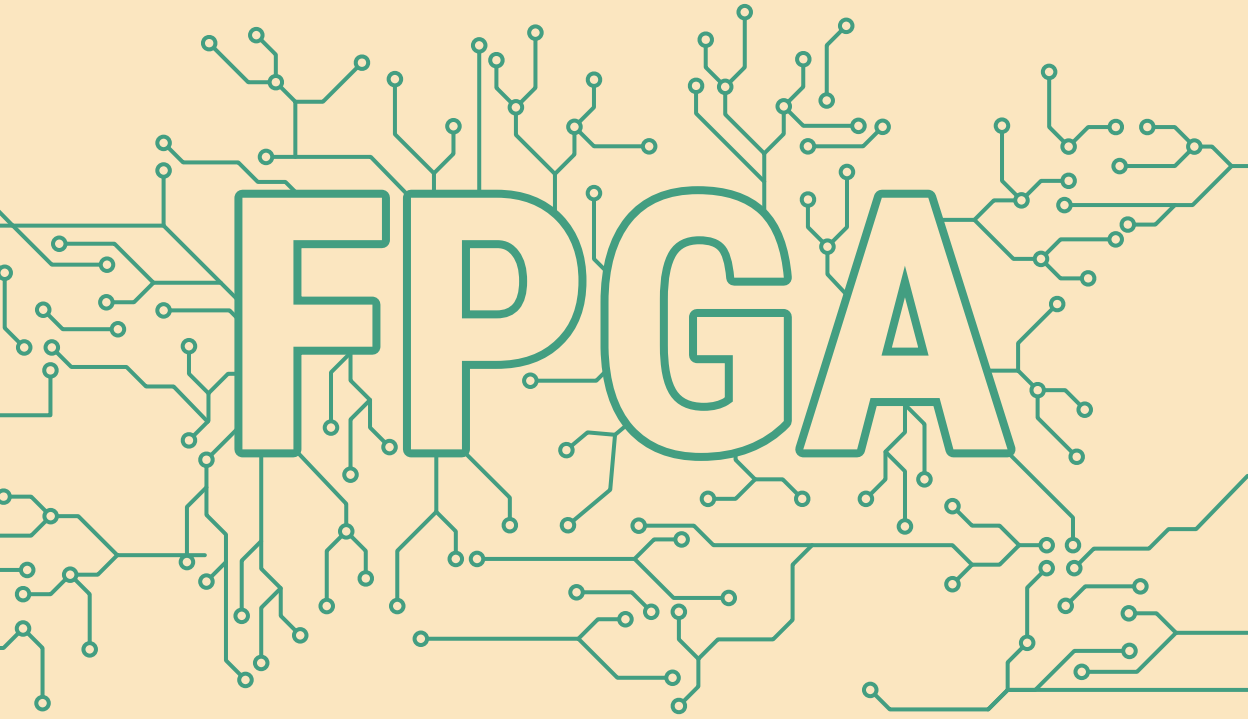
SYNTEZA SYSTEMÓW CYFROWYCH W JĘZYKU SYSTEMVERILOG

Valery Salauyou
Adam Klimowicz



Politechnika Białostocka

Valery Salauyou • Adam Klimowicz

SYNTEZA SYSTEMÓW CYFROWYCH W JĘZYKU SYSTEMVERILOG

Politechnika Białostocka

OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2025

Recenzent:
prof. dr hab. inż. Alexander Barkalov

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

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-edytor-8505529/

ISBN 978-83-68077-70-4 (e-Book)
DOI: 10.24427/978-83-68077-70-4

Oficyna Wydawnicza Politechniki Białostockiej
ul. Wiejska 45C, 15-351 Białystok
e-mail: oficyna.wydawnicza@pb.edu.pl
www.pb.edu.pl

Spis treści

Wykaz skrótów

ALU – *arithmetic logic unit* (jednostka arytmetyczno-logiczna)
ANSI – American National Standards Institute (Amerykański Narodowy Instytut ds. Standardów)
API – *application programming interface* (interfejs programowania aplikacji)
ASIC – *application-specific integrated circuit* (układ scalony specyficzny dla zastosowania)
DRI – *direct programming interface* (interfejs programowania bezpośredniego)
ECL – *emitter-coupled logic* (logika sprzężona emiterowo)
EDA – *electronic design automation* (automatyzacja projektowania elektronicznego)
FPGA – *field programmable gate array* (układ bramek programowalnych)
HDL – *hardware description language* (język opisu sprzętu)
IEEE – Institute of Electrical and Electronics Engineers (Instytut Inżynierów Elektryków i Elektroników)
IP – *intellectual property* (własność intelektualna)
MOS – *metal oxide semiconductor* (półprzewodnik metalowo-tlenkowy)
PLI – *programming language interface* (interfejs języka programowania)
RAM – *random access memory* (pamięć o dostępie swobodnym)
RCA – *ripple-carry adder* (sumator z szeregowym przeniesieniem)
ROM – *read-only memory* (pamięć tylko do odczytu)
RTL – *register transfer level* (poziom przesłań międzyrejestrowych)
SDF – *standard delay format* (standardowy format opóźnienia)
UDP – *user defined primitive* (prymityw zdefiniowany przez użytkownika)

Wstęp

Postęp technologiczny ciągle stwarza nowe wyzwania dla projektantów sprzętu cyfrowego, a systemy i zadania, które wykonują, stają się coraz bardziej złożone, co wymaga automatyzacji procesu projektowania sprzętu na wyższych poziomach. Z jednej strony stwarza to nowe wyzwania przy projektowaniu sprzętu, zwłaszcza w przypadku złożonych systemów z dużą liczbą komponentów i różnorodnymi połączeniami między nimi. Ważną rolę we współczesnych systemach cyfrowych odgrywają również transakcje, które są obsługiwane przez jednostki sprzętowe komunikujące się za pomocą standardowych lub własnych protokołów transmisji danych.

Z drugiej strony te złożone jednostki i systemy cyfrowe muszą być testowane, najlepiej na wszystkich poziomach projektowania. Jednym ze sposobów weryfikacji urządzeń i systemów jest symulacja sterowana zdarzeniami wykonywana na komputerze przy użyciu narzędzi programowych. Należy także zauważyć, że rosnącej złożoności systemów cyfrowych towarzyszą zaostrzone wymagania dotyczące ograniczonego czasu projektowania i zwiększonej niezawodności projektów.

Praktyka projektowania inżynierskiego pokazuje, że możliwości języka opisu sprzętu Verilog mogą być niewystarczające do tworzenia nowoczesnych systemów cyfrowych. W wyniku aktywnej pracy różnych grup deweloperów zaproponowano ulepszenie języka Verilog, co doprowadziło do powstania nowego języka projektowania sprzętu cyfrowego – SystemVerilog.

W pełni dziedziczy on właściwości języka Verilog i dostarcza nowych możliwości tworzenia dużych i złożonych projektów na najwyższych poziomach projektowania: systemowym, abstrakcyjnym i transakcyjnym. Ponadto SystemVerilog zawiera nowe konstrukcje językowe do weryfikacji projektów, dzięki którym na nowo można opisać środowisko testowe złożonego projektu. W rezultacie powstał język o bardzo szerokich i różnorodnych możliwościach, które trudno przedstawić szczegółowo w jednej książce. W proponowanej pracy omówiono głównie konstrukcje języka SystemVerilog, które są przeznaczone do syntezy. Nie można jednak całkowicie pominąć pewnych konstrukcji językowych używanych wyłącznie do symulacji, dlatego w tej pracy również zostaną wspomniane.

W rozdziale 1 w skrócie przedstawiono historię języka SystemVerilog i jego pierwotne źródła, omówiono podstawowe elementy i podstawowe konstrukcje języka oraz wskazano ulepszenia języka SystemVerilog w zakresie syntezy.

W rozdziale 2 opisano moduły. Moduł jest główną jednostką strukturalną języków Verilog i SystemVerilog. Nowością w języku SystemVerilog jest wygoda przekazywania sygnałów przez porty modułów, gdy nazwy sygnałów są takie same jak nazwy portów. Oprócz zmiennych i sieci język SystemVerilog umożliwia przekazywanie przez porty modułów tablic, struktur, unii i interfejsów. Dodatkowo zmienne mogą być przekazywane przez referencję. Nowością jest również to, że typy danych mogą być używane jako parametry modułu.

W rozdziale 3 scharakteryzowano główne typy danych: sieci, zmienne, wektory, stałe, liczby całkowite, liczby rzeczywiste i ciągi znaków. W języku Verilog każdy bit danych może przyjmować 4 wartości lub stany. Jednak na poziomie abstrakcyjnym i w językach programowania stosuje się typy o dwóch stanach. W języku SystemVerilog wprowadzono nowe typy danych: **logic** do reprezentowania danych o czterech stanach (czterowartościowych) i **bit** do reprezentowania danych o dwóch stanach (dwuwartościowych). Ponadto typy liczb całkowitych i rzeczywistych zostały rozszerzone tak, aby były zgodne z językiem programowania C. W języku SystemVerilog, w porównaniu z Verilogiem, znacznie rozszerzono możliwości deklarowania i inicjowania zmiennych.

Rozdział 4 zawiera omówienia typów zdefiniowanych przez użytkownika i typów wyliczeniowych. Jeśli podstawowe typy danych nie są wystarczające do reprezentacji danych w projekcie, użytkownik może zadeklarować własne typy danych. Wyliczeniowe typy danych zapożyczono z języka programowania C.

W rozdziale 5 opisano kompatybilność i konwersje typów danych oraz wprowadzono poziomy kompatybilności typów danych. Jeśli typy danych obiektów są niekompatybilne w SystemVerilog, jeden typ danych może być konwertowany na inny za pomocą rzutowania typów lub funkcji systemowej.

Rozdział 6 przedstawia struktury i unie w SystemVerilog, które są w dużej mierze takie same jak w języku C. Szczególną cechą struktur i unii w SystemVerilog jest to, że mogą być rozpakowywane lub pakowane (jako wektor bitowy), przekazywane przez porty modułów oraz mogą być argumentami zadań i funkcji. Ponadto język SystemVerilog ma unie oznaczone, które zapewniają dodatkową warstwę ochrony przed przypadkowym odczytem informacji.

W rozdziale 7 omówiono tablice. Ponieważ są one bardzo szeroko stosowane w projektach na poziomie systemu, język SystemVerilog znacznie rozszerza możliwości pracy z tablicami. W tym celu wprowadzono nowe operacje na tablicach, dodano funkcje systemowe i metody pracy oraz specjalny operator pętli dla elementów tablicowych.

Rozdział 8 jest poświęcony metodom i sposobom deklarowania elementów złożonych projektów. Nowością są tu pakiety i jednostki kompilacji, a także sposoby odwoływania się do nazw w drzewie hierarchii projektu. Pakiety umożliwiają jednorazowe zadeklarowanie elementów projektu, a następnie wykorzystanie tych deklaracji w dowolnej jednostce projektowej. Język SystemVerilog umożliwia kompilację dużych projektów w częściach. Jednostka kompilacji to pewna część projektu reprezentująca pliki źródłowe, które są kompilowane w tym samym czasie.

W rozdziale 9 omówiono operacje i wyrażenia. W SystemVerilog operacje języka Verilog są rozszerzone o operacje języka programowania C, operacje przynależności do zbioru, operacje dystrybucji i operacje strumieniowe. Nowością w języku SystemVerilog są również wyrażenia agregujące, w których można stosować struktury spakowane i tablice.

W rozdziale 10 przedstawiono operatory przypisania, które są takie same jak w języku Verilog. W języku SystemVerilog dodano nowe operatory przypisania z oddzielnymi operacjami, tak jak w języku C.

W rozdziale 11 skupiono się na procesach. W SystemVerilog proces jest ogólnym pojęciem zbioru czynności służących do przetwarzania danych. Jest tworzony przez procedury lub operatory proceduralne. W języku SystemVerilog rozszerzono procedury strukturalne i dodano precyzyjną kontrolę nad procesami.

W rozdziale 12 opisano operatory proceduralne. Język SystemVerilog wprowadza kwalifikatory unikalności i priorytetu do operatorów warunkowych i wyboru oraz rozszerza listę operatorów pętli i przejść.

W rozdziale 13 scharakteryzowano zadania i funkcje języka SystemVerilog. Ponieważ usuwa on większość ograniczeń dotyczących zadań i funkcji Verilog, dzięki niemu zadania i funkcje są równoważne z podobnymi elementami spotykanymi w językach programowania i upraszczają tworzenie dużych, złożonych projektów.

W rozdziale 14 wprowadzono pojęcie interfejsów. Są to nowe mechanizmy opisywania połączeń między komponentami w dużym projekcie. Umożliwiają one zastąpienie grupy nazw portów jedną nazwą, dzięki czemu programista może skupić się na funkcjonalności systemu, a nie na sprawdzaniu, czy sygnały są prawidłowo podłączone do portów modułu. Interfejsy mogą mieć własną funkcjonalność odpowiadającą protokołowi transmisji danych. Język SystemVerilog umożliwia także tworzenie drzewiastej struktury interfejsów łączących elementy złożonego systemu.

Rozdział 15 jest poświęcony automatycznemu generowaniu kodu. Konstrukcje SystemVerilog do generowania kodu pozwalają na generowanie sekwencji powtarzających się fragmentów kodu oraz na wybieranie poszczególnych fragmentów kodu na podstawie warunku. Konstrukcje generowania kodu języka SystemVerilog pełnią funkcje preprocesora języka programowania.

W rozdziale 16 opisano dyrektywy kompilatora języka SystemVerilog. Przekazują one kompilatorowi instrukcje dotyczące sposobu interpretowania kodu źródłowego projektu. Niektóre dyrektywy języka SystemVerilog są takie same jak dyrektywy języków programowania (dyrektywy dla makrodefinicji, kompilacji warunkowej, dołączania plików). Inne zaś są specyficzne dla języka SystemVerilog (dyrektywy dotyczące definiowania wartości jednostek czasu, definiowania domyślnych typów, definiowania wartości logicznych dla wejść, które nie są połączone, itp.).

W rozdziale 17 omówiono prymitywy języka SystemVerilog. Jest to dość ubogi zestaw predefiniowanych prymitywów, jednak udostępnia on użytkownikowi mechanizm tworzenia prymitywów językowych. Użytkownik może tworzyć prymitywy kombinacyjne albo sekwencyjne. Te ostatnie mogą być wrażliwe na poziom lub zbocze sygnału sterującego, jak również jednocześnie na poziom i na zbocze.

Rozdział 18 dotyczy konfiguracji projektu. Jest ona bezpośrednio związana z problemami tworzenia dużych projektów przez kilka grup deweloperów. Konfiguracje projektu pozwalają uniknąć powielania nazw modułów i elementów prymitywnych w różnych częściach projektu. Wskazują również fizyczną lokalizację plików kodu źródłowego projektu.

W dodatku znajduje się lista słów kluczowych języka SystemVerilog.

Każdy rozdział ma podobną strukturę. Na początku znajduje się opis problemów, które następnie są rozwiązywane za pomocą języka SystemVerilog. Rozdziały kończą się wnioskami zawierającymi podsumowania omówionych punktów, ponadto skupiają uwagę czytelnika na najważniejszych kwestiach, które mogły mu umknąć podczas czytania.

Książka jest przeznaczona przede wszystkim dla konstruktorów i deweloperów systemów cyfrowych oraz studentów uczelni technicznych. Materiały te mogą być wykorzystywane przez nauczycieli do prowadzenia wykładów, zajęć laboratoryjnych i praktycznych. W publikacji zawarto wiele szczegółowych opisów, dzięki czemu może być ona źródłem informacji o języku SystemVerilog.

Powstanie tej książki zostało częściowo sfinansowane ze środków Politechniki Białostockiej na badania naukowe oraz z subwencji badawczej przekazanej przez Ministerstwo Edukacji i Nauki.

1. Wprowadzenie do języka SystemVerilog

Język SystemVerilog w odróżnieniu od języków programowania jest *językiem opisu sprzętu* (*hardware description language* – HDL) lub *językiem projektowania*. Służy do dwóch celów: opisu modeli projektowanych urządzeń oraz przeprowadzania symulacji sterowanej zdarzeniami.

Aby zaprojektować jakieś urządzenie cyfrowe, należy je opisać w języku SystemVerilog, a następnie automatycznie zsyntetyzować za pomocą *syntezatora* (*synthesizer*), który jest częścią *kompilatora* (*compiler*).

Opis urządzenia w jakimś języku projektowania (tzn. stworzenie modelu urządzenia), a nawet zaimplementowanie go w pewnej bazie technologicznej (np. FPGA, ASIC itp.) zwykle nie wystarcza, aby stwierdzić, że urządzenie pracuje stabilnie. Dlatego obowiązkowym etapem projektowania jest symulacja urządzenia cyfrowego. Weryfikację działania modeli sprzętowych przeprowadza się za pomocą *symulacji zdarzeniowej* (*event simulation*). W tym celu w SystemVerilog opisuje się *testbench*, czyli środowisko (stanowisko) testowe, a symulację zdarzeń w systemie wykonuje się za pomocą specjalnego *symulatora* (*Simulator*).

Cechą wyróżniającą język SystemVerilog jest to, że za pomocą tego samego języka można opisać zarówno model urządzenia przeznaczony do syntezy (tzn. do implementacji sprzętowej), jak i model układu do symulacji (tzn. do symulacji sterowanej zdarzeniami) projektu. Ponadto oba te modele projektu mogą się znacząco od siebie różnić.

Język SystemVerilog jest przeznaczony do tworzenia dużych i złożonych projektów, głównie na abstrakcyjnych poziomach projektowania (stąd słowo System w jego nazwie). Zawiera wiele konstrukcji obejmujących różne aspekty syntezy i symulacji systemów cyfrowych. Jednak nie wszystkie konstrukcje języka SystemVerilog są syntezowalne, tzn. mogą być zaimplementowane w sprzęcie. Znaczna ich część jest przeznaczona wyłącznie do symulacji. W niniejszej książce omówiono podzbiór konstrukcji języka SystemVerilog przeznaczony głównie do syntezy, a także niektóre konstrukcje służące tylko do symulacji.

1.1. Historia języka SystemVerilog

Język SystemVerilog pojawił się po raz pierwszy w czerwcu 2002 r. jako standard firmy Accellera o nazwie SystemVerilog 3.0. Wersja 3.0 oznacza trzecią generację języka Verilog, przy czym Verilog-1995 jest uważany za pierwszą generację, a Verilog-2001 za drugą.

W maju 2003 r. Accellera wydała SystemVerilog 3.1, w którym dodano znaczną liczbę funkcji służących do weryfikacji projektów.

Rok później, w maju 2004 r., Accellera wprowadziła SystemVerilog 3.1a, w którym zdefiniowano kilka dodatkowych konstrukcji do symulacji i weryfikacji. W następnym miesiącu SystemVerilog 3.1a został przekazany do IEEE (Institute of Electrical and Electronics Engineers) w celu stworzenia standardu języka SystemVerilog.

W listopadzie 2005 r. IEEE opublikował pierwszy oficjalny standard IEEE 1800-2005 dla języka SystemVerilog. Jeden z pierwszych opisów języka SystemVerilog jest przedstawiony w [2].

1.2. Podstawowe źródła języka SystemVerilog

Język SystemVerilog nie pojawił się znikąd. Jego poprzednikiem był język opisu sprzętu Verilog [3]. Przypomnijmy, że język Verilog powstał na bazie języka programowania C [8], który został przystosowany do opisu modeli urządzeń cyfrowych i wykonywania ich symulacji sterowanych zdarzeniami. Następnie języka tego użyto do syntezy urządzeń i systemów cyfrowych [4]. Wraz z rosnącą złożonością projektów systemów cyfrowych pojawiła się pilna potrzeba stworzenia języka do projektowania dużych i złożonych projektów. Językami opisu sprzętu na poziomie systemowym są języki SystemC i SpecC oraz właśnie język SystemVerilog. Tylko on jest obsługiwany przez systemy projektowe Quartus i Vivado przeznaczone do realizacji projektów na *układach* FPGA (*field programmable gate array*).

Pierwotnym twórcą języka SystemVerilog była firma Accellera. Jest to organizacja non profit, której celem jest wspieranie rozwoju i wykorzystania języków *automatyzacji projektowania elektronicznego* (*electronic design automation* – EDA). Pracując nad projektem języka SystemVerilog, Accellera nawiązała kontakt z wieloma firmami zajmującymi się projektowaniem elektronicznym. Przedsiębiorstwa te udostępniły jej swoje konstrukcje językowe, które zostały zintegrowane z językiem SystemVerilog.

Główne konstrukcje językowe, z których składa się SystemVerilog, to:

- syntetyzowalny podzbiór SUPERLOG ESS firmy Co-Design Automation;
- język weryfikacji OpenVERA firmy Synopsys;
- twierdzenia (*assertions*) PSL z IBM;
- twierdzenia OpenVERA firmy Synopsys;
- DirectC i *interfejsy do programowania aplikacji* (*application programming interfaces* – API) firmy Synopsys;
- oddzielna kompilacja i rozszerzenie $readmem firmy Mentor Graphics;
- *oznaczone unie* (*tagged unions*) i funkcje wysokiego poziomu z BlueSpec.

Należy zauważyć, że pomimo wszystkich rozszerzeń SystemVerilog zachowuje kompatybilność i wspólny styl z językiem Verilog.

1.3. Standardy języka SystemVerilog

W standardzie SystemVerilog 1800-2005 zawarto następujące kluczowe ulepszenia języka Verilog-2001:

- interfejsy do grupowania połączeń i sprawdzania protokołów w ramach projektu;
- typy danych języka C (np. **int**);
- typy zdefiniowane przez użytkownika (typy użytkownika);
- typy enumeratywne (typy wyliczeniowe);
- rzutowanie typów (*type casting*);
- struktury (*structures*) i unie (*unions*);
- pakiety (*packages*) definicji, które mogą być współdzielone przez kilka bloków projektu;
- definicja kontekstu (zakresu – *scope*) jednostki kompilacji (*compilation unit*);
- operacje przypisania z języka C: ++, --, -=, += itd.;
- bloki proceduralne (**always_comb**, **always_ff** i **always_latch**);
- kwalifikatory priorytetu (**priority**) i unikalności (**unique**) w operatorach **if** i **case**;
- doskonalenie operatorów proceduralnych;
- przesyłanie argumentów za pomocą referencji (**ref**) do zadań i funkcji oraz sygnałów do modułów.

Język SystemVerilog jest stale ulepszany. W ostatnich latach IEEE wydał następujące standardy SystemVerilog:

- SystemVerilog Std 1800-2009;
- SystemVerilog Std 1800-2009 (Rewizja 2005);
- SystemVerilog Std 1800-2012 (Rewizja 2009);
- SystemVerilog Std 1800-2017 (Rewizja 2012).

Standard Verilog Std 1364-2005 jest podstawą pierwszego standardu SystemVerilog Std 1800-2005. Te dwa standardy zostały opracowane tak, aby można było ich używać jako jednego języka, co nie zawsze jest wygodne. Standard SystemVerilog Std 1800-2009 łączy w jednym dokumencie dwa standardy Verilog Std 1364-2005 i SystemVerilog Std 1800-2005. W porównaniu z SystemVerilog Std 1800-2005 w standardzie SystemVerilog Std 1800-2009 dodano także kilka słów kluczowych.

W grudniu 2012 r. IEEE wprowadził nowy standard, SystemVerilog Std 1800-2012. Uwzględniono w nim doświadczenia związane z wykorzystaniem SystemVerilog w opisie, projektowaniu i weryfikacji sprzętu. Redakcja standardu z 2012 r. zawiera także udoskonalone funkcje, które upraszczają projektowanie, poprawiają weryfikację i ułatwiają komunikację między językami. W standardzie tym dodano też niewielką liczbę nowych słów kluczowych w porównaniu ze standardem z 2009 r.

W grudniu 2017 r. IEEE wprowadził standard SystemVerilog Std 1800-2017, w którym poprawiono błędy i uściślono niektóre aspekty poprzedniego standardu. W tej wersji wprowadzono także ulepszone funkcje ułatwiające projektowanie, weryfikację i upraszczające interakcje między językami.

W tej książce będziemy omawiać aspekty projektowania systemów zgodnie ze standardem SystemVerilog Std 1800-2017 [1].

1.4. Podstawowe elementy języka SystemVerilog

Podobnie jak języki programowania, kod źródłowy języka SystemVerilog składa się z *tokenów leksykalnych* (*lexical tokens*) w pliku tekstowym. Są one rozmieszczone w kodzie bez ograniczeń, co przyczynia się do wizualnego odbioru projektu. Innymi słowy – spacje, tabulatory i nowe linie mogą być wstawiane w dowolnym miejscu kodu źródłowego z wyjątkiem specjalnych przypadków (takich jak separatory tokenów i ukryte identyfikatory).

Leksemami w SystemVerilog są następujące elementy języka:

- *białe znaki* (*white spaces*);
- *komentarze* (*comments*);
- *operatory* (*operators*);
- *liczby* (*numbers*);
- *literały łańcuchowe* (*string literals*) lub *linie* (*wiersze*);
- *identyfikatory* (*identifiers*);
- *słowa kluczowe* (*keywords*).

1.4.1. Białe znaki

Białe znaki w kodzie źródłowym projektu są używane w celu ułatwienia czytania tekstu, tzn. służą poprawie *czytelności* kodu źródłowego. Do tych znaków w języku SystemVerilog należą:

- *spacje* (*spaces*);
- *znaki tabulacji* (*tabs*);
- *znaki nowej linii* (*newlines*).

Kompilator ignoruje znaki białych spacji, chyba że służą one do oddzielenia innych tokenów. Również znaki białych spacji i tabulacji są traktowane jako znaki znaczące w literałach łańcuchowych.

1.4.2. Komentarze

W SystemVerilog stosowane są dwie formy komentarzy: *jednowierszowy* (*one-line comment*) i *blokowy* (*block comment*).

Komentarz jednowierszowy zaczyna się od dwóch znaków // i kończy znakiem nowej linii. Blokowy zaś zaczyna się parą symboli /* i kończy parą symboli */, może też obejmować kilka wierszy kodu. W języku SystemVerilog komentarze blokowe nie mogą być zagnieżdżane.

1.4.3. Operatory

Operatory SystemVerilog obejmują operatory *jednoargumentowe* (*unary operators*), *dwuargumentowe* (*binary operators*) i *warunkowe* (*conditional operator*).

Operatory jednoargumentowe znajdują się po lewej stronie operandu, np.: -a, +b, -8d6.

Operatory dwuargumentowe są umieszczane między operandami, np.: a+b, c*d, e&f.

Operator warunkowy ma dwa znaki oddzielające trzy operandy, np.: (a==b) ? c : d.

Operatory języka SystemVerilog zostały szczegółowo omówione w rozdziale 9.

1.4.4. Identyfikatory

Służą do przypisania obiektowi unikatowej nazwy. Identyfikatory SystemVerilog są używane jako nazwy zmiennych, modułów, funkcji, instancji modułów itp. Dzielą się na *identyfikatory proste* (*simple identifiers*) i *identyfikatory ukryte* (*escaped identifiers*).

Prosty identyfikator to dowolny ciąg liter, cyfr, znaków dolara ($) i podkreśleń (_). Nie może zaczynać się od liczby lub znaku dolara ($); pierwszym znakiem prostego identyfikatora musi być litera bądź znak podkreślenia (_). Identyfikator kończy się białym znakiem (spacją, tabulatorem albo znakiem nowej linii). Język SystemVerilog rozróżnia wielkie i małe litery w identyfikatorach, więc abc, Abc, ABc i ABC są różnymi identyfikatorami.

Przykłady poprawnych identyfikatorów:

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

Przykłady nieprawidłowych identyfikatorów:

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

W tym przypadku identyfikator XOR zapisany wielkimi literami jest poprawny, a xor zapisany małymi literami jest niepoprawny, ponieważ jest taki sam jak słowo kluczowe **xor** (wszystkie słowa kluczowe w języku SystemVerilog są pisane małymi literami).

Identyfikatory ukryte zaczynają się od odwrotnego ukośnika (\) i kończą się białym znakiem. Mają one zawierać dowolny reprezentowalny znak ASCII (w systemie dziesiętnym od 33 do 126 lub w systemie szesnastkowym od 21 do 7E). Wiodący znak odwrotnego ukośnika nie jest traktowany jako część identyfikatora. Dlatego identyfikator *abc jest postrzegany tak samo jak zwykły identyfikator *abc. Przykłady ukrytych identyfikatorów:

```
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

1.4.5. Słowa kluczowe

Słowa kluczowe (*keywords*) są zarezerwowanymi identyfikatorami języka SystemVerilog, które są używane do definiowania konstrukcji językowych. Listę wszystkich słów kluczowych języka SystemVerilog można znaleźć w dodatku.

Wstępne poznanie słów kluczowych jest dobrą praktyką podczas nauki języków programowania i projektowania. Znajomość ta jest konieczna choćby dlatego, że nie można ich używać jako identyfikatorów. Należy pamiętać, że uważna lektura spisu słów kluczowych nowego języka daje do niego pewien wgląd, a także rodzi pewne uwagi i pytania, na które odpowiedzi można znaleźć podczas dalszych studiów nad językiem.

Wszystkie słowa kluczowe SystemVerilog są pisane małymi literami (*lower case*). W tej książce są one wyróżnione pogrubioną czcionką (np. **logic**, **int**, **assign**), aby łatwiej było je zapamiętać.

Słowo kluczowe SystemVerilog poprzedzone ukośnikiem (/) nie jest interpretowane jako słowo kluczowe.

1.5. Wartości logiczne

W języku SystemVerilog *wartości logiczne* (*logical values*) pojedynczego bitu zależą od typu zmiennej. Istnieją typy danych z wartościami logicznymi używające dwóch i czterech stanów.

Wartości logiczne czterowartościowe są odziedziczone po języku Verilog, w którym każdy bit może przyjąć cztery następujące wartości:

- 0 – logiczne zero bądź „fałsz”;
- 1 – logiczna jedynka lub „prawda”;

- x – *wartość nieznana* albo *niezdefiniowana* (*don't care*), tzn. może być równa 0 lub 1;
- z – *stan wysokiej impedancji* (*high impedance state*), *trzeci stan* lub *stan pływający* (*floating*).

Gdy w danych wejściowych lub wyrażeniu pojawia się wartość z, wynik jest taki sam jak w przypadku x. W ten sposób wartość z różni się od wartości x tylko w przypadku sygnałów wyjściowych. Wartości logiczne czterowartościowe są stosowane w modelach niskiego poziomu, gdy konieczne jest odzwierciedlenie procesów fizycznych zachodzących w przewodach.

W modelach wysokiego poziomu oraz w komunikacji z językami programowania wygodniej jest używać wartości logicznych o dwóch stanach. Niektóre typy danych SystemVerilog mają wartości logiczne o dwóch stanach, tzn. każdy bit wektora może przyjmować tylko dwie wartości: 0 lub 1. Podczas symulacji dużych i złożonych projektów typy zmiennych z dwoma stanami zajmują mniej miejsca w pamięci, a proces symulacji jest szybszy.

1.6. Liczby

Język SystemVerilog pozwala na używanie w modelach zarówno liczb *całkowitych* (*integer*), jak i *rzeczywistych* (*real*).

1.6.1. Reprezentacja liczb całkowitych

Język SystemVerilog używa następującego formatu do reprezentacji liczb:

size'base value

gdzie: *size* – to rozmiar, liczba dziesiętna określająca liczbę bitów w reprezentacji liczby; ' – to znak apostrofu; *base* – to symbol *bazy* określający system liczbowy, w którym liczba jest reprezentowana; *value* – to *wartość liczby* zapisanej w systemie liczbowym określonym przez bazę. Nie są dozwolone spacje między apostrofem a znakiem podstawowym. Do reprezentowania systemu liczbowego jako podstawa (baza) są używane następujące znaki:

- b lub B – dla liczb binarnych;
- d lub D – dla liczb dziesiętnych;
- h lub H – dla liczb szesnastkowych;
- o lub O – dla liczb ósemkowych.

Należy pamiętać, że przy podawaniu dowolnej liczby w języku SystemVerilog można określić liczbę bitów (*size*), na których liczba ta zostanie zapisana.

Ogólnie rzecz biorąc, rozmiar (*size*) i symbol podstawy (*base*) w reprezentacji liczb są opcjonalne. Wartością domyślną jest zwykle liczba dziesiętna (*base*=D) zapisana w postaci 32 bitów (*size*=32). Konkretne wartości dla rozmiaru i podstawy zależą od implementacji kompilatora.

Jeśli liczba całkowita ma być przedstawiona ze znakiem, to znak bazy musi być poprzedzony znakiem *s*. Na przykład:

```
4'shf           /*      4-bitowa liczba szesnastkowa ze znakiem,
                        której binarnym odpowiednikiem jest wartość 'b1111,
                        co zostanie zinterpretowane jako uzupełnienie do dwóch liczby -1. */
```

Przypomnijmy, że liczby ujemne w języku SystemVerilog są reprezentowane w kodzie uzupełnienia do dwóch (*two's – complement number*).

Liczby ujemne mogą być poprzedzone jednoargumentowym znakiem minus. Na przykład:

```
-4'sd15         // odpowiednik -(-4'd1) = 'b0001.
```

Jeśli binarna reprezentacja wartości liczbowej przekracza określony rozmiar liczby (*size*), najbardziej znaczące wartości są w niej odrzucane. Jeśli wartość liczby zajmuje mniej cyfr niż wartość określona przez jej rozmiar, liczba jest umieszczana w najmłodszych bitach, natomiast najstarsze bity uzupełniane są zerami. Wyjątkiem są przypadki, gdy najstarszym znaczącym bitem liczby jest z lub x – w tym przypadku najstarsze bity są wypełniane odpowiednio przez z bądź x. Zamiast z i x w reprezentacji liczb mogą być używane znaki zapytania („?").

Aby poprawić czytelność, w reprezentacji dużych liczb można używać podkreślenia («_»), np.:

```
16'b0001_1011_1010_1100         // 16-bitowa liczba binarna.
```

Podczas kompilacji podkreślenie jest ignorowane, nie może też być pierwszym znakiem liczby.

Przykłady reprezentacji liczb całkowitych:

```
// liczby całkowite z oznaczeniem rozmiaru
4'b1001         // 4-bitowa liczba binarna
5 'D 3          // 5-bitowa liczba dziesiętna
3'b01x          // 3-bitowa liczba binarna
                // z nieznanym najmłodszym bitem

/* bezwymiarowe stałe całkowite (rozmiar określony domyślnie lub przez rozmiar zmiennej) */
235             //liczba dziesiętna
'h 1AFF         //liczba szesnastkowa
```

```
'o 7650              // liczba ósemkowa
1af                  // Błąd! Brakujący znak podstawy ‘h

// reprezentacja liczb całkowitych ze znakiem
-8 'd 6              // 8-bitowa reprezentacja liczby 6 w kodzie uzupełnienia do 2,
                     // odpowiednik - (8'd6)
 8'd-6               // Błąd! Znak minus jest umieszczany przed reprezentacją liczby,
                     // a nie przed wartością liczby
16'sd?               // odpowiednik 16'sdz

// użycie podkreślenia
27_185_000                      // liczba dziesiętna bez znaku 27185000
16 'b0011_1010_1110_1011        // 16-bitowa liczba binarna 'b001110110101011
32 'h 12ff_a001                 // 32-bitowa liczba szesnastkowa 'h12ffa001
```

Język SystemVerilog w odróżnieniu od Veriloga pozwala na wypełnienie wszystkich bitów zmiennych tą samą wartością przy użyciu wartości jednobitowych '0, '1, 'x lub 'z. Na przykład:

```
logic [15:0] a, b, c, d;    // deklarowanie zmiennych 16-bitowych
a = '0;                     // ustawia wszystkie 16 bitów na 0
b = '1;                     // ustawia wszystkie 16 bitów na 1
c = 'x;                     // ustawia wszystkie 16 bitów w x
d = 'z;                     // ustawia wszystkie 16 bitów w z
```

W powyższym przykładzie zastosowano typ zmiennych **logic**, który zostanie omówiony dalej.

1.6.2. Reprezentacja liczb rzeczywistych

Istnieją dwa formaty reprezentacji liczb rzeczywistych w języku SystemVerilog: reprezentacja dziesiętna i reprezentacja wykładnicza.

Dziesiętna reprezentacja liczb rzeczywistych ma następujący format:

value_i.value_f

gdzie *value_i* – to część całkowita liczby; *value_f* – to część ułamkowa liczby.

Reprezentacja wykładnicza liczb rzeczywistych ma następujący format:

*base***e***exponen*t lub
*base***E***exponent*,

gdzie *base* – to podstawa (baza) liczby; *exponent* – to wartość wykładnika.

W reprezentacji dziesiętnej liczby całkowite muszą się znajdować po obu stronach kropki dziesiętnej. W reprezentacji wykładniczej po literze „e" lub „E" nie może być spacji. Przykłady reprezentacji liczb rzeczywistych:

```
0.5          // reprezentacja dziesiętna
3e4          // reprezentacja wykładnicza, równoważna 3·104 = 30000
5.8E-3       // reprezentacja wykładnicza, równoważna 5,8·10-3=0,0058.
```

1.7. Atrybuty

Czasami konieczne jest określenie właściwości obiektu, które są następnie wykorzystywane przez różne narzędzia syntezy, takie jak syntezator czy symulator. Do przekazywania tych informacji służą *atrybuty* (*attributes*). Innymi słowy – atrybuty języka SystemVerilog określają specyficzne właściwości operatorów i konstrukcji zaimplementowanych w konkretnym oprogramowaniu, takim jak kompilator systemu Quartus czy Vivado. Format atrybutów języka SystemVerilog:

(* *attribute, attribute*,... ***)**

gdzie (*– początek listy atrybutów; *) – koniec listy atrybutów; *attribute* – atrybut do zdefiniowania. Nawiasy atrybutów, tj. pary znaków (* i *), są obowiązkowe, nawet jeśli określony jest tylko jeden atrybut.

Atrybuty SystemVerilog mogą być prefiksami do deklaracji, instancji modułów, operatorów, portów itp. Można je również przypisywać jako przyrostki do operacji lub wywołań funkcji. Atrybuty mogą mieć wartości. Jeśli ich nie określono, domyślną wartością jest 1. Możliwe jest także jednoczesne zdefiniowanie kilku atrybutów. W tym przypadku są one oddzielone przecinkami na liście atrybutów.

Przykłady użycia atrybutów:

```
(* INIT="1" *) reg Q;                  // użycie atrybutu w deklaracji zmiennej Q

sum = a + (* ripple_adder *) b;        // użycie atrybutu w wyrażeniu
                                       // jako przyrostek operacji dodawania
```

Atrybuty jako możliwe jednostki konstrukcyjne po raz pierwszy pojawiły się w języku Verilog-2001. Standard ten nie definiuje jednak konkretnych atrybutów. Specyficzne atrybuty języka SystemVerilog są wprowadzane przez twórców narzędzi projektowych (kompilatorów, syntezatorów, symulatorów itp.).

Chociaż konkretne atrybuty języka SystemVerilog są definiowane przez twórców narzędzi projektowych, jego wszystkie nowoczesne kompilatory obsługują dwa atrybuty: *full_case* i *parallel_case*.

1.8. Równoległość języka SystemVerilog

W przeciwieństwie do języków programowania język SystemVerilog pozwala na równoległe wykonywanie swoich konstrukcji podczas symulacji. Do takich konstrukcji należą:

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

Równoległe wykonywanie poszczególnych konstrukcji języka Verilog może być również uwzględniane przez syntezator.

Równoległość języków projektowania w pełni odpowiada fizycznej naturze obwodów elektronicznych: gdy przyłożone jest napięcie, wszystkie elementy obwodu działają jednocześnie i równolegle, a sygnały elektryczne są przesyłane przez obwody jednocześnie i równolegle. W SystemVerilog instancje prymitywów i modułów odpowiadają elementom obwodu, operatory przypisania (**assign**) połączeniom przewodów, a bloki procedur blokom funkcyjnym urządzeń cyfrowych. Ponadto równoległość SystemVerilog jest bardzo przydatna do symulacji urządzeń cyfrowych.

1.9. Synteza i symulacja projektu

Jak wiadomo, projektowanie składa się z dwóch nierozłącznych części: syntezy i symulacji. Synteza jest czasem nazywana projektowaniem, ale nim nie jest, jest natomiast jego częścią. Język SystemVerilog został zaprojektowany właśnie do syntezy i symulacji złożonych urządzeń i systemów cyfrowych. Obsługuje wszystkie poziomy projektowania, poczynając od najwyższych: systemowego i abstrakcyjnego.

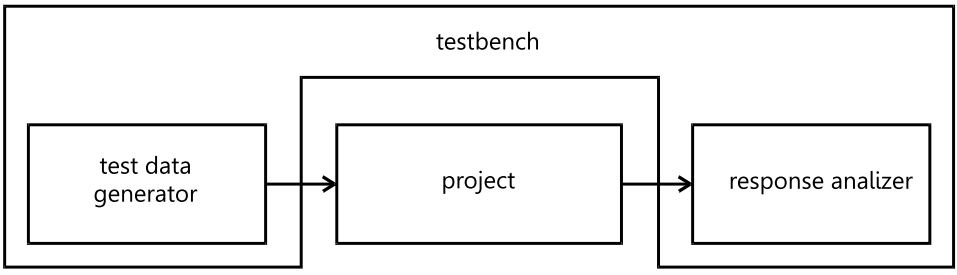
Synteza z wykorzystaniem języka SystemVerilog polega na przekształceniu wstępnego opisu projektu w postać odpowiednią do jego sprzętowej realizacji. Na przykład podczas implementacji projektu na FPGA, w czasie syntezy, źródłowy opis projektu SystemVerilog jest przekształcany w plik z danymi konfiguracyjnymi FPGA.

Symulacja jest znacznie bardziej złożona, ponieważ w tym procesie rozwiązywanych jest wiele różnych zadań. Istnieje kilka rodzajów symulacji:

- *funkcjonalna*, podczas której sprawdza się, czy projekt wykonuje daną funkcję bez względu na inne parametry, np. czy procesor wykonuje określoną listę instrukcji;
- *czasowa*, podczas której sprawdza się, czy projekt spełnia określone wymagania lub parametry czasowe, np. czy maksymalna częstotliwość taktowania procesora nie jest niższa od podanej;
- *zużycia energii*, podczas której sprawdza się, czy projekt spełnia limity zużycia energii;
- *fizyczna*, podczas której sprawdza się np. rozkład ciepła na powierzchni układu scalonego.

Dodatkowo w czasie symulacji poruszane są kwestie kompletności weryfikacji projektu, np. czy wszystkie węzły i bloki projektu są zweryfikowane we wszystkich trybach pracy.

Na rys. 1.1 przedstawiono uproszczony schemat interakcji między *syntetyzowanym projektem* (*project*) a blokiem wykonującym symulację, zwanym *testbenchem*, *stanowiskiem testowym* lub *środowiskiem testowym*.



RYS. 1.1. Struktura interakcji między stanowiskiem testowym a projektem podczas symulacji

ŹRÓDŁO: oprac. własne.

Generator danych testowych (*test data generator*) służy do generowania wejściowych sekwencji testowych, które są podawane na wejścia projektu. Uzyskane wyniki są analizowane przez *analizator odpowiedzi* (*response analyzer*), który odpowiada na pytania zdefiniowane przez programistę i wyświetla komunikaty oraz dane na ekranie lub w pliku. Należy pamiętać, że w języku SystemVerilog można opisać zarówno projekt, który ma być syntetyzowany, jak i środowisko symulacyjne bądź stanowisko testowe.

Zazwyczaj projekt wykonuje się w jakimś zintegrowanym środowisku projektowym. W przykładach przedstawionych w niniejszej książce wykorzystano program Quartus Prime firmy Intel w wersjach 18.1 i 21.1 oraz płytę ewaluacyjną DE1-SoC.

Uproszczony proces projektowania systemu w strukturze FPGA z wykorzystaniem języka SystemVerilog jest następujący.

Najpierw tworzony jest syntezowalny model projektu (*project*) w języku SystemVerilog. Na tym etapie wykorzystywane są zsyntetyzowane konstrukcje

SystemVerilog. Projekt jest kompilowany (np. w programie Quartus) i, opcjonalnie, wcześniej symulowany za pomocą prostych narzędzi (np. University Program w programie Quartus).

Aby spełnić wymagania czasowe, analizę czasową projektu wykonuje się za pomocą oddzielnego oprogramowania narzędziowego (np. TimeQuest [7]), zwanego *analizatorem czasowym* (*time analyzer*).

Następnie tworzony jest *testbench projektu*, również w języku SystemVerilog. Na tym etapie mogą być używane wszystkie dostępne konstrukcje tego języka. Symulacja projektu sterowana zdarzeniami jest wykonywana za pomocą narzędzia programowego zwanego *symulatorem* (*simulator*). Zazwyczaj jest on oddzielnym narzędziem programowym (np. ModelSim czy Questa, które są kompatybilne z systemem Quartus).

Trzeba pamiętać, że model projektu do syntezy i stanowisko testowe do symulacji w języku SystemVerilog mogą być opracowywane równolegle.

Jeśli wyniki symulacji są zadowalające, przeprowadza się dodatkowe testowanie projektu (symulacja zużycia energii, symulacja fizyczna itp.) za pomocą systemu do projektowania (np. Quartus).

W przypadku wykrycia błędów na którymkolwiek z etapów wprowadza się niezbędne modyfikacje projektu zarówno na potrzeby syntezy, jak i symulacji.

Po pomyślnym zakończeniu wszystkich testów i spełnieniu wszystkich wymagań projektowych docelowy układ FPGA jest konfigurowany na płytce ewaluacyjnej (na potrzeby książki użyto DE1-SoC) i testowany jest projekt fizyczny.

Należy zauważyć, że w niektórych przypadkach projekt przedstawiony na rys. 1.1 może być przeznaczony tylko do symulacji. Na przykład do celów badawczych, gdy konieczne jest wykorzystanie modelu do testowania parametrów systemu lub do weryfikacji koncepcji przyszłego projektu.

Zauważmy jeszcze raz, że język SystemVerilog może być stosowany i do syntezy, i symulacji. Dlatego można w nim opisywać nie tylko projekty przeznaczone do implementacji sprzętowej (czyli do syntezy), lecz także takie, za pomocą których można weryfikować (sprawdzać i testować) wcześniej opisane projekty do syntezy (*testbenches*). W języku SystemVerilog można również opisywać duże systemy programowo-sprzętowe, które komunikują się z procesorami za pomocą interfejsów, a z urządzeniami komunikacyjnymi przy użyciu różnych protokołów komunikacyjnych.

W niniejszej pracy omówiono głównie konstrukcje języka SystemVerilog, które są przeznaczone do opisu projektów do syntezy.

Prostym przykładem syntezy i symulacji w SystemVerilog jest projekt jednobitowego sumatora.

1.9.1. Model projektu do syntezy

Opiszmy *model do syntezy*, tj. *projekt realizowalny*, jednobitowego sumatora w następujący sposób.

Przykład 1.1. Opis sumatora jednobitowego z wykorzystaniem operacji arytmetycznych

```
module add_1 (input a, b, cin,              // wejścia
              output s, cout);              // wyjścia

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

endmodule
```

W powyższym przykładzie jednobitowy sumator został opisany za pomocą modułu o nazwie add_1. Wszystkie porty modułu add_1 mają domyślnie typ danych **logic** i szerokość 1 bita. Ciało modułu jest reprezentowane przez pojedynczy operator przypisania ciągłego **assign**. Operacja konkatenacji ({}) jest tu stosowana po lewej stronie znaku równości, a operacja dodawania arytmetycznego – po prawej. Operacja konkatenacji jest konieczna, ponieważ wynik dodawania może generować sygnał przeniesienia do następnego bitu, tzn. wynikiem dodawania może być dwubitowa (łącznie) wartość sumy s i przeniesienia cout.

Nie będziemy wchodzić w szczegóły modelu projektu do syntezy, ponieważ zagadnienie to stanowi treść książki. Skupimy się bardziej na symulacji projektu.

1.9.2. Symulacja projektu

Po opisaniu projektu należy go sprawdzić (zweryfikować), czyli przeprowadzić symulację. W przeszłości, przed pojawieniem się języka Verilog, do symulacji używano edytorów graficznych, specjalnych języków innych niż języki opisu sprzętu itp. Ponieważ jednak Verilog został pierwotnie zaprojektowany do symulacji, język SystemVerilog może być również używany do tego celu.

Moduł do symulacji (*testbench* – stanowisko testowe) jednobitowego sumatora jest pokazany w przykładzie 1.2.

Przykład 1.2. Stanowisko testowe do symulacji sumatora jednobitowego

```
`timescale 1ns/1ps                      // definicja jednostki czasu symulacji
module test_add_1();                    // początek modułu do symulacji
      reg tcin, ta, tb;                 // deklaracja zmiennych dla wektorów wejściowych
      wire ts, tcout;                   // deklaracja zmiennych dla wyjść sumatora

      add_1 sum(tcin,ta,tb,ts,tcout);   // tworzenie instancji modułu add_1

      initial begin: test               // blok proceduralny initial o nazwie test
```

```
        integer i;                    // deklaracja zmiennej lokalnej
        $display(„Wyniki symulacji sumatora jednobitowego:");
        $timeformat(-9,1,"ns",8);     // definicja formatu czasu
        $monitor($time,": cin=%b a=%b b=%b s=%b cout=%b", tcin,ta,tb,ts,tcout);
        for (i=0; i<8; i=i+1) begin   // operator początku pętli
            #10;                      // operator opóźnienia o 10 jednostek czasu
            {tcin,ta,tb} = i; // tworzenie wektora wejściowego
        end                           // koniec operatora pętli
    end                               // koniec operatora proceduralnego test
endmodule                             // koniec modułu
```

Zwróćmy uwagę na pewne osobliwości powyższego opisu. Pierwszy wiersz kodu wykorzystuje dyrektywę systemową **`timescale** do zdefiniowania wartości jednostki czasu równej 1 ns (jedna nanosekunda) i precyzji czasowej równej 1 ps (jedna pikosekunda). Ponieważ moduł symulacyjny o nazwie *test_add_1* nie odbiera żadnych sygnałów z zewnątrz ani nie przekazuje sygnałów do środowiska zewnętrznego, nie ma żadnych portów. Następnie deklarowane są zmienne umożliwiające wygenerowanie wektorów wejściowych *tcin*, *ta*, *tb* oraz uzyskanie wyników sumowania *ts*, *tcout*. Kolejny wiersz kodu tworzy model projektu w postaci instancji modułu *add_1* o nazwie *sum*.

Symulacja modelu odbywa się za pomocą bloku proceduralnego **initial** o nazwie *test*, który jest wykonywany tylko raz. Deklarowana jest tu zmienna lokalna *i* typu integer, a następnie wywoływane są zadania systemowe: **$display** – wyświetlanie łańcucha znaków; **$timeformat** – określanie formatu reprezentacji czasu oraz **$monitor** – monitorowanie określonych zmiennych i wyświetlanie ich wartości w przypadku zmiany wartości przynajmniej jednej ze zmiennych (z wyjątkiem wartości czasu). Po nim następuje operator proceduralny **for**, w którego ciele wykonywane jest opóźnienie o 10 jednostek czasu i generowana jest wartość wektorów wejściowych.

Ponieważ w języku SystemVerilog bloki proceduralne i instancje modułów są wykonywane równolegle, wartości wektorów wejściowych wygenerowanych w bloku **initial** (zmienne *tcin*, *ta* i *tb*) będą automatycznie przekazywane na wejście instancji *sum*. Czas jej rozpoczęcia nie jest określony, więc przyjmuje się, że wynosi on 0,0. Dzięki temu wartości zmiennych wyjściowych *ts* i *tcont* zostaną wygenerowane bez żadnego opóźnienia, a zadanie systemowe **$monitor** wypisze na ekranie ich wartości oraz wartości zmiennych wejściowych.

Zadanie. Porównaj opis projektu sumatora jednobitowego z przykładu 1.1 z modułem (środowiskiem testowym) do jego symulacji z przykładu 1.2. Jak myślisz, co zajmuje projektantowi więcej czasu – opisanie projektu czy jego symulacja?

Wyniki symulacji jednobitowego sumatora wykonanej w programie ModelSim pokazano na rys. 1.2.

Analizując wyniki symulacji, deweloper może wyciągnąć wnioski na temat poprawności działania projektu i w razie potrzeby wprowadzić odpowiednie zmiany w jego opisie.

