźnieniem 5 ns), które widoczne są w chwili czasu 32 ns.

Analiza przebiegów czasowych na rysunku 10.10 sugeruje, że przełączenie stanów automatu ma miejsce z opóźnieniem około 2 ns, podczas gdy opóźnienie w układzie kombinacyjnym podczas generowania sygnałów wyjściowych wynosi około 5 ns. Porównanie przebiegów czasowych na rysunkach 10.9 i 10.10 ujawnia, że w przypadku symulacji czasowej przełączanie stanów automatu skończonego zachodzi z opóźnieniem 2 ns, a generowanie sygnałów wyjściowych następuje z opóźnieniem 5–7 ns.

10.4. Projektowanie automatu Moore'a

10.4.1. Opis automatu Moore'a

Poniżej zamieszczony został jeden z możliwych opisów w języku Verilog automatu Moore'a z naszego przykładu na rysunku 10.3.

```
module Moore_style_1(  
    input clk,reset,  
    input [2:0] x,  
    output reg [2:0] y);  
  
    reg [2:0] state, next;  
  
    localparam [2:0]  
        s0=3'b000,  
        s1=3'b001,  
        s2=3'b010,  
        s3=3'b011,  
        s4=3'b100,  
        s5=3'b101;  
  
    always @(posedge clk, negedge reset)  
        if (~reset)    state <= s0;  
        else          state <= next;  
  
    always @(*)          // opis funkcji przejść  
        case (state)
```

```

s0:  next = s1;
s1:  if (~x[0])                next = s2;
      else if (~x[1] && x[0])    next = s3;
      else if (~x[2] && x[1] && x[0]) next = s4;
      else                    next = s5;
s2:  if (~x[2])      next = s4;
      else          next = s5;
s3:  next = s4;
s4:  next = s0;
s5:  next = s0;
endcase

always @(*)                // opis funkcji wyjść
  case (state)
    s0:  y = 3'b000;
    s1:  y = 3'b011;
    s2:  y = 3'b101;
    s3:  y = 3'b001;
    s4:  y = 3'b100;
    s5:  y = 3'b010;
  endcase

endmodule

```

Główną różnicą w przytoczonym kodzie, w porównaniu z podobnym opisem automatu Mealy'ego, jest inny sposób definiowania funkcji wyjściowych automatu. W naszym kodzie do tego celu używana jest instrukcja **case**, która analizuje aktualny stan automatu i generuje odpowiednie wartości zmiennych wyjściowych. Alternatywnie można także skorzystać z instrukcji **if**, na przykład:

```

if (state==s0)      y = 3'b000;
else if (state==s1) y = 3'b011;
else if (state==s2) y = 3'b101;
      else if (state==s3) y = 3'b001;
else if (state==s4) y = 3'b100;
else y = 3'b010;

```

Inną metodą opisu wyjść automatu jest definiowanie wartości każdej zmiennej wektora wyjściowego za pomocą instrukcji ciągłego przypisywania **assign**:

```

...
output [2:0] y;
...

```

```

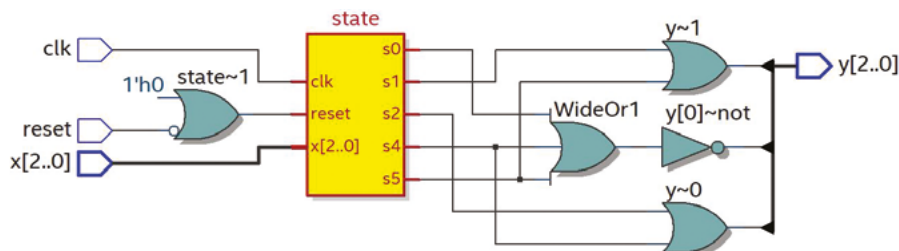
assign y[0] = state==s1 ||
           state==s2 ||
           state==s3;
assign y[1] = state==s1 ||
           state==s5;
assign y[2] = state==s2 ||
           state==s4;

```

Zauważmy, że w ostatnim przypadku wektor y jest deklarowany bez użycia słowa kluczowego **reg**, a opis automatu zawiera tylko dwa procesy (instrukcje **always**).

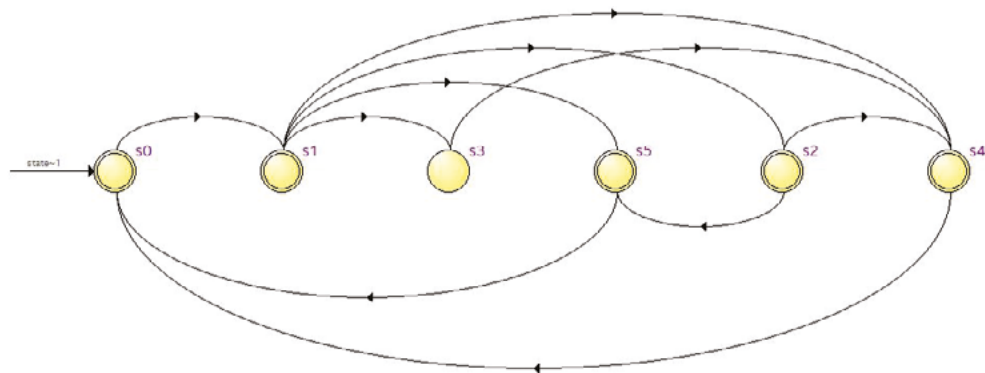
10.4.2. Synteza automatu Moore'a

Wyniki syntezy automatu Moore'a przedstawione zostały na rysunkach 10.11–10.14.



RYS. 10.11. Wynik syntezy projektu *Moore_style_1* na poziomie przesłań międzyrejestrowych

Na rysunku 10.11 można zauważyć, że w automacie Moore'a logika generowania sygnałów wyjściowych jest znacznie prostsza w porównaniu z automatem Mealy'ego (rysunek 10.4).



RYS. 10.12. Graf zaprojektowanego automatu Moore'a

Zgodnie z oczekiwaniami liczba stanów wewnętrznych automatu Moore'a wzrosła do 6 (rysunek 10.12). Zwiększyła się także liczba przejść pomiędzy stanami.

	Source State	Destination State	Condition
1	s0	s1	
2	s1	s2	(!x[0])
3	s1	s5	(x[2])(x[0])(x[1])
4	s1	s4	(!x[2])(x[0])(x[1])
5	s1	s3	(x[0])(!x[1])
6	s2	s5	(x[2])
7	s2	s4	(!x[2])
8	s3	s4	
9	s4	s0	
10	s5	s0	

RYS. 10.13. Tablica przejść zaprojektowanego automatu Moore'a

Zwróćmy uwagę na to, że w automacie Moore'a (rysunek 10.13), w porównaniu z automatem Mealy'ego (rysunek 10.6), warunki przejść pomiędzy stanami stały się bardziej złożone, a także zwiększyła się ich liczba.

	Name	s5	s4	s3	s2	s1	s0
1	s0	0	0	0	0	0	0
2	s1	0	0	0	0	1	1
3	s2	0	0	0	1	0	1
4	s3	0	0	1	0	0	1
5	s4	0	1	0	0	0	1
6	s5	1	0	0	0	0	1

RYS. 10.14. Kody stanów wewnętrznych zaprojektowanego automatu Moore'a

W zaprojektowanym automacie Moore'a zwiększył się też rozmiar kodu stanów wewnętrznych (rysunek 10.14).

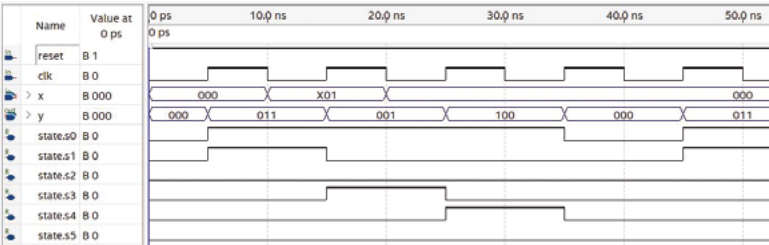
10.4.3. Analiza funkcjonowania automatu Moore'a

Podczas symulacji automatu Moore'a wykorzystujemy te same dane wejściowe co dla automatu Mealy'ego (rysunek 10.8). Wyniki symulacji funkcjonalnej automatu Moore'a pokazane są na rysunku 10.15.

Na początku, po włączeniu zasilania, wszystkie rejestry FPGA są zerowane. To oznacza, że nasz automat skończony zostanie ustawiony w stan początkowy s0 z kodem zerowym. Na wyjściach wygenerowane zostaną wartości 000.

W chwili czasu 5 ns narastające zbocze sygnału synchronizacji powoduje przejście automatu do stanu s1 (kod 000011), a na wyjściach wygenerowane zostają wartości 011. Warto zauważyć, że podanie na wejścia automatu wektora x01 o czasie 10 ns

nie zmienia wartości wyjść, w przeciwieństwie do tego, co miało miejsce w automacie Mealy’ego.



RYS. 10.15. Wyniki symulacji funkcjonalnej automatu Moore’a

W momencie czasu 15 ns kolejne narastające zbocze sygnału synchronizacji powoduje przejście automatu do stanu s3 (kod 001001), a na wyjściach pojawiają się wartości 001. Ponownie zauważmy, że zmiana wartości na wejściach w chwili 20 ns z x01 na 000 nie przełącza stanu wyjść automatu.

W chwili czasu 25 ns kolejne zbocze sygnału synchronizacji powoduje przejście automatu do stanu s4 (kod 010001), generując na wyjściach sygnały 100, zgodnie z grafem automatu przedstawionym na rysunku 10.3.

W chwili 35 ns po kolejnym zboczach sygnału synchronizacji automat powraca do stanu początkowego s0, a wyjścia automatu przyjmują zerowe wartości 000.

Porównanie wyników symulacji funkcjonalnej automatów Moore’a (rysunek 10.15) i Mealy’ego (rysunek 10.9) ukazuje, że pomimo podawania identycznych wartości na wejścia obu automatów w tych samych chwilach czasu ich działanie znacząco różni się od siebie. Te różnice można zaobserwować w tabeli 10.3.

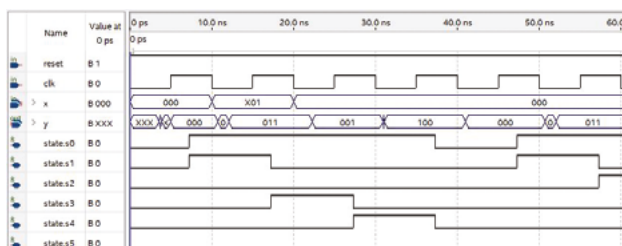
TABELA 10.3. Porównanie wyników symulacji funkcjonalnej automatów Mealy’ego i Moore’a

Czas (ns)	Wejście	Automat Mealy’ego		Automat Moore’a	
		Stan	Wyjście	Stan	Wyjście
0	000	s0	011	s0	000
5	000	s1	101	s1	011
10	x01	s1	001	s1	011
15	x01	s3	100	s3	001
20	000	s3	100	s3	001
25	000	s0	011	s4	100
30	000	s0	011	s4	100
35	000	s1	101	s0	000
40	000	s1	101	s0	000
45	000			s1	011
50	000			s1	011

Zauważmy, że na wyjściach automatu Mealy'ego w chwilach czasu 5 ns, 35 ns i 40 ns generowane są wartości 101, które w ogóle nie pojawiają się na wyjściach automatu Moore'a w przedziale czasowym od 0 ns do 50 ns. Proponujemy czytelnikowi wyjaśnienie tego paradoksu.

Proponujemy też, aby czytelnik przeprowadził formalną konwersję automatu Mealy'ego z rysunku 10.2 na automat Moore'a, korzystając z jednej ze znanych metod. Następnie sugerujemy dokonanie syntezy i symulacji uzyskanego automatu Moore'a oraz porównanie wyników symulacji funkcjonalnej z przedstawionymi powyżej.

Wyniki symulacji czasowej automatu Moore'a pokazane są na rysunku 10.16.



RYS. 10.16. Wyniki symulacji czasowej automatu Moore'a

Przedyskutujmy szczegółowo wyniki symulacji czasowej automatu Moore'a, jakie przedstawiono na rysunku 10.16, ponieważ dostarczają one bardziej szczegółowych informacji na temat rzeczywistego działania układu.

W chwili czasu 0 ns wyjścia automatu mają wartości nieokreślone, mimo że znajduje się on w stanie s_0 . Wartości 000 na wyjściach pojawiają się z opóźnieniem 5 ns w porównaniu z przebiegami symulacji funkcjonalnej. To opóźnienie odpowiada opóźnieniu generowanemu przez układ kombinacyjny CL_ψ (rysunek 10.1), który formuje sygnały wyjściowe.

Narastające zbocza sygnału synchronizacji pojawiają się w chwilach 5 ns, 15 ns, 25 ns, 35 ns, 45 ns itd., natomiast przełączenia stanów automatu wykonywane są z opóźnieniem 2 ns, tj. o czasie 7 ns, 17 ns, 27 ns, 37 ns, 47 ns, odpowiednio. Wartości zmiennych wyjściowych po przełączeniu automatu do odpowiedniego stanu generowane są z opóźnieniem 5 ns, tj. w chwilach 12 ns, 22 ns, 32 ns, 42 ns, 52 ns, odpowiednio.

Warto zauważyć, że w chwili czasu 42 ns na wyjściach automatu Moore'a generowane są wartości 000, co odpowiada powrotowi automatu do stanu początkowego s_0 , co nie jest obserwowane w przypadku automatu Mealy'ego. Dodatkowo automat Moore'a nie generuje wartości wyjściowych 101, które pojawiają się na wyjściach automatu Mealy'ego w odpowiedzi na podanie wektora zmiennych wejściowych $x01$.

Uwaga. Podczas realizacji algorytmu sterowania należy dokładnie dobierać typ automatu skończonego (Mealy'ego czy Moore'a), ponieważ ich funkcjonowanie może znacznie różnić się od siebie.

Aby zweryfikować poprawność działania automatu skończonego, zawsze należy przeprowadzić symulację czasową, która umożliwia dokładniejszą ocenę funkcjonowania zaprojektowanego układu. Symulacja funkcjonalna służy jedynie do sprawdzenia logiki działania projektu bez uwzględnienia opóźnień czasowych.

W naszych przykładach korzystamy z uproszczonej wersji narzędzi symulacyjnych z programu University Program, która jest przydatna jedynie do weryfikacji prostych projektów i przeznaczona głównie do celów edukacyjnych. Symulację bardziej złożonych projektów można przeprowadzać przy użyciu analizatora czasowego Time Quest w systemie Quartus. Alternatywnie symulację można również przeprowadzać, wykorzystując zasoby języka Verilog [14].

10.5. Style opisu automatów skończonych

Język Verilog oferuje użytkownikom różnorodne metody opisu automatów skończonych. W literaturze znane są trzy podstawowe style opisu automatów w języku Verilog: z trzema procesami, z dwoma procesami oraz z jednym procesem. Warto zaznaczyć, że styl opisu z jednym procesem może być stosowany jedynie do opisu automatów Moore'a.

Przeanalizujemy różne style opisu automatów skończonych w języku Verilog. W przedstawionych wcześniej przykładach opisu automatów wykorzystywano trzy procesy. Taki sposób opisu będziemy nazywać *stylem 1*.

10.5.1. Opis automatów skończonych z dwoma procesami

Bloki **always** (procesy), które opisują pamięć automatu, funkcje przejść oraz funkcje wyjść, mogą być łączone ze sobą w różnych kombinacjach. Na przykład połączenie logiki przejść z pamięcią automatu w jednym procesie prowadzi do uzyskania opisu automatu skończonego z dwoma procesami. Taki sposób opisu będziemy nazywać *stylem 2*. Poniżej przedstawiony jest kod opisu w stylu 2 automatu Mealy'ego z naszego przykładu:

```
module Mealy_style_2(  
    input clk,reset,  
    input [2:0] x,  
    output reg [2:0] y);  
  
    reg [1:0] state;  
  
    localparam [1:0]  
        s0=2'b00,
```



```

s1=2'b01,
s2=2'b10,
s3=2'b11;

always @(posedge clk, negedge reset)
    if (~reset)    state <= s0;
    else
        case (state)
        s0:    state = s1;
        s1:    if (~x[0])                                state = s2;
                else if (~x[1] && x[0])                    state = s3;
                else if (~x[2] && x[1] && x[0])            state = s0;
                else                                        state = s0;
        s2:    if (~x[2])    state = s0;
                else        state = s0;
        s3:    state = s0;
        endcase

always @(*)
    case (state)
    s0:    y = 3'b011;
    s1:    if (~x[0])                                y = 3'b101;
                else if (~x[1] && x[0])                    y = 3'b001;
                else if (~x[2] && x[1] && x[0])            y = 3'b100;
                else                                        y = 3'b010;
    s2:    if (~x[2])    y = 3'b100;
                else        y = 3'b010;
    s3:    y = 3'b100;
    endcase

endmodule

```

Wyniki syntezy projektu *Mealy_style_2* są zgodne z wynikami uzyskanymi dla projektu *Mealy_style_1* (rysunek 10.4). Ponadto wyniki funkcjonalnej i czasowej symulacji są identyczne z odpowiednimi wynikami przedstawionymi na rysunkach 10.9 i 10.10.

Zachęcamy czytelnika do samodzielnego opisania automatu Moore'a w stylu 2 oraz porównania otrzymanych wyników syntezy i symulacji z analogicznymi wynikami projektu *Moore_style_1* na rysunkach 10.11, 10.15 i 10.16.

Można również połączyć opis funkcji przejść oraz wyjść w jeden proces. Taki sposób opisu automatów skończonych będziemy nazywać *stylem 3*. Opis automatu Mealy'ego w stylu 3 można przedstawić następująco:

```

module Mealy_style_3(
    input clk,reset,
    input [2:0] x,
    output reg [2:0] y);

    reg [1:0] state, next;

    localparam [1:0]
        s0=2'b00,
        s1=2'b01,
        s2=2'b10,
        s3=2'b11;

    always @(posedge clk, negedge reset)
        if (~reset)    state <= s0;
        else          state <= next;

    always @(*)
        case (state)
            s0:    begin next = s1; y = 3'b011; end
            s1:    if (~x[0])                begin next = s2; y = 3'b101; end
        else if (~x[1] && x[0])                begin next = s3; y = 3'b001; end
            else if (~x[2] && x[1] && x[0])    begin next = s0; y = 3'b100; end
            else                               begin next = s0; y = 3'b010; end
            s2:    if (~x[2])                begin next = s0; y = 3'b100; end
            else                               begin next = s0; y = 3'b010; end
            s3:    begin next = s0; y = 3'b100; end
        endcase
endmodule

```

Cechą charakterystyczną takiego opisu jest obecność tylko jednej instrukcji **case** w kodzie. W każdym warunku stałym instrukcji **case** określany jest stan przejścia (zmienna *next*), a jednocześnie generowane są odpowiednie sygnały wyjściowe (zmienna *y*). Wyniki syntezy projektu *Mealy_style_3* są zgodne z wynikami uzyskanymi w przypadku syntezy projektu *Mealy_style_1* (rysunek 10.4). Dodatkowo wyniki symulacji funkcjonalnej i czasowej również nie różnią się od analogicznych wyników przedstawionych na rysunkach 10.9 i 10.10.

Opis automatu Moore'a w stylu 3 można przedstawić następująco:

```

module Moore_style_3(
    input clk,reset,
    input [2:0] x,
    output reg [2:0] y);

```

```

reg [2:0] state, next;

localparam [2:0]
    s0=3'b000,
    s1=3'b001,
    s2=3'b010,
    s3=3'b011,
    s4=3'b100,
    s5=3'b101;

always @(posedge clk, negedge reset)
    if (~reset)    state <= s0;
    else           state <= next;

always @(*)
    case (state)
    s0:    begin y = 3'b000; next = s1; end
    s1:    begin y = 3'b011;
            if (~x[0])                next = s2;
            else if (~x[1] && x[0])    next = s3;
            else if (~x[2] && x[1] && x[0]) next = s4;
            else                     next = s5;
        end
    s2:    begin y = 3'b101;
            if (~x[2])    next = s4;
            else         next = s5;
        end
    s3:    begin y = 3'b001; next = s4; end
    s4:    begin y = 3'b100; next = s0; end
    s5:    begin y = 3'b010; next = s0; end
    endcase
endmodule

```

Warto zwrócić uwagę, że również w tym opisie występuje tylko jedna instrukcja **case**. Jej każdy element stały zawiera blok, w którym oddzielnie określone są wartości wyjść automatu w danym stanie oraz sprawdzane są warunki przejść z tego stanu. Wynik syntezy projektu *Moore_style_3* jest zgodny z wynikiem syntezy projektu *Moore_style_1* (rysunek 10.11). Dodatkowo wyniki funkcjonalnej i czasowej symulacji pokrywają się z analogicznymi wynikami przedstawionymi na rysunkach 10.15 i 10.16.

Można też opisać automat skończony za pomocą dwóch procesów, łącząc opis pamięci i funkcji wyjść automatu w jeden proces. Zachęcamy czytelnika do samodzielnego opisanie w ten sposób automatów Mealy'ego i Moore'a z naszego przykładu,

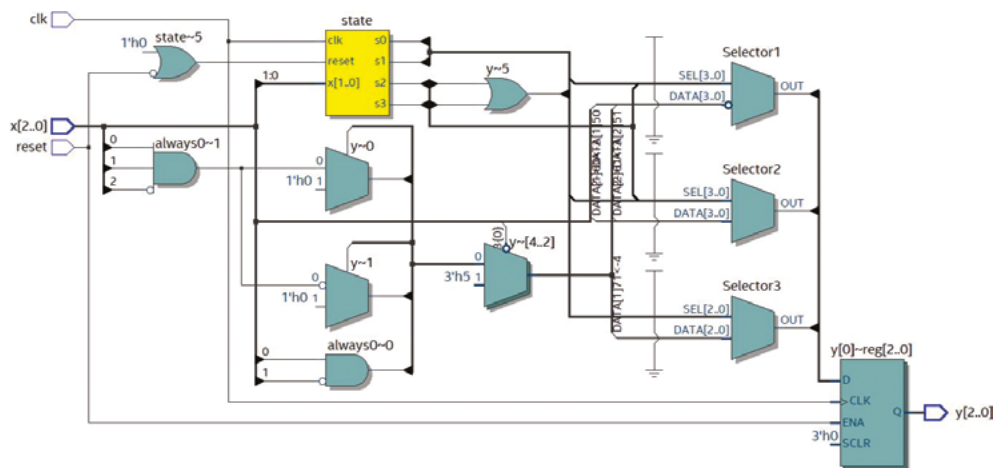
a następnie porównania wyników syntezy i symulacji zaprojektowanych automatów z analogicznymi wynikami projektów *Mealy_style_3* i *Moore_style_3*.

10.5.2. Opis automatów skończonych z jednym procesem

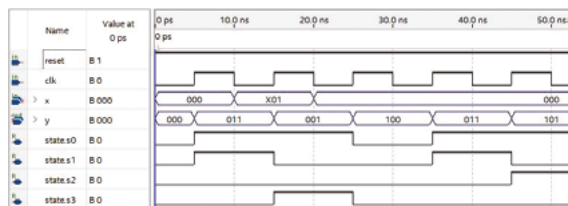
Wszystkie trzy procesy opisu automatu skończonego można także połączyć w jeden proces. Taki sposób opisu automatów skończonych będziemy nazywać *stylem 4*. Automat Mealy'ego w stylu 4 można przedstawić w następujący sposób:

```
module Mealy_style_4(  
    input clk,reset,  
    input [2:0] x,  
    output reg [2:0] y);  
  
    reg [1:0] state;  
  
    localparam [1:0]  
        s0=2'b00,  
        s1=2'b01,  
        s2=2'b10,  
        s3=2'b11;  
  
    always @(posedge clk, negedge reset)  
        if (~reset) state <= s0;  
        else  
            case (state)  
s0: begin state = s1; y = 3'b011; end  
s1: if (~x[0]) begin state = s2; y = 3'b101; end  
        else if (~x[1] && x[0]) begin state = s3; y = 3'b001; end  
        else if (~x[2] && x[1] && x[0]) begin state = s0; y = 3'b100; end  
        else begin state = s0; y = 3'b010; end  
s2: if (~x[2]) begin state = s0; y = 3'b100; end  
        else begin state = s0; y = 3'b010; end  
s3: begin state = s0; y = 3'b100; end  
            endcase  
endmodule
```

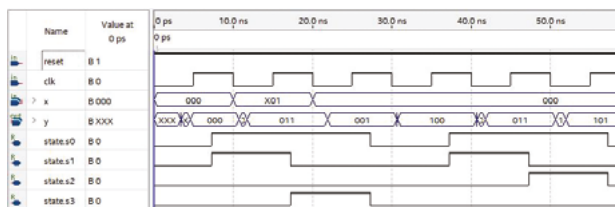
W powyższym kodzie występuje tylko jedna zmienna *state* do opisu stanów automatu skończonego. Wynik syntezy projektu *Mealy_style_4* zaprezentowany został na rysunku 10.17, natomiast wyniki symulacji funkcjonalnej i czasowej – odpowiednio, na rysunkach 10.18 i 10.19.



RYS. 10.17. Wynik syntezy projektu *Mealy_style_4* na poziomie przesłań międzyrejestrowych



RYS. 10.18. Wyniki symulacji funkcjonalnej projektu *Mealy_style_4*



RYS. 10.19. Wyniki symulacji czasowej projektu *Mealy_style_4*

Porównanie rysunków 10.17 i 10.4 pokazuje, że w schemacie automatu Mealy'ego opisanego w stylu 4 na wyjściach automatu został umieszczony rejestr. Z tego powodu wartości zmiennych wyjściowych będą generowane z opóźnieniem o jeden cykl sygnału synchronizacji w porównaniu z automatem Mealy'ego opisanym w stylu 1. Można się o tym upewnić, porównując wyniki funkcjonalnego i czasowego modelowania na rysunkach 10.18 i 10.19 z wynikami zamieszczonymi na rysunkach 10.9 i 10.10.

Czytelnikowi proponujemy samodzielne opisanie automatu Moore'a z naszego przykładu w stylu 4 i zbadanie, czy na wyjściach automatu również zostaną ustawione rejestry. W konsekwencji może wystąpić opóźnienie w generowaniu zmiennych wyjściowych o jeden cykl sygnału synchronizacji w porównaniu z automatem opisanym w stylu 1.

Pokażemy, że automat Moore'a można także opisać za pomocą jednego procesu, ale bez wprowadzenia opóźnień generowania sygnałów wyjściowych o jeden cykl. W tym celu można wykorzystać styl 2, w którym zamiast drugiego procesu (opisu- jącego funkcje wyjść) należy umieścić instrukcję (lub instrukcje) ciągłego przypisy- wania **assign**. Poniżej znajduje się przykład takiego programu.

```

module Moore_1_process(
    input clk,reset,
    input [2:0] x,
    output [2:0] y);

    reg [2:0] state;

    localparam [2:0]
        s0=3'b000,
        s1=3'b001,
        s2=3'b010,
        s3=3'b011,
        s4=3'b100,
        s5=3'b101;

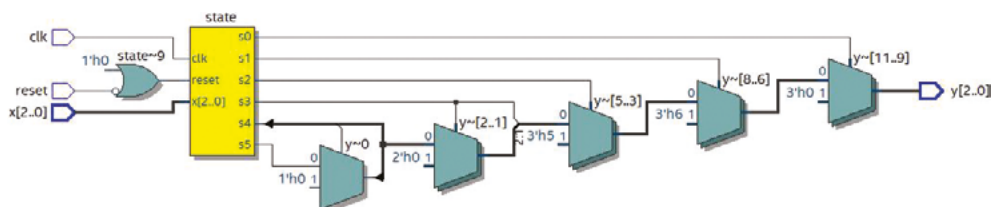
    always @(posedge clk, negedge reset)
        if (~reset)    state <= s0;
        else
            case (state)
                s0:    state = s1;
                s1:    if          (~x[0])            state = s2;
                     else if (~x[1] && x[0])        state = s3;
                     else if (~x[2] && x[1] && x[0]) state = s4;
                     else                          state = s5;
                s2:    if (~x[2])    state = s4;
                     else          state = s5;
                s3:    state = s4;
                s4:    state = s0;
                s5:    state = s0;
            endcase

    assign y = ((state==s0) ? 3'b000:(
        (state==s1) ? 3'b011:(
        (state==s2) ? 3'b101:(
        (state==s3) ? 3'b001:(
        (state==s4) ? 3'b100:(
        (state==s5) ? 3'b010: 3'b000))))));

endmodule

```

Wyniki syntezy projektu *Moore_1_process* ukazane są na rysunku 10.20.



RYS. 10.20. Wynik syntezy projektu *Moore_1_process* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Moore_1_process* dokładnie się zgadzają z odpowiednimi wynikami na rysunkach 10.15 i 10.16.