```
# Wyniki symulacji sumatora jednobitowego:
#  0:     cin=x   a=x   b=x   s=x   cout=x
# 10:     cin=0   a=0   b=0   s=0   cout=0
# 20:     cin=0   a=0   b=1   s=1   cout=0
# 30:     cin=0   a=1   b=0   s=1   cout=0
# 40:     cin=0   a=1   b=1   s=0   cout=1
# 50:     cin=1   a=0   b=0   s=1   cout=0
# 60:     cin=1   a=0   b=1   s=0   cout=1
# 70:     cin=1   a=1   b=0   s=0   cout=1
# 80:     cin=1   a=1   b=1   s=1   cout=1
```

RYS. 1.2. Wyniki symulacji addera jednobitowego

ŹRÓDŁO: oprac. własne.

Należy zauważyć, że analiza wyników symulacji może być również przeprowadzona przy użyciu języka SystemVerilog, a wówczas tylko wyniki tej analizy będą wyświetlane na ekranie.

1.10. Elementy konstrukcyjne języka SystemVerilog

Projekty do syntezy i symulacji w języku SystemVerilog są opisywane w postaci kodu, który składa się z *elementów projektowych* (*design element*). Można je porównać z blokami konstrukcyjnymi, z których tworzy się projekt. Większość elementów konstrukcyjnych można wykorzystać zarówno do syntezy, jak i symulacji. Niektóre elementy projektu są jednak wykorzystywane tylko do symulacji.

Blokiem konstrukcyjnym lub elementem konstrukcyjnym języka SystemVerilog może być:

- moduł (**module**);
- prymityw (**primitive**);
- pakiet (**package**);
- interfejs (**interface**);
- program (**program**);
- kontroler (**checker**);
- konfiguracja (**config**).

W nawiasach podano słowa kluczowe użyte do zidentyfikowania odpowiedniego bloku konstrukcyjnego.

Programy i kontrolery są wykorzystywane tylko w projektach symulacyjnych; pozostałe elementy mogą występować zarówno w projektach syntezowanych, jak i symulacyjnych.

Ogólnie rzecz biorąc, w kodzie projektu mogą występować dowolne elementy języka SystemVerilog. Podczas syntezy kompilator po prostu ignoruje te konstrukcje językowe, które są przeznaczone jedynie do symulacji.

1.10.1. Moduły

Podstawowym elementem konstrukcyjnym języka SystemVerilog jest *moduł* (**module**), który jest opisany za pomocą słów kluczowych **module** i **endmodule**. Moduły są używane głównie do opisywania syntetyzowanych projektów, ale mogą być także używane do reprezentowania kodu symulacyjnego. Mogą też służyć do łączenia jednostek weryfikacji i syntezy.

Moduł może zawierać:

- porty z deklaracjami portów;
- deklaracje parametrów z wartościami domyślnymi;
- deklaracje danych, takich jak sieci, zmienne, struktury i unie;
- deklaracje stałych;
- definicje typów zdefiniowanych przez użytkownika;
- definicje klas;
- import deklaracji z pakietów;
- definicje podprogramów (zadań i funkcji);
- instancje innych modułów, programów, interfejsów, kontrolerów i prymitywów;
- instancje obiektów klas;
- przypisania ciągłe (**assign**);
- bloki proceduralne (**always**, **initial**);
- generowanie bloków (*generate blocks*);
- bloki specyfikacji (*specify blocks*).

Należy zauważyć, że lista ta nie wyczerpuje wszystkich elementów języka SystemVerilog, które może zawierać moduł.

Poniżej znajduje się kod prostego modułu opisującego multiplekser 2-1 N-bitowej szyny.

Przykład 1.3. Multiplekser 2-1 N-bitowej szyny

```
module mux_2_1_N_bits                  // nazwa modułu
      #(parameter N=8)                 // definicja parametru N
                                       // z wartością domyślną 8

      (input [N-1:0] a, b,             // deklaracja portu modułu
      input c,
      output logic [N-1:0] y);         // każdy bit y ma 4 wartości logiczne
```

```
    always_comb                    // blok proceduralny typu kombinacyjnego
    begin                          // początek bloku proceduralnego
        if (c)   y = a;            // operator proceduralny if
        else     y = b;
    end                            // koniec bloku proceduralnego
endmodule: mux_2_1_N_bits          // koniec modułu mux_2_1_N_bits
```

Bardziej szczegółowo moduły omówiono w rozdziale 2.

1.10.2. Pakiety

Elementy konstrukcyjne, takie jak moduły, interfejsy, programy i kontrolery, definiują lokalne przestrzenie widoczności nazw (zakresy lokalne). Identyfikatory zadeklarowane w zakresie lokalnym nie kolidują z identyfikatorami w innych zakresach, a ich nazwy mogą być takie same. Często jednak konieczne jest ponowne użycie tych samych deklaracji (np. typów użytkownika) w innych elementach projektu. Aby uniknąć powielania deklaracji, w języku SystemVerilog stosuje się pakiety.

Pakiet (*package*) tworzy zakres, który zawiera deklaracje przeznaczone do współużytkowania przez wiele konstrukcji języka lub jednostek kompilacji. Nazwa pakietu jest globalna, tzn. jest dostępna z każdego miejsca w kodzie projektu. Oprócz deklaracji struktur danych i typów zdefiniowanych przez użytkownika pakiety mogą zawierać również deklaracje funkcji.

Przykład 1.4. Deklarowanie liczb zespolonych za pomocą pakietu

```
package ComplexPkg;                    // deklaracja pakietu ComplexPkg
    typedef struct {                   // deklaracja typu użytkownika Complex
        int i, r;                      // części urojona i rzeczywista liczby
    } Complex;

    // deklaracja funkcji add wewnątrz pakietu ComplexPkg
    // do dodawania liczb zespolonych
    function Complex add(Complex a, b);      // a i b mają typ Complex
                                             // funkcja zwraca również typ Complex
        add.r = a.r + b.r;
        add.i = a.i + b.i;
    endfunction

    // deklaracja funkcji mul w pakiecie ComplexPkg
    // do mnożenia liczb zespolonych
    function Complex mul(Complex a, b);
        mul.r = (a.r * b.r) - (a.i * b.i);
```

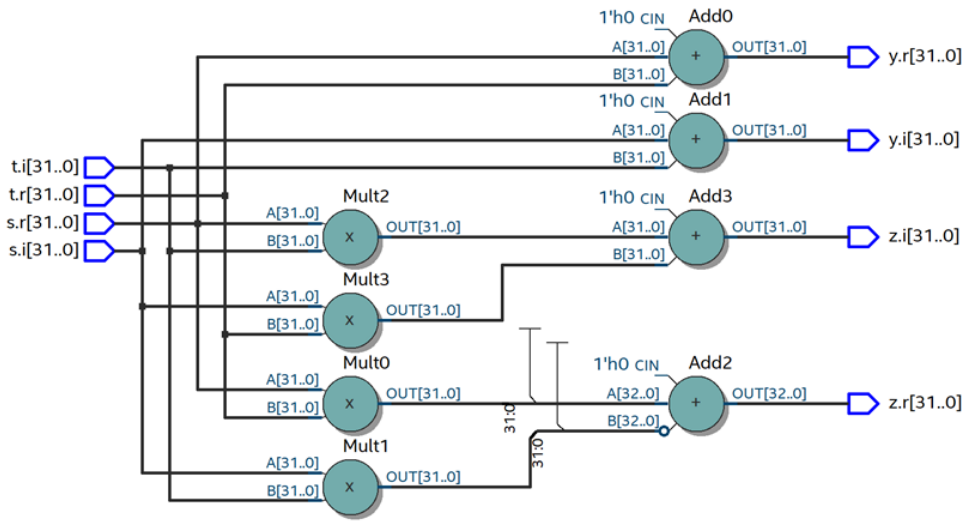
```
        mul.i = (a.r * b.i) + (a.i * b.r);
    endfunction
endpackage : ComplexPkg                    // koniec deklaracji pakietu ComplexPkg
```

Powyższy kod deklaruje pakiet ComplexPkg, który zawiera deklarację typu użytkownika Complex do reprezentowania liczb zespolonych oraz dwie funkcje add i mul do dodawania i mnożenia liczb zespolonych. Aby uzyskać bezpośredni dostęp do deklaracji pakietu w SystemVerilog, należy użyć *operatora rozpoznawania kontekstu* (::).

Przykład 1.5. Użycie pakietu ComplexPkg

```
module using_ComplexPkg(
    input ComplexPkg::Complex s,t,     // deklaracja wejść i wyjść
    output ComplexPkg::Complex y,z);   // jako typ Complex pakietu ComplexPkg

    assign y = ComplexPkg::add(s,t);   // wywołanie funkcji add z pakietu ComplexPkg

    assign z = ComplexPkg::mul(s,t);   // wywołanie funkcji mul z pakietu ComplexPkg

endmodule
```

Wynik syntezy programu using_ComplexPkg jest przedstawiony na rys. 1.3.



RYS. 1.3. **Wynik syntezy projektu przy użyciu ComplexPkg**

ŹRÓDŁO: **oprac. własne.**

Pakiety zostały omówione bardziej szczegółowo w rozdziale 8.

1.10.3. Interfejsy

W dużych i złożonych systemach, takich jak mikroprocesory, znaczną część kodu zajmuje opis portów modułu, których nazwy powtarzają się w każdym module. W najprostszym przypadku *interfejs* (*interface*) jest nazwaną grupą deklaracji sieci i zmiennych, zawartą między słowami kluczowymi **interface** i **endinterface**. Możliwe jest tworzenie instancji interfejsu i przekazywanie jej między modułami przez port interfejsu jako jednego elementu. Każdy moduł będzie miał wtedy dostęp do wszystkich łańcuchów i zmiennych zadeklarowanych w interfejsie.

Różne moduły mogą mieć różne kierunki dla sygnałów o tej samej nazwie, np. w module nadrzędnym „master" sygnał resetowania reset jest wyjściem, a w module podrzędnym „slave" – wejściem. Konstrukcja *modport* służy do wskazywania kierunku sygnałów zadeklarowanych w interfejsie. Dodatkowo konstrukcje modportów umożliwiają interfejsom uczynienie poszczególnych sygnałów widocznymi dla jednych modułów, a niewidocznymi dla innych.

Interfejsy umożliwiają również korzystanie z funkcji. W tym celu interfejsy mogą zawierać parametry, stałe, zmienne, instrukcje proceduralne (**initial** i **always**) oraz instrukcje przypisania ciągłego (**assign**), a także mogą mieć własne porty. Typy danych mogą być przekazywane do interfejsów jako parametry. Dzięki temu interfejs może zawierać np. własną weryfikację protokołu, która automatycznie sprawdza, czy wszystkie moduły podłączone do interfejsu są zgodne z określonym protokołem. W interfejs można wbudować także różne aplikacje, takie jak raportowanie testów pokrycia funkcjonalnego.

Przykład 1.6. Deklaracja i użycie interfejsu

```
// definicja interfejsu
interface simple_bus(input logic clk);      // port wejściowy clk interfejsu simple_bus
        logic req, gnt;                     // zmienne interfejsu simple_bus
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;
endinterface: simple_bus

// moduł, który używa portu interfejsu
module memMod(simple_bus a);                // a jest portem interfejsu typu simple_bus
        logic avail;
        always @(posedge a.clk)
                a.gnt <= a.req & avail;     // użyj sygnałów gnt i req interfejsu simple_bus
endmodule

// inny moduł korzystający z portu interfejsu
module cpuMod(simple_bus b);                // b – port interfejsu typu simple_bus
```

```
endmodule

module top;                              // moduł najwyższego poziomu

    logic clk = 0;

    simple_bus sb_intf(.clk(clk));       // tworzenie instancji sb_intf
                                         // interfejsu simple_bus
    memMod mem(.a(sb_intf));             // podłączenie instancji interfejsu sb_intf
                                         // do instancji modułu memMod
    cpuMod cpu(.b(sb_intf));             // podłączenie instancji interfejsu sb_intf
                                         // do instancji modułu cpuMod
endmodule
```

Interfejsy zostały omówione bardziej szczegółowo w rozdziale 14.

1.10.4. Prymitywy

Prymitywy (*primitives*) w języku projektowania to podstawowe elementy układów elektronicznych, które mogą być używane bez wcześniejszej deklaracji. Język SystemVerilog zawiera 14 prymitywów bramkowych i 12 prymitywów przełączających. Ta pierwsza grupa zawiera bramki logiczne **and**, **nand**, **or**, **nor**, **xor** i **xnor**, bufory **buf** i **not**, bufory o trzech stanach **bufif1**, **bufif0**, **notif1** i **notif0** oraz bramki podciągające **pulldown** i **pullup**. Grupa prymitywów przełączających reprezentuje różne typy tranzystorów.

Deweloperzy mogą uzupełniać prymitywy predefiniowane w języku o *prymitywy użytkownika*. W tym celu SystemVerilog udostępnia mechanizm zwany UDP (*user defined primitive* – prymityw zdefiniowany przez użytkownika). Za jego pomocą można zdefiniować prymitywy kombinacyjne lub sekwencyjne. Maksymalna liczba wejść dla prymitywów kombinacyjnych UDP wynosi 10, a dla prymitywów sekwencyjnych UDP wynosi 9. Należy pamiętać, że te ostatnie mogą być wrażliwe zarówno na zbocze, jak i na poziom sygnału sterującego. Opis niestandardowych prymitywów znajduje się między słowami kluczowymi **primitive** i **endprimitive**.

Przykład 1.7. Prymityw multipleksera dwuwejściowego

```
primitive mux (y, a, b, sel);   // deklaracja portów, stary styl
    output y;                   // wyjście prymitywu
    input sel, a, b;            // wejście prymitywu

    table                       // tabela prawdy prymitywu
      // a b sel : y            – opis działania prymitywu
         0 ?  0 :  0;           // a jest wybrane, wartość b nie jest określona
```

```
        1 ? 0 : 1;      // a jest wybrane, wartość b nie jest określona
        ? 0 1 : 0;      // b jest wybrane, wartość a nie jest określona
        ? 1 1 : 1;      // b jest wybrane, wartość a nie jest określona
        0 0 x: 0;       // x na wejściu sterującym
        1 1 x: 1;
    endtable
endprimitive
```

Prymitywy omówiono bardziej szczegółowo w rozdziale 17.

1.10.5. Programy

Moduły dobrze nadają się do opisywania projektów na poziomie sprzętowym, skupiając się na takich szczegółach, jak przewody, hierarchia strukturalna, połączenia i inne rozwiązania układów. Jednak pomimo możliwości opisywania algorytmów za pomocą operatorów proceduralnych moduły nie nadają się do opisywania stanowiska testowego (*testbench*). W tym celu w języku SystemVerilog używa się *programów* (*programs*).

Opis programu znajduje się między słowami kluczowymi **program** i **endprogram**. Programy mają za zadanie symulować środowisko testowe. Służą także jako separator między projektem, który ma być syntetyzowany, a stanowiskiem testowym, ponadto zaś program określa specjalistyczną semantykę przebiegu symulacji.

Blok programu może zawierać deklaracje danych, definicje klas, definicje podprogramów (zadań i funkcji), instancje obiektów oraz jedną bądź więcej procedur **initial** lub **final**. Nie może zawierać procedur **always**, instancji prymitywów, modułów, interfejsów ani instancji innych programów.

Programy zapewniają wolną od wyścigów interakcję między syntetyzowanym projektem a środowiskiem testowym, a także możliwość abstrakcji na poziomie pętli i transakcji. Umożliwiają również wykorzystanie ich jako *modeli uogólnionych* (*generalized models*).

Przykład deklaracji programu:

```
program test (               // deklaracja programu o nazwie test
    input clk,               // program może mieć porty
    input [16:1] addr,
    inout [7:0] data);

    initial begin            // blok inicjalizacji

    end

endprogram
```

Należy pamiętać, że programy w SystemVerilog obsługują programowanie obiektowe. Dlatego można w nich deklarować klasy i tworzyć instancje obiektów. Ponieważ jednak w książce tej skoncentrowano się na podzbiorze języka SystemVerilog przeznaczonym do syntezy, programy i klasy nie są w niej omawiane.

1.10.6. Kontrolery

Konstrukcja *kontrolera* (**checker**) znajdująca się między słowami kluczowymi **checker** i **endchecker** jest blokiem weryfikacji, który łączy *zapewnienia* (*assertions*) z symulowanym kodem. Kontrolery są przeznaczone do stosowania jako jednostki biblioteki weryfikacyjnej lub jako bloki konstrukcyjne do tworzenia abstrakcyjnych modeli pomocniczych używanych w weryfikacji formalnej.

Zarówno kontrolery, jak i programy nie są omawiane w tej książce.

1.10.7. Konfiguracje

Język SystemVerilog daje możliwość definiowania konfiguracji projektu, które określają informacje o powiązaniu instancji modułów z konkretnym kodem źródłowym SystemVerilog. Konfiguracje wykorzystują biblioteki, czyli zbiór modułów, interfejsów, programów, kontrolerów, prymitywów, pakietów i innych konfiguracji. Specjalny plik zwany *plikiem mapy biblioteki* (*library map file*) służy do mapowania biblioteki na fizyczną lokalizację pliku.

Nazwa „plik mapy biblioteki" jest zwykle podawana jako parametr podczas wywoływania symulatora lub innych narzędzi programowych, które są określone w kodzie źródłowym SystemVerilog.

Konfiguracje są omówione bardziej szczegółowo w rozdziale 18.

1.11. Jednostki kompilacji

Kompilator Verilog zawsze kompiluje wszystkie moduły projektu. Dlatego podczas wprowadzania niewielkich zmian w dużym projekcie konieczne jest skompilowanie całego projektu, co może trwać dość długo.

Koncepcja *jednostki kompilacji* (*compilation unit*) została wprowadzona do języka SystemVerilog, ponieważ jest on ukierunkowany na tworzenie dużych i złożonych projektów. Jednostka kompilacji to wszystkie pliki źródłowe, które są kompilowane w tym samym czasie. Deklaracje mogą być w niej tworzone poza zakresem pakietu, modułu, interfejsu lub programu. Znajdują się wówczas w *zakresie jednostki kompilacji* (*compilation unit scope*) i są widoczne dla wszystkich modułów, które są kompilowane w tym samym czasie.

Kontekst jednostki kompilacji może zawierać deklaracje:

- jednostek czasu i precyzji;
- sieci i zmiennych;
- stałych;
- typów i klas zdefiniowanych przez użytkownika;
- zadań i funkcji.

Deklaracje zewnętrzne w jednostce kompilacji nie są globalne i nie można ich syntetyzować. Służą one jedynie do poprawnej kompilacji modułów tworzących jednostkę kompilacji. Dlatego w przypadku projektów syntetyzowalnych wszystkie deklaracje zewnętrzne powinny być tworzone w pakietach, które są w pełni syntetyzowalne.

Do odwoływania się do modułów, które nie znajdują się w kontekście jednostki kompilacji, służą *prototypy modułów* (*module prototypes*) lub *moduły zewnętrzne* (*external modules*).

Język SystemVerilog używa specjalnej nazwy **$unit** do oznaczenia kontekstu jednostki kompilacji. Nazwa **$unit** jest przeznaczona do odwoływania się do zewnętrznych deklaracji jednostki kompilacji. Można tu wykorzystać operator rozpoznawania kontekstu (::), który jest również używany przy odwoływaniu się do zawartości pakietów.

Na przykład:

```
bit b;                    // zewnętrzna deklaracja jednostki kompilacji
task t;                   // deklaracja zadania
      int b;              // deklaracja zmiennej lokalnej
      b = 5 + $unit::b;   // $unit::b jest zmienną zewnętrzną b
endtask
```

W ten sposób jednostki kompilacji pozwalają kompilować duże projekty po kawałku. Aby jednak utworzyć projekt syntetyzowany, należy jeszcze raz skompilować wszystkie moduły projektu.

Jednostki kompilacji zostały omówione bardziej szczegółowo w rozdziale 8.

1.12. Ulepszenia wprowadzone przez język SystemVerilog do syntezy

W tej książce omówiono wersję języka SystemVerilog przeznaczoną do syntezy. Dodano do niego następujące kluczowe ulepszenia, które w porównaniu z językiem Verilog umożliwiają tworzenie syntezowalnych projektów:

- interfejsy;
- typy danych języka C, takie jak **int**;
- typy użytkownika;

- typy wyliczeniowe;
- konwersja typów;
- struktury i unie;
- pakiety;
- jednostki kompilacji;
- operacje języka C++, takie jak ++, -- itd.;
- operacje przypisania języka C++, takie jak -=, += itd.;
- kwalifikatory priorytetu (**priority**) i unikalności (**unique**) operatorów **case** i **if**;
- ulepszenie operatorów proceduralnych;
- przekazywanie argumentów przez referencję do modułów, zadań i funkcji.

Wszystkie te udoskonalenia języka Verilog zostaną omówione w tej książce.

1.13. Wnioski

Język SystemVerilog jest naturalnym rozwinięciem języka opisu sprzętu Verilog i jest przeznaczony do projektowania dużych i złożonych projektów. Powstanie SystemVerilog wynikło z potrzeby rozwiązania problemów związanych z tworzeniem nowoczesnych systemów cyfrowych. SystemVerilog podsumowuje doświadczenia w zakresie języków opisu sprzętu kilku wiodących firm w dziedzinie automatyzacji projektowania elektronicznego.

Język SystemVerilog obejmuje szereg opracowań takich firm, jak Co-Design Automation, Synopsys, IBM, Mentor Graphics i Blue Spec. Ponadto do SystemVerilog zostały dodane niektóre operatory i operacje języka C.

Język SystemVerilog jest kompatybilny ze swoim poprzednikiem, językiem Verilog, a więc ich podstawowe elementy są takie same.

Język SystemVerilog jest przeznaczony zarówno do syntezy, czyli tworzenia opisów projektów przeznaczonych do implementacji sprzętowej, jak i do symulacji, czyli tworzenia modułu testowego zwanego *testbench* lub *środowiskiem testowym*.

W książce omówiono głównie konstrukcje języka SystemVerilog przeznaczone do syntezy.

Podstawowymi elementami konstrukcyjnymi języka SystemVerilog są: moduł (**module**); pakiet (**package**); interfejs (**interface**); prymityw (**primitive**); program (**program**); kontroler (**checker**); konfiguracja (**config**).

Programy i kontrolery są używane tylko w stanowiskach testowych, inne elementy można znaleźć zarówno w projektach syntezowanych, jak i tych do symulacji.

Podstawowym elementem składowym języka SystemVerilog jest moduł. Są one używane głównie do opisywania syntetyzowanych projektów, ale mogą też reprezentować kod stanowiska testowego. Ponadto moduły można wykorzystać do łączenia bloków weryfikacji i syntezy.

Pakiet (**package**) tworzy zakres, który zawiera deklaracje przeznaczone do współużytkowania przez wiele elementów projektu lub jednostek kompilacji.

Interfejsy umożliwiają grupowanie wielu sygnałów i przekazywanie ich przez porty jako pojedynczego elementu. Ponadto interfejsy definiują różne kierunki sygnałów dla różnych jednostek i mogą zawierać funkcjonalności w postaci zadań, funkcji i instrukcji proceduralnych służących do wstępnego przetwarzania danych.

Programy są przeznaczone do symulowania środowiska testowego i mogą definiować specjalistyczną semantykę prowadzenia symulacji.

Kontrolery wykorzystuje się jako jednostki biblioteki weryfikacyjnej lub jako bloki konstrukcyjne do tworzenia abstrakcyjnych modeli pomocniczych używanych w weryfikacji formalnej.

Konfiguracje umożliwiają określenie informacji o powiązaniu instancji modułów z określonym kodem źródłowym języka SystemVerilog.

Jednostki kompilacji umożliwiają kompilację dużych projektów w częściach. Są to wszystkie pliki źródłowe, które są kompilowane w tym samym czasie. Jednostka kompilacji tworzy swój własny zakres (kontekst), w którym widoczne są wszystkie moduły kompilowane w tym samym czasie.

2. Moduły

Moduł (*module*) jest podstawowym elementem konstrukcyjnym projektów w języku SystemVerilog. Umożliwia połączenie razem danych, funkcjonalności i synchronizację komponentu projektu. Może reprezentować komponenty projektu na dowolnym poziomie, od najniższego (np. pojedyncza bramka) do najwyższego (np. złożony system cyfrowy). Moduły pozwalają opisać komponenty projektu w różnym stopniu szczegółowości, bardzo dokładnie ze wszystkimi detalami lub na poziomie abstrakcyjnym. Moduły mogą zawierać instancje innych modułów, tworząc w ten sposób hierarchię projektu. Ponadto środowisko testowe (*testbench*) projektu jest opisywane także za pomocą modułu (patrz punkt 1.9).

Język SystemVerilog w porównaniu z językiem Verilog znacznie rozszerza liczbę konstrukcji językowych, które mogą pełnić funkcję elementów modułu. Mogą to być np. interfejsy, moduły zagnieżdżone, aliasy sieciowe itp.

W języku Verilog przez porty modułów mogą być przekazywane tylko sieci i zmienne (skalarne i wektorowe). Nakłada to znaczne ograniczenia i stwarza spore niedogodności przy realizacji dużych projektów. Język SystemVerilog usuwa te ograniczenia, pozwalając na przekazywanie struktur, unii, interfejsów i tablic o dowolnej liczbie wymiarów przez porty modułów. Ponadto zmienne mogą być przekazywane przez porty modułu przez referencję, tak jak w językach programowania.

Język SystemVerilog upraszcza również deklarację portu, pozwalając w niektórych przypadkach na pominięcie kierunku portu.

W Verilogu podczas tworzenia złożonych projektów, w których moduły mają dużą liczbę portów, deweloperzy borykają się z problemem nadmiarowych opisów instancji modułów. W tym przypadku nazwy sygnałów często pokrywają się z nazwami portów. Przykładem jest system mikroprocesorowy. Aby rozwiązać ten problem, język SystemVerilog oferuje notację **.name** oraz **.***. Zapis **.name** dopuszcza tylko jedną nazwę, jeśli nazwa sygnału jest taka sama jak nazwa portu. Notacja .* określa, że wszystkie porty i sygnały o tej samej nazwie muszą być połączone razem dla danej instancji modułu.

W języku Verilog wszystkie nazwy modułów są widoczne z dowolnego miejsca w projekcie. W dużych projektach tworzonych przez różne zespoły inżynierów może to prowadzić do konfliktów nazw, gdy różne moduły mają tę samą nazwę. Aby rozwiązać ten problem, w języku SystemVerilog wprowadzono moduły zagnieżdżone, które są deklarowane wewnątrz innych modułów. Nazwy modułów zagnieżdżonych są widoczne w module nadrzędnym, a także w dowolnym miejscu drzewa hierarchii poniżej modułu zagnieżdżonego.

Gdy duże projekty są opracowywane przez różne zespoły inżynierów, przypisywanie różnych nazw tym samym portom może być trudne, np. wejście zegarowe może być nazwane clk, CLK, clock itp. Aby rozwiązać ten problem, język SystemVerilog udostępnia operator sieci aliasów **alias**. Jego użycie umożliwia efektywne wykorzystanie notacji **.name** i **.*** w modułach, które mają różne nazwy portów.

W SystemVerilog typy danych mogą być określane jako parametry. Dzięki temu można tworzyć instancje modułów, które pracują z różnymi typami danych.

Deweloperzy często borykają się z problemem kompilacji dużych projektów, co wymaga dużo czasu i zasobów komputerowych. Aby rozwiązać ten problem, SystemVerilog umożliwia kompilację dużych projektów w częściach. W tym celu wprowadzono pojęcie jednostki kompilacji (*compilation unit*): zbioru kodu źródłowego, który jest kompilowany w tym samym czasie. Może się ona jednak odwoływać do modułów, które są zadeklarowane w innych jednostkach kompilacji. W SystemVerilog problem ten rozwiązuje się za pomocą prototypów modułów zwanych *modułami zewnętrznymi* (*extern modules*).

2.1. Definicja modułu

Język SystemVerilog obsługuje dwa formaty definicji modułów: *stary format* (styl non-ANSI) i *nowy format* (styl ANSI), gdzie ANSI oznacza Amerykański Narodowy Instytut Standaryzacji (American National Standards Institute).

Stary format definicji modułu używany w Verilog-1995 jest następujący:

```
module module_name (port_name_list);
        parameter_declaration_list
        port_ declarations
        module_items
endmodule
```

Nowy format definicji modułu wprowadzony w Verilogu-2001 jest następujący:

```
module module_name
        # (parameter_list)
        (port_declaration_list);
        module_items
endmodule
```

gdzie: **module, endmodule** – słowa kluczowe; *module_name* – nazwa modułu; *port_name_list* – lista nazw portów modułu; *parameter_declaration_list* – lista deklaracji parametrów; *port_declarations* – deklaracje portów; *module_items* – elementy modułu; *parameter_list* – lista parametrów; *port_declaration_list* – lista deklaracji portów.

W tej książce będziemy trzymać się nowego formatu deklaracji modułów, dlatego przyjrzymy się mu bardziej szczegółowo.

Podczas deklarowania modułu w stylu ANSI po słowie kluczowym **module** bezpośrednio występuje nazwa modułu (*module_name*). Po nim następuje lista parametrów modułu w nawiasach po znaku „#", a następnie lista portów w nawiasach, zakończona średnikiem. Należy zauważyć, że średnik w tym miejscu jest obowiązkowym elementem formatu.

Nazwa modułu wraz z listą parametrów i listą portów jest czasami nazywana *nagłówkiem modułu* (*module header*). Po nim zaś następują jego elementy, czyli *ciało modułu* (*module body*).

Opis modułu kończy się słowem kluczowym **endmodule**. Zamiast słowa kluczowego **module** można użyć słowa kluczowego **macromodule** (dla dużych części systemu).

Lista parametrów jest opcjonalnym elementem deklaracji modułu. Jeśli moduł nie ma parametrów, bezpośrednio po jego nazwie podawana jest lista portów modułu, która jest również opcjonalnym elementem deklaracji modułu. Jest to związane z tym, że środowisko testowe projektu nie ma portów.

Aby zwiększyć czytelność projektu, po słowie kluczowym **endmodule** można umieścić dwukropek i nazwę modułu, np.

```
module counter
    # (...)          // lista parametrów
    (...);           // lista portów

    // ... elementy modułu

endmodule: counter
```

W starym formacie deklaracja modułu (w stylu non-ANSI) po jego nazwie po prostu wymienia nazwy portów w nawiasach, a deklaracja parametrów i portów modułu jest wykonywana w jego treści.

Jako przykład deklaracji modułu rozpatrzmy opis projektu w języku SystemVerilog dla jednobitowego sumatora, którego działanie można przedstawić za pomocą następujących równań logicznych:

$$s = a \oplus b \oplus cin;$$

$$cout = a \cdot b + (a \oplus b) \cdot cin,$$

gdzie: znak $\oplus$ oznacza operację logiczną alternatywy wykluczającej (XOR), znak · oznacza operację iloczynu logicznego (AND), a znak + oznacza operację sumy logicznej (OR).

Przykład 2.1. Opis sumatora jednobitowego w stylu ANSI

```
module add_ANSI
        (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 poniższym przykładzie opisano ten sam projekt, ale w starym stylu non-ANSI.

Przykład 2.2. Opis sumatora jednobitowego w stylu non-ANSI

```
module add_non_ANSI (cin, a, b, s, cout);   // opis portu modułu
        input cin, a, b;                    // deklaracja portu wejściowego
        output s, cout;                     // deklaracja portu wyjściowego

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

W przykładach 2.1 i 2.2 do opisu projektu jednobitowego sumatora użyto operatorów przypisania ciągłego **assign**. Operatory te są zawsze wykonywane równolegle, niezależnie od ich miejsca w kodzie projektu. Tutaj po prawej stronie znaku równości opisywane są wyrażenia wykorzystujące operacje logiczne oznaczone następującymi znakami: „|" – OR, „&" – AND, „^" – XOR. Należy również pamiętać, że negacja jest oznaczana znakiem „~".

2.2. Style i poziomy opisu projektu

Omówione powyżej style non-ANSI i ANSI są związane z opisem nagłówka modułu. Aby opisać ciało modułu, w języku SystemVerilog istnieją dwa podstawowe *style opisu modułu*: *strukturalny* (*structural*) i *behawioralny* (*behavioral*). Możliwy jest także mieszany styl opisu modułu *behawioralno-strukturalny*. Styl opisu projektu przedstawiony w przykładach 2.1 i 2.2 jest najczęściej stosowany do opisu układów kombinacyjnych opartych na *równaniach logicznych* (*boolowskich*) – jest on stylem behawioralnym opisu.

2.2.1. Styl strukturalny opisu projektu

Styl strukturalny opisuje instancje modułów i prymitywów oraz połączenia między nimi. Zmienne pośrednie mogą być używane do przekazywania sygnałów między elementami struktury. W stylu strukturalnym schematy logiczne, strukturalne i schematy obwodów są opisywane w podobny sposób jak w edytorach graficznych.

Aby zademonstrować styl strukturalny, rozważmy opis jednobitowego sumatora za pomocą prymitywów SystemVerilog, takich jak bramki or, and, xor i not.

Przykład 2.3. Strukturalny opis sumatora jednobitowego

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

        wire g1_o, g2_o, g3_o;                  // zmienne pośrednie

        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
```

Jeśli do opisu projektu potrzebne są dodatkowe zmienne, należy je zadeklarować przed pierwszym użyciem. W naszym przykładzie są to *sieci* (**wire** – przewody) odpowiadające wyjściom bramek: *g1_o*, *g2_o* i *g3_o*. Rzeczywisty opis naszego projektu składa się z opisów instancji każdej bramki. Ponieważ bramki logiczne należą do prymitywów języka Verilog, można ich używać bez wcześniejszego opisu lub deklaracji.

Opis każdej bramki składa się z nazwy jej typu (**and, or, xor**), nazwy instancji (*g1*, *g2*, *g3*, *g4*, *g5*) oraz listy portów, która jest listą sygnałów przypisanych do portów danej instancji. Charakterystyczne dla bramek jest to, że wyjście bramki na liście portów jest zawsze na pierwszej pozycji, po której następują jej wejścia. Wiąże się to z tym, że bramki mogą mieć dowolną liczbę wejść i tylko jedno wyjście. Zadeklarowane wcześniej dodatkowe zmienne *g1_o*, *g2_o* i *g3_o* służą do przekazywania sygnałów między bramkami.

Skalarne (jednobitowe) połączenia wewnętrzne w języku SystemVerilog nie muszą być deklarowane, kompilator tworzy je automatycznie. W związku z tym linię:

```
wire g1_o, g2_o, g3_o;
```

w przykładzie 2.3 można pominąć.

2.2.2. Styl behawioralny opisu projektu

Język SystemVerilog pozwala również na opisywanie projektów na poziomie behawioralnym, czyli algorytmu działania, za pomocą *operatorów proceduralnych*. Aby opisać nasz projekt sumatora jednobitowego na poziomie behawioralnym, zidentyfikujmy pewne prawidłowości w jego działaniu. Na przykład funkcja przenoszenia *cout* będzie równa jeden, jeśli na wejściach sumatora pojawią się jednocześnie co najmniej dwie jedynki. W tym przypadku opis działania sumatora jednobitowego można przedstawić w następujący sposób.

Przykład 2.4. Opis sumatora jednobitowego z użyciem operatorów proceduralnych

```
module add_proc
        (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)      // opis funkcji przeniesienia
                cout=1;
        else
                cout=0;
        s = a ^ b ^ cin;                    // opis funkcji sumy
        end
endmodule
```

W przykładzie 2.4 zastosowano proceduralny operator **if-else** oraz proceduralny operator przypisania blokującego „=". Ponadto w funkcjach wyjściowych *s* i *cout* wykorzystano typ **reg**. Należy pamiętać, że w języku SystemVerilog zamiast **reg** można użyć typu **logic**.

2.2.3. Poziomy opisu projektu

Ogólnie rzecz biorąc, język SystemVerilog pozwala na opisywanie projektów na następujących poziomach:

- tranzystorów;
- bramek;
- równań logicznych;
- przesłań międzyrejestrowych (RTL – *register transfer level*);
- algorytmicznym;
- behawioralnym (*behavioral*);
- systemowym;

- abstrakcyjnym;
- transakcji.

Opis projektów na poziomie tranzystorów jest rzadko używany. Na przykład ten styl opisu może być stosowany podczas projektowania nowych elementów biblioteki dla układów ASIC (*application-specific integrated circuit*).

Przykład opisu projektu na poziomie bramek pokazano w przykładzie 2.3.

Aby opisać projekt na poziomie równań logicznych, wystarczy przedstawić go jako system równań logicznych. Można stosować zmienne pośrednie. Przykład opisu projektu na poziomie równań logicznych przedstawiono w przykładach 2.1 i 2.2.

Opis na poziomie przesłań międzyrejestrowych (RTL) różni się od opisu projektów na poziomie bramek tylko tym, że zamiast bramek używane są elementy funkcjonalne poziomu RTL: rejestry, liczniki, sumatory, dekodery, multipleksery itp.

Operatory proceduralne języka SystemVerilog są używane do opisywania projektów na poziomie behawioralnym. Przykład opisu jednobitowego sumatora na tym poziomie przedstawiono w przykładzie 2.4.

Opis projektu na poziomie systemu jest podobny do opisu na poziomie RTL z tą różnicą, że zamiast prostych elementów funkcjonalnych (np. rejestry) używa się bloków funkcjonalnych na poziomie systemu (procesory, pamięć, magistrale, urządzenia sterujące, urządzenia wejścia/wyjścia itp.).

W języku SystemVerilog, w porównaniu z językiem Verilog, dodano wiele konstrukcji ułatwiających opisywanie dużych i złożonych projektów, w tym na poziomie systemu. W języku SystemVerilog wyróżnia się czasem *poziom abstrakcyjny*, który jest używany podczas symulacji systemów cyfrowych. Język SystemVerilog umożliwia również opisywanie projektów na *poziomie transakcji*.

2.3. Elementy modułu

Elementy modułu (*module items*) definiują zawartość modułu lub *ciała modułu* (*module body*).

Elementy modułu mogą stanowić następujące konstrukcje języka Verilog:

- deklaracje typów danych;
- deklaracje parametrów;
- instancje (*instances*) modułów;
- instancje prymitywów;
- bloki generowania kodu (*generate blocks*);
- bloki proceduralne (*procedural blocks*);
- przypisania ciągłe (*continuous assignments*);
- definicje zadań (*tasks*);
- definicje funkcji (*functions*);
- bloki specyfikacji (*specify blocks*).

Mogą również tworzyć dodatkowe konstrukcje języka SystemVerilog:

- instancje interfejsu (*interface*);
- instancje programu (*program*);
- elementy zapewnień (*assertion*);
- aliasy sieciowe (*net alias*);
- deklaracje innych modułów (*module declaration*);
- deklaracje programów (*program declaration*);
- deklaracje interfejsów (*interface declaration*);
- deklaracje jednostek czasu (*timeunits declaration*);
- region generowania (*generate region*).

Należy pamiętać, że nie wszystkie konstrukcje języka SystemVerilog są syntetyzowalne.

2.4. Deklaracje portów

Porty modułu służą do przekazywania danych do komponentu projektu opisywanego przez ten moduł. Język SystemVerilog obsługuje dwa formaty deklaracji portów. Format non-ANSI odpowiada formatowi deklaracji w języku Verilog-1995, natomiast format ANSI odpowiada formatowi Verilog-2001. W celu zapewnienia kompatybilności z językiem Verilog język SystemVerilog obsługuje oba formaty.

Zapis deklaracji portu w formacie ANSI jest następujący:

port_direction data_type **signed** *range port_name, port_name, ... ;*

gdzie: *port_direction* jest kierunkiem transmisji sygnału; *data_type* jest typem przesyłanych danych; **signed** jest słowem kluczowym; *range* jest określeniem zakresu pola bitowego; *port_name* jest nazwą portu.

Zapis deklaracji portu w formacie non-ANSI jest następujący:

port_direction **signed** *range port_name, port_name, ... ;*
port_type_declarations

gdzie *port_type_declarations* są deklaracjami typów portów.

Kierunek przesyłania sygnału (*port_direction*) może mieć następującą wartość:

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

Język Verilog miał dość ścisłe ograniczenia dotyczące *typów danych* przesyłanych przez porty modułów. Na przykład przez porty modułów można w nim przekazywać tylko zmienne i sieci. Ponadto aby przekazać zmienne typu rzeczywistego (**real**), ich wartości muszą być najpierw przekształcone na strumień bitów za pomocą *funkcji systemowych* **$realtobits** i **$bitstoreal**.

Język SystemVerilog znacznie rozszerza zakres typów danych przesyłanych przez porty modułów. Można przez nie przekazywać następujące elementy:

- zmienne;
- dowolne typy danych sieciowych;
- tablice;
- struktury;
- unie;
- interfejsy.

W SystemVerilog zdarzenia (**event**) i referencje zmiennych (**ref**) mogą być również przekazywane przez porty, ale te typy danych nie są syntetyzowalne i nie są omawiane w tej książce.

Słowo kluczowe **signed** oznacza, że wartości przekazywane przez port są interpretowane jako liczba ze znakiem w kodzie uzupełnienia do 2. Jeśli typ danych portu lub typ danych sygnału podłączonego do portu jest zadeklarowany jako wartość ze znakiem, oba są traktowane jako wartości ze znakiem.

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

[*msb*:*lsb*]

gdzie *msb* to numer najbardziej znaczącego bitu, a *lsb* to numer najmniej znaczącego bitu.

Jeśli nie podano zakresu, port ma szerokość jednego bitu, a przesyłane sygnały są traktowane jako *skalar* (*scalar*); w przeciwnym razie zaś jako *wektor* (*vector*).

Jako numer najbardziej (*msb*) i najmniej (*lsb*) znaczącego bitu mogą występować: liczby, stałe, wyrażenia i odwołania do *funkcji stałej* (funkcja, która zwraca stałą wartość). Specyfikacja zakresu dopuszcza zarówno rosnącą (*big-endian*), jak i malejącą (*little-endian*) *kolejność liczenia bitów* (tzn. [0:7] i [8:1] są dozwolonymi pozycjami zakresu).

Poniżej przedstawiono przykłady deklaracji portów w stylu non-ANSI:

```
module test_non_ANSI
    (a, b, sel, c, d, result, sum, data_bus, addr, addr2, addr3); // lista nazwy portów
    input a, b, sel;              // trzy porty skalarne

    input signed [15:0] c, d;     /* dwa porty 16-bitowe, które przekazują dane w kodzie
                                     uzupełnienia do 2; kolejność bitów to little endian */
```

```
        output signed [31:0] result;          // to samo, ale port 32-bitowy

        output reg signed [32:1] sum;         /* port 32-bitowy, sygnały wewnętrzne
                                                 podłączone do portu mają typ danych reg
                                                 ze znakiem; kolejność bitów jest little-endian */

        input [0:15] data_bus;                /* dwukierunkowy port 16-bitowy
                                                 z kolejnością bitów big-endion */

        inout [15:12] addr;                   /* cztery bity wektora addr są używane
                                                 jako wejścia (msb i lsb mogą mieć
                                                 dowolne wartości) */

        parameter WORD = 32;
        input [WORD-1:0] addr2;               /* w deklaracji portu można używać
                                                 wyrażeń stałych */

        parameter SIZE=4096;
        input [log2(SIZE)-1:0] addr3;         /* można też wywoływać funkcje stałe */
        …
endmodule: test_non_ANSI
```

Te same deklaracje portów w stylu ANSI:

```
module test_ANSI #( parameter WORD = 32, SIZE=4096)
        (input a, b, sel,                     // trzy porty skalarne

        input signed [15:0] c, d,             // dwa porty 16-bitowe

        output signed [31:0] result,          // port 32-bitowy

        output reg signed [32:1] sum,         // 32-bitowy port typu reg ze znakiem

        inout [0:15] data_bus,                // dwukierunkowy port 16-bitowy

        input [15:12] addr,                   // cztery bity wektora addr są używane
                                                 jako wejścia

        input [WORD-1:0] addr2,               // w deklaracji portu można używać
                                                 wyrażeń stałych

        input [log2(SIZE)-1:0] addr3);        // i można wywoływać funkcje stałe
        …
```

endmodule: test_ANSI

W języku SystemVerilog wprowadzono kilka ulepszeń, które upraszczają deklaracje portów dzięki temu, że są one bardziej zwarte.

Po pierwsze, w języku SystemVerilog można nie określać kierunku portu. W takim przypadku domyślnym kierunkiem będzie **inout**, czyli zostanie zadeklarowany port dwukierunkowy.

Po drugie, jeśli następny port na liście portów jest określonego typu, ale nie określono kierunku, domyślnie ustawiany jest kierunek poprzedniego portu na liście. Umożliwia to zmianę specyfikacji typu bez konieczności ponownego określania kierunku portu.

Załóżmy, że mamy opis portu w stylu ANSI w języku Verilog-2001 dla modułu accum:

```
module accum (inout wire [31:0] data,
        output reg [31:0] result,
        output reg co,
        input [31:0] a, b,
        input tri1 ci );

endmodule: accum
```

Ten opis w SystemVerilog może wyglądać następująco:

```
module accum (wire [31:0] data,
        output reg [31:0] result, reg co,
        input [31:0] a, b, tri1 ci );

endmodule
```

Pierwszy port na liście (*data*) jest typu (**wire**), ale nie ma wyspecyfikowanego kierunku portu. Dlatego domyślnie port ten ma kierunek **inout**. Port *co* jest również typu (**reg**), ale nie ma podanego kierunku portu. Domyślnie port ten ma kierunek zgodny z kierunkiem poprzedniego portu (*result*) na liście, czyli **output**. Porty *a* i *b* mają zadeklarowany kierunek portu (**input**), ale nie mają typu. Podobnie jak w Verilogu-2001, wyprowadzany jest niejawny typ sieciowy, którym domyślnie jest typ **wire**. Wreszcie port *ci* ma zadeklarowany typ (**tri1**), ale nie ma kierunku portu. Port ten przejmuje kierunek poprzedniego portu na liście, czyli **input**.

2.5. Instancje modułów

Zanim moduł będzie mógł zostać użyty jako komponent projektu, musi zostać utworzona *instancja modułu* (*module instance*). W pewnym sensie stanowi ona analogię do wywołania funkcji w językach programowania. O ile jednak w językach programowania wywołanie funkcji odpowiada wywołaniu części kodu programu, o tyle utworzenie instancji modułu odpowiada umieszczeniu na płycie drukowanej instancji układu scalonego o określonym oznaczeniu.

Podobnie jak po umieszczeniu układu scalonego na płytce należy podłączyć do jego wyprowadzeń linie z przesyłanymi sygnałami, tak podczas tworzenia instancji modułu należy określić sygnały, które mają być podłączone do portów modułu.

Istnieją dwa podstawowe sposoby podłączania sygnałów do portów instancji modułu: *przez pozycję* (*by position*) portu i *przez nazwę* (*by name*) portu. Pierwsza metoda jest nazywana *uporządkowanym połączeniem portów* (*ordered port connections*), a lista portów – *uporządkowaną listą portów* (*ordered port list*). Druga metoda nosi nazwę *nazwanych połączeń portów* (*named port connections*), a lista portów – *nazwanej listy portów* (*named port list*).

Format tworzenia instancji modułu, gdy sygnały są przekazywane przez pozycję, jest następujący:

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

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

Gdy do jakiegoś portu instancji modułu nie jest podłączony żaden sygnał, w danym miejscu umieszczane są dwa przecinki w rzędzie. Technika ta jest nazywana *pomijaniem sygnału* (*skipping signal*). Jeśli trzeba pominąć kilka sygnałów, na liście sygnałów umieszcza się kilka przecinków.

Na przykład konstrukcja sumatora dwubitowego z dwóch instancji sumatora jednobitowego powinna wyglądać następująco.

Przykład 2.5. Opis sumatora dwubitowego na podstawie dwóch sumatorów jednobitowych

```
module add_2(
        input wire [1:0] A, B,          // szyny wejściowe
        input CIN,                      // przeniesienie wejściowe
        output wire [1:0] S,            // suma
        output COUT);                   // przeniesienie wyjściowe

        wire C0;                        // przeniesienie od zerowego bitu
```

```
    // tworzenie instancji modułu add_ANSI
    add_ANSI    sum0 (A[0], B[0], CIN, S[0], C0);
    add_ANSI    sum1 (A[1], B[1], C0, S[1], COUT);
endmodule
```

W powyższym przykładzie *add_ANSI* to nazwa modułu sumatora jednobitowego opisanego w przykładzie 2.1; *sum0* i *sum1* to nazwy dwóch instancji tego modułu. Należy zauważyć, że w tym projekcie linia pośrednia *C0* została użyta do transferu sygnału przeniesienia między dwoma instancjami jednobitowego sumatora.

Format tworzenia instancji modułu, gdy sygnały są przesyłane przez *nazwę portu* (*by name*), jest następujący

module_name instance_name (.*port_name* (*signal*), .*port_name* (*signal*),…);

gdzie *port_name* jest nazwą portu, do którego przesyłany jest odpowiedni sygnał. Należy pamiętać, że w tym formacie każda nazwa portu jest poprzedzona kropką („.").

Jeśli do jakiegoś portu instancji modułu nie jest podłączony żaden sygnał, to nazwa tego portu po prostu nie jest wyświetlana na liście portów instancji modułu.

Przekazywanie sygnałów według nazw portów wymaga większej liczby znaków w kodzie, ale zwalnia dewelopera z wiedzy o kolejności portów w deklaracji modułu. Ponadto metoda ta zmniejsza liczbę błędów w kodzie projektu. Dlatego jest ona używana podczas tworzenia instancji modułów z dużą liczbą portów.

Jako przykład rozważmy projekt rejestru czterobitowego opartego na prymitywie **dff** przerzutnika typu D. Przy tym kolejność portów jest nam nieznana, ale znamy ich nazwy: *d* – wejście danych; *q* – wyjście; *clk* – wejście sygnału zegarowego.