10.5.3. Opis automatów skończonych z rejestrami na wyjściach

W niektórych projektach warto rozważyć umieszczenie rejestrów na wyjściach automatu skończonego, co może być korzystne dla precyzyjnego powiązania momentów generowania wartości zmiennych wyjściowych z sygnałem synchronizacji. Aby to osiągnąć, sugeruje się modyfikację trzeciego procesu w stylu 1, który opisuje funkcje wyjść, przez zastąpienie konstrukcji **always @(*)** przez **always @(posedge clk)**. Taki styl opisu automatów skończonych będziemy nazywać *stylem 5*. Zachęcamy czytelnika do samodzielnego opisania automatów Mealy'ego i Moore'a z naszego przykładu z zastosowaniem stylu opisu 5. Następnie proponujemy przeprowadzenie syntezy i symulacji obu automatów oraz porównanie otrzymanych wyników ze stylem 1 i stylem 4.

10.5.4. Opis automatów skończonych za pomocą instrukcji case

W dotychczasowych opisach automatów skończonych warunki przejścia między stanami były zazwyczaj sprawdzane za pomocą instrukcji warunkowych **if**. Jednakże w celu osiągnięcia tego samego efektu można również zastosować instrukcje **case**. Takie podejście do opisu automatów skończonych będziemy określać jako *styl 6*. Poniżej przedstawiono kod automatu Mealy'ego opisanego z wykorzystaniem stylu 6.

```
module Mealy_style_6(
    input clk,reset,
    input [2:0] x,
    output reg [2:0] y);

    reg [1:0] state,next;
```

```

localparam [1:0]
    s0=2'b00,
    s1=2'b01,
    s2=2'b10,
    s3=2'b11;

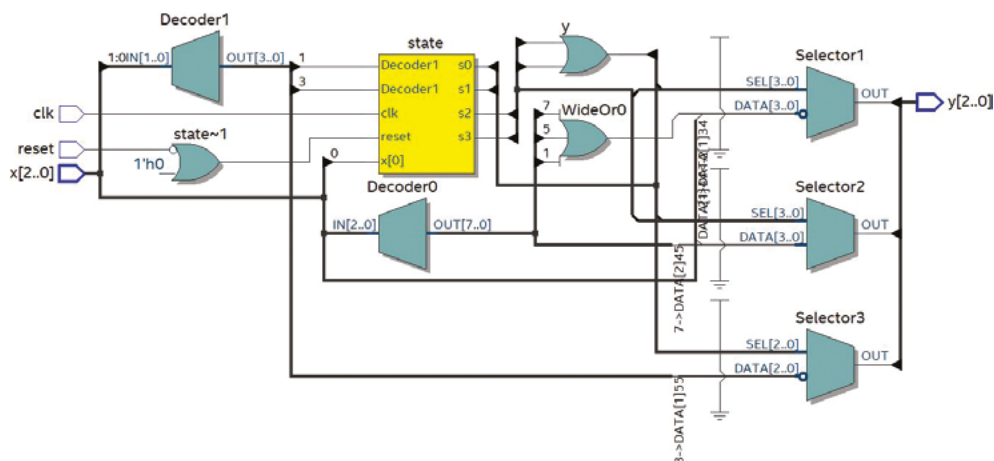
always @(posedge clk, negedge reset)
    if (~reset)    state <= s0;
    else           state <= next;

always @(*)
    case (state)
        s0:    next = s1;
        s1:    casex (x)
                3'b??0:    next = s2;
                3'b?01:    next = s3;
                3'b011: next = s0;
                3'b111: next = s0;
            endcase
        s2:    casex (x)
                3'b0??:    next = s0;
                3'b1??: next = s0;
            endcase
        s3:    next = s0;
    endcase

always @(*)
    case (state)
        s0:    y = 3'b011;
        s1:    casex (x)
                3'b??0:    y = 3'b101;
                3'b?01: y = 3'b001;
                3'b011: y = 3'b100;
                3'b111: y = 3'b010;
            endcase
        s2:    casex (x)
                3'b0??:    y = 3'b100;
                3'b1??: y = 3'b010;
            endcase
        s3:    y = 3'b100;
    endcase
endmodule

```


Wynik syntezy projektu *Mealy_style_6* zaprezentowany został na rysunku 10.21.



RYS. 10.21. Wynik syntezy projektu *Mealy_style_6* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Mealy_style_6* są całkowicie zgodne z wynikami projektu *Mealy_style_1*. Zachęcamy czytelnika do samodzielnego opisanego automatu Moore'a z naszego przykładu, zgodnie ze stylem 6, a następnie do wykonania jego syntezy oraz symulacji. Po czym sugerujemy porównanie uzyskanych wyników z analogicznymi wynikami projektu *Moore_style_1*.

Dokładne badania wykorzystania instrukcji **if** i **case** w opisach automatów skończonych można znaleźć w [15].

10.5.5. Opis niezawodnych automatów skończonych

Poprawny opis automatu skończonego obejmuje zdefiniowanie dla każdego stanu wszystkich możliwych przejść z niego. Dodatkowo suma logiczna wszystkich warunków przejść z każdego stanu powinna być równa jedynce logicznej. Oprócz tego każdy stan powinien mieć przynajmniej jedno przejście z innych stanów prowadzące do niego, a z każdego stanu powinno być przynajmniej jedno przejście do innych stanów. Przy zachowaniu tych warunków w automacie nie powinny występować nieprzewidziane przejścia lub warunki, które nie inicjują żadnych przejść.

Sytuacja staje się bardziej złożona w przypadku kodów stanów wewnętrznych. Jeśli w wyniku usterki rejestr pamięci automatu skończonego zawiera kod, który nie odpowiada żadnemu ze stanów wewnętrznych, automat zacznie działać nieprawidłowo.

Poniższy styl opisu pozwala na zwiększenie niezawodności działania automatów skończonych, co zostało osiągnięte przez modyfikację stylu 1. W instrukcji **case** została dodana konstrukcja **default**, która zapewnia powrót do stanu początkowego

w przypadku, gdy aktualny stan wewnętrzny (zmienna *state*) nie odpowiada żadnemu zdefiniowanemu stanowi w automacie. Na wyjściach automatu w tym przypadku generowane są wartości zerowe. Taki sposób opisu niezawodnego automatu skończonego określamy jako *styl 7*. Poniższy kod demonstruje opis automatu Mealy'ego z naszego przykładu przy użyciu stylu 7.

```

module Mealy_style_7(
    input clk,reset,
    input [2:0] x,
    output reg [2:0] y);

    reg [1:0] state, next;

    localparam [1:0]
        s0=2'b00,
        s1=2'b01,
        s2=2'b10,
        s3=2'b11;

    always @(posedge clk, negedge reset)
        if (~reset)    state <= s0;
        else           state <= next;

    always @(*)
        case (state)
            s0:    next = s1;
            s1:    if (~x[0])                next = s2;
                   else if (~x[1] && x[0])    next = s3;
                   else if (~x[2] && x[1] && x[0]) next = s0;
                   else                     next = s0;
            s2:    if (~x[2])    next = s0;
                   else        next = s0;
            s3:    next = s0;
            default:    next = s0;
        endcase

    always @(*)
        case (state)
            s0:    y = 3'b011;
            s1:    if (~x[0])                y = 3'b101;
                   else if (~x[1] && x[0])    y = 3'b001;
                   else if (~x[2] && x[1] && x[0]) y = 3'b100;
                   else                     y = 3'b010;
        endcase

```

```

        s2:    if (~x[2])    y = 3'b100;
               else        y = 3'b010;
        s3:    y = 3'b100;
        default:    y = 3'b000;
    endcase
endmodule

```

Wyniki syntezy i symulacji projektu *Mealy_style_7* są całkowicie zgodne z analogicznymi wynikami dla projektu *Mealy_style_1*. Zachęcamy czytelnika do samodzielnego opisanie automatu Moore'a z naszego przykładu w stylu 7 oraz porównania otrzymanych wyników syntezy i symulacji z odpowiednimi wynikami projektu *Moore_style_1*.

Możemy uzyskać jeszcze bardziej niezawodny automat, stosując specjalny sposób opisu. W tym celu zmodyfikujemy styl 6 w następujący sposób: w przypadku, gdy aktualny stan (zmienna *state*) nie odpowiada żadnemu istniejącemu stanowi wewnętrznemu, automat powraca do stanu początkowego (analogicznie do stylu 7), natomiast gdy napotkany zostanie nieistniejący warunek przejścia, automat pozostaje w obecnym stanie (realizowany jest stan oczekiwania). Taki sposób opisu niezawodnych automatów skończonych określimy jako *styl 8*. Poniżej znajdziemy opis automatu Mealy'ego z naszego przykładu z wykorzystaniem stylu 8.

```

module Mealy_style_8(
    input clk,reset,
    input [2:0] x,
    output reg [2:0] y);

    reg [1:0] state, next;

    localparam [1:0]
        s0=2'b00,
        s1=2'b01,
        s2=2'b10,
        s3=2'b11;

    always @(posedge clk, negedge reset)
        if (~reset)    state <= s0;
        else            state <= next;

    always @(*)
        case (state)
            s0:    next = s1;

```

```

s1:    case x (x)
        3'b??0:    next = s2;
        3'b?01:    next = s3;
        3'b011: next = s0;
        3'b111: next = s0;
        default: next = s1;

    endcase
s2:    case x (x)
        3'b0??:    next = s0;
        3'b1??: next = s0;
        default: next = s2;

    endcase
s3:    next = s0;
default: next = s0;
endcase

always @(*)
    case (state)
        s0:    y = 3'b011;
        s1:    case x (x)
                3'b??0:    y = 3'b101;
                3'b?01:    y = 3'b001;
                3'b011: y = 3'b100;
                3'b111: y = 3'b010;
                default: y = 3'b000;

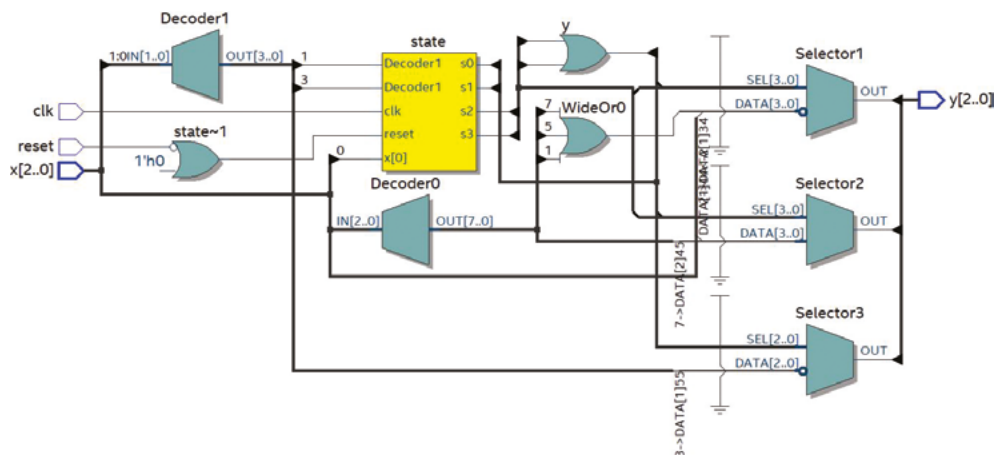
            endcase

        s2:    case x (x)
                3'b0??:    y = 3'b100;
                3'b1??: y = 3'b010;
                default: y = 3'b000;

            endcase
        s3:    y = 3'b100;
        default:    y = 3'b000;
    endcase
endmodule

```

Wynik syntezy projektu *Mealy_style_8* przedstawiony został na rysunku 10.22.



RYS. 10.22. Wynik syntezy projektu *Mealy_style_8* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Mealy_style_8* są identyczne z analogicznymi wynikami projektu *Mealy_style_1*. Zachęcamy czytelnika do opisanie automatu Moore'a z naszego przykładu, zgodnie ze stylem 8, oraz do porównania wyników syntezy i symulacji z odpowiednimi wynikami projektu *Moore_style_1*.

10.5.6. Porównanie stylów opisu automatów skończonych

Dokonano próby odpowiedzi na pytanie, który z przedstawionych stylów opisu automatów skończonych jest najlepszy. W tym celu przeprowadzono badania efektywności realizacji różnych opisów automatu z naszego przykładu w systemie Quartus Prime wersji 18.1 dla różnych rodzin układów FPGA. Ocena efektywności realizacji obejmowała szybkość działania i złożoność zaprojektowanych automatów. Wyniki przeprowadzonych badań zamieszczono w tabeli 10.4, gdzie L – liczba elementów logicznych FPGA wykorzystanych do realizacji automatu skończonego, F – maksymalna częstotliwość (w MHz) funkcjonowania automatu skończonego.

Najpierw omówimy style opisu automatu Mealy'ego. Dokładnie takie same wyniki pod względem złożoności realizacji oraz szybkości uzyskano dla stylów 1, 2, 3 oraz 7. Bardzo zbliżone wyniki uzyskano również dla stylów 6 i 8, jednak zaprojektowane w nich automaty okazały się wolniejsze (1217 w porównaniu z 1241) dla układów z rodziny Arria II, natomiast szybsze (703 w porównaniu z 701) dla układów z rodziny Cyclone V.

Style 4 i 5 opisują automaty skończone z rejestrami na wyjściach, gdzie sygnały wyjściowe są generowane z opóźnieniem o jeden cykl sygnału synchronizacji. Należy analizować te style osobno, ponieważ opisują one zupełnie inny rodzaj automatów.

TABELA 10.4. Wyniki badań stylów opisu automatów skończonych

Styl opisu	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy_														
Style_1	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
Style_2	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
Style_3	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
Style_4	10	466	10	257	11	572	7	1174	10	785	5	646	11	572
Style_5	10	459	10	236	11	644	7	1144	10	799	5	672	10	777
Style_6	10	539	10	310	11	760	7	1217	10	935	5	703	10	891
Style_7	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
Style_8	10	539	10	310	11	760	7	1217	10	935	5	703	10	891
Best Mealy	10	539	10	310	11	760	7	1241	10	935	5	703	10	891
Moore_														
Style_1	10	504	10	276	12	748	9	1175	11	833	6	669	11	902
Style_2	10	504	10	276	12	748	9	1175	11	833	6	669	11	902
Style_3	10	504	10	276	12	748	9	1175	11	833	6	669	11	902
Style_4	10	491	10	280	12	680	9	1115	11	902	6	656	11	820
Style_5	10	504	10	274	12	675	9	1168	11	895	6	670	11	917
Style_6	10	572	10	314	11	805	9	1084	10	962	5	701	10	964
Style_7	10	504	10	276	12	750	9	1175	11	833	6	669	11	902
Style_8	10	572	10	314	11	805	9	1086	10	962	5	701	10	964
1_process	10	502	10	278	11	762	9	1157	10	894	6	682	10	909
Best Moore	10	572	10	314	11	805	9	1175	10	962	5	701	10	964

Złożoność realizacji automatów opisanych w stylach 4 i 5 jest prawie identyczna jak w przypadku innych stylów, z wyjątkiem stylu 4 dla rodziny Cyclone 10, gdzie złożoność jest nieco większa (11 elementów logicznych zamiast 10). Prawie jednakowa złożoność realizacji automatów dla opisów w stylach 4 i 5 (z rejestrami na wyjściach), w porównaniu z innymi stylami (bez rejestrów na wyjściach), może być tłumaczona tym, że każdy element logiczny FPGA może być skonfigurowany z rejestrem na wyjściu, co eliminuje potrzebę dodatkowych elementów logicznych do realizacji rejestru.

Układy zaprojektowane w stylach 4 i 5 są wolniejsze dla wszystkich rodzin układów FPGA w porównaniu z innymi stylami. Ponadto szybkość działania układów opisanych w stylu 4 jest większa od tych opisanych w stylu 5 dla rodzin układów MAX II, MAX V oraz Arria II. Obniżenie szybkości działania w przypadku stylów 4 i 5, w porównaniu z innymi stylami, może być wyjaśnione obecnością dodatkowych rejestrów na wyjściach automatu skończonego.

Wnioski. Style 1, 2, 3, 6, 7 i 8 umożliwiają projektowanie automatów Mealy’ego o bardzo zbliżonych wartościach kosztu realizacji i szybkości działania. Wybór konkretnego stylu opisu zależy od preferencji użytkownika. Z punktu widzenia niezawodności funkcjonowania automatu skończonego rekomendowany jest wybór stylu 8. Style 4 i 5 pozwalają na budowę automatów skończonych z rejestrami na wyjściach, o podobnych parametrach, lecz wolniejszych w porównaniu ze stylami 1, 2, 3, 6, 7 i 8.

W przypadku automatu Moore’a można wyróżnić dwie grupy stylów opisu o bardzo zbliżonych parametrach wyników syntezy. Pierwsza grupa to style 1, 2, 3 i 7, a druga grupa to style 6 i 8. Ta druga grupa stylów jest lepsza pod względem złożoności realizacji dla następujących rodzin: MAX 10, Cyclone IV, Cyclone V i Cyclone 10. Ponadto jest także lepsza pod względem szybkości działania dla wszystkich rodzin układów, z wyjątkiem rodziny Arria II.

Style 4 i 5, które zawierają rejestry na wyjściach, prezentują podobną złożoność realizacji jak style z pierwszej grupy (1, 2, 3 i 7). Jednakże szybkość ich działania może się nieco różnić w zależności od rodziny układów FPGA. Dla niektórych rodzin szybkość działania może być nieznacznie wyższa, a dla innych trochę niższa.

Styl *Moore_1_process* wykazuje podobne wyniki pod względem złożoności jak druga grupa stylów (6 i 8), z wyjątkiem rodziny Cyclone V, gdzie występuje nieznaczne zwiększenie złożoności (6 elementów logicznych zamiast 5). Jeśli chodzi o szybkość działania, to jest on bliższy pierwszej grupie stylów (1, 2, 3 i 7). Dla niektórych rodzin układów FPGA szybkość jest nieznacznie większa (MAX V, MAX 10, Cyclone IV, Cyclone V i Cyclone 10), ale dla innych jest nieco mniejsza.

Wnioski. Najbardziej odpowiednie do opisu automatu Moore’a są style 6 i 8. Jednakże do praktycznego zastosowania zaleca się wybór stylu 8, ponieważ umożliwia on projektowanie bardziej niezawodnych automatów skończonych.

Aby odpowiedzieć na pytanie, który rodzaj automatów – Mealy’ego czy Moore’a – charakteryzuje się lepszymi parametrami syntezy, można porównać najlepsze wyniki z tabeli 10.4 dla automatu Mealy’ego (wiersz *Best Mealy*) oraz automatu Moore’a (wiersz *Best Moore*).

Według złożoności realizacji wyniki są podobne, z wyjątkiem rodziny układów Arria II (7 elementów logicznych dla automatu Mealy’ego oraz 9 dla automatu Moore’a). Natomiast pod względem szybkości działania automat Moore’a jest lepszy dla większości rodzin: MAX II, MAX V, MAX 10, Cyclone IV i Cyclone 10. Wynika to z tego, że funkcje wyjściowe w automacie Moore’a są znacznie prostsze niż w automacie Mealy’ego.

Wniosek. Do projektowania bardzo szybkich automatów skończonych najlepiej nadaje się automat Moore’a.

Przeprowadzone badania nie pozwalają na dokładną ocenę przedstawionych stylów automatów skończonych. Wyniki mogą być użyte jedynie do wstępnej oceny efektywności stylów opisu. W celu dokładniejszej oceny każdego stylu opisu automatów skończonych konieczne jest przeprowadzenie szerszych badań eksperymentalnych na dużej liczbie przykładów testowych.

Podsumujemy krótko wyniki przeprowadzonych badań efektywności stylów opisu automatów skończonych w języku Verilog.

Liczba procesów, czy to trzy, czy dwa, w opisie automatu skończonego (style 1, 2 i 3) nie wpływa na złożoność realizacji ani szybkość działania zaprojektowanego automatu skończonego.

Należy ostrożnie stosować styl opisu z jednym procesem (styl 4), ponieważ w wyniku syntezy otrzymany zostanie automat z rejestrem na wyjściach, co skutkuje generowaniem sygnałów wyjściowych z opóźnieniem o jeden cykl sygnału synchronizacji.

Styl opisu z jednym procesem (styl *Moore_1_process*), w którym funkcje wyjść opisywane są za pomocą instrukcji ciągłego przypisania **assign**, może być wykorzystany wyłącznie do opisu automatu Moore'a.

Styl 5 opisu automatów skończonych z rejestrami na wyjściach można stosować wtedy, gdy jest wskazane umieszczenie rejestrów na wyjściach automatu skończonego. W tym przypadku sygnały wyjściowe będą generowane z opóźnieniem o jeden cykl sygnału synchronizacji w porównaniu ze stylami 1, 2, 3, 6, 7 i 8.

Wykorzystanie instrukcji **case** zamiast instrukcji **if** przy określeniu warunków przejść (styl 6) nie wpływa znacząco na parametry realizowanego automatu skończonego, jednak często upraszcza opis i czyni go bardziej czytelnym.

Sposoby opisu niezawodnych automatów skończonych (style 7 i 8) też znacząco nie pogarszają parametrów projektowanych automatów, lecz istotnie podnoszą niezawodność ich działania. Dlatego naszą rekomendacją przy wyborze sposobu opisu automatów skończonych jest styl 8.

Przedstawione style nie wyczerpują całej różnorodności sposobów opisu automatów skończonych, jakie oferuje język Verilog. Więcej informacji na ten temat można znaleźć w [15]. Czytelnicy mogą również zaproponować własne style opisu automatów skończonych, jednak istotne jest, aby wyniki ich funkcjonalnego i czasowego modelowania zgadzały się ze stylem 1.

10.6. Kodowanie stanów wewnętrznych automatów skończonych

10.6.1. Kodowanie stanów wewnętrznych przy realizacji automatów skończonych na układach FPGA

Wybór sposobu opisu automatu niewątpliwie ma wpływ na złożoność realizacji oraz szybkość działania automatu skończonego, jednakże nie jest on znaczący. W znacznie większym stopniu przyczynia się do tego wybór sposobu kodowania stanów wewnętrznych automatu. Dodatkowo użytkownik ma pełną swobodę wyboru kodów stanów wewnętrznych przy projektowaniu automatu. Jedynym ograniczeniem

przy realizacji automatów na układach FPGA jest to, żeby stan początkowy automatu zawsze przyjmował zerową wartość kodu. Jest to związane z tym, że po włączeniu zasilania wszystkie rejestry FPGA są zerowane, więc automat samoczynnie ustawiany jest w stan początkowy z zerową wartością kodu.

Ogólnie rzecz biorąc, przy realizacji automatu skończonego na FPGA stan początkowy niekoniecznie musi mieć kod zerowy. Jednakże w takim przypadku w projekcie konieczne jest zrealizowanie układu przełączającego automat w stan początkowy po włączeniu zasilania. To z kolei wymaga użycia dodatkowych zasobów logicznych FPGA i może prowadzić do obniżenia szybkości działania automatu.

Systemy projektowania na układach FPGA zwykle oferują różnorodne metody kodowania stanów wewnętrznych automatów. Na przykład w systemie Quartus dostępne są następujące sposoby kodowania stanów automatów skończonych:

- *Auto* – domyślny sposób kodowania ustalony w systemie dla danego typu układów programowalnych;
- *One-Hot* – kodowanie unarne (rysunek 10.23);
- *Gray* – kodowanie w kodzie Graya (rysunek 10.24);
- *Johnson* – kodowanie w kodzie Johnsona (rysunek 10.25);
- *Minimal Bits* – kodowanie kodem dwójkowym minimalnego rozmiaru (rysunek 10.26);
- *Sequential* – sekwencyjne kodowanie dwójkowe (rysunek 10.27);
- *User-Encoded* – kodowanie definiowane przez użytkownika (rysunek 10.28).

	Name	s3	s2	s1	s5	s4	s0
1	s0	0	0	0	0	0	0
2	s4	0	0	0	0	1	1
3	s5	0	0	0	1	0	1
4	s1	0	0	1	0	0	1
5	s2	0	1	0	0	0	1
6	s3	1	0	0	0	0	1

RYS. 10.23. Kodowanie stanów wewnętrznych automatu Moore'a w trybie *One-Hot*

	Name	state_bit_2	state_bit_1	state_bit_0
1	s0	0	0	0
2	s4	0	0	1
3	s5	0	1	1
4	s1	0	1	0
5	s2	1	1	0
6	s3	1	1	1

RYS. 10.24. Kodowanie stanów wewnętrznych automatu Moore'a w trybie *Gray*

	Name	state_bit_2	state_bit_1	state_bit_0
1	s0	0	0	0
2	s4	1	0	0
3	s5	1	1	0
4	s1	1	1	1
5	s2	0	1	1
6	s3	0	0	1

RYS. 10.25. Kodowanie stanów wewnętrznych automatu Moore'a w trybie *Johnson*

	Name	state_bit_2	state_bit_1	state_bit_0
1	s0	0	0	0
2	s4	0	0	1
3	s5	0	1	0
4	s1	0	1	1
5	s2	1	0	0
6	s3	1	0	1

RYS. 10.26. Kodowanie stanów wewnętrznych automatu Moore'a w trybie *Minimal Bits*

	Name	state_bit_2	state_bit_1	state_bit_0
1	s0	0	0	0
2	s4	0	0	1
3	s5	0	1	0
4	s1	0	1	1
5	s2	1	0	0
6	s3	1	0	1

RYS. 10.27. Kodowanie stanów wewnętrznych automatu Moore'a w trybie *Sequential*

	Name	state~6	state~5	state~4
1	s0	0	0	0
2	s4	0	0	1
3	s5	0	1	0
4	s1	1	0	0
5	s2	1	0	1
6	s3	1	1	0

RYS. 10.28. Kodowanie stanów wewnętrznych automatu Moore'a w trybie *User-Encoded*

Aby w systemie Quartus ustawić sposób kodowania stanów automatów skończonych, należy w głównym menu systemu wybrać: **Assignments > Settings**, następnie w otwartym oknie wybrać pole kategorii (**Category**) i zaznaczyć opcję **Compiler Settings**, po czym w prawym okienku kliknąć na **Advanced Settings (Synthesis)**. Pojawi się okno z listą zaawansowanych ustawień syntezy i analizy (**Advanced Analysis**

& **Synthesis Settings**), gdzie należy odszukać parametr **State Machine Processing** i wybrać odpowiedni sposób kodowania. Po dokonaniu wyboru należy zatwierdzić zmiany przez kliknięcie na **OK, Apply, OK**.

Po przeprowadzeniu syntezy w celu sprawdzenia, czy został zastosowany wymagany sposób kodowania stanów wewnętrznych, należy na liście **Table of Contents** wybrać **Analysis & Synthesis > Settings**. W otwartym oknie zostaną wymienione wszystkie parametry syntezy (**Setting**) oraz ich domyślne wartości (**Default Value**).

10.6.2. Sposoby kodowania stanów wewnętrznych automatów skończonych w systemie Quartus

Jak już wspomniano, system Quartus umożliwia wybór spośród różnych metod kodowania stanów wewnętrznych automatów skończonych. Wartość parametru określającego sposób kodowania stanów wewnętrznych automatu domyślnie ustawiona jest w systemie na **Auto**. Domyślny sposób kodowania zależy od rodzaju docelowego układu: dla wszystkich układów FPGA będzie to metoda *One-Hot*, natomiast dla układów CPLD (*Complex Programmable Logic Device*) – metoda *Minimal Bits*. Omówimy cechy każdej z nich.

Metoda *One-Hot* umożliwia kodowanie stanów automatów skończonych w kodzie unarnym. W tym kodowaniu każdy stan automatu jest reprezentowany przez osobny bit, a każde słowo kodowe zawiera tylko jedną jedynkę, która identyfikuje odpowiedni stan automatu. Jednak implementacja metody *One-Hot* w systemie Quartus ma swoje szczególne cechy. Ponieważ stan początkowy automatu powinien mieć kod zerowy, zatem w przypadku każdego kodowania jeden bit zostaje zanegowany – zazwyczaj jest to najmłodszy bit kodu. W efekcie otrzymujemy kodowanie, które jest przedstawione na rysunku 10.23. Zauważmy, że wszystkie kody stanów, poza stanem początkowym, mają dokładnie dwie jedynki: w bicie odpowiadającym danemu stanowi oraz w najmłodszym bicie (zerowym).

Kodowanie unarne zazwyczaj stosuje się w celu zwiększenia szybkości działania automatów skończonych oraz uproszczenia monitorowania aktualnego stanu automatu (w tym przypadku wystarczy testować wartość jednego bitu w rejestrze pamięci). Warto zaznaczyć, że rozmiar bitowy kodu przy kodowaniu unarnym odpowiada liczbie stanów wewnętrznych automatu, oznaczanej jako M . Ogólnie, korzystając z kodu o rozmiarze M , można zakodować 2^M różnych stanów, jednak przy kodowaniu unarnym możliwe jest zakodowanie tylko M stanów. Z tego powodu jest to kod nadmiarowy, w którym $2^M - M$ kombinacji nie jest wykorzystywanych. W wyniku usterki lub błędu w rejestrze pamięci automatu może się pojawić jeden z takich niewykorzystywanych kodów, oznacza to, że automat przechodzi w *stan niedozwolony* (*illegal*). Zachowanie automatu skończonego w takim stanie nie jest zdefiniowane, co może prowadzić do błędnego działania układu. Rozwiązaniem problemu jest projektowanie *niezawodnych automatów skończonych* (*Safe State Machine*). Proces ich syntezy zostanie omówiony w dalszej części rozdziału.

W kodzie Graya dwa sąsiednie słowa kodowe różnią się od siebie tylko na jednym bicie (rysunek 10.24). Ten rodzaj kodowania jest powszechnie wykorzystywany do reprezentowania *łańcuchów przejść bezwarunkowych* w grafach automatów skończonych. Przykładem takiego automatu jest licznik, który składa się z jednego łańcucha przejść bezwarunkowych. Przez kodowanie sąsiednich stanów w kodzie Graya obniża się dynamiczna składowa mocy pobieranej przez zaprojektowany automat skończony. Wynika to z redukcji liczby zmian stanów przerzutników pamięci, ponieważ przy wykorzystaniu kodu Graya na każdym przejściu pomiędzy stanami przełączany jest tylko jeden przerzutnik.

W kodzie Johnsona (rysunek 10.25), podobnie jak w kodzie Graya, dwa sąsiednie słowa kodowe różnią się tylko na jednym bicie. Jednak kod Johnsona jest konstruowany inaczej. Pierwsze słowo kodowe zawiera wszystkie zera. Kolejne sekwencje kodowe generowane są przez przesunięcie poprzedniego kodu o jeden bit w prawo, przy czym bit wpisywany na najstarszą pozycję słowa kodowego jest negacją bitu wychodzącego z pozycji najmłodszej. Za pomocą kodu Johnsona można zakodować $2R$ stanów, gdzie R to liczba bitów rejestru pamięci automatu skończonego. Kod Johnsona ma podobne zastosowanie jak kod Graya, jednak jego realizacja wymaga mniej zasobów logicznych w porównaniu z kodem Graya.

W metodzie *Minimal Bits* (rysunek 10.26) wykorzystywane są kody dwójkowe o minimalnej liczbie bitów wystarczającej do zakodowania wszystkich stanów automatu skończonego:

$$R = \text{int} \log_2 M, \quad (10.7)$$

gdzie: M – liczba stanów; R – rozmiar bitowy kodu; $\text{int}(A)$ – najmniejsza liczba całkowita, większa lub równa A . Przy wykorzystaniu metody *Minimal Bits* do kodowania stanów w systemie Quartus stosowane są specjalne algorytmy, które zapewniają przypisanie kodów dwójkowych stanom automatu w taki sposób, aby maksymalnie uprościć schemat automatu skończonego.

W metodzie *Sequential* (rysunek 10.27) kody stanów automatu są sekwencyjnie kodowane kombinacjami dwójkowymi, które odpowiadają kolejnym liczbom dziesiętnym 0, 1, 2, Przy tym rozmiar bitowy kodu określany jest zgodnie z (10.7)

Przy wyborze sposobu kodowania *User-Encoded* użytkownik przypisuje kody stanów automatu w opisie w języku Verilog. Należy jednak być ostrożnym w przypadku takiej sytuacji, gdy stanowi początkowemu zostanie przypisany kod niezerowy. Wtedy syntezytor systemu Quartus zaneguje we wszystkich pozostałych kodach te bity, którym odpowiadają jedynki w kodzie stanu początkowego. W rezultacie użytkownik otrzyma zupełnie inne kodowanie niż było zaprogramowane. Dlatego przy wyborze tego sposobu kodowania zaleca się zawsze przypisywać stanowi początkowemu wartość zerową kodu.

10.6.3. Kodowanie stanów wewnętrznych automatów skończonych w celu minimalizacji kosztów realizacji

Omówimy metodę kodowania stanów wewnętrznych, której celem jest uproszczenie funkcji przejść automatu skończonego, co w rezultacie prowadzi do minimalizacji kosztów realizacji projektowanego układu. Ta metoda jest dosyć prosta, może być wykonana ręcznie i wykorzystana do kodowania zdefiniowanego przez użytkownika w opisie automatu skończonego w języku Verilog.

Niech automat skończony ma M stanów. W proponowanej metodzie wykorzystywany jest dwójkowy kod minimalnego rozmiaru R , gdzie R określane jest wzorem (10.7). Do kodowania stanów można wykorzystać zbiór K zawierający 2^R kombinacji dwójkowych, za pomocą których należy zakodować M stanów. Zadanie polega na takim kodowaniu stanów wewnętrznych automatu skończonego, żeby otrzymać najprostsze możliwe funkcje przejść automatu.

Tradycyjnie złożoność funkcji logicznych określana jest poprzez liczbę zmiennych w wyrażeniu funkcji albo liczbę składników w postaci prostej funkcji [1]. W naszej metodzie przyjmiemy drugie podejście.

Główną ideą metody jest kodowanie stanów, do których prowadzi najwięcej przejść, za pomocą kombinacji dwójkowych zawierających najmniejszą liczbę jedynek. W ten sposób wyrażenia funkcji przejść zawierają minimalną liczbę składników.

Zastosujemy proponowaną metodę do zakodowania stanów wewnętrznych automatu Moore'a z naszego przykładu. Nasz automat Moore'a ma sześć stanów ($M = 6$), co oznacza, że rozmiar bitowy kodu wynosi $R = 3$. Oznaczmy przez $C(a_i)$ liczbę przejść automatu prowadzących do stanu a_i . Wartości $C(a_i)$ można łatwo określić z tablicy przejść automatu skończonego (tabela 10.2), licząc, ile razy stan a_i występuje w kolumnie a_s tablicy przejść. Obliczone wartości $C(a_i)$ dla automatu Moore'a z naszego przykładu zamieszczone zostały w tabeli 10.5.

TABELA 10.5. Wartości $C(a_i)$ i $K(a_i)$ przy kodowaniu stanów wewnętrznych automatu Moore'a

a_i	$C(a_i)$	$K(a_i)$
s_0	2	000
s_1	1	100
s_2	1	101
s_3	1	110
s_4	3	001
s_5	2	010

Najpierw stanowi początkowemu automatu przypisywany jest kod zerowy. Stan początkowy oraz kod zerowy nie będą już uwzględniane w dalszych rozważaniach.

Następnie z tabeli 10.5 należy wybrać stan a_i o największej wartości $C(a_i) = \max$. Stanowi a_i przypisywany jest kod $K(a_i)$ z najmniejszą liczbą jedynek, wybierany ze zbioru K niewykorzystanych kodów. Stan a_i zostaje zakodowany, a jego kod $K(a_i)$

usuwany jest ze zbioru K . Proces ten należy powtarzać do momentu, aż wszystkie stany zostaną zakodowane.

W naszym przykładzie na początku kod 000 zostaje przypisany stanowi początkowemu s_0 . W dalszej kolejności stanowi s_4 , dla którego $C(a_i) = \max = 3$, przyporządkowany zostaje kod 001. Następny kod 010 z minimalną liczbą jedynek zostaje wybrany do zakodowania stanu s_5 , dla którego $C(a_i) = 2$, kolejne kody przypisywane są stanom s_1 , s_2 i s_4 . Wyniki kodowania stanów wewnętrznych automatu Moore'a pokazane są w tabeli 10.5 (kolumna $K(a_i)$). Takie kodowanie zostało przedstawione w przykładach opisu automatu Moore'a z naszego przykładu.

Czytelnikowi proponujemy zastosowanie omówionej metody kodowania stanów wewnętrznych do automatu Mealy'ego z naszego przykładu.

10.6.4. Porównanie sposobów kodowania stanów wewnętrznych automatów skończonych

Aby ocenić, która z metod kodowania stanów wewnętrznych automatów skończonych jest najbardziej efektywna, przeprowadzono badania w systemie Quartus Prime wersja 18.1 dla różnych rodzin układów FPGA. Wyniki syntezy zostały ocenione pod względem złożoności realizacji (L) oraz szybkości działania (F) zaprojektowanych układów. Opisy automatów skończonych z naszych przykładów w języku Verilog wykonane były w stylu 1. Wyniki badań przedstawiono w tabeli 10.6.

Dla automatu Mealy'ego można zidentyfikować trzy grupy, które osiągają dokładnie takie same wyniki pod względem złożoności i szybkości. Pierwsza grupa to *Auto* i *One-Hot*, druga to *Gray* i *Johnson*, a trzecia to *Minimal Bits* i *Sequential*. Metodę *User-Encoded* można rozważać osobno.

Nie jest zaskoczeniem, że metody kodowania *Auto* i *One-Hot* wykazują takie same wyniki, ponieważ w systemie Quartus dla układów FPGA domyślnie wykorzystywana jest metoda *One-Hot*. Prezentuje ona najgorsze wyniki według złożoności realizacji. To było do przewidzenia, ponieważ rozmiar bitowy kodu dla tej metody jest największy. Nieco zaskakujące jest to, że metoda ta ma także najgorsze wyniki pod względem szybkości działania. Można to jednak wyjaśnić faktem, że w przypadku kodowania *One-Hot* w istocie stosowany jest zmodyfikowany kod unarny, w którym każdy kod zawiera dwie jedynki, z wyjątkiem stanu początkowego, który jest kodowany wszystkimi zerami.

Wniosek. Metoda kodowania *One-Hot* nie zawsze pozwala projektować najszybsze automaty skończone.

Pozostałe metody kodowania stanów wewnętrznych automatu Mealy'ego dla naszego przykładu wykazują bardzo zbliżone wyniki. Jest to zrozumiałe, gdyż automat ten jest bardzo prosty, ma jedynie cztery stany, dla których we wszystkich metodach stosowany jest dwubitowy kod. Dodatkowo wszystkie cztery kombinacje dwójkowe zostały wykorzystane. W takiej sytuacji istnieje niewiele możliwości przy przypisywaniu kodów stanom. Warto zauważyć, że metody drugiej

grupy (*Gray* i *Johnson*) wykazały największą szybkość działania dla rodzin układów MAX II i MAX V, metody trzeciej grupy (*Minimal Bits* i *Sequential*) dla rodzin MAX 10 i Cyclone IV, a metoda *User-Encoded* dla rodzin MAX II i MAX 10.

TABELA 10.6. Wyniki badań sposobów kodowania stanów wewnętrznych automatów skończonych

Sposób kodowania	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy														
Auto	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
One-Hot	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
Gray	7	646	7	384	8	1071	5	1422	7	1228	4	730	7	1221
Johnson	7	646	7	384	8	1071	5	1422	7	1228	4	730	7	1221
Minimal Bits	7	633	7	383	8	1073	5	1422	7	1232	4	730	7	1221
Sequential	7	633	7	383	8	1073	5	1422	7	1232	4	730	7	1221
User-Encoded	7	646	7	383	7	1073	5	1422	7	1229	4	730	7	1221
Best Mealy	7	646	7	384	7	1073	5	1422	7	1232	4	730	7	1221
Moore														
Auto	10	504	10	276	12	748	9	1175	11	833	6	669	11	902
One-Hot	10	504	10	276	12	748	9	1175	11	833	6	669	11	902
Gray	8	409	8	212	8	631	6	1134	7	789	4	686	7	787
Johnson	8	503	8	300	9	666	6	1134	8	836	4	670	8	836
Minimal Bits	8	416	8	234	8	582	5	1171	7	737	4	685	7	725
Sequential	8	416	8	234	8	582	5	1171	7	737	4	685	7	725
User-Encoded	7	512	7	250	7	664	4	1148	7	837	4	678	7	833
Best Moore	7	512	7	300	7	748	4	1175	7	837	4	686	7	902

W przypadku automatu Moore'a zaobserwowano dokładnie takie same wyniki (pod względem złożoności oraz szybkości działania) dla metod *Auto* i *One-Hot*, a także dla metod *Minimal Bits* i *Sequential*. Jednakże wyniki metod *Gray* i *Johnson* różnią się od siebie.

Dla automatu Moore'a metoda *One-Hot* zademonstrowała najgorsze wyniki pod względem złożoności realizacji układu, co było do przewidzenia. Metoda *User-Encoded* okazała się najlepszą pod względem złożoności realizacji dla rodzin układów MAX II, MAX V, MAX 10 i Arria II. Pozostałe metody pod względem kosztów realizacji pokazały bardzo podobne wyniki.

Analiza szybkości działania automatu Moore'a wykazała, że metoda *One-Hot* zaprezentowała lepsze wyniki dla rodzin układów MAX 10, Arria II i Cyclone 10. Kodowanie w kodzie Graya osiągnęło najszybsze wyniki dla rodziny Cyclone V,

a metoda *Johnson* – dla rodziny MAX 10. Metoda *User-Encoded* okazała się najlepsza pod względem szybkości dla rodzin MAX II i Cyclone IV.

Dla automatu Moore’a metody *Minimal Bits* i *Sequential* pokazały uśrednione wyniki zarówno pod względem złożoności realizacji, jak i szybkości działania.

Przy porównywaniu najlepszych wyników automatu Mealy’ego (*Best Mealy*) oraz automatu Moore’a (*Best Moore*) można zauważyć, że pod względem złożoności realizacji są one prawie identyczne, z wyjątkiem rodziny Arria II, gdzie automat Moore’a zaprezentował się lepiej (cztery elementy logiczne zamiast pięciu). Jednak pod względem szybkości działania automat Mealy’ego okazał się lepszy dla wszystkich rodzin układów FPGA.

Wniosek. Do projektowania szybkich automatów skończonych najlepiej nadaje się model automatu Mealy’ego, do którego należy zastosować wszystkie możliwe sposoby kodowania stanów wewnętrznych, po czym z otrzymanych wyników wybrać realizację automatu o maksymalnej szybkości działania.

Przeprowadzone badania nad prostym projektem automatu wykazały, że nie istnieje jedna bezkonkurencyjna metoda kodowania stanów wewnętrznych automatów skończonych, która zapewniłaby najlepsze wyniki pod względem zarówno złożoności, jak i szybkości działania dla wszystkich rodzin układów FPGA.

10.7. Projektowanie bezpiecznych automatów skończonych

W systemie Quartus dostępny jest specjalny tryb syntezy, który umożliwia projektowanie *niezawodnych automatów skończonych* (*Safe State Machines*). Domyślnie tryb ten jest wyłączony. Aby go aktywować, należy przejść do opcji **Assignment > Settings > Compiler Settings > Advanced Settings (Synthesis)** i ustawić parametr **Safe State Machines** na wartość **On**. Po włączeniu tego trybu syntezy system automatycznie wzbogaca projekt o dodatkową logikę, która wykrywa *niedozwolone stany* (*illegal states*) automatu skończonego i zapewnia bezwarunkowe przejście w stan początkowy z każdego niedozwolonego stanu.

Należy podkreślić, że włączenie opcji **Safe State Machines** może znacząco zwiększyć złożoność realizacji automatu skończonego. Zazwyczaj opcję tę aktywuje się w przypadku, gdy sygnały sterujące automatem skończonym pochodzą z innej domeny synchronizacji lub gdy projekt wykorzystuje asynchroniczne dane wejściowe. Jednak firma Intel zaleca w takich przypadkach zamiast opcji **Safe State Machines** wykorzystać dodatkowe rejestry synchronizacji w celu przywiązania sygnałów wejściowych do sygnału synchronizacji automatu skończonego.

Zwróćmy uwagę, że w języku Verilog istnieje możliwość opisanie zachowania automatu w niedozwolonych stanach bez konieczności korzystania z opcji **Safe State Machines**, co zostało przedstawione w stylach opisu 7 i 8 w podrozdziale 10.5.5.

W celu oceny skuteczności opcji **Safe State Machines** przeprowadzono syntezę projektów *Mealy_style_1* oraz *Moore_style_1* zarówno z ustawioną opcją **Safe State Machines**, jak i bez niej. Wyniki tych badań zostały przedstawione w tabeli 10.7.

TABELA 10.7. Wyniki syntezy projektów *Mealy_style_1* i *Moore_style_1* bez opcji oraz z opcją **Safe State Machines**

Projekt	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy_style_1	10	539	10	310	11	760	7	1241	10	935	5	701	10	891
Safe Mealy_style_1	10	413	10	219	11	578	7	1125	10	710	6	646	11	578
Moore_style_1	10	504	10	276	12	748	9	1175	11	833	6	669	11	902
Safe Moore_style_1	15	289	15	160	16	418	11	839	15	451	9	479	15	521

Analiza danych z tabeli 10.7 wykazuje, że dla automatu Mealy’ego złożoność realizacji praktycznie nie uległa zmianie, z wyjątkiem rodzin Cyclone V i Cyclone 10. Jednakże zdecydowanie spadła szybkość działania. Natomiast w przypadku automatu Moore’a włączenie opcji **Safe State Machines** przyniosło istotne zwiększenie złożoności realizacji oraz obniżenie szybkości działania.

Wniosek. Do projektowania niezawodnych automatów skończonych zamiast opcji **Safe State Machines** lepiej stosować style 7 lub 8 opisu automatów skończonych w języku Verilog.

10.8. Wykorzystanie wartości zmiennych wejściowych i wyjściowych do kodowania stanów automatów skończonych

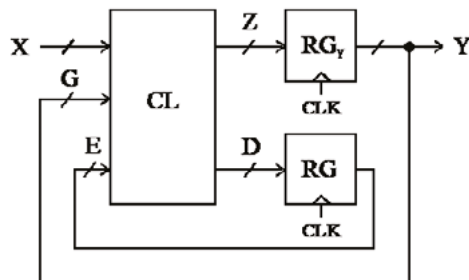
W podrozdziale 10.1 przedstawiono sześć struktur automatów skończonych: automaty klas A, B, C, D, E i F. Dotychczas omówiono jedynie dwa modele automatów: automat Mealy’ego, zwany również automatem klasy A (rysunek 10.1), oraz automat Moore’a, określany jako automat klasy B (rysunek 10.1b). W tym podrozdziale rozpatrzone zostaną zagadnienia dotyczące syntezy pozostałych klas automatów – C, D, E i F [6].

10.8.1. Synteza automatów klasy C

W praktyce często projektuje się automaty w taki sposób, aby wektor wyjściowy jednoznacznie definiował stan wewnętrzny automatu skończonego [6]. Przykładem tego rodzaju automatów są liczniki binarne: każdy zestaw sygnałów wyjściowych precyzyjnie określa stan wewnętrzny licznika, gdzie wszystkie kombinacje sygnałów wyjściowych są unikatowe. W takiej sytuacji wartości wektorów wyjściowych można przypisać kodom stanów wewnętrznych automatu. W efekcie uzyskujemy automat klasy C, który charakteryzuje się strukturą zaprezentowaną na rysunku 10.1c, a jego działanie opisują równania (10.3).

Podobne właściwości mogą występować w dowolnym automacie skończonym (nie tylko w licznikach), mogą one również dotyczyć części układu (wybranego podzbioru stanów). Warto zauważyć, że automat klasy C jest automatem Moore'a.

Na rysunku 10.1c przedstawiona jest struktura idealnego automatu klasy C, w którym we wszystkich jego stanach generowane są wzajemnie ortogonalne wektory wyjściowe. Jednak w praktyce, poza przypadkiem liczników, taka sytuacja występuje rzadko. Na przykład zestawy sygnałów wyjściowych mogą się powtarzać w różnych stanach, wektory wyjściowe niekoniecznie są wzajemnie ortogonalne lub też liczba sygnałów wyjściowych może być niewystarczająca do zakodowania wszystkich stanów wewnętrznych automatu skończonego. We wszystkich wymienionych sytuacjach wektory wyjściowe nie mogą być używane do kodowania stanów wewnętrznych, ponieważ to naruszyłoby deterministyczne zachowanie automatu skończonego. Aby rozwiązać ten problem, można wprowadzić dodatkowe bity kodów stanów wewnętrznych. W ten sposób otrzymujemy strukturę automatu klasy C przedstawioną na rysunku 10.29.



RYS. 10.29. Struktura automatu klasy C

Układ kombinacyjny CL na rysunku 10.29 zawiera logikę kombinacyjną automatu skończonego. Pamięć automatu jest zrealizowana przy użyciu dwóch rejestrów: RG_Y i RG. Na wyjściach rejestru RG_Y generowane są wartości zmiennych ze zbioru Y, które są przekazywane poprzez obwody sprzężeń zwrotnych na wejścia układu kombinacyjnego CL jako zmienne ze zbioru G. Rejestr RG przechowuje wartości dodatkowych

bitów kodów stanów wewnętrznych. Na jego wejścia podawane są wartości funkcji przełączania elementów pamięci ze zbioru D . Na wyjściach rejestru RG generowane są wartości funkcji sprzężenia zwrotnego, które tworzą zbiór E .

Kod stanu wewnętrznego automatu na rysunku 10.29 jest tworzony przez konkatencję sekwencji wartości zmiennych zbiorów G i E , zatem $state = \{G, E\}$. Układ kombinacyjny CL, bazując na kodzie stanu wewnętrznego oraz wartościach zmiennych wejściowych ze zbioru X , generuje wartości funkcji dla zbiorów Z i D , które są następnie przekazywane odpowiednio na wejścia rejestrów RG_Y i RG .

Kodowanie stanów wewnętrznych automatów klasy C odbywa się w sposób następujący. Definiowana jest macierz boolowska $\mathbf{W}$ (tabela 10.8). Jej wiersze odpowiadają stanom wewnętrznym automatu skończonego, natomiast kolumny odpowiadają zmiennym wyjściowym automatu. W każdym wierszu macierzy w kolumnach $g_{N-1}, \dots, g_0$ zapisywane są zmienne wyjściowe generowane w danym stanie. Przyjmuje się tutaj, że N to liczba zmiennych wejściowych, M to liczba stanów wewnętrznych automatu, a R to liczba dodatkowych bitów kodu stanu.

TABELA 10.8. Uogólniona macierz $\mathbf{W}$ do kodowania stanów wewnętrznych automatu klasy C

a_i	$Y(G)$			E		
	g_{N-1}	...	g_0	e_{R-1}	...	e_0
s_0						
...						
s_{M-1}						

Zadanie polega na dodaniu do macierzy $\mathbf{W}$ minimalnej liczby R kolumn $e_{R-1}, \dots, e_0$ i określeniu ich wartości w taki sposób, aby wszystkie wiersze macierzy $\mathbf{W}$ były wzajemnie ortogonalne. Wartości kolumn macierzy $\mathbf{W}$ odpowiadające zmiennym wyjściowym zbioru Y traktowane są jako wartości zmiennych zbioru G , a wartości dodatkowych kolumn jako wartości zmiennych zbioru E . Następnie zawartości wierszy macierzy $\mathbf{W}$ wykorzystywane są do kodowania stanów wewnętrznych automatu klasy C, którego struktura podana jest na rysunku 10.29.

Aby rozwiązać postawione zadanie, należy utworzyć graf H ortogonalności wierszy macierzy $\mathbf{W}$ [5], który następnie trzeba podzielić na minimalną liczbę Q pełnych podgrafów $H_1, \dots, H_Q$. Są one kodowane kodami dwójkowymi o minimalnym rozmiarze $R = \text{int} \log_2 Q$, przy czym podgraf zawierający wierzchołek odpowiadający stanowi początkowemu kodowany jest kodem zerowym. Kody podgrafów $H_1, \dots, H_Q$ stanowią wartości bitów dodatkowych kodów stanów wewnętrznych, czyli zmienne zbioru E . Szczegółowy opis metod kodowania stanów wewnętrznych automatów skończonych klas C, D, E i F znajduje się w [5].

Zastosujemy omówioną metodę syntezy automatu klasy C do automatu Moore'a z naszego przykładu. Macierz $\mathbf{W}$ do kodowania stanów wewnętrznych automatu klasy C przedstawiona została na w tabeli 10.9.

TABELA 10.9. Macierz **W** dla przykładu kodowania stanów wewnętrznych automatu klasy C

a_i	$Y(G)$		
	g_2	g_1	g_0
s_0	0	0	0
s_1	0	1	1
s_2	1	0	1
s_3	0	0	1
s_4	1	0	0
s_5	0	1	0

Z tabeli 10.9 wynika, że wszystkie kody stanów wewnętrznych są wzajemnie ortogonalne, więc dodatkowe bity do ortogonalizacji kodów nie są potrzebne. Dodatkowo w naszym przykładzie stan s_0 został zakodowany kodem zerowym. W związku z tym kodowanie stanów wewnętrznych naszego automatu klasy C jest kompletne. Kody stanów wewnętrznych automatu przyjmują wartości zapisane w wierszach tabeli 10.9.

Uwzględniając otrzymane kody, opis w języku Verilog automatu klasy C z naszego przykładu można przedstawić następująco:

```

module Moore_C(
    input clk,reset,
    input [2:0] x,
    output [2:0] y);

    reg [2:0] state, next;

    localparam [2:0]
        s0=3'b000,
        s1=3'b011,
        s2=3'b101,
        s3=3'b001,
        s4=3'b100,
        s5=3'b010;

    always @(posedge clk, negedge reset)
        if (~reset)    state <= s0;
        else           state <= next;

    always @(*)
        case (state)
            s0:    next = s1;
            s1:    if (~x[0])                next = s2;

```

```

        else if (~x[1] && x[0])          next = s3;
        else if (~x[2] && x[1] && x[0])  next = s4;
        else                            next = s5;
s2:   if (~x[2])      next = s4;
        else          next = s5;
s3:   next = s4;
s4:   next = s0;
s5:   next = s0;
    endcase

    assign y = state;
endmodule

```

Zamieszczony powyżej opis różni się od projektu *Moore_style_1* innym kodowaniem stanów oraz wykorzystaniem zamiast drugiego procesu instrukcji **assign** **y = state**.

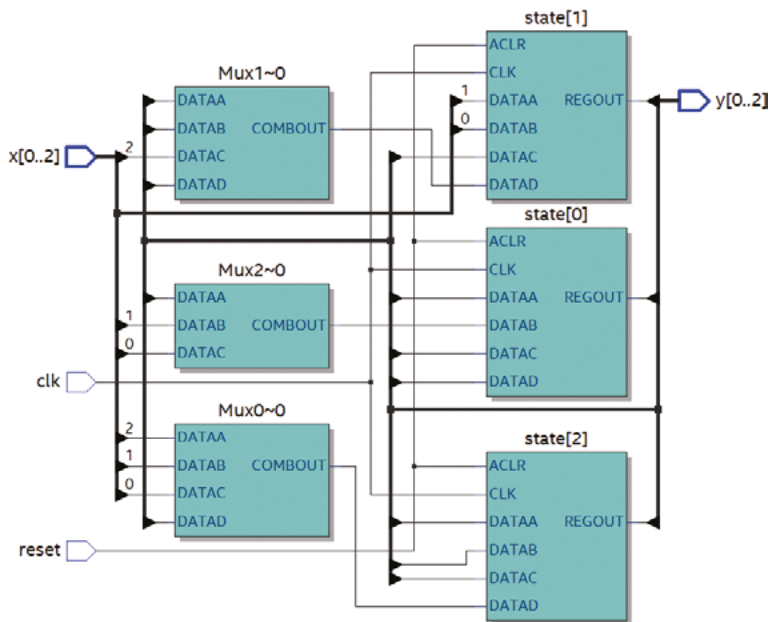
Porównamy opis automatu w języku Verilog ze strukturą przedstawioną na rysunku 10.29. W projekcie *Moore_C* zbiór funkcji *Z* jest reprezentowany przez zmienną *next*, natomiast zbiór *G* przez zmienną *state*. Ponieważ w naszym przykładzie nie było konieczności dodawania dodatkowych bitów kodu dla ortogonalizacji wierszy macierzy *W*, w strukturze na rysunku 10.29 w naszym automacie nie będą obecne rejestr *RG* oraz zbiory *D* i *E*. W związku z tym kody stanów wewnętrznych automatu są jedynie określone przez wartości zmiennych należących do zbioru *G* (w opisie są to wartości zmiennej *state*).

Jeśli nie zostaną zmienione ustawienia parametrów przed kompilacją projektu, wynik syntezy automatu klasy *C* będzie identyczny z wynikiem syntezy automatu Moore’a przedstawionego na rysunku 10.11. Spowodowane będzie to domyślnym użyciem przez system Quartus kodowania *One-Hot*. Aby rzeczywiście zaimplementować automat klasy *C*, konieczna jest zmiana opcji **State Machine Processing** na wartość **User-Encoded**. Można to zrobić poprzez: **Assignment > Settings > Compiler Settings > Advanced Settings (Synthesis)**.

Jednakże w przypadku naszego automatu nawet po przełączeniu opcji kodowania **State Machine Processing** na wartość **User-Encoded** wynik syntezy będzie identyczny z tym, który został przedstawiony na rysunku 10.11 – kodowanie stanów wewnętrznych będzie poprawne, dokładnie takie same jak w kodzie projektu. Można to potwierdzić, sprawdzając wartości kodów stanów w panelu **Encoding** w oknie narzędzia przeglądania wyników syntezy automatów skończonych **State Machine Viewer**.