Przykład 2.6. Opis rejestru czterobitowego na podstawie prymitywu **dff** przerzutnika typu D

```
module reg_4
    (input wire [3:0] D,                    // szyna danych wejściowych
    input wire CLK,                         // sygnał zegarowy
    output wire [3:0] Q);                   // szyna danych wyjściowych

    // tworzenie instancji prymitywu dff
    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 tym przykładzie użyliśmy tylko trzech portów prymitywu **dff**: *clk*, *d* i *q*, przy czym możemy nie znać informacji o kolejności portów, jak również nazw pozostałych portów.

Tworzone instancje modułów mogą być w zapisie oddzielone przecinkami, bez powtarzania nazwy modułu, np.:

```
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]));
```

Jeśli nie ma potrzeby odwoływania się do konkretnej instancji modułu, można również pominąć jej nazwę, np.:

```
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]));
```

2.6. Niejawne połączenia portów

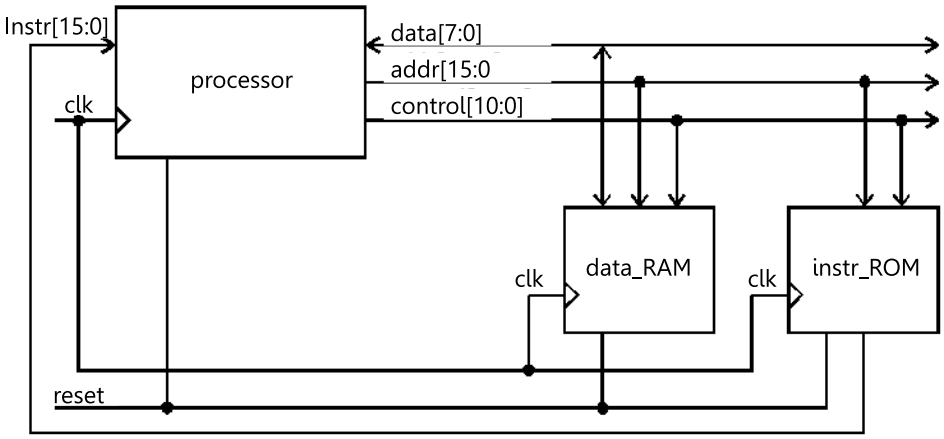
Wcześniej omówiono dwa sposoby podłączania sygnałów do portów instancji modułów: *według pozycji portu* (*by position*) oraz według *nazwy portu* (*by name*). Obie te metody mają wady: pierwsza z nich wymaga znajomości kolejności deklarowania portów w module, a druga jest zbyt czasochłonna, gdyż wymaga podania nazwy portu i nazwy sygnału dla każdego portu.

Dla *niejawnych połączeń portów* (*implicit port connections*) język SystemVerilog oferuje notacje **.name** i **.*** oraz interfejsy. W tej części omówimy notacje **.name** i **.***, interfejsy omówiono w rozdziale 14.

2.6.1. Notacja .name dla niejawnego połączenia portów

W praktyce, zwłaszcza w projektach wyższego poziomu, nazwa sygnału i nazwa portu instancji modułu, do którego sygnał jest podłączony, często są takie same. Na przykład w systemie mikroprocesorowym szyna danych *data* jest połączona z portami o nazwie *data*. Notacja **.name** pozwala na podanie tylko jednej nazwy, jeśli nazwa sygnału jest taka sama jak nazwa portu. Oznacza to, że zapis .data(data) można zredukować do .data.

Rozważmy prosty system mikroprocesorowy przedstawiony na rys. 2.1.



RYS. 2.1. Prosty system mikroprocesorowy

ŹRÓDŁO: oprac. własne.

Opis portów modułów dla elementów układu z rys. 2.1 może wyglądać następująco:

Przykład 2.7. Opis portów prostego systemu mikroprocesorowego

```
module processor
        (inout [7:0] data,
         input [15:0] Instr,
        output [15:0] addr,
         output [10:0] control,
         input clk, reset);
         …
endmodule: processor

module data_RAM
        (inout [7:0] data,
         input [15:0] addr,
         input [10:0] control,
         input clk, reset);
        …
endmodule: data_RAM

module instr_ROM
        (input [15:0] addr,
        input [10:0] control,
        output [15:0] Instr,
        input clk, reset);
        …
endmodule: instr_ROM
```

Moduł najwyższego poziomu dla schematu z rys. 2.1 jest następujący:

```
module micro_processor
        (input clk, reset,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        processor my_proc (.data(data), .Instr(Instr), .addr(addr),
                .control(control), .clk(clk), .reset(reset));

        data_RAM dRAM (.data(data), .addr(addr), .control(control),
                .clk(clk), .reset(reset));

        instr_ROM iROM (.addr(addr), .control(control), .Instr(Instr),
                .clk(clk), .reset(reset));
endmodule: micro_processor
```

Zastosowano tu łączenie sygnałów i portów modułów według nazw. Użycie notacji **.name** w opisie modułu najwyższego poziomu pozwala skrócić opis całego systemu.

Przykład 2.8. Zastosowanie notacji **.name** w opisie modułu najwyższego poziomu systemu mikroprocesorowego

```
module micro_processor
        (input clk, reset,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        processor my_proc (.data, .Instr, .addr, .control, .clk, .reset);

        data_RAM dRAM (.data, .addr, .control, .clk, .reset);

        instr_ROM iROM (.addr, .control, .Instr, .clk, .reset);

endmodule: micro_processor
```

Jeśli nazwa sygnału jest inna niż nazwa portu, można użyć połączenia przez nazwę. Załóżmy np., że elementy data_RAM i instr_ROM są taktowane odpowiednio sygnałami clk1 i clk2. W tym przypadku opis modułu najwyższego poziomu wyglądałby następująco:

Przykład 2.9. Zastosowanie notacji **.name** i łączenia przez nazwę

```
module micro_processor
        (input clk, reset,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        processor my_proc (.data, .Instr, .addr, .control, .clk, .reset);

        data_RAM dRAM (.data, .addr, .control, clk(clk1), .reset);

        instr_ROM iROM (.addr, .control, .Instr, clk(clk2), .reset);
endmodule: micro_processor
```

2.6.2. Notacja .* dla niejawnego połączenia portów

Notacja .* określa, że wszystkie porty i sygnały o tej samej nazwie muszą być połączone razem dla tej instancji modułu. Podobnie jak w przypadku notacji **.name**, nazwa i rozmiar wektorów muszą być dokładnie takie same, aby utworzyć połączenie, a typy sygnałów i portów muszą być zgodne.

Używając notacji .*, opis modułu najwyższego poziomu dla naszego przykładu wygląda następująco.

Przykład 2.10. Zastosowanie notacji .* w opisie modułu najwyższego poziomu systemu mikroprocesorowego

```
module micro_processor
        (input clk, reset,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        processor my_proc (.*);

        data_RAM dRAM (.*);

        instr_ROM iROM (.*);

endmodule: micro_processor
```

Jeśli nazwa sygnału jest inna niż nazwa portu, można użyć połączeń przez nazwę, aby połączyć je w notacji .*.

Przykład 2.11. Zastosowanie notacji .* i łączenia przez nazwę

```
module micro_processor
        (input clk, reset,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        processor my_proc (.*);

        data_RAM dRAM (.*, clk(clk1));

        instr_ROM iROM (.*, clk(clk2));

endmodule: micro_processor
```

2.7. Moduły zagnieżdżone

W języku Verilog nazwy wszystkich modułów, prymitywów użytkownika oraz nazwy zadań i funkcji systemowych są umieszczane w *globalnej przestrzeni nazw* (*global name space*). Oznacza to, że do ich nazw można się odwoływać z dowolnego miejsca w hierarchii projektu. Na przykład dowolny moduł może utworzyć instancję dowolnego innego modułu, niezależnie od kolejności kompilacji plików.

Jednak globalny dostęp do wszystkich nazw modułów w projekcie może prowadzić do *konfliktów nazw* (*naming conflicts*). Na przykład jeśli projekt użytkownika oraz używany *blok IP* (*intellectual property*) zawierają moduł o nazwie RAM.

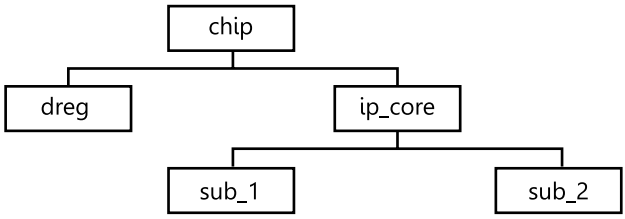
Aby rozwiązać problem konfliktu nazw, język SystemVerilog oferuje proste i eleganckie rozwiązanie polegające na zastosowaniu *modułów zagnieżdżonych* (*nested modules*). Rozwiązaniem jest jego zdefiniowanie w innym module. W rezultacie nazwy zagnieżdżonych modułów nie są widoczne poza kontekstem hierarchii, w której zostały zadeklarowane. Poniższy przykład ilustruje opis modułów zagnieżdżonych.

Przykład 2.12. Deklaracje modułów zagnieżdżonych

```
module chip (input wire clock);      // moduł najwyższego poziomu
        dreg i1 (clock);
        ip_core i2 (clock);
endmodule: chip

module dreg (input wire clock);      // definicja modułu globalnego
```

```
endmodule: dreg

module ip_core (input wire clock);  // definicja modułu globalnego
        sub1 u1 (...);
        sub2 u2 (...);

        module sub1(...);           // definicja modułu zagnieżdżonego

endmodule: sub1

module sub2(...);                   // definicja modułu zagnieżdżonego

endmodule: sub2

endmodule: ip_core
```

Drzewo hierarchii modułów z przykładu 2.12 jest pokazane na rys. 2.2.



RYS. 2.2. Drzewo hierarchii modułów z przykładu 2.7

ŹRÓDŁO: oprac. własne.

Nazwa modułu zagnieżdżonego nie jest umieszczana w globalnej przestrzeni nazw projektu, ale jest nazwą lokalną w module nadrzędnym. Oznacza to, że moduł zagnieżdżony może mieć taką samą nazwę jak inny moduł zdefiniowany w innym miejscu projektu, bez konfliktu w globalnej przestrzeni nazw.

Instancję modułu zagnieżdżonego można utworzyć zarówno w module nadrzędnym, jak i w dowolnym innym miejscu drzewa hierarchii poniżej modułu zagnieżdżonego. Instancji modułu zagnieżdżonego nie można utworzyć w żadnym innym miejscu w hierarchii projektu. W przykładzie 2.12 moduły *chip*, *dreg* oraz *ip_core* znajdują się w globalnym kontekście nazw. Moduły te mogą tworzyć instancje dowolnych innych modułów w dowolnym miejscu hierarchii projektu. Moduły *sub1* i *sub2* są zagnieżdżone w definicji modułu *ip_core*. Nazwy modułów *sub1* i *sub2* są nazwami lokalnymi w obrębie modułu *ip_core* i mogą być wywoływane tylko w module *ip_core* lub w modułach, które są w nim wywoływane. *Konkretyzacja* (*instantiation*) to inaczej tworzenie instancji.

Moduł zagnieżdżony, podobnie jak zwykły moduł, może tworzyć instancje innych modułów. Ich definicje muszą należeć do następujących *zakresów nazw* (*name scopes*):

- kontekst nazw globalnych;
- kontekst modułu nadrzędnego;
- kontekst modułu zagnieżdżonego.

Kod projektu z zagnieżdżonymi modułami może być dość duży i trudny do odczytania. Aby poprawić czytelność kodu projektu z modułami zagnieżdżonymi, zaleca się używanie nazwy modułu po słowie kluczowym **endmodule** dla modułu nadrzędnego i wszystkich modułów zagnieżdżonych, np.:

```
module A (...);
    ...
    module A_B (...);
    ...
    endmodule: A_B
    module A_C (...);
    ...
    endmodule: A_C
    ...
endmodule: A
```

Powszechną praktyką inżynierską jest opisywanie każdego modułu w osobnym pliku. Pozwala to na sprawną modyfikację złożonych projektów: jeśli w opisie modułu zostanie wykryty błąd, wystarczy poprawić tylko jeden plik. Należy zauważyć, że wiele narzędzi projektowych (Quartus, Vivado itp.) obsługuje właśnie taką organizację plików projektowych.

W przypadku modułów zagnieżdżonych sytuacja jest inna: pojedynczy plik zawiera opis kilku modułów (modułu nadrzędnego i wszystkich modułów zagnieżdżonych). Aby zapewnić, że każdy moduł zagnieżdżony jest opisany w osobnym pliku źródłowym, można użyć dyrektywy kompilatora **`include**, np.:

```
module A (...);
    ...
    `include A_B.sv      // moduł zagnieżdżony A_B
    `include A_C.sv      // moduł zagnieżdżony A_C
    ...
endmodule: A
```

2.8. Aliasy sieci

2.8.1. Operator alias

Język SystemVerilog udostępnia operator **alias**, który pozwala dwóm różnym nazwom odnosić się do tego samego obwodu, tzn. tworzy *alias obwodu* (*circuit alias*):

```
wire clock;
wire clk;
alias clk = clock;
```

W powyższym przykładzie operator **alias** określa, że tworzona jest jedna linia o dwóch różnych równych nazwach: *clk* i *clock*. Wszelkie zmiany w linii *clock* będą również widoczne w linii *clk* i odwrotnie, ponieważ to ta sama linia.

Zwróćmy uwagę na różnicę między operatorem **alias** a operatorem ciągłego przypisania **assign**. Ten ostatni w sposób ciągły kopiuje wyrażenie z prawej strony przypisania do sieci lub zmiennej z lewej strony. Jest to jednokierunkowa operacja kopiowania. Sieć bądź zmienna po lewej stronie odzwierciedla każdą zmianę w wyrażeniu po prawej stronie. Jednak zmiany w układzie lub zmiennej po prawej stronie nie mają odzwierciedlenia w wyrażeniu po lewej stronie. Na przykład:

```
wire clock;
wire clk;
assign clk = clock;
```

W tym przykładzie zostaną utworzone dwie linie: *clock* i *clk*. Zmiany w linii *clock* będą również odzwierciedlone w linii *clk*, ale zmiany w linii *clk* nie będą widoczne w linii *clock*.

Wiele linii można łączyć za pomocą operatora **alias**, np.:

```
wire clk, CLK, clock, Clock;
alias clk = CLK;
alias CLK = clock;
alias clock = Clock;
```

Ostatnie trzy operatory można zapisać za pomocą pojedynczego operatora **alias**:

```
alias clk = CLK = clock = Clock;
```

Powyższy operator definiuje listę nazw linii (typu sieciowego), które odnoszą się do tego samego obiektu.

Język SystemVerilog nakłada szereg ograniczeń na sygnały, których nazwy mogą być łączone za pomocą operatora **alias**. Ograniczenia te są następujące:

- łączyć można tylko typy sieciowe (**wire**, **uwire**, **wand**, **wor**, **tri**, **triand**, **trior**, **tri0**, **tri1** i **trireg**);
- łączyć można tylko linie tego samego typu, np. **wire** z **wire**, **wand** z **wand** itp.;
- łączone mogą być tylko elementy sieci o tym samym rozmiarze, tzn. rozmiary wektorów po lewej i prawej stronie operatora **alias** muszą być takie same (liczba wymiarów nie ma znaczenia).

2.8.2. Niejawne deklaracje obiektów sieciowych w operatorze alias

Jeśli obiekt typu sieć nie został jawnie zadeklarowany, operator **alias** może tworzyć deklaracje sieci. Zasady niejawnej deklaracji sieci są takie same jak w przypadku, gdy niejawny identyfikator jest dołączony do portu modułu:

- niejawna nazwa identyfikatora określa domyślny typ sieci;
- domyślnym typem sieci jest **wire**, który można zmienić za pomocą dyrektywy kompilatora **`default_nettype**;
- jeśli port modułu, który zawiera to wyrażenie **alias**, jest określony jako typ sieciowy, dołączona sieć będzie miała taki sam rozmiar wektora jak port;
- jeśli nazwa obiektu sieciowego nie jest portem modułu, deklarowany jest 1-bitowy element sieciowy.

Przykład niejawnej deklaracji sieci:

```
module register
        (output [63:0] q,
        input [63:0] d,
        input clock, reset);

        wire [63:0] out, in;
        alias in = d;           // jawna deklaracja 64-bitowej sieci in dla wejścia d
        alias out = q;          // jawne zadeklarowanie 64-bitowej sieci out dla wyjścia q
        alias rstN = reset;     // niejawna deklaracja 1-bitowej sieci rstN do wejścia reset

        ...
endmodule
```

Jednobitowy element sieci *rstN* jest tu zadeklarowany niejawnie, a 64-bitowe wektorowe elementy sieci *in* i *out* są zadeklarowane jawnie.

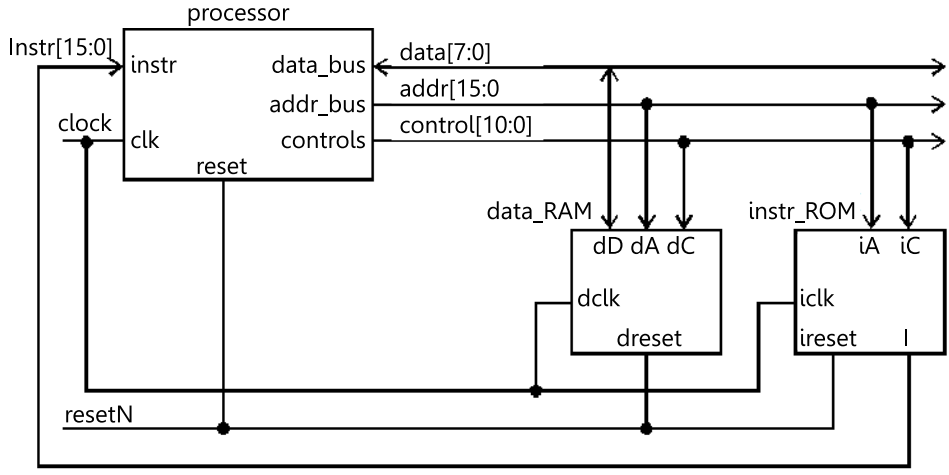
Operatora **alias** można również użyć do definiowania elementów sieci, które są częścią innej sieci. W poniższym przykładzie *lo_byte* to alias dla najmłodszego bajtu wektora, a *hi_byte* to alias dla najstarszego bajtu tego wektora.

```
module (...);
        wire [63:0] data;
        wire [7:0] lo_byte, hi_byte;
        alias data[7:0] = lo_byte;
        alias hi_byte = data[63:56];
        ...
endmodule
```

2.9. Użycie aliasów sieci z notacjami .name i .*

Notacje **.name** i **.*** sprawdzały się przy podłączaniu sygnałów do portów w modułach o tej samej nazwie. Często jednak zdarza się, że podobne porty w różnych modułach mają różne nazwy (np. gdy moduły zostały stworzone przez różnych deweloperów). W takich sytuacjach można użyć operator **alias**, aby utworzyć aliasy dla kompatybilnych portów modułu.

Rozważmy przykład systemu mikroprocesorowego przedstawiony na rys. 2.3.



RYS. 2.3. Przykład systemu mikroprocesorowego z różnymi nazwami portów modułów

ŹRÓDŁO: oprac. własne.

W tym przypadku różne komponenty mają różne nazwy portów, do których są podłączone te same sygnały. Moduł najwyższego poziomu, w którym sygnały są podłączone do portów instancji modułów według nazwy, wygląda następująco.

Przykład 2.13. Moduł systemu mikroprocesorowego najwyższego poziomu z różnymi nazwami portów modułów

```
module micro_processor_2
        (input clock, resetN,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        processor my_proc (.data_bus(data), .instr(Instr), .addr_bus(addr),
                .controls(control), .clk(clock), .reset(resetN));

        instr_ROM iROM (.iA(addr), .iC(control), .I(Instr),
                .iclk(clock), .ireset(resetN));

        data_RAM dRAM (.dD(data), .dA(addr), .dC(control),
                .dclk(clock), .dreset(resetN));

endmodule: micro_processor_2
```

Powyższy kod można uprościć, stosując operatory **alias** w następujący sposób.

Przykład 2.14. Zastosowanie operatora **alias** w systemie mikroprocesorowym z różnymi nazwami portów modułowych

```
module micro_processor_2
        (input clock, resetN,
        inout [7:0] data,
        output [15:0] addr,
        output [10:0] control);

        alias clock = clk = iclk = dclk;
        alias resetN = reset = ireset = dreset;
        alias Instr = I = instr;
        alias data = data_bus = dD;
        alias addr = addr_bus = iA = dA;
        alias control = controls = iC = dC;

        processor my_proc (.*);

        instr_ROM iROM (.*);

        data_RAM dRAM (.*);

endmodule: micro_processor_2
```

Przykład 2.14 tworzy aliasy nazw sygnałów i zgodnych portów instancji modułów. Notacja .* wyszukuje zgodność nazwy sygnału wśród wszystkich jego aliasów i podłącza sygnał, jeśli nazwa aliasu pasuje do nazwy sygnału.

Notacja **.name** działa w podobny sposób.

2.10. Przekazywanie wartości przez porty modułu

Język SystemVerilog w porównaniu z językiem Verilog znacznie rozszerza listę wartości, które mogą być przekazywane przez porty modułu. W języku SystemVerilog, oprócz sieci i zmiennych, przez porty modułów można przekazywać:

- wartości dowolnego typu;
- tablice (spakowane i rozpakowane) o dowolnym rozmiarze;
- struktury i unie.

Język SystemVerilog nakłada tylko dwa ograniczenia na wartości przekazywane przez porty modułów:

- zmienne mogą otrzymywać wartości tylko z jednego źródła (kontrolera);
- typy niespakowane (tablice, struktury i związki) muszą mieć ten sam format.

Pierwsze ograniczenie polega na tym, że zmienna może mieć tylko jedno źródło wartości. Może nim być:

- jeden port **output** lub **inout** modułu lub elementu prymitywnego;
- jeden ciągły operator przypisania **assign**;
- dowolna liczba zadań proceduralnych.

Dowolna liczba przypisań proceduralnych jest uważana za jedyne źródło wartości dla zmiennej. Dzieje się tak dlatego, że przypisania proceduralne są operatorami natychmiastowymi, które umożliwiają przechowywanie wartości, ale nie aktualizują jej w sposób ciągły (jak operator **assign**). Na przykład w instrukcji **if...else** do aktualizacji wartości zmiennej można użyć albo jednej, albo drugiej gałęzi, ale nie obu naraz. W symulacji wielokrotne przypisanie proceduralne do tej samej zmiennej zachowuje się tak, jak tymczasowe przypisanie do zmiennej, a zmienna przechowuje zawsze wartość ostatniego przypisania.

Należy również pamiętać, że przez porty modułów mogą być przekazywane także referencje do zmiennych (**ref**). Porty te są nazywane *portami referencyjnymi* (*reference ports*). Nie są one jednak syntetyzowalne, lecz wykorzystywane jedynie w symulacji.

2.11. Moduły parametryzowane

2.11.1. Deklaracje modułów parametryzowanych

Używanie parametrów w deklaracjach modułów oraz podczas tworzenia instancji modułów może w niektórych przypadkach znacznie zmniejszyć objętość kodu projektu, zwiększyć jego czytelność oraz zmniejszyć liczbę błędów przy kodowaniu.

W deklaracji modułu lista parametrów modułu zaczyna się od znaku „#" i poprzedza listę portów w module. Na przykład opis sparametryzowanego modułu sumatora może wyglądać następująco.

Przykład 2.15. Opis sparametryzowanego sumatora, w którym parametr N określa szerokość słów wejściowych i sumy

```
module add_N  # ( parameter N = 4)        // definicja parametru N za pomocą wartości 4
      (input [N-1: 0] a, b, input cin,     // definicja szerokości słów wejściowych
      output [N-1: 0] s,                   // definicja szerokości słowa wyjściowego
      output cout);

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

endmodule
```

W powyższym przykładzie parametr *N* służy do określenia szerokości słów wejściowych *a* i *b* oraz sumy *s*. W deklaracji modułu można określić domyślne wartości parametrów, które są stosowane, gdy parametr ten nie zostanie nadpisany podczas tworzenia instancji modułu. W naszym przykładzie domyślna wartość parametru *N* wynosi 4.

W przykładzie tym użyto również następujących operatorów: przypisania ciągłego **assign**, konkatenacji (sklejania bitów i pól bitowych) „{}" oraz sumy arytmetycznej „+". Operator konkatenacji po lewej stronie operatora przypisania **assign** jest konieczny, ponieważ w wyniku sumowania może pojawić się sygnał przeniesienia ze starszego bitu sumy.

Ogólnie rzecz biorąc, moduł może mieć więcej niż jeden parametr. W takim przypadku w deklaracji modułu jest podawana lista parametrów.

2.11.2. Tworzenie instancji modułów parametryzowanych

Tworzenie instancji parametryzowanego modułu również ma dwa formaty, *według pozycji* (*by position*):

module_name # (*value, value*, ...) *instance_name* (*signal*, ...)

i *przez nazwę* (*by name*):

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

gdzie *value* jest wartością przekazaną parametru dla danej instancji modułu, a *parameter_name* jest nazwą parametru.

Jak widać, formaty przekazywania parametrów do instancji modułu są niemal identyczne jak formaty przekazywania sygnałów do portów instancji modułu. Jedyną różnicą jest obecność znaku „#" przed listą przesyłanych wartości.

Przykłady tworzenia instancji modułów:

```
// Przykład tworzenia instancji 32-bitowego sumatora
wire [31:0] A, B, S;
wire CIN, COUT;
add_N # (32) sum_32_v1 (A, B, CIN, S, COUT );        // przekazywanie parametrów
                                                     // według pozycji
add_N # (.N (32)) sum_32_v2 (A, B, CIN, S, COUT);    // przekazywanie parametrów
                                                     // według nazwy
```

2.11.3. Typy parametryzowane

Język Verilog definiuje parametry modułów jako stałe lub stałe wyrażenia. Pozwala także na parametryzację typów sieci i zmiennych. W tym celu stosuje się parę słów kluczowych **parameter type**. *Typy parametryzowane* (*parameterized types*) mogą być definiowane na nowo dla każdej instancji modułu. Funkcja ta wprowadza dodatkowy poziom polimorfizmu do modelu języka SystemVerilog. Innymi słowy, w SystemVerilog zachowanie modułu może być zmieniane na podstawie typów sieci lub zmiennych instancji modułu.

Typy parametryzowane są syntetyzowalne, pod warunkiem że takie są typy domyślne lub typy nadrzędne.

W poniższym przykładzie parametrem jest typ zmiennych używanych przez sumator. Domyślnym typem jest **shortint**. Moduł *big_chip* zawiera trzy instancje sumatora. W instancji *i1* używany jest domyślny typ zmiennej sumatora, co oznacza, że jest to 16-bitowy sumator ze znakiem. Instancja *i2* zmienia typ zmiennej na **int**, dzięki czemu ta instancja jest 32-bitowym sumatorem ze znakiem. W instancji *i3* zmieniono typ zmiennej na **int unsigned**, dzięki czemu trzecia instancja jest 32-bitowym sumatorem bez znaku.

Przykład 2.16. Polimorficzny sumator wykorzystujący sparametryzowane typy zmiennych

```
module adder #(parameter type ADDERTYPE = shortint)
        (input ADDERTYPE a, b,      // redefiniowalny typ
        output ADDERTYPE sum,       // redefiniowalny typ
        output logic carry);

        ADDERTYPE temp;             // zmienna lokalna redefiniowalnego typu
        ... // inne operatory
endmodule

module big_chip( ... );
        shortint a, b, r1;
        int c, d, r2;
        int unsigned e, f, r3;
        wire carry1, carry2, carry3;

        // 16-bitowy sumator ze znakiem
        adder i1 (a, b, r1, carry1);

        // 32-bitowy sumator ze znakiem
        adder #(.ADDERTYPE(int)) i2 (c, d, r2, carry2);

        // 32-bitowy sumator bez znaku
        adder #(.ADDERTYPE(int unsigned)) i3 (e, f, r3, carry3);
endmodule
```

2.12. Moduły zewnętrzne (prototypy modułów)

W języku Verilog każdy projekt jest zawsze kompilowany jako całość. Nie można skompilować projektu w częściach, a następnie połączyć ich w jeden projekt (tak jak w językach programowania). Kompilator języka Verilog musi mieć zawsze dostęp do wszystkich modułów projektu. Dlatego może on utworzyć instancję dowolnego modułu w dowolnym miejscu projektu, nawet zanim moduł zostanie zadeklarowany.

Język SystemVerilog umożliwia kompilację dużych i złożonych projektów w częściach. W tym celu w języku tym wprowadzono pojęcie *jednostki kompilacji* **$unit**. Może się zdarzyć, że w jednostce kompilacji trzeba będzie utworzyć instancję modułu, który jest zadeklarowany w innej jednostce kompilacji. W takim przypadku kompilator potrzebuje informacji o portach modułu, aby wykonać kompilację. Do tego celu służą *prototypy modułów* (*module prototypes*).

Prototyp modułu w SystemVerilog jest nazywany *modułem zewnętrznym* (*extern module*). Może on być zadeklarowany w stylu ANSI lub non-ANSI i składa się z pary słów kluczowych **extern module**, po których następują nazwa modułu, lista parametrów (dla stylu ANSI) oraz lista portów modułu. Na przykład:

```
extern module m (a,b,c,d);              // prototyp modułu m w stylu non-ANSI

extern module a                         // prototyp modułu a w stylu ANSI
#(parameter size = 8, parameter type TP = logic [7:0])
(input [size:0] a, output TP b);

module top ();
        wire [8:0] a;
        logic [7:0] b;
        wire c, d;

        m mm (.*);

        a aa (.*);

endmodule
```

Zakłada się tu, że moduły m i a są tworzone w następujący sposób:

```
module top ();
        wire [8:0] a;
        logic [7:0] b;
        wire c, d;

        m mm (a,b,c,d);

        a aa (a,b);

endmodule
```

Moduły zewnętrzne mogą być używane z notacją .* jako porty modułów. Na przykład:

```
extern module m (a,b,c,d);                              // prototyp modułu m

extern module a                                         // prototyp modułu a
#(parameter size = 8, parameter type TP = logic [7:0])
input [size:0] a, output TP b);

module m (.*);          // użycie notacji .*
```

```
        input a,b,c;
        output d;
endmodule

module a (.*);                  // użycie notacji .*
…
endmodule
```

jest równoznaczne z napisaniem

```
module m (a,b,c,d);
        input a,b,c;
        output d;
endmodule

module a
        #(parameter size = 8, parameter type TP = logic [7:0])
        (input [size:0] a, output TP b);
        …
endmodule
```

Deklaracje modułów zewnętrznych mogą pojawić się na dowolnym poziomie hierarchii tworzenia instancji, ale są widoczne tylko na tym poziomie hierarchii, na którym zostały zadeklarowane. Deklaracja modułu zewnętrznego musi być zgodna z listą portów i parametrów rzeczywistej deklaracji modułu, zgodnie z nazwami, pozycjami i odpowiadającymi im typami.

Należy pamiętać, że język SystemVerilog pozwala również na ustawienie wartości początkowych dla portów wejściowych **input**. Jednak są one używane tylko podczas symulacji i ignorowane podczas syntezy.

2.13. Wnioski

Moduł (*module*) jest podstawowym elementem konstrukcyjnym projektów SystemVerilog umożliwiającym reprezentowanie komponentów projektu na dowolnym poziomie, od najniższego do najwyższego, oraz opisywanie komponentów projektu z różnym stopniem szczegółowości. Moduły mogą tworzyć instancje innych modułów, budując w ten sposób hierarchię projektu.

Język SystemVerilog obsługuje dwa formaty definicji modułów: stary format (styl non-ANSI) i nowy format (styl ANSI). Nazwa modułu wraz z listą parametrów i listą portów tworzą nagłówek modułu, po którym następują elementy modułu składające się na jego treść (*body*).

W SystemVerilog istnieją dwa podstawowe style opisu modułów: *strukturalny* (*structural*) i *behawioralny* (*behavioral*). Możliwy jest także mieszany styl opisu modułu behawioralno-strukturalny.

Styl strukturalny opisuje instancje modułów i prymitywów oraz połączenia między nimi. Jest też używany do opisywania diagramów logicznych, strukturalnych i schematów w sposób podobny do tego, jaki występuje w edytorach graficznych.

Styl behawioralny umożliwia opisywanie projektów na poziomie opisu działania algorytmu za pomocą operatorów proceduralnych.

Język SystemVerilog pozwala na użycie następujących konstrukcji jako elementów modułu (pomijając konstrukcje języka Verilog):

- instancje interfejsu (*interface*);
- instancje programu (*program*);
- elementy zapewnień (*assertion*);
- aliasy sieciowe (*net alias*);
- deklaracje innych modułów (*module declaration*);
- deklaracje programów (*program declaration*);
- deklaracje interfejsów (*interface declaration*);
- deklaracje jednostek czasu (*timeunits declaration*);
- region generowania (*generate region*).

W języku SystemVerilog zmienne, sieci, tablice, struktury, związki i interfejsy mogą być przekazywane przez porty modułów.

Jeśli port lub typ danych sygnału podłączonego do portu są zadeklarowane jako wartości ze znakiem, to obie te wartości będą tak traktowane (jako wartości ze znakiem).

W języku SystemVerilog można nie określać kierunku portu. W takim przypadku domyślnym kierunkiem będzie **inout**. Jeśli następny port na liście portów jest określonego typu, ale nie podano kierunku, domyślnie ustawiany jest kierunek poprzedniego portu na liście.

Przed użyciem modułu jako części projektu należy utworzyć jego *instancję* (*instance*).

Istnieją dwa podstawowe sposoby łączenia sygnałów z portami instancji modułu: przez pozycję portu i przez nazwę portu.

Język SystemVerilog oferuje następujące mechanizmy niejawnego mapowania portów: notacje **.name, .*** oraz interfejsy.

Notacja **.name** pozwala na podanie tylko jednej nazwy, jeśli nazwa sygnału jest taka sama jak nazwa portu. Notacja .* określa, że wszystkie porty i sygnały o tej samej nazwie muszą być połączone dla danej instancji modułu.

Moduł zagnieżdżony (*nested modules*) definiowany jest jako moduł wewnątrz innego modułu. Jego instancję można utworzyć w module nadrzędnym, a także w dowolnym miejscu drzewa hierarchii poniżej modułu zagnieżdżonego. Moduł zagnieżdżony, podobnie jak zwykły moduł, może tworzyć instancje innych modułów.

Operator **alias** pozwala, aby dwie różne nazwy odnosiły się do tego samego elementu sieci, tzn. tworzy alias dla sieci. Użycie aliasów sieci umożliwia stosowanie notacji **.name** i **.*** dla modułów, które mają różne nazwy portów.

Język SystemVerilog pozwala na parametryzację sieci i typów zmiennych. Ta funkcja wprowadza dodatkowy poziom polimorfizmu do modelu tworzonego w tym języku.

Prototyp modułu zwany *modułem zewnętrznym* (*extern module*) w danej jednostce kompilacji dostarcza informacji o porcie dla modułu, który jest zadeklarowany w innej jednostce kompilacji.

3. Podstawowe typy danych

Typy danych w Verilogu są reprezentowane przez cztery wartości (stany) mogące wystąpić w każdym bicie: 0, 1, z oraz x. Jednak na abstrakcyjnych poziomach projektowania wygodniej jest reprezentować każdy bit danych, podobnie jak w językach programowania, za pomocą dwóch wartości (stanów): 0 i 1. Aby rozwiązać problem danych z czterema i dwoma wartościami w każdym bicie, w języku SystemVerilog wprowadzono typy danych **logic** – do reprezentowania danych z czterema stanami oraz **bit** – do reprezentowania danych z dwoma stanami.

W języku Verilog nie ma ścisłego rozróżnienia między typami danych a obiektami, z którymi są one związane. W języku SystemVerilog wprowadzono natomiast rozróżnienie między *typami danych* a *obiektami danych*. Pozwala to na bardziej precyzyjną formalizację różnych konstrukcji językowych.

Syntetyzowalne typy danych języka SystemVerilog obejmują typy **logic** i **bit** oraz wszystkie typy od nich pochodzące: dla **logic** są to **reg**, **integer** i **time**, dla **bit** są to **byte**, **shortint**, **int** i **longint**. Obiekty danych w SystemVerilog są identyfikatorami, z którymi związane są wartości i typy danych, np. zmienne, sieci, parametry, stałe itp.

Głównymi grupami obiektów danych w SystemVerilog są zmienne i sieci. Te ostatnie są zapożyczone z języka Verilog bez żadnych zmian.

W języku SystemVerilog liczba miejsc do deklarowania i inicjowania zmiennych została znacznie rozszerzona w porównaniu z językiem Verilog. Oprócz modułów, funkcji i zadań zmienne można deklarować również w blokach proceduralnych. Zwiększa to możliwości ukrywania zmiennych lokalnych i używania zmiennych o tej samej nazwie.

Wszystkie zmienne w Verilogu są deklarowane niejawnie przez określenie typu danych. W niektórych przypadkach może to być mylące. Język SystemVerilog pozwala na jawne deklarowanie zmiennych za pomocą słowa kluczowego **var**. Dodatkowo można użyć słowa kluczowego **const**, aby zadeklarować zmienną, która może być zainicjalizowana, ale nie można jej przypisać wartości. Język SystemVerilog ma również znacznie rozszerzone możliwości inicjalizacji zmiennych, ale wiele z nich nie jest syntetyzowalnych.

W języku SystemVerilog możliwe są synteza i symulacja poprzez deklarowanie zmiennych automatycznych w zadaniach i funkcjach statycznych oraz odwrotnie – zmiennych statycznych w zadaniach i funkcjach automatycznych.

Należy też pamiętać o tym, że indeksy wektorowe w języku SystemVerilog mogą mieć wartości ujemne.

W Verilogu liczby całkowite są reprezentowane przez typy o czterech stanach **integer** i **time**, a liczby rzeczywiste przez typy **real** i **realtime**. Nie ma jednak typów, które odpowiadałyby liczbom w językach programowania. Aby rozwiązać ten problem, język SystemVerilog udostępnia następujące typy liczb całkowitych z dwoma stanami: **byte**, **shortint**, **int** i **longint**, a także liczby rzeczywiste **real** i **shortreal**.

W języku SystemVerilog wprowadzono także pewne usprawnienia dotyczące postępowania ze stałymi, takimi jak wartość parametru $.

3.1. Definicje

3.1.1. Typy danych i obiekty danych

W języku SystemVerilog istnieje rozróżnienie między *typami danych* (*data types*) a *obiektami danych* (*data objects*).

Typ danych definiuje zbiór wartości oraz zbiór operacji, które można wykonywać na tych wartościach, np. **logic** i **bit**. Typy danych mogą być używane do deklarowania obiektów danych, ale także do definiowania własnych typów danych, które są tworzone na podstawie innych typów danych.

Obiekt danych to pewien identyfikator, z którym związane są wartości i typ danych, np. parametr, zmienna lub *sieć* (*net*).

3.1.2. Singularne, agregatowe, całkowite i proste typy wektorów bitowych

W języku SystemVerilog typy danych są również klasyfikowane jako *singularne* (*singular*) lub *pojedyncze* i *zagregowane* (*aggregate*) bądź *zespołowe*.

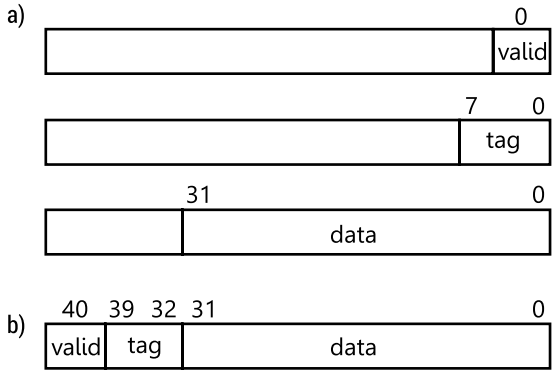
Singularny (tzn. unikalny) typ danych reprezentuje pojedynczą wartość, symbol lub *deskryptor* (*handle*). Pojedynczy typ danych może być reprezentowany jako pojedynczy wektor bitów i obsługiwany jako wektor bitów. Typy pojedyncze to wszystkie typy danych z wyjątkiem rozpakowanych (*unpacked*) struktur, unii i tablic.

Zagregowane typy danych definiują zbiór danych, które mogą być różnie reprezentowane w pamięci w zależności od używanego narzędzia projektowego (kompilatora, syntezatora albo symulatora). *Zagregowane* typy danych to wszystkie rozpakowane struktury, unie i tablice.

Na rys. 3.1 przedstawiono przykład reprezentacji struktury rozpakowanej (domyślnie) i spakowanej w pamięci.

Typ całkowity (*integral type*) to wartość, którą można przedstawić w postaci wektora za pomocą wartości bitowych (z dwoma stanami) lub logicznych (z czterema stanami). W języku SystemVerilog typ całkowity jest używany do definiowania typów danych, które mogą reprezentować podstawowy typ całkowity, tablicę spakowaną,

strukturę spakowaną, unię spakowaną, zmienną typu wyliczeniowego bądź zmienną czasową. Na przykład typami całkowitymi są **byte**, **int**, **logic** itd. Są to zawsze typy singularne, nawet jeśli można je podzielić na kilka wartości singularnych. Na przykład typ łańcuchowy jest typem pojedynczym, mimo że każdy znak łańcucha może być traktowany jako bajt.



RYS. 3.1. Przykład reprezentacji w pamięci: a) struktura rozpakowana, typ agregujący, b) struktura spakowana, typ singularny

ŹRÓDŁO: oprac. własne.

Termin *prosty typ wektora bitowego* (*simple bit vector type*) jest używany w standardzie języka SystemVerilog w odniesieniu do typów danych, które mogą bezpośrednio reprezentować jednowymiarową spakowaną tablicę bitów. *Typy całkowitoliczbowe* (**shortint**, **int**, **longint**, **byte**, **bit**, **logic**, **reg**, **integer** i **time**) to proste typy wektorów bitowych, które mają predefiniowaną szerokość. Spakowane typy strukturalne i spakowane wielowymiarowe typy tablicowe nie są typami prostych wektorów bitowych, ale każdy z nich jest równoważny pewnemu typowi prostego wektora bitowego, na który i z którego można go łatwo przekonwertować.

3.1.3. Sieci i zmienne

W języku SystemVerilog istnieją dwie główne grupy obiektów danych: *zmienne* (*variables*) i *sieci* (*nets*). Różnią się one sposobem przypisywania i przechowywania danych.

W modelach projektów sieci są używane do opisywania połączeń między komponentami projektu, a wartości sygnałów mogą być przekazywane przez te połączenia. Główną cechą sieci jest to, że nie przechowują one wartości sygnałów. Wyjątkiem jest sieć typu **trireg**, która modeluje połączenie przewodowe z kondensatorem. Wartość sygnału w sieci jest określana przez źródło sygnału zwane *sterownikiem* (*driver*). Sterowniki w sieci mogą być wyjściami prymitywów, portami wyjściowymi modułów, operatorami przypisania ciągłego **assign** itp.

W sieci może znajdować się więcej niż jeden sterownik. Wówczas jej wartość jest określana za pomocą *funkcji decyzyjnych* (*resolution function*). Funkcja decyzyjna zależy od typu sieci, np. typ **wand** odpowiada przewodowemu AND, a typ **wor** przewodowemu OR. Wartości sieci nie można zdefiniować za pomocą operatorów proceduralnych. Gdy przez port modułu przepuszczana jest wartość sieci, implikowane jest przypisanie ciągłe. Wartość sieci może być również zastąpiona przez operator **force** (niesyntetyzowany).

Zmienne są przeznaczone do stosowania w operatorach proceduralnych, które są czasem nazywane *operatorami programowania*, ponieważ można ich używać do opisywania algorytmu działania urządzenia. Należy rozróżnić zmienne w językach programowania od zmiennych w języku SystemVerilog.

Każda zmienna w języku programowania ma swoje miejsce w pamięci, w którym przechowywana jest jej wartość. W języku SystemVerilog zmienne są używane do opisywania modelu projektu za pomocą instrukcji proceduralnych. To, czy zmienna w zsyntetyzowanym schemacie odpowiada rejestrowi (dla zmiennych wektorowych) czy przerzutnikowi (dla zmiennych skalarnych), zależy tylko od kontekstu kodu projektu. Na przykład podczas kombinacyjnego opisywania sieci zmiennym nie odpowiadają ani rejestry, ani przerzutniki.

Wartość zmiennej można określić za pomocą jednego lub kilku poleceń proceduralnych. Ponieważ w przypadku przypisań wielokrotnych zmienne nie mają funkcji decyzyjnych, przypisania wielokrotne zmiennych nie są syntetyzowane. W symulacji wartość zmiennej jest określana przez ostatnie (w czasie) przypisanie do zmiennej. Wartość zmiennej może być również ustawiona przez pojedynczy operator ciągłego przypisania **assign**, jak też przez pojedynczy port modułu lub prymitywu.

W SystemVerilog zmienne mogą być zagregowanymi typami danych, tzn. *spakowanymi* (*packed*) lub *rozpakowanymi* (*unpacked*) uniami, strukturami lub tablicami.

Podczas definiowania wartości zmiennych nie należy mieszać przypisań proceduralnych i ciągłych. Na przykład następujący fragment kodu spowoduje wystąpienie błędu podczas kompilacji:

```
input sel, clk;                    // sieci
output reg [7:0] count;            // zmienna

assign count = sel ? 1'b0 : 1'b1;  // przypisanie ciągłe
always @ (posedge clk)
      count <= count + 1;          // przypisanie proceduralne: błąd!
```

Gdy zmienna jest podłączona do portów wejściowych **input** lub wyjściowych **output**, implikuje to przypisanie ciągłe. Dlatego niedopuszczalne jest przypisywanie zmiennej zadeklarowanej jako portu wejściowego, a także przypisywanie proceduralne lub ciągłe do zmiennej podłączonej do portu wyjściowego. Zmienne nie mogą być także podłączone do żadnej ze stron dwukierunkowego portu **inout**.

Synteza sieci odpowiada połączeniom przewodowym, a zmienne nie zawsze odpowiadają rejestrom lub przerzutnikom. Zależy to od kontekstu kodu projektu.

3.2. Sieci

3.2.1. Deklaracja sieci

Sieciowe typy danych są deklarowane w następujących trzech formatach:

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

net_type (*strength) data_type* **signed** [*range*] **#** (*delay*) *net_name=continuous_assignment;*

trireg (*capacitance_strength) data_type* **signed** [*range*] **#** (*decay_time*) *net_name* [*array*]*,...;*

Pierwszy format jest używany przy deklarowaniu listy sieci i tablic sieci tego samego typu. Drugi format jest używany podczas deklarowania sieci wraz z niejawnym operatorem przypisania ciągłego. Trzeci format jest używany do deklarowania sieci typu **trireg**.

W formatach tych konstrukcje *data_type*, **signed**, [*range*], *delay*, [*array*], (*strength*), *decay_time* są opcjonalne.

W powyższych formatach *net_type* jest jednym z następujących słów kluczowych: **wire**, **tri**, **wor**, **trior**, **wand**, **triand**, **trireg**, **supply0** lub **supply1**, natomiast *data_type* jest typem danych. Do zadeklarowania sieci można użyć dowolnego typu danych z czterema stanami (**logic**, **reg**, **integer**, **time**).

Słowo kluczowe **signed** oznacza, że wartość jest interpretowana jako liczba całkowita ze znakiem w kodzie uzupełnienia do dwóch. Jeśli port lub sieć połączona z portem zostaną zadeklarowane jako wartość ze znakiem, obie zostaną potraktowane jako wartości ze znakiem.

Konstrukcja [*range*] definiuje zakres bitów sygnału jako [*msb:lsb*]. Jeśli w deklaracji nie występuje określenie zakresu, domyślnie przyjmuje się, że szerokość sieci wynosi 1 bit. Parametry *msb* i *lsb* określające najbardziej znaczący i najmniej znaczący bit zakresu mogą być liczbami, stałymi, wyrażeniami lub wywołaniami funkcji stałych. Dozwolone jest określanie granic zakresów przy użyciu zarówno *indeksów rosnących* (*big-endian*), jak i *malejących* (*little-endian*). Maksymalna szerokość zakresu jest ograniczona do 216 = 65 536 bitów. Jednak wiele kompilatorów dopuszcza szerokość zakresu do 1 miliona bitów.

Konstrukcja opóźnienia *delay* definiuje wielkość opóźnienia sygnału w trakcie jego przechodzenia przez sieć i zostanie omówiona w podrozdziale 3.2.2.

Konstrukcja [*array*] jest używana do deklarowania tablic i ma następujący format:

[*first_address*:*last_address*] [*first_address*:*last_address*]...,

gdzie każdy nawias kwadratowy definiuje jeden wymiar tablicy.

Tablica może mieć dowolną liczbę wymiarów. Każdy wymiar tablicy jest określony przez zakres adresów. Elementy *first_address* i *last_address* mogą być liczbami, stałymi, wyrażeniami lub wywołaniami funkcji stałych. Wartości adresów można ustawić w porządku rosnącym lub malejącym. Maksymalny zakres jednego wymiaru jest ograniczony do 224 = 16 777 216, ale wiele kompilatorów nie ogranicza maksymalnej wartości tego zakresu.

W drugim formacie konstrukcja (*strength*) definiuje moc logiczną źródła sygnału, w trzecim zaś konstrukcja (*capacitance_strength*) definiuje moc pojemnościową sieci. Oba konstrukty mają następujący format:

(*strength0*, *strength1*)

gdzie *strength0* i *strength1* są liczbami od 0 do 7 określającymi moc logiczną sygnału odpowiednio przy wartości sygnału 0 i 1 (moc logiczna sygnału omówiona jest w podrozdziale 3.2.4).

Konstrukcja *decay_time* określa czas, przez jaki sieć **trireg** będzie pamiętać (utrzymywać) wartość sygnału po wyłączeniu wszystkich jego źródeł, tzn. przełączeniu ich wyjść w stan wysokiej impedancji. Po upływie tego czasu sygnał przyjmie wartość x. Konstrukcja *decay_time* ma następujący format:

(*rise_delay*, *fall_delay*, *decay_time*).

gdzie: *rise_delay* to opóźnienie dodatniego zbocza sygnału (0-1); *fall_delay* to opóźnienie ujemnego zbocza sygnału (1-0); *decay_time* to czas zanikania sygnału. Parametry *rise_delay*, *fall_delay* i *decay_time* są dodatnimi liczbami całkowitymi wyrażającymi przedział czasowy za pomocą liczby jednostek czasu (wartość jednostki czasu określa się za pomocą dyrektywy systemowej `**timescale**).

Ponadto bezpośrednio po słowie kluczowym net_type może wystąpić słowo kluczowe **vectored** lub **scalared**, wskazujące na wektorowy bądź skalarny charakter sygnału. Jeśli typ danych jest zdefiniowany jako wektor, kompilator umożliwi dostęp do bitów wektora.

Język SystemVerilog ma ograniczenia co do użycia słowa kluczowego **reg** w deklaracji sieci: słowo kluczowe **reg** nie może występować bezpośrednio po słowie kluczowym definiującym typ sieci. Innymi słowy, między słowem kluczowym typu sieciowego a **reg** muszą występować jakieś elementy leksykalne. Na przykład:

```
output wire reg result;          // niedopuszczalne
tri reg r;                       // niedopuszczalne
```

3.2.2. Opóźnienia

Konstrukcja opóźnienia *delay* określa wielkość opóźnienia sygnału w trakcie jego przechodzenia przez sieć. Wartość opóźnienia można ustawić jako jedną, dwie lub trzy wartości:

```
# del                    // określenie opóźnienia za pomocą jednej wartości
# (del10, del01)         // określenie opóźnienia za pomocą dwóch wartości
#  (del10, del01, del2)  // określenie opóźnienia za pomocą trzech wartości
```

gdzie: *del* to wartość opóźnienia dla wszystkich zmian sygnału w sieci; *del10* to wartość opóźnienia, gdy sygnał zmienia się z 1 na 0; *del01* to wartość opóźnienia, gdy sygnał zmienia się z 0 na 1; *del2* to wartość opóźnienia, gdy sygnał przechodzi w stan wysokiej impedancji.

Każda z liczb *del*, *del10*, *del01* i *del2* może być przedstawiona jako pojedyncza liczba lub w następującym formacie:

min : typ : max

gdzie: *min* to wartość minimalna; *max* to wartość maksymalna; *typ* to typowa wartość opóźnienia.

Przykłady definicji opóźnień:

#3 – opóźnienie o 3 jednostki czasu dla wszystkich przełączanych sygnałów;

#2:3:4 – minimalne opóźnienie 2, maksymalne 4 i typowe 3 jednostki czasu dla wszystkich przejść sygnału;

#(5,6) – opóźnienie sygnału wynosi 5 jednostek czasu przy przełączaniu z 1 na 0 i 6 jednostek czasu przy przełączaniu z 0 na 1;

#(2:3:4, 3:4:5, 4:5:6) – opóźnienie sygnału przy przełączaniu z 1 na 0 wynosi 2:3:4, przy przełączaniu z 0 na 1 wynosi 3:4:5, a przy przełączaniu w stan wysokiej impedancji wynosi 4:5:6.

3.2.3. Typy sieciowe

W języku SystemVerilog istnieją dwa rodzaje sieci: *wbudowane* (*built-in*) i *definiowane przez użytkownika* (*user-defined*). W tym rozdziale omówiono tylko sieci wbudowane. Natomiast sieci zdefiniowane przez użytkownika, jako typy zdefiniowane przez użytkownika, zostały omówione w rozdziale 4.

Typy sieciowe są używane do reprezentowania fizycznych połączeń między obiektami strukturalnymi, np. między prymitywami i instancjami modułów. Sieci to abstrakcyjne modele połączeń przewodowych. Różne typy sieci odzwierciedlają różne zasady fizyczne, na których opierają się fizyczne połączenia przewodowe. W języku SystemVerilog dostępne są następujące wbudowane typy sieci: **wire**, **tri**, **wand**, **triand**, **wor**, **trior**, **uwire**, **tri0**, **tri1**, **trireg**, **supply0** i **supply1**.

Sieci nie mogą przechowywać wartości, z wyjątkiem sieci typu **trireg**. Jest to zgodne z zasadami fizyki: połączenie przewodowe nie przechowuje wartości sygnału, o ile nie jest ono podłączone do źródła sygnału (sterownika). Sieć **trireg** symuluje przewodnik połączony z kondensatorem, który umożliwia przechowywanie wartości sygnału (ładunku) przez pewien czas.

Wartość sygnału w sieci jest określana przez jego źródło (sterownik). Sterowniki mogą być portami wyjściowymi modułu, wyjściami prymitywów, wyjściami instancji modułu, operatorami przypisania ciągłego **assign** itp. Jeśli do sieci nie jest podłączony żaden sterownik, mówi się, że sieć ma logiczną wartość *z*, zwaną stanem wysokiej impedancji lub stanem trzecim. Wyjątkiem jest sieć typu **trireg**, która symuluje pojemnościowe podtrzymywanie ustalonej wartości sygnału.

W języku SystemVerilog sieci **wire** i **tri**, **wor** i **trior**, **wand** i **triand** są sobie równoważne. W kodach projektów używa się sieci **tri**, **trior** i **triand**, aby podkreślić, że obwód ma wiele źródeł. Innymi słowy, typy sieci mogą mieć jeden sterownik lub więcej. Jednak **wire** i jego pochodne (**wand** i **wor**) są używane głównie w obwodach z jednym źródłem, natomiast **tri** i jego pochodne (**triand** i **trior**) są używane do wskazania, że obwód może mieć wiele sterowników.

Typ **uwire** jest używany do oznaczania sieci, które zawsze muszą mieć tylko jeden sterownik. Jeśli dla sieci **uwire** przypadkowo określono więcej niż jeden sterownik, kompilator wygeneruje błąd.

3.2.4. Moc logiczna sygnałów sieci

Wartości sygnałów w sieciach mają osiem poziomów *mocy logicznej* (*logic strength*) [tabela 3.1], która jest określana przez *moc* (*strength*) *sterownika*.

Im wyższa liczba jest przypisana poziomowi mocy, tym większa jest moc logiczna sygnału. Poziomy **supply**, **strong**, **pull** i **weak** służą do określenia mocy logicznej sygnałów na wyjściach sterownika, a poziomy **large**, **medium** i **small** określają moc pojemnościową sygnałów w sieciach **trireg**. Stan wysokiej impedancji (*z*) ma zerową moc logiczną.

TABELA 3.1. Poziomy mocy źródła sygnału

Poziom mocy	Nazwa poziomu mocy	Słowa kluczowe	Mnemonik oznaczenia
7	*supply drive* (sterowanie energią)	**supply0** **supply1**	Su0 Su1
6	*strong drive* (silne sterowanie)	**strong0** **strong1**	St0 St1
5	*pull drive* (sterowanie podciąganiem)	**pull0** **pull1**	Pu0 Pu1
4	*large capacitive* (duża pojemność)	**large**	La0 La1
3	*weak drive* (słabe zarządzanie)	**weak0** **weak1**	We0 We1
2	*medium capacitive* (średnia pojemność)	**medium**	Me0 Me1
1	*small capacitive* (mała pojemność)	**small**	Sm0 Sm1
0	*high impedance* (wysoka impedancja)	**highz0** **highz1**	HiZ0 HiZ1

ŹRÓDŁO: oprac. własne.

Przykłady definicji sieciowych:

```
wire a, b, c;                    // trzy 1-bitowe sieci skalarne

tri1 [7:0] date_bus;             // sieć 8-bitowa, w trzecim stanie
                                 // jest podciągnięta do poziomu wysokiego

wire signed [1:8] result;        // sieć 8-bitowa, interpretowana jako
                                 // liczba całkowita ze znakiem

wire [7:0] Q[0:15][0:256];       // dwuwymiarowa tablica 8-bitowych sieci

wire  # (2.4, 1.8) carry;        /* sieć z określonymi opóźnieniami narastania (2.4)
                                 i opadania (1,8) poziomu sygnału */

wire [0:15] (strong1, pull0) sum = a + b;   /* sieć 16-bitowa
                                               z zadaną mocą źródła logicznego
                                               i niejawny operator przypisania ciągłego */

trireg (small) # (0,0,35) d;     /* sieć o małej mocy logicznej
                                 i czasie podtrzymania 35 jednostek */
```

```
trireg (large) logic # (0, 0, 0) cap1;  /* deklaracja sieci cap1 typu trireg
                                           z typem danych logic,
                                           z dużą pojemnością magazynowania ładunku
                                           i zerowym opóźnieniem sygnału */
```

W języku SystemVerilog sieci są domyślnie deklarowane za pomocą typu danych **logic**, np.:

```
wire w;              // równoważne wire logic w;

wire [15:0] y;       // równoważne wire logic [15:0] y;
```

3.2.5. Wartości sygnałów sieciowych

Gdy sieć ma wiele źródeł sygnału, jej wartość zależy od wartości na wyjściach sterowników oraz od typu sieci. Przypomnijmy, że typy sieciowe (w przeciwieństwie do zmiennych) mają funkcje pozwalające rozwiązywać konflikty wielu sterowników. W tabeli 3.2 przedstawiono wartości dla różnych typów sieci w przypadku dwóch źródeł sygnału o tej samej mocy logicznej.

TABELA 3.2. Wartości sygnałów w różnych sieciach dla dwóch źródeł sygnału

Wartości źródłowe		Wartość sygnału sieciowego				
a	**b**	**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
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

ŹRÓDŁO: oprac. własne.

Typy **wire** i **tri** są używane do symulowania tradycyjnego połączenia przewodowego. Konflikt wartości logicznych w przypadku kilku źródeł o tej samej mocy logicznej powoduje, że wartość jest nieznana (x).

Typy **wor** i **trior** umożliwiają symulację obwodów z *przewodowym OR* (*wired or*), co odpowiada układom z *otwartym kolektorem* (*open collector*, technologia bipolarna) lub *otwartym drenem* (*open drain*, technologia MOS – *metal oxide semiconductor*). Rozwiązywanie konfliktów między wieloma źródłami: jeśli jedno ze źródeł ma wartość logiczną 1, to sieć ma wartość logiczną 1.

Podobnie typy **wand** i **triand** symulują *przewodowe AND* (*wired and*), tzn. wartość sygnału jest określana za pomocą reguł *układu logicznego ECL* (*emitter-coupled logic*). Rozwiązanie konfliktu wielu źródeł: jeśli jedno ze źródeł ma wartość logiczną 0, to sieć ma wartość logiczną 0.

Typy **tri0** i **tri1** symulują sieci z elementami podciągającymi *pulldown* i *pullup*. Jeśli żaden sterownik nie steruje siecią **tri0**, wartością logiczną sieci jest 0 z mocą **pull**. Jeśli żaden sterownik nie steruje siecią **tri1**, wartością logiczną układu jest 1 z mocą **pull**.

Zakłada się, że w sieci typu **trireg** zachowana jest ostatnia wartość logiczna. Ten typ jest używany do symulowania magazynowania ładunku. Sieć **trireg** może znajdować się w jednym z dwóch stanów: *sterowanym* (*driven state*) lub *pojemnościowym* (*capacitive state*). W stanie sterowanym, gdy co najmniej jeden sterownik sieci **trireg** ma wartość logiczną 1, 0 lub x, wartość ta odnosi się do całej sieci **trireg** (tzn. sieć **trireg** ma wartość 1, 0 lub x). W stanie pojemnościowym, gdy wszystkie sterowniki w sieci **trireg** znajdują się w stanie wysokiej impedancji *z*, obwód **trireg** zachowuje ostatnią wartość. Nie powoduje to propagacji wartości *z* ze sterownika w element sieci **trireg**.

Typy **supply0** i **supply1** są używane do modelowania zasilaczy w obwodzie. Odpowiadają im stałe 1'b0 i 1'b1 w kodzie projektu.

3.2.6. Cechy charakterystyczne syntezy

Zastosowanie różnych typów sieci pokazano w przykładzie 3.1.

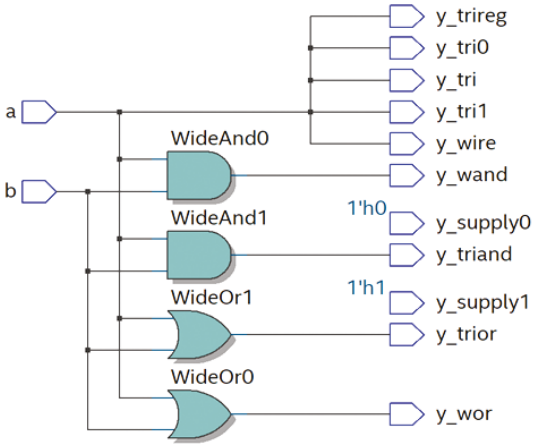
Przykład 3.1. Użycie sieciowych typów danych

```
module nets
        (input a,b,
        output wire y_wire,
        output tri y_tri,
        output wand y_wand,
        output triand y_triand,
        output wor y_wor,
        output trior y_trior,
        // output uwire y_uwire,       nie jest syntetyzowany
        output tri0 y_tri0,
        output tri1 y_tri1,
        output trireg y_trireg,
        output supply0 y_supply0,
        output supply1 y_supply1);
```

```
assign y_wire = a;
// assign y_wire = b;          typ wire nie może być sterowany przez inny sygnał
assign y_tri = a;
// assign y_tri = b;           typ tri nie może być sterowany przez inny sygnał
assign y_wand = a;
assign y_wand = b;
assign y_triand = a;
assign y_triand = b;
assign y_wor = a;
assign y_wor = b;
assign y_trior = a;
assign y_trior = b;
assign y_tri0 = a;
// assign y_tri0 = b;          typ tri0 nie może być sterowany przez inny sygnał
assign y_tri1 = a;
// assign y_tri1 = b;          typ tri1 nie może być sterowany przez inny sygnał
assign y_trireg = a;
// assign y_trireg = b;        typ trireg nie może być sterowany przez inny sygnał
// assign y_supply0 = a;       typ supply0 nie może być sterowany przez inny sygnał
// assign y_supply0 = b;       typ supply0 nie może być sterowany przez inny sygnał
// assign y_supply1 = a;       typ supply1 nie może być sterowany przez inny sygnał
// assign y_supply1 = b;       typ supply1 nie może być sterowany przez inny sygnał
```

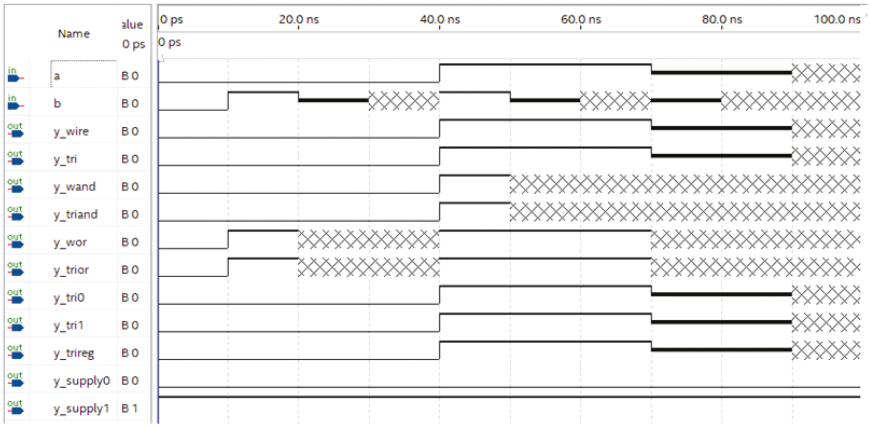
endmodule

Wynik syntezy projektu sieci pokazano na rys. 3.2, a wyniki symulacji są przedstawione na rys. 3.3.



RYS. 3.2. Wynik syntezy projektu z przykładu 3.1

ŹRÓDŁO: oprac. własne.



RYS. 3.3. Wyniki symulacji projektu z przykładu 3.1

ŹRÓDŁO: oprac. własne.

Z analizy przykładu 3.1 wynika, że przy syntezie za pomocą Quartusa typy **wire**, **tri**, **tri0**, **tri1** i **trireg** dopuszczają tylko jeden sterownik, a typy **supply0** i **supply1** mogą w ogóle nie mieć sterowników, sieci tego typu odpowiadają stałym logicznym 1'b0 i 1'b1. System Quartus również nie obsługuje typu **uwire**, ponieważ jest to typ **wire**.

W tabeli 3.3 przedstawiono wartości sygnałów w różnych sieciach podczas syntezy w systemie Quartus.

TABELA 3.3. Wartości sygnałów w różnych sieciach dla dwóch źródeł w syntezie Quartus

a	b	wire tri	wand triand	wor trior	tri0 tri1 trireg	supply0	supply1
0	0	0	0	0	0	0	1
0	1	0	0	1	0	0	1
0	z	0	0	x	0	0	1
0	x	0	0	x	0	0	1
1	1	1	1	1	1	0	1
1	z	1	x	1	1	0	1
1	x	1	x	1	1	0	1
z	z	z	x	x	z	0	1
z	x	z	x	x	z	0	1
x	x	x	x	x	x	0	1

ŹRÓDŁO: oprac. własne.

Czytelnik proszony jest o porównanie tabel 3.2 i 3.3 w celu znalezienia i wyjaśnienia różnic.

3.3. Zmienne

3.3.1. Jawne i niejawne deklaracje zmiennych

Przypomnijmy, że każdy zadeklarowany element w języku SystemVerilog ma typ obiektu oraz typ danych, które ten obiekt mogą przyjąć. Typami obiektów są sieci i zmienne. Słowa kluczowe typów sieci (**wire**, ...) są używane do jej oznaczania. W SystemVerilog do jawnego oznaczania zmiennych można używać słowa kluczowego **var**, które jest skrótem od słowa *variable*. Typ danych dla zmiennych w SystemVerilog jest definiowany przez dwa słowa kluczowe: **logic** i **bit**. Dane o czterech stanach są reprezentowane przez słowo kluczowe **logic**, a dane o dwóch stanach są reprezentowane przez słowo kluczowe **bit**. Na przykład:

```
var logic [7:0] a;      // jawna zmienna 8-bitowa z czterema stanami
var bit [15:0] b;       // jawna zmienna 16-bitowa z dwoma stanami
```

W tym przykładzie zadeklarowano *zmienne jawne* (*explicit variable*). Słowo kluczowe **var** w deklaracji zmiennej może być pominięte, wówczas mówimy o deklaracji *zmiennej niejawnej* (*implicit variable*). Na przykład:

```
logic [7:0] a;          // niejawna zmienna 8-bitowa o czterech stanach
bit [15:0] b;           // niejawna zmienna 16-bitowa z dwoma stanami
```

Oprócz typu **logic** język SystemVerilog ma trzy syntetyzowalne typy zmiennych **reg**, **integer** i **time** (**time** jest taki sam jak **integer**) do reprezentowania zmiennych o czterech stanach (0, 1, x i z) oraz, oprócz typu **bit**, cztery syntetyzowalne typy zmiennych **byte**, **short**, **int** i **longint** do reprezentowania zmiennych o dwóch stanach (0 i 1). Typy danych **integer**, **time**, **byte**, **short**, **int** i **longint** definiują zmienne o określonych rozmiarach wektora. Na przykład:

```
var integer j;          // zmienna 32-bitowa z czterema stanami
var byte c;             // zmienna 8-bitowa z dwoma stanami
var short d;            // zmienna 16-bitowa z dwoma stanami
var int e;              // zmienna 32-bitowa z dwoma stanami
var longint f;          // zmienna 64-bitowa z dwoma stanami
```

Słowo kluczowe **var** nie ma wpływu na syntezę ani modelowanie, służy głównie do dokumentowania kodu projektu. Jeśli nie zostanie określony typ danych, domyślnie przyjmuje się typ **logic** z czterema stanami, np.:

```
var [10:0] d;           // zmienna 11-bitowa z czterema stanami
```

3.3.2. Deklaracja zmiennych

Formalna składnia deklaracji zmiennych ma następujący format:

[const][**var**][*lifetime*][*object_type*][*signing*][*data_type*][*range*] *name_list*;
[const][**var**][*lifetime*][*object_type*][*signing*][*data_type*][*range*] *name = initial_value*, ...;
[const][**var**][*lifetime*][*object_type*][*signing*][*data_type*][*range*] *name* [*array*], ...;

Pierwszy format jest używany do deklarowania listy zmiennych tego samego typu, drugi – przy deklarowaniu zmiennych wraz z ich wartościami początkowymi, trzeci zaś do deklarowania tablic zmiennych. Konstrukcje w nawiasach kwadratowych są opcjonalne.

Jak wspomniano wcześniej, słowo kluczowe **var** może być pominięte. Konstrukcja **const** jest używana dla zmiennych niemodyfikowalnych.

Czas życia zmiennej *lifetime* może przyjąć wartość **static** lub **automatic** i definiuje zmienną jako statyczną bądź automatyczną. Domyślnie wszystkie zmienne są deklarowane jako statyczne, z wyjątkiem zmiennych w funkcjach i zadaniach automatycznych.

Konstrukcja *signing* może przyjmować wartość **signed** lub **unsigned** i definiuje zmienną ze znakiem lub bez znaku.

Konstrukcja *data_type* może przyjmować wartość **bit** dla zmiennych o dwóch stanach i wartość **logic** dla zmiennych o czterech stanach (zamiast **logic** można użyć **reg**).

Konstrukcje *range* i *array* mają te same wartości (patrz podrozdział 3.2.1).

Konstrukcja *object_type* określa typ obiektu i może mieć wartości **byte**, **shortint**, **int**, **longint**, **integer** i **time** dla liczb całkowitych oraz **shortreal**, **real** i **realtime** dla liczb rzeczywistych. Ponadto typ obiektu można określić jako:

- **enum** – dla typów wyliczeniowych;
- **string** – dla ciągów znaków;
- **virtual interface** – dla interfejsów wirtualnych (niesyntetyzowane);
- **event** – dla zdarzeń (niesyntetyzowane).

Ogólnie rzecz biorąc, w SystemVerilog pojęcie zmiennej jest jeszcze bardziej ogólne. Na przykład zmienną może być struktura, unia, tablica, klasa lub interfejs.

Należy zauważyć, że nie wszystkie z powyższych typów danych są syntetyzowalne. Aby dowiedzieć się więcej o typach danych, które można syntetyzować, należy zapoznać się z dokumentacją używanego kompilatora.

3.3.3. Inicjalizacja zmiennych

Drugi format w podrozdziale 3.3.2 deklaruje zmienną, przy czym jest ona inicjalizowana pewną wartością. Określenie wartości początkowej podczas deklarowania zmiennej jest nazywane *inicjalizacją zmiennej w wierszu* (*in-line variable initialization*). Inicjalizacja zmiennej w deklaracji jest równoważna z inicjalizacją zmiennej w bloku **initial**. Na przykład:

```
wire c = 1'b1;
logic [7:0] m = 8'h0A;
```

jest równoważne

```
wire c;
logic [7:0] m;

initial begin
        c = 1'b1;
        m = 8'h0A;
end
```

W SystemVerilog wartości początkowe zmiennych nie są ograniczone do prostych stałych. Wartości początkowe mogą obejmować:

- wyrażenia w *czasie rzeczywistym* (*run-time*);
- dynamiczny przydział pamięci;
- statyczny *deskryptor klasy* (*class handle*) lub *skrzynkę pocztową* (*mailbox*), przez wywołanie metody **new**;
- wartości losowe generowane przez zadanie systemowe **$urandom**.

Wszystkie wymienione metody inicjalizacji zmiennych są związane z symulacją i nie są dostępne przy syntezie.

Domyślnie wszystkie zmienne o typie danych z czterema stanami (**logic**) są inicjowane wartością x, a z dwoma stanami (**bit**) wartością 0. Zmienne typów **real** i **shortreal** są inicjowane wartością 0.0, a typu **string** są inicjowane pustym łańcuchem. Z kolei zmienne niesyntetyzowalnych typów **class**, **interface class**, **chandle** i **virtual interface** są inicjalizowane wartością **null**, a typu **event** – wartością „new event".

3.3.4. Zmienne z czterema i dwoma stanami

Podczas deklarowania zmiennych w języku SystemVerilog używane są dwa typy danych: **logic** i **bit**. Ten pierwszy definiuje typ danych o czterech stanach, tzn. jeden bit zmiennej typu danych **logic** może przechowywać wartości logiczne 0, 1, x oraz z. Typ danych **logic** można również zdefiniować jako wektor o dowolnej szerokości. Przypomnijmy, że wszystkie typy sieciowe domyślnie przyjmują typ danych z czterema stanami. Dlatego dopuszczalne jest łączenie dowolnego typu sieciowego ze słowem kluczowym **logic**.

W języku Verilog do deklarowania zmiennych używany jest typ **reg**. Słowo kluczowe **reg** często wprowadza w błąd początkujących użytkowników, którzy myślą, że zmienna typu **reg** podczas syntezy zostanie zaimplementowana w rejestrze (**reg** – register). Tak jednak nie jest, a konkretna implementacja zmiennych jest całkowicie zależna od kontekstu kodu projektu.

W języku SystemVerilog słowo **logic** służy do zastąpienia niefortunnego słowa kluczowego **reg**. Typ **logic** jest semantycznie równoważny typowi **reg** języka Verilog i ma go zastąpić, aby uniknąć kojarzenia typu **reg** z rejestrem podczas implementacji sprzętowej. Innymi słowy, wszędzie tam, gdzie w Verilogu stosuje się typ **reg**, w SystemVerilog należy stosować typ **logic**. Jedyna różnica między typem **logic** a typem **reg** polega na tym, że słowa kluczowego **reg** nie można łączyć z typami sieciowymi, natomiast słowo kluczowe **logic** można łączyć z typami sieciowymi. Na przykład:

```
wire reg [7:0] d;          // niedozwolone
wire logic [7:0] d;        // dozwolone
```

Zmienne o typie danych **logic** są idealne do opisywania projektów na poziomie przesłań międzyrejestrowych (RTL), czyli na poziomie sprzętowym. Dlatego zmienne **logic** są zawsze syntetyzowalne.

Typ danych **bit** został dodany do języka SystemVerilog specjalnie w celu modelowania projektów na bardziej abstrakcyjnych poziomach niż RTL, takich jak *poziom systemowy* (*system level*), *poziom abstrakcji* (*abstract level*) i *poziom transakcji* (*transaction level*). Do języka SystemVerilog dodano następujące typy z dwoma stanami:

- **bit** – 1-bitowa liczba całkowita o dwóch stanach;
- **byte** – 8-bitowa liczba całkowita, podobna do char w języku C;
- **shortint** – 16-bitowa liczba całkowita, podobna do short w języku C;
- **int** – 32-bitowa liczba całkowita, podobna do int w języku C;
- **longint** – 64-bitowa liczba całkowita, podobna do long w języku C.

Typy z dwoma stanami są przeznaczone głównie do interakcji modeli języka SystemVerilog z modelami wyższego poziomu napisanymi w językach C/C++ przy użyciu *bezpośredniego interfejsu programistycznego* (*direct programming interface* – DRI).

Oczywiście możliwe jest przypisanie wartości zmiennych o dwóch stanach do zmiennych o czterech stanach, podobnie jak przypisanie wartości z czterema stanami do zmiennych z dwoma stanami. W tym przypadku wartości x lub z w typie czterostanowym zostaną zamienione na logiczne 0 na odpowiednim miejscu binarnym zmiennej dwustanowej.

Z punktu widzenia składni zmienna typu **bit** może być użyta w dowolnym miejscu zamiast zmiennej typu **reg** lub **logic**. Jedyna różnica polega na tym, że w każdym bicie zmienna typu **bit** może przechowywać tylko dwie wartości: 0 i 1.

Zmienne typu **bit** można deklarować w dokładnie taki sam sposób jak zmienne typów **reg** lub **logic**. Na przykład:

```
bit resetn;                // zmienna 1-bitowa dwuwartościowa
bit [15:0] d;              // 16-bitowy wektor dwuwartościowy
bit [7:0] mem [0:255];     // tablica pamięci 256 × 8 elementów dwuwartościowych
```

Słowo kluczowe **bit** nie jest w rzeczywistości typem zmiennej, lecz typem danych, który wskazuje, że zmienna może mieć wartość o dwóch stanach. Gdy słowo kluczowe **bit** jest używane oddzielnie, domyślnie implikowana jest zmienna. Zmienną o dwóch stanach można również jawnie zadeklarować za pomocą pary słów kluczowych **var bit**, np.:

```
var bit [31:0] addr;          // zmienna 32-bitowa z dwoma stanami
```

Najbardziej efektywne jest używanie zmiennych o dwóch stanach jako zmiennych pętli w instrukcjach **for**. Zmienne pętli nie wymagają wartości z czterema stanami. Dlatego wszędzie tam, gdzie dla zmiennej pętli używany jest typ **integer**, w języku SystemVerilog należy stosować typ **int**.

Przy symulacji zmienne o dwóch stanach zajmują mniej miejsca w pamięci niż zmienne o czterech stanach. Obliczanie wartości logicznych każdego bitu w wyrażeniach ze zmiennymi dwustanowymi jest również znacznie szybsze. Można się spodziewać, że zastosowanie zmiennych o dwóch stanach będzie też bardziej efektywne w syntezie w porównaniu ze zmiennymi o czterech stanach.

Wykorzystanie zmiennych dwustanowych do syntezy wiąże się jednak z pewnym ryzykiem. Zmienne o czterech stanach (**reg**, **logic**, **integer**) domyślnie rozpoczynają symulację z wartością logiczną x we wszystkich bitach. Zmienne te pozostają w nieznanym stanie x, dopóki nie zostanie im przypisana jakaś wartość. Dlatego wartość x można wykorzystać do wykrywania błędów w projekcie, gdy do zmiennej nie jest przypisana żadna wartość.

Wszystkie typy danych z dwoma stanami rozpoczynają symulację z wartością logiczną 0, nie mogą więc przechowywać wartości logicznej x. Dlatego w przypadku zmiennych o dwóch stanach wartość x nie może być wykorzystywana do wykrywania błędów.

Wnioski. Do opisu projektów syntetyzowanych zaleca się stosowanie typów z czterema stanami.

Należy pamiętać, że w SystemVerilog zmienne mogą nie mieć żadnych wartości. Są one oznaczane słowem kluczowym **void**. Typ **void** jest używany do definiowania funkcji, które nie zwracają wartości, oraz w *uniach oznaczonych* (*tagged unions*).

3.3.5. Zmienne statyczne i automatyczne

Pojęcie zmiennych statycznych i automatycznych jest związane głównie z symulacją projektu. W języku SystemVerilog są one równoważne podobnym zmiennym języków programowania. Można założyć, że przy syntezie wszystkie zmienne powinny być statyczne. Nie jest to jednak do końca prawdą i w niektórych przypadkach język SystemVerilog pozwala na stosowanie automatycznych zmiennych przy syntezie.

W SystemVerilog do deklarowania zmiennych statycznych i automatycznych (dynamicznych) używa się słów kluczowych **static** i **automatic**. Jeśli założymy, że wszystkie

zmienne są domyślnie statyczne, to po co wprowadzać do języka słowo kluczowe **static?** Spróbujmy dotrzeć do sedna sprawy.

W Verilogu-1995 wszystkie zmienne są wyłącznie statyczne. W standardzie języka Verilog-2001 dodano możliwość definiowania zadań i funkcji automatycznych, więc wszystkie zmienne w takich zadaniach i funkcjach są również automatyczne. Przy każdym wywołaniu zadania lub funkcji automatycznej w pamięci komputera (wykonującego symulację) jest alokowane miejsce na zmienne automatyczne, które jest zwalniane po zakończeniu zadania bądź funkcji.

Pojawia się pytanie: jak to działa przy syntezie? Odpowiedź: synteza zadań i funkcji automatycznych jest możliwa, jeśli liczba ich wywołań jest znana z wyprzedzeniem.

Poniższy przykład dotyczy deklaracji i wywołania funkcji, która sumuje elementy tablicy w zakresie określonym przez indeksy *lo* i *hi*. W tym przykładzie zmienna *mid* jest automatyczna, ponieważ funkcja *b_add* została zadeklarowana jako automatyczna.

Przykład 3.2. Deklaracja i wywoływanie funkcji rekurencyjnej do obliczania sumy elementów w tablicy

```
module balance_adder #(parameter N=8, LO_INDEX=0, HI_INDEX=N-1)
        (input int array [N-1:0],
        output int balance);

        int lo_index = LO_INDEX;                    // zmienna statyczna
        int hi_index = HI_INDEX;                    // zmienna statyczna

        function automatic int b_add (int lo, hi);  // funkcja automatyczna
                int mid = (lo + hi + 1) >> 1;       // zmienna automatyczna

                if (lo + 1 != hi)
                        return(b_add(lo,(mid-1)) + b_add(mid,hi));
                else
                        return(array[lo] + array[hi]);
        endfunction

        always
                if (lo_index >= 0 & hi_index < N)
                        balance = b_add (lo_index, hi_index);
                else
                        balance = 0;
endmodule
```

Należy pamiętać, że syntezator Quartus nie obsługuje rekurencyjnych wywołań funkcji.

Na poziomie modułu wszystkie zmienne są statyczne, a zmienne automatyczne lub statyczne mogą być deklarowane tylko w zadaniach, funkcjach, blokach **begin... end** bądź **fork...join**. W poniższym przykładzie przedstawiono deklarację zmiennych automatycznych *count* i *temp* w funkcji statycznej *count_ones*.

Przykład 3.3. Deklaracja zmiennych automatycznych w funkcji statycznej

```
function int count_ones (input [31:0] data);        // funkcja statyczna
        automatic logic [31:0] count = 0;           // zmienna automatyczna
        automatic logic [31:0] temp = data;         // zmienna automatyczna

        for (int i=0; i<=32; i++)
        begin
                if (temp[0]) count++;
                temp >>= 1;
        end
        return count;
endfunction
```

W następnym podrozdziale wyjaśnimy, dlaczego zmienne automatyczne są deklarowane w funkcji statycznej.

Czasami wymagana jest zmienna statyczna w automatycznym zadaniu lub funkcji. W takich przypadkach używa się słowa kluczowego **static**.

W poniższym kodzie zadanie automatyczne *check_result* sprawdza, czy w wyniku nie wystąpił błąd. Jeśli zostanie wykryty błąd, wartość zmiennej *error_count* jest zwiększana. Gdyby *error_count* był automatyczny (jak każda zmienna w zadaniu automatycznym), byłby tworzony na nowo przy każdym wywołaniu zadania i przechowywałby licznik błędów tylko dla tego wywołania. Ponieważ jednak *error_count* jest zadeklarowana jako zmienna statyczna, zachowuje ona swoją wartość od jednego wywołania zadania do następnego i przechowuje sumę wszystkich błędów.

Przykład 3.4. Deklaracja zmiennej statycznej w zadaniu automatycznym

```
typedef struct packed {...} packet_t;

task automatic check_result
        (input packet_t sent, received);
        output int total_errors);

        static int error_count;         // deklaracja zmiennej statycznej
                                        // w zadaniu automatycznym
        ...
        if (sent ! == received) error_count++;
        total_errors = error_count;
endtask
```

3.3.6. Inicjalizacja zmiennych statycznych i automatycznych

Język Verilog umożliwia deklarowanie zmiennych, które mają być inicjalizowane tylko na poziomie modułu. Deklaracje zmiennych na poziomie zadań, funkcji, bloków **begin...end** lub **fork...join** nie mogą zawierać wartości początkowych.

Język SystemVerilog rozszerza możliwości języka Verilog, umożliwiając nadawanie wartości początkowych zmiennym zadeklarowanym w funkcjach i zadaniach. Ustawianie wartości początkowych zmiennym w ramach funkcji i zadań jest nazywane *wewnętrzną inicjalizacją zmiennych* (*internal initialization of variables*).

Zmienne statyczne funkcji i zadań są inicjalizowane tylko raz przed rozpoczęciem symulacji. Wywołanie funkcji lub zadania nie powoduje ponownej inicjalizacji zmiennych statycznych. Jednak taka inicjalizacja może nie być obsługiwana przez niektóre syntezatory. W poniższym kodzie funkcja *count_ones* nie będzie działać poprawnie.

Przykład 3.5. Nieprawidłowa inicjalizacja zmiennych statycznych

```
function int count_ones (input [31:0] data);
        logic [31:0] count = 0;          // jest inicjalizowana jeden raz
        logic [31:0] temp = data;        // jest inicjalizowana jeden raz

        for (int i=0; i<=32; i++)
        begin
                if (temp[0]) count++;
                temp >>= 1;
        end
        return(count);
endfunction
```

W tym przykładzie zmienna *count* będzie miała wartość początkową 0 przy pierwszym wywołaniu funkcji *count_ones*. Jednak przy kolejnych wywołaniach zmienna *count* nie zostanie ponownie zainicjowana, lecz zachowa swoją wartość z poprzedniego wywołania. Ponadto przy pierwszym wywołaniu funkcji zmienna *temp* zostanie zainicjowana wartością 0, a nie wartością danych z portu wejściowego *data*. Dzieje się tak dlatego, że wewnętrzna inicjalizacja zmiennych statycznych jest wykonywana przed zerowym czasem symulacji, a nie w momencie wywołania funkcji.

Zmienne automatyczne w zadaniu lub funkcji statycznej są tworzone i inicjalizowane przy każdym wywołaniu zadania bądź funkcji. Na przykład poniższa przykładowa funkcja *count_ones* poprawnie zliczy liczbę jedynek portu wejściowego *data*.

Przykład 3.6. Poprawianie błędu inicjalizacji zmiennej

```
function int count_ones (input [31:0] data);
        automatic logic [31:0] count = 0;
```

```
        automatic logic [31:0] temp = data;
        for (int i=0; i<=32; i++)
        begin
                if (temp[0]) count++;
                temp >>= 1;
        end
        return(count);
endfunction
```

Aby zmienne automatyczne mogły być syntetyzowane, muszą być używane tylko do tymczasowego przechowywania danych, które nie wykracza poza zadanie, funkcję lub blok proceduralny. Dozwolona jest wewnętrzna inicjalizacja zmiennych automatycznych. Wewnętrzna inicjalizacja zmiennych statycznych nie nadaje się do syntezy, ale można ją wykorzystać w opisach symulacji. Należy zauważyć, że wewnętrzna inicjalizacja zmiennych za pomocą specyfikatora **const** jest syntetyzowalna.

Zalecenia dotyczące syntezy:

- w blokach **always** i **initial** powinny być używane zmienne statyczne, jeśli nie ma inicjalizacji wewnętrznej, i zmienne automatyczne, jeśli jest inicjalizacja wewnętrzna;
- jeśli funkcja lub zadanie ma być używane wielokrotnie, to ona i wszystkie jej zmienne muszą być automatyczne;
- jeśli zmienna ma zachować swoją wartość od jednego wywołania zadania lub funkcji do drugiego, musi być statyczna;
- jeśli zadanie lub funkcja reprezentuje zachowanie tylko jednego elementu sprzętowego i nie jest ponownie wykorzystywana, to ona i wszystkie jej zmienne muszą być statyczne.

3.3.7. Zakres i czas życia zmiennych

Informacje o *czasie życia* (*lifetime*) i *zakresie zmiennych* (*scope*) są bardziej istotne dla symulacji, ale mogą być również przydatne przy pisaniu kodu do syntezy.

Należy pamiętać, że SystemVerilog pozwala na deklarowanie zmiennych zarówno na zewnątrz, jak i wewnątrz modułów, interfejsów, programów i kontrolerów (*checkers*). Zmienne w SystemVerilog mogą być także deklarowane w statycznych i automatycznych zadaniach i funkcjach, jak też w nazwanych i nienazwanych blokach proceduralnych. Dodatkowo deklaracje zmiennych mogą mieć kwalifikatory **static** i **automatic**.

Zakres zmiennej nazywany jest również *kontekstem* (*scope*). Jest on określany przez miejsce, w którym zmienna jest zdefiniowana. Zmienne zadeklarowane poza modułem, programem, interfejsem, kontrolerem, zadaniem lub funkcją mają *zakres jednostki kompilacji* (*compilation unit scope*). Zmienne zadeklarowane w obrębie modułu, interfejsu, programu lub sterownika, ale poza zadaniem, funkcją bądź

blokiem proceduralnym, są *lokalne* i mają kontekst odpowiednio modułu, interfejsu, programu albo kontrolera. Zmienne zadeklarowane wewnątrz zadania statycznego, funkcji lub bloku są lokalne w swoim kontekście. Wszystkie te zmienne mają statyczny czas życia (istnieją przez cały czas trwania symulacji).

Poszczególne zmienne w zadaniu statycznym, funkcji lub bloku można zadeklarować jako automatyczne przy użyciu kwalifikatora **automatic**. Takie zmienne mają czas życia wywołania zadania, funkcji bądź bloku i są inicjalizowane przy każdym wywołaniu.

Zmienne zadeklarowane w automatycznym zadaniu, funkcji albo bloku są *lokalne* w odpowiednim kontekście i mają czas życia wywołania zadania, funkcji lub bloku, tzn. są domyślnie automatyczne. Są one inicjowane przy każdym takim wywołaniu. Zadania i funkcje automatyczne są deklarowane z kwalifikatorem **automatic**. Blok automatyczny to blok, w którym deklaracje są domyślnie automatyczne. Poszczególne zmienne w zadaniu automatycznym, funkcji albo bloku mogą być jawnie zadeklarowane jako statyczne za pomocą kwalifikatora **static**. Takie zmienne mają statyczny czas życia.

Deklaracja zmiennej musi poprzedzać każde proste (niehierarchiczne) odwołanie do niej. Podobnie deklaracje zmiennych w blokach procedur muszą poprzedzać wszelkie instrukcje wewnątrz bloku proceduralnego. Na przykład:

```
module msl;
        int st0;                        // zmienna statyczna
        initial begin
                int st1;                // zmienna statyczna
                static int st2;         // zmienna statyczna
                automatic int auto1;    // zmienna automatyczna
        end
        task automatic t1();
                int auto2;              // zmienna automatyczna
                static int st3;         // zmienna statyczna
                automatic int auto3;    // zmienna automatyczna
        endtask
endmodule
```

Język SystemVerilog umożliwia hierarchiczne odwołanie do dowolnej zmiennej statycznej. Dotyczy to również zmiennych statycznych zadeklarowanych w zadaniach i funkcjach automatycznych.

Zmienne w pętlach **for** są domyślnie automatyczne. Wartości dla zmiennych automatycznych nie mogą być przypisywane za pomocą operatora przypisania nieblokującego (<=) oraz operatora przypisania ciągłego (**assign**).

3.4. Wektory

3.4.1. Wektory i skalary

W języku SystemVerilog obiekt danych zadeklarowany z typem **logic** (**reg**) lub **bit** bez określenia zakresu jest traktowany jako obiekt jednobitowy i nazywany jest *skalarem* (*scalar*). Podobny obiekt ze specyfikacją zakresu nazywany jest *wektorem* (*vector*) i jest traktowany jako obiekt wielobitowy. Wektory to spakowane tablice skalarów, czyli ciągła sekwencja bitów.

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

[*msb*:*lsb*],

gdzie *msb* to numer najbardziej znaczącego bitu (po lewej stronie), a *lsb* to numer najmniej znaczącego bitu (po prawej stronie).

Elementy zakresu *msb* i *lsb* są wyrażeniami stałymi i mogą być liczbami całkowitymi (dodatnimi, ujemnymi lub równymi zero), stałymi, wyrażeniami lub wywołaniami funkcji stałych. Wyrażenia *msb* i *lsb* nie mogą zawierać wartości x i z. Wartość *lsb* może być większa, równa lub mniejsza od wartości *msb*.

Domyślnie wektory typu **logic** (**reg**) i **bit** są traktowane jako wartości bez znaku. Wektory ze znakiem muszą być jawnie zadeklarowane za pomocą słowa kluczowego **signed** lub podłączone do portu uprzednio zadeklarowanego ze znakiem.

Minimalny rozmiar zakresu wektorów w języku SystemVerilog wynosi 216 = 65 536, ale kompilatory mogą zwiększyć ten rozmiar do 1 miliona.

Przykłady deklaracji skalarów i wektorów:

```
wand w;                          // skalarna sieć typu wand
tri [15:0] bus_a;                // wektor 16-bitowy

logic a;                         // zmienna skalarna
logic [3:0] v;                   // 4-bitowy wektor składający się z elementów
                                 // v[3], v[2], v[1] i v[0]

logic signed [3:0] signed_reg;   /* wektor 4-bitowy, który może mieć
                                 wartości od -8 do 7 */

logic [-1:4] b;                  // wektor 6-bitowy składający się z elementów
                                 // b[-1], b[0],...,b[4]
wire w1, w2;                     // deklaracja dwóch elementów sieci
logic [4:0] x, y, z;             // deklaracja trzech zmiennych 5-bitowych
```

```
parametr SIZE = 256;
logic [SIZE-1 : 0] cells;                    // zastosowanie stałej SIZE
logic [clogb2(SIZE)-1 : 0] address;          // użycie stałej funkcji clogb2
```

W języku SystemVerilog do deklaracji wektorów można dodawać słowa kluczowe **vectored** i **scalared**. Jeśli używane są słowa kluczowe **vectored** i **scalared**, niektóre operacje na sieci wektorów mogą być ograniczone. Jeśli używane jest słowo kluczowe **vectored**, specyfikacja mocy, wybór bitów i wybór części wektorowej mogą być niedozwolone. Jeśli zostanie użyte słowo kluczowe **scalared**, dozwolone jest wybieranie bitów i części wektora. Na przykład:

```
tri scalared [63:0] bus64;    // szyna umożliwiająca wybór bitów i wybór części wektora
tri vectored [31:0] data;     // szyna nie pozwala na wybór bitów i części
```

3.4.2. Wybór bitów i wybór pola bitów

Wybór bitu (*bit-select*) lub adresowanie pojedynczego bitu wektora ma następujący format:

vector_name [*bit_number*],

gdzie *vector_name* jest nazwą wektora, *bit_number* jest numerem bitu, który ma zostać wybrany.

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

Pole bitowe (*bit field*) lub *selekcja częściowa* (*part-select*) to ciągła sekwencja bitów pewnego wektora. Wybór pola bitowego ma trzy następujące formaty:

vector_name [*msb* : *lsb*]
vector_name [*lsb_base* +: *width*]
vector_name [*msb_base* -: *width*],

gdzie: *msb* to najbardziej znaczący, a *lsb* to najmniej znaczący bit zakresu; *lsb_base* to numer najmniej znaczącego bitu początkowego (bazowego); *msb_base* to numer najbardziej znaczącego bitu początkowego (bazowego); *width* to szerokość wybranego pola.

W pierwszym formacie określa się część pola bitowego, podając numery najbardziej znaczącego i najmniej znaczącego bitu zakresu. W tym formacie *msb* i *lsb* mogą być liczbami, stałymi lub wywołaniami funkcji stałych.

Drugi i trzeci format umożliwiają identyfikację pola bitowego na podstawie numeru bitu początkowego i szerokości pola bitowego. Drugi format określa liczbę bitów pola w stosunku do bitu startowego w kierunku rosnącym, a trzeci format – w kierunku malejącym. W obu formatach szerokość *width* musi być stała, a *lsb_base* i *msb_base* mogą być zmienne, tzn. część wektora, która ma być wybrana, może znajdować się w różnych miejscach wektora.

Przykłady wyboru pola bitowego:

```
wire [7:0] data;        // deklaracja wektora 8-bitowego

/* przykłady poprawnego wyboru pola bitowego */
data [7:3];
data [7-:5];

/* przykłady nieprawidłowego wyboru pola bitowego */
data [3:7];             // kolejność bitów została naruszona
data [3+:5];            // poza zakresem
```

Należy zauważyć, że język SystemVerilog pozwala na wypełnienie wszystkich bitów wektora tymi samymi wartościami przy użyciu pojedynczych wartości bitowych '0, '1, 'x i 'z. Na przykład:

```
parameter N = 16;
reg [N-1 : 0] a, b, c, e; // deklaracja czterech zmiennych

a = '0;                 // wypełnienie wszystkich bitów zerami
e = '1;                 // wypełnienie wszystkich bitów jedynkami
b = 'z;                 // wypełnienie wszystkich bitów wartością z
c = 'x;                 // wypełnienie wszystkich bitów wartością x
```

3.5. Typy danych liczb całkowitych

Język SystemVerilog udostępnia następujące typy danych do reprezentacji liczb całkowitych: **bit**, **byte**, **shortint**, **int**, **longint**, **logic**, **reg**, **integer** i **time**. W rzeczywistości w SystemVerilog istnieją tylko dwa typy danych do reprezentowania liczb całkowitych: **bit** i **logic**. Pozostałe zaś to typy predefiniowane, które wywodzą się z typów **bit** i **logic** przez dodanie specyfikacji znaku i stałego zakresu:

```
byte        =    signed bit [7:0];
shortint    =    signed bit [15:0];
int         =    signed bit [31:0];
```

```
longint     =     signed bit [63:0];
reg         =     logic;
integer     =     signed logic [31:0];
time        =     unsigned logic [63:0] = 0;
```

W tabeli 3.4 scharakteryzowano typy danych całkowitych języka SystemVerilog.

TABELA 3.4. Typy danych liczb całkowitych

Typ danych	Liczba stanów bitów	Rozmiar w bitach	Domyślne znaczenie znakowości	Wartość początkowa	Właściwości
bit	2	zdefiniowane przez użytkownika	bez znaku	0	typ danych
byte	2	8	ze znakiem	0	podobne do char w C
shortint	2	16	ze znakiem	0	tak samo jak short w C
int	2	32	ze znakiem	0	tak samo jak int w C
longint	2	64	ze znakiem	0	tak samo jak long w C
logic	4	zdefiniowane przez użytkownika	bez znaku	x	typ danych
reg	4	zdefiniowane przez użytkownika	bez znaku	x	tak samo jak **logic**
integer	4	32	ze znakiem	x	tak samo jak w Verilog
time	4	64	bez znaku	x	niesyntetyzowany

ŹRÓDŁO: oprac. własne.

Głównymi typami danych używanymi do opisywania projektów w SystemVerilog są **bit** i **logic**. Różnica między nimi polega na tym, że typ **bit** wykorzystuje logikę z dwoma stanami, a typ **logic** – z czterema stanami. Dlatego typ **bit** jest częściej używany na wyższych poziomach abstrakcji, a typ **logic** jest używany na poziomie RTL do opisu projektów syntezowalnych.

Typ **byte** jest podobny do typu char w języku C, z tą różnicą, że domyślnie jest to typ ze znakiem. Typy **shortint**, **int** i **longint** mają swoje odpowiedniki w języku C i zostały dodane do języka Verilog, aby umożliwić wywoływanie funkcji napisanych w języku C z poziomu języka SystemVerilog i odwrotnie. Typ **reg** jest równoważny typowi **logic**, pozostawionemu w języku SystemVerilog dla zachowania kompatybilności z językiem Verilog.

Typ **integer** jest również zapożyczony z języka Verilog. Różni się on od typu **int** tym, że ma logikę o czterech stanach. Typ **time** pokrywa się z typem **time** języka Verilog, nie jest syntetyzowalny i jest przeznaczony do stosowania w modułach testowych.

3.6. Typy danych liczb rzeczywistych

Chociaż liczby rzeczywiste nie są syntetyzowalne i są używane tylko w modułach testujących, czytelnik powinien mieć świadomość ich istnienia.

Język SystemVerilog ma następujące typy danych do reprezentacji liczb rzeczywistych: **real**, **shortreal** i **realtime**. Typ **real** jest taki sam jak typ danych double w języku C, a typ **shortreal** jest taki sam jak typ danych float w języku C. Typ **realtime** jest synonimem typu **real** i jest przeznaczony do reprezentowania wartości czasu rzeczywistego.

3.7. Typ danych string

Typ danych **string** jest używany do reprezentowania łańcuchów znaków w języku SystemVerilog. Ciąg znaków to uporządkowany zbiór znaków. Długość łańcucha to liczba znaków w łańcuchu. Zmienne typu **string** są dynamiczne, ponieważ ich długość może się zmieniać podczas symulacji. Każdy znak zmiennej łańcuchowej może zostać wybrany do odczytu lub zapisu poprzez indeksowanie zmiennej. Pojedynczy znak zmiennej łańcuchowej jest typu **byte**.

Indeksy zmiennych łańcuchowych są numerowane od 0 do N-1, gdzie N jest długością łańcucha. Indeks 0 reprezentuje najbardziej wysunięty na lewo (pierwszy) znak łańcucha, a N-1 najbardziej wysunięty na prawo (ostatni) znak łańcucha. Zmienne łańcuchowe mogą przyjmować specjalną wartość „”, która jest łańcuchem *pustym*. Zmienna łańcuchowa nie może zawierać znaku specjalnego „\0”. Przypisanie wartości „\0” do znaku łańcuchowego jest ignorowane.

Do zadeklarowania zmiennej łańcuchowej używa się następującej składni:

string *variable_name* [= *initial_value*];

gdzie *variable_name* jest nazwą zmiennej łańcuchowej; *initial_value* jest wartością początkową, która jest literałem łańcuchowym, pustą wartością łańcuchową „” lub wyrażeniem typu danych łańcuchowych. Na przykład:

```
parameter string FSM_encoding = “user”;          // deklaracja wartości parametru
                                                 // jako zmienna łańcuchowa
(* syn_encoding = FSM_encoding *)                // definicja metody kodowania stanów
                                                 // wewnętrznych automatu skończonego
logic [2:0] state, next_state;
localparam [2:0] s0 = 3’d0, s1 = 3’d5, s2 = 3’d4, s3 = 3’d2, s4 = 3’d1;
```

W tym przykładzie zmienna łańcuchowa *FSM_encoding* jest używana do określenia sposobu kodowania wewnętrznych stanów automatu skończonego.

Jeśli w deklaracji nie podano wartości początkowej, zmienna łańcuchowa jest inicjalizowana jako pusty łańcuch „”.

Język SystemVerilog udostępnia zestaw operacji umożliwiających wykonywanie różnych działań na zmiennych łańcuchowych i literałach łańcuchowych: porównywanie, konkatenację, replikację, indeksowanie i wywoływanie metod. Ponadto język SystemVerilog zawiera szereg specjalnych metod do obsługi łańcuchów znaków.

Łańcuchowe typy danych nie są syntetyzowalne, ale mogą być używane do definiowania wartości parametrów, jak w powyższym przykładzie.

3.8. Stałe

Stałe (*constants*) to pewne dane, które muszą pozostać niezmienne przez pewien czas. Przyjrzyjmy się, jakie możliwości deklarowania stałych ma język SystemVerilog.

3.8.1. Stałe w języku SystemVerilog

Język SystemVerilog udostępnia trzy typy stałych na poziomie opisu (*elaboration-time*), **parameter**, **localparam** i **specparam**, które pozostają niezmienne podczas tworzenia modelu projektu, czyli w czasie kompilacji, oraz jeden typ stałych **const** na poziomie wykonywania (*run-time*), które pozostają niezmienne podczas symulacji projektu za pomocą symulatora. Stałe **parametr**, **localparam** i **specparam** są nazywane *stałymi parametrów* (*parameter constants*).

Stałe można także zdefiniować w kodzie źródłowym projektu za pomocą dyrektywy kompilatora `**define**.

Przykłady definicji stałych:

```
parameter SIZE =32;                 // parametr SIZE jest zdefiniowany
localparam byte divider = ":";      // zdefiniowany jest parametr lokalny divider
specparam delay = 5.0;              // zdefiniowana jest stała bloku specyfikacji delay
const int C =15;                    // zdefiniowana jest stała C czasu symulacji
`define ADD 3'h2;                   // określa nazwę ADD dla łańcucha znaków „3'h2”
```

3.8.2. Stałe parameter i localparam

Stałe **parameter** mogą być deklarowane wewnątrz modułu, poza modułem, a także mogą być używane do deklarowania parametrów modułu. Wszystkie stałe muszą mieć wartość początkową w momencie ich zadeklarowania. Po zadeklarowaniu instancji modułu można przypisać jego nowe wartości parametrów.

Uogólniona składnia deklaracji parametrów ma następującą postać:

```
parameter type [range] constant_name = value, … ;
```

gdzie: *type* jest typem danych stałej; [*range*] jest zakresem; *constant_name* jest nazwą stałej; *value* jest wartością stałej; elementy *type* i [*range*] są opcjonalne.

Jeśli podczas definiowania stałej nie zostanie podany zakres [*range*], to do reprezentacji stałej w pamięci zostanie przydzielona minimalna liczba bitów wystarczająca do reprezentacji wartości inicjującej. Jeśli dla stałej nie zostanie zdefiniowany typ danych, domyślnie zostanie ona przypisana do typu danych ostatniej przypisanej wartości.

Przykłady definicji parametrów:

```
parameter integer period = 10;        // stała całkowita

parameter [2:0]      s1 = 3'b001,   // trzy stałe 3-bitowe
                     s2 = 3'b010,   // ten styl jest używany przy ustawianiu kodów
                     s3 = 3'b100;   // stanów wewnętrznych automatów skończonych
```

Język SystemVerilog pozwala na definiowanie parametrów dowolnego typu. Na przykład:

```
// deklaracja parametrów typu real w wyniku przypisania wartości rzeczywistych
parameter r =5.7;
parameter delay = (r + 5)/2;
parameter pi = 3.14159265;
```

Oprócz modułów parametry mogą być deklarowane w interfejsach, programach i klasach języka SystemVerilog. Na liście portów modułu można pominąć słowo kluczowe **parameter**, np.:

```
module counter #(SIZE = 4)(…);        // tutaj parametr jest oznaczony znakiem #
      …
      logic [SIZE -1 : 0] v;
      …
endmodule
```

Na liście parametrów dany parametr może zależeć od wcześniejszych parametrów, np.:

```
module mnc # (int N = 3, M = N**2)(…);      // M wyznacza się na podstawie N
      …
endmodule
```

Wartości stałych **parameter** mogą być przedefiniowane za pomocą operatora **defparam** (nie jest to zalecane przez deweloperów SystemVerilog, ponieważ operator ten zostanie w przyszłości wyłączony z tego języka).

Typ danych **localparam** umożliwia określenie wartości stałych lokalnych, które obowiązują tylko w danym module. W przeciwieństwie do stałych deklarowanych za pomocą słowa kluczowego **parameter** parametry lokalne mogą być deklarowane w bloku generacyjnym, pakiecie, ciele klasy oraz w kontekście jednostki kompilacji. W tych kontekstach słowo kluczowe **parameter** jest synonimem słowa kluczowego **localparam**.

Parametrów lokalnych nie można zmieniać bezpośrednio za pomocą operatora **defparam**. Mogą być jednak one definiowane przez wyrażenia zawierające parametry, które można zmieniać za pomocą operatora **defparam**. Na przykład:

```
parameter N = 4;
localparam logic [N -1 : 0] s0 = 1, s1 = 1, s2 = 2;
```

3.8.3. Stałe specparam

Typ danych **specparam** umożliwia określenie wartości stałych dla bloku specyfikacji i może być zadeklarowany w zakresie modułu lub w zakresie bloku specyfikacji. Wartości stałe można zmienić za pomocą plików SDF (*standard delay format – standardowy format opóźnienia*) lub PLI (*programming language interface – interfejs języka programowania*).

Format danych typu **specparam**:

```
specparam constant_name = value, … .
```

Stałe zadeklarowane za pomocą słowa kluczowego **specparam** są przeznaczone wyłącznie do reprezentowania opóźnień synchronizacji i mogą być wyświetlane w dowolnym wyrażeniu. Na przykład:

```
specparam rise_clk = 150, fall_clk = 200;    // ustawienie wartości całkowitych
specparam thold = 5.5, tsetup = 10.0;        // ustawienie wartości rzeczywistych
specparam delay_1 = 10;      //  ustawienie stałej delay_1 na 10 jednostek czasu
specparam t_2 = 3 : 4 : 6 ;  //  ustawienie wartości opóźnienia t_2 w postaci ciągu znaków
```

3.8.4. Stałe const

Stałe **parameter**, **localparam** i **specparam** nie mogą być deklarowane w zadaniach i funkcjach automatycznych, a także w blokach **begin...end** i **fork...join**.

Wartość stałej **const** nie jest ustalana podczas tworzenia projektu, lecz podczas symulacji projektu. Dlatego takie stałe mogą być deklarowane w zadaniach i funkcjach automatycznych, blokach **begin...end** i **fork...join**. Nazwy hierarchiczne są również dozwolone przy deklarowaniu stałych. Na przykład:

```
const logic option = a.b.c;
```

Innymi słowy, stałą **const** można traktować jako zmienną, którą można zainicjować, ale której nie można przypisać wartości. Należy zauważyć, że typ **const** jest syntetyzowalny.

3.8.5. Znak $ jako wartość parametru

W języku SystemVerilog znak $ może być określony jako wartość parametru typu liczby całkowitej. Znak $ oznacza, że parametr jest niezdefiniowany. *Funkcja systemowa* **$isunfunded** służy do sprawdzania, czy parametr jest wartością $. Funkcja **$isunfunded** zwraca 1'b1, jeśli parametrem jest $.

W projektach znak $ jest często używany do określania wartości górnej granicy zakresu. Jeśli nie jest ona zdefiniowana, wykonywane jest odpowiednie działanie. Przykład użycia znaku $ jako parametru:

```
my_cheker # (0,0) quiet_never (...);          // sprawdzanie zakresu zerowego
my_cheker # (2,4) quiet_in_widow (...);       // sprawdzanie konkretnego zakresu
my_cheker # (0,$) quiet_any (...);            // sprawdzanie dowolnego zakresu
```

3.9. Wnioski

W języku SystemVerilog rozróżnia się *typy danych* (*data types*) i *obiekty danych* (*data objects*). Typ danych definiuje zbiór wartości oraz zbiór operacji, które można wykonywać na tych wartościach. Obiekt danych to identyfikator, z którym związane są wartości i typ danych, np. parametr, zmienna lub *sieć* (*net*).

Typy danych dzielą się na *singularne* (*pojedyncze*), reprezentujące pojedynczą wartość, i *zagregowane*, określające zbiór danych.

Typ *całkowity* (*integral*) to wartość, którą można przedstawić w postaci wektora. Typy całkowite są zawsze singularne.

Prosty typ *wektora bitowego* (*simple bit vector*) definiuje spakowaną tablicę bitów.

Zmienne (*variables*) i *sieci* (*nets*) w SystemVerilog reprezentują dwie główne grupy obiektów danych, które różnią się sposobem przypisywania i przechowywania wartości.

Sieci są używane do opisywania połączeń między komponentami projektu. Sieć nie przechowuje wartości sygnału (wyjątkiem jest sieć **trireg**).

Wartość sygnału w sieci jest określana przez źródło sygnału zwane *sterownikiem* (*driver*). Sterowniki sieciowe mogą być wyjściami prymitywów, portami wyjściowymi modułów, operatorami przypisania ciągłego **assign** itp.

W przypadku wielu sterowników wartość sieci jest określana za pomocą *funkcji decyzyjnej* (*resolution function*), która zależy od typu sieci, np. typ **wand** odpowiada przewodowemu AND, a typ **wor** odpowiada przewodowemu OR. Wartości sieci nie można przypisać za pomocą operatorów proceduralnych.

Zmienne są przeznaczone do stosowania w operatorach proceduralnych. To, czy zmienna w zsyntetyzowanym projekcie odpowiada rejestrowi czy przerzutnikowi, zależy tylko od kontekstu kodu źródłowego projektu.

Aby można było dokonać syntezy, wartość zmiennej musi być ustawiona przez pojedynczy operator proceduralny, pojedynczy operator ciągłego przypisania **assign**, pojedynczy moduł lub port prymitywu. Przypisanie wielu zmiennych nie jest syntetyzowane.

W SystemVerilog zmienne mogą być również uniami, strukturami lub tablicami.

Podczas definiowania wartości zmiennych nie należy mieszać przypisań proceduralnych i ciągłych.

Różne typy sieci odzwierciedlają różne zjawiska fizyczne, na których opierają się połączenia przewodowe. W SystemVerilog dostępne są następujące wbudowane typy sieci: **wire**, **tri**, **wand**, **triand**, **wor**, **trior**, **uwire**, **tri0**, **tri1**, **trireg**, **supply0** i **supply1**.

Typy **wire** i **tri** są używane do symulowania połączeń przewodowych. Typy **wor** i **trior** symulują „przewodowe OR". Typy **wand** i **triand** symulują „przewodowe AND". Typy **tri0** i **tri1** symulują sieci z elementami *pulldown* i *pullup*. Typy **supply0** i **supply1** służą do symulowania źródeł zasilania w obwodzie. Typ **trireg** jest używany do symulowania przechowywania ładunków i przechowuje ostatnią wartość logiczną.

Wartości sygnałów w sieciach mają osiem poziomów *mocy logicznej*: **supply**, **strong**, **pull**, **large**, **weak**, **medium**, **small** i **highz**.

Gdy obwód ma kilka źródeł sygnału, wartość sieci zależy od wartości na wyjściach sterowników oraz od typu sieci.

Zmienne w SystemVerilog mogą być deklarowane jawnie za pomocą słowa kluczowego **var** lub niejawnie bez słowa kluczowego **var**.

Typami danych dla zmiennych mogą być **logic** i **bit**. Typ danych **logic** definiuje dane z czterema wartościami logicznymi (stanami) każdego bitu, a typ danych **bit** – z dwoma wartościami logicznymi.

W języku SystemVerilog występują następujące typy zmiennych całkowitych o czterech stanach: **logic**, **reg**, **integer** i **time** oraz następujące typy zmiennych całkowitych o dwóch stanach: **bit**, **byte**, **short**, **int** i **longint**.

Oprócz liczb całkowitych obiektami danych mogą być liczby rzeczywiste (**shortreal**, **real** i **realtime**), wartości typów wyliczeniowych (**enum**), łańcuchy znaków (**string**), interfejsy wirtualne (**virtual interface**) (niesyntetyzowane) i zdarzenia (**event**) (niesyntetyzowane).

W języku SystemVerilog pojęcie zmiennej ma szersze znaczenie. Na przykład zmienną może być struktura, unia, tablica, klasa lub interfejs.

Inicjalizacja zmiennej w deklaracji jest nazywana *inicjalizacją zmiennej w wierszu* (*in-line variable initialization*) i jest równoważna inicjalizacji w bloku **initial**. Domyślnie wszystkie zmienne o typie danych z czterema stanami (**logic**) są inicjowane przez x, a z dwoma stanami (**bit**) przez 0.

Typ danych **logic** jest semantycznie równoważny typowi danych **reg** w Verilogu i domyślnie go zastępuje.

Typ danych **bit** został dodany do języka SystemVerilog specjalnie w celu modelowania projektów na bardziej abstrakcyjnych poziomach, takich jak *poziom systemu* (*system level*) i *poziom transakcji* (*transaction level*).

Możliwe jest przypisanie wartości zmiennych o dwóch stanach do zmiennych o czterech stanach i odwrotnie.

Do opisu projektów syntetyzowanych zaleca się stosowanie typów z czterema stanami.

W SystemVerilog typ **void** jest używany do definiowania funkcji, które nie zwracają wartości.

W SystemVerilog do deklarowania zmiennych statycznych i automatycznych (dynamicznych) używa się słów kluczowych **static** i **automatic**. Domyślnie wszystkie zmienne są statyczne.

Na poziomie modułu wszystkie zmienne są statyczne, a zmienne automatyczne mogą być deklarowane tylko w zadaniach, funkcjach, blokach **begin...end** lub **fork... join**.

Czasami wymagana jest zmienna statyczna w automatycznym zadaniu lub funkcji. W takich przypadkach stosuje się słowo kluczowe **static**.

Ustawianie wartości początkowych zmiennym w ramach funkcji i zadań jest nazywane *inicjalizacją wewnętrzną* (*internal initialization*).

Wewnętrzna inicjalizacja zmiennych statycznych nie jest syntetyzowalna. Jednak wewnętrzna inicjalizacja zmiennych za pomocą specyfikatora **const** jest możliwa do zsyntetyzowania.

Zalecenia dotyczące syntezy:

- w blokach **always** i **initial** powinny być używane zmienne statyczne, jeśli nie ma inicjalizacji wewnętrznej, oraz zmienne automatyczne, jeśli jest inicjalizacja wewnętrzna;
- jeśli funkcja lub zadanie mają być używane wielokrotnie, to zarówno one, jak i wszystkie ich zmienne muszą być automatyczne;
- jeśli zmienna ma zachować swoją wartość od jednego wywołania zadania lub funkcji do drugiego, musi być statyczna;
- jeśli zadanie lub funkcja reprezentują zachowanie tylko jednego elementu sprzętu i nie jest ponownie wykorzystywana, to one jak i wszystkie ich zmienne muszą być statyczne.

Kontekst i czas życia zmiennej zależą od tego, gdzie zmienna jest zdefiniowana: poza modułem, programem, interfejsem, sterownikiem, zadaniem lub funkcją; wewnątrz modułu, interfejsu, programu bądź sterownika; w statycznym zadaniu, funkcji albo bloku; w statycznym zadaniu, funkcji lub bloku z użyciem kwalifikatora **automatic**; w automatycznym zadaniu, funkcji lub bloku; w automatycznym zadaniu, funkcji bądź bloku z użyciem kwalifikatora **static**.

Zmienne w pętlach **for** są domyślnie automatyczne.

Wartości nie mogą być przypisywane do zmiennych automatycznych przy użyciu operatorów przypisania nieblokującego (<=) oraz przypisania ciągłego (**assign**).

W SystemVerilog dozwolone jest hierarchiczne odwołanie do dowolnej zmiennej statycznej.

W SystemVerilog obiekt danych typu **logic** lub **bit** bez określenia zakresu ([msb:lsb]) jest *skalarem*, a z określeniem zakresu – *wektorem*.

Elementy zakresów *msb* i *lsb* są wyrażeniami stałymi i mogą być liczbami całkowitymi (dodatnimi, ujemnymi lub zerowymi), stałymi, wyrażeniami i wywołaniami funkcji stałych.

Domyślnie wektory typu **logic** (**reg**) i **bit** są traktowane jako wartości bez znaku. Wektory ze znakiem muszą być jawnie zadeklarowane za pomocą słowa kluczowego **signed** lub podłączone do portu uprzednio zadeklarowanego ze znakiem.

Wybór bitu to odwołanie do pojedynczego bitu wektora. *Wybór pola bitowego wektora* to ciągła sekwencja bitów pewnego wektora. Słowo kluczowe **vectored** może być użyte do ograniczenia wyboru bitów i pola bitowego w wektorze.

Język SystemVerilog udostępnia trzy formaty wyboru pola bitowego, określając numer najstarszego i najmłodszego bitu, jak również numer bitu początkowego oraz malejącego lub rosnącego kierunku szerokości pola.

W języku SystemVerilog wektor o dowolnej szerokości może być wypełniony tymi samymi wartościami przy użyciu wartości jednobitowych '0, '1, 'x i 'z.

Typ danych **string** jest używany do opisywania łańcuchów w systemie Verilog. Pojedynczy znak zmiennej łańcuchowej jest typu **byte**. Łańcuchowe typy danych nie są syntetyzowalne, ale mogą być używane do definiowania wartości parametrów.

Stałe to pewne dane, które muszą pozostać niezmienne przez pewien czas.

Język SystemVerilog udostępnia trzy typy stałych na poziomie opisu (*elaboration-time*), **parameter**, **localparam** i **specparam**, które pozostają niezmienne podczas tworzenia modelu projektu, czyli w czasie kompilacji, oraz jeden typ stałych **const** na poziomie wykonywania (*run-time*), które pozostają niezmienne podczas symulacji projektu przez symulator.

Stałe można również zdefiniować za pomocą nazw w kodzie źródłowym projektu, używając dyrektywy kompilatora **‘define**.

Stałe **parameter** mogą być deklarowane wewnątrz modułu, poza modułem, a także mogą być używane do deklarowania jego parametrów. Oprócz modułów parametry mogą być deklarowane w interfejsach, programach i klasach języka SystemVerilog.

Typ danych **localparam** umożliwia określenie wartości stałych lokalnych, które są ważne tylko w obrębie danego modułu. Parametry lokalne mogą być deklarowane w bloku generacyjnym, pakiecie, ciele klasy oraz w kontekście jednostki kompilacji. W tych kontekstach słowo kluczowe **parameter** jest synonimem słowa kluczowego **localparam**.

Stałe **specparam** są przeznaczone wyłącznie do reprezentowania opóźnień synchronizacji w blokach specyfikacji.

Stałą **const** można traktować jako zmienną, którą można zainicjować, ale nie można jej przypisać wartości.

W języku SystemVerilog znak $ może być podany jako wartość parametru typu integer. Znak $ oznacza, że parametr jest niezdefiniowany. Funkcja systemowa **$isunfunded** służy do sprawdzania, czy parametr jest wartością $.

4. Typy użytkownika i typy enumeratywne

Niezależnie od tego, jak bogata jest oferta wbudowanych typów danych, w trakcie tworzenia złożonego projektu może dojść do momentu, w którym ich możliwości staną się niewystarczające. Aby rozwiązać ten problem, język SystemVerilog udostępnia mechanizm zwany *typami definiowanymi przez użytkownika* (*user-defined types*) lub po prostu *typami użytkownika*.

Typy danych zdefiniowane przez użytkownika są tworzone za pomocą słowa kluczowego **typedef** z istniejących typów danych zdefiniowanych przez użytkownika i predefiniowanych – w ten sposób powstaje złożona struktura danych dla projektu. Dlatego mechanizm typów użytkownika jest prostym, ale potężnym narzędziem do tworzenia złożonych projektów, zwłaszcza na poziomach systemowym i abstrakcyjnym. Dzięki typom użytkownika kod źródłowy projektu jest bardziej zwarty i czytelny. Mogą być one deklarowane lokalnie w modułach, w pakietach, dzięki czemu są dostępne dla wielu modułów, oraz zewnętrznie w jednostce kompilacji.

Typy enumeratywne (wyliczeniowe) są szeroko stosowane w językach programowania. Brak typów enumeratywnych w Verilogu utrudnia opisywanie projektów, które wykorzystują numerację kilku elementów, takich jak stany automatów skończonych. Język SystemVerilog eliminuje to niedociągnięcie, udostępniając enumeratywne typy danych. Ponadto umożliwia też stosowanie specjalnych notacji do automatycznego generowania nazw typów enumeratywnych, a także ma wiele metod pracy z typami wyliczeniowymi.

Należy pamiętać, że typy enumeratywne mają półścisłe ograniczenia dotyczące użycia, co przyczynia się do zwiększenia odporności projektu.

4.1. Typy zdefiniowane przez użytkownika

Język SystemVerilog znacznie rozszerza możliwości języka Verilog, pozwalając użytkownikom na definiowanie nowych typów sieci i zmiennych. *Typy zdefiniowane przez użytkownika* (*user-defined types*) lub *typy użytkownika* są syntetyzowalne i umożliwiają opisywanie, modelowanie i syntezę złożonych konstrukcji na bardziej abstrakcyjnym poziomie. Używanie typów zdefiniowanych przez użytkownika zapewnia więcej możliwości funkcjonalnych w procesie projektowania, a jednocześnie sprawia, że kody projektów są bardziej zwarte i czytelne.

Typy użytkownika w SystemVerilog są tworzone na podstawie istniejących typów przy użyciu słowa kluczowego **typedef**. Po zdefiniowaniu nowego typu można zadeklarować zmienne tego typu. Na przykład:

```
typedef int unsingned uint;  // definicja typu użytkownika uint
```

Po zdefiniowaniu nowego typu danych można zadeklarować zmienne typu użytkownika w zwykły sposób. Na przykład:

```
uint a, b;                   // deklaracja zmiennych a i b typu uint
```

Typy użytkownika mogą być deklarowane lokalnie, w pakiecie lub zewnętrznie. Jeśli typ użytkownika zostanie zadeklarowany lokalnie, będzie widoczny tylko w określonej części projektu: w module lub interfejsie (interfejsy omówiono w rozdziale 14). Gdy typ użytkownika jest zadeklarowany w pakiecie (pakiety są omówione w rozdziale 8), można się do niego bezpośrednio odwoływać bądź importować go do każdego modułu, interfejsu albo bloku programowego, w którym typ użytkownika jest używany. Zewnętrzne deklaracje typów zdefiniowanych przez użytkownika są wykonywane przez umieszczenie instrukcji **typedef** poza dowolnym modułem, interfejsem lub blokiem programu. W tym drugim przypadku typ użytkownika będzie widoczny w kontekście (zakresie) jednostki kompilacji **$unit** (jednostki kompilacji są omówione w rozdziale 8).

Bardziej złożone przykłady typów użytkownika zostaną obszernie przedstawione w dalszej części książki.

4.2. Typy enumeratywne

Typy enumeratywne lub *wyliczeniowe* (*enumerated types*) języka SystemVerilog pozwalają na używanie w kodzie *nazwanych stałych* (*named constants*). Ich składnia i semantyka są bardzo podobne do enumeratywów w języku C. Używanie nazw zamiast wartości zmniejsza prawdopodobieństwo wprowadzenia błędów do kodu projektu i poprawia jego czytelność.

4.2.1. Deklarowanie zmiennych typu enumeratywnego

Ogólna składnia deklaracji zmiennych typu enumeratywnego jest następująca:

enum [*base_type*] {*enumeration_list*} *variable_list*;

gdzie: **enum** jest słowem kluczowym; *base_type* jest typem bazowym elementów typu enumeratywnego; *enumeration_list* jest listą elementów typu enumeratywnego (lista enumeratywna); *variable_list* jest listą zmiennych typu enumeratywnego.

Elementy listy enumeratywów są nazywane *etykietami* (labels). Przykłady enumeratywów:

```
enum {BMW, FORD, TOYOTA, VW} car1, car2;
enum {red, green, blue} RGB;
```

W pierwszym przykładzie zadeklarowano dwie zmienne *car1* i *car2*, które mogą przyjmować tylko wartości BMW, FORD, TOYOTA lub VW. Podobnie w drugim przykładzie zadeklarowano zmienną *RGB*, która przyjmuje wartości red, green albo blue.

Zmienne typu enumeratywnego mogą przyjmować tylko wartości z zadeklarowanej listy wartości, mogą być porównywane tylko ze zmiennymi tego samego typu. Na przykład:

```
car1 = FORD;                    // dozwolone
car2 = BMW;                     // dozwolone
if (car1 == car2) y = 1;        // dozwolone
car1 = 7:                       // błąd: niedopasowanie typów
if (car2 ! = 8) ...;            // błąd: niedopasowanie typów
```

4.2.2. Typ bazowy typów enumeratywnych

Pytanie: jak są implementowane zmienne typu enumeratywnego? Czy to możliwe, że trzeci wiersz kodu z poprzedniego przykładu porównuje „FORD" i „BMW"? W rzeczywistości każdemu elementowi na liście typu enumeratywnego odpowiada jakaś liczba całkowita. Domyślnie elementy listy enum mają typ bazowy **int** i przyjmują kolejne wartości, zaczynając od 0. Jako typ bazowy można jawnie określić dowolny typ całkowity z punktu 3.5, np.:

```
enum bit {TRUE, FALSE} Boolean;
enum logic [1:0] {WAITE, LOAD, READY} state;
```

Wyraźne określenie typu bazowego dla zmiennych typu enumeratywnego pozwala uszczegółowić implementację sprzętową oraz zmniejszyć koszty implementacji (przypomnijmy, że typ **int** ma rozmiar 32 bitów).

Należy pamiętać, że rozmiar typu podstawowego musi być wystarczający do reprezentowania liczby każdego elementu na liście enumeratywnej. Na przykład:

```
enum logic {A, B, C} state;          // błąd: za dużo etykiet dla zmiennej 1-bitowej
```

4.2.3. Automatyczne generowanie etykiet typu enumeratywnego

Język SystemVerilog udostępnia dwie notacje do automatycznego generowania nazw etykiet w postaci listy enumeratywnej:

name[*N*] i *name*[*N:M*].

W tym przypadku nawiasy kwadratowe są częścią formatu.
Notacja *name*[*N*] tworzy sekwencję etykiet z nazwami:

name0, name1,...,nameN−1.

Zapis *name*[*N:M*] tworzy sekwencję etykiet, która zaczyna się od etykiety *nameN*, a kończy na etykiecie *nameM*. Jeśli *N*<*M*, numery w nazwach etykiet są uporządkowane rosnąco; jeśli *N*>*M*, są uporządkowane malejąco. Na przykład deklaracja typu enumeratywnego:

```
enum {RESET, S[3], W[9:11], STOP} state;
```

generuje następującą listę etykiet:

```
RESET, S0, S1, S2, W9, W10, W11, STOP.
```

4.2.4. Wartości typów enumeratywnych

Domyślnie pierwsza etykieta na liście enumeratywnej jest reprezentowana przez wartość 0, druga etykieta przez wartość 1, trzecia przez wartość 2 itd. Język SystemVerilog umożliwia jawne zdefiniowanie wartości dla każdej etykiety z listy enumeratywnej. Pozwala to na jeszcze większą szczegółowość w sprzętowej implementacji projektu ze zmiennymi typu enumeratywnego. Przykładowo stan automatu skończonego można zakodować w kodzie one-hot, kodzie Graya, kodzie Johnsona lub w inny sposób. Na przykład:

```
enum {ONE = 1, FIVE = 5, TEN = 10} state;
```

Nie trzeba podawać wartości każdej etykiety z listy enumeratywnej. Jeśli dla etykiety nie zostanie określona żadna wartość, wartość poprzedniej etykiety zostanie zwiększona o jeden. Na przykład:

```
enum {A = 1, B, C, D = 24, E, F} list1;
```

W tym przykładzie do etykiet przypisano następujące wartości:
A = 1, B = 2, C = 3, D = 24, E = 25 i F = 26.

Jeśli jednak takie ręczne przypisanie wartości spowoduje, że dwie etykiety będą miały tę samą wartość, zostanie wykryty błąd. Na przykład:

```
enum {A = 1, B, C, D = 3} list2;          // Błąd: C i D mają tę samą wartość
```

Błędem jest również przypisanie etykiecie wartości, która ma inny rozmiar niż typ podstawowy. Na przykład:

```
enum {WAITE =     3'b001,                 // Błąd: niedopasowanie typów
      LOAD =      3'b010,
      READY =     3'b100} state;
```

W tym przykładzie domyślnym typem podstawowym jest **int**, który może przyjmować wartości całkowite wielkości 32 bitów, a etykietom przypisane są wartości wielkości 3 bitów. Poprzedni błąd można skorygować w następujący sposób:

```
enum logic [2:0]
{WAITE =     3'b001,
LOAD =       3'b010,
READY =      3'b100} state;
```

Jeśli typem bazowym jest typ o czterech stanach, to etykietom można przypisać wartości zawierające x i z. Na przykład:

```
enum logic {ON = 1'b1, OFF = 1'bz} out;
```

Jeśli jakiejś etykiecie z listy enumeratywnej zostanie przypisana wartość z *x* lub *z* w którymkolwiek z bitów, to następna etykieta musi mieć przypisaną jawną wartość (tzn. nieznana wartość nie może być automatycznie zwiększana). Na przykład:

```
enum logic [1:0] {WAITE, ERR = 2'bxx, LOAD, READY} state;      // Błąd:
// dla etykiety LOAD nie można zdefiniować żadnej wartości
```

4.2.5. Enumeratywy typowane i anonimowe

Typy enumeratywne są często deklarowane jako typy użytkownika. Jeśli typ enumeratywny jest zadeklarowany jako typ zdefiniowany przez użytkownika za pomocą słowa kluczowego **typedef**, to jest on nazywany *typowanym typem enumeratywnym* (*typed enumerated type*). Jeśli podczas deklarowania typu enumeratywnego nie zostanie użyte słowo kluczowe **typedef**, to typ ten jest nazywany *anonimowym typem*

enumeratywnym (*anonymous enumerated type*). Do tej pory omówiliśmy tylko anonimowe typy enumeratywne. Przykład wpisanego typu enumeratywnego:

```
typedef enum {WAITE, LOAD, READY} states_t;    // deklaracja wpisanego
                                               // typu enumeratywnego states_t
states_t state, next_state;                    // deklaracja zmiennych typu states_t
```

4.2.6. Wykonywanie operacji na zmiennych typu enumeratywnego

Większość typów zmiennych w językach Verilog i SystemVerilog jest wprowadzana *luźno* (*loosely*). Oznacza to, że do zmiennej dowolnego typu można przypisać dowolną wartość, która zostanie automatycznie przekonwertowana na typ zmiennej zgodnie z regułami konwersji zdefiniowanymi w standardach Verilog i SystemVerilog.

W SystemVerilog typy enumeratywne są wprowadzane *półściśle* (*semi-strongly*). Oznacza to, że można przypisać tylko zmienne o podanym typie enumeratywnym:

- etykietę z listy o podanym typie enumeratywnym;
- inną zmienną tego samego typu enumeratywnego;
- wartość przekonwertowaną do podanego typu enumeratywnego.

Rozważmy następujący przykład:

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;
int a;
```

Etykiety na liście enumeratywnej są domyślnie reprezentowane przez typ bazowy **int**. Zmienne *state* i *next_state* są zadeklarowane jako typ enumeratywny states_t, a zmienna *a* jest zadeklarowana jako typ **int**. Następne przypisania są dopuszczalne:

```
state = next_state;          //przypisanie zmiennej tego samego typu
a = state + 1;
```

W ostatnim wyrażeniu zmienna *state* typu enumeratywnego jest reprezentowana jako typ podstawowy **int**, jej wartość jest zwiększana o jeden, wynik jest przypisywany do zmiennej *a* również typu **int**, a wszystko to jest dopuszczalne. Jednak odwrotne wyrażenie:

```
state = a + 1;
```

jest nieprawidłowe. Chodzi o to, że wyrażenie a + 1 ma typ **int**, natomiast *state* ma typ enumeratywny states_t, czyli typ wyrażenia po prawej stronie nie jest zgodny z typem

zmiennej typu enumeratywnego po lewej stronie. Następujące zadania są również nieprawidłowe:

```
state = state + 1;
state ++;
next_state + = state;
```

We wszystkich tych przykładach wyrażenie po prawej stronie znaku równości jest typu **int**, którego wartość jest przypisana do zmiennej *state* typu enumeratywnego states_t, co jest niedozwolone.

4.2.7. Konwersja wyrażeń na typ enumeratywny

Aby rozszerzyć zastosowanie zmiennych enumeratywnych, język SystemVerilog udostępnia dwa sposoby konwersji wyrażenia na typ enumeratywny: *operację konwersji typu* (') oraz *systemowe funkcje konwersje typu* **$cast**.

Zastosowanie operacji konwersji typu do wcześniej pokazanych przykładów ma postać:

```
state = state_t'(state + 1);
state = state_t'(state ++);
next_state = state_t'(next_state + state);
```

Te same przykłady z użyciem funkcji systemowej konwersji typów **$cast** pokazano poniżej:

```
$cast (state, state + 1);
$cast (state, state ++);
$cast (next_state, next_state + state);
```

Istnieje jedna bardzo ważna różnica między operacją konwersji (') a funkcją systemową **$cast**. Operacja konwersji (') nie sprawdza, czy wartość wyrażenia znajduje się na liście enumeratywnej, natomiast funkcja systemowa **$cast** robi to i generuje komunikat o błędzie, jeśli wartość znajduje się poza listą enumeratywną. Jednak nie wszystkie syntezatory obsługują funkcję systemową **$cast**.

Operacja konwersji typu (') jest syntetyzowalna. Jednak jej użycie, np. przy opisie automatów skończonych, może prowadzić do sytuacji, w której automat przechodzi do nieznanego stanu, z którego wyjście jest niezdefiniowane, czyli automat przestaje działać. Dlatego podczas korzystania z operacji przekształcania typu (') należy sprawdzić wartość wyrażenia za pomocą zewnętrznej logiki uzupełniającej.

Funkcja systemowa **$cast** do swojej implementacji może wymagać dodatkowej logiki, aby sprawdzić, czy znaleziono wartość wyrażenia na liście enumeratywnej. Ponadto funkcja **$cast** może nie być syntetyzowalna. Dlatego w projektach przeznaczonych do symulacji częściej stosuje się funkcję **$cast**.

Jeśli jesteśmy pewni, że wartość wyrażenia nigdy nie wyjdzie poza typ enumeratywny, to używamy konwersji typu ('), w przeciwnym razie używamy funkcji systemowej **$cast**. Zatem do dewelopera należy decyzja, czy użyć konwersji typu ('), czy funkcji systemowej **$cast**.

Operacja konwersji typu została omówiona bardziej szczegółowo w punkcie 5.2.1, a funkcja systemowa **$cast** w punkcie 5.2.2.

4.2.8. Specjalne metody systemowe dla typów enumeratywnych

Język SystemVerilog udostępnia kilka wbudowanych funkcji zwanych *metodami* (*methods*), służących do pracy ze zmiennymi typu enumeratywnego. Metody te są wywoływane w sposób podobny do wywołań metod klas w języku C++: nazwa metody jest dodawana na końcu nazwy zmiennej typu enumeratywnego z kropką jako separatorem.

Język SystemVerilog udostępnia następujące metody pracy ze zmiennymi typu enumeratywnego:

- <*variable_name*>.first – zwraca wartość pierwszego elementu listy enumeratywnej;
- <*variable_name*>.last – zwraca wartość ostatniego elementu listy enumeratywnej;
- <*variable_name*>.next [(<N>)] – zwraca wartość następnego elementu listy enumeratywnej; jeśli jako argument podano liczbę całkowitą N, to zwracana jest następna N-ta wartość, poczynając od pozycji wartości bieżącej;
- <*variable_name*>.prev [(<N>)] – zwraca wartość poprzedniego elementu listy enumeratywnej; jeśli jako argument podano liczbę całkowitą N, to zwracana jest poprzednia N-ta wartość, poczynając od pozycji wartości bieżącej;
- <*variable_name*>.num – zwraca liczbę etykiet na liście enumeratywnej;
- <*variable_name*>.name – zwraca łańcuchową reprezentację etykiety dla bieżącej wartości zmiennej, jeśli bieżąca wartość nie jest elementem typu enumeratywnego, metoda zwraca pusty łańcuch.

W powyższych definicjach *variable_name* jest nazwą zmiennej typu enumeratywnego.

Metoda next przenosi na początek listy, gdy osiągnięty zostanie koniec listy enumeratywnej, a metoda prev przenosi na koniec listy, gdy osiągnięty zostanie jej początek. Jeśli bieżąca wartość zmiennej w metodzie next nie jest elementem listy enumeratywnej, metoda next zwraca wartość pierwszego elementu listy. Podobnie jeśli aktualna wartość zmiennej w metodzie prev nie jest elementem listy enumeratywnej, metoda prev zwraca wartość ostatniego elementu.

Przykład 4.1 pokazuje użycie metod dla typów enumeratywnych SystemVerilog.

Przykład 4.1. Licznik rewersyjny modulo M

```
module counter_up_down_mod_M
        #(parameter M=4)
        (input clk, reset, up,                          // sygnały wejściowe
        output [N-1:0] Q,                               // wyjście licznika
        output roll);                                   // flaga osiągnięcia wartości modulo

        localparam N=clogb2(M-1);                       // rozmiar szyny wyjściowej Q

        function int clogb2(input [31:0] v);            // funkcja stała intlog2(M)
                for (clogb2 = 0; v > 0; clogb2 = clogb2 + 1)
                        v = v >> 1;
        endfunction

        enum {s[0:15]} state;                           // stany automatu skończonego

        always_ff @(posedge clk, negedge reset)  // określenie następnego stanu
                if (!reset) state <= s0;
                else
                        if (up)  state <= state.next;
                        else     state <= state.prev;

        assign Q = state;                               // określenie wyjścia licznika
        assign roll = (Q == M-1) ? 1'b1 : 1'b0;         // określenie flagi roll
endmodule
```

W powyższym przykładzie opisano rewersyjny licznik modulo. Wartość modulo jest określana przez parametr *M*, a kierunek licznika jest określany przez wartość sygnału wejściowego *up*: gdy *up* = 1, licznik jest inkrementowany, a gdy *up* = 0, licznik jest dekrementowany. Gdy licznik osiągnie wartość *M*-1, ustawiana jest flaga *roll*. Bieżąca wartość licznika jest również wyprowadzana na szynę *Q*. Funkcja stała clogb2(M) = intlog2(M) służy do wyznaczania szerokości *N* szyny *Q*.

Licznik jest zaimplementowany jako automat skończony. Stany automatu są definiowane przez zmienną *state*, która może przyjmować wartości etykiet od *s0* do s15 typu enumeratywnego. Wartości następnego stanu licznika są określane metodami next (gdy rośnie) i prev (gdy maleje). Wyjścia *Q* i *roll* są generowane przy użyciu operatora przypisania ciągłego **assign**. W tym przykładzie użyto bloku proceduralnego **always_ff**, który zostanie omówiony w rozdziale 11.

Wadą typów enumeratywnych jest to, że w notacjach, które automatycznie definiują nazwy etykiet, nie mogą być używane parametry, np.:

```
enum {s[0:M-1]} state;          // nie działa w syntezie
```

4.2.9. Wypisywanie typów enumeratywnych

Podczas wyświetlania można wypisywać zarówno wewnętrzną wartość zmiennej typu enumeratywnego, tj. wartość pewnego typu całkowitego, jak i nazwę etykiety z listy enumeratywnej. W tym drugim przypadku używana jest wbudowana metoda name. Na przykład:

```
enum logic [2:0]    {WAITE=3'b001,
                     LOAD =3'b010,
                     READY=3'b010} State, Next;
...
$display("\nCurrent state is %s (%b)", State.name, State);
...
$display("Next state will be %s (%b)", Next.name, Next);
```

W tym przykładzie metody State.name i Next.name zwracają nazwę etykiety w postaci ciągu znaków, a zmienne *State* i *Next* reprezentują wartości zmiennych typu całkowitego **logic** [2:0].

4.3. Wnioski

Typy zdefiniowane przez użytkownika (*user-defined types*) lub *typy użytkownika* są syntetyzowalne i umożliwiają opis, modelowanie i syntezę złożonych projektów na bardziej abstrakcyjnym poziomie. Typy zdefiniowane przez użytkownika w SystemVerilog są tworzone z istniejących typów za pomocą słowa kluczowego **typedef**. Po zdefiniowaniu nowego typu można zadeklarować jego zmienne.

Typy użytkownika mogą być deklarowane lokalnie, w pakiecie, lub zewnętrznie, w kontekście jednostki kompilacji **$unit**.

Typy enumeratywne SystemVerilog umożliwiają używanie w kodzie *nazwanych stałych*, zwanych *etykietami*.

Zmienne typu enumeratywnego mogą przyjmować wartości tylko z zadeklarowanej listy wartości, mogą być porównywane jedynie ze zmiennymi tego samego typu.

Z każdą etykietą na liście typu enumeratywnego jest związana liczba całkowita. Domyślnie elementy listy enumeratywnej mają typ bazowy **int** i przyjmują kolejne wartości, zaczynając od 0. Jako typ bazowy można jawnie określić dowolny typ całkowity. Rozmiar typu bazowego musi być wystarczający do reprezentowania liczby każdego elementu listy enumeratywnej.

Notacji *name*[*N*] i *name*[*N:M*] można użyć do automatycznego generowania nazw etykiet w postaci listy enumeratywnej.

Dla każdej etykiety z listy enumeratywnej można jawnie określić wartość. Jeśli dla danej etykiety nie zostanie określona wartość, zostanie jej przypisana wartość poprzedniej etykiety zwiększona o jeden. Nie można przypisać tej samej wartości

do dwóch etykiet ani przypisać wartości do etykiety, której rozmiar jest inny niż rozmiar typu bazowego. Nie wolno także automatycznie inkrementować wartości nieznanej.

Typy enumeratywne są często deklarowane jako typy użytkownika.

Jeśli typ enumeratywny jest zadeklarowany jako typ zdefiniowany przez użytkownika za pomocą słowa kluczowego **typedef**, to jest on nazywany *wpisanym typem enumeratywnym* (*typed enumerated type*). Jeśli podczas deklarowania typu enumeratywnego nie zostanie użyte słowo kluczowe **typedef**, wówczas typ ten jest nazywany *anonimowym typem enumeratywnym* (*anonymous enumerated type*).

Zmiennym danego typu enumeratywnego można przypisać: etykietę z listy enumeratywnego tego typu; inną zmienną tego samego typu enumeratywnego; wartość przekształcaną na ten typ enumeratywny.

Aby konwertować wartość wyrażenia na typ enumeratywny, język SystemVerilog udostępnia *operację konwersji* (') oraz *funkcję systemową* **$cast**. Operacja konwersji typu (') nie sprawdza, czy wartość wyrażenia znajduje się na liście enumeratywnej, podczas gdy robi to funkcja systemowa **$cast**. Operacja przekształcenia typu (') jest syntetyzowalna. Jednak nie wszystkie syntezatory obsługują funkcję systemową **$cast**.

Jeśli wiadomo, że wartość wyrażenia nigdy nie opuści typu enumeratywnego, można używać operacji konwersji ('), w przeciwnym razie – funkcji systemowej **$cast**.

Język SystemVerilog do pracy ze zmiennymi typu enumeratywnego udostępnia następujące wbudowane funkcje zwane *metodami* (*methods*): first, last, next, prev, num i name. Aby użyć danej metody, należy dodać jej nazwę do końca nazwy zmiennej typu enumeratywnego z kropką jako separatorem.

Podczas wypisywania za pomocą metody name można wypisać zarówno wewnętrzną wartość zmiennej enumeratywnej, tj. wartość używanego typu całkowitego, jak i nazwę etykiety z listy enumeratywnej.

5. Zgodność i konwersja typów

Jak wynika z poprzednich rozdziałów, typy danych odgrywają ważną rolę przy opisywaniu złożonych projektów, zwłaszcza na poziomach systemowym i abstrakcyjnym. Pojawia się naturalne pytanie: jakie funkcje są dostępne w języku SystemVerilog, aby obsługiwać typy danych?

Język Verilog wymaga ścisłej zgodności typów danych, np. podczas przypisywania wartości. W przypadku niedopasowania typu danych między zmienną a wyrażeniem kompilator generuje komunikat o błędzie. Język SystemVerilog znosi szereg ograniczeń dotyczących kompatybilności typów danych. W tym celu wprowadzono w nim pięć poziomów kompatybilności (zgodności) typów danych. Ponadto język SystemVerilog umożliwia zadeklarowanie dowolnego typu sieciowego jako kompatybilnego za pomocą słowa kluczowego **nettype**. W razie potrzeby można użyć *funkcji systemowej* **$typename**, aby uzyskać łańcuchową reprezentację typu danych widzianą przez kompilator.

W Verilogu konwersja wartości pewnego typu danych na wartość innego typu danych jest możliwa tylko poprzez przypisanie, tzn. musi zostać wykonana instrukcja przypisania. Nie zawsze jest to wygodne, a w niektórych przypadkach wręcz niemożliwe. Jedyną możliwą konwersją typu w Verilogu jest konwersja wartości ze znakiem na wartość bez znaku i odwrotnie za pomocą *funkcji systemowych* **$signed** i **$unsigned**.

Język SystemVerilog znacznie rozszerza możliwości konwersji typów danych, co zbliża go do języków programowania.

Aby konwertować (przekształcać) jeden typ danych na inny, SystemVerilog wykorzystuje *statyczną operację konwersji* (') oraz *dynamiczną funkcję systemową* **$cast**. Ponadto udostępnia szereg funkcji systemowych służących do konwersji liczb rzeczywistych na typy całkowite i odwrotnie.

Nowością w języku SystemVerilog jest także *typ bit-stream*, który można wykorzystać do symulowania transmisji pakietów danych przez szeregowe kanały komunikacyjne.

Oprócz tego język SystemVerilog udostępnia operację **type** umożliwiającą porównywanie ze sobą różnych typów danych.

5.1. Zgodność typów

Niektóre konstrukcje i operacje SystemVerilog wymagają pewnego poziomu *zgodności* (*kompatybilności*) *typów* (*type compatibility*), aby ich operandy były zgodne. W SystemVerilog istnieje pięć poziomów zgodności typów:

- *dopasowany* (*matching*);
- *równoważny* (*equivalent*);
- *zgodny przy przypisaniu* (*assignment compatible*);
- *zgodny przy przekształcaniu* (*cast compatible*);
- *niezgodny* (*nonequivalent*).

Użytkownicy mogą zdefiniować własny poziom identyfikacji typu za pomocą funkcji systemowej **$typename** lub interfejsu języka programowania (PLI – *programming language interface*).

5.1.1. Typy dopasowane

Dwa typy danych są *dopasowanymi typami danych* (*matching data types*) w następujących przypadkach:

a) każdy typ wbudowany jest dopasowany z innym takim samym typem:

```
typedef bit node1;
typedef bit node2;          // węzeł1 i węzeł2 są typami dopasowanymi
```

b) prosta redefinicja typu tworzy typy dopasowane:

```
typedef bit node1;          // bit i node1 są typami dopasowanymi
typedef type1 type2;        // typ1 i typ2 są typami dopasowanymi
```

c) anonimowy typ **enum**, **struct** lub **union** pasuje sam do siebie w przypadku obiektów danych zadeklarowanych w tej samej deklaracji i nie pasuje do żadnego innego typu danych:

```
struct packed {int a; int b;} ab1, ab2;     // ab1 i ab2 są typami dopasowanymi
struct packed {int a; int b;} ab3;          // typ ab3 nie jest taki sam jak typ ab1
```

d) predefiniowanie typu dla enumeratywów (**enum**), struktury (**struct**), związku (**union**) lub klasy (**class**) odpowiada samemu sobie oraz typowi obiektów danych zadeklarowanych przy użyciu tego typu danych:

```
typedef struct packed {int a; int b;} ab_t;   // deklaracja typu ab_t
```

```
ab_t ab1;                                   // deklaracja obiektów typu ab_t
ab_t ab2;                                   // ab1 i ab2 mają dopasowane typy

typedef struct packed {int a; int b;} ab_q; // deklaracja typu ab_q
ab_q ab3;                                   // typ ab3 nie jest taki sam jak ab1 lub ab2
```

e) prosty typ wektora bitowego, który nie ma predefiniowanej szerokości, i typ wektora bitowego, który ma predefiniowaną szerokość, są takie same (dopasowane), jeśli oba mają tę samą liczbę stanów (2 lub 4), oba są ze znakiem lub bez znaku, oba mają tę samą szerokość i zakres w postaci [width - 1 : 0]:

```
typedef bit signed [7:0] BYTE;          // typ BYTE jest taki sam jak typ byte
typedef bit signed [0:7] ETYB;          // typ ETYB nie jest taki sam jak typ byte
```

f) typy dwóch tablic są takie same, jeśli obie są *spakowane* (*packed*) lub obie są *rozpakowane* (*unpacked*); są tego samego typu tablicami, tzn. *o stałej wielkości* (*fixed-size*), *dynamiczne* (*dinamic*), *asocjacyjne* (*associative*) lub *kolejkowe* (*queue*); mają pasujące typy indeksów (w przypadku tablic asocjacyjnych); mają pasujące typy elementów; mają te same granice zakresów (w przypadku tablic o stałej wielkości):

```
typedef byte mem1 [256];
typedef bit signed [7:0] mem2 [256];    // typ mem1 jest taki sam jak typ mem2

typedef logic [1:0] [3:0] a1;
typedef logic [7:0] a2;                 // typ a1 nie jest taki sam jak typ a2

typedef logic b1 [] [2:0];
typedef logic b2 [] [0:2];              // typ b1 nie jest taki sam jak typ b2
```

g) jawne dodanie modyfikatorów **signed** lub **unsigned** do typu, który ma ten sam domyślny znacznik, tworzy typ dopasowany:

```
typedef byte signed my_byte;            // typ my_byte jest taki sam jak typ byte
```

h) predefiniowany typ enumeratywu, struktury, unii lub klasy zadeklarowany w pakiecie zawsze odpowiada samemu sobie, niezależnie od kontekstu, do którego importowany jest typ.

Innymi słowy, zgodność typów danych na poziomie dopasowania wymaga, aby obiekty były identyczne w wielu aspektach. Obecność znaku, szerokość zakresu, kolejność bitów (*big-endian* lub *little-endian*), liczba stanów każdego bitu (2 lub 4), liczba rozmiarów i wymiarów tablicy itp. muszą być zgodne.

5.1.2. Typy równoważne

Dwa typy danych są *równoważnymi typami danych* (*equivalent data types*) w następujących przypadkach:

a) jeśli dwa typy danych są dopasowane, to są równoważne;
b) anonimowe enumeratywy, struktura rozpakowana lub unia rozpakowana są równoważne samym sobie w przypadku obiektów danych zadeklarowanych w tej samej deklaracji i nie są równoważne żadnemu innemu typowi danych:

```
struct {int a; int b;} ab1, ab2;          // ab1 i ab2 mają równoważne typy
struct {int a; int b;} ab3;               // typ ab3 nie jest równoważny typowi ab1
```

c) tablice spakowane, struktury spakowane, unie spakowane i wbudowane typy całkowite są równoważne, jeśli zawierają tę samą liczbę bitów wspólnych, mają tę samą liczbę stanów (2 lub 4) i mają ten sam znak:

```
typedef bit signed [7:0] BYTE;                    // typ BYTE jest równoważny typowi byte
typedef struct packed signed {bit [3:0] a, b;} uint8;// typ uint8 jest równoważny
                                                  // typowi byte
```

d) rozpakowane typy tablic o stałym rozmiarze są równoważne, jeśli mają równoważne typy elementów i ten sam rozmiar, natomiast granice zakresów mogą się różnić:

```
bit [9:0] A [0:5];
bit [1:10] B [6];

typedef bit [10:1] uint10;
uint10 C [6:1];                   // A, B i C mają równoważne typy

typedef int anint [0:0];          // typ anint nie jest równoważny typowi int
```

e) typy tablic dynamicznych, tablic asocjacyjnych i kolejek są równoważne, jeśli mają równoważne typy elementów i równoważne typy indeksów (w przypadku tablic asocjacyjnych).

Równoważność typów danych usuwa niektóre ograniczenia związane z dopasowaniem typów. Równoważne typy danych mogą mieć różne granice zakresów, gdy ich wymiary są takie same, a wektory mogą mieć różną kolejność bitów. Poniższy przykład pokazuje możliwe sposoby deklarowania typów równoważnych oraz to, czy można ich używać.

Przykład 5.1. Różne sposoby deklaracji typów równoważnych

```
package p1;                          // deklaracja pakietu p1
    typedef struct {int A;} t_1;     // w pakiecie p1 jest zadeklarowany typ użytkownika t_1
endpackage

typedef struct {int A;} t_2;         // deklaracja typu użytkownika t_2
                                     // w jednostce kompilacji

module sub();                        // deklaracja modułu sub()
    import p1::t_1;                  // import typu t_1 z pakietu p1
    parameter type t_3 = int;        // przekazanie typu t_3 do modułu sub jako parametr
    parameter type t_4 = int;        // przekazanie typu t_4 do modułu sub jako parametr
    typedef struct {int A;} t_5;     // deklaracja typu użytkownika t_5
    t_1 v1;                          // deklaracja zmiennej
    t_2 v2;
    t_3 v3;
    t_4 v4;
    t_5 v5;
endmodule

module top();                        // deklaracja modułu najwyższego poziomu
    typedef struct {int A;} t_6;     // deklaracja typu użytkownika t_6
    sub #(.t_3(t_6)) s1 ();          // tworzenie instancji s1 modułu sub
                                     // z przekazywaniem typu t_6 jako parametr
    sub #(.t_3(t_6)) s2 ();          // tworzenie instancji s2 modułu sub
                                     // z przekazywaniem typ t_6 jako parametr
    initial begin
        s1.v1 = s2.v1;               // dopuszczalne – oba typy z pakietu p1
        s1.v2 = s2.v2;               // dopuszczalne – oba typy z jednostki kompilacji
        s1.v3 = s2.v3;               // dopuszczalne – oba typy z modułu górnego
        s1.v4 = s2.v4;               // dopuszczalne – oba typy to int
        s1.v5 = s2.v5;               // niedopuszczalne – typy z s1 i s2
    end
endmodule
```

5.1.3. Zgodność typów przy przypisaniu, przy przekształcaniu i niezgodność typów

Wszystkie typy równoważne i nierównoważne, między którymi zdefiniowano *reguły konwersji niejawnej* (*implicit casting rules*), są typami *zgodnymi przy przypisaniu* (*assignement-compatible types*). Na przykład wszystkie typy całkowite są kompatybilne przy przypisywaniu. Oznacza to, że wartość zmiennej o dowolnym typie całkowitym można przypisać dowolnej zmiennej o innym typie całkowitym.

Należy pamiętać, że konwersja między typami, które mają być zgodne, może spowodować utratę danych w wyniku obcięcia lub zaokrąglenia. Na przykład jeśli zmienna całkowita po lewej stronie przypisania zawiera mniej bitów niż wyrażenie po prawej stronie, to najbardziej znaczące bity prawej części są odrzucane, a zapisane w nich informacje są tracone.

Zgodność przy przypisywaniu może istnieć tylko w jednym kierunku. Na przykład typ **enum** można przekształcić na typ całkowity bez konwersji, ale nie odwrotnie.

Typy zgodne przy przekształcaniu (*cast-compatible types*) to wszystkie typy zgodne przy rzutowaniu, a także typy nierównoważne, dla których zdefiniowano jawne reguły konwersji. Na przykład aby przypisać typ całkowity do wyliczeniowego, wymagana jest konwersja typu (omówienie konwersji typów znajduje się w punkcie 5.2).

Typy niezgodne lub *niekompatybilne* (*incompatible types*) to wszystkie typy nierównoważne, które nie mają jawnych lub ukrytych reguł konwersji. Na przykład *uchwyty klas* (*class handles*), *uchwyty klas interfejsów* (*interface class handles*) i *chandles* **są niekompatybilne z innymi typami.**

5.1.4. Zgodność typów sieciowych

Wszystkie omówione powyżej problemy związane ze zgodnością typów dotyczyły typów danych zmiennych. Może pojawić się naturalne pytanie: jak wygląda sytuacja z kompatybilnością typów sieciowych? Odpowiedź jest bardzo prosta: język SystemVerilog pozwala na zadeklarowanie dowolnego typu sieciowego jako kompatybilnego za pomocą słowa kluczowego **nettype**. Na przykład:

```
nettype wand my_AND;          // typ my_AND zgodny z typem wand
```

5.1.5. Funkcja systemowa $typename

Jeśli użytkownik ma jakiekolwiek wątpliwości odnośnie do typu danych lub typu wyrażenia, można użyć *funkcji systemowej* **$typename**. Ma ona następujące formaty:

```
$typename ( expression )
$typename ( data_type )
```

gdzie *expression* jest wyrażeniem, którego typ danych ma być przedstawiony jako ciąg znaków; *data_type* jest typem danych, który ma być przedstawiony jako ciąg znaków.

Pierwszy format służy do określenia typu danych wyrażenia i przedstawienia go jako łańcucha znaków. Drugi zaś **służy do przedstawiania typu danych w postaci łańcucha znaków.**

Funkcja systemowa **$typename** zwraca łańcuch znaków odpowiadający typowi danych rozpoznanemu przez kompilator. Poniżej przedstawiono fragment kodu i łańcuch znaków, który funkcja **$typename** zwraca dla odpowiedniego typu danych.

```
// kod źródłowy                    // co zwróciłaby funkcja $typename

typedef bit node;                  // "bit"
node [2:0] X;                      // "bit [2:0]"
int signed Y;                      // "int"

package A;
      enum {A,B,C=99} X;           // "enum{A=32'sd0,B=32'sd1,C=32'sd99}A::e$1"
      typedef bit [9:1'b1] word;   // "A::bit[9:1]"
endpackage : A

import A::*;
module top;
      typedef struct {node A,B;} AB_t;
      AB_t AB[10];                 // "struct{bit A;bit B;}top.AB_t$[0:9]"
      ...
endmodule
```

Funkcji systemowej **$typename** można użyć do bardziej rygorystycznego porównywania łańcuchów typów danych i typów wyrażeń.

5.2. Konwersja typu

W przypadku niezgodności typów język SystemVerilog umożliwia przekształcenie typu danych obiektu na wymagany typ danych. W tym celu SystemVerilog wykorzystuje statyczną *operację konwersji typu*, oznaczaną znakiem apostrofu ('), oraz dynamiczną *funkcję systemowej konwersji typu* **$cast** (słowo „dynamiczna" zostanie wyjaśnione poniżej).

5.2.1. Statyczna operacja konwersji typów

Operacja konwersji (') służy do konwersji wartości z jednego typu na inny, wektor można konwertować na inny rozmiar, a wartości ze znakiem na wartości bez znaku lub odwrotnie.

Składnia statycznej operacji konwersji typu jest następująca:

casting_type '(*expression*)

gdzie *casting_type* jest typem danych, do którego przekształca się wyrażenie *expression*; (') jest apostrofem.

Typ danych, do którego przekształcany jest typ wyrażenia, nazywany jest *typem konwersji* (*casting type*). W tym formacie wyrażenie może być zwykłym wyrażeniem ze zmiennymi lub wyrażeniem stałym. Zwróć uwagę, że wyrażenie jest ujęte w nawiasy poprzedzone znakiem apostrofu (').

Operacja konwersji pozwala na przekształcenie wartości na dowolny typ, w tym na typy zdefiniowane przez użytkownika. Na przykład:

```
5 + int '(2.0 * 3.0)          // wynik stałego wyrażenia (o wartości rzeczywistej) 2,0 * 3,0
                              // jest konwertowany na typ int i dodawany do 5
```

Operacja konwersji umożliwia również konwersję wartości liczbowej do wektora o dowolnym rozmiarze. Na przykład:

```
logic [15:0] a, b, y;
y = a + b**16'(2);            // konwertuje wartość liczbową 2 do wektora 16-bitowego
```

Operacja konwersji może zamienić wartość na wartość ze znakiem lub bez znaku. Na przykład:

```
shortint a, b;
int y;
y = y - signed'({a,b});  // konwersja wyniku do wartości ze znakiem
```

Jeśli wyrażenie jest zgodne z przekształcanym typem, to operacja konwersji zwróci wartość, jaką będzie miała zmienna przekształcanego typu po przypisaniu wartości wyrażeniu.

Jeśli wyrażenie nie jest zgodne z przekształcanym typem, to jeżeli jest on typem ciągu bitów, zachowanie powinno być takie, jak opisano w podrozdziale 5.2.3 (Przekształcanie strumienia bitów). Jeśli zaś przekształcany typ jest typem enumeratywnym, zachowanie powinno być takie, jak opisano w podrozdziale 5.2.4 (Typy enumeratywne w wyrażeniach numerycznych).

Wraz z operacją konwersji typu można użyć słów kluczowych **signed** i **unsigned**, aby wskazać znak wyrażenia. Na przykład:

```
logic [7:0] a;
logic signed [7:0] b;

a = unsigned '(−4);           // a = 8'b1111100
b = signed '(4 'b1100);       // b = −4
```

Gdy typ **shortreal** jest konwertowany na 32-bitowy typ **int**, może dojść do utraty informacji z powodu zaokrągleń. Do konwersji wartości **shortreal** na **int** i odwrotnie bez utraty informacji powinny być używane *funkcje systemowe* **$shortrealtobits** i **$bitstoshortreal**.

Podczas wykonywania konwersji typów można również korzystać z funkcji systemowych **$itor**, **$rtoi**, **$bitstoreal**, **$realtobits**, **$signed** i **$unsigned**.

Należy zauważyć, że operacja konwersji typu jest statyczna, tzn. jest wykonywana bez sprawdzania, czy przekształcane wyrażenie znajduje się w dopuszczalnym zakresie typu, na który jest przekształcana wartość.

W poniższym przykładzie operacja konwersji jest używana do zwiększenia wartości zmiennej enumeratywnej *state* o jeden. Operacja konwersji statycznej nie sprawdza, czy wynik działania *state* + 1 jest poprawną wartością dla typu enumeratywnego *states_t*. Przypisanie wartości spoza zakresu *states_t* przy użyciu konwersji statycznej nie spowoduje wystąpienia błędu podczas kompilacji lub wykonywania. Dlatego należy uważać, aby nie spowodować przypisania nieprawidłowej wartości do zmiennej *next_state*.

Przykład 5.2. Użycie konwersji typu, która może spowodować błąd (wyjście poza zakres typu wyliczeniowego)

```
typedef enum {S1, S2, S3} states_t;          // deklaracja typu enumeratywnego states_t

states_t state, next_state;                  // deklaracja zmiennych typu states_t

always_comb begin
        if (state != S3)
                next_state = states_t'(state + 1);     // może wystąpić błąd
        else
                next_state = S1;                       // dopuszczalne
end
```

5.2.2. Dynamiczna systemowa funkcja konwersji typów $cast

Język SystemVerilog udostępnia *dynamiczną systemową funkcję konwersji typów* **$cast**. Jej składnia jest następująca:

$cast (*dest_var*, *source_exp*);

gdzie *dest_var* jest zmienną docelową, do której przypisywana jest wartość wyrażenia *source_exp*; *source_exp* jest wyrażeniem źródłowym, którego wynik jest przekształcany na typ zmiennej docelowej *dest_var*.

Dynamiczna funkcja systemowa **$cast** próbuje przypisać wyrażenie *source_exp* do docelowej zmiennej *dest_var*. Jeśli przypisanie jest niedopuszczalne, wyświetlany jest komunikat o błędzie w czasie wykonywania, a zmienna docelowa *dest_var* pozostaje niezmieniona.

Funkcję systemową **$cast** można wywołać jako funkcję lub jako zadanie, np.:

```
int flag;
int a, b;

$cast (b, 3.0 + 5.0);              // wywołanie zadania $cast
flag = $cast (a, 2.0 + 3.0);       // wywołanie funkcji $cast
```

Gdy **$cast** jest wywoływane jako zadanie, **$cast** próbuje przypisać wyrażenie źródłowe do zmiennej docelowej. Jeśli przypisanie jest niedopuszczalne, podczas wykonywania wystąpi błąd, a zmienna docelowa pozostanie niezmieniona.

Gdy **$cast** jest wywołane jako funkcja, **$cast** próbuje przypisać początkowe wyrażenie do zmiennej docelowej i zwraca flagę stanu wskazującą, że konwersja się powiodła. Jeśli konwersja jest poprawna, wartość wyrażenia jest przypisywana do zmiennej, a polecenie **$cast** zwraca 1. Jeśli konwersja jest niedopuszczalna, zmienna docelowa pozostaje niezmieniona, a funkcja **$cast** zwraca 0 i nie występuje błąd podczas wykonywania.

Różnica między statycznym przekształcaniem (') a dynamiczną funkcją systemową **$cast** polega na tym, że funkcja **$cast** oprócz konwersji sprawdza również, czy wartość wyrażenia należy do danego typu w czasie wykonywania (stąd jej nazwa *dynamiczna*). Jest to szczególnie istotne podczas pracy z typami enumeratywnymi. Na przykład:

```
// deklaracja typu enumeratywnego  colors
typedef enum {red, green, blue, yellow, white, black} colors;

colors col;                        // deklaracja zmiennej col typu colors

if (!$cast (col, 2 + 8))           // sprawdzenie wyniku działania funkcji $cast
$display ("Error color");          // nieprawidłowa wartość koloru
```

To samo przy użyciu operacji konwersji typu ('):

```
col = colors'(2 + 8);
```

Spowoduje to przypisanie zmiennej *col* wartości 10, która nie pasuje do żadnego elementu typu enumeratywnego colors. Kompilator nie wygeneruje żadnego błędu ani ostrzeżenia, a zmienna *col* przyjmie w czasie wykonywania nieprawidłową wartość, również bez generowania błędu.

W niektórych przypadkach użycie **$cast** może spowodować niepoprawną konwersję:

- konwersja typu **real** na typ **int**, gdy wartość liczby rzeczywistej jest zbyt duża, aby można ją było przedstawić w postaci **int**;
- konwersja wartości na typ **enum**, gdy wartość nie istnieje na liście typu enumeratywnego, jak w poniższym przykładzie:

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_latch begin
$cast(next_state, state + 1);
end
```

Statyczna operacja konwersji typu (') jest szybsza i nie wymaga dodatkowej logiki do jej wykonania w porównaniu z dynamiczną funkcją **$cast**. Operacja konwersji typu (') nie sprawdza jednak, czy wartość wyrażenia jest danego typu. Wybór, czy użyć operacji (') czy funkcji **$cast**, pozostawiamy użytkownikowi.

Należy pamiętać, że funkcji **$cast** nie można używać z operatorami, które bezpośrednio modyfikują oryginalne wyrażenie, takimi jak ++ lub +=:

```
$cast(next_state, ++state);          // zabronione
```

Głównym zastosowaniem funkcji **$cast** jest przypisywanie wyników wyrażenia do zmiennych typu enumeratywnego, które są zmiennymi ściśle typowanymi.

5.2.3. Konwersja strumieni bitów

Przekształcanie typu można również zastosować do tablic i struktur rozpakowanych, wykorzystując *przekształcanie strumienia bitów* (*bit-stream casting*). Umożliwia ono znaczne rozszerzenie możliwości konwersji różnych typów danych w systemie Verilog. *Typy strumienia bitowego* (*bit-stream types*) można spakować do strumienia bitowego. Są wśród nich:

- typy liczb całkowitych;
- typ łańcuchowy;
- rozpakowane tablice, struktury lub klasy elementów poprzednich typów;
- tablice dynamiczne (tablice dynamiczne i asocjacyjne oraz kolejki) dowolnego z poprzednich typów.

Na przykład struktura zawierająca tablicę elementów typu **int** jest typem strumienia bitów. Niech A będzie strumieniem bitów typu *source_t*, a B strumieniem bitów typu *dest_t*. Dozwolona jest konwersja *A* na *B* za pomocą jawnej konwersji typów:

```
B = dest_t'(A);
```

W celu dopasowania do strumienia bitów typ łańcuchowy jest traktowany jako dynamiczna tablica bajtów.

Niezależnie od tego, czy typ docelowy zawiera tylko elementy o stałym rozmiarze, czy też elementy o rozmiarze dynamicznym, dane są pobierane w kolejności od lewej do prawej.

Jeśli typy *source_t* i *dest_t* są typami o stałym rozmiarze, ale o różnych rozmiarach, konwersja powoduje wystąpienie błędu w czasie kompilacji. Jeśli *source_t* lub *dest_t* zawierają typy o rozmiarze dynamicznym, różnica w ich rozmiarach spowoduje wygenerowanie błędu albo w czasie kompilacji, albo w czasie wykonywania, gdy tylko będzie można określić niedopasowanie rozmiarów. Na przykład:

```
// Nieprawidłowa konwersja z 24-bitowej struktury na 32-bitowy typ int
struct {bit[7:0] a; shortint b;} a;
int b = int'(a);                    // błąd podczas kompilacji
```

Konwersja strumienia bitów może być używana do konwersji między różnymi typami zagregowanymi, takimi jak dwa typy struktur, struktura i tablica oraz dwa typy tablic. Konwersja ta może być przydatna do symulacji pakietowej transmisji danych za pomocą szeregowych strumieni komunikacyjnych.

5.2.4. Typy enumeratywne w wyrażeniach numerycznych

W wyrażeniach numerycznych można używać elementów zmiennych typu enumeratywnego. Wartością użytą w wyrażeniu jest wartość numeryczna związana z wartością enumeratywną. Na przykład:

```
// deklaracja enumeratywu jako typu użytkownika Colors
typedef enum { red, green, blue, yellow, white, black } Colors;

Colors col;          // deklaracja zmiennej col typu Colors
integer a, b;        // deklaracja dwóch zmiennych integer

a = blue * 3;        // wykonywanie przypisań
col = yellow;
b = col + green;
```

W tym przypadku zmiennej *a* zostanie przypisana liczba 6 ((blue = 2) * 3), a zmiennej *b* – liczba 4 ((yellow = 3) + (green = 1)).

W wyrażeniu zmienne lub elementy typu enumeratywnego są automatycznie konwertowane na typ bazowy. Jeśli typ wyrażenia nie jest równoważny typowi zmiennej typu enumeratywnego, wymagana jest konwersja typu wyrażenia na typ enumeratywny. Na przykład:

```
typedef enum {Red, Green, Blue} Colors;          // deklaracja typu Colors
typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;        // deklaracja typu Week

Colors C;          // deklaracja zmiennej C typu Colors
Week W;            // deklaracja zmiennej W typu Week
int a;

C = Colors'(C+1);  // C jest przekształcony na liczbę całkowitą, a następnie dodawana jest
                   // liczba jeden, a następnie przekształcony z powrotem na typ Colors

C = C + 1;         // niedopuszczalne, ponieważ wszystkie przypisania będą
                   // przypisaniami wyrażenia bez konwersji

C++;
C+=2;
C = 1;

C = Colors'(Su);   // dopuszczalne, umieszcza wartość poza zakresem w C
a = C + W;         // dopuszczalne, C i W są automatycznie konwertowane na int
```

5.2.5. Operacja type

Operacja **type** zwraca typ danych wyrażenia. Odwołanie do typu może być używane w konwersjach typów, w deklaracjach obiektów danych oraz w przypisywaniu lub predefiniowaniu parametrów typu. Operacji **type** można również używać w wyrażeniach warunkowych do porównywania różnych typów ze sobą.

Gdy operacja **type** jest używana w deklaracji sieci, musi być poprzedzona słowem kluczowym odpowiedniej sieci, a w deklaracji zmiennej – słowem kluczowym **var**. Na przykład:

```
var type (a + b) c, d;          // zadeklarowane zmienne c i d z typem wyrażenia (a + b)
c = type (i + 3)'(v[15:0]);     // wektor v jest przekształcany na typ wyrażenia (i + 3)
```

Operację **type** można również zastosować do typu danych. Na przykład:

```
localparam type T = type (bit[12:0]);     // deklarowany jest parametr lokalny T
                                          // z typem wektora bitowego bit[12:0]
```

Operacji **type** można używać w instrukcjach **case**. Dwa odwołania do **type** są odczytywane jako równe, jeśli typy, do których się odwołują, są takie same.

Przykład 5.3. Użycie operacji **type** do porównania typów danych w operatorze **case**

```
bit [12:0] A_bus, B_bus;
parameter type bus_t = type(A_bus);
generate
        case (type(bus_t))
                type(bit[12:0]):        addfixed_int #(bus_t) (A_bus,B_bus);
                type(real):             add_float #(type(A_bus)) (A_bus,B_bus);
        endcase
endgenerate
```

W przykładzie 5.3 zadeklarowano dwie zmienne *A_bus* i *B_bus* jako wektory **bit**[12:0]. Parametr *bus_t* jest wtedy zdefiniowany jako typ danych równy typowi zmiennej *A_bus*. Blok generujący (**generate**) wykorzystuje operator **case** do wyboru funkcji, która ma zostać dołączona do kodu (addfixed_int lub add_float) w zależności od typu argumentów (**bit**[12:0] lub **real**). W tym celu typ *bus_t* jest obecny w operatorze **case** jako wyrażenie sprawdzane, a typy danych (**bit**[12:0] lub **real**) są używane jako wyrażenia stałe. Funkcja addfixed_int przyjmuje jako parametr typ *bus_t*, a funkcja add_float typ zmiennej *A_bus*.

Ten przykład pokazuje, że język SystemVerilog pozwala na deklarowanie typów danych jako parametrów i przekazywanie ich do funkcji.

5.2.6. Funkcje systemowe do konwersji typów

Oprócz operacji konwersji typów (') i systemowej funkcji **$cast** język SystemVerilog udostępnia szereg *funkcji systemowych* służących do konwersji typów, które mają następujące formaty:

integer $rtoi (*real_val* **)**

real $itor (*int_val* **)**

[63:0] **$realtobits (** *real_val* **)**

real $bitstoreal (*bit_val* **)**

[31:0] **$shortrealtobits (** *shortreal_val* **)**

shortreal $bitstoshortreal (*bit_val* **)**

gdzie: *real_val* jest wartością rzeczywistą; *int_val* jest wartością całkowitą; *bit_val* jest wektorem bitowym; *shortreal_val* jest wartością typu **shortreal**.

Rozważane funkcje systemowe służą głównie do przekształcania wartości rzeczywistych na całkowite i odwrotnie. Takie przeliczenia można wykorzystać np. do przesyłania przez porty modułów liczb rzeczywistych.

Funkcja systemowa **$itor** konwertuje wartość całkowitą na liczbę rzeczywistą typu **real**. Funkcja systemowa **$rtoi** konwertuje wartość rzeczywistą na wartość całkowitą typu **integer**.

Funkcja systemowa **$realtobits** konwertuje wartość z typu rzeczywistego na 64-bitową reprezentację wektorową liczby rzeczywistej. Funkcja systemowa **$bitstoreal** konwertuje obraz bitowy utworzony przez funkcję **$realtobits** na wartość typu **real**.

Funkcja systemowa **$shortrealtobits** konwertuje wartość z typu **shortreal** na 32-bitową wektorową reprezentację liczby rzeczywistej. Funkcja systemowa **$bitstoshortreal** konwertuje obraz bitowy utworzony przez funkcję **$shortrealtobits** na wartość typu **shortreal**.

Poniższy przykład pokazuje użycie funkcji systemowych **$realtobits** i **$bitstoreal**.

Przykład 5.4. Użycie funkcji systemowych **$realtobits** i **$bitstoreal** do przekazania przez port modułu liczb rzeczywistych

```
module driver (output [64:1] net_r);                  // wyjście: 64-bitowy wektor
      real r;                                         // liczba rzeczywista

      wire [64:1] net_r = $realtobits(r);             // konwersja liczby rzeczywistej
                                                      // do 64-bitowego wektora

endmodule

module receiver (input [64:1] net_r);                 // wejście: 64-bitowy wektor
      real r;                                         // liczba rzeczywista

      initial assign r = $bitstoreal(net_r);          // konwersja wektora 64-bitowego
                                                      // na liczbę rzeczywistą

endmodule
```

5.3. Wnioski

Język SystemVerilog ma pięć poziomów zgodności typów: *dopasowany* (*matching*), równoważny (*equivalent*), *zgodny przy przypisaniu* (*assignment compatible*), *zgodny przy przekształcaniu* (*cast compatible*) i *nierównoważny* (*nonequivalent*).

Użytkownicy mogą zdefiniować własny poziom identyfikacji typów za pomocą funkcji systemowej **$typename** lub interfejsu języka programowania PLI.

Gdy typy danych są zgodne, znak, szerokość zakresu, kolejność bitów (*big-endian* lub *little-endian*), liczba stanów każdego bitu (2 lub 4), liczba wymiarów i rozmiary tablicy itp. muszą być takie same na poziomie zgodności.

W równoważnych typach danych granice zakresów mogą być różne, gdy rozmiary są takie same, a wektory mogą mieć różną kolejność bitów.

Wszystkie typy równoważne i nierównoważne, między którymi zdefiniowano *reguły konwersji niejawnej* (*implicit casting rules*), są typami *zgodnymi przy przypisaniu* (*assignement-compatible types*). Na przykład wszystkie typy całkowite są kompatybilne przy przypisywaniu. Konwersja między typami zgodnymi przy przypisywaniu może spowodować utratę danych w wyniku obcięcia lub zaokrąglenia.

Typy zgodne przy przekształcaniu (*cast-compatible types*) to wszystkie typy zgodne przy przekształcaniu oraz typy nierównoważne, dla których zdefiniowano jawne reguły konwersji.

Typy niekompatybilne (*incompatible types*) to wszystkie typy nierównoważne, które nie mają jawnych lub ukrytych reguł konwersji.

Język SystemVerilog umożliwia zadeklarowanie dowolnego typu sieci jako kompatybilnego za pomocą słowa kluczowego **nettype**.

Funkcja systemowa **$typename** zwraca łańcuch znaków, który odpowiada typowi danych rozpoznanemu przez kompilator. Można jej użyć do dokładniejszego porównywania łańcuchów typów danych i typów wyrażeń.

W przypadku niezgodności typów język SystemVerilog umożliwia przekształcanie typu danych obiektu na wymagany typ danych. W języku SystemVerilog do tego służą *statyczna operacja konwersji* (') oraz dynamiczna *funkcja systemowa* **$cast**.

Typ danych, do którego przekształcany jest typ wyrażenia, nazywany jest *typem konwersji* (*casting type*).

Operacja konwersji umożliwia przekształcanie wartości na dowolny typ, w tym na typy zdefiniowane przez użytkownika. Przekształcanie umożliwia konwersję wartości na wektor o dowolnym rozmiarze, wektora na wektor o dowolnym innym rozmiarze oraz przekształcanie wartości ze znakiem na wartości bez znaku lub odwrotnie.

Podczas wykonywania konwersji typów można korzystać z funkcji systemowych **$itor**, **$rtoi**, **$bitstoreal**, **$realtobits**, **$shortrealtobits**, **$bitstoshortreal**, **$signed** i **$unsigned**.

Operacja konwersji typu jest statyczna, tzn. jest wykonywana bez sprawdzania, czy przekształcane wyrażenie znajduje się w prawidłowym zakresie typu, na który jest przekształcana wartość.

Funkcję systemową **$cast** można wywołać jako funkcję lub jako zadanie.

Gdy **$cast** jest wywoływane jako zadanie, **$cast** próbuje przypisać oryginalne wyrażenie do zmiennej docelowej. Jeśli przypisanie jest niedopuszczalne, podczas wykonywania symulacji wystąpi błąd, a zmienna docelowa pozostanie niezmieniona.

Gdy **$cast** jest wywołane jako funkcja, **$cast** próbuje przypisać początkowe wyrażenie do zmiennej docelowej i zwraca flagę stanu wskazującą, że konwersja się powiodła. Jeśli konwersja jest poprawna, wartość wyrażenia jest przypisywana do zmiennej,

a polecenie **$cast** zwraca 1. Jeśli konwersja jest niepoprawna, zmienna docelowa pozostaje niezmieniona, a funkcja **$cast** zwraca 0 i nie występuje błąd podczas wykonywania.

Różnica między przekształcaniem statycznym (') a dynamiczną funkcją systemową **$cast** polega na tym, że funkcja **$cast** oprócz konwersji sprawdza również, czy wartość wyrażenia należy do danego typu w czasie wykonywania (stąd jej nazwa *dynamiczna*). Jest to szczególnie istotne podczas pracy z typami enumeratywnymi.

Statyczna operacja konwersji typu (') jest szybsza i nie wymaga dodatkowej logiki do jej wykonania w porównaniu z dynamiczną funkcją **$cast**.

Funkcji **$cast** nie można używać z operatorami, które bezpośrednio modyfikują oryginalne wyrażenie, takimi jak ++ lub +=. Jej podstawowym zastosowaniem jest przypisywanie wyników wyrażenia do zmiennych typu enumeratywnego, które są zmiennymi ściśle typowanymi.

Typy strumienia bitowego (*bit-stream types*) to typy, które można spakować do strumienia bitowego: typy całkowite, typy łańcuchowe, tablice, struktury lub klasy rozpakowane oraz tablice dynamiczne.

Konwersja strumienia bitów może być używana do konwersji między różnymi typami zagregowanymi, takimi jak dwa typy struktur, struktura i tablica oraz dwa typy tablic. Konwersja ta może być przydatna do symulacji pakietowej transmisji danych za pomocą szeregowych strumieni komunikacyjnych.

Jeśli typ wyrażenia nie jest równoważny enumeratywnemu typowi zmiennej, wymagana jest konwersja typu wyrażenia na typ enumeratywny.

Operacja **type** zwraca typ danych wyrażenia. Można jej używać w wyrażeniach warunkowych do porównywania ze sobą różnych typów.

Gdy operacja **type** jest używana w deklaracji sieci, musi być poprzedzona słowem kluczowym odpowiedniej sieci, a w deklaracji zmiennej musi być poprzedzona słowem kluczowym **var**. Operacji **type** można używać w instrukcjach **case** do sprawdzania typów wyrażeń.

6. Struktury i unie

Jak już wcześniej wspomniano, typy danych w SystemVerilog również dzielą się na *pojedyncze* lub *singularne* (*singular*) i *zagregowane* lub *zespołowe* (*aggregate*). Do tych ostatnich zaliczamy: *struktury* (*structures*), *unie* (*unions*) i *tablice* (*arrays*). Nowością w porównaniu z językiem Verilog są struktury i unie. Pod wieloma względami przypominają one struktury i unie w znanych językach programowania, co jeszcze bardziej zbliża SystemVerilog do tych języków.

Struktury pozwalają łączyć elementy różnych typów w jedną całość (np. słowa instrukcji procesora), co jest bardzo przydatne do opisywania danych na poziomie abstrakcji i projektowania złożonych projektów na poziomie systemu. Język SystemVerilog udostępnia wiele sposobów przypisywania wartości do elementów struktury, co umożliwia elastyczną pracę ze strukturami.

Podobnie jak wszystkie zagregowane typy danych, struktury można rozpakowywać i pakować. W SystemVerilog struktury spakowane mogą być obsługiwane jako wektory bitowe. Dodatkową elastyczność w pracy ze strukturami zapewnia możliwość przekazywania struktur przez porty modułu oraz jako argumentów zadań i funkcji.

Unie w SystemVerilog, tak jak w językach programowania, reprezentują fragment pamięci, który może być różnie interpretowany dla różnych typów danych.

Nowym typem unii w SystemVerilog jest *unia oznaczona* (*tagged union*). Oprócz wartości elementu przechowuje ona również wartość *znacznika* (*tag*) określającego nazwę ostatniego elementu unii, w którym wartość została zapisana. Unie oznaczone zapewniają kolejny poziom ochrony przed przypadkowym odczytem danych, zwiększając tym samym niezawodność projektów.

W ten sposób struktury i unie są zagregowanymi typami danych, które pozwalają na spełnienie wymagań poziomu systemowego przy opracowywaniu projektu w języku SystemVerilog.

6.1. Struktury

6.1.1. Deklaracja struktury

Struktura (*structure*) jest zbiorem różnych typów danych, do których można się odwoływać przez jej nazwę jako do całości, a do każdego elementu struktury przez nazwę jej elementu. Deklaracja struktury ma następującą składnię:

struct [**packed**] [*signing*] {*struct_member*; ...} *struct_name*;

gdzie: **struct** jest słowem kluczowym określającym strukturę; **packed** jest słowem kluczowym określającym strukturę spakowaną; *signing* jest określeniem użycia znaku w strukturze, może być **signed** lub **unsigned**; *struct_member* jest elementem struktury; *struct_name* jest nazwą struktury. Konstrukcje [**packed**] i [*signing*] mogą być nieobecne.

Przykład deklaracji struktury:

```
struct {
        int a, b;                 // zmienne 32-bitowe
        opcode_t opcode;          // zmienna typu użytkownika
        logic [23:0] address;     // zmienna 24-bitowa
        bit error;                // zmienna 1-bitowa o dwóch stanach
} Instruction_Word;
```

W tym przykładzie zadeklarowano strukturę *Instruction_Word*, która definiuje słowo instrukcji procesora składające się z 32-bitowych operandów *a* i *b*, kodu operacji *opcode*, 24-bitowego adresu *address* i flagi błędu *error*. Wszystkie te elementy różnych typów są połączone w jednej strukturze *Instruction_Word*.

Składnia dostępu do elementu w strukturze jest taka sama jak w języku C:

```
<structure_name>.< struct_member >
```

gdzie *structure_name* jest nazwą struktury, a *struct_member* jest nazwą elementu struktury. Na przykład:

```
Instruction_Word.address = 32'hF000001E;
```

Struktura różni się od tablicy tym, że tablica jest zbiorem elementów tego samego typu i rozmiaru, natomiast struktura jest zbiorem zmiennych lub stałych, które mogą być różnego typu i rozmiaru. Kolejna różnica polega na tym, że do elementów tablicy odwołujemy się za pomocą indeksu, do elementów struktury zaś za pomocą nazwy elementu.

W SystemVerilog struktura może być zadeklarowana przykładowo jako zmienna (przy użyciu słowa kluczowego **var**) lub jako element sieci (przy użyciu słowa kluczowego **wire...**), np.:

```
var struct {                      // deklaracja struktury Instruction_Word_var jako zmiennej
        logic [31:0] a, b;
        logic [ 7:0] opcode;
        logic [23:0] address;
} Instruction_Word_var;
```

```
wire struct {                    // deklaracja struktury Instruction_Word_net jako elementu sieci
        logic [31:0] a, b;
        logic [ 7:0] opcode;
        logic [23:0] address;
} Instruction_Word_net;
```

Podczas deklarowania struktury jako typu sieciowego wszystkie elementy struktury muszą być czterowartościowe. Domyślnie struktura jest traktowana jako zmienna.

Chociaż strukturę jako całość można zadeklarować jako typ sieciowy, to nie można ich używać w strukturach. Należy pamiętać, że sieci w języku SystemVerilog mogą być grupowane pod tą samą nazwą za pomocą interfejsów, które omówiono w rozdziale 14.

W projektach struktury są często deklarowane jako typy użytkownika za pomocą słowa kluczowego **typedef**. Takie struktury, podobnie jak enumeratywy, są nazywane *typowanymi* (*typed*); a w przeciwnym razie *anonimowymi* (*anonymous*). Na przykład:

```
typedef struct {                 // deklaracja wprowadzonej struktury instruction_word_t
        logic [31:0] a, b;
        logic [ 7:0] opcode;
        logic [23:0] address;
} instruction_word_t;

instruction_word_t IW;           // tworzenie instancji IW struktury instruction_word_t,
                                 // tzn. przydzielenie pamięci zmiennej IW

struct {                         // deklaracja anonimowej struktury instruction
        logic [31:0] a, b;
        logic [ 7:0] opcode;
        logic [23:0] address;
} instruction;
```

6.1.2. Szablony przypisania

Język SystemVerilog ma nową konstrukcję w porównaniu z językiem Verilog: *szablony przypisania* (*assignment patterns*). Są one używane do przypisywania wartości elementom struktur i tablic. Składnia szablonu przypisania jest podobna do składni przypisania listy wartości do tablicy w języku C, ale przed otwierającym nawiasem klamrowym jest dodany apostrof.

W instrukcjach przypisania szablon przypisania może znajdować się po prawej lub lewej stronie znaku równości. Jeśli miejsce docelowe znajduje się po lewej stronie znaku równości, każde wyrażenie składowe musi mieć typ danych strumienia bitów zgodny z miejscem docelowym i mieć taką samą liczbę bitów jak typ danych odpowiedniego elementu po prawej stronie.

Szablon przypisania jest opatrzony apostrofem i jest zbudowany z nawiasów klamrowych, *kluczy* (nazw zmiennych) i wyrażeń. Składnia szablonów przypisania jest następująca:

'{*expression* {, *expression*}}
'{*structure_pattern_key*: *expression* {, *structure_pattern_key*: *expression*}}
'{*array_pattern_key*: *expression* {, *array_pattern_key*: *expression*}}
'{*constant_expression* { *expression* {, *expression*}}}

gdzie: *expression* jest wyrażeniem; *structure_pattern_key* jest kluczem szablonu struktury; *array_pattern_key* jest kluczem szablonu tablicy; *constant_expression* jest wyrażeniem stałym.

Obowiązkowym elementem składni są zewnętrzne nawiasy klamrowe. Jako klucze szablonów można używać identyfikatora elementu struktury, wyrażenia stałego, typu prostego i słowa kluczowego **default**.

W przypadku szablonów przypisania struktur obowiązują następujące zasady:

- *member: value* – klucz nazwy, ustawia jawną wartość *value* dla elementu w strukturze o nazwie *member*;
- *type: value* – klucz typu, ustawia jawną wartość *value* dla każdego elementu w strukturze, którego typ jest taki sam jak typ *type*, i nie został ustawiony za pomocą klucza nazwy *member: value*;
- **default:** *value* – dotyczy elementów w strukturze, które nie pasują do kluczy nazwy lub kluczy typu.

Ponadto w przypadku szablonów przypisania do tablicy obowiązuje następująca zasada:

- *index: value* – klucz indeksu, definiuje jawną wartość dla elementu o podanym indeksie.

Na przykład:

```
var int A[N] = '{default:1};      // wszystkie elementy tablicy A otrzymują wartość 1

var integer i = '{31:1, 23:1, 15:1, 8:1, default:0}; /* 32-bitowy wektor o wartości 1 w bitach
                                                        31, 23, 15 i 8, w pozostałych – 0 */
typedef struct {real r, th;} C; // deklaracja struktury C z elementami r i th
var C x = '{th:PI/2.0, r:1.0};              /* deklaracja zmiennej typu C z inicjalizacją
                                               zmiennych: th=PI/2.0, r=1.0 */

var real y [0:1] = '{0.0, 1.1}, z [0:9] = '{default: 3.1416};      // deklaracja dwóch tablic y i z
```

W ostatnim przykładzie zadeklarowano dwie tablice liczb rzeczywistych y i z. Tablica y ma dwa elementy, które są inicjalizowane wartościami 0.0 i 1.1. Tablica z ma 10 elementów, wszystkie zostały zainicjowane pojedynczą wartością 3,1416.

Można również stosować bezkluczową notację pozycyjną. Na przykład:

```
var int B[4] = '{a, b, c, d};      /* zadeklarowano tablicę B składającą się z czterech zmiennych
                                      typu int, które są inicjalizowane wartościami a, b, c, d */

var C y = '{1.0, PI/2.0};          /* deklarowana jest instancja struktury C
                                      z inicjalizacją elementu: r = 1.0, th = PI/2.0 */

'{a, b, c, d} = B;                 // szablon wartości na lewo od znaku równości
```

6.1.3. Przypisywanie wartości do struktur

Elementy struktury mogą być inicjalizowane w momencie tworzenia instancji struktury za pomocą zestawu wartości szablonu przypisania zawartego między znacznikami '{ i }. Liczba wartości w nawiasach musi dokładnie odpowiadać liczbie elementów w strukturze. Na przykład:

```
typedef struct {                   // deklaracja struktury instruction_word_t
        logic [31:0] a, b;
        logic [ 7:0] opcode;
        logic [23:0] address;
} instruction_word_t;

instruction_word_t IW = '{100, 3, 8'hFF, 0};   // tworzenie instancji struktury
                                               // z przypisaniem wartości początkowych
```

Ostatni wiersz kodu tworzy instancję *IW* struktury instruction_word_t z następującymi wartościami początkowymi przypisanymi do elementów tej struktury: *a* = 100; *b* = 3; *opcode* = 8'hFF; *address* = 0.

W języku SystemVerilog istnieją trzy sposoby przypisywania wartości do elementów struktury.

W pierwszej metodzie wartość można przypisać do dowolnego elementu struktury, odwołując się do nazwy elementu. Na przykład:

```
IW.a = 100;                        // przypisywanie wartości do elementów struktury
IW.b = 5;
IW.opcode = 8'hFF;
IW.address = 0;
```

Drugi sposób wygląda następująco: całej strukturze można przypisać *wyrażenie strukturalne* (*structural expression*) zawarte między znacznikami '{ i } szablonu przypisania, tak jak w przypadku inicjalizacji struktury. W tym przypadku wartości są przypisywane do elementów struktury *w kolejności ich zadeklarowania*, czyli *według pozycji*. Na przykład:

```
IW = '{100, 5, 8'hFF, 0};
```

Trzecia metoda polega na tym, że w wyrażeniu strukturalnym wartości mogą być określone *przez nazwę elementu*, przy czym nazwa struktury i wartość są oddzielone dwukropkiem. Jeśli podano nazwy elementów, lista nazw elementów i odpowiadających im wyrażeń może następować w dowolnej kolejności. Na przykład:

```
IW = '{address:0, opcode:8'hFF, a:100, b:5};
```

Nie należy jednak mylić przypisywania wartości do elementów struktury według pozycji i według nazwy. Na przykład:

```
IW = '{address:0, 8'hFF, 100, 5};              // błąd !
```

Istnieje także możliwość przypisania elementom struktury wartości domyślnych. Dla wszystkich elementów struktury można przypisać wartość domyślną za pomocą słowa kluczowego **default**. Na przykład:

```
IW = '{default:0};      // wszystkie elementy instancji IW struktury mają przypisaną wartość 0
```

Wartość domyślną można również podać dla elementów struktury określonego typu tylko za pomocą słowa kluczowego tego typu. Słowo kluczowe **default** lub słowo kluczowe typu jest oddzielone od wartości dwukropkiem. Na przykład:

```
typedef struct {                    // deklaracja struktury word_t
        real r0, r1;
        int i0, i1;
        logic [ 7:0] opcode;
        logic [23:0] address;
} word_t;

word_t IW;                          // tworzenie instancji IW struktury instruction_word_t

IW = '{ real:1.0, default:0 };      // przypisywanie wartości domyślnych do elementów IW
```

W powyższym przykładzie wszystkim elementom struktury typu **real** przypisano wartość domyślną 1.0, a pozostałym elementom struktury przypisano wartość domyślną 0.

W języku SystemVerilog istnieje priorytet przypisywania wartości domyślnych do elementów struktury. Słowo kluczowe **default** ma najniższy priorytet i zostanie zastąpione przez dowolny typ domyślny. Wartości domyślne dla danego typu zostaną zastąpione przez przypisanie wartości do elementu według jego nazwy. Na przykład:

```
IW = '{ real:1.0, default:0, r1:3.1415 };
```

W tym przykładzie wyrażeniu struktury przypisano wartość 1.0 do elementu r0, wartość 3.1415 do elementu r1 oraz wartość 0 do wszystkich pozostałych elementów struktury.

Domyślnych wartości początkowych nie można jednak przypisać do elementów struktur spakowanych ani do struktur rozpakowanych zawierających związki.

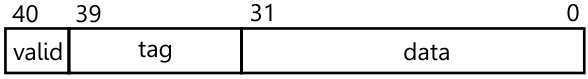
6.1.4. Struktury rozpakowane i spakowane

Struktury są domyślnie *rozpakowane* (*unpacked*) i mogą zawierać dowolny typ danych. Elementy rozpakowanej struktury są traktowane jako niezależne zmienne lub stałe zgrupowane pod wspólną nazwą. Język SystemVerilog nie określa, w jaki sposób narzędzia programistyczne powinny przechowywać rozpakowane elementy struktury w pamięci.

Struktura spakowana (*packed structure*) jest definiowana za pomocą słowa kluczowego **packed** i składa się z pól bitowych, które są umieszczone w pamięci bez przerw i traktowane jako jeden wektor. Pierwszym elementem struktury spakowanej jest najbardziej znaczące (lewe) pole wektora. Wysunięty najbardziej na prawo bit ostatniego elementu struktury jest najmniej znaczącym bitem wektora i jest numerowany jako bit 0. Na przykład:

```
struct packed {                 // deklaracja struktury spakowanej
        logic valid;            // najbardziej znaczący bit
        logic [7:0] tag;
        logic [31:0] data;      // data[0] jest najmniej znaczącym bitem
} data_word;
```

Położenie struktury data_word w pamięci pokazano na rys. 6.1.



RYS. 6.1. Położenie w pamięci spakowanej struktury data_word

ŹRÓDŁO: oprac. własne.

Do elementów struktury spakowanej można się odwoływać albo przez nazwę elementu, albo przez wybranie części wektora reprezentowanej przez tę strukturę. Poniższe dwa przypisania spowodują przypisanie wartości 8'hf0 do elementu *tag* struktury data_word:

```
data_word.tag = 8'hf0;          // odwołuje się do elementu tag w strukturze data_word

data_word[39:32] = 8'hf0;       // to samo, ale przez wybranie części wektora
```

Struktury spakowane mogą zawierać tylko wartości całkowite. Przypomnijmy, że wartość całkowitą można przedstawić w postaci wektora. Innymi słowy, elementami struktury spakowanej mogą być typy całkowite oraz wektory tworzone za pomocą typów **bit** i **logic**. Oznacza to, że spakowana struktura nie może zawierać zmiennych **real** ani **shortreal**, rozpakowanych struktur, rozpakowanych związków ani rozpakowanych tablic.

Ponieważ struktura spakowana jest przechowywana jako wektor, wszystkie operacje na strukturze spakowanej są traktowane jako operacje wektorowe. Dlatego operacje matematyczne i logiczne, a także wszelkie inne operacje, które można wykonywać na wektorach, można również wykonywać na strukturach spakowanych. Na przykład:

```
data_word packet_in, packet_out;   // tworzenie dwóch instancji struktury data_word

packet_out <= packet_in << 2;      // wykonywanie operacji wektorowych
                                   // na strukturach spakowanych
```

Zwróćmy uwagę, że gdy do spakowanej struktury przypisana jest lista wartości pomiędzy znacznikami '{ i }, jak opisano w punkcie 6.1.3, wartości z listy są przypisywane elementom struktury. Struktura spakowana jest w tym przypadku traktowana tak samo jak struktura rozpakowana, a nie jak wektor. Wartości w nawiasach klamrowych „{}" są oddzielnymi wartościami dla każdego elementu struktury, a nie konkatenacją wartości. Na przykład wpis

```
packet_in = '{1, '1, 1024};
```

przypisuje 1 do elementu *valid*, FF (szesnastkowo) do elementu *tag* oraz 1024 (dziesiętnie) do elementu *data* (przypomnijmy, że w SystemVerilog wyrażenie '1 przypisuje wszystkie jedynki do wektora).

Struktury spakowane mogą być deklarowane przy użyciu słów kluczowych **signed** lub **unsigned**. Modyfikatory te wpływają na sposób postrzegania całej struktury, gdy jest ona używana jako wektor w operacjach matematycznych lub relacyjnych. Nie mają one wpływu na to, jak postrzegane są elementy struktury. Każdy element

struktury jest traktowany jako ze znakiem lub bez znaku, zgodnie z deklaracją typu tego elementu. Wybór części spakowanej struktury jest zawsze wartością bez znaku. Na przykład:

```
typedef struct packed signed {      // deklaracja struktury spakowanej ze znakiem
                                    // jako typ użytkownika data_word_t

        logic valid;
        logic [ 7:0] tag;
        logic signed [31:0] data;
} data_word_t;

data_word_t A, B;                   // deklaracja zmiennych typu data_word_t

always @(posedge clock)
if ( A < B )                        // porównanie znakowe
```

Nie jest dozwolone określanie znakowości struktur rozpakowanych. Na przykład poniższa deklaracja jest uważana za niedopuszczalną:

```
typedef struct signed {       // błąd!
        int f1 ;
        logic f2 ;
} illegal_type;
```

Może pojawić się naturalne pytanie: jak wygląda sytuacja z wartościami logicznymi poszczególnych bitów w strukturze spakowanej, gdy pewne jej elementy są typami o czterech stanach, a inne o dwóch stanach?

Jeśli wszystkie typy danych w spakowanej strukturze są wektorami dwustanowymi, to cała struktura jest uważana za wektor dwustanowy. Jeśli jakikolwiek typ danych w spakowanej strukturze ma cztery stany, to cała struktura jest traktowana jako wektor z czterema stanami. Jeśli struktura zawiera również elementy o dwóch stanach, to odczytanie tych elementów powoduje ich niejawną konwersję z typu o czterech stanach na typ o dwóch stanach, a zapisanie ich powoduje ich niejawną konwersję z typu o dwóch stanach na typ o czterech stanach.

6.1.5. Przekazywanie struktur przez porty modułów

Struktury można przekazywać przez porty modułu i interfejsu. W tym celu strukturę należy zdefiniować jako typ użytkownika za pomocą słowa kluczowego **typedef**. Na przykład:

```
package definitions;            // definicja struktury w pakiecie definitions
    typedef struct {            // struktura jako typ użytkownika instruction_word_t
        logic [31:0] a, b;
        opcode_t opcode;
        logic [23:0] address;
        logic error;
    } instruction_word_t;
endpackage

module alu                      // deklaracja modułu alu
    (input definitions:: instruction_word_t IW,
    input wire clock);
    ...
endmodule
```

W powyższym przykładzie struktura typu użytkownika instruction_word_t jest zadeklarowana w pakiecie definitions. Instancja struktury *IW* jest następnie przekazywana do modułu *alu* jako port wejściowy.

Gdy rozpakowana struktura przechodzi przez port modułu, po każdej stronie portu musi być podłączona struktura tego samego typu. Struktury anonimowe (bez słowa kluczowego **typedef**) zadeklarowane w dwóch różnych modułach, nawet jeśli mają taką samą nazwę, elementy i nazwy elementów, nie są tym samym typem strukturalnym. Dlatego też przez porty modułu nie można przekazywać anonimowych struktur.

6.1.6. Przekazywanie struktur jako argumenty do zadań i funkcji

Struktury mogą być przekazywane jako argumenty do zadań lub funkcji. W tym celu strukturę należy określić jako typ zdefiniowany przez użytkownika za pomocą słowa kluczowego **typedef**. Na przykład:

```
module processor (...);
    ...
    typedef enum {ADD, SUB, MULT, DIV} opcode_t;

    typedef struct {            // deklaracja lokalnego typu użytkownika
        logic [31:0] a, b;
        opcode_t opcode;
        logic [23:0] address;
        logic error;
    } instruction_word_t;
```

```
        function alu (input instruction_word_t IW); // przekazywanie struktury IW
                                                    // jako argument funkcji alu
        ...
        endfunction
endmodule
```

Podczas wywoływania zadania lub funkcji, której formalnym argumentem jest struktura rozpakowana, należy przekazać strukturę dokładnie tego samego typu. Struktury anonimowe nie mogą być przekazywane do zadań bądź funkcji jako argumenty.

6.2. Unie

Unia (*union*) to typ danych reprezentujący pojedynczy fragment pamięci, który może być różnie interpretowany dla różnych typów danych. W danym momencie może być używany tylko jeden typ danych zdefiniowany w unii. Domyślnie unia jest *rozpakowana* (*unpacked*), tzn. nie ma żadnej reprezentacji tego, jak elementy związku są przechowywane w pamięci. Typy dynamiczne i **chandle** mogą być używane tylko w *uniach oznaczonych* (*tagged unions*).

Deklaracja unii ma następującą składnię:

union [**tagged**] [**packed**] [*signing*] {*union_member*; ...} *union_name*;

gdzie: **tagged** jest słowem kluczowym określającym unię oznaczoną; **packed** jest słowem kluczowym określającym unię spakowaną; *signing* wskazuje na obecność znaku w unii (**signed** lub **unsigned**); *union_member* jest elementem unii; *union_name* jest nazwą unii.

Konstrukcje [**tagged**], [**packed**] i [*signing*] są opcjonalne. Unie oznaczone mogą być uznane za nieokreślone za pomocą słowa kluczowego **void**.

Unia zdefiniowana jako typ użytkownika za pomocą słowa kluczowego **typedef** jest nazywana *unią wprowadzoną* (*typed union*). Jeśli nie zostanie użyte słowo kluczowe **typedef**, unia jest nazywana *unią anonimową* (*anonymous union*). Na przykład:

```
typedef union { int i; shortreal f; } num;      // wprowadzono unię jako typ
                                                // użytkownika num
num n;                                          // deklaracja zmiennej n typu num
n.f = 0.0;                                      // ustawianie n na format zmiennoprzecinkowy

typedef struct {
        bit isfloat;
```

```
        union { int i; shortreal f; } n;  // anonimowy typ unii w ramach typu
                                          // użytkownika jako struktura
} tagged_st;
```

6.2.1. Unie rozpakowane i spakowane

Domyślnie unia jest traktowana jako *rozpakowana* (*unpacked*). Słowo kluczowe **packed** jest używane do deklarowania unii *spakowanej* (*packed*).

Unie rozpakowane nie podlegają syntezie. Są one typem abstrakcyjnym, który jest przydatny w modelach systemów wysokiego poziomu i transakcyjnych. W tych modelach użyteczne może być przechowywanie związku dowolnego typu, w tym typów czterostanowych, dwustanowych i niesyntezowalnych, takich jak typ **real**.

Unia spakowana (*packed*) może zawierać tylko elementy typu danych całkowitych i typów danych spakowanych. Elementy unii spakowanej muszą mieć ten sam rozmiar, dlatego można je zapisywać jako jeden typ danych, a odczytywać jako inny typ danych. Unie spakowane są syntetyzowalne (*synthesizable*). Nie mogą zawierać zmiennych typu **real** lub **shortreal**, rozpakowanych struktur, rozpakowanych unii ani rozpakowanych tablic.

W SystemVerilog spakowana unia jest traktowana jako pojedynczy wektor. Można jej używać jako całości w wyrażeniach z operacjami arytmetycznymi i logicznymi. Jeden lub więcej bitów spakowanej unii można wybrać tak jak w wektorze bitów. Spakowana unia może być również traktowana jako ze znakiem lub bez znaku (domyślnie).

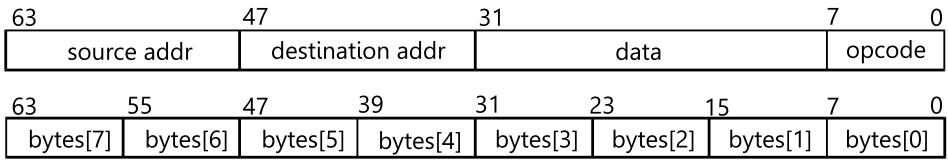
Unia spakowana umożliwia zapisywanie danych w jednym formacie i odczytywanie ich w innym formacie. Jeśli zawiera ona składowe z dwoma stanami i składowe z czterema stanami, to cała unia jest czterowartościowa. Podczas odczytu następuje niejawna konwersja z typu o czterech stanach na typ o dwóch stanach, a podczas zapisu składowej o dwóch stanach również następuje niejawna konwersja z typu o dwóch stanach na typ o czterech stanach.

Poniższy przykład definiuje unię spakowaną, w której wartość może być reprezentowana na dwa sposoby: jako pakiet danych (przy użyciu struktury spakowanej) lub jako tablica bajtów.