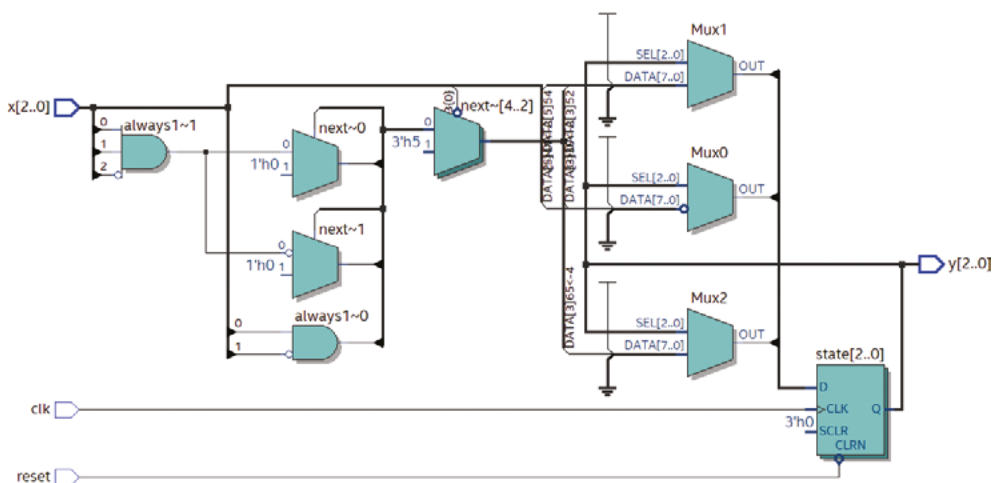
Aby potwierdzić, że w wyniku syntezy otrzymano automat klasy *C*, konieczne jest przeanalizowanie mapy technologicznej projektu (rysunek 10.30).

Na rysunku 10.30 możemy zauważyć, że wartości zmiennych wyjściowych generowane są na wyjściach elementów pamięci automatu skończonego.



RYS. 10.30. Mapa technologiczna projektu *Moore_C* z opcją **State Machine Processing** ustawioną na wartość **User-Encoded**

Innym sposobem realizacji automatu Moore'a klasy C w systemie Quartus jest włączenie opcji **Extract Verilog State Machines** (wyodrębnienie opisu automatu z kodu w języku Verilog), która domyślnie zawsze jest wyłączona. Wynik syntezy automatu Moore'a klasy C z włączoną opcją **Extract Verilog State Machines** możemy zaobserwować na rysunku 10.31.



RYS. 10.31. Wynik syntezy projektu *Moore_C* na poziomie przesłań międzyrejestrów z włączoną opcją **Extract Verilog State Machines**

Na rysunku 10.31 można zauważyć, że wartości zmiennych wyjściowych są rzeczywiście generowane na wyjściach rejestru pamięci automatu skończonego, co oznacza, że zaprojektowany schemat dokładnie odpowiada strukturze na rysunku 10.1c.

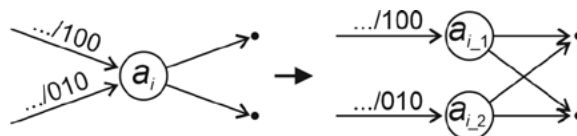
Wyniki funkcjonalnej i czasowej symulacji projektu *Moore_C* dokładnie pokrywają się z analogicznymi wynikami projektu *Moore_style_1* zamieszczonymi na rysunkach 10.15 i 10.16.

Pozostaje jeszcze kwestia zerowej wartości kodu stanu początkowego dla automatów klasy C przy realizacji na układach FPGA. Zazwyczaj w automatach Moore'a w stanie początkowym generowany jest zerowy zestaw sygnałów wyjściowych. W takiej sytuacji nie pojawiają się żadne problemy z realizacją automatu klasy C. Jednakże jeśli w stanie początkowym automatu Moore'a muszą być sformowane niezerowe sygnały wyjściowe, synteza automatu klasy C na FPGA staje się niemożliwa bez wprowadzenia dodatkowego stanu z zerowym zestawem sygnałów na wyjściach i zdefiniowania go jako stanu początkowego. Wprowadzenie dodatkowego stanu z zerowym wektorem wyjściowym zmienia funkcjonowanie automatu skończonego, więc powinno być wykonane bardzo ostrożnie.

10.8.2. Synteza automatów klasy D

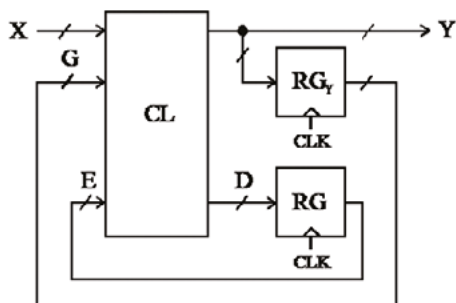
Często się zdarza, że w automatach skończonych generowane są identyczne sygnały wyjściowe przy przejściach do tego samego stanu. Przykładem takich automatów mogą być liczniki realizowane w postaci automatów Mealy'ego. W przypadku, gdy ta sytuacja występuje dla wszystkich stanów wewnętrznych automatu skończonego, a wektory wyjściowe są wzajemnie ortogonalne, otrzymujemy idealny automat klasy D, którego struktura pokazana jest na rysunku 10.1d. Funkcjonowanie tego typu automatu można opisać za pomocą równań (10.4).

Jeśli do pewnego stanu a_i prowadzą przejścia, na których generowane są różne sygnały wyjściowe, taki stan można rozszcześcić na kilka stanów tak, aby na przejściach do każdego z nich generowany był ten sam zestaw sygnałów wyjściowych (rysunek 10.32). Operacja rozszczepienia stanów wewnętrznych automatu realizuje ekwiwalentne przekształcenie automatu skończonego, co nie zmienia algorytmu jego funkcjonowania. Podobna operacja wykonywana jest przy zamianie automatu Mealy'ego na automat Moore'a [5].



RYS. 10.32. Rozszczepienie stanów wewnętrznych przy syntezie automatów klasy D

Nawet po wykonaniu operacji rozszczepienia stanów wewnętrznych zestawy sygnałów wyjściowych mogą się nie nadawać do kodowania stanów wewnętrznych: na przejściach do różnych stanów mogą być generowane takie same zbiory sygnałów wyjściowych lub wektory wyjściowe nie są wzajemnie ortogonalne. W takiej sytuacji, podobnie jak w przypadku syntezy automatu klasy C, konieczne jest wprowadzenie dodatkowych bitów kodów stanów wewnętrznych, a struktura automatu klasy D przyjmuje postać zaprezentowaną na rysunku 10.33.



RYS. 10.33. Struktura automatu klasy D

Metoda syntezy automatów klasy D została szczegółowo opisana w [6]. Zastosujemy ją do syntezy automatu Mealy’ego z naszego przykładu. Najpierw dla każdego stanu wewnętrznego a_i utworzymy zbiór $Y(a_i)$ zestawów zmiennych wyjściowych generowanych na przejściach do stanu a_i (tabela 10.10).

TABELA 10.10. Zbiory $Y(a_i)$ wektorów wyjściowych generowanych na przejściach do stanów automatu Mealy’ego z przykładu

a_i	$Y(a_i)$
s_0	{100,010}
s_1	{011}
s_2	{101}
s_3	{001}

Analizując tabelę 10.10, zauważamy, że na przejściach do stanu s_0 generowane są dwa zestawy sygnałów wyjściowych: 100 i 010. Zgodnie z rysunkiem 10.32 stan s_0 zostaje rozszczepiony na dwa stany: s_{0-1} oraz s_{0-2} . Jeden z nich może zostać wybrany jako stan początkowy – s_{0-1} lub s_{0-2} . Niech stanem początkowym będzie stan s_{0-1} . Tablica przejść automatu skończonego po rozszczepieniu stanu s_0 zamieszczona została w tabeli 10.11.

TABELA 10.11. Tablica przejść automatu po rozszczepieniu stanu s_0 przy syntezie automatu klasy D

a_m	$X(a_m, a_s)$	$Y(a_m, a_s)$	a_s
$s_{0,1}$	---	011	s_1
$s_{0,2}$	---	011	s_1
s_1	--0	101	s_2
s_1	-01	001	s_3
s_1	011	100	$s_{0,1}$
s_1	111	010	$s_{0,2}$
s_2	0--	100	$s_{0,1}$
s_2	1--	010	$s_{0,2}$
s_3	---	100	$s_{0,1}$

Stany wewnętrzne automatu klasy D są kodowane w podobny sposób, jak w automacie klasy C. W tym celu tworzona jest macierz **W** (tabela 10.12).

TABELA 10.12. Macierz **W** do kodowania stanów wewnętrznych automatu klasy D

a_i	$Y(G)$		
	g_2	g_1	g_0
$s_{0,1}$	1	0	0
$s_{0,2}$	0	1	0
s_1	0	1	1
s_2	1	0	1
s_3	0	0	1

Z tabeli 10.12 wynika, że wszystkie kody stanów wewnętrznych są wzajemnie ortogonalne, więc nie ma potrzeby wprowadzenia dodatkowych bitów kodów stanów. Automat Mealy'ego klasy D może być zatem opisany w języku Verilog w następujący sposób:

```

module Mealy_D(
  input clk,reset,
  input [2:0] x,
  output [2:0] y);

  reg [2:0] state, next;

  localparam [2:0]
    s0_1= 3'b100,

```

```

s0_2= 3'b010,
s1=   3'b011,
s2=   3'b101,
s3=   3'b001;

always @(posedge clk, negedge reset)
  if (~reset)   state <= s0_1;
  else         state <= next;

always @(*)
  case (state)
    s0_1: next = s1;
    s0_2: next = s1;
    s1:   if (~x[0])                next = s2;
          else if (~x[1] && x[0])    next = s3;
          else if (~x[2] && x[1] && x[0]) next = s0_1;
          else                     next = s0_2;
    s2:   if (~x[2])    next = s0_1;
          else         next = s0_2;
    s3:   next = s0_1;
    default: next = s0_1;
  endcase

assign y = next;
endmodule

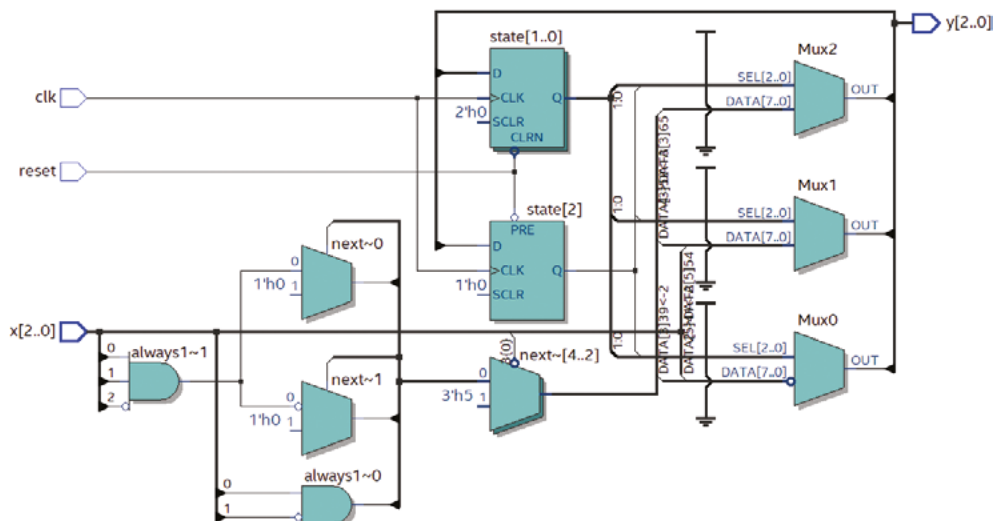
```

W przedstawionym opisie automatu zmienna *next* odpowiada zbiorowi *Y* na rysunku 10.33, natomiast zmienna *state* – zbiorowi *G*. Ponieważ w naszym przykładzie nie zostały wprowadzone dodatkowe bity kodów stanów, struktura automatu na rysunku 10.33 nie będzie zawierała rejestru *RG*, a także zbiorów *D* i *E*.

Podczas przeprowadzenia syntezy projektu *Mealy_D* w systemie Quartus konieczne jest włączenie opcji **Extract Verilog State Machine**, która domyślnie zawsze jest wyłączona. Wynik syntezy projektu *Mealy_D* zaprezentowany został na rysunku 10.34.

Na rysunku 10.34 można zauważyć, że wartości funkcji wyjściowych są podawane na wejścia rejestru pamięci automatu skończonego, co jest w pełni zgodne ze strukturą zaprezentowaną na rysunku 10.33.

Przy syntezie automatu klasy *D* zamiast włączenia opcji **Extract Verilog State Machine** można ustawić parametr syntezy **State Machine Processing** na wartość **User-Encoded**. Jednak w systemie Quartus syntezy automatycznie neguje bity kodów, które odpowiadają jedynie w kodzie stanu początkowego. W takim przypadku efektywność modelu automatu klasy *D* może zostać obniżona.



RYS. 10.34. Wynik syntezy projektu *Mealy_D* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Mealy_D* nie różnią się od analogicznych wyników projektu *Mealy_style_1* przedstawionych na rysunkach 10.9 i 10.10.

10.8.3. Synteza automatów klasy D₁ bez rozszczepienia stanów wewnętrznych

Przy projektowaniu automatu klasy D o strukturze przedstawionej na rysunku 10.33 można przyjąć, że to zestaw sygnałów wyjściowych określa kod następnego stanu, a nie odwrotnie. Ta koncepcja całkowicie odpowiada strukturze przedstawionej na rysunku 10.33. Rejestr RG_Y przechowuje część kodu stanu, która jest określana zestawem sygnałów wyjściowych, natomiast rejestr RG zawiera dodatkową część kodu, służącą do rozróżnienia stanów wewnętrznych, czyli do ortogonalizacji kodów. Pełny kod stanów wewnętrznych tworzony jest przez konkatencję wektorów G i E , co można zapisać jako $state = \{G, E\}$. Układ kombinacyjny CL generuje wartości zmiennych zbiorów Y i D .

Automat skończony klasy D, projektowany bez rozszczepienia stanów wewnętrznych, będziemy nazywać automatem klasy D₁.

Przeanalizujemy teraz metodę syntezy automatu klasy D₁. W naszym przykładzie można wyznaczyć następujące części kodów stanów wewnętrznych, określone sygnałami wyjściowymi:

- s_0 : --0,
- s_1 : 011,
- s_2 : 101,
- s_3 : 001.

Ponieważ wszystkie kody są wzajemnie ortogonalne, nie ma potrzeby wprowadzania dodatkowych bitów kodu. Z tego powodu w schemacie nie występują rejestr RG, a także zbiory *D* i *E*. Metodę syntezy automatu klasy *D_1* dla naszego przykładu można opisać w języku Verilog w następujący sposób:

```

module Mealy_D_1(
  input clk,reset,
  input [2:0] x,
  output reg [2:0] y);

  reg [2:0] g;

  localparam [2:0]
    s0=3'bxx0,
    s1=3'b011,
    s2=3'b101,
    s3=3'b001;

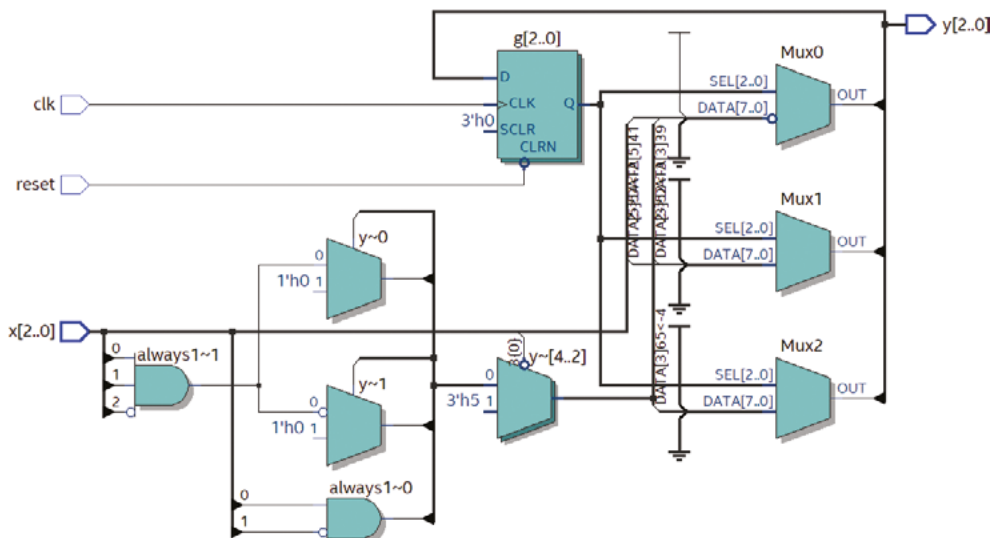
  always @(posedge clk, negedge reset)
    if (~reset)    g <= 3'b000;
    else          g <= y;

  always @(*)
    casex (g)
      s0:    y = 3'b011;
      s1:    if (~x[0])          y = 3'b101;
             else if (~x[1] && x[0]) y = 3'b001;
             else if (~x[2] && x[1] && x[0]) y = 3'b100;
             else                y = 3'b010;
      s2:    if (~x[2])    y = 3'b100;
             else        y = 3'b010;
      s3:    y = 3'b100;
      default: y = 3'b000;
    endcase
endmodule

```

Ponieważ niektóre bity stanów wewnętrznych mają nieokreślone wartości *x*, w opisie automatu została użyta instrukcja **case**x zamiast **case**.

Synteza projektu *Mealy_D_1* również jest przeprowadzana z włączoną opcją **Extract Verilog State Machine**. Wynik syntezy projektu *Mealy_D_1* ukazany jest na rysunku 10.35.



RYS. 10.35. Wynik syntezy projektu *Mealy_D_1* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Mealy_D_1* całkowicie się zgadzają z wynikami projektu *Mealy_style_1*, które zostały przedstawione na rysunkach 10.9 i 10.10.

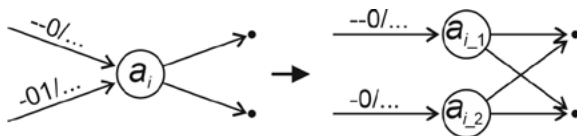
Należy zauważyć, że w projekcie *Mealy_D_1* stan początkowy s_0 jest kodowany kodem xx0, co umożliwia wykorzystanie kombinacji 000, która pojawia się po włączeniu zasilania.

10.8.4. Synteza automatów klasy E

W omawianych wcześniej automatach klas C i D do kodowania stanów wewnętrznych wykorzystano wartości zmiennych wyjściowych. Podobnie jako kody stanów wewnętrznych (lub określoną część kodu) można wykorzystać zestawy zmiennych wejściowych automatu, co zostało zrealizowane w modelach automatów klas E i F.

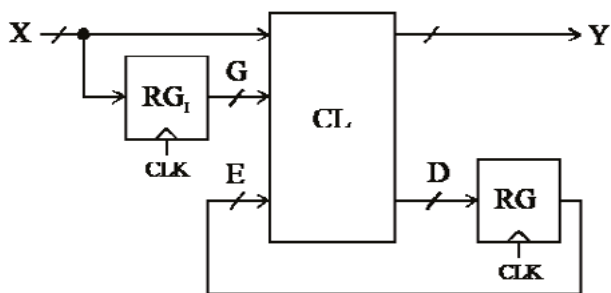
Czasem w automatach skończonych Mealy'ego może się zdarzyć, że przejścia do tego samego stanu są inicjowane tą samą kombinacją zmiennych wejściowych. Gdy zachodzi to dla wszystkich stanów, a wszystkie wektory wejściowe są wzajemnie ortogonalne, otrzymujemy idealny automat klasy E. Jego struktura została przedstawiona na rysunku 10.1e, a funkcjonowanie można opisać za pomocą równań (10.5).

W praktyce powyższe warunki rzadko są spełnione w idealny sposób. W sytuacji, gdy przejścia do pewnego stanu a_i inicjowane są różnymi kombinacjami sygnałów wejściowych, stan a_i można rozszcześcić na kilka stanów tak, żeby przejścia do każdego z nich uwarunkowane były tym samym zestawem zmiennych wejściowych (rysunek 10.36).



RYS. 10.36. Rozszczepienie stanów wewnętrznych przy syntezie automatów klasy E

Jednak nawet po rozszczepieniu takich stanów wektory wejściowe mogą się nie nadawać do kodowania stanów wewnętrznych automatu, ponieważ nie są wzajemnie ortogonalne. Ten problem rozwiązywany jest poprzez wprowadzenie dodatkowych bitów kodu, podobnie jak w przypadku syntezy automatów klas C i D. W rezultacie struktura automatu klasy E sprowadza się do postaci ukazanej na rysunku 10.37.



RYS. 10.37. Struktura automatu klasy E

Powyższa struktura opiera się na dwóch rejestrach: RG_i i RG . Rejestr RG_i pełni funkcję bufora wejściowego, przechowując wartości zmiennych wejściowych ze zbioru X . Jeśli automat już dysponuje buforem wejściowym, będzie on spełniał funkcję rejestru RG_i . Na wyjściach rejestru RG_i generowane są wartości zmiennych zbioru G . Rejestr RG , analogicznie do automatów klas C i D, przechowuje dodatkowe bity kodów wewnętrznych automatu skończonego.

Kod stanu wewnętrznego automatu przedstawionego na rysunku 10.37 jest wynikiem konkatencji dwóch zestawów zmiennych zbiorów G oraz E , czyli $state = \{G, E\}$. Układ kombinacyjny CL , na podstawie kodu stanu wewnętrznego $\{G, E\}$ i wartości zmiennych wejściowych ze zbioru X , generuje wartości funkcji wyjść ze zbioru Y oraz dodatkową część kodu następnego stanu, czyli wartości zmiennych zbioru D .

Metoda syntezy automatów klasy E została szczegółowo opisana w [6]. Teraz zastosujemy ją do automatu Mealy'ego z naszego przykładu. Na początku dla każdego stanu a_i wyznaczamy zbiór $U(a_i)$ wektorów wejściowych, które inicjują przejścia do stanu a_i (tabela 10.13).

TABELA 10.13. Zbiory $U(a_i)$ kombinacji wejściowych inicjujących przejścia do stanów automatu Mealy'ego

a_i	$U(a_i)$
s_0	$\{011, 111, ---, 0--, 1--\}$
s_1	$\{---\}$
s_2	$\{--0\}$
s_3	$\{-01\}$

Z tabeli 10.13 wynika, że przejścia do większości stanów automatu Mealy'ego są inicjowane dokładnie jedną kombinacją zmiennych wejściowych. Wyjątkiem jest stan s_0 , do którego przejścia są inicjowane za pomocą pięciu wektorów wejściowych. Zgodnie z rysunkiem 10.36 stan s_0 zostaje rozszczepiony na pięć stanów $s_{0_1}, \dots, s_{0_5}$. Tabela 10.14 zawiera tablicę przejść automatu Mealy'ego po rozszczepieniu stanu s_0 .

TABELA 10.14. Tablica przejść automatu Mealy'ego po rozszczepieniu stanu s_0

a_m	$X(a_m, a_s)$	$Y(a_m, a_s)$	a_s
s_{0_1}	---	011	s_1
s_{0_2}	---	011	s_1
s_{0_3}	---	011	s_1
s_{0_4}	---	011	s_1
s_{0_5}	---	011	s_1
s_1	--0	101	s_2
s_1	-01	001	s_3
s_1	011	100	s_{0_1}
s_1	111	010	s_{0_2}
s_2	0--	100	s_{0_4}
s_2	1--	010	s_{0_5}
s_3	---	100	s_{0_3}

W celu zakodowania stanów wewnętrznych automatu skończonego klasy E, stosując analogię do automatów klas C i D, tworzymy macierz **W**. Jej wiersze odpowiadają stanom wewnętrznym, a kolumny reprezentują elementy zbioru G (czyli zmienne wejściowe). W każdym wierszu macierzy **W** zapisywane są wartości zmiennych wejściowych, które inicjują przejścia do odpowiednich stanów. Utworzona w ten sposób macierz **W** dla naszego przykładu została zamieszczona w tabeli 10.15 (pierwsze trzy kolumny).

TABELA 10.15. Macierz **W** do kodowania stanów wewnętrznych automatu klasy E

a_i	$X(G)$			E	
	g_2	g_1	g_0	e_1	e_0
s_{0_1}	0	1	1	0	1
s_{0_2}	1	1	1	0	1
s_{0_3}	–	–	–	1	1
s_{0_4}	0	–	–	0	0
s_{0_5}	1	–	–	0	0
s_1	–	–	–	1	0
s_2	–	–	0	0	1
s_3	–	0	1	0	1

Macierz **W** kodowania stanów wewnętrznych automatów klas E i F różni się od analogicznej macierzy **W** dla automatów klas C i D w następujący sposób:

- kolumny macierzy **W** dla automatów klas E i F odpowiadają zmiennym wejściowym, a nie wyjściowym, jak to było w automatach klas C i D;
- macierz **W** dla automatów klas E i F jest trójwartościowa, a nie dwuwartościowa, ponieważ zmiennym wejściowym przypisane mogą być trzy różne wartości.

Możemy wybrać dowolny ze stanów $s_{0_1}, \dots, s_{0_5}$ jako stan początkowy. Zważywszy na to, że nasz automat będzie implementowany na FPGA, najlepiej wybrać stan s_{0_4} jako stan początkowy z kodem 0--, co pozwoli zwykłym wyzerowaniem rejestru pamięci przełączyć automat w stan początkowy.

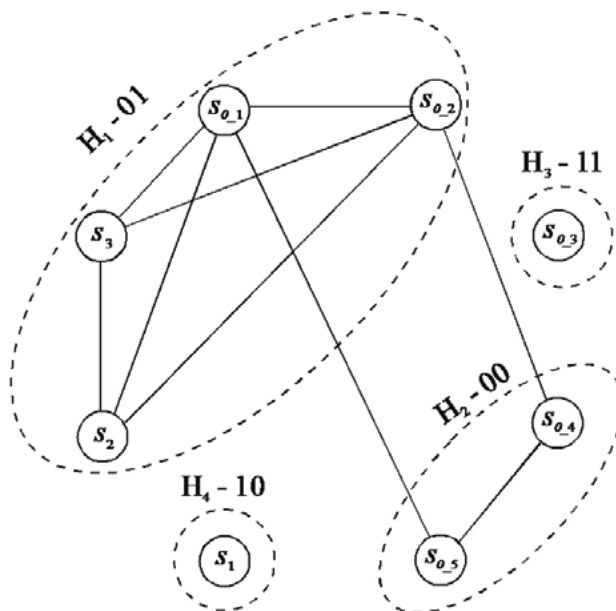
Następnie w celu kodowania stanów należy do macierzy **W** dodać minimalną liczbę kolumn, które będą odpowiadać dodatkowym bitom kodów stanów wewnętrznych, zapewniając wzajemną ortogonalność wierszy macierzy **W**. To zadanie można rozwiązać analogicznie do projektowania automatów klas C i D: tworzymy graf H ortogonalności wierszy macierzy **W**, dzielimy go na minimalną liczbę pełnych podgrafów i kodujemy poszczególne podgrafy kodem dwójkowym o minimalnym rozmiarze. Następnie wyznaczone kody podgrafów są umieszczane w dodatkowych bitach kodów stanów wewnętrznych. W naszych wcześniejszych przykładach nie było potrzeby tworzenia grafu H , ponieważ wszystkie wiersze macierzy **W** były wzajemnie ortogonalne.

Graf ortogonalności wierszy macierzy **W** dla syntezy automatu klasy E zilustrowany został na rysunku 10.38.

Wierzchołki grafu H odpowiadają wierszom macierzy **W**, czyli stanom wewnętrznym automatu skończonego. Dwa wierzchołki grafu H są łączone krawędzią, jeśli odpowiednie wiersze macierzy **W** są ortogonalne, czyli różnią się od siebie znaczącymi wartościami (0 lub 1) przynajmniej na jednym bicie.

W naszym przykładzie graf H jest podzielony na cztery podgrafy: $H_1 = \{s_{0_1}, s_{0_2}, s_2, s_3\}$; $H_2 = \{s_{0_4}, s_{0_5}\}$; $H_3 = \{s_{0_3}\}$; $H_4 = \{s_1\}$. Każdy z tych podgrafów jest kodowany dwubitowym kodem, gdzie kod zerowy jest przypisany do podgrafu H_2 , który zawiera stan

początkowy s_{0_4} . Kody podgrafów $H_1, \dots, H_4$ są zapisywane do macierzy $\mathbf{W}$ w kolumnach e_1 i e_0 , które odpowiadają zmiennym zbioru E . Kodowanie stanów wewnętrznych automatu jest determinowane przez zawartość wierszy macierzy $\mathbf{W}$.