```
typedef struct packed {              // deklarowanie pakietu danych jako struktury spakowanej
        logic [15:0] source_address;
        logic [15:0] destination_address;
        logic [23:0] data;
        logic [ 7:0] opcode;
} data_packet_t;

union packed {                       // unia pakietu danych i tablicy bajtów
        data_packet_t packet;        // struktura spakowana
```

```
    logic [7:0][7:0] bytes;          // tablica spakowanych bajtów
} dreg;
```

Rysunek 6.2 przedstawia reprezentację unii *dreg* w pamięci.



RYS. 6.2. Reprezentacja unii *dreg* w pamięci

ŹRÓDŁO: oprac. własne.

Do elementów unii można dostać się, używając różnych szerokości dostępu. W naszym przykładzie oznacza to, że wartość można zapisać przy użyciu tablicy w formacie *bytes*, a następnie tę samą wartość można odczytać przy użyciu formatu *packet*. Na przykład:

```
always @(posedge clock, negedge resetN)
    if (!resetN) begin
        dreg.packet <= '0;          // zapis w formacie packet
        i <= 0;
    end
    else if (load_data) begin
        dreg.bytes[i] <= byte_in;   // zapis w formacie bytes
        i <= i + 1;
    end

always @(posedge clock)
    if (data_ready)
        case (dreg.packet.opcode)   // odczyt w formacie packet
    ...
```

6.2.2. Unie oznaczone

Unia oznaczona (*tagged union*) jest deklarowana za pomocą słowa kluczowego **tagged**. Na przykład:

```
union tagged {      // unia oznaczona
    int i;          // liczba całkowita
    real r;         // liczba rzeczywista
} data;
```

Unia oznaczona przechowuje zarówno wartość elementu, jak i *znacznik* (*tag*), czyli dodatkowe bity reprezentujące bieżącą nazwę ostatniego elementu unii, w którym wartość została zapisana.

Unia normalna może być aktualizowana przy użyciu wartości elementu jednego typu i odczytywana jako wartość elementu innego typu. Istnieje luka polegająca na zapisywaniu danych jednego typu i interpretowaniu ich jako danych innego typu. Unie oznaczone służą więc jako zabezpieczenie przed przypadkowym błędnym odczytem.

Znacznik i wartość elementu są aktualizowane tylko razem. Wartość elementu można odczytać jedynie za pomocą typu, który odpowiada bieżącej wartości znacznika, tj. nazwie elementu. Nie jest więc możliwe przechowywanie wartości jednego typu i interpretowanie tych bitów jako innego typu.

Wartość można zapisać do elementu unii oznaczonej za pomocą *wyrażenia oznaczonego* (*tagged expression*). Zawiera ono słowo kluczowe **tagged**, po którym następuje nazwa elementu, a następnie wartość, która ma zostać zapisana. Do nazwy unii przypisywane jest wyrażenie oznaczone. Na przykład:

```
data = tagged i 5;      // zapis wartości 5 w data.i i niejawne ustawienie znacznika
```

Wartości z unii oznaczonej są odczytywane przy użyciu nazwy elementu unii:

```
d_out = data.i;         // odczytuje wartość elementu i z unii data
```

Jeśli z unii oznaczonej zostanie odczytany element innego typu niż ten, z którym zostało zapisane ostatnie oznaczone wyrażenie, narzędzie programowe wygeneruje błąd:

```
d_out = data.r;         // Błąd: element nie pasuje do znacznika
```

Po przypisaniu wartości do unii oznaczonej przy użyciu wyrażenia znacznika inne wartości mogą być zapisywane do tego samego elementu związku przy użyciu nazwy elementu:

```
data.i = 7;             // zapis wartości do elementu unii
                        // nazwa elementu jest taka sama jak bieżącego znacznika
```

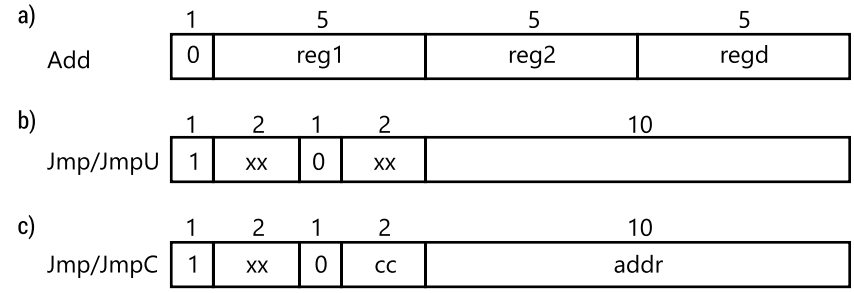
Jeśli podana nazwa elementu nie pasuje do bieżącego znacznika unii, rejestrowany jest błąd w czasie wykonywania.

Stosowanie nazw elementów unii oznaczonych oprócz tego, że chroni przed przypadkowym odczytaniem, sprawia, że kod jest prostszy i łatwiejszy do odczytania.

Unie oznaczone można wykorzystać do reprezentowania rozkazów procesora o różnych formatach, jak pokazano w poniższym przykładzie:

```
typedef union tagged {
        struct {
                bit [4:0] reg1, reg2, regd;
        } Add;                          // rozkaz dodawania, 15 bitów + 1 bit znacznika
        union tagged {
                bit [9:0] JmpU;         // rozkaz skoku JmpU, 12 bitów + 1 bit znacznika
                struct {
                        bit [1:0] cc;
                        bit [9:0] addr;
                } JmpC;                 // rozkaz skoku JmpC, 12 bitów + 1 bit znacznika
        } Jmp;                          // rozkaz skoku Jmp
} Instr;                                // unia wszystkich rozkazów
```

W pamięci unia *Instr* wygląda tak, jak pokazano na rys. 6.3.



RYS. 6.3. Reprezentacja unii Instr w pamięci: a) dla rozkazu Add, b) dla rozkazu JmpU, c) dla rozkazu JmpC

ŹRÓDŁO: oprac. własne.

W powyższym przykładzie struktura Add i instrukcja skoku Jmp mają wspólne 15 bitów danych i 1 bit znacznika, które je rozróżniają. Z kolei unia Jmp składa się z dwóch elementów, JmpU i JmpC, które współdzielą 12 bitów danych (JmpU zajmuje tylko 10 bitów) oraz 1 bit własnego znacznika, służącego do rozróżniania poleceń JmpU i JmpC.

Należy zauważyć, że w tym przykładzie użyto dwóch znaczników. Jeden jest używany do odróżnienia struktury Add od innej unii oznaczonej Jmp, drugi zaś do odróżnienia wektora bitowego JmpU od struktury JmpC w unii oznaczonej Jmp.

Unię oznaczoną można zadeklarować jako spakowaną. W tym przypadku jej elementy mogą mieć różne rozmiary, ale muszą być typami całkowitymi. Na przykład:

```
union tagged packed {       // spakowana unia oznaczona
logic [15:0] short_word;    // element 16-bitowy
logic [31:0] word;          // element 32-bitowy
logic [63:0] long_word;     // element 64-bitowy
} data_word;
```

Unie spakowane są syntetyzowalne. Unie spakowane oznaczone nie są syntezowalne w niektórych kompilatorach.

6.3. Przykład użycia struktur i unii

Nasze rozważania na temat struktur i unii w SystemVerilog podsumujemy prostym przykładem. Struktury zapewniają mechanizm grupowania powiązanych danych pod wspólną nazwą. Do każdej części danych można odwoływać się indywidualnie, podając jej nazwę, lub do całej grupy jako całości. Dzięki grupom jeden element pamięci można wykorzystać na kilka sposobów. Przyjrzyjmy się, jak można to wykorzystać w praktyce.

Poniższy przykład symuluje prostą *jednostkę arytmetyczno-logiczną* (*arithmetic logic unit* – ALU), która może operować na wartościach ze znakiem lub bez znaku. Są one przekazywane do ALU jako instrukcje – struktury. ALU może operować na wartościach ze znakiem lub bez znaku, ale nie na obu jednocześnie. Dlatego wartości te są modelowane jako unie tych dwóch typów. Dzięki temu jedna zmienna może reprezentować zarówno wartości ze znakiem, jak i bez znaku.

Przykład 6.1. Przykład użycia struktur i unii

```
package definitions;                          // pakiet z deklaracjami
    typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;   // typ kodu operacji

    typedef enum {UNSIGNED, SIGNED} operand_type_t;      // typ operandu

    typedef union packed {                    // typ danych jako unia
        logic [31:0] u_data;                  // bez znaku
        logic signed [31:0] s_data;           // ze znakiem
    } data_t;

    typedef struct packed {                   // typ polecenia jako struktura
        opcode_t opc;                         // kod operacji
        operand_type_t op_type;               // typ operandów
        data_t op_a;                          // operand a
        data_t op_b;                          // operand b
    } instr_t;
endpackage

import definitions::*; // import pakietu z deklaracjami do jednostki kompilacji
```

```
module alu                          // moduł ALU
        (input instr_t IW,          // polecenie do wykonania
        output data_t alu_out);     // wynik

        always @(IW) begin                      // wykonanie polecenia
                if (IW.op_type == SIGNED) begin // wykonywanie operacji ze znakiem
                case (IW.opc)
                        ADD : alu_out.s_data = IW.op_a.s_data + IW.op_b.s_data;
                        SUB : alu_out.s_data = IW.op_a.s_data - IW.op_b.s_data;
                        MULT: alu_out.s_data = IW.op_a.s_data * IW.op_b.s_data;
                        DIV : alu_out.s_data = IW.op_a.s_data / IW.op_b.s_data;
                        SL : alu_out.s_data = IW.op_a.s_data <<< 2;
                        SR : alu_out.s_data = IW.op_a.s_data >>> 2;
                        default : alu_out.s_data = 'x;
                endcase
        end
        else begin                              // wykonywanie operacji bez znaku
                case (IW.opc)
                        ADD : alu_out.u_data = IW.op_a.u_data + IW.op_b.u_data;
                        SUB : alu_out.u_data = IW.op_a.u_data - IW.op_b.u_data;
                        MULT: alu_out.u_data = IW.op_a.u_data * IW.op_b.u_data;
                        DIV : alu_out.u_data = IW.op_a.u_data / IW.op_b.u_data;
                        SL : alu_out.u_data = IW.op_a.u_data << 2;
                        SR : alu_out.u_data = IW.op_a.u_data >> 2;
                        default : alu_out.u_data = 'x;
                endcase
        end
        end
endmodule
```

6.4. Wnioski

Struktura (*structure*) to zbiór różnych typów danych, do których można się odwoływać za pomocą nazwy jako do pojedynczej całości, jak również do każdego elementu struktury.

Struktura różni się od tablicy tym, że tablica jest zbiorem elementów tego samego typu i rozmiaru, natomiast struktura jest zbiorem zmiennych lub stałych, które mogą być różnego typu i rozmiaru. Do elementów tablicy odwołujemy się za pomocą indeksu, a do elementów struktury za pomocą nazwy elementu.

Domyślnie struktura jest traktowana jak zmienna. Strukturę można zadeklarować jako sieć, używając słów kluczowych określających jej typ.

Wartość może być przypisana do dowolnego elementu struktury. Całej strukturze można przypisać *wyrażenie strukturalne* za pomocą szablonu przypisania. W wyrażeniu strukturalnym wartości mogą być również określone przez pozycję lub nazwę elementu. Nie wolno mieszać przypisywania wartości do elementów struktury według pozycji i według nazwy.

Istnieje także możliwość przypisania elementom struktury wartości domyślnych. W tym celu używa się słowa kluczowego **default** lub słowa kluczowego typu elementów. Domyślnych wartości początkowych nie można przypisywać elementom struktur spakowanych i rozpakowanych, które zawierają unie.

Rozpakowana struktura może zawierać dowolny typ danych. Jej elementy są traktowane jako niezależne zmienne lub stałe, zgrupowane pod wspólną nazwą.

Struktura spakowana jest definiowana za pomocą słowa kluczowego **packed** i składa się z pól bitowych, które są umieszczone w pamięci bez przerw i traktowane jako pojedynczy wektor. Domyślnie struktury są rozpakowane.

Do elementów struktury spakowanej można się odwoływać albo przez nazwę elementu, albo przez wybór części wektora.

Struktury spakowane mogą zawierać tylko wartości całkowite, nie mogą zaś zawierać zmiennych typu **real** ani **shortreal**, rozpakowanych struktur, rozpakowanych związków ani rozpakowanych tablic.

Wszystkie operacje dozwolone na wektorach mogą być wykonywane na strukturach spakowanych.

Struktury spakowane mogą być zadeklarowane ze znakiem lub bez znaku, zależy to od ich typu. Obecności znaku nie można określić dla struktur rozpakowanych.

Jeśli wszystkie typy danych w spakowanej strukturze mają dwa stany, to cała struktura jest traktowana jako wektor z dwoma stanami. Jeśli jakikolwiek typ danych w spakowanej strukturze ma cztery stany, to cała struktura jest traktowana jako wektor z czterema stanami. Jeśli struktura zawiera również elementy o dwóch stanach, to odczytanie tych elementów powoduje ich niejawną konwersję z typu o czterech stanach na typ o dwóch stanach, a zapisanie ich do typu o dwóch stanach powoduje ich niejawną konwersję z typu o dwóch stanach na typ o czterech stanach.

Struktury mogą być przekazywane przez porty modułów i interfejsów oraz jako argumenty zadań i funkcji. W tym celu strukturę należy określić jako typ zdefiniowany przez użytkownika za pomocą słowa kluczowego **typedef**.

Unia (*union*) to typ danych reprezentujący pojedynczy fragment pamięci, który może być różnie interpretowany w przypadku różnych typów danych. W danym momencie może być używany tylko jeden typ danych zadeklarowany w unii.

Unie są domyślnie rozpakowane, nie są syntetyzowane. Słowo kluczowe **packed** jest używane do deklarowania unii spakowanych, które są syntetyzowalne.

Unie spakowane mogą zawierać tylko elementy typów całkowitych i spakowanych, które muszą mieć ten sam rozmiar. Elementy unii spakowanej mogą być zapisywane jako jeden typ danych i odczytywane jako inny typ danych.

Unia spakowana jest traktowana jako pojedynczy wektor. Można jej używać jako całości w wyrażeniach z operacjami arytmetycznymi i logicznymi. Jeden bit lub więcej bitów spakowanej unii można wyodrębniać tak jak w wektorze bitów. Unia spakowana może być również traktowana jako ze znakiem lub bez znaku (domyślnie).

Unię można określić jako typ zdefiniowany przez użytkownika za pomocą słowa kluczowego **typedef**. Wówczas są one nazywane wprowadzonymi, w przeciwnym razie są traktowane jako anonimowe.

Jeśli unia spakowana zawiera składowe z dwoma stanami i składowe z czterema stanami, to cała unia jest czterowartościowa. Podczas odczytu następuje niejawna konwersja z typu z czterema stanami na typ z dwoma stanami, a podczas zapisu składowej z dwoma stanami następuje niejawna konwersja z typu z dwoma stanami na typ z czterema stanami.

Unia oznaczona jest deklarowana za pomocą słowa kluczowego **tagged**. Przechowuje ona wartość elementu oraz *znacznik* (*tag*), tj. dodatkowe bity reprezentujące bieżącą nazwę ostatniego elementu, w którym przechowywana była wartość.

Unie oznaczone służą do ochrony przed przypadkowym błędnym odczytem, gdy dane są zapisywane jako jeden typ, a odczytywane jako inny typ.

Wartość jest zapisywana do unii oznaczonej przy użyciu wyrażenia oznaczonego. Zawiera ono słowo kluczowe **tagged**, po którym następuje nazwa elementu, a następnie przechowywana wartość.

Wartości z unii oznaczonej są odczytywane przy użyciu nazwy elementu unii. Jeśli z unii oznaczonej zostanie odczytany element innego typu niż ten, z którym zostało zapisane ostatnie oznaczone wyrażenie, kompilator wygeneruje błąd.

Unie oznaczone można zadeklarować jako spakowane za pomocą słowa kluczowego **packed**. W tym przypadku jej elementy mogą mieć różne rozmiary, ale muszą mieć typy całkowite.

Unie spakowane oznaczone nie są syntetyzowalne za pomocą niektórych kompilatorów.

7. Tablice

Język Verilog umożliwia tworzenie wielowymiarowych tablic zmiennych i sieci, a także niesyntezowalnych tablic *zdarzeń* (*event*). W porównaniu z językiem Verilog znacznie rozszerza on możliwości pracy z tablicami.

Tablice w SystemVerilog, podobnie jak struktury i unie, mogą być s*pakowane* (*packed*) i *rozpakowane* (*unpacked*). Spakowane obiekty są umieszczane w pamięci bez przerw i traktowane jako pojedynczy wektor. Z tablicami spakowanymi można postępować w taki sam sposób, jak z wektorami.

Język SystemVerilog udostępnia różne sposoby inicjalizacji tablic: przez proste przypisanie do tablic spakowanych przy użyciu szablonów oraz domyślnie przy użyciu słowa kluczowego **default**. Zawartość tablic i struktur w SystemVerilog może być wzajemnie kopiowana za pomocą *konwersji strumienia bitów*.

Oprócz tradycyjnych operacji odczytu i zapisu elementów tablicy, fragmentów tablicy i całej tablicy język SystemVerilog pozwala na wykonywanie operacji odczytu i zapisu fragmentu zmiennego tablicy, porównywania całej tablicy lub fragmentów tablicy, konkatenacji tablic, a także przypisywania liczby całkowitej do tablicy spakowanej i traktowania jej w wyrażeniach jako liczby całkowitej.

W języku SystemVerilog tablice, jak również struktury i unie mogą być przekazywane przez porty modułów lub jako argumenty zadań i funkcji.

Tablice mogą zawierać struktury i unie jako elementy. Struktury i unie mogą z kolei być elementami tablic.

Język SystemVerilog udostępnia wiele narzędzi do pracy z tablicami. Ma specjalny operator **foreach** służący do wykonywania pętli w obrębie elementów tablicy.

Przypisywanie wartości do elementów tablic i struktur odbywa się za pomocą specjalnej konstrukcji zwanej *szablonem przypisania*, która pozwala przypisać wartość do każdego elementu, do elementów określonego typu danych oraz określić wartość domyślną. Ponadto język SystemVerilog udostępnia wiele funkcji systemowych i metod służących do obsługi tablic podobnych do tych używanych w języku C.

Język SystemVerilog udostępnia także inne typy tablic niesyntezowalnych: dynamiczne (*dynamic arrays*), asocjacyjne (*associative arrays*) i kolejki (*queues*).

Można zauważyć, że tablice, struktury i unie oraz wszystko, co jest z nimi związane, są potężnymi narzędziami do tworzenia dużych i złożonych projektów.

7.1. Tablice w języku SystemVerilog

Tablice w języku SystemVerilog, podobnie jak struktury i unie, mogą być spakowane lub rozpakowane. Jednak typ tablicy (spakowana lub rozpakowana) nie jest określany przez słowo kluczowe, lecz przez pozycję zakresu w stosunku do nazwy w deklaracji tablicy.

W języku SystemVerilog zakres w deklaracji tablicy może być przed jej nazwą lub za nazwą. Jeśli zakres znajduje się przed nazwą tablicy, to jest ona *tablicą spakowaną* (*packed array*) lub *wektorem* (*vector*) i jest reprezentowana w pamięci jako ciągła sekwencja bitów. Na przykład:

```
bit [7:0] c;          // wektor bitów, rozmiar 8 bitów
                      // każdy bit ma 2 stany

logic [15:0] d;       // wektor bitów, rozmiar 16 bitów
                      // każdy bit ma 4 stany
```

Jeśli zakres występuje po nazwie tablicy w deklaracji tablicy, to jest ona *tablicą rozpakowaną* (*unpacked array*), czyli pewnym zbiorem elementów. Są one rozmieszczone w pamięci w sposób specyficzny dla danej implementacji, tzn. ich położenie jest określane przez kompilator, niekoniecznie w kolejności ustalonej w kodzie. Na przykład:

```
real u [7:0];         // rozpakowana tablica składająca się z 8 elementów typu real
```

Tablice, zarówno spakowane, jak i rozpakowane, mogą mieć wiele wymiarów. W tym przypadku mówimy o *tablicach wielowymiarowych*.

Tablice rozpakowane mogą być:

- *tablicami o stałym rozmiarze* (*fixed-size arrays*);
- *tablicami dynamicznymi* (*dynamic arrays*);
- *tablicami asocjacyjnymi* (*associative arrays*);
- *kolejkami* (*queues*).

Tablice rozpakowane mogą być tworzone z dowolnego typu danych, w tym z innych tablic spakowanych lub rozpakowanych.

7.1.1. Tablice spakowane

Tablica spakowana (*packed array*) to wektor bitów podzielony na *podpola* (*subfields*), do których można uzyskać dostęp jako do elementów tablicy. Gdy spakowana tablica jest przywoływana jako całość, jest traktowana jako pojedynczy wektor. Dlatego może być ona *ze znakiem* (***signed***) lub *bez znaku* (***unsigned***).

Poszczególne elementy tablicy są bez znaku, chyba że są to typy zdefiniowane przez użytkownika i zadeklarowane są ze znakiem. *Wybór części* (*part-select*) tablicy spakowanej jest zawsze bez znaku. Na przykład:

```
bit signed [31:0] a;        // wektor 32-bitowy,
                            // jest traktowany jako liczba całkowita ze znakiem

b = a [7:0];                // część wektora bitów a, liczba całkowita bez znaku
```

Tablice spakowane umożliwiają reprezentację typów danych całkowitych o dowolnej długości, np. 37 bitów, do których można zastosować arytmetykę 37-bitową.

Tablice spakowane mogą zawierać jednobitowe typy danych **bit**, **logic** i **reg** oraz rekursywnie inne tablice spakowane i struktury spakowane.

Przykłady tablic spakowanych:

```
logic [31:0] w;         /* tablica spakowana składająca się
                           z 32 elementów typu logic (wektor 32-bitowy) */

typedef bit [10:0] g;   // deklaracja typu g tablicy spakowanej o rozmiarze 11 bitów

g [5:0] g_aray;         // tablica spakowana g_aray składająca się z 6 tablic typu g
```

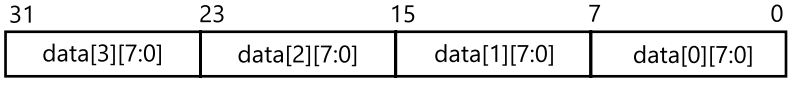
Tablice spakowane można również tworzyć z sieciowych typów (**wire**, **uwire**, **wand**, **tri**, **triand**, **trior**, **tri0**, **tri1** lub **trireg**). Na przykład:

```
wire [31:0] out;        // spakowana tablica elementów typu wire
wand [7:0] and_out;     // spakowana tablica elementów typu wand
```

W języku SystemVerilog dodano możliwość deklarowania wielu wymiarów w spakowanej tablicy:

```
logic [3:0] [7:0] data;         // dwuwymiarowa tablica spakowana
```

W ostatnim przykładzie zadeklarowano tablicę składającą się z czterech 8-bitowych *podtablic* (*sub-arrays*). Na rys. 7.1 pokazano, jak powyższa dwuwymiarowa tablica będzie przechowywana w pamięci.



RYS. 7.1. Reprezentacja dwuwymiarowej tablicy danych **logic** [3:0] [7:0] w pamięci

ŹRÓDŁO: oprac. własne.

Elementy tablicy spakowanej mogą być strukturami spakowanymi. Na przykład:

```
typedef struct packed {       // spakowana struktura typu my_struct

        bit [10:0] a;
        logic [7:0] b;
        reg [3:0] c;
} my_struct

my_struct [2:0] y;            /* jednowymiarowa spakowana tablica y,
                                 której elementami są trzy spakowane struktury my_struct */
```

Do tablicy spakowanej można odwoływać się jako do całości, jako do *selekcji bitów* (*bit-select*) lub jako do *wyboru części* (*part-select*). Do wielowymiarowych tablic spakowanych można również odwoływać się jako do *wycinka* (*slice*). Wycinek to jeden lub więcej ciągłych wymiarów tablicy. Na przykład:

```
logic [3:0][7:0] data;              // tablica dwuwymiarowa

wire [31:0] out = data;             // cała tablica

wire sign = data[3][7];             // wybór bitu

wire [3:0] nib = data [0][3:0];     // wybór części

byte high_byte;
assign high_byte = data[3];         // fragment 8-bitowy, taki sam jak
                                    // high_byte = data[3][7:0];

logic [15:0] word;
assign word = data[1:0];            // dwa fragmenty, tak samo jak
                                    // word = {data[1][7:0], data[0][7:0]};
```

Należy zauważyć, że predefiniowane typy całkowite **byte**, **shortint**, **int**, **longint**, **integer** i **time** są tablicami spakowanymi o określonej szerokości. Na przykład:

```
byte e;             // to samo co bit signed [7 : 0] e;

integer d;          // to samo co logic signed [31 : 0] d;
```

Ponieważ tablice spakowane są przechowywane jako wektory, wszelkie operacje dozwolone na wektorach mogą być wykonywane na tablicach spakowanych. Na przykład:

```
logic [3:0][15:0] a, b, result;          // tablice spakowane
...
result = (a << 1) + b;                   // operacje na tablicach spakowanych
```

Maksymalny rozmiar tablicy w języku SystemVerilog zależy od kompilatora, ale nie powinien być mniejszy niż 224 = 16 777 216.

7.1.2. Tablice rozpakowane

Tablice rozpakowane (*unpacked arrays*) mogą zawierać elementy dowolnego typu danych, tzn. nie ma ograniczeń co do typów danych tak jak w przypadku tablic spakowanych. Zakres adresów rozpakowanych elementów tablicy jest podawany w deklaracji po jej nazwie. Każdy wymiar rozpakowanej tablicy jest reprezentowany przez zakres adresów [msb:lsb]. Na przykład:

```
int m1 [0:7] [0:31];          // deklaracja rozpakowanej tablicy z użyciem zakresów
```

Również w SystemVerilog zakresy rozpakowanych tablic mogą być reprezentowane przez rozmiar [n], tzn. pojedynczą liczbę dodatnią n (jak w języku C), co jest równoważne zakresowi [0:n-1]. Na przykład:

```
int m1 [8] [32];      // taka sama deklaracja rozpakowanej tablicy
                      // z użyciem wymiarów
```

Należy pamiętać, że tablice spakowane nie mogą być deklarowane tylko przy użyciu rozmiaru. Na przykład poniższa deklaracja powoduje wystąpienie błędu:

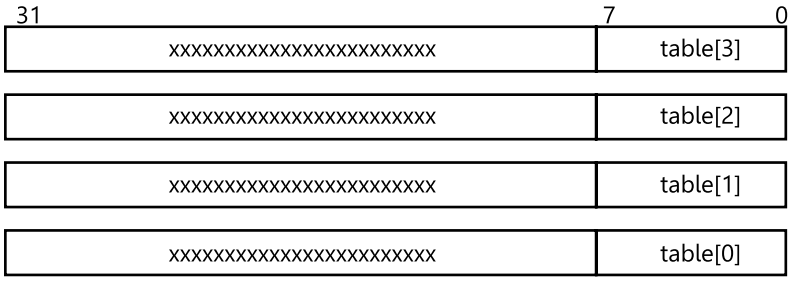
```
logic [7] data;       // Błąd!
```

Elementy rozpakowanej tablicy mogą być umieszczane w pamięci niezależnie od siebie. Na przykład następującą tablicę

```
wire [7:0] table [3:0];
```

można umieścić w pamięci w sposób przedstawiony na rys. 7.2.

Dokładne rozmieszczenie elementów tablicy rozpakowanej w pamięci zależy od kompilatora. Przykładowo między elementami mogą występować przerwy. Dlatego nie można traktować rozpakowanej tablicy jako wektora bitów. Jeśli tablica rozpakowana jest zadeklarowana ze znakiem, to odnosi się on do elementów tablicy, a nie do całej tablicy.



RYS. 7.2. Niezależne rozmieszczanie elementów tablicy rozpakowanej

ŹRÓDŁO: oprac. własne.

Język SystemVerilog rozszerza typy tablicy rozpakowanej o typ zdarzenia **event**, a także o typy **logic**, **bit**, **byte**, **int**, **longint**, **shortreal** i **real**. Elementy tablic rozpakowanych mogą być również strukturami (**struct**) i typami enumeratywnymi (**enum**), zadeklarowanymi jako typy własne za pomocą słowa kluczowego **typedef**. Na przykład:

```
bit [63:0] d_array [1:128];          // tablica wektorów 64-bitowych

shortreal cosines [0:89];            // tablica liczb rzeczywistych

typedef enum {Mo, Tu, We, Th, Fr, Sa, Su} Week;
Week Year [1:10];                    // tablica składająca się z 10 elementów typu
                                     // enumeratywnego Week
```

Język SystemVerilog pozwala na odwołanie się do całej rozpakowanej tablicy bądź do fragmentu kilku elementów w jej obrębie. Fragment to jeden lub więcej elementów ponumerowanych w jednym wymiarze tablicy. Umożliwia to kopiowanie zawartości całej tablicy lub jej określonego wymiaru do innej tablicy.

Aby skopiować wiele elementów bezpośrednio do rozpakowanej tablicy, położenie i typ elementu tablicy lub części tablicy po lewej stronie miejsca docelowego muszą dokładnie odpowiadać położeniu i typowi elementów po prawej stronie. Na przykład:

```
int a1 [7:0][1023:0];        // tablica rozpakowana
int a2 [1:8][1:1024];        // tablica rozpakowana

a2 = a1;                     // kopiowanie całej tablicy
a2[3] = a1[0];               // kopiowanie fragmentu tablicy, tak samo jak
                             // a2[3][1023:0]= a1[0][1:1024];
```

7.1.3. Inicjalizacja tablic podczas deklaracji

Spakowane tablice można zainicjować za pomocą prostego przypisania, tak jak wektory po ich zadeklarowaniu. Przypisać można stałą, *konkatenację* (*concatenation*) stałych lub *replikację* (*replication*) stałych. Na przykład:

```
logic [3:0][7:0] a = 32'h0;          // inicjalizacja spakowanej tablicy wektorem

logic [3:0][7:0] b = {16'hz,16'h0};  // inicjalizacja spakowanej tablicy
                                     // przez konkatenację

logic [3:0][7:0] c = {16{2'b01}};    // inicjalizacja spakowanej tablicy
                                     // przez replikację
```

Tablice rozpakowane po ich zadeklarowaniu można zainicjować dla każdego wymiaru tablicy za pomocą listy wartości szablonów przypisania ujętych w nawiasy klamrowe ‹{ i }. Przypisania wymagają również zagnieżdżone zestawy nawiasów klamrowych, które dokładnie odpowiadają wymiarom tablicy. Na przykład:

```
int d [0:1][0:3] = '{ '{7,3,0,5}, '{2,0,1,6} };
// d[0][0] = 7
// d[0][1] = 3
// d[0][2] = 0
// d[0][3] = 5
// d[1][0] = 2
// d[1][1] = 0
// d[1][2] = 1
// d[1][3] = 6
```

W komentarzach do tego przykładu przedstawiono wartości przypisane poszczególnym elementom tablicy. Zwróćmy uwagę, że pierwszy wymiar ([0:1]) (wiersze macierzy) jest ostatnim wymiarem, który zostanie zmieniony; odpowiada on elementom '{7,3,0,5} i '{2,0,1,6} na liście wartości szablonu przypisania. Jako pierwszy zmienia się drugi wymiar ([0:3]) (kolumny macierzy), któremu odpowiadają elementy wewnątrz list '{7,3,0,5} i '{2,0,1,6} szablonu przypisania. Należy również zauważyć, że zarówno zewnętrzne, jak i wewnętrzne nawiasy klamrowe są poprzedzone apostrofem, aby odróżnić listę wartości szablonu przypisania od operacji konkatenacji.

Operację replikacji także można wykorzystać do określania wartości elementów tablicy. Na przykład:

```
int e [0:1][0:3] = '{ 2{7,3,0,5} };
// e[0][0] = 7
// e[0][1] = 3
```

```
// e[0][2] = 0
// e[0][3] = 5
// e[1][0] = 7
// e[1][1] = 3
// e[1][2] = 0
// e[1][3] = 5
```

W tym przypadku lista wartości {7,3,0,5} powtarza się dwukrotnie ze względu na mnożnik replikacji 2. Zwróćmy uwagę, że nawiasy klamrowe w operacji replikacji są używane bez poprzedzającego je apostrofu.

Język SystemVerilog umożliwia również inicjalizację wszystkich elementów rozpakowanej tablicy lub fragmentu rozpakowanej tablicy przez podanie wartości domyślnej. Wartość domyślna określana jest w szablonie przypisania za pomocą słowa kluczowego **default**, które jest oddzielone od wartości dwukropkiem. Na przykład:

```
int a1 [0:7][0:1023] = '{default:8'h55};
```

W powyższym przykładzie wszystkie elementy tablicy są inicjowane wartością 8'h55.

Rozpakowana tablica może być też tablicą struktur lub innych typów zdefiniowanych przez użytkownika. W przypadku tablic struktur można określić jeden lub więcej pasujących kluczy typów. Na przykład:

```
struct {int a; time b;} abkey[1:0];                    // deklaracja tablicy struktur

abkey = '{'{a:1, b:2ns}, '{int:5, time:$time}};        // przypisanie wartości za pomocą
                                                       // szablonu
```

W ostatnim przykładzie jako kluczy szablonu abkey[1] użyto nazw elementów, a w abkey[0] typów prostych **int** i **time**.

7.1.4. Przypisywanie wartości do tablic

Ponieważ tablica spakowana jest wektorem, można do niej przypisać dowolne wyrażenie wektorowe. Zakres spakowanej tablicy nie ma wpływu na przypisanie.

Rozpakowana tablica lub jej fragment mogą zostać przypisane do innej rozpakowanej tablicy bądź jej fragmentu, jeśli spełnione są następujące warunki:

- typy elementów źródłowych i docelowych muszą być *równoważne* (*equivalent*) (patrz podrozdział 5.1.1);
- tablica źródłowa (fragment) musi mieć taką samą liczbę elementów jak tablica docelowa (fragment).

Aby rozpakowane tablice były zgodne przy przypisaniu, muszą mieć taką samą liczbę wymiarów rozpakowanych. Odbywa się to przez przypisanie każdemu elementowi tablicy źródłowej odpowiadającego mu elementu tablicy docelowej. W każdej tablicy pasujące elementy są określane na podstawie kolejności elementów od lewej do prawej. Na przykład:

```
int A [7:0];
int B [1:8];

A = B;
```

To ostatnie przypisanie spowoduje przyporządkowanie elementu B[1] do elementu A[7], elementu B[2] do elementu A[6] itd.

Tablicę sieci można przypisać do tablicy zmiennych i odwrotnie, jeśli typy danych tablicy źródłowej i docelowej są zgodne przy przypisaniu. Na przykład:

```
logic [7:0] V1 [10:0];      // tablica zmiennych, 10 elementów wektorów 8-bitowych
logic [7:0] V2 [10];        // to samo
wire [7:0] W [9:0];         // tablica sieci, typ danych logic,
                            // 10 elementów wektorów 8-bitowych
assign W = V1;              // dopuszczalne
initial  V2 = W;            // dopuszczalne
```

Język SystemVerilog daje duże możliwości podczas przypisywania wartości do tablic. Wartość lub lista wartości może być przypisana do pojedynczego elementu tablicy, jej części (*part*), wycinka (*slice*) lub całej tablicy. Podczas przypisywania wartości do rozpakowanych tablic stosuje się *szablony przypisania* dla tablic (patrz podrozdział 6.1.2).

Przykłady przypisywania wartości do rozpakowanej tablicy:

```
byte a [0:3][0:3];          // deklaracja rozpakowanej tablicy

a[1][0] = 8'h5;             // przypisanie wartości do pojedynczego elementu tablicy

a = '{   '{0,1,2,3},        // przypisanie listy wartości do całej tablicy
                            // użycie szablonu przypisania
        '{4,5,6,7},
        '{7,6,5,4},
        '{3,2,1,0}};

a[3] = '{'hF, 'hA, 'hC, 'hE};       // przypisywanie wartości do fragmentu tablicy
if (!resetN) begin
        a = '{default:0};           // inicjalizacja całości tablicy
```

```
        a[0] = '{default:4};              // inicjalizacja fragmentu tablicy
end
```

Podczas przypisywania wartości do tablic spakowanych używana jest składnia przypisywania wartości do wektorów.

Przykłady przypisywania wartości do tablicy spakowanej:

```
logic [1:0][1:0][7:0] a;          // deklaracja tablicy spakowanej

a[1][1][0] = 1'b0;                // przypisanie jednego bitu

a = 32'hF1A3C5E7;                 // przypisanie wartości do całej tablicy

a[1][0][3:0] = 4'hF;              // przypisanie wartości do części tablicy

a[0] = 16'hFACE;                  // przypisanie wartości do fragmentu tablicy

a = {16'bz, 16'b0};               // przypisanie konkatenacji do całej tablicy
```

7.1.5. Kopiowanie tablic

Przyjrzyjmy się, w jaki sposób można kopiować tablice. Ponieważ tablice spakowane są traktowane jak wektory, każda z nich może zostać przypisana do innej tablicy spakowanej, nawet jeśli mają one różne rozmiary i typy. W przypadku niedopasowania wielkości tablic do ich obcięcia lub rozszerzenia stosuje się standardowe reguły przypisywania wektorów. Na przykład:

```
bit [1:0][15:0] a;      // 32-bitowy wektor z dwoma stanami
logic [3:0][ 7:0] b;    // 32-bitowy wektor z czterema stanami
logic [15:0] c;         // 16-bitowy wektor z czterema stanami
logic [39:0] d;         // 40-bitowy wektor z czterema stanami

b = a;                  // przypisanie 32-bitowej tablicy a do 32-bitowej tablicy b
c = a;                  // starsze 16 bitów wektora a zostanie obcięte
d = a;                  // starsze 16 bitów wektora d zostanie wypełnione zerami
```

Tablicę rozpakowaną można bezpośrednio przypisać do innej tablicy rozpakowanej tylko wtedy, gdy obie mają tę samą liczbę wymiarów, te same rozmiary elementów i te same typy. Przypisanie odbywa się przez skopiowanie każdego elementu jednej tablicy do odpowiadającego mu elementu tablicy docelowej. Elementy w tablicach nie muszą być ponumerowane identycznie. Ułożenie tablic i typów musi jednak dokładnie odpowiadać sobie nawzajem. Na przykład:

```
logic [31:0] a [2:0][9:0];      // deklaracja rozpakowanej tablicy
logic [0:31] b [1:3][1:10];     // deklaracja kolejnej rozpakowanej tablicy

a = b;                          // kopiowanie jednej rozpakowanej tablicy
                                // do innej rozpakowanej tablicy
```

Jeśli dwie rozpakowane tablice nie są identyczne pod względem położenia, można je przypisać za pomocą *rzutowania strumienia bitów* (*bit-stream casting*) (patrz podrozdział 5.2.3).

7.1.6. Kopiowanie tablic i struktur za pomocą strumieni bitów

Tablicy rozpakowanej nie można bezpośrednio przypisać do tablicy spakowanej. Dzieje się tak dlatego, że w rozpakowanej tablicy każdy element jest przechowywany niezależnie i dlatego nie może być traktowany jako wartość całkowita (wektor). Jednak tablice rozpakowane można przypisać do tablic spakowanych za pomocą rzutowania (konwersji) strumienia bitów.

Spakowanej tablicy nie można bezpośrednio przypisać do rozpakowanej tablicy. Nawet jeśli rozmiary obu tablic są identyczne, tablica spakowana jest traktowana jako wektor, którego nie można bezpośrednio przypisać do tablicy rozpakowanej, w której jej każdy element może być przechowywany niezależnie od innych elementów. Przypisanie można jednak wykonać za pomocą operacji konwersji strumienia bitów.

Rzutowanie strumienia bitów (*bit-stream casting*) jest mechanizmem służącym do:

- przypisywania rozpakowanej tablicy do rozpakowanej tablicy o innej lokalizacji;
- przypisywania rozpakowanej tablicy do tablicy spakowanej;
- przypisywania spakowanej tablicy do rozpakowanej tablicy;
- przypisywania struktury do spakowanej lub rozpakowanej tablicy;
- przypisywania tablicy o stałym lub dynamicznym rozmiarze do tablicy o rozmiarze dynamicznym;
- przypisania jednej struktury do innej struktury z innym układem wewnętrznym.

Rzutowanie strumienia bitów tymczasowo przekształca rozpakowaną tablicę w strumień bitów w postaci wektorowej. Ten tymczasowy wektor można następnie przypisać do innej tablicy, która może być tablicą spakowaną lub rozpakowaną. Całkowita liczba bitów w tablicy źródłowej i docelowej musi być taka sama, ale rozmiar każdego elementu w obu tablicach może być różny.

Rzutowanie strumienia bitów wykorzystuje operację konwersji typu ('). Użycie konwersji bitowej wymaga, aby tablica docelowa była typu użytkownika zdefiniowanego za pomocą słowa kluczowego **typedef**. Na przykład:

```
typedef int data_t [3:0][7:0];          // zdefiniowany przez użytkownika typ
                                        // rozpakowanej tablicy
data_t a;                               // rozpakowana tablica a

int b [1:0][3:0][3:0];                  // kolejna rozpakowana tablica b

a = data_t'(b);                         // przypisanie jednej rozpakowanej tablicy
                                        // do innej rozpakowanej tablicy
```

7.1.7. Operacje na tablicach

W języku SystemVerilog można wykonywać następujące operacje na wszystkich tablicach, zarówno spakowanych, jak i rozpakowanych:

- odczyt i zapis elementu tablicy, np. A [i] = B [j];
- odczyt i zapis tablicy, np. A = B;
- odczyt i zapis fragmentów tablic, np. A [i:j] = B [h:t];
- odczyt i zapis zmiennego fragmentu tablicy, np. A [x+:c] = B [y+:c];
- porównanie całej tablicy lub jej fragmentu, np. A == B; A [i:j] != B [h:t].

Ponadto w przypadku tablic spakowanych można dodatkowo wykonać:

- przypisanie liczby całkowitej, np. A = 8'h0A;
- przetwarzanie w wyrażeniu z traktowaniem tablicy jako liczby całkowitej, np. (A + 3).

7.1.8. Pamięć

Jednowymiarowa rozpakowana tablica z elementami typu **logic** (**reg**) lub **bit** jest nazywana *pamięcią* (*memory*). Element o spakowanym wymiarze w tablicy jest nazywany *elementem pamięci* (*memory element*) lub *słowem pamięci* (*memory word*). Tablice pamięci mogą być używane do modelowania *pamięci o dostępie swobodnym* (*random access memory* – RAM), *pamięci tylko do odczytu* (*read-only memory* – ROM), a także *pliku rejestrów* (*register file*) [5]. Na przykład:

```
logic [7:0] mem [0:255];        // deklaracja macierzy pamięci składającej się z 256
                                // 8-cyfrowych słów, indeksy tablicowe od 0 do 255

mem[5] = 0;                     // wpisanie słowa pod adresem 5

data = mem[addr];               // odczyt słowa z adresu addr
```

7.1.9. Tablice wielowymiarowe

Tablica wielowymiarowa (*multidimensional array*) to inaczej *tablica tablic* (*array of arrays*). Definiuje się ją za pomocą kilku (więcej niż jednego) zakresów zwanych także *wymiarami tablicy* (*array dimensions*) lub po prostu *wymiarami* (*dimensions*). Te, które poprzedzają nazwę tablicy, określają wymiary spakowane, a wymiary następujące po nazwie tablicy określają wymiary rozpakowane. Na przykład:

```
bit [3:0] [7:0] m [1:10];        // 10 elementów po 4 bajty 8-bitowe
                                 // (każdy element jest pakowany w 32 bity)
```

Przykłady użycia tablicy wielowymiarowej:

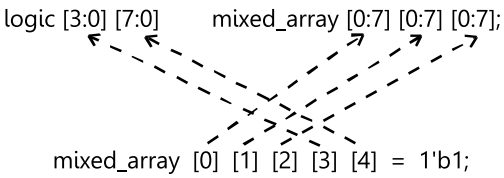
```
m[9] = m[8] + 1;                 // dodaje od 1 do 4 bajtów
m[7][3:2] = m[6][1:0];           // kopiuje 2 bajty
```

W tablicy wielowymiarowej wymiary spakowane ([3:0][7:0] w poprzednim przykładzie) zmieniają się szybciej niż rozpakowane ([1:10] w poprzednim przykładzie). Przy indeksowaniu wymiary spakowane podawane są po rozpakowanych.

Na liście wymiarów najszybciej zmienia się wymiar najbardziej po prawej stronie, tak jak w języku C. Jednak wymiar spakowany zmienia się szybciej niż wymiar rozpakowany. Na przykład:

```
bit [1:10] v1 [1:5];             // 1 do 10 zmienia się najszybciej;
                                 // kompatybilne z tablicami pamięci

bit v2 [1:5] [1:10];             // 1 do 10 zmienia się najszybciej, zgodne z C

bit [1:5] [1:10] v3 ;            // 1 do 10 zmienia się najszybciej

bit [1:5] [1:6] v4 [1:7] [1:8];      // od 1 do 6 zmienia się najszybciej, następnie od 1 do 5,
                                     // potem od 1 do 8, a następnie od 1 do 7
```

Podczas indeksowania tablic w pierwszej kolejności są przywoływane wymiary rozpakowane, od wymiaru najbardziej wysuniętego na lewo do wymiaru najbardziej wysuniętego na prawo. Wymiary spakowane (pola wektorowe) są przywoływane jako drugie, od wymiaru najbardziej wysuniętego na lewo do wymiaru najbardziej wysuniętego na prawo. Na rys. 7.3 pokazano kolejność indeksowania wymiarów w mieszanej wielowymiarowej tablicy spakowanej i rozpakowanej.



RYS. 7.3. Kolejność indeksowania dla mieszanej wielowymiarowej tablicy spakowanej i rozpakowanej

ŹRÓDŁO: oprac. własne.

Wielowymiarową tablicę spakowaną można zdefiniować na podstawie innej tablicy spakowanej za pomocą słowa kluczowego **typedef**. Na przykład:

```
typedef bit [1:5] my_byte;          // typ my_byte jest zadeklarowany jako jednowymiarowa
                                    // tablica spakowana
my_byte [1:10] pm_2_dim;            // deklaracja dwuwymiarowej tablicy spakowanej
                                    // pm_2_dim, od 1 do 5 zmienia się najszybciej
                                    // tak samo jak bit [1:10][1:5] pm_2_dim;
```

Podobnie wielowymiarową tablicę rozpakowaną można zdefiniować na podstawie innej tablicy rozpakowanej za pomocą słowa kluczowego **typedef**. Na przykład:

```
typedef my_byte mem_t [0:3];        // tablica składająca się z czterech elementów my_byte

mem_t my_mem [0:7];                 // tablica ośmiu elementów mem_t, takich samych
                                    // jak bit [1:5] my_mem [0:7][0:3];
```

Podtablica (*subarray*) to tablica, która jest elementem innej tablicy. Podobnie jak w języku C, do podtablic odwołujemy się przez pominięcie indeksów jednego lub więcej wymiarów tablicy, zawsze ignorując te, które zmieniają się najszybciej. Brak indeksów dla wszystkich wymiarów oznacza odwoływanie się do całej tablicy. Na przykład:

```
int A[2][3][4], B[2][3][4], C[5][4];    // deklaracja trzech rozpakowanych tablic A, B i C
...
A[0][2] = B[1][1];          // przypisanie podtablicy składającej się z czterech tablic int
                            // tak samo jak A[0][2][0:3] = B[1][1][0:3];
A[1] = B[0];                // przypisanie podtablicy składającej się z trzech tablic
                            // po cztery int każda, tak samo jak
                            // A[1][0:2][0:3] = B[0][0:2][0:3];
A = B;                      // przypisanie całej tablicy
A[0][1] = C[4];             // przypisanie zgodnej podtablicy złożonej z czterech tablic int,
                            // tak samo jak A[0][1][0:3] = C[4][0:3];
```

W języku SystemVerilog tablice mogą być deklarowane jako lista, tzn. rozdzielane przecinkami. Wszystkie tablice na liście muszą mieć ten sam typ danych i taki sam rozmiar jak tablice spakowane. Na przykład:

```
bit [7:0] [31:0] v7 [1:5] [1:10], v8 [0:255];          // zadeklarowano dwie tablice v7 i v8
```

7.1.10. Dostęp do elementów i fragmentów (części) tablic

Jeden element w spakowanej lub rozpakowanej tablicy można wybrać za pomocą *nazwy indeksowanej* (*indexed name*). Na przykład:

```
bit [3:0] [7:0] j;        // j jest tablicą spakowaną
byte k;

k = j[2];                 // wybór jednego 8-bitowego elementu z j
```

W SystemVerilog istnieje rozróżnienie między pojęciami *wycinka* lub *fragmentu* (*slice*) i *wyboru części* (*part-select*).

Termin „wybór" części odnosi się do wyboru jednego lub kilku *sąsiednich bitów* w spakowanej tablicy o jednym rozmiarze. Na przykład:

```
logic [63:0] data;      // spakowana jednowymiarowa tablica danych o długości 64 bitów
logic [7:0] byte2;      // spakowana jednowymiarowa tablica byte2 o długości 8 bitów

byte2 = data[23:16];  // wybór 8-bitowej części z tablicy data
```

Termin *fragment* (wycinek, kawałek) odnosi się do wyboru jednego lub kilku *sąsiednich elementów* tablicy. Nazwa tablicy służy do wybierania sąsiednich elementów tablicy, czyli jej wycinka. Wycinek spakowanej tablicy jest także tablicą spakowaną, a wycinek rozpakowanej tablicy jest tablicą rozpakowaną. Na przykład:

```
bit signed [31:0] busA;              // tablica spakowana busA
byte B;                              // typ integer byte

B = busA [16 +: 8];                  // wybór fragmentu spakowanej tablicy
```

Przykład wyboru fragmentu rozpakowanej tablicy:

```
bit signed [31:0] busA [7:0] ; // rozpakowana tablica 8 wektorów 32-bitowych
int busB [1:0];                // rozpakowana tablica 2 liczb całkowitych

busB = busA[7:6];              // wybór fragmentu dwóch wektorów z tablicy busA
```

Rozmiar wybranej części lub fragmentu tablicy musi być stały (niezmienny), ale ich położenie może być zmienne. Na przykład:

```
int i = bitvec[j +: k];                    // k musi być stałą.
int a[x:y], b[y:z], e;

a = {b[c -: d], e};                        // d musi być stałą
```

Wybór części tablicy może być zastosowany tylko do jednego wymiaru tablicy, a pozostałe wymiary tablicy muszą mieć pojedyncze wartości, tzn. nie mogą być zakresami wartości.

Podczas wybierania elementu tablicy lub jej części indeks nie może znajdować się poza zakresem, a wartości x i z w wyrażeniu indeksu nie są dozwolone.

7.1.11. Przekazywanie tablic przez porty modułów i jako argumenty do zadań i funkcji

Język Verilog pozwala przekazywać wektory, czyli spakowane tablice jednowymiarowe, przez porty, a także jako argumenty do zadań i funkcji. Język SystemVerilog rozszerza te możliwości, umożliwiając przekazywanie dowolnych tablic jako argumentów do zadań i funkcji.

Aby przekazać tablicę przez port oraz jako argumenty do zadania lub funkcji, zarówno port, jak i formalny argument zadania bądź funkcji muszą być zadeklarowane jako tablica. Obowiązują te same zasady co w przypadku przypisywania wartości do tablic. Na przykład:

```
module CPU (...);
        ...
        logic [7:0] m [0:255];          // rozpakowana tablica m

        lookup i1 (.LUT(m));            // przekazanie tablicy m do instancji i1 modułu lookup
        ...
endmodule

module lookup (output logic [7:0] LUT [0:255]);    // tablica LUT
                                                   // jako port wejściowy modułu
        ...
        initial load(LUT);                         // wywołanie zadania load

        task load (inout logic [7:0] t [0:255]);   // zadanie,
                                                   // którego argumentem jest tablica
        ...
        endtask
endmodule
```

7.1.12. Tablice, struktury i unie

Tablice spakowane i rozpakowane mogą zawierać struktury i unie jako elementy tablicy. W spakowanej tablicy musi być również spakowana struktura lub unia. Na przykład

```
typedef struct packed {          // struktura spakowana
        logic [31:0] a;
        logic [ 7:0] b;
} packet_t;

packet_t [23:0] packet_array;    // spakowana tablica 24 struktur

typedef struct {                 // rozpakowana struktura
        int a;
        real b;
} data_t;

data_t data_array [23:0];        // rozpakowana tablica 24 struktur
```

Z kolei struktury i unie mogą zawierać tablice spakowane lub rozpakowane. Struktura spakowana albo unia mogą zawierać tylko tablice spakowane. Na przykład:

```
struct packed {                  // struktura spakowana
        logic parity;
        logic [3:0][ 7:0] data;  // dwuwymiarowa tablica spakowana
} data_word;

struct {                         // rozpakowana struktura
        logic data_ready;
        logic [7:0] data [0:3];  // tablica rozpakowana
} packet_t;
```

7.1.13. Implementacja pętli w obrębie elementów tablicy (operator foreach)

Język SystemVerilog udostępnia instrukcję **foreach** umożliwiającą iterację po elementach tablic jedno- i wielowymiarowych bez określania rozmiaru jej każdego wymiaru. Argumentem pętli **foreach** jest nazwa tablicy, po której następuje oddzielona przecinkami lista zmiennych pętli ujętych w nawiasy kwadratowe. Każda zmienna w pętli odpowiada jednemu z wymiarów tablicy. Na przykład:

```
int sum [1:8] [1:3];                    // rozpakowana tablica dwuwymiarowa

foreach ( sum[i,j] )                    // pętla foreach
        sum[i][j] = i + j;              // inicjalizacja elementów tablicy
```

Wiele zmiennych pętli tworzy pętle zagnieżdżone, które iterują po podanych indeksach. Pętle zewnętrzne odpowiadają najniższym indeksom. W powyższym przykładzie pętla zewnętrzna iteruje po j.

Zmienne pętli **foreach** (i oraz j w naszym przykładzie) nie muszą być zadeklarowane. Rozmiar można pominąć, wskazując pozycję zmiennej za pomocą dwóch przecinków bez nazwy zmiennej. Puste zmienne pętli oznaczają, że nie będzie ona iterować nad tym wymiarem tablicy. Puste zmienne pętli na końcu listy zmiennych można również pominąć bez konieczności umieszczania dodatkowych przecinków.

Poniższy przykład przedstawia funkcję, która generuje bit kontrolny dla każdego bajtu w 128-bitowym wektorze. Wektor jest reprezentowany jako dwuwymiarowa spakowana tablica składająca się z 16 elementów 8-bitowych. W pętli **foreach** zdefiniowana jest tylko jedna zmienna, która reprezentuje pierwszy wymiar tablicy (wymiar [15:0]).