RYS. 10.38. Graf ortogonalności wierszy macierzy $\mathbf{W}$ przy kodowaniu stanów automatu klasy E

Na podstawie tablicy przejść (tabela 10.14) oraz kodów stanów wewnętrznych (tabela 10.15) przedstawiamy poniżej opis automatu klasy E w języku Verilog:

```

module Mealy_E(
  input clk,reset,
  input [2:0] x,
  output reg [2:0] y);

  reg [1:0] next;
  reg [1:0] e;
  reg [2:0] g;

  localparam [4:0]
    s0_1= 5'b011_01,
    s0_2= 5'b111_01,
    s0_3= 5'bxxx_11,

```

```

s0_4= 5'b0xx_00,
s0_5= 5'b1xx_00,
s1=   5'bxxx_10,
s2=   5'bx0_01,
s3=   5'bx01_01;

```

```

always @(posedge clk, negedge reset)
  if (~reset)    begin g <= 3'b000;   e <= 2'b00; end
  else          begin g <= x;       e <= next; end

```

```

always @(*)
  casex ({g,e})
s0_1: begin y = 3'b011; next = 2'b10; end
s0_2: begin y = 3'b011; next = 2'b10; end
s0_3: begin y = 3'b011; next = 2'b10; end
s0_4: begin y = 3'b011; next = 2'b10; end
s0_5: begin y = 3'b011; next = 2'b10; end
s1:   if (~x[0])          begin y = 3'b101; next = 2'b01; end
      else if (~x[1] && x[0]) begin y = 3'b001; next = 2'b01; end
      else if (~x[2] && x[1] && x[0]) begin y = 3'b100; next = 2'b01; end
      else                begin y = 3'b010; next = 2'b01; end
s2:   if (~x[2])          begin y = 3'b100; next = 2'b00; end
      else                begin y = 3'b010; next = 2'b00; end
s3:   begin y = 3'b100; next = 2'b11; end
  default: begin y = 3'b000; next = 2'b00; end
  endcase

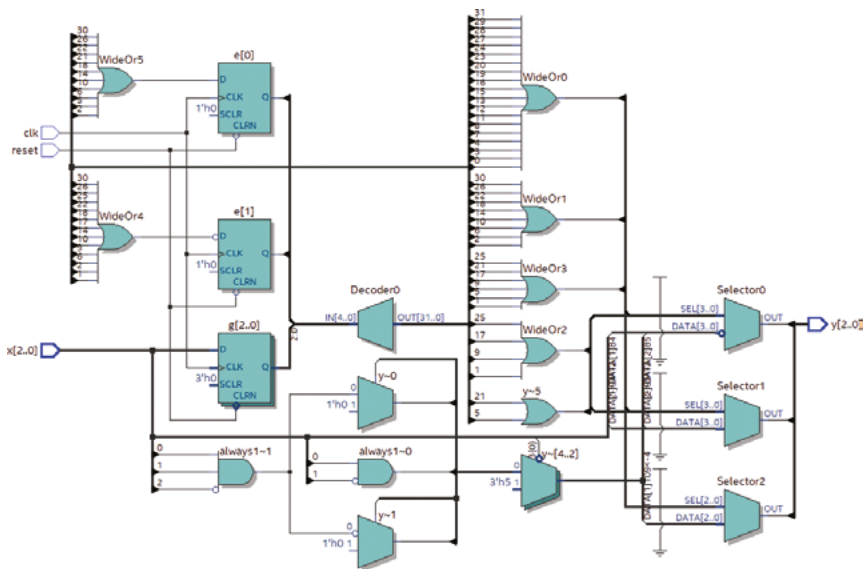
```

endmodule

W przedstawionym opisie zmienna *next* odpowiada zbiorowi *D* w strukturze na rysunku 10.37, zmienna *e* – zbiorowi *E*, a zmienna *g* – zbiorowi *G*. W celu zwiększenia czytelności w opisie kodów stanów wewnętrznych zastosowano znak podkreślenia, który oddziela część kodu określającą stanem wejść od dodatkowej części kodu tworzonej na podstawie zmiennych zbioru *E*.

Wynik syntezy projektu *Mealy_E* przedstawiony został na rysunku 10.39.

Na rysunku 10.39 możemy dostrzec, że wartości funkcji wyjściowych są przekazywane na wejścia rejestru pamięci automatu skończonego. Następnie sygnały z wyjść rejestru i zmienne wejściowe wykorzystywane są do formowania funkcji ze zbiorów *G* i *D*, co idealnie odzwierciedla strukturę automatu przedstawioną na rysunku 10.33.



RYS. 10.39. Wynik syntezy projektu *Mealy_E* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Mealy_E* całkowicie się zgadzają z analogicznymi wynikami projektu *Mealy_style_1* pokazanymi na rysunkach 10.9 i 10.10.

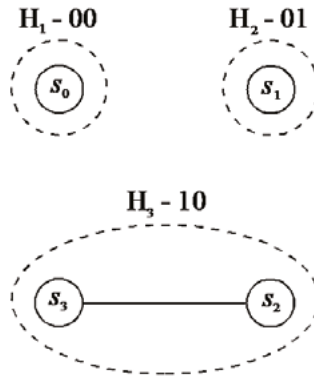
10.8.5. Synteza automatów klasy E₁ bez rozszczepienia stanów wewnętrznych

Podobnie jak w przypadku syntezy automatu klasy D₁, przystąpimy teraz do projektowania automatu klasy E₁ bez konieczności rozszczepiania stanów wewnętrznych. Przeanalizujemy tabelę 10.13, aby przypisać kod stanom wewnętrznym, do których prowadzi więcej niż jeden wektor wejściowy. Te stany otrzymają kod złożony wyłącznie z nieokreślonych wartości -...- ($x...x$). Macierz **W** dla naszego przykładu przyjmuje postać pokazaną w tabeli 10.16.

TABELA 10.16. Macierz **W** kodowania stanów wewnętrznych automatu klasy E₁ bez rozszczepienia stanów

a_i	$X(G)$			E	
	g_2	g_1	g_0	e_1	e_0
s_0	–	–	–	0	0
s_1	–	–	–	0	1
s_2	–	–	0	1	0
s_3	–	0	1	1	0

Graf H ortogonalności wierszy macierzy $\mathbf{W}$ oraz sposób jego podziału na podgrafy zilustrowany został na rysunku 10.40.



RYS. 10.40. Graf H przy syntezie automatu klasy E_1 bez rozszczepienia stanów wewnętrznych

Graf H został podzielony na trzy podgrafy zakodowane dwubitowym kodem dwójkowym w następujący sposób: $H_1 - 00$; $H_2 - 01$; $H_3 - 10$. Kody podgrafów wpisywane są do kolumn e_1 i e_0 macierzy $\mathbf{W}$, co tworzy dodatkową część kodu stanów wewnętrznych. Zawartość poszczególnych wierszy macierzy $\mathbf{W}$ z tabeli 10.16 jest wykorzystywana do kodowania stanów wewnętrznych automatu.

Poniżej zaprezentowano opis automatu klasy E_1 w języku Verilog:

```

module Mealy_E_1(
  input clk,reset,
  input [2:0] x,
  output reg [2:0] y);

  reg [1:0] next;
  reg [1:0] e;
  reg [2:0] g;

  localparam [4:0]
    s0=   5'bxxx_00,
    s1=   5'bxxx_01,
    s2=   5'bx0_10,
    s3=   5'bx01_10;

  always @(posedge clk, negedge reset)
    if (~reset) begin g <= 3'bxxx; e <= 2'b00; end
    else begin g <= x; e <= next; end

```

```

always @(*)
  casex ({g,e})
    s0:   begin y = 3'b011; next = 2'b01; end

    s1:   if (~x[0])          begin y = 3'b101; next = 2'b10; end
         else if (~x[1] && x[0]) begin y = 3'b001; next = 2'b10; end
         else if (~x[2] && x[1] && x[0]) begin y = 3'b100; next = 2'b00; end
         else                  begin y = 3'b010; next = 2'b00; end

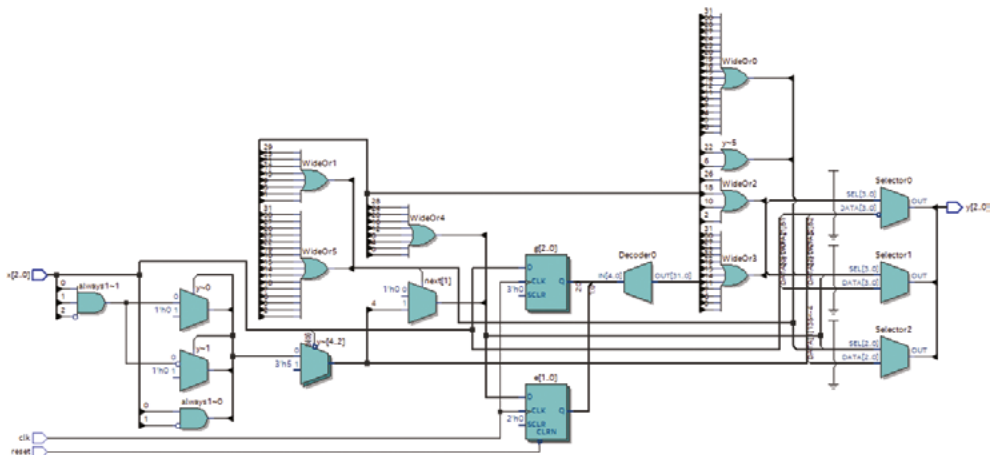
    s2:   if (~x[2])          begin y = 3'b100; next = 2'b00; end
         else                  begin y = 3'b010; next = 2'b00; end

    s3:   begin y = 3'b100; next = 2'b00; end
  default: begin y = 3'b000; next = 2'b00; end
  endcase

endmodule

```

Wynik syntezy projektu *Mealy_E_1* przedstawiony został na rysunku 10.41.



RYS. 10.41. Wynik syntezy projektu *Mealy_E_1* na poziomie przesłań międzyrejestrowych

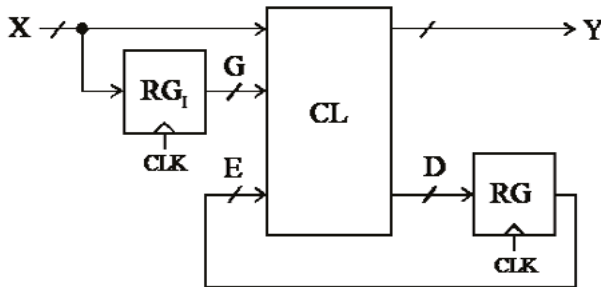
Wyniki funkcjonalnej i czasowej symulacji projektu *Mealy_E_1* dokładnie zgadzają się z odpowiednimi wynikami projektu *Mealy_style_1* zamieszczonymi na rysunkach 10.9 i 10.10.

10.8.6. Synteza automatów klasy F

Metoda syntezy automatu klasy F jest podobna do syntezy automatu klasy E z tą różnicą, że dotyczy automatu Moore'a. W automacie klasy F do kodowania stanów wykorzystywane są wektory wejściowe. Struktura idealnego automatu klasy F została pokazana na rysunku 10.1f, a jego funkcjonowanie opisywane jest równaniami (10.6).

W porównaniu z automatem klasy E struktura automatu klasy F różni się tym, że układ kombinacyjny CL_ψ nie jest bezpośrednio połączony z wejściami automatu, co oznacza, że funkcje wyjściowe automatu Moore'a nie zależą od zmiennych wejściowych.

Ponieważ w praktyce rzadko można spotkać idealny automat klasy F, konieczne jest wprowadzenie dodatkowych bitów kodów stanów wewnętrznych. W rezultacie automat realizowany jest w postaci struktury przedstawionej na rysunku 10.42.



RYŚ. 10.42. Struktura automatu klasy F

Zaprezentowana struktura automatu klasy F dokładnie odpowiada strukturze przedstawionej na rysunku 10.37. Chociaż w idealnym układzie automatu klasy F (rysunek 10.1f) sygnały wejściowe automatu nie są bezpośrednio połączone z wejściami układu kombinacyjnego CL_ψ , to w strukturze na rysunku 10.42 zauważamy, że zmienne wejściowe zbioru X są bezpośrednio podawane na wejścia układu kombinacyjnego CL . W tym przypadku zmienne wejściowe są wykorzystywane w układzie kombinacyjnym do generowania wartości funkcji przejść ze zbioru D , ponieważ zmienne zbiorów G i E nie mogą jednoznacznie określić przejścia do następnego stanu (one wskazują tylko aktualny stan, w którym znajduje się automat skończony).

Zastosujemy metodę syntezy automatu klasy F do automatu Moore'a z naszego przykładu. Zbiory $U(a_i)$ wektorów wejściowych inicjujących przejścia do każdego stanu a_i są ukazane w tabeli 10.17.

TABELA 10.17. Zbiory $U(a_i)$ wektorów wejściowych inicjujących przejścia do stanów automatu Moore'a

a_i	$U(a_i)$
s_0	{---}
s_1	{---}
s_2	{--0}
s_3	{-01}
s_4	{011,0--,---}
s_5	{111,1--}

Z tabeli 10.17 można zauważyć, że w celu syntezy automatu klasy F, zgodnie z rysunkiem 10.36, należy rozszcześcić stan s_4 na trzy stany: s_{4_1} , s_{4_2} i s_{4_3} , natomiast stan s_5 – na dwa stany: s_{5_1} i s_{5_2} . Tablica przejść automatu Moore’a po rozszczepleniu stanów s_4 i s_5 została przedstawiona w tabeli 10.18.

TABELA 10.18. Tablica przejść automatu Moore’a po rozszczepleniu stanów s_4 i s_5

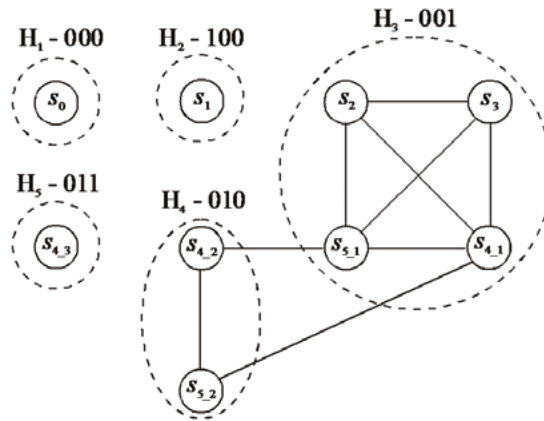
a_m	$X(a_m, a_s)$	$Y(a_m)$	a_s
s_0	---	000	s_1
s_1	--0	011	s_2
s_1	-01	011	s_3
s_1	011	011	s_{4_1}
s_1	111	011	s_{5_1}
s_2	0--	101	s_{4_2}
s_2	1--	101	s_{5_2}
s_3	---	001	s_{4_3}
s_{4_1}	---	100	s_0
s_{4_2}	---	100	s_0
s_{4_3}	---	100	s_0
s_{5_1}	---	010	s_0
s_{5_2}	---	010	s_0

Następnie tworzymy macierz **W** do kodowania stanów wewnętrznych automatu klasy F. W macierzy tej w poszczególnych wierszach zapisujemy odpowiednie zestawy zmiennych wejściowych (pierwsze trzy kolumny w tabeli 10.19).

TABELA 10.19. Macierz **W** kodowania stanów wewnętrznych automatu klasy F

a_i	$X(G)$			E		
	g_2	g_1	g_0	e_2	e_1	e_0
s_0	–	–	–	0	0	0
s_1	–	–	–	1	0	0
s_2	–	–	0	0	0	1
s_3	–	0	1	0	0	1
s_{4_1}	0	1	1	0	0	1
s_{4_2}	0	–	–	0	1	0
s_{4_3}	–	–	–	0	1	1
s_{5_1}	1	1	1	0	0	1
s_{5_2}	1	–	–	0	1	0

Graf H ortogonalności wierszy macierzy $\mathbf{W}$ został zaprezentowany na rysunku 10.43.



RYS. 10.43. Graf H ortogonalności wierszy macierzy $\mathbf{W}$ przy kodowaniu stanów wewnętrznych automatu klasy F

Graf H został podzielony na pięć podgrafów zakodowanych trzybitowymi kodami dwójkowymi: $H_1 - 000$; $H_2 - 100$; $H_3 - 001$; $H_4 - 010$; $H_5 - 011$. Kody podgrafów $H_1, \dots, H_5$ zostały wpisane do odpowiednich wierszy macierzy $\mathbf{W}$ (kolumny e_2, e_1 i e_0 w tabeli 10.19). Zawartość jej poszczególnych wierszy odpowiada kodom stanów wewnętrznych automatu skończonego. Na podstawie tablicy przejść automatu (tabela 10.18) oraz wyznaczonych kodów stanów (tabela 10.19) można opisać automat klasy F w języku Verilog:

```

module Moore_F(
  input clk,reset,
  input [2:0] x,
  output reg [2:0] y);

  reg [2:0] g, e, next;

  localparam [5:0]
    s0=   6'bxxx_000,
    s1=   6'bxxx_100,
    s2=   6'bx0_001,
    s3=   6'bx01_001,
    s4_1= 6'b011_001,
    s4_2= 6'b0xx_010,
    s4_3= 6'bxxx_011,
    s5_1= 6'b111_001,
    s5_2= 6'b1xx_010;

```



```

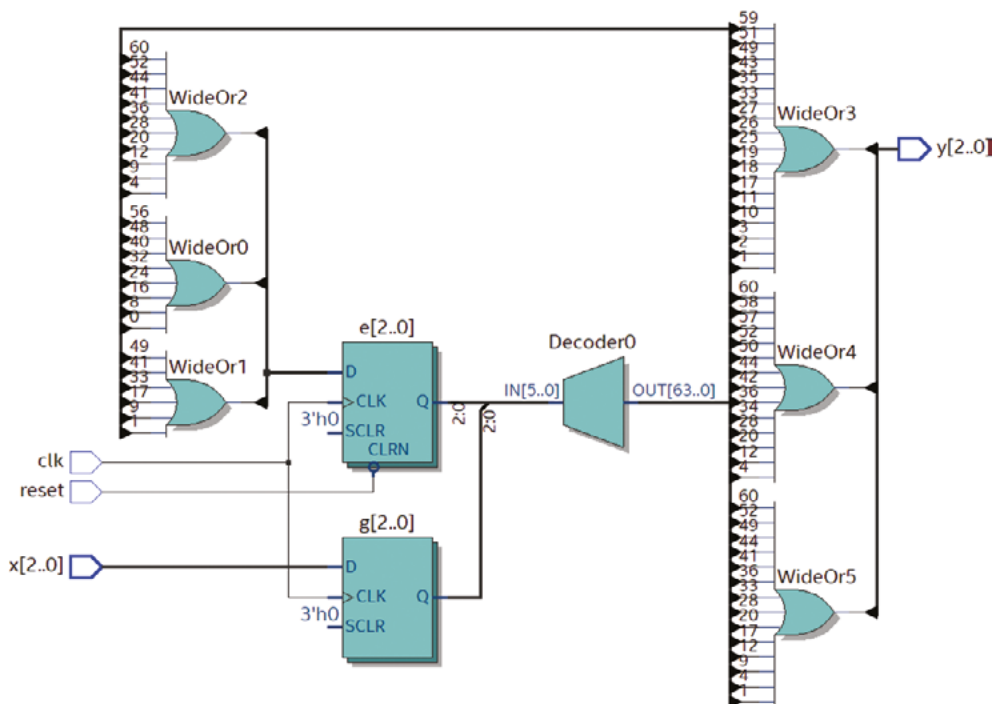
always @(posedge clk, negedge reset)
    if (~reset)    begin g <= 3'bxxx;   e <= 3'b000; end
    else          begin g <= x;       e <= next;   end

always @(*)
    case ({g,e})
        s0:    next = 3'b100;
        s1:    if (~x[0])                next = 3'b001;
                else if (~x[1] && x[0])    next = 3'b001;
                else if (~x[2] && x[1] && x[0]) next = 3'b001;
                else                    next = 3'b001;
        s2:    if (~x[2])    next = 3'b010;
                else        next = 3'b010;
        s3:    next = 3'b011;
        s4_1: next = 3'b000;
        s4_2: next = 3'b000;
        s4_3: next = 3'b000;
        s5_1: next = 3'b000;
        s5_2: next = 3'b000;
        default: next = 3'b000;
    endcase

always @(*)
    case ({g,e})
        s0:    y = 3'b000;
        s1:    y = 3'b011;
        s2:    y = 3'b101;
        s3:    y = 3'b001;
        s4_1: y = 3'b100;
        s4_2: y = 3'b100;
        s4_3: y = 3'b100;
        s5_1: y = 3'b010;
        s5_2: y = 3'b010;
        default: y = 3'b000;
    endcase
endmodule

```

Wynik syntezy projektu *Moore_F* został zamieszczony na rysunku 10.44.



RYS. 10.44. Wynik syntezy projektu *Moore_F* na poziomie przesłań międzyrejestrowych

Wyniki funkcjonalnej i czasowej symulacji projektu *Moore_F* są zgodne z odpowiednimi wynikami projektu *Moore_style_1* pokazanymi na rysunkach 10.15 i 10.16.

Automat klasy F można również zaprojektować bez rozszczepienia stanów wewnętrznych, analogicznie do wykonanej wcześniej syntezy automatów klas D_1 i E_1. Czytelnikowi proponujemy samodzielne przeprowadzenie syntezy automatu klasy F_1 bez rozszczepienia stanów wewnętrznych i porównanie uzyskanych wyników z odpowiednimi wynikami automatu klasy F z rozszczepieniem stanów.

10.8.7. Porównanie modeli automatów skończonych

Na początku przeprowadzimy analityczną ocenę złożoności realizacji różnych struktur automatów skończonych. W tym celu obliczymy minimalną liczbę elementów logicznych FPGA wymaganych do realizacji automatu.

Niech L będzie liczbą zmiennych wejściowych automatu skończonego, N – liczbą zmiennych wyjściowych, a M – liczbą stanów wewnętrznych automatu. Załóżmy, że każda z funkcji przejść lub wyjść automatu będzie realizowana na jednym elemencie logicznym FPGA. W rzeczywistości funkcje te mogą wymagać więcej elementów logicznych do ich realizacji. Dlatego w naszych obliczeniach otrzymamy minimalną wartość kosztów realizacji automatów.

Dla automatu klasy A (tradycyjnego automatu Mealy'ego) minimalny koszt realizacji jest określany wzorem:

$$L_{min} = N + \text{int} \log_2 M \quad (10.8)$$

W wyrażeniu (10.8) zakłada się, że każda funkcja wyjść realizowana jest na jednym elemencie logicznym FPGA, co oznacza, że wykorzystywane są N elementy logiczne dla funkcji wyjściowych. Dodatkowo przy kodowaniu stanów automatu za pomocą kodu dwójkowego minimalnego rozmiaru, zgodnie z równaniem (10.7), potrzebne są $\text{int} \log_2 M$ elementów logicznych do realizacji funkcji przejść. Dla naszego przykładu automatu Mealy'ego, gdzie $N = 3$ i $M = 4$, otrzymujemy $L_{minA} = 5$.

Dla automatu klasy B (tradycyjnego automatu Moore'a) minimalny koszt realizacji również jest obliczany na podstawie wyrażenia (10.8). Jednak liczba stanów wewnętrznych automatu Moore'a zwykle jest większa w porównaniu z automatem Mealy'ego, więc drugi składnik w wyrażeniu (10.8) często może być większy niż w przypadku automatu Mealy'ego. Dla automatu Moore'a z naszego przykładu, gdzie $N = 3$ i $M = 5$, otrzymujemy $L_{minB} = 6$.

Minimalny koszt realizacji automatu klasy C można opisać następującym wyrażeniem:

$$L_{min} = N + \text{int} \log_2 Q, \quad (10.9)$$

gdzie Q to minimalna liczba pełnych podgrafów $H_1, \dots, H_Q$, na które podzielony zostaje graf H ortogonalności wierszy macierzy W . W naszym przykładzie automatu klasy C mamy $N = 3$, $Q = 1$, wobec tego $L_{minC} = 3$.

Dla automatu klasy D minimalna złożoność realizacji także jest opisywana wyrażeniem (10.9). W automacie klasy D z naszego przykładu parametry przyjmują wartości $N = 3$, $Q = 1$, a zatem $L_{minD} = 3$. W przypadku syntezy automatu klasy D_1 bez rozszczepienia stanów wewnętrznych automat ma podobne wartości parametrów $N = 3$, $Q = 1$, stąd $L_{minD_1} = 3$.

Minimalny koszt realizacji automatu klasy E obliczany jest według następującego wzoru:

$$L_{min} = N + \text{int} \log_2 Q + L \quad (10.10)$$

W porównaniu z (10.9) wyrażenie (10.10) uwzględnia dodatkowy parametr L . Wynika to z obecności buforów sygnałów wejściowych w strukturze przedstawionej na rysunku 10.37, co wymaga użycia dodatkowych elementów logicznych FPGA do ich realizacji. W przypadku automatu klasy E z naszego przykładu, gdzie $N = 3$, $Q = 4$, $L = 3$, minimalna liczba elementów logicznych wynosi $L_{minE} = 8$. Jeśli zmienne wejściowe automatu są buforowane w zewnętrznym buforze, przyjmujemy $L = 0$, co skutkuje $L_{minE} = 5$. Natomiast dla przykładu automatu klasy E_1 bez rozszczepienia stanów, gdzie $N = 3$, $Q = 3$, $L = 3$, minimalna liczba elementów logicznych wynosi $L_{minE_1} = 8$.

Dla automatu klasy F minimalny koszt realizacji również jest obliczany według wzoru (10.10). W naszym przykładzie automatu klasy F mamy $N = 3$, $Q = 5$, $L = 3$, co oznacza, że $L_{minF} = 9$. Jeśli jednak zmienne wejściowe są buforowane w buforze zewnętrznym, to $L = 0$, a minimalny koszt realizacji wynosi $L_{minF} = 6$. W przypadku automatu klasy F_1 bez rozszczepienia stanów wewnętrznych parametry automatu przyjmują wartości $N = 3$, $Q = 5$, $L = 3$, wobec tego $L_{minF_1} = 9$.

W celu porównania modeli automatów skończonych przeprowadzono syntezę automatów klas C–F w systemie Quartus, a wyniki porównano z automatami klas A i B. Do opisu automatów klas A i B wybrano styl 1. Ponieważ metody syntezy automatów klas C–F są skoncentrowane na zmniejszeniu rozmiaru bitowego kodów stanów, do kodowania stanów wewnętrznych klas A i B zastosowano metodę *Minimal Bits*. Wyniki przeprowadzonych badań zebrane są w tabeli 10.20, gdzie: L – liczba elementów logicznych FPGA użytych do realizacji automatu skończonego; F – maksymalna częstotliwość działania automatu skończonego (w MHz); L_{min} – minimalna wymagana liczba elementów logicznych FPGA określana zgodnie z wyrażeniami (10.8), (10.9) i (10.10); D_1, E_1 i F_1 – automaty klas D, E i F projektowane bez rozszczepienia stanów wewnętrznych.

TABELA 10.20. Wyniki badań metod syntezy automatów klas A, B, C, D, E i F

Klasa automatu	L_{min}	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
		L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy															
A	5	7	633	7	383	8	1073	5	1422	7	1232	4	730	7	1221
D	3	7	466	7	258	8	502	4	1009	7	611	4	541	7	595
D_1	3	6	515	6	237	9	504	3	1037	8	595	4	546	8	635
E	8	9	570	9	301	12	766	5	1176	11	896	5	667	11	920
E_1	8	8	643	8	384	9	1069	5	1422	8	1248	4	730	8	1244
Best Mealy	3	6	643	6	384	8	1073	4	1422	7	1248	4	730	7	1244
Moore															
B	6	8	416	8	234	8	582	5	1171	7	737	4	685	7	725
C	3	6	484	6	252	7	640	3	1165	6	801	4	681	6	802
F	9	11	464	11	260	12	579	6	1106	11	715	5	645	11	716
F_1	9	11	399	11	216	12	579	8	854	11	721	6	466	11	672
Best Moore	3	6	484	6	260	7	640	3	1171	6	801	4	685	6	802

W tabeli 10.20 zauważalne jest, że dla rodzin Arria II i Cyclone V niektóre modele automatów otrzymują wartość L_{min} mniejszą od liczby zmiennych wejściowych L . Wynika to z faktu, że w raporcie kompilatora dla rodzin układów Arria II i Cyclone V podawana jest nie liczba elementów logicznych FPGA, lecz liczba użytych

konfigurowalnych modułów logicznych (*Adaptive Logic Modules – ALMs*), które mają 8 wejść i 4 wyjścia. Warto zauważyć, że element logiczny FPGA zwykle ma 4 wejścia lub 6 wejść oraz jedno wyjście.

Analiza automatów Mealy'ego pokazuje, że najlepszymi pod względem złożoności realizacji okazały się automaty klas D i D_1, zgodnie z przewidywaniami opartymi na szacunkach wartości L_{min} . Jednak zaskoczeniem było to, że najszybszym modelem okazał się automat klasy E_1 dla wszystkich rodzin układów, z wyjątkiem rodziny MAX 10.

W przypadku automatów Moore'a najlepsze wyniki pod względem złożoności realizacji oraz szybkości działania uzyskał automat klasy C (prawie dla wszystkich rodzin układów FPGA). Otrzymane wyniki są w pełni zgodne z oczekiwaniami, co potwierdza obliczona wartość L_{min} . Wyjątkiem są rodziny Arria II i Cyclone V, gdzie najlepszym pod względem szybkości okazał się automat klasy B, oraz rodzina MAX V, gdzie największą szybkość uzyskał automat klasy F.

Wnioski:

- aby zmniejszyć koszty realizacji automatu Mealy'ego, można zastosować metody syntezy automatów klasy D i D_1 i wybrać lepszy z otrzymanych wyników;
- aby zwiększyć szybkość działania automatu Mealy'ego, można zastosować metodę syntezy automatu klasy E_1, porównać wyniki z automatem klasy A i wybrać najlepszy automat;
- w celu zmniejszenia kosztów realizacji oraz zwiększenia szybkości działania automatu Moore'a należy wybrać metodę syntezy automatu klasy C.

Porównamy najlepsze wyniki syntezy automatów Mealy'ego (*Best Mealy*) oraz Moore'a (*Best Moore*). Okazało się, że automaty Mealy'ego klas A i E_1 są najszybsze dla wszystkich rodzin układów FPGA. Niemniej jednak, jeśli chodzi o złożoność realizacji, najlepsze wyniki dla rodzin układów MAX 10, Arria II, Cyclone IV i Cyclone 10 wykazał automat Moore'a klasy C.

Wniosek: podczas projektowania automatów skończonych o najwyższej szybkości rekomenduje się wybór modelu automatu Mealy'ego oraz zastosowanie metod syntezy automatów klas A i E_1. Natomiast przy dążeniu do minimalizacji kosztów realizacji automatów skończonych najlepszą opcją jest zastosowanie metody syntezy automatu Moore'a klasy C.

10.9. Ocena efektywności metod projektowania automatów skończonych

Istnieją trzy podstawowe kryteria optymalizacji automatów skończonych:

- koszt realizacji,
- szybkość działania,
- pobierana moc.

Podczas projektowania automatów skończonych na FPGA koszt realizacji jest wyrażany liczbą elementów logicznych użytych do implementacji automatu. Z kolei szybkość działania automatów jest określana przez maksymalną częstotliwość (F_{max}) funkcjonowania automatu skończonego, która wyznaczana jest częstotliwością sygnału synchronizacji. Zazwyczaj maksymalna częstotliwość F_{max} jest raportowana w kompilacyjnych raportach systemu Quartus.

Moc pobierana przez automaty skończone dzieli się na dwie składowe: statyczną i dynamiczną. Składowa statyczna mocy jest bezpośrednio proporcjonalna do liczby elementów logicznych FPGA użytych do implementacji automatu. Natomiast składowa dynamiczna zależy od liczby przerzutników, które są przełączane w jednym cyklu sygnału synchronizacji. Im mniejsza liczba przełączeń, tym mniejsza wartość składowej dynamicznej mocy.

Statyczna składowa mocy pobieranej stanowi około 95% całkowitej mocy zużywanej przez układ automatu skończonego, natomiast pozostałe 5% to dynamiczna składowa mocy rozpraszanej. Z tego powodu metody syntezy, które zmniejszają koszty realizacji automatów skończonych, jednocześnie przyczyniają się do obniżenia zużycia mocy przez te automaty.

W niniejszym rozdziale porównamy różne metody optymalizacji automatów skończonych, biorąc pod uwagę zarówno koszt ich realizacji, jak i szybkość działania. Efektywność strukturalnych modeli automatów została już oceniona w poprzednim podrozdziale.

10.9.1. Optymalizacja automatów skończonych za pomocą stylów opisu w języku Verilog

W rozdziale 10.5 przedstawiono różne style opisu automatów skończonych w języku Verilog i dokonano ich porównania. W trakcie analizy stosowano domyślny sposób kodowania stanów wewnętrznych (*Auto*), który w systemie Quartus jest reprezentowany przez metodę *One-Hot*. W późniejszych metodach syntezy najczęściej stosowano kodowanie stanów za pomocą kodu dwójkowego o minimalnej długości. W celu kompleksowego porównania różnych metod optymalizacji automatów skończonych przeprowadzono ponowne badania stylów opisu automatów w języku Verilog, tym razem wykorzystując metodę kodowania *Minimal Bits*. Wyniki tych badań zostały przedstawione w tabeli 10.21.

Uwaga. Style opisów 4 i 5 nie są rozpatrywane, ponieważ opisują one zupełnie inne automaty skończone (omówione w podrozdziale 10.5.3).

W tabeli 10.21 można zauważyć, że dla automatu Mealy'ego z naszego przykładu wyniki syntezy pod względem kosztów realizacji (L) oraz szybkości działania (F) są identyczne dla wszystkich stylów opisu, niezależnie od rodziny układów FPGA. Można to tłumaczyć tym, że automat Mealy'ego z naszego przykładu jest bardzo prosty, ma tylko cztery stany, dlatego różne style opisu nie mają wpływu na parametry optymalizacji.

TABELA 10.21. Wyniki badań stylów opisu automatów skończonych przy wykorzystaniu metody kodowania *Minimal Bits*

Styl opisu	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy_														
Style_1	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Style_2	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Style_3	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Style_6	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Style_7	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Style_8	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Best Mealy	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Moore_														
Style_1	8	409	8	219	8	582	5	1171	7	737	4	685	7	725
Style_2	9	397	9	236	10	574	6	1143	9	715	4	691	9	712
Style_3	9	397	9	236	10	574	6	1143	9	715	4	691	9	712
Style_6	8	401	8	212	9	599	5	1167	8	755	4	678	8	745
Style_7	9	397	9	236	10	574	6	1143	9	715	4	691	9	712
Style_8	8	401	8	212	9	599	5	1167	8	755	4	678	8	745
Moore_1_process	9	417	9	212	9	724	6	1114	9	686	4	706	9	717
Best Moore	8	417	8	236	8	724	5	1179	7	755	4	706	7	745

W tabeli 10.21 zauważamy, że dla automatu Moore’a z naszego przykładu możemy wyodrębnić dwie grupy stylów opisu, które charakteryzują się dokładnie tymi samymi wynikami: pierwsza grupa to style 2, 3 i 7, a druga to style 6 i 8. Jeśli chodzi o koszty realizacji dla automatu Moore’a, najlepszym stylem jest styl 1, natomiast pod względem szybkości działania wyróżniamy styl *Moore_1_process* (dla rodzin MAX II, MAX 10 i Cyclone V), style 6 i 8 (dla rodzin Cyclone IV i Cyclone 10), styl 1 (dla rodziny Arria II) oraz style 2, 3 i 7 (dla rodziny MAX V).

Porównanie najlepszych wyników dla automatu Mealy’ego (*Best Mealy*) i automatu Moore’a (*Best Moore*) wykazuje, że pod względem kosztów realizacji automat Mealy’ego prezentuje nieco lepsze wyniki dla rodzin MAX II i MAX V. Ponadto jest znacznie szybszy dla wszystkich rodzin układów.

Należy zauważyć, że zastosowanie kodu dwójkowego o minimalnej długości, w porównaniu z metodą *One-Hot* (patrz tabela 10.4), znacznie zmniejszyło koszty realizacji obu rodzajów automatów oraz zwiększyło szybkość działania automatu Mealy’ego. Niemniej jednak wydajność automatu Moore’a w tym przypadku nieco się zmniejszyła.

10.9.2. Optymalizacja automatów skończonych za pomocą metod oferowanych przez system Quartus

W systemie Quartus dostępne są wbudowane metody optymalizacji projektowanych układów obejmujące nie tylko automaty skończone, lecz także inne komponenty. Te metody są zgrupowane i umożliwiają optymalizację projektów pod względem trzech głównych parametrów. Aby wybrać określoną strategię optymalizacji, należy w systemie Quartus przejść do opcji **Assignments > Settings**. W otworzonym oknie zaproponowane zostaną następujące tryby optymalizacji:

- *Balanced (Normal flow)* – zbalansowany (normalny przebieg projektowania);
- *Performance (High effort – increases runtime)* – wydajność (duży wysiłek – zwiększa czas wykonania syntezy);
- *Performance (Aggressive – increases runtime and area)* – wydajność (agresywny – zwiększa czas syntezy oraz koszt realizacji);
- *Power (High effort – increases runtime)* – pobór mocy (duży wysiłek – zwiększa czas wykonania syntezy);
- *Power (Aggressive – increases runtime, reduces performance)* – pobór mocy (agresywny – zwiększa czas syntezy, obniża szybkość działania);
- *Area (Aggressive – reduces performance)* – koszt realizacji (agresywny – obniża szybkość działania).

Strategia *Balanced* jest ustawiana domyślnie i ma na celu zapewnienie ograniczeń czasowych, które są określone przez użytkownika.

Zrealizowano dwie strategie zwiększenia szybkości działania układu (wydajności): *Performance (High effort – increases runtime)* oraz *Performance (Aggressive – increases runtime and area)*. Pierwsza z nich zwiększa szybkość działania układu kosztem zwiększenia czasu wykonania syntezy, a druga dodatkowo zwiększa też koszt realizacji.

Istnieją również dwie strategie zmniejszenia poboru mocy: *Power (High effort – increases runtime)* oraz *Power (Aggressive – increases runtime, reduces performance)*. Pierwsza z nich zmniejsza pobór mocy układu kosztem wydłużenia czasu syntezy, natomiast druga może dodatkowo obniżyć szybkość działania projektowanego układu.

Strategia *Area* pozwala obniżyć złożoność realizacji kosztem zmniejszenia szybkości działania projektowanego układu.

Przeprowadzono badania strategii optymalizacji systemu Quartus zastosowanych do automatów Mealy’ego i Moore’a z naszego przykładu. Automaty opisane zostały z wykorzystaniem stylu 1 oraz sposobu kodowania stanów wewnętrznych *Minimal Bits*. Wyniki badań przedstawione zostały w tabeli 10.22. Nie uwzględniono tutaj strategii optymalizujących pobór mocy.

Analiza wyników w tabeli 10.22 wykazuje, że dla automatu Moore’a koszt realizacji jest identyczny dla wszystkich strategii optymalizacji. Wykorzystanie strategii *Performance 1* i *2* pozwoliło zwiększyć szybkość działania dla rodzin MAX II, MAX 10, Cyclone IV i Cyclone 10. W przeciwieństwie do oczekiwań strategia *Area* pokazała najwyższą wydajność dla rodzin układów MAX V i Cyclone V.

TABELA 10.22. Wyniki badań metod optymalizacji automatów systemu Quartus (styl opisu 1, kodowanie stanów wewnętrznych metodą *Minimal Bits*)

Strategia optymalizacji	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy														
Balanced	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Performance 1	7	646	7	382	8	1074	5	1422	7	1282	4	730	7	1284
Performance 2	7	646	7	382	8	1074	5	1422	7	1282	4	730	7	1284
Area	7	631	7	384	8	1073	5	1422	7	1232	4	731	7	1221
Best Mealy	7	646	7	384	8	1074	5	1422	7	1282	4	731	7	1284
Moore														
Balanced	8	409	8	219	8	582	5	1171	7	737	4	685	7	725
Performance 1	8	419	8	235	8	583	5	1167	7	737	4	684	7	737
Performance 2	8	419	8	235	8	583	5	1167	7	737	4	684	7	737
Area	8	409	8	219	8	582	5	1171	7	737	4	685	7	725
Best Moore	8	419	8	235	8	583	5	1171	7	737	4	685	7	737

Podobne wyniki uzyskano również w przypadku automatu Moore'a, gdzie, wybór strategii także nie miał wpływu na koszt realizacji dla wszystkich rodzin układów FPGA. Strategie *Performance 1* i *2* przyczyniły się do zwiększenia szybkości działania dla rodzin MAX II, MAX V, MAX 10 i Cyclone 10, podczas gdy strategie *Balanced* i *Area* wykazały wysoką wydajność dla rodzin układów Arria II i Cyclone V.

Wniosek. Strategie optymalizacji systemu Quartus nie wpływają istotnie na koszt realizacji automatów skończonych z naszego przykładu i jedynie nieznacznie zmieniają ich szybkość działania. Ponadto zastosowanie strategii *Performance* nie zawsze prowadzi do zwiększenia wydajności.

10.9.3. Wybór najbardziej efektywnego podejścia do optymalizacji automatów skończonych

Wszystkie omówione wcześniej style opisu automatów skończonych (podrozdział 10.5), metody kodowania stanów wewnętrznych (podrozdział 10.6), modele strukturalne automatów skończonych (podrozdział 10.8) oraz strategie optymalizacji systemu Quartus mogą być wykorzystane w celu zmniejszenia kosztów realizacji i zwiększenia szybkości działania automatów skończonych. Postaramy się odpowiedzieć na pytanie, które z tych podejść jest najbardziej efektywne pod względem podanych kryteriów optymalizacji.

W tym celu dokonamy porównania najlepszych wyników dla automatów Mealy’ego i Moore’a z naszego przykładu, uzyskanych w wyniku badań efektywności różnych stylów opisu (*Description*), które przedstawiono w tabeli 10.21, metod kodowania (*Encoding*), jak pokazano w tabeli 10.6, strukturalnych modeli (*Models*) opisanych w tabeli 10.20 oraz strategii optymalizacji w systemie Quartus (*Quartus*), które zostały zaprezentowane w tabeli 10.22. Wszystkie te wyniki zestawiono w tabeli 10.23.

TABELA 10.23. Porównanie różnych sposobów optymalizacji automatów skończonych

Sposób optymalizacji	MAX II		MAX V		MAX 10		Arria II		Cyclone IV E		Cyclone V		Cyclone 10	
	L	F	L	F	L	F	L	F	L	F	L	F	L	F
Mealy														
Description	7	631	7	384	8	1073	5	1422	7	1232	4	730	7	1221
Encoding	7	646	7	384	7	1073	5	1422	7	1232	4	730	7	1221
Models	6	643	6	384	8	1073	3	1422	7	1248	4	730	7	1244
Quartus	7	646	7	384	8	1074	5	1422	7	1282	4	731	7	1284
Best Mealy	6	646	6	384	7	1074	3	1422	7	1282	4	731	7	1284
Moore														
Description	8	417	8	236	8	724	5	1171	7	755	4	706	7	745
Encoding	7	512	7	300	7	748	4	1175	7	837	4	686	7	902
Models	3	484	6	260	7	640	3	1171	6	801	4	685	6	802
Quartus	8	419	8	235	8	583	5	1171	7	737	4	685	7	737
Best Moore	3	512	6	300	7	748	3	1175	6	837	4	706	6	902

Analiza tabeli 10.23 pokazuje, że w przypadku automatu Mealy’ego koszt realizacji różnych metod optymalizacji jest zbliżony. Wykorzystanie strukturalne modeli automatów (klas D i D_1) pozwala na obniżenie złożoności realizacji dla rodzin MAX II, MAX V i Arria II. Ponadto zauważamy, że dla rodziny MAX 10 najniższy koszt realizacji osiągnięto przy użyciu metody kodowania *User-Encoded* opisanej w podrozdziale 10.6.3.

Wnioski. Niewielką różnicę w kosztach realizacji przy stosowaniu różnych metod optymalizacji można wyjaśnić faktem, że nasz automat Mealy’ego jest stosunkowo prosty. Niemniej jednak udało się obniżyć koszty realizacji poprzez zastosowanie modeli automatów D i D_1 oraz metod kodowania, które zmniejszają złożoność realizacji automatów skończonych.

Różne podejścia do minimalizacji mają również niewielki wpływ na szybkość działania automatu Mealy’ego z naszego przykładu. Możemy zauważyć, że strategie optymalizacji w systemie Quartus (*Performance 1* i *2*) przyczyniły się do zwiększenia szybkości działania automatu dla rodzin MAX II, MAX 10, Cyclone IV i Cyclone 10.

Wniosek. Niewielkie zwiększenie szybkości działania automatu skończonego Mealy'ego może być osiągnięte przy wykorzystaniu metod optymalizacji *Performance* systemu Quartus.

Dla automatu Moore'a złożoność realizacji znacznie różni się w zależności od zastosowanej metody optymalizacji. Na przykład dla rodziny MAX II koszt realizacji L osiągnięty przy podejściach *Description* i *Quartus* wynosi 8, natomiast przy podejściu *Models* $L = 3$, co oznacza, że koszt realizacji jest mniejszy o $8/3 = 2,67$ razy. Dla wszystkich rodzin układów FPGA najlepszym podejściem do optymalizacji automatu pod względem kosztów realizacji jest wykorzystanie strukturalne modeli (automatu klasy C).

Najlepszym podejściem do osiągnięcia maksymalnej szybkości działania automatu Moore'a jest wybór sposobu kodowania stanów wewnętrznych. Jednakże dla rodziny Cyclone V największą szybkość uzyskuje się metodą opisu Moore_1_process. Trudno jednoznacznie wskazać najlepszy sposób kodowania. Rozwiązania trzeba szukać między metodami *User-Encoded*, *Johnson*, *One-Hot* oraz *Gray*. Niespodzianką jest to, że strategie *Performance* systemu Quartus nie zwiększają szybkości działania automatu Moore'a z naszego przykładu.

Wnioski. Zastosowanie modelu strukturalnego automatu klasy C podczas syntezy automatów skończonych Moore'a znacząco obniża koszty realizacji, co w niektórych przypadkach przekłada się na oszczędność rzędu 2,67 razy. Zwiększenie szybkości działania automatu Moore'a z naszego przykładu można osiągnąć przez dobór odpowiedniego sposobu kodowania stanów wewnętrznych.

Analiza najlepszych rozwiązań dla automatów Mealy'ego (*Best Mealy*) i Moore'a (*Best Moore*) wykazuje, że pod względem złożoności realizacji automat Moore'a uzyskuje lepsze wyniki dla rodzin MAX II, Cyclone IV i Cyclone 10. Niemniej jednak pod względem szybkości działania automat Moore'a ustępuje automatu Mealy'ego dla wszystkich rodzin układów FPGA.

Wnioski. Najbardziej efektywnym podejściem do obniżenia kosztów realizacji jest wykorzystanie strukturalne modeli automatów klas D i D_1 dla automatów Mealy'ego oraz modelu automatu klasy C dla automatów Moore'a.

Najbardziej efektywnym sposobem zwiększenia szybkości działania automatu Mealy'ego jest wykorzystanie strategii optymalizacji *Performance*, natomiast dla automatu Moore'a będzie to dobór odpowiedniego sposobu kodowania stanów wewnętrznych.

Przeprowadzone badania nie pozwalają jednak na dogłębną analizę każdej z metod optymalizacji automatów skończonych. Dzięki otrzymanym wynikom można jedynie wstępnie ocenić te metody. Aby dokładniej zbadać efektywność poszczególnych metod optymalizacji automatów skończonych, konieczne byłoby przeprowadzenie rozszerzonych badań na znacznie większej liczbie przykładów.

10.10. Podsumowanie rozdziału

Modelem matematycznym układów sekwencyjnych jest *automat skończony* (*finite state machine* – *FSM*). Ogólnie każdy system cyfrowy można potraktować jako automat skończony, który ma bardzo dużą liczbę stanów wewnętrznych. W systemach wbudowanych automaty skończone często pełnią funkcję układów sterowania kontrolujących system, podsystem, blok lub schemat. Przy projektowaniu systemów wbudowanych standardowe bloki funkcjonalne często się powtarzają, podczas gdy układy sterowania są unikatowe i muszą być projektowane od nowa za każdym razem. Z tego powodu projektowanie automatów skończonych zawsze wymaga dużo czasu.

Podczas implementacji automatów skończonych na FPGA wykorzystuje się sześć strukturalnych modeli: automatów klas A, B, C, D, E i F. Automaty klas A i B reprezentują tradycyjne modele automatów Mealy’ego i Moore’a odpowiednio. Automat klasy C jest typu Moore’a, gdzie wyjścia pokrywają się z kodami stanów wewnętrznych automatu. Analogicznie automat klasy D jest automatem typu Mealy’ego, gdzie każdy zestaw sygnałów wyjściowych pokrywa się z kodem następnego stanu. Natomiast automaty klas E i F są automatami Mealy’ego i Moore’a odpowiednio, gdzie każdy zestaw sygnałów wejściowych odpowiada kodowi następnego stanu automatu skończonego.

W strukturach automatów klas C i D nie ma układu kombinacyjnego, który realizuje funkcje wyjściowe, podczas gdy w automatach klas E i F brak jest układu kombinacyjnego generującego funkcje przejść. To prowadzi do zmniejszenia kosztów realizacji oraz zwiększenia szybkości działania automatów skończonych.

Wyjścia automatu Mealy’ego są asynchroniczne względem wejść, co oznacza, że w automacie Mealy’ego przebywającym w tym samym stanie zmiana sygnałów wejściowych będzie powodować zmianę sygnałów na jego wyjściach.

Automaty skończone zazwyczaj są opisywane za pomocą *grafu automatu* lub *tablicy przejść*. W języku Verilog nie ma dedykowanych konstrukcji do opisu automatów skończonych. Są one automatycznie rozpoznawane w kodzie projektu przez narzędzia syntezy systemu Quartus.

Do niezbędnych sygnałów sterowania automatem skończonym należą sygnał synchronizacji sterujący przełączeniem pamięci automatu, a także sygnał zerowania, który resetuje automat do stanu początkowego. Funkcje przejść i wyjść automatu mogą być opisywane w różny sposób. Stany wewnętrzne automatu oraz ich kody są definiowane w języku Verilog za pomocą parametrów lokalnych. Stan początkowy automatu skończonego powinien być umieszczony jako pierwszy na liście parametrów lokalnych. Przejścia automatu skończonego są zazwyczaj opisywane przy użyciu instrukcji **case**, a sprawdzanie warunków przejść często odbywa się za pomocą instrukcji **if**.

W systemie Quartus dostępne jest narzędzie **State Machine Viewer**, które umożliwia wizualizację grafu automatu, tabeli przejść oraz kodów stanów wewnętrznych zaprojektowanego automatu.

Należy dokładnie dobierać typ automatu skończonego (Mealy’ego lub Moore’a) podczas realizacji algorytmu sterowania, ponieważ sposób działania tych automatów może znacznie się od siebie różnić.

W celu sprawdzenia poprawności działania automatu skończonego zawsze zaleca się przeprowadzenie symulacji czasowej, która umożliwia dokładniejszą ocenę funkcjonowania zaprojektowanego układu. Symulacja funkcjonalna służy jedynie do sprawdzenia logiki działania projektu, bez uwzględnienia opóźnień czasowych.

Istnieją trzy podstawowe style opisu automatów w języku Verilog: z wykorzystaniem trzech procesów, z dwoma procesami oraz z jednym procesem. Warto zaznaczyć, że ten ostatni styl może być stosowany jedynie do opisu automatów Moore'a.

W stylu opisu automatu skończonego z trzema procesami (styl 1) każdy obszar funkcjonalny – czyli pamięć automatu, funkcje przejść oraz funkcje wyjść – jest opisywany oddzielnie za pomocą osobnych bloków **always**. Natomiast w przypadku stylu opisu z dwoma procesami opis pamięci automatu jest łączony z opisem funkcji przejść lub funkcji wyjść. Istnieje także możliwość opisu zarówno funkcji przejść, jak i wyjść w jednym procesie, a pamięci automatu w drugim. W przypadku jednego procesu możliwe jest opisanie jedynie automatu Moore'a, gdzie wewnątrz tego procesu zdefiniowane są funkcje przejść i pamięć automatu, a funkcje wyjść są opisane oddzielnie z użyciem instrukcji ciągłego przypisania **assign**. Jednakże automat Mealy'ego nie może być kodowany w ten sposób, ponieważ doprowadziłoby to do umieszczenia rejestrów na wyjściach automatu.

W celu umieszczenia rejestrów na wyjściach automatu skończonego wystarczy w trzecim procesie stylu 1, który opisuje funkcje wyjść, zamiast **always @(*)** użyć **always @(posedge clk)**.

Aby zapewnić niezawodność automatu skończonego, należy zagwarantować jego powrót do stanu początkowego w przypadku, gdy kod aktualnego stanu wewnętrznego nie jest zgodny z żadnym zdefiniowanym w automacie kodem. Można to osiągnąć poprzez użycie konstrukcji **default** w instrukcji **case**.

Wśród omówionych ośmiu stylów opisu automatów skończonych style 1, 2, 3, 6, 7 i 8 pozwalają projektować automaty Mealy'ego o bardzo podobnych parametrach, jeśli chodzi o koszt realizacji oraz szybkość działania. Wybór konkretnego stylu opisu zależy od preferencji użytkownika. Z punktu widzenia niezawodności funkcjonowania automatu skończonego rekomendowany jest wybór stylu 8. Style 4 i 5 pozwalają budować automaty skończone z rejestrami na wyjściach, o podobnych wartościach parametrów, jednak wolniejszych w porównaniu ze stylami 1, 2, 3, 6, 7 i 8. Najlepsze do opisu automatu Moore'a są style 6 i 8. Do praktycznego zastosowania rekomendowany jest styl 8, ponieważ pozwala projektować bardziej niezawodne automaty skończone.

Aby po włączeniu zasilania automat rozpoczynał swoją pracę od stanu początkowego, należy zakodować ten stan za pomocą wartości zerowej kodu.

Ogólnie przy realizacji automatu skończonego na FPGA stan początkowy nie musi koniecznie mieć kodu zerowego. Jednak w takim przypadku w projekcie musi zostać zrealizowany układ, który po włączeniu zasilania przestawi automat w stan początkowy. To z kolei wymaga użycia dodatkowych zasobów logicznych FPGA i może prowadzić do obniżenia szybkości działania automatu.

W systemie Quartus istnieją różne sposoby kodowania stanów wewnętrznych automatów skończonych: *Auto* – domyślny sposób kodowania ustalony w systemie dla określonego typu układów programowalnych; *One-Hot* – kodowanie unarne; *Gray* – kodowanie w kodzie Graya; *Johnson* – kodowanie w kodzie Johnsona; *Minimal Bits* – kodowanie kodem dwójkowym o minimalnym rozmiarze; *Sequential* – sekwencyjne kodowanie dwójkowe; *User-Encoded* – kodowanie definiowane przez użytkownika.

Dla układów FPGA metoda kodowania *Auto* odpowiada kodowaniu unarnemu *One-Hot*. Kodowanie unarne zazwyczaj jest wykorzystywane do zwiększenia szybkości działania automatów skończonych. Jednak implementacja metody *One-Hot* w systemie Quartus jest specyficzna. Ponieważ stan początkowy automatu powinien mieć kod zerowy, więc we wszystkich kodach zostaje zanegowany jeden bit, ten, w którym kod stanu początkowego zawiera jedynkę. Metody kodowania *Gray* i *Johnson* stosowane są w celu zmniejszenia poboru mocy. Przy czym realizacja kodu Johnsona wymaga mniej zasobów logicznych w porównaniu z kodem Graya. Kodowanie *Minimal Bits* umożliwia obniżenie kosztów realizacji automatów skończonych. W metodzie *Sequential* stany automatu są kodowane kolejnymi liczbami dwójkowymi. Przy wyborze sposobu kodowania *User-Encoded* stanom automatu przypisywane są kody określone użytkownikiem w opisie projektu.

Metoda kodowania *One-Hot* nie zawsze zapewnia najszybsze działanie automatów skończonych.

Najlepszym podejściem do projektowania szybkich automatów skończonych jest wykorzystanie modelu automatu Mealy’ego i przetestowanie różnych metod kodowania stanów wewnętrznych. Następnie można wybrać tę realizację automatu, która zapewnia maksymalną szybkość działania.

Nie istnieje jedna uniwersalna metoda kodowania stanów wewnętrznych automatów skończonych, która zapewniłaby najlepsze wyniki pod względem zarówno złożoności, jak i szybkości działania dla wszystkich rodzin układów FPGA.

Tryb syntezy niezawodnych automatów skończonych w systemie Quartus wprowadza do projektu dodatkową logikę, która wykrywa *niedozwolone stany* (*illegal states*) automatu skończonego i zapewnia bezwarunkowe przejście w stan początkowy z każdego takiego stanu.

Przy projektowaniu niezawodnych automatów skończonych należy zamiast opcji **Safe State Machines** stosować style 7 lub 8 opisu automatów skończonych w języku Verilog.

W praktyce podczas projektowania automatów klas C–F często wprowadza się dodatkowe bity kodów stanów. W przypadku syntezy automatów klas C–F celem jest minimalizacja liczby tych dodatkowych bitów kodów stanów wewnętrznych. Pomocne w rozwiązywaniu tego zadania może się okazać rozszczepienie niektórych stanów.

Aby zmniejszyć koszty realizacji automatu Mealy'ego, można skorzystać z metod syntezy automatów klasy D oraz D_1, a następnie wybrać najlepszy wynik spośród otrzymanych. Natomiast w celu zwiększenia szybkości działania automatu Mealy'ego warto rozważyć zastosowanie metody syntezy automatu klasy E_1, porównać uzyskane wyniki z modelem klasy A i wybrać optymalny automat.

Aby zmniejszyć koszty realizacji i ewentualnie zwiększyć szybkość działania automatu Moore'a, zaleca się wykorzystanie metody syntezy automatu klasy C.

Podczas projektowania automatów skończonych o najwyższej szybkości działania zaleca się wybór modelu automatu Mealy'ego i zastosowanie metod syntezy automatów klas A lub E_1. Natomiast w przypadku automatów skończonych o najmniejszych kosztach realizacji najlepiej nadaje się metoda syntezy automatu Moore'a klasy C.

Istnieją trzy podstawowe kryteria optymalizacji automatów skończonych: koszt realizacji, szybkość działania oraz pobierana moc.

Podczas projektowania automatów skończonych na FPGA koszt realizacji jest wyrażany liczbą elementów logicznych wykorzystanych do implementacji automatu. Natomiast szybkość działania automatu określana jest maksymalną częstotliwością (F_{max}) funkcjonowania, wyznaczaną poprzez częstotliwość sygnału synchronizacji. Pobór mocy automatu składa się z dwóch składowych: statycznej i dynamicznej. Składowa statyczna jest bezpośrednio proporcjonalna do liczby elementów logicznych FPGA wykorzystanych do implementacji automatu. Natomiast składowa dynamiczna zależy od liczby przerzutników przełączanych w jednym takcie sygnału synchronizacji.

W systemie Quartus zaimplementowane są wbudowane metody optymalizacji projektowanych układów, które są zgrupowane i umożliwiają optymalizację projektów względem trzech wymienionych parametrów.

Najbardziej efektywnym podejściem do obniżenia kosztów realizacji jest wykorzystanie strukturalne modeli automatów klas D i D_1 dla automatów Mealy'ego oraz modelu automatu klasy C dla automatów Moore'a.

Najbardziej efektywnym sposobem zwiększenia szybkości działania automatów Mealy'ego jest wykorzystanie strategii optymalizacji *Performance*, natomiast dla automatów Moore'a będzie to wybór odpowiedniego sposobu kodowania stanów wewnętrznych.

Bibliografia

1. Brayton R.K., Hachtel G., McMullen C., Sangiovanni-Vincentelli A., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston 1984.
2. *Cyclone Device Handbook*, vol. 1: *Device Interface and Integration*, Altera, San Jose 2018.
3. *3IEEE Standard Verilog Hardware Description Language*. IEEE Computer Society, The Institute of Electrical and Electronics Engineers Inc., New York 2001.
4. Kania D., *Układy logiki programowalnej. Podstawy syntezy i sposoby odwzorowania technologicznego*, Wydawnictwo Naukowe PWN, Warszawa 2021.
5. Klimowicz A., Salauyou V., *State Merging and Splitting Strategies for Finite State Machines Implemented in FPGA*, „Applied Sciences-Basel” 2022, vol. 12, no. 16, s. 1–19.
6. Klimowicz A., Salauyou V., *Structural models of finite-state machines for their implementation on programmable logic devices and systems on chip*, „Journal of Computer and Systems Sciences International” 2015, vol. 54, no. 2, s. 230–242.
7. LaMeres Brock J., *Introduction to Logic Circuit and Logic Design with Verilog*, Springer Int. Publishing A, Cham 2017.
8. Majewski J., Zbysiński P., *Układy FPGA w przykładach*, Wydawnictwo BTC, Legionowo 2007.
9. Palnitkar S., *Verilog HDL: A guide to digital design and synthesis*, Mountain View, Prentice Hall PTR 2003.
10. Readler B., *Verilog by Example. A concise Introduction for FPGA Design*, Full ARC Press, 2011.
11. Rzeszut P., *Systemy embedded w FPGA*, Wydawnictwo BTC, Legionowo 2019.
12. Salauyou V., *Implementation of finite state machines based on programmable logic ICs with the help of the merged model of Moore and Mealy machines*, „Journal of Communications Technology and Electronics” 2013, vol. 58, no. 2, s. 172–177.
13. Salauyou V., Klimowicz A., *Projektowanie systemów wbudowanych w układach FPGA*, Oficyna Wydawnicza Politechniki Białostockiej, Białystok 2022.
14. Salauyou V., Klimowicz A., Grześ T., Bułatowa I., *Język Verilog w projektowaniu systemów wbudowanych na układach FPGA*, Oficyna Wydawnicza Politechniki Białostockiej, Białystok 2022.
15. Salauyou V., Zabrocki Ł., *Coding Techniques in Verilog for Finite State Machine Designs in FPGA*, w: *Computer Information Systems and Industrial Management. 18th International Conference, CISIM 2019, Belgrade, Serbia, September 19–21, 2019, Proceedings*, red. K. Saeed, R. Chaki, V. Janev, Springer, Cham 2019, s. 493–505.
16. Unsalan C., Tar B., *Digital System Design with FPGA: Implementation Using Verilog and VHDL*, McGraw-Hill Education, New York 2017.
17. White E., *Making Embedded Systems*, Oreilly Media, 2011.

18. Wilmshurst T., *Designing embedded systems with PIC microcontrollers: principles and applications*, Elsevier, Oxford 2006.
19. Zwoliński M., *Projektowanie układów cyfrowych z wykorzystaniem języka VHDL*, WKŁ, Warszawa 2016.

Spis tabel

Tabela 2.1.	Tablica prawdy sumatora jednobitowego.....	41
Tabela 2.2.	Przykłady reprezentacji liczb całkowitych.....	56
Tabela 2.3.	Bramkowe prymitywy języka Verilog	58
Tabela 2.4.	Przełącznikowe prymitywy języka Verilog.....	58
Tabela 2.5.	Wartość sygnału w różnych węzłach dla dwóch źródeł sygnałów	60
Tabela 2.6.	Operatory bitowe.....	69
Tabela 2.7.	Wyniki operacji bitowych	70
Tabela 2.8.	Operatory redukcji.....	71
Tabela 2.9.	Przykłady wykorzystania operatorów redukcji.....	71
Tabela 2.10.	Operatory logiczne.....	73
Tabela 2.11.	Operatory relacji.....	74
Tabela 2.12.	Operatory arytmetyczne	75
Tabela 2.13.	Operatory różne	76
Tabela 2.14.	Priorytety operatorów.....	78
Tabela 2.15.	Rozmiary wyrażeń bitowych	78
Tabela 3.1.	Przyporządkowanie wyprowadzeń na płycie DE1-SoC	149
Tabela 3.2.	Nazwy wyprowadzeń FPGA połączonych z przełącznikami SW0-SW9 na płycie DE1-SoC.....	154
Tabela 3.3.	Nazwy wyprowadzeń FPGA połączonych diodami LED LEDR0-LEDR9 na płycie DE1-SoC	155
Tabela 3.4.	Nazwy wyprowadzeń FPGA połączonych z przyciskami KEY0-KEY3 na płycie DE1-SoC	155
Tabela 4.1.	Tablica prawdy funkcji f	177
Tabela 5.1.	ablica prawdy kodera 8:3.....	205
Tabela 5.2.	Tablica prawdy kodera priorytetowego 8:3.....	212
Tabela 5.3.	Tablica prawdy pełnego binarnego dekodera 3:8	217
Tabela 5.4.	Tablica prawdy dekodera 3:8 z wejściem zezwalającym	223
Tabela 5.5.	Tablica prawdy niepełnego dekodera 3:5.....	224
Tabela 5.6.	Tablica prawdy dekodera cyfr dziesiętnych do sterowania wyświetlaczem siedmiosegmentowym.....	228
Tabela 5.7.	Tablica prawdy dekodera do wyświetlania słowa „HELLO” na wyświetlaczu siedmiosegmentowym	230
Tabela 6.1.	Tablica prawdy zatrzasku typu SR.....	237

Tabela 6.2.	Funkcje uniwersalnego rejestru przesuwneego	268
Tabela 7.1.	Wyniki realizacji liczników na układach FPGA różnych rodzin.....	296
Tabela 8.1.	Tablica prawdy jednobitowego sumatora.....	315
Tabela 8.2.	Wyniki badań projektów 16-bitowych sumatorów	340
Tabela 8.3.	Tablica prawdy pełnego subtraktora.....	342
Tabela 9.1.	Grupowanie iloczynów częściowych 4-bitowego układu mnożącego według wag.....	372
Tabela 9.2.	Oznaczenia iloczynów częściowych zerowego poziomu.....	374
Tabela 9.3.	Rozmieszczenie ścieżek w grupach zerowego poziomu	375
Tabela 9.4.	Rozmieszczenie ścieżek w grupach pierwszego poziomu.....	376
Tabela 9.5.	Rozmieszczenie ścieżek w grupach drugiego poziomu hierarchii.....	376
Tabela 9.6.	Podział ścieżek zerowego poziomu na grupy przy syntezie układu mnożącego Dadda 4×4 bity	381
Tabela 9.7.	Rozmieszczenie ścieżek pierwszego poziomu w grupach przy syntezie układu mnożącego Dadda 4×4 bity	382
Tabela 9.8.	Rozmieszczenie ścieżek ostatniego poziomu w grupach przy syntezie układu mnożącego Dadda 4×4 bity	382
Tabela 9.9.	Realizacja algorytmu Bootha przy obliczeniu iloczynu $4_{10} \times 3_{10} = 12_{10}$ lub $0100_2 \times 0011_2 = 00001100_2$	392
Tabela 9.10.	Wykonanie algorytmu Bootha przy obliczeniu $4_{10} \times (-3_{10}) = -12_{10}$ lub $0100_2 \times 1101_2 = 11110100_2$	392
Tabela 9.11.	Instrukcje wykonywane w zmodyfikowanym algorytmie Bootha.....	395
Tabela 9.12.	Działanie zmodyfikowanego algorytmu Bootha przy obliczeniu $4_{10} \times 6_{10} = 24_{10}$ lub $0100_2 \times 0110_2 = 11000_2$	396
Tabela 9.13.	Wyniki badań projektów układów mnożących 4×4 bity.....	399
Tabela 10.1.	Tablica przejść automatu Mealy'ego.....	407
Tabela 10.2.	Tablica przejść automatu Moore'a.....	408
Tabela 10.3.	Porównanie wyników symulacji funkcjonalnej automatów Mealy'ego i Moore'a	420
Tabela 10.4.	Wyniki badań stylów opisu automatów skończonych	436
Tabela 10.5.	Wartości $C(a_i)$ i $K(a_i)$ przy kodowaniu stanów wewnętrznych automatu Moore'a.....	443
Tabela 10.6.	Wyniki badań sposobów kodowania stanów wewnętrznych automatów skończonych	445
Tabela 10.7.	Wyniki syntezy projektów <i>Mealy_style_1</i> i <i>Moore_style_1</i> bez opcji oraz z opcją Safe State Machines	447
Tabela 10.8.	Uogólniona macierz W do kodowania stanów wewnętrznych automatu klasy C.....	449
Tabela 10.9.	Macierz W dla przykładu kodowania stanów wewnętrznych automatu klasy C.....	450
Tabela 10.10.	Zbiory $Y(a_i)$ wektorów wyjściowych generowanych na przejściach do stanów automatu Mealy'ego z przykładu.....	454

Tabela 10.11. Tablica przejść automatu po rozszczepieniu stanu s_0 przy syntezie automatu klasy D.....	455
Tabela 10.12. Macierz W do kodowania stanów wewnętrznych automatu klasy D	455
Tabela 10.13. Zbiory $U(a_i)$ kombinacji wejściowych inicjujących przejścia do stanów automatu Mealy'ego	461
Tabela 10.14. Tablica przejść automatu Mealy'ego po rozszczepieniu stanu s_0	461
Tabela 10.15. Macierz W do kodowania stanów wewnętrznych automatu klasy E.....	462
Tabela 10.16. Macierz W kodowania stanów wewnętrznych automatu klasy E_1 bez rozszczepienia stanów.....	465
Tabela 10.17. Zbiory $U(a_i)$ wektorów wejściowych inicjujących przejścia do stanów automatu Moore'a.....	468
Tabela 10.18. Tablica przejść automatu Moore'a po rozszczepieniu stanów s_4 i s_5	469
Tabela 10.19. Macierz W kodowania stanów wewnętrznych automatu klasy F.....	469
Tabela 10.20. Wyniki badań metod syntezy automatów klas A, B, C, D, E i F	474
Tabela 10.21. Wyniki badań stylów opisu automatów skończonych przy wykorzystaniu metody kodowania <i>Minimal Bits</i>	477
Tabela 10.22. Wyniki badań metod optymalizacji automatów systemu Quartus (styl opisu 1, kodowanie stanów wewnętrznych metodą <i>Minimal Bits</i>).....	479
Tabela 10.23. Porównanie różnych sposobów optymalizacji automatów skończonych.....	480

Spis rysunków

Rys. 1.1. Tranzystory	18
Rys. 1.2. Układy o małym i średnim stopniu scalenia	18
Rys. 1.3. Mikroprocesory	19
Rys. 1.4. Komputer jednopłytkowy.....	19
Rys. 1.5. Mikrokontrolery	19
Rys. 1.6. Układ FPGA	20
Rys. 1.7. Płyta główna komputera osobistego	20
Rys. 1.8. Gotowa płyta do systemów wbudowanych opartych na FPGA/SoC.....	22
Rys. 1.9. Proces projektowania systemów wbudowanych	25
Rys. 1.10. Struktura PLA	28
Rys. 1.11. Struktura układu PROM	29
Rys. 1.12. Struktura PAL.....	29
Rys. 1.13. Struktura uniwersalnego PAL.....	30
Rys. 1.14. Struktura CPLD	31
Rys. 1.15. Uogólniona struktura FPGA.....	32
Rys. 1.16. Uogólniona struktura elementu logicznego FPGA	33
Rys. 1.17. Uogólniona struktura SoC	34
Rys. 2.1. Schemat jednobitowego sumatora na poziomie bramek logicznych	42
Rys. 2.2. Realizacja modułu <i>add_1_2</i> : przykład opisu sumatora jednobitowego za pomocą równań logicznych	44
Rys. 2.3. Realizacja modułu <i>add_1_3</i> : przykład opisu jednobitowego sumatora za pomocą instrukcji proceduralnych	44
Rys. 2.4. Realizacja modułu <i>add_2</i> dwubitowego sumatora stworzonego z dwóch sumatorów jednobitowych	47
Rys. 2.5. Realizacja modułów <i>reg_4_1</i> , <i>reg_4_2</i> oraz <i>reg_4_3</i>	49
Rys. 2.6. Realizacja modułu <i>add_N</i>	50
Rys. 2.7. Realizacja modułu <i>buf_4_1</i>	53
Rys. 2.8. Realizacja przykładu połączenia wyjść bramek za pomocą węzłów różnego typu: <i>y1</i> – wire ; <i>y2</i> – wor ; <i>y3</i> – wand ; <i>y4</i> – tri0 ; <i>y5</i> – tri1 ; <i>y6</i> – triereg ; <i>y7</i> – supply0 ; <i>y8</i> – supply1	62
Rys. 2.9. Realizacja modułu <i>mult_on_10</i>	70
Rys. 2.10. Realizacja modułu <i>inputs_test</i>	72
Rys. 2.11. Realizacja modułu <i>comp_eq</i>	72

Rys. 2.12. Realizacja modułu <i>if_subtraction</i>	74
Rys. 2.13. Realizacja modułu <i>comparator</i>	75
Rys. 2.14. Realizacja modułu <i>mux_2_1</i>	77
Rys. 2.15. Realizacja modułu <i>simple_ALU</i>	82
Rys. 2.16. Realizacja modułu <i>maj_3</i>	86
Rys. 2.17. Realizacja modułu <i>circ_latch</i>	88
Rys. 2.18. Realizacja modułu <i>circ_no_latch</i>	89
Rys. 2.19. Realizacja modułu <i>my_DFF</i>	89
Rys. 2.20. Wyniki syntezy modułu <i>blocking_AND</i>	90
Rys. 2.21. Wyniki syntezy modułu <i>non_blocking_AND</i>	91
Rys. 2.22. Realizacja modułu <i>blocking_assignments</i>	94
Rys. 2.23. Realizacja modułu <i>non_blocking_assignments</i>	95
Rys. 2.24. Wyniki syntezy modułu <i>nb_9_10</i>	96
Rys. 2.25. Realizacja modułu dwuwejściowego multipleksa.....	98
Rys. 2.26. Realizacja modułu <i>latch_1</i> zatrasku za pomocą instrukcji if.....	99
Rys. 2.27. Realizacja modułu <i>latch_2</i> : wynik błędnego opisu układu kombinacyjnego – na wyjściu schematu zostaje ustawiony zatrask	99
Rys. 2.28. Realizacja modułów <i>comb_circuit_1-3</i>	100
Rys. 2.29. Realizacja modułu <i>sm_IF</i>	102
Rys. 2.30. Realizacja modułów <i>sm_Case1-3</i>	103
Rys. 2.31. Realizacja modułu <i>sm_Case4</i> : przykład wykorzystania nieokreśloności do nadania wartości wyjścia w operatorze case	104
Rys. 2.32. Realizacja modułu <i>decoder</i> : przykład wykorzystania nieokreślonej wartości wejść w operatorze casex	105
Rys. 2.33. Realizacja modułu <i>sm_Case5</i> : przykład wykorzystania znaku ? przy opisie wartości wejść w operatorze casex	106
Rys. 2.34. Realizacja modułu <i>use_full_case</i>	107
Rys. 2.35. Realizacja przykładu bez atrybutu <i>full_case</i>	107
Rys. 2.36. Realizacja modułu <i>one_hot_assign1</i>	109
Rys. 2.37. Realizacja przykładu <i>one_hot_assign1</i> bez atrybutu <i>parallel_case</i>	109
Rys. 2.38. Realizacja modułu <i>one_hot_assign2</i>	110
Rys. 2.39. Realizacja przykładu z pominięciem atrybutów <i>full_case</i> i <i>parallel_case</i>	110
Rys. 2.40. Realizacja modułu <i>count_zero1</i>	112
Rys. 2.41. Realizacja modułu <i>gray_bin</i>	118
Rys. 2.42. Realizacja modułu <i>ram_model</i>	124
Rys. 3.1. Proces projektowania w systemie Quartus	130
Rys. 3.2. Strona startowa systemu Quartus.....	131
Rys. 3.3. Tworzenie nowego projektu za pomocą kreatora New Project Wizard.....	132
Rys. 3.4. Zapytanie o nazwę katalogu, projektu oraz modułu najwyższego poziomu.....	132
Rys. 3.5. Podanie nazwy katalogu, projektu oraz modułu najwyższego poziomu dla projektu <i>sum1</i>	133

Rys. 3.6. Zgoda na utworzenie katalogu projektu.....	133
Rys. 3.7. Wybór typu projektu.....	133
Rys. 3.8. Dodawanie plików projektu.....	134
Rys. 3.9. Wybór rodziny FPGA oraz konkretnego układu.....	134
Rys. 3.10. Określenie narzędzi projektowania innych producentów	135
Rys. 3.11. Podsumowanie utworzonego projektu	135
Rys. 3.12. Widok okna systemu Quartus z utworzonym projektem <i>sum1</i>	136
Rys. 3.13. Tworzenie nowego pliku z kodem projektu w języku Verilog	136
Rys. 3.14. Okno edytora tekstu.....	137
Rys. 3.15. Zapis pliku do katalogu projektu.....	137
Rys. 3.16. Szablony języka Verilog	138
Rys. 3.17. Szablony gotowych projektów w języku Verilog	139
Rys. 3.18. Szablony konstrukcji języka Verilog	139
Rys. 3.19. Szablony elementów logicznych	140
Rys. 3.20. Szablony atrybutów syntezy	141
Rys. 3.21. Szablony prymitywów.....	142
Rys. 3.22. Okno dodawania plików do projektu	143
Rys. 3.23. Okno wyboru pliku	143
Rys. 3.24. Zgoda na zapis pliku przed kompilacją	144
Rys. 3.25. Okno systemu Quartus po zakończeniu kompilacji projektu.....	145
Rys. 3.26. Schemat projektu <i>sum1</i> na poziomie przesłań międzyrejestrowych.....	145
Rys. 3.27. Mapa technologiczna projektu <i>sum1</i> po odwzorowaniu na strukturę docelowego układu FPGA	146
Rys. 3.28. Mapa technologiczna projektu <i>sum1</i> po upakowaniu w docelowym układzie FPGA.....	147
Rys. 3.29. Komunikaty o błędach kompilacji	148
Rys. 3.30. Okno komunikatów o błędach kompilacji.....	148
Rys. 3.31. Okno edytora przypisań	150
Rys. 3.32. Okno narzędzia wyszukiwania węzłów Node Finder	150
Rys. 3.33. Zwiększona liczba parametrów wyszukiwania.....	150
Rys. 3.34. Lista zewnętrznych sygnałów projektu w module Node Finder.....	151
Rys. 3.35. Lista zewnętrznych sygnałów projektu w edytorze przypisań.....	151
Rys. 3.36. Rozwijane menu Assignment Name.....	151
Rys. 3.37. Wynik przyporządkowania wyprowadzeń.....	152
Rys. 3.38. Przypomnienie o zapisie przyporządkowania wyprowadzeń do pliku.....	152
Rys. 3.39. Sprawdzanie przyporządkowania wyprowadzeń układu FPGA.....	152
Rys. 3.40. Fragment pliku ustawień <i>sum1.qsf</i> określający przyporządkowanie wyprowadzeń.....	153
Rys. 3.41. Eksport pliku ustawień	153
Rys. 3.42. Import pliku ustawień.....	154
Rys. 3.43. Okno edytora przebiegów czasowych.....	158

Rys. 3.44. Określenie czasu trwania symulacji.....	158
Rys. 3.45. Okno wyboru węzłów lub szyn	159
Rys. 3.46. Okno narzędzia Node Finder wyszukiwania węzłów.....	159
Rys. 3.47. Lista obserwowanych węzłów w edytorze przebiegów czasowych.....	159
Rys. 3.48. Przebiegi czasowe sygnałów wejściowych.....	161
Rys. 3.49. Grupowanie sygnałów	161
Rys. 3.50. Wynik grupowania sygnałów.....	162
Rys. 3.51. Grupowanie szyny z pojedynczym sygnałem.....	162
Rys. 3.52. Zmiana porządku bitów w zgrupowanym hierarchicznym sygnale <i>level2</i>	163
Rys. 3.53. Wynik symulacji funkcjonalnej.....	163
Rys. 3.54. Komunikat systemu operacyjnego o braku sterownika	164
Rys. 3.55. Okno programatora.....	165
Rys. 3.56. Ustawienie rodzaju płytki prototypowej	165
Rys. 3.57. Ustawienie modułu SOCVHPS do programowania układu FPGA rodziny Cyclone V	166
Rys. 3.58. Ustawienie modułu do programowania układu 5CSEMA5F31	166
Rys. 4.1. Niepoprawna realizacja funkcji <i>f</i> z umieszczeniem zatrzasku na wyjściu.....	173
Rys. 4.2. Realizacja funkcji <i>f</i> z wykorzystaniem konstrukcji else w operatorze if	173
Rys. 4.3. Niepoprawna realizacja funkcji <i>f</i> za pomocą instrukcji case	175
Rys. 4.4. Realizacja funkcji <i>f</i> z wykorzystaniem konstrukcji default w operatorze case	175
Rys. 4.5. Tablica Karnaugh dla funkcji <i>f</i>	178
Rys. 4.6. Binarny diagram decyzyjny dla funkcji <i>f</i>	178
Rys. 4.7. Wynik syntezy sumatora BCD.....	181
Rys. 4.8. Wynik symulacji funkcjonalnej sumatora cyfr BCD.....	181
Rys. 5.1. Wynik syntezy bufora trójstanowego.....	185
Rys. 5.2. Multiplexer 2-kanalowy: a) schemat na poziomie bramek; b) symbol graficzny	186
Rys. 5.3. Schemat multiplexera <i>MUX_2_1_struct</i>	187
Rys. 5.4. Wynik syntezy multiplexera <i>MUX_2_1_assign</i> przy opisie za pomocą wyrażenia.....	187
Rys. 5.5. Wynik syntezy multiplexera <i>MUX_2_1_assign_2</i> przy opisie za pomocą operacji warunkowej	188
Rys. 5.6. Wynik syntezy multiplexera <i>MUX_2_1</i> przy opisie za pomocą instrukcji case	188
Rys. 5.7. Wynik syntezy multiplexera <i>MUX_2_1</i> przy opisie za pomocą instrukcji if	189
Rys. 5.8. Wynik symulacji funkcjonalnej multiplexera <i>MUX_2_1</i>	189
Rys. 5.9. Wynik syntezy modułu <i>MUX_2_1_N_bits</i>	190
Rys. 5.10. Wyniki symulacji modułu <i>MUX_2_1_N_bits</i>	190
Rys. 5.11. Multiplexer 4-kanalowy: a) struktura; b) symbol graficzny	191
Rys. 5.12. Wynik syntezy modułu <i>MUX_4_1_struct</i>	191

Rys. 5.13. Wynik syntezy modułu <i>MUX_4_1_case</i>	192
Rys. 5.14. Wyniki symulacji multipleksera <i>MUX_4_1</i>	192
Rys. 5.15. Grupowy multipleksler <i>MUX_4_1_2_bits</i> o 2-bitowej szerokości danych: a) struktura; b) symbol graficzny	193
Rys. 5.16. Wynik syntezy modułu <i>MUX_4_1_2_bits</i>	194
Rys. 5.17. Wyniki symulacji modułu <i>MUX_4_1_2_bits</i>	194
Rys. 5.18. Struktura multipleksera <i>MUX_8_1</i>	194
Rys. 5.19. Wynik syntezy modułu <i>MUX_8_1_struct</i>	195
Rys. 5.20. Wyniki symulacji modułu <i>MUX_8_1_struct</i>	196
Rys. 5.21. Demultipleksler 2-kanałowy: a) schemat na poziomie bramek; b) symbol graficzny	196
Rys. 5.22. Wynik syntezy demultipleksera <i>DMUX_1_2</i>	197
Rys. 5.23. Wyniki symulacji demultipleksera <i>DMUX_1_2</i>	197
Rys. 5.24. Demultipleksler 4-kanałowy zbudowany z trzech demultipleksów 2-kanałowych: a) schemat; b) symbol graficzny	198
Rys. 5.25. Schemat modułu <i>DMUX_1_4_struct</i>	198
Rys. 5.26. Schemat demultipleksera <i>DMUX_1_4</i> przy wykorzystaniu instrukcji assign	200
Rys. 5.27. Schemat demultipleksera <i>DMUX_1_4</i> przy wykorzystaniu instrukcji case	200
Rys. 5.28. Schemat demultipleksera <i>DMUX_1_4</i> przy wykorzystaniu instrukcji if	201
Rys. 5.29. Wyniki symulacji demultipleksera <i>DMUX_1_4</i>	201
Rys. 5.30. 4-kanałowy demultipleksler grupowy z 2-bitową szerokością danych zbudowany z dwóch demultipleksów jednobitowych.....	202
Rys. 5.31. Wynik syntezy modułu <i>DMUX_1_4_N_bits</i>	202
Rys. 5.32. Wyniki symulacji modułu <i>DMUX_1_4_N_bits</i>	203
Rys. 5.33. Wynik syntezy modułu <i>DMUX_1_8_case</i>	204
Rys. 5.34. Wyniki symulacji modułu <i>DMUX_1_8_case</i>	204
Rys. 5.35. Wynik syntezy modułu <i>encoder_8_3_case</i>	206
Rys. 5.36. Wyniki symulacji modułu <i>encoder_8_3_case</i>	206
Rys. 5.37. Wynik syntezy modułu <i>encoder_8_3_case_v2</i>	208
Rys. 5.38. Wyniki symulacji modułu <i>encoder_8_3_case_v2</i>	208
Rys. 5.39. Wynik syntezy modułu <i>encoder_8_3_case_valid</i>	209
Rys. 5.40. Wyniki symulacji modułu <i>encoder_8_3_case_valid</i>	209
Rys. 5.41. Wynik syntezy modułu <i>encoder_8_3_if</i>	210
Rys. 5.42. Wyniki symulacji modułu <i>pr_encoder_8_3_case</i>	213
Rys. 5.43. Wyniki symulacji modułu <i>pr_encoder_8_3_case_v2</i>	214
Rys. 5.44. Wyniki symulacji modułu <i>pr_encoder_8_3_case_valid</i>	215
Rys. 5.45. Wynik syntezy modułu <i>pr_encoder_8_3_if</i>	215
Rys. 5.46. Wyniki symulacji modułu <i>pr_encoder_8_3_if</i>	216
Rys. 5.47. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji case	218
Rys. 5.48. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji case	218
Rys. 5.49. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji if	219

Rys. 5.50. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji if	219
Rys. 5.51. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji assign	220
Rys. 5.52. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji assign	221
Rys. 5.53. Wynik syntezy dekodera 3:8 z wykorzystaniem instrukcji for	222
Rys. 5.54. Wyniki symulacji dekodera 3:8 z wykorzystaniem instrukcji for	222
Rys. 5.55. Wynik syntezy dekodera 3:8 z wejściem zezwolenia	224
Rys. 5.56. Wyniki symulacji dekodera 3:8 z wejściem zezwolenia	224
Rys. 5.57. Wynik syntezy niepełnego dekodera 3:5 z zerowymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych	225
Rys. 5.58. Wyniki symulacji niepełnego dekodera 3:5 z zerowymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych	226
Rys. 5.59. Wynik syntezy niepełnego dekodera 3:5 z nieokreślonymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych	226
Rys. 5.60. Wyniki symulacji niepełnego dekodera 3:5 z nieokreślonymi wartościami na wyjściach dla niewykorzystywanych kombinacji wejściowych	227
Rys. 5.61. Wyświetlacz siedmiosegmentowy	227
Rys. 5.62. Wynik syntezy dekodera cyfr dziesiętnych	229
Rys. 5.63. Wyniki symulacji dekodera cyfr dziesiętnych	229
Rys. 5.64. Wynik syntezy dekodera do wypisywania słowa „HELLO” na wyświetlaczach siedmiosegmentowych	231
Rys. 5.65. Wyniki symulacji dekodera do wypisywania słowa „HELLO” na wyświetlaczach siedmiosegmentowych	231
Rys. 5.66. Dekoder adresów w systemie mikroprocesorowym	232
Rys. 6.1. Schemat zatrasku typu SR: a) na elementach NAND; b) na elementach NOR ...	236
Rys. 6.2. Wynik syntezy zatrasku SR realizowanego za pomocą modułów <i>latch_SR_NAND</i> i <i>latch_SR_assign_NAND</i>	238
Rys. 6.3. Wyniki symulacji zatrasku SR realizowanego za pomocą modułów <i>latch_SR_NAND</i> i <i>latch_SR_assign_NAND</i>	238
Rys. 6.4. Wynik syntezy zatrasku SR realizowanego na podstawie modułów <i>latch_SR_NOR</i> i <i>latch_SR_assign_NOR</i>	239
Rys. 6.5. Wyniki symulacji zatrasku SR realizowanego na podstawie modułów <i>latch_SR_NOR</i> i <i>latch_SR_assign_NOR</i>	239
Rys. 6.6. Wynik syntezy zatrasku SR realizowanego na podstawie modułów <i>latch_SR_if</i> i <i>latch_SR_assign</i>	240
Rys. 6.7. Wyniki symulacji zatrasku SR realizowanego na podstawie modułów <i>latch_SR_if</i> i <i>latch_SR_assign</i>	240
Rys. 6.8. Wynik syntezy zatrasku SR na podstawie modułu <i>latch_SR_if_v2</i>	241
Rys. 6.9. Schemat zatrasku typu D	242
Rys. 6.10. Wynik syntezy modułów <i>latch_if</i> i <i>latch_assign</i>	243
Rys. 6.11. Wyniki symulacji modułów <i>latch_if</i> i <i>latch_assign</i>	243
Rys. 6.12. Wynik syntezy modułów <i>latch_if_reset</i> i <i>latch_assign_reset</i>	244
Rys. 6.13. Wyniki symulacji modułów <i>latch_if_reset</i> i <i>latch_assign_reset</i>	245
Rys. 6.14. Wynik syntezy modułów <i>not_latch_always</i> i <i>not_latch_assign</i>	246

Rys. 6.15. Wyniki symulacji modułów <i>not_latch_always</i> i <i>not_latch_assign</i>	246
Rys. 6.16. Struktura zatrzasku typu <i>master-slave</i>	247
Rys. 6.17. Wynik syntezy zatrzasku <i>master-slave</i>	248
Rys. 6.18. Wyniki symulacji zatrzasku <i>master-slave</i>	248
Rys. 6.19. Umieszczenie zatrzasków na wyjściach układu kombinacyjnego w wyniku błędnie sporządzonego kodu projektu	249
Rys. 6.20. Schemat projektu <i>my_DFF</i> na poziomie przesłań międzyrejestrowych	251
Rys. 6.21. Schemat projektu <i>my_DFF_with_control_signals</i> na poziomie RTL	251
Rys. 6.22. Schemat projektu <i>my_DFF_with_sync_control_signals</i> na poziomie RTL	253
Rys. 6.23. Schemat projektu <i>my_DFF_with_nQ</i> na poziomie RTL	254
Rys. 6.24. Schemat projektu <i>my_DFF_with_ce</i> na poziomie RTL	254
Rys. 6.25. Schemat projektu <i>my_SRE</i> na poziomie przesłań międzyrejestrowych	256
Rys. 6.26. Schemat projektu <i>my_JKE</i> na poziomie przesłań międzyrejestrowych	257
Rys. 6.27. Schemat projektu <i>my_TE</i> na poziomie RTL	257
Rys. 6.28. Schemat projektu <i>dff_power_up_1</i> na poziomie RTL	258
Rys. 6.29. Mapa technologiczna projektu <i>dff_power_up_1</i>	258
Rys. 6.30. Schemat projektu <i>my_register</i> na poziomie przesłań międzyrejestrowych	260
Rys. 6.31. Schemat projektu <i>my_master_slave_register</i> na poziomie RTL	261
Rys. 6.32. Schemat projektu <i>my_register_with_areset</i> na poziomie RTL	261
Rys. 6.33. Schemat projektu <i>my_register_with_set_and_reset</i> na poziomie RTL	262
Rys. 6.34. Schemat projektu <i>slow_impulses</i> na poziomie RTL	264
Rys. 6.35. Wyniki symulacji projektu <i>slow_impulses</i>	264
Rys. 6.36. Mapa technologiczna projektu <i>slow_impulses</i>	264
Rys. 6.37. Schemat 4-bitowego rejestru przesuwającego wykonującego przesunięcie w prawo	265
Rys. 6.38. Schemat projektu <i>right_shift_register</i> na poziomie RTL	266
Rys. 6.39. Wyniki symulacji projektu <i>right_shift_register</i>	266
Rys. 6.40. Schemat projektu <i>left_shift_register</i> na poziomie RTL	267
Rys. 6.41. Wyniki symulacji projektu <i>left_shift_register</i>	267
Rys. 6.42. Schemat projektu <i>right_shift_register_load</i> na poziomie RTL	268
Rys. 6.43. Wyniki symulacji projektu <i>right_shift_register_load</i>	268
Rys. 6.44. Schemat projektu <i>universal_shift_register</i> na poziomie RTL	269
Rys. 6.45. Wyniki symulacji projektu <i>universal_shift_register</i>	270
Rys. 6.46. Symbol graficzny cyklicznego rejestru przesuwającego	270
Rys. 6.47. Schemat projektu <i>barrel_shift_register</i> na poziomie przesłań międzyrejestrowych	271
Rys. 6.48. Wyniki symulacji modułu <i>barrel_shift_register</i>	271
Rys. 6.49. Schemat projektu <i>linear_feedback_shift_register</i> na poziomie RTL	272
Rys. 6.50. Wyniki symulacji projektu <i>linear_feedback_shift_register</i>	273
Rys. 6.51. Schemat projektu bloku rejestrów <i>register_file</i> na poziomie RTL	273
Rys. 7.1. Struktura 4-bitowego licznika na przerzutnikach typu T	277

Rys. 7.2. Wynik syntezy przerzutnika T z asynchronicznym zerowaniem	278
Rys. 7.3. Wyniki symulacji przerzutnika T z asynchronicznym zerowaniem	278
Rys. 7.4. Wynik syntezy 4-bitowego licznika zbudowanego z przerzutników T	279
Rys. 7.5. Wyniki symulacji 4-bitowego licznika zbudowanego z przerzutników T	279
Rys. 7.6. Wynik syntezy sparametryzowanego N-bitowego licznika dla $N=4$	280
Rys. 7.7. Wynik syntezy licznika modulo M dla $M=16$	282
Rys. 7.8. Wynik syntezy licznika modulo 10 zaprojektowanego na podstawie sparametryzowanego modułu <i>counter_mod_M</i>	282
Rys. 7.9. Wyniki symulacji licznika modulo 10	282
Rys. 7.10. Wynik syntezy licznika modulo M z wyjściem <i>rollover</i> dla $M=16$	283
Rys. 7.11. Wyniki symulacji licznika z wyjściem <i>rollover</i>	283
Rys. 7.12. Wynik syntezy licznika z asynchronicznym ładowaniem danych.....	285
Rys. 7.13. Wyniki symulacji licznika z asynchronicznym ładowaniem danych.....	285
Rys. 7.14. Wynik syntezy licznika z synchronicznym ładowaniem danych.....	286
Rys. 7.15. Wyniki symulacji licznika z synchronicznym ładowaniem danych.....	286
Rys. 7.16. Wynik syntezy licznika z synchronicznym ładowaniem danych i synchronicznym zerowaniem	287
Rys. 7.17. Wyniki symulacji licznika z synchronicznym ładowaniem danych i synchronicznym zerowaniem	288
Rys. 7.18. Wynik syntezy licznika rewersyjnego z asynchronicznym ładowaniem danych ..	289
Rys. 7.19. Wyniki symulacji licznika rewersyjnego z asynchronicznym ładowaniem danych	289
Rys. 7.20. Wynik syntezy licznika rewersyjnego z jednym sumatorem.....	290
Rys. 7.21. Opis licznika w postaci grafu automatu skończonego	291
Rys. 7.22. Wynik syntezy 4-bitowego licznika opisanego w postaci automatu skończonego klasy C.....	293
Rys. 7.23. Mapa technologiczna 4-bitowego licznika opisanego w postaci automatu skończonego klasy C.....	294
Rys. 7.24. Graf zaprojektowanego automatu skończonego.....	295
Rys. 7.25. Wyniki symulacji 4-bitowego licznika opisanego w postaci automatu skończonego.....	295
Rys. 7.26. Schemat projektu <i>gray_counter</i> na poziomie przesłań międzyrejestrowych.....	298
Rys. 7.27. Wyniki symulacji funkcjonalnej projektu <i>gray_counter</i>	299
Rys. 7.28. Schemat projektu <i>johnson_counter</i> na poziomie RTL.....	300
Rys. 7.29. Wyniki symulacji funkcjonalnej projektu <i>johnson_counter</i>	300
Rys. 7.30. Wynik syntezy schematu opóźnienia sygnału o M cykli zegarowych <i>clk</i> zbudowa- nego na podstawie rejestru przesuwnego w prawo.....	301
Rys. 7.31. Wyniki symulacji schematu opóźnienia sygnału o M cykli zegarowych dla $M=5$	301
Rys. 7.32. Czterobitowy licznik jako dzielnik częstotliwości.....	302
Rys. 7.33. Wynik syntezy dzielnika częstotliwości na poziomie RTL	303
Rys. 7.34. Wynik symulacji dzielnika częstotliwości dla $M=2$	303
Rys. 7.35. Wynik syntezy generatora impulsów	304

Rys. 7.36. Wyniki symulacji generatora impulsów z okresem równym pięciu cyklom sygnału zegarowego	305
Rys. 7.37. Schemat projektu modułu <i>slow_counter</i> na poziomie RTL	306
Rys. 7.38. Wyniki symulacji modułu <i>slow_counter</i>	306
Rys. 7.39. Schemat powolnego licznika o okresie 1s na poziomie RTL	307
Rys. 7.40. Wynik syntezy czasomierza	308
Rys. 7.41. Wyniki symulacji czasomierza	308
Rys. 7.42. Struktura zegara czasu rzeczywistego	310
Rys. 7.43. Wynik syntezy zegara czasu rzeczywistego	311
Rys. 8.1. Wynik syntezy pełnego sumatora jednobitowego	316
Rys. 8.2. Wyniki symulacji pełnego sumatora jednobitowego	316
Rys. 8.3. Wynik syntezy półsumatora	317
Rys. 8.4. Wyniki symulacji półsumatora	317
Rys. 8.5. Struktura <i>N</i> -bitowego sumatora z przeniesieniami szeregowymi	317
Rys. 8.6. Wynik syntezy sumatora z przeniesieniami szeregowymi dla <i>N</i> =4	318
Rys. 8.7. Wyniki symulacji sumatora z przeniesieniami szeregowymi	319
Rys. 8.8. Zredukowany pełny sumator	320
Rys. 8.9. Wynik syntezy zredukowanego pełnego sumatora RFA	320
Rys. 8.10. Wyniki symulacji zredukowanego pełnego sumatora RFA	321
Rys. 8.11. Struktura 4-bitowego sumatora z przeniesieniami równoległymi	321
Rys. 8.12. Wynik syntezy generatora przeniesień dla 4-bitowej grupy sumatorów	323
Rys. 8.13. Wyniki symulacji generatora przeniesień dla 4-bitowej grupy sumatorów	323
Rys. 8.14. Wynik syntezy 4-bitowego sumatora z przeniesieniami równoległymi	324
Rys. 8.15. Wyniki symulacji 4-bitowego sumatora z przeniesieniami równoległymi	324
Rys. 8.16. Struktura 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień szeregowych pomiędzy blokami	325
Rys. 8.17. Wynik syntezy 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień szeregowych pomiędzy blokami	326
Rys. 8.18. Wyniki symulacji 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień szeregowych pomiędzy blokami	326
Rys. 8.19. Struktura 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami	326
Rys. 8.20. Wynik syntezy 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami	328
Rys. 8.21. Wyniki symulacji 16-bitowego sumatora z 4-bitowych bloków z przeniesieniami równoległymi i realizacją przeniesień równoległych pomiędzy blokami	328
Rys. 8.22. Blok sumatora z wyborem przeniesienia CSEA	329

Rys. 8.23. Struktura 4-bitowego bloku sumatora z wyborem przeniesienia CSEA.....	329
Rys. 8.24. Struktura 16-bitowego sumatora z wyborem przeniesienia CSEA zestawionego z 4-bitowych bloków	330
Rys. 8.25. Wynik syntezy 16-bitowego sumatora z wyborem przeniesienia CSEA zbudowanego z 4-bitowych bloków.....	331
Rys. 8.26. Wyniki symulacji 16-bitowego sumatora z wyborem przeniesienia CSEA zbudowanego z 4-bitowych bloków.....	331
Rys. 8.27. Wynik syntezy 16-bitowego sumatora z wyborem przeniesienia CSEA utworzonego z bloków zmiennego rozmiaru.....	333
Rys. 8.28. Wyniki symulacji 16-bitowego sumatora z wyborem przeniesienia CSEA utworzonego z bloków zmiennego rozmiaru.....	333
Rys. 8.29. 4-bitowy blok sumatora z przeskokiem przeniesień CSKA.....	333
Rys. 8.30. Wynik syntezy 16-bitowego sumatora z przeskokiem przeniesień CSKA	335
Rys. 8.31. Wyniki symulacji 16-bitowego sumatora z przeskokiem przeniesień CSKA	335
Rys. 8.32. Wynik syntezy 16-bitowego sumatora z sumami warunkowymi CSA	338
Rys. 8.33. Wyniki symulacji 16-bitowego sumatora z sumami warunkowymi CSA	338
Rys. 8.34. Wynik syntezy sumatora-subtraktora dla $N=4$	343
Rys. 8.35. Wyniki symulacji sumatora-subtraktora dla $N=4$	343
Rys. 8.36. Struktura sumatora szeregowego na podstawie jednego pełnego sumatora i rejestrów przesuwnych.....	344
Rys. 8.37. Wynik syntezy szeregowego sumatora na podstawie jednego pełnego sumatora.	346
Rys. 8.38. Wyniki symulacji szeregowego sumatora na podstawie jednego pełnego sumatora	346
Rys. 8.39. Struktura akumulatora.....	347
Rys. 8.40. Wynik syntezy sumatora akumulującego.....	348
Rys. 8.41. Wyniki symulacji sumatora akumulującego.....	348
Rys. 8.42. Struktura sumatora dwóch cyfr BCD	348
Rys. 8.43. Wynik syntezy sumatora dwóch cyfr BCD.....	350
Rys. 8.44. Wyniki symulacji sumatora dwóch cyfr BCD	350
Rys. 8.45. Struktura 4-bitowego inkrementatora.....	351
Rys. 8.46. Wynik syntezy inkrementatora z przeniesieniem szeregowym dla $N=4$	352
Rys. 8.47. Wyniki syntezy inkrementatora z przeniesieniem szeregowym dla $N=4$	352
Rys. 8.48. Wynik syntezy inkrementatora z przeniesieniem równoległym dla $N=4$	354
Rys. 8.49. Wyniki symulacji inkrementatora z przeniesieniem równoległym dla $N=4$	354
Rys. 8.50. Wynik syntezy 16-bitowego inkrementatora z przeniesieniem równoległym.....	355
Rys. 8.51. Wyniki symulacji 16-bitowego inkrementatora z przeniesieniem równoległym..	355
Rys. 8.52. Wynik syntezy modułu <i>half_sub</i>	356
Rys. 8.53. Wyniki symulacji modułu <i>half_sub</i>	356
Rys. 8.54. Wynik syntezy 4-bitowego dekrementatora z przeniesieniem szeregowym.....	357
Rys. 8.55. Wyniki symulacji 4-bitowego dekrementatora z przeniesieniem szeregowym	357
Rys. 8.56. Wynik syntezy licznika zbudowanego na podstawie rejestru i inkrementatora....	358

Rys. 8.57. Wyniki symulacji licznika zbudowanego na podstawie rejestru i inkrementatora.....	358
Rys. 9.1. Mnożenie liczb w systemie dziesiętnym.....	362
Rys. 9.2. Mnożenie liczb w dwójkowym systemie liczbowym.....	362
Rys. 9.3. Mnożenie 4-bitowych liczb dwójkowych.....	363
Rys. 9.4. Struktura matrycowego układu mnożącego z przechowywaniem przeniesień dla 4-bitowych liczb dwójkowych.....	363
Rys. 9.5. Struktura matrycowego układu mnożącego 4×4 bity po uproszczeniu.....	364
Rys. 9.6. Matrycowy układ mnożący 4×4 bitów z dodawaniem iloczynów częściowych w kolumnach	364
Rys. 9.7. Wynik syntezy modułu matrycowego układu mnożącego z przechowywaniem przeniesień dwóch 4-bitowych liczb dwójkowych.....	367
Rys. 9.8. Wyniki symulacji modułu matrycowego układu mnożącego z przechowywaniem przeniesień dwóch 4-bitowych liczb dwójkowych.....	367
Rys. 9.9. Struktura matrycowego układu mnożącego z wykorzystaniem n-bitowych sumatorów	368
Rys. 9.10. Wynik syntezy modułu matrycowego układu mnożenia z wykorzystaniem sumatorów dwóch 4-bitowych liczb dwójkowych	369
Rys. 9.11. Wyniki symulacji modułu matrycowego układu mnożenia z wykorzystaniem sumatorów dwóch 4-bitowych liczb dwójkowych	369
Rys. 9.12. Dodawanie iloczynów częściowych w jednej kolumnie matrycowego układu mnożącego.....	371
Rys. 9.13. Zmniejszenie liczby poziomów sumatorów przy dodawaniu iloczynów częściowych w jednej kolumnie matrycowego układu mnożącego.....	372
Rys. 9.14. Struktura układu mnożącego Wallace'a 4×4	377
Rys. 9.15. Wynik syntezy układu mnożącego Wallace'a 4×4 bity	378
Rys. 9.16. Wyniki symulacji układu mnożącego Wallace'a 4×4 bity	379
Rys. 9.17. Diagram kropkowy oryginalnej macierzy układu mnożącego 4×4 bity.....	379
Rys. 9.18. Diagram kropkowy zreorganizowanej macierzy mnożenia 4×4 bity.....	379
Rys. 9.19. Diagramy kropkowe procesu syntezy układu mnożącego Wallace'a 4×4 bity	380
Rys. 9.20. Diagramy kropkowe syntezy układu mnożącego Dadda 4×4 bity	381
Rys. 9.21. Struktura matrycowego układu mnożącego Dadda 4×4 bity.....	382
Rys. 9.22. Wynik syntezy układu mnożącego Dadda 4×4 bity na poziomie przesłań międzyrejestrowych	384
Rys. 9.23. Wyniki symulacji układu mnożącego Dadda 4×4 bity	384
Rys. 9.24. Struktury układów mnożących: a) z przesunięciem mnożnika w prawo, mnożnej w lewo, analizowany jest młodszy bit mnożnika; b) z przesunięciem mnożnika w lewo, mnożnej w prawo, analizowany jest starszy bit mnożnika; c) z przesunięciem mnożnika w lewo, akumulatora w lewo, analizowany jest starszy bit mnożnika; d) z przesunięciem mnożnika w prawo, akumulatora w prawo, analizowany jest młodszy bit mnożnika	385
Rys. 9.25. Wynik syntezy modułu <i>algorithm_mult</i> na poziomie przesłań międzyrejestrowych.....	387

Rys. 9.26. Wyniki symulacji modułu <i>algorithm_mult</i>	387
Rys. 9.27. Wynik syntezy modułu <i>algorithm_mult_b</i> na poziomie przesłań międzyrejestrowych.....	388
Rys. 9.28. Wyniki symulacji modułu <i>algorithm_mult_b</i>	388
Rys. 9.29. Wynik syntezy układu mnożącego Bootha na poziomie przesłań międzyrejestrowych.....	394
Rys. 9.30. Wyniki modelowania układu mnożącego Bootha.....	394
Rys. 9.31. Wynik syntezy zmodyfikowanego układu mnożącego Bootha.....	397
Rys. 9.32. Wyniki symulacji zmodyfikowanego układu mnożącego Bootha.....	398
Rys. 10.1. Struktury automatów skończonych: a) – automat klasy A (Mealy’ego); b) – automat klasy B (Moore’a); c) – automat klasy C (Moore’a); d) – automat klasy D (Mealy’ego); e) – automat klasy E (Mealy’ego); f) – automat klasy F (Moore’a).....	403
Rys. 10.2. Przykładowy graf automatu Mealy’ego	406
Rys. 10.3. Przykład grafu automatu Moore’a	408
Rys. 10.4. Wynik syntezy projektu <i>Mealy_style_1</i> na poziomie przesłań międzyrejestrowych.....	412
Rys. 10.5. Graf zaprojektowanego automatu Mealy’ego.....	412
Rys. 10.6. Tablica przejść zaprojektowanego automatu Mealy’ego	412
Rys. 10.7. Kody stanów wewnętrznych zaprojektowanego automatu Mealy’ego	413
Rys. 10.8. Plik przebiegów czasowych	414
Rys. 10.9. Wyniki symulacji funkcjonalnej automatu Mealy’ego.....	414
Rys. 10.10. Wyniki symulacji czasowej automatu Mealy’ego	415
Rys. 10.11. Wynik syntezy projektu <i>Moore_style_1</i> na poziomie przesłań międzyrejestrowych	418
Rys. 10.12. Graf zaprojektowanego automatu Moore’a	418
Rys. 10.13. Tablica przejść zaprojektowanego automatu Moore’a	419
Rys. 10.14. Kody stanów wewnętrznych zaprojektowanego automatu Moore’a.....	419
Rys. 10.15. Wyniki symulacji funkcjonalnej automatu Moore’a.....	420
Rys. 10.16. Wyniki symulacji czasowej automatu Moore’a.....	421
Rys. 10.17. Wynik syntezy projektu <i>Mealy_style_4</i> na poziomie przesłań międzyrejestrowych	427
Rys. 10.18. Wyniki symulacji funkcjonalnej projektu <i>Mealy_style_4</i>	427
Rys. 10.19. Wyniki symulacji czasowej projektu <i>Mealy_style_4</i>	427
Rys. 10.20. Wynik syntezy projektu <i>Moore_1_process</i> na poziomie przesłań międzyrejestrowych	429
Rys. 10.21. Wynik syntezy projektu <i>Mealy_style_6</i> na poziomie przesłań międzyrejestrowych	431
Rys. 10.22. Wynik syntezy projektu <i>Mealy_style_8</i> na poziomie przesłań międzyrejestrowych	435
Rys. 10.23. Kodowanie stanów wewnętrznych automatu Moore’a w trybie <i>One-Hot</i>	439
Rys. 10.24. Kodowanie stanów wewnętrznych automatu Moore’a w trybie <i>Gray</i>	439

Rys. 10.25. Kodowanie stanów wewnętrznych automatu Moore'a w trybie <i>Johnson</i>	440
Rys. 10.26. Kodowanie stanów wewnętrznych automatu Moore'a w trybie <i>Minimal Bits</i>	440
Rys. 10.27. Kodowanie stanów wewnętrznych automatu Moore'a w trybie <i>Sequential</i>	440
Rys. 10.28. Kodowanie stanów wewnętrznych automatu Moore'a w trybie <i>User-Encoded</i> ...	440
Rys. 10.29. Struktura automatu klasy C	448
Rys. 10.30. Mapa technologiczna projektu <i>Moore_C</i> z opcją State Machine Processing ustawioną na wartość User-Encoded	452
Rys. 10.31. Wynik syntezy projektu <i>Moore_C</i> na poziomie przesłań międzyrejestrowych z wyłączoną opcją Extract Verilog State Machines	452
Rys. 10.32. Rozszczępienie stanów wewnętrznych przy syntezie automatów klasy D	453
Rys. 10.33. Struktura automatu klasy D	454
Rys. 10.34. Wynik syntezy projektu <i>Mealy_D</i> na poziomie przesłań międzyrejestrowych ...	457
Rys. 10.35. Wynik syntezy projektu <i>Mealy_D_1</i> na poziomie przesłań międzyrejestrowych	459
Rys. 10.36. Rozszczępienie stanów wewnętrznych przy syntezie automatów klasy E	460
Rys. 10.37. Struktura automatu klasy E	460
Rys. 10.38. Graf ortogonalności wierszy macierzy W przy kodowaniu stanów automatu klasy E	463
Rys. 10.39. Wynik syntezy projektu <i>Mealy_E</i> na poziomie przesłań międzyrejestrowych	465
Rys. 10.40. Graf <i>H</i> przy syntezie automatu klasy E_1 bez rozszczępienia stanów wewnętrznych	466
Rys. 10.41. Wynik syntezy projektu <i>Mealy_E_1</i> na poziomie przesłań międzyrejestrowych	467
Rys. 10.42. Struktura automatu klasy F	468
Rys. 10.43. Graf <i>H</i> ortogonalności wierszy macierzy W przy kodowaniu stanów wewnętrznych automatu klasy F	470
Rys. 10.44. Wynik syntezy projektu <i>Moore_F</i> na poziomie przesłań międzyrejestrowych	472

Streszczenie

Książka stanowi wprowadzenie do projektowania systemów wbudowanych w układach FPGA z wykorzystaniem języka Verilog. Autorzy definiują, czym są systemy wbudowane, wskazując jednocześnie na szeroki wachlarz ich zastosowań oraz realizowane funkcje. Przedstawione zostały podstawy języka opisu sprzętu Verilog ze szczególnym uwzględnieniem jego możliwości w kontekście projektowania systemów wbudowanych na FPGA. Ponadto czytelnik otrzymuje wprowadzenie do systemu zautomatyzowanego projektowania Quartus. W książce omawiane są zagadnienia projektowania w strukturach FPGA układów kombinacyjnych oraz różnych bloków funkcjonalnych, zarówno kombinacyjnych, jak i sekwencyjnych, takich jak multipleksery, kodery, dekodery, rejestry, liczniki, czasomierze i wiele innych. Opisano również metody projektowania układów arytmetycznych, w tym sumatorów oraz układów mnożących. Rozpatrzono zagadnienia projektowania automatów skończonych na FPGA, przedstawiono ich modele strukturalne, style opisu w języku Verilog, sposoby kodowania stanów wewnętrznych oraz metody projektowania. Książka skupia się na praktycznych aspektach projektowania, prezentując liczne przykłady projektów w języku Verilog, dokonuje szczegółowej analizy różnorodnych technik syntezy oraz proponuje rekomendacje efektywnego wykorzystania metod projektowania.

Abstract

The book serves as an introduction to designing embedded systems based on FPGA devices using the Verilog language. It defines the concept of embedded systems, specifies the scope of applications, and functions performed by embedded systems. The fundamentals of the Verilog hardware description language are presented, with attention to its capabilities in the context of designing embedded systems on FPGAs. Additionally, it introduces the reader to the Quartus automated design system. The book covers the design issues of combinational and various functional blocks in FPGA structures, both combinational and sequential, such as multiplexers, encoders, decoders, registers, counters, timers, and many others. Methods for designing arithmetic circuits, such as adders and multipliers, are described. Finite state machine design issues on FPGAs are addressed, presenting their structural models, Verilog description styles, methods of encoding internal states, and design methodologies. The book focuses on the practical aspects of design, presenting numerous project examples in Verilog, conducting a detailed analysis of various synthesis techniques, and proposing recommendations for the effective use of design methods.