```
function [15:0] gen_crc (logic [15:0] [7:0] d);
foreach (gen_crc[i]) gen_crc[i] = ^d[i];        // obliczanie bitu kontrolnego
endfunction
```

Zmienne pętli w instrukcji **foreach** są automatyczne, tylko do odczytu i lokalne dla pętli. Typ każdej zmiennej pętli jest niejawnie zadeklarowany jako zgodny z typem indeksu tablicy, który dla tablic syntetyzowalnych jest typem **int**. Należy zauważyć, że w języku SystemVerilog istnieją również tablice asocjacyjne (niesyntezowalne), które mogą używać innego typu dla swoich indeksów.

7.1.14. Przykład użycia tablic

Poniższy przykład opisuje model rejestru instrukcji w postaci spakowanej tablicy 32 instrukcji. Każda instrukcja jest konstrukcją złożoną, przedstawioną w postaci spakowanej struktury. Operandy w instrukcji mogą być ze znakiem lub bez znaku, które są reprezentowane jako unia tych dwóch typów. Wejściami do rejestru rozkazów są poszczególne operandy, kod operacji oraz flaga wskazująca, czy operandy są ze znakiem czy bez znaku. Konstrukcja rejestru instrukcji ładuje te poszczególne części informacji do spakowanej tablicy. Wynikiem działania modelu jest tablica 32 instrukcji.

Przykład 7.1. Przykład użycia tablic

```
package definitions;                        // pakiet z deklaracjami
        typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;   // typ kodu operacji

        typedef enum {UNSIGNED, SIGNED} operand_type_t;      // typ operandu

        typedef union packed {              // unia określająca typ danych
                logic          [31:0] u_data;
                logic signed   [31:0] s_data;
        } data_t;

        typedef struct packed {             // struktura określająca typ polecenia
                opcode_t opc;               // kod operacji
                operand_type_t op_type;     // typ operandu
                data_t op_a;                // operand a
                data_t op_b;                // operand b
                } instr_t;
endpackage

import definitions::*;                      // import pakietu do jednostki kompilacji

module instruction_register (               // moduł opisujący rejestr instrukcji
        output instr_t [0:31] instr_reg,    // rejestr instrukcji jako spakowana tablica struktur
        input data_t operand_a,             // operand a
        input data_t operand_b,             // operand b
        input operand_type_t op_type,       // typ operandu
        input opcode_t opcode,              // kod operacji
        input logic [4:0] addr              // adres rejestru
        );

        always @(addr) begin
                instr_reg[addr].op_type = op_type;   // definicja typu operandu
                instr_reg[addr].opc = opcode;        // definicja kodu operacji

                                // zapis operandów do rejestru w zależności od typu operandu
        if (op_type == SIGNED) begin
                instr_reg[addr].op_a.s_data = operand_a.s_data;
                instr_reg[addr].op_b.s_data = operand_b.s_data;
                end
                else begin
```

```
                instr_reg[addr].op_a.u_data = operand_a.u_data;
                instr_reg[addr].op_b.u_data = operand_b.u_data;
            end
        end
endmodule
```

7.2. Konkatenacja tablic rozpakowanych

Konkatenacja tablic rozpakowanych (*unpacked array concatenation*) to kolejny elastyczny sposób specyfikacji wartości elementów w tablicy rozpakowanej. Na przykład:

```
int m [1 : 3];              // rozpakowana tablica składająca się z trzech elementów
m = {1, 2, 3};              // odpowiednik m[1] = 1; m[2] = 2; m[3] = 3;
```

Konkatenacja rozpakowanych tablic jest podobna do szablonu przypisania dla tablic. Na przykład:

```
m = '{1, 2, 3};         // służy do tego samego celu
```

Szablony przypisania umożliwiają jednak stosowanie replikacji ('{n{wartość}}), a także ustawianie wartości domyślnych za pomocą konstrukcji **default**.

Poniższe przykłady ilustrują niektóre różnice między tymi dwiema formami:

```
typedef int AI3[1:3];       // deklaracja typu AI3 jako tablicy trzech liczb całkowitych

AI3 A3;                     // tworzenie instancji A3 typu AI3
int A9[1:9];                // A9 jest rozpakowaną tablicą dziewięciu liczb całkowitych

A3 = '{1, 2, 3};                    // odpowiednik A3[1]=1, A3[2]=2, A3[3]=3
A9 = '{3{A3}};                      // nieprawidłowe, A3 nieprawidłowy typ elementu
A9 = '{A3, 4, 5, 6, 7, 8, 9};       // nieprawidłowe, A3 nieprawidłowy typ elementu
A9 = {A3, 4, 5, A3, 6};             // dopuszczalne, daje A9='{1,2,3,4,5,1,2,3,6}.
A9 = '{9{1}};                       // dopuszczalne, daje A9='{1,1,1,1,1,1,1,1,1}.
A9 = {9{1}};                        // nieprawidłowe, brak replikacji w konkatenacji
                                    // rozpakowanej tablicy
A9 = {A3, {4,5,6,7,8,9} };          // nieprawidłowe, tutaj {...} nie jest samookreśleniem
A9 = {A3, '{4,5,6,7,8,9} };         // nieprawidłowe, '{...} nie jest samookreśleniem
A9 = {A3, 4, AI3'{5, 6, 7}, 8, 9};  // dopuszczalne, A9='{1,2,3,4,5,6,7,8,9}
```

7.3. Inne rodzaje tablic

Wszystkie omówione powyżej typy tablic i konwersje tablicowe są syntetyzowalne. Język SystemVerilog udostępnia kilka innych typów tablic, które nie są syntetyzowalne i mogą być używane tylko w symulacji. Deweloperzy powinni jednak wiedzieć o ich istnieniu. Tablice niesyntezowalne obejmują:

- *tablice dynamiczne* (*dynamic arrays*);
- *tablice asocjacyjne* (*associative arrays*);
- *kolejki* (*queues*).

7.3.1. Tablice dynamiczne

Tablica dynamiczna (*dynamic array*) to rozpakowana tablica, której rozmiar można ustawić lub zmienić w czasie działania, tzn. w czasie symulacji. Domyślnym rozmiarem niezainicjowanej tablicy dynamicznej jest zero. Rozmiar tablicy dynamicznej jest ustawiany przez *konstruktor* **new** lub *przypisanie do tablicy*. Elementami tablic dynamicznych mogą być zmienne wszystkich typów danych, a także inne tablice.

W deklaracjach wymiary tablicy dynamicznej zwane *wymiarami dynamicznymi* są oznaczane nawiasami kwadratowymi []. Na przykład:

```
bit [3:0] n [];          // dynamiczna tablica wektorów 4-bitowych
int m [2][];             // rozpakowana tablica o stałym rozmiarze składająca się
                         // z dwóch dynamicznych podtablic liczb całkowitych
```

Aby identyfikator reprezentował tablicę dynamiczną, rozmiar dynamiczny [] musi określać najbardziej na prawo wysunięty rozmiar rozpakowany. Język SystemVerilog udostępnia następujące mechanizmy pracy z tablicami dynamicznymi:

- konstruktor **new**[] służący do ustawiania lub zmiany rozmiaru tablicy oraz inicjalizowania jej elementów;
- metoda **size**() zwraca bieżący rozmiar tablicy;
- metoda **delete**() usuwa wszystkie elementy tablicy, zwraca pustą tablicę.

7.3.2. Tablice asocjacyjne

Tablica asocjacyjna (*associative array*) implementuje *tabelę przeglądową* lub *przeszukiwania* (*lookup table*) elementów określonego typu. Typ danych użyty jako indeks jest *kluczem wyszukującym* (*lookup key*). Tablica asocjacyjna nie ma pamięci, dopóki nie zostanie wykorzystana, a wyrażenie indeksowe może być dowolnego typu, niekoniecznie całkowitego.

Do deklarowania tablicy asocjacyjnej używa się następującej składni:

data_type array_name [*index_type*];

gdzie: *data_type* jest typem danych elementów tablicy, może być dowolnego typu; *array_name* jest nazwą tablicy; *index_type* jest typem danych używanym jako indeks lub * (gwiazdka).

Jeśli jako *index_type* użyto *, tablica jest indeksowana przez dowolne wyrażenie całkowite o dowolnym rozmiarze. Przykłady deklaracji tablic asocjacyjnych:

```
integer i_array[*];          // tablica asocjacyjna liczb całkowitych (indeks
                             // niezdefiniowany)
bit [20:0] array_b[string];  // tablica asocjacyjna wektora 21-bitowego, indeksowane
                             // według typu string
event ev_array[myClass];     // tablica asocjacyjna zdarzeń, indeksowana klasą myClass
```

Elementy w tablicach asocjacyjnych są przydzielane dynamicznie. Pamięć jest alokowana do elementu tablicy asocjacyjnej, gdy jest on używany jako cel przypisania. Tablica asocjacyjna przechowuje rekordy, do których przypisane są wartości, oraz ich względną kolejność zgodnie z typem danych indeksu. Aby można było użyć pojedynczego elementu tablicy asocjacyjnej, musi on zostać wybrany.

Język SystemVerilog udostępnia następujące metody obsługi tablic asocjacyjnych:

- **num**() i **size**() – zwracają liczbę wpisów w tablicy asocjacyjnej;
- **delete**() – usuwa rekord o podanym indeksie;
- **exists**() – sprawdza, czy element o podanym indeksie istnieje w podanej tablicy;
- **first**() – przypisuje zmiennej indeksowej wartość pierwszego (najmniejszego) indeksu w tablicy asocjacyjnej;
- **last**() – przypisuje zmiennej indeksowej wartość ostatniego (największego) indeksu w tablicy asocjacyjnej;
- **next**() – znajduje najmniejszy indeks, którego wartość jest większa od podanego argumentu index;
- **prev**() – znajduje największy indeks, którego wartość jest mniejsza od podanego argumentu index.

7.3.3. Kolejki

Kolejka (*queue*) to uporządkowany zbiór jednakowych elementów o zmiennej wielkości. Obsługuje ona dostęp do wszystkich swoich elementów, a także umożliwia wstawianie elementu na początek i usuwanie elementu z końca kolejki. Każdy element w kolejce jest identyfikowany *liczbą porządkową* (*ordinal number*), która określa jego pozycję w kolejce, gdzie 0 oznacza pierwszy element, a $ ostatni. Kolejka

jest podobna do jednowymiarowej tablicy rozpakowanej, której rozmiar zmienia się automatycznie.

Kolejki są deklarowane w taki sam sposób jak tablice rozpakowane, ale ze znakiem $ jako rozmiarem tablicy. Można ograniczyć maksymalny rozmiar kolejki, określając jej opcjonalną prawą granicę. Na przykład:

```
byte q1[$];                    // kolejka bajtów
string names[$] = { „Bob” };   // kolejka łańcuchów jednoelementowych
integer Q[$] = { 3, 2, 7 };    // inicjalizacja kolejki liczb całkowitych
bit q2[$:255];                 // kolejka o maksymalnym rozmiarze 256 bitów
```

Jeśli w deklaracji nie podano wartości początkowej, zmienna kolejki jest inicjalizowana jako *pusta kolejka* {}.

Kolejki obsługują wszystkie operacje na rozpakowanych tablicach. Ponadto język SystemVerilog udostępnia następujące metody obsługi kolejek:

- **size**() – zwraca liczbę elementów w kolejce;
- **insert**() – wstawia element na podaną pozycję w indeksie;
- **delete**() – usuwa element z podanej pozycji indeksu;
- **pop_front**() – usuwa i zwraca pierwszy element kolejki;
- **pop_back**() – usuwa i zwraca ostatni element kolejki;
- **push_front**() – wstawia element jako pierwszy element kolejki;
- **push_back**() – wstawia element jako ostatni element kolejki.

7.4. Funkcje i metody systemowe do obsługi tablic

Język SystemVerilog udostępnia wiele *funkcji* i metod *systemowych* ułatwiających pracę z tablicami.

7.4.1. Funkcje systemowe

Język SystemVerilog udostępnia następujące funkcje systemowe do pracy z tablicami:

- **$left** – zwraca lewą granicę tablicy; dla rozmiaru spakowanego zwraca indeks najbardziej znaczącego elementu;
- **$right** – zwraca prawą granicę tablicy; dla rozmiaru spakowanego zwraca indeks najmniej znaczącego elementu;
- **$increment** – zwraca 1, jeśli **$left** >= **$right**, i -1 w przeciwnym wypadku;
- **$low** – zwraca najmniejszą granicę zakresu, tzn. zwraca **$left**, jeśli **$left** < **$right**, i zwraca **$right**, jeśli **$left** >= **$right**;
- **$high** – zwraca największą granicę zakresu, tzn. zwraca **$right**, jeśli **$left** < **$right**, i zwraca **$left**, jeśli **$left** >= **$right**;

- **$size** – zwraca liczbę elementów w wymiarze, **$size = $high** - **$low** + 1;
- **$dimensions** – zwraca całkowitą liczbę wymiarów tablicy (spakowanych i rozpakowanych); 1 dla typu danych **string** lub dowolnego innego typu, który jest równoważny prostemu typowi wektora bitowego, 0 dla wszystkich innych typów;
- **$unpacked_dimensions** – zwraca całkowitą liczbę rozpakowanych wymiarów tablicy, 0 dla wszystkich innych typów.

Rozważane funkcje systemowe mają następującą składnię:

func_name (*array_expression* [, *dimension_expression*]);

gdzie: *func_name* jest nazwą funkcji; *array_expression* jest wyrażeniem tablicowym (tablicą, częścią tablicy lub fragmentem tablicy); *dimension_expression* jest wyrażeniem całkowitym, którego wynikiem jest liczba określająca wymiar tablicy. Nie jest ono wymaganym argumentem, np. nie występuje w składni funkcji **$dimensions** i **$unpacked_dimensions**.

Rozmiary tablic są numerowane w następujący sposób: najwolniej zmieniający się rozmiar otrzymuje numer 1. Następnie kolejno numeruje się szybciej zmieniające się rozmiary. Na przykład w deklaracji:

```
bit [3:0] [2:1] m [1:5] [2:8];
```

rozmiar [1:5] to numer 1, [2:8] to numer 2, [3:0] to numer 3, [2:1] to numer 4.

W przypadku typów całkowitych o stałym rozmiarze (**integer**, **shortint**, **longint** i **byte**) rozmiar jest predefiniowany za pomocą liczby 1. Dla liczby całkowitej N zadeklarowanej bez specyfikatora zakresu przyjmuje się, że jej granice są równe [**$bits**(N)-1:0].

Na przykład załóżmy, że zdefiniowano następujące deklaracje:

```
typedef logic [16:1] Word;          // jednowymiarowa tablica spakowana
Word ram [0:9];                     // tablica dwuwymiarowa o rozmiarze [0:9]
                                    // rozpakowanym i rozmiarze [16:1] spakowanym
```

dla tej deklaracji **$size**(Word) = 16, a **$size**(ram, 2) = 16, ponieważ rozmiar [0:9] ma numer 1 jako najwolniejszy, a rozmiar [16:1] ma numer 2.

Przykłady użycia funkcji systemowych do pracy z tablicami:

```
// użycie funkcji $left
logic [1:2][7:0] word [0:3][4:1];
int y[3];
```

```
y[0] = $left(word,1);          // zwraca 0
y[1] = $left(word,2);          // zwraca 4
y[2] = $left(word,3);          // zwraca 1
y[3] = $left(word,4);          // zwraca 7

// użycie funkcji $low
logic [7:0] word [1:4];

y[0] = $low(word,1)            // zwraca 1
y[1] = $low(word,2)            // zwraca 0
```

Poniższy fragment kodu pokazuje, jak można użyć niektórych z tych funkcji systemowych do inkrementacji tablicy, bez konieczności opisywania rozmiaru każdego wymiaru tablicy:

```
logic [3:0][7:0] array [0:1023];        // deklaracja tablicy

int d = $dimensions(array);             // d jest liczbą wymiarów tablicy

if (d > 0) begin                        // obiekt jest tablicą
        for (int j = $right(array,1);   // to jest jedna instrukcja for
                j != ($left(array,1) + $increment(array,1) );
                j += $increment(array,1))
        begin
                ...
        end
end
```

W tym przykładzie $right(array,1) zwraca 1023, $left(array,1) zwraca 0, $increment(array,1) zwraca -1. Dlatego pętla **for** ma następującą postać:

```
for (int j = 1023; j != -1; j += -1)
begin
        ...
end
```

Powyższy przykład można również zrealizować za pomocą pętli **foreach** w następujący sposób:

```
foreach ( array[j] )
begin
        ...
end
```

Należy zauważyć, że funkcje systemowe dla tablic są syntetyzowalne, pod warunkiem że tablica ma stały rozmiar, a argument określający wymiar jest stały lub w ogóle nie jest podany. Operator pętli **foreach** jest również syntetyzowalny, pod warunkiem że tablica ma stały rozmiar.

7.4.2. Funkcja systemowa $bits

W języku SystemVerilog dodano *funkcję systemową* **$bits**, która zwraca liczbę bitów dowolnego wyrażenia. Wyrażenie może zawierać dowolny typ wartości, w tym spakowane lub rozpakowane tablice, struktury, unie i literały liczbowe.

Składnia funkcji **$bits** jest następująca:

$bits(*expression***)**,

gdzie *expression* jest wyrażeniem, które ma być sprawdzone.

Przykłady zastosowania funkcji **$bits** do tablic:

```
bit [63:0] a;
logic [63:0] b;
wire [3:0][7:0] c [0:15];
struct packed {byte tag; logic [31:0] addr;} d;

$bits(a) zwraca 64
$bits(b) zwraca 64
$bits(c) zwraca 512
$bits(d) zwraca 40
$bits(a+b) zwraca 128
```

7.4.3. Metody przetwarzania tablic

Oprócz funkcji systemowych język SystemVerilog udostępnia także wiele metod do manipulowania tablicami. Składnia wywoływania metod do przetwarzania tablic jest następująca:

```
array_name.method_name {attribute_instance}
      [(iterator_argument)] [with (expression)];
```

gdzie: *array_name* – nazwa tablicy, może być wyrażeniem; *method_name* – nazwa metody; *attribute_instance* – atrybuty instancji; *iterator_argument* – (*iteration argument*) zmienna, która jest używana przez wyrażenie **with** do oznaczania elementu tablicy w każdej iteracji; **with** – słowo kluczowe; *expression* – wyrażenie użyte w wyrażeniu **with**.

7.4.3.1. Metody wyszukiwania

Metody wyszukiwania lub *lokalizacji* (*locator methods*) umożliwiają znalezienie takich elementów bądź ich indeksów w tablicy, które spełniają podane wyrażenie. Metody wyszukiwania działają na dowolnej rozpakowanej tablicy, w tym na kolejkach, ale ich typem zwracanym jest kolejka.

W przypadku wszystkich tablic, z wyjątkiem tablic asocjacyjnych, metody wyszukiwania zwracają kolejkę typu **int**. W przypadku tablic asocjacyjnych metody wyszukujące zwracają kolejkę tego samego typu co typ indeksu. Jeśli żadne elementy nie spełniają podanego wyrażenia lub tablica jest pusta (w przypadku kolejki bądź tablicy dynamicznej), zwracana jest pusta kolejka; w przeciwnym razie zwracana jest kolejka zawierająca wszystkie elementy spełniające podane wyrażenie. Wyrażenie opcjonalne w klauzuli **with** musi być wyrażeniem logicznym (boolowskim).

Język SystemVerilog obsługuje następujące metody wyszukiwania (warunek **with** jest obowiązkowy):

- **find**() – zwraca wszystkie elementy spełniające podane wyrażenie;
- **find_index**() – zwraca wszystkie indeksy wszystkich elementów spełniających podane wyrażenie;
- **find_first**() – zwraca pierwszy element spełniający podane wyrażenie;
- **find_first_index**() – zwraca indeks pierwszego elementu spełniającego podane wyrażenie;
- **find_last**() – zwraca ostatni element spełniający podane wyrażenie;
- **find_last_index**() – zwraca indeks ostatniego elementu spełniającego podane wyrażenie.

W przypadku poniższych metod można pominąć klauzulę **with**:

- **min**() – zwraca element o minimalnej wartości lub taki, którego wyrażenie jest obliczone jako minimalne;
- **max**() – zwraca element o maksymalnej wartości lub taki, którego wyrażenie jest obliczone jako maksymalne;
- **unique**() – zwraca wszystkie elementy z unikalnymi wartościami lub takie, których wyrażenie jest obliczone jako unikalna wartość;
- **unique_index**() – zwraca indeksy wszystkich elementów z unikalnymi wartościami lub których wyrażenie jest obliczane jako unikalna wartość.

Przykłady zastosowania metod wyszukiwania:

```
string SA[10], qs[$];          // rozpakowana tablica łańcuchów SA i kolejka łańcuchów qs
int IA[int], qi[$];            // rozpakowana tablica liczb całkowitych IA
                               // i kolejka liczb całkowitych qi

// Znalezienie wszystkich elementów większych niż 5
qi = IA.find( x ) with ( x > 5 );
```

```
// Znalezienie indeksów wszystkich elementów równych 3
qi = IA.find_index with ( item == 3 );

// Znalezienie pierwszego elementu równego Bob
qs = SA.find_first with ( item == „Bob” );

// Znalezienie ostatniego elementu równego Henry
qs = SA.find_last( y ) with ( y == „Henry” );

// Znalezienie indeksu ostatniego elementu większego od Z
qi = SA.find_last_index( s ) with ( s > “Z” );

// Znalezienie najmniejszego elementu
qi = IA.min;

// Znalezienie wierszy o największej wartości liczbowej
qs = SA.max with ( item.atoi );

// Znalezienie wszystkich unikalnych elementów ciągu znaków
qs = SA.unique;

// Znalezienie wszystkich unikalnych elementów ciągu znaków z s.tolower
qs = SA.unique( s ) with ( s.tolower );
```

7.4.3.2. Metody porządkowania tablic

Metody porządkujące tablicę (*array ordering methods*) zmieniają kolejność elementów dowolnej rozpakowanej tablicy, z wyjątkiem tablic asocjacyjnych. Są to następujące metody:

- **reverse**() – odwraca kolejność elementów w tablicy;
- **sort**() – sortuje tablicę w porządku rosnącym;
- **rsort**() – sortuje tablicę w porządku malejącym;
- **shuffle**() – powoduje, że kolejność elementów w tablicy jest losowa.

Na przykład:

```
string s[] = { „hello”, „sad”, „world” };      // inicjalizacja dynamicznej tablicy ciągów znaków
s.reverse;                                     // s staje się { “world”, “sad”, “hello” };

int q[$] = { 4, 5, 3, 1 };                     // kolejka liczb całkowitych
q.sort;                                        // q staje się {1, 3, 4, 5}.

struct { byte red, green, blue; } c [512];     // tablica 512 struktur
```

```
c.sort with ( item.red );                    // sortuj c, używając tylko pola red
c.sort( x ) with ( {x.blue, x.green} );      // sortuj według blue,
                                             a następnie według green
```

7.4.3.3. Metody redukcji tablic

Metody redukcji tablic (*array reduction methods*) są stosowane do dowolnej rozpakowanej tablicy wartości całkowitych w celu zredukowania jej do pojedynczej wartości. Wyrażenie w klauzuli **with** może być zastosowane do określenia wartości, które mają być użyte przy redukcji.

Metoda redukcji zwraca wartość tego samego typu co typ elementów tablicy lub typ wyrażenia w klauzuli **with** (jeśli jest określone).

Język SystemVerilog obsługuje następujące metody redukcji tablic:

- **sum**() – zwraca sumę wszystkich elementów tablicy lub, jeśli podano klauzulę **with**, zwraca sumę wartości uzyskanych przez obliczenie wyrażenia dla każdego elementu tablicy;
- **product**() – zwraca iloczyn wszystkich elementów tablicy lub, jeśli podano klauzulę **with**, zwraca iloczyn wartości otrzymanych podczas obliczania wyrażenia dla każdego elementu tablicy;
- **and**() – zwraca rezultat bitowej operacji AND wszystkich elementów tablicy lub, jeśli podano klauzulę **with**, zwraca rezultat bitowej operacji AND wartości otrzymanych po obliczeniu wyrażenia dla każdego elementu tablicy;
- **or**() – zwraca rezultat bitowej operacji OR wszystkich elementów tablicy lub, jeśli podano klauzulę **with**, zwraca rezultat bitowej operacji OR wartości otrzymanych podczas obliczania wyrażenia dla każdego elementu tablicy;
- **xor**() – zwraca rezultat bitowej operacji XOR wszystkich elementów tablicy lub, jeśli podano klauzulę **with**, zwraca rezultat bitowej operacji XOR wartości otrzymanych podczas obliczania wyrażenia dla każdego elementu tablicy.

Na przykład:

```
byte b[] = { 1, 2, 3, 4 };                   // inicjalizacja dynamicznej tablicy bajtów
int y;
y = b.sum ;                                  // y równa się 10 => 1 + 2 + 3 + 4
y = b.product ;                              // y równa się 24 => 1 * 2 * 3 * 4
y = b.xor with ( item + 4 );                 // y równa się 12 => 5 ^ 6 ^ 7 ^ 8

logic [7:0] m [2][2] = '{ '{5, 10}, '{15, 20} };   // inicjalizacja rozpakowanej tablicy bajtów
int y;
y = m.sum with (item.sum with (item));       // y równa się 50 => 5+10+15+20

logic g [1024];                              // rozpakowana tablica 1024 bitów
int y;
y = g.sum with ( int'(item) );               // wynik 32-bitowy
```

7.4.3.4. Zapytanie o indeks iteratora

Wyrażenia używane przez metody manipulowania tablicami wymagają czasami podania rzeczywistego indeksu tablicy w każdej iteracji, a nie tylko elementu tablicy. *Metoda zapytania o indeks iteratora* (*index querying method*) zwraca wartość indeksu określonego wymiaru. Prototyp metody index jest następujący:

```
function int_or_index_type index ( int dimension = 1 );
```

gdzie: *int_or_index_type* to typ zwracanej wartości; index to nazwa metody; *dimension* to numer wymiaru tablicy, domyślną wartością jest 1.

Typem zwracanym przez metodę index jest typ **int** dla wszystkich elementów iteratora tablicy, z wyjątkiem tablic asocjacyjnych, które zwracają indeks tego samego typu co typ indeksu asocjacyjnego. Na przykład:

```
int arr[];          // dynamiczna tablica liczb całkowitych
int q[$];           // kolejka liczb całkowitych

// znajdź wszystkie elementy równe ich pozycji (indeks)
q = arr.find with ( item == item.index );
```

7.5. Wnioski

Typ tablicy (spakowana lub rozpakowana) w języku SystemVerilog nie jest określany przez słowo kluczowe, lecz przez położenie w deklaracji zakresu względem nazwy tablicy. Jeśli przed nazwą tablicy znajduje się zakres, to jest ona spakowana i reprezentowana w pamięci jako wektor, czyli jako ciągła sekwencja bitów. Jeśli zakres występuje po nazwie tablicy w deklaracji tablicy, to jest ona rozpakowana. Elementy rozpakowanej tablicy nie muszą być ułożone w pamięci w kolejności, ich położenie określa kompilator.

Tablice rozpakowane (*unpacked array*) są tworzone z dowolnych typów danych, w tym z innych tablic spakowanych lub rozpakowanych. Tablice rozpakowane mogą być *tablicami o stałym rozmiarze* (*fixed-size arrays*), *tablicami dynamicznymi* (*dynamic arrays*), *tablicami asocjacyjnymi* (*associative arrays*) i *kolejkami* (*queues*).

Zarówno spakowane, jak i rozpakowane tablice mogą mieć wiele wymiarów, nazywane są wówczas *tablicami wielowymiarowymi*.

Tablica spakowana (*packed array*) to wektor bitów podzielony na *podpola* (*subfields*), do których można uzyskać dostęp jako do elementów tablicy. Gdy spakowana tablica jest przywoływana jako całość, jest traktowana jako pojedynczy wektor. Dlatego może być ona *ze znakiem* (**signed**) lub *bez znaku* (**unsigned**).

Poszczególne elementy tablicy są wartościami bez znaku, chyba że są to typy zdefiniowane przez użytkownika i zadeklarowane są ze znakiem. *Wybór części* (*part-select*) tablicy spakowanej jest zawsze bez znaku.

Tablice spakowane mogą zawierać pojedyncze dane bitowe typu **bit**, **logic** (**reg**), a także inne tablice i struktury spakowane. Takie tablice mogą być również tworzone z sieciowych typów danych (**wire**).

Do tablicy spakowanej można odwoływać się jako do całości, *selekcji bitu* (*bit-select*) lub *selekcji części* (*part-select*). Do wielowymiarowych tablic spakowanych można również odwoływać się jako do *fragmentu* (*slice*). Fragment (wycinek) to jeden lub więcej ciągłych wymiarów tablicy.

Każda operacja dozwolona na wektorach może być też wykonana na tablicach spakowanych.

Rozpakowane tablice mogą zawierać elementy dowolnego typu danych, które mogą być rozmieszczone w pamięci niezależnie od siebie; ich położenie określa kompilator. Rozpakowana tablica może być tablicą struktur lub innych typów zdefiniowanych przez użytkownika.

Zakresy tablic rozpakowanych mogą być reprezentowane przez rozmiar [n], czyli pojedynczą liczbę dodatnią, tak jak w języku C (tablice spakowane nie mogą być deklarowane tylko przez podanie ich rozmiaru).

Jeśli rozpakowana tablica jest zadeklarowana ze znakiem, to odnosi się on do elementów tablicy, a nie do całej tablicy.

Aby bezpośrednio skopiować wiele elementów do rozpakowanej tablicy, typ i rozmiar elementów, a także liczba kopiowanych wymiarów muszą być takie same.

Tablice spakowane można zainicjować za pomocą prostego przypisania, tak jak wektory po ich zadeklarowaniu.

Tablice rozpakowane po zadeklarowaniu mogą być inicjalizowane dla każdego wymiaru tablicy za pomocą wzorca przypisania.

Do tablicy spakowanej można przypisać dowolne wyrażenie wektorowe. Granice spakowanej tablicy nie mają wpływu na przypisanie.

Rozpakowana tablica lub jej fragment mogą zostać przypisane do innej rozpakowanej tablicy bądź jej fragmentu w przypadku spełnienia określonych warunków.

Każda tablica spakowana może być przypisana do innej tablicy spakowanej, a tablice mogą mieć różne rozmiary i typy.

Tablica rozpakowana może być bezpośrednio przypisana do innej tablicy rozpakowanej tylko wtedy, gdy obie mają ten sam rozmiar, ten sam rozmiar elementów i ten sam typ. Jeśli dwie rozpakowane tablice nie mają identycznego układu, przypisanie można wykonać za pomocą *rzutowania strumienia bitów* (*bit-stream casting*).

Tablicę rozpakowaną można przypisać do tablicy spakowanej, a tablicę spakowaną do tablicy rozpakowanej za pomocą rzutowania strumienia bitów.

W języku SystemVerilog można wykonywać następujące operacje na wszystkich tablicach spakowanych i rozpakowanych: odczyt i zapis tablicy; odczyt i zapis fragmentów tablicy; odczyt i zapis zmiennego fragmentu tablicy; odczyt i zapis elementu tablicy; porównanie całej tablicy lub fragmentu tablicy. Ponadto w przypadku tablic

spakowanych można dodatkowo wykonać przypisanie liczby całkowitej i przetwarzanie w wyrażeniu jako liczba całkowita.

Jednowymiarowa rozpakowana tablica z elementami typu **logic** (**reg**) lub **bit** jest nazywana *pamięcią* (*memory*). Element o wymiarze spakowanym w tablicy jest nazywany *elementem pamięci* (*memory element*) lub *słowem* (*word*).

Tablica wielowymiarowa (*multidimensional array*) to *tablica tablic* (*array of arrays*). Wymiary spakowane zmieniają się w niej szybciej niż wymiary rozpakowane. Na liście wymiarów wymiar najbardziej wysunięty na prawo zmienia się najszybciej.

Tablicę wielowymiarową można zdefiniować na podstawie innej tablicy za pomocą słowa kluczowego **typedef**.

Podtablica (*subarray*) to tablica, która jest elementem innej tablicy. Do podtablic odwołujemy się, pomijając indeksy jednego lub kilku wymiarów tablicy, zawsze ignorując te, które zmieniają się najszybciej.

Pojedynczy element w spakowanej lub rozpakowanej tablicy można wybrać za pomocą *nazwy indeksowanej* (*indexed name*).

W języku SystemVerilog istnieje rozróżnienie między pojęciami fragmentu (*slice*) i części (*part*) tablicy. Termin „wybór części" (*part-select*) odnosi się do wyboru jednego lub kilku *sąsiednich bitów* tablicy spakowanej o jednym rozmiarze. Termin „fragment" odnosi się do wyboru jednego lub kilku *sąsiednich elementów tablicy*.

Język SystemVerilog pozwala na przekazywanie dowolnej tablicy przez porty modułu oraz jako argumentów zadań i funkcji.

Tablice spakowane i rozpakowane mogą zawierać struktury i unie jako elementy tablicy. W spakowanej tablicy spakowana musi być także struktura lub unia. Z kolei struktury i unie mogą zawierać tablice spakowane lub rozpakowane. Struktura spakowana bądź unia mogą zawierać tylko tablice spakowane.

Język SystemVerilog udostępnia instrukcję **foreach** umożliwiającą iterację po elementach tablic jedno- i wielowymiarowych bez określania rozmiaru każdego wymiaru tablicy.

Konkatenacja rozpakowanych tablic jest podobna do schematu przypisania dla tablic, nie pozwala jednak na replikację.

Tablica dynamiczna (*dynamic array*) to rozpakowana tablica, której rozmiar można ustawiać lub zmieniać w czasie wykonywania zadania, tzn. w czasie symulacji. Elementami tablic dynamicznych mogą być zmienne wszystkich typów danych, a także inne tablice.

Tablica asocjacyjna (*associative array*) tworzy *tablicę przeglądową* (*lookup table*) elementów określonego typu. Typem danych używanym jako indeks jest *klucz wyszukujący* (*lookup key*).

Kolejka (*queue*) to uporządkowany zbiór jednakowych elementów o zmiennej wielkości. Umożliwia ona dostęp do wszystkich swoich elementów, a także wstawianie elementu na początku i usuwanie elementu z końca kolejki.

Tablice dynamiczne, tablice asocjacyjne i kolejki nie są syntetyzowalne.

Język SystemVerilog udostępnia szereg funkcji systemowych do obsługi tablic (**$left**, **$right**, **$increment**, **$low**, **$high**, **$size**, **$dimensions**, **$unpacked_dimensions**).

Funkcja systemowa **$bits** zwraca liczbę bitów dowolnego wyrażenia, w tym tablicy.

Język SystemVerilog udostępnia również szereg metod do manipulowania tablicami. Są to metody wyszukiwania (find, find_index, find_first, find_first_index, find_last, find_last_index, min, max, unique, unique_index); porządkowania (reverse, sort, rsort, shuffle); redukcji (suma, iloczyn, and, or, xor); oraz metoda zapytania o iterator indeksu (index).

8. Miejsca deklaracji elementów projektu

Język Verilog ma ograniczoną liczbę miejsc, w których deweloper może deklarować elementy projektu. Na przykład wszystkie zmienne, sieci, zadania i funkcje w Verilogu są deklarowane w module między słowami kluczowymi **module** i **endmodule** i mogą być używane tylko w tym module. Język Verilog pozwala także na stosowanie hierarchicznych nazw dla elementów różnych modułów. Nazwy hierarchiczne nie są obsługiwane jednak przy syntezie i mogą być używane tylko podczas symulacji.

Język Verilog nie pozwala także na globalne deklaracje dla całego projektu. Dlatego w każdym module należy zadeklarować konstrukcje używane w całym projekcie. Wprowadzenie zmian do takich wspólnych deklaracji wymaga modyfikacji wszystkich modułów i często prowadzi do błędów.

Aby zadeklarować obiekty używane przez wiele modułów, język SystemVerilog udostępnia pojęcie pakietów, które zostało zapożyczone z języka VHDL. Stałe, zmienne, typy użytkownika, zadania automatyczne i funkcje wystarczy zadeklarować w pakiecie jednokrotnie, a następnie można używać tych deklaracji w dowolnym module projektu.

Kolejnym problemem jest to, że w języku Verilog wszystkie moduły projektu są zawsze kompilowane w tym samym czasie. Jeśli brakuje jednego modułu, kompilator zgłasza błąd kompilacji. Język SystemVerilog jest przeznaczony do tworzenia dużych projektów składających się z dużej liczby modułów. Dlatego umożliwia on kompilację takich projektów także w częściach. W tym celu SystemVerilog wprowadza pojęcie *jednostki kompilacji* (*compilation unit*). Są to wszystkie pliki źródłowe kompilowane w tym samym czasie.

Gdy projekt składa się z wielu jednostek kompilacji, pakiety są importowane na początku każdej z nich i może pojawić się problem wielokrotnej deklaracji tych samych elementów projektu. Rozwiązaniem tego problemu jest *warunkowa kompilacja pakietów*.

Język SystemVerilog rozszerza również język Verilog o możliwość deklarowania zmiennych lokalnych w nienazwanych blokach proceduralnych.

Złożone projekty mogą być opracowywane przez różne zespoły projektowe. Dlatego identyfikatory (np. clk, clock, reset itp.) mogą być powielane. Aby rozwiązać ten problem, język SystemVerilog udostępnia osiem obszarów widoczności dla identyfikatorów.

Zmienne, zadania i funkcje w SystemVerilog mogą być statyczne i automatyczne, tak jak w językach programowania. Język SystemVerilog pozwala również na deklarowanie zmiennych, zadań i funkcji poza modułami, wewnątrz modułów, w pakietach, a także zmiennych w nazwanych i nienazwanych blokach proceduralnych. Zakres i czas życia zmiennych w SystemVerilog są określane przez miejsce deklaracji zmiennej oraz kwalifikator **static** lub **automatic**.

W dużych projektach ważny jest dostęp do każdego elementu projektu. Takim mechanizmem w SystemVerilog są *nazwy hierarchiczne*. Jest to sekwencja nazw komponentów projektu oddzielonych kropkami. Nazwy poszczególnych obiektów w nazwie hierarchicznej są określane jako *nazwy oddzielone kropkami* (*dotted names*).

Nazwy hierarchiczne tworzą drzewo hierarchii nazw projektów. Mogą być one używane do odwoływania się do obiektów projektu zarówno w dół, jak i w górę drzewa hierarchii nazw projektu.

Czasami konieczne jest oddzielenie kodu do symulacji od kodu projektu. Język SystemVerilog udostępnia konstrukcję **bind**, która jest używana do definiowania jednej lub więcej instancji modułu, interfejsu bądź programu weryfikacyjnego bez zmiany kodu projektu.

8.1. Pakiety

Język SystemVerilog umożliwia definiowanie typów użytkownika za pomocą słowa kluczowego **typedef**. Często typy te są wykorzystywane w kilku modułach. Dlatego naturalnym życzeniem deweloperów jest posiadanie mechanizmu, który umożliwia zadeklarowanie typów użytkownika tylko raz, a następnie wykorzystanie tych deklaracji w różnych modułach. Takim mechanizmem w języku SystemVerilog są *pakiety* (*packages*).

Pakiety mogą zawierać elementy zadeklarowane w innych pakietach. Są to jawnie nazwane obszary, które są widoczne na najwyższym poziomie, tzn. na poziomie modułu i prymitywu. Deklaracja pakietu tworzy zakres, który zawiera deklaracje przeznaczone do współużytkowania przez kilka jednostek kompilacji, modułów, interfejsów lub programów. Kompilacja pakietu musi być poprzedzona kompilacją zakresu, do którego pakiet jest importowany.

8.1.1. Definicja pakietu

Pakiet (*package*) to część deklaracji projektu, która może być używana w różnych modułach. Koncepcja pakietów została zapożyczona z języka VHDL. W SystemVerilog pakiety są definiowane pomiędzy słowami kluczowymi **package** i **endpackage**. Pakiety mogą zawierać następujące konstrukcje syntetyzowalne:

- definicje parametrów **parameter** i parametrów lokalnych **localparam**;

- definicje zmiennych **const**;
- typy zdefiniowane przez użytkownika za pomocą słowa kluczowego **typedef**;
- definicje zadań i funkcji automatycznych;
- operatory **import** do importowania deklaracji z innych pakietów.

Ponadto pakiety mogą zawierać następujące konstrukcje, które nie są syntetyzowalne:
- deklaracje zmiennych globalnych;
- statyczne definicje zadań i funkcji.

W kodzie projektu pakiet jest opisany poza jakimkolwiek modułem. Na przykład:

Przykład 8.1. Deklaracja pakietu *definitions*

```
package definitions;
        parameter VERSION = „1.1”;                  // deklaracja parametru

        typedef enum {ADD, SUB, MUL} opcodes_t;     // deklaracja typu wyliczeniowego

        typedef struct {                            // deklaracja struktury
                logic [31:0] a, b;
                opcodes_t opcode;
        } instruction_t;

        // deklaracja funkcji automatycznej
        function automatic [31:0] multiplier (input [31:0] a, b);
                return a * b;
        endfunction
endpackage
```

Należy pamiętać, że stałe **parameter** zdefiniowane w pakietach są traktowane w modułach jak parametry lokalne **localparam** i nie można ich przedefiniować.

8.1.2. Odniesienia do zawartości pakietu

Język SystemVerilog udostępnia cztery sposoby odwoływania się do elementów zadeklarowanych w pakiecie
- bezpośrednie odwołanie przy użyciu operacji *rozstrzygania kontekstu* (::);
- importowanie określonych elementów pakietu za pomocą operatora **import**;
- importowanie elementów pakietu z użyciem symbolu wieloznacznego (*);
- importowanie elementów pakietu do kontekstu *jednostki kompilacji* (**$unit**).

Język SystemVerilog zawiera operację *rozstrzygania kontekstu* (*scope resolution*) (::) pozwalającą na bezpośrednie odwoływanie się do elementów pakietu. Na przykład:

Przykład 8.2. Jawne odwołania do elementów pakietu *definitions* za pomocą operacji rozstrzygania kontekstu (::)

```
module ALU
        (input definitions::instruction_t IW,          // przesyłanie struktury przez port modułu
        input logic clock,
        output logic [31:0] result
        );

        always_ff @(posedge clock)
        begin
                case (IW.opcode)
                        // odniesienie do elementów pakietu
                        definitions::ADD:       result = IW.a + IW.b;
                        definitions::SUB:       result = IW.a - IW.b;
                        definitions::MUL:       result = definitions::multiplier(IW.a, IW.b);
                endcase
        end
endmodule
```

W tym przykładzie port wejściowy IW jest zdefiniowany za pomocą typu instruction_t, który został wcześniej zadeklarowany w pakiecie definitions. Operacja rozstrzygania kontekstu (::) jest używana do bezpośredniego odwoływania się do pakietu definicji. W ten sposób każdy element modułu może być zdefiniowany jako element pakietu. Przykładem są stałe elementy operatora **case** (definicje::ADD, definicje::SUB i definicje::MUL). W przykładzie pokazano również bezpośrednie odwołanie do funkcji zadeklarowanej w pakiecie (definicje::multiplier (IW.a, IW. b);).

Gdy trzeba wielokrotnie odwoływać się do elementów pakietu, użycie nazwy pakietu w operacji rozstrzygania kontekstu (::) może być zbyt czasochłonne. W takim przypadku poszczególne elementy pakietu można importować za pomocą operatora importu **import**. Na przykład:

Przykład 8.3. Importowanie do modułu określonych elementów pakietu

```
module ALU
        (input definitions::instruction_t IW,
        input logic clock,
        output logic [31:0] result
        );
        import definitions::ADD;              // importowanie elementów pakietu
        import definitions::SUB;
        import definitions::MUL;
        import definitions::multiplier;
```

```
        always_comb begin
                case (IW.opcode)
                        ADD :   result = IW.a + IW.b;
                        SUB :   result = IW.a - IW.b;
                        MUL : result = multiplier(IW.a, IW.b);
                endcase
        end
endmodule
```

W tym przykładzie etykiety wyliczeniowe ADD, SUB i MUL oraz nazwa funkcji mnożenia są importowane z pakietu definitions. Dzięki temu można zmniejszyć objętość kodu projektu.

Należy pamiętać, że podczas importowania elementów pakietu za pomocą operatora importu **import** każda etykieta typu wyliczeniowego musi być importowana osobno, tzn.

```
import definitions :: opcodes_t;
```

nie będzie działać.

Gdy w pakiecie jest zbyt wiele elementów do zaimportowania, również to może powodować pewne niedogodności w definicji. W takim przypadku można użyć znaku wieloznacznego * w instrukcji importu **import.** Na przykład:

Przykład 8.4. Importowanie pakietu przy użyciu symbolu wieloznacznego

```
module ALU
        (input definitions::instruction_t IW,
        input logic clock,
        output logic [31:0] result
        );

        import definitions::*; // zaimportowanie całego pakietu
                               // przy użyciu symbolu wieloznacznego
        always_comb
        begin
                case (IW.opcode)
                        ADD :   result = IW.a + IW.b;
                        SUB :   result = IW.a - IW.b;
                        MUL : result = multiplier(IW.a, IW.b);
                endcase
        end
endmodule
```

W tym przykładzie wszystkie elementy pakietu używane w module są automatycznie importowane do modułu ALU. Należy pamiętać, że elementy zadeklarowane w pakiecie, ale nieużywane w module, nie będą do niego importowane. Dlatego do importowania niezbędnych elementów zadeklarowanych w pakiecie można bezpiecznie użyć symbolu wieloznacznego *.

Cechą szczególną operacji importu elementów zadeklarowanych w pakiecie jest to, że przy określaniu typów portów należy zawsze używać jawnego odwołania z użyciem operacji rozstrzygania kontekstu (::). Na przykład:

```
import definitions :: instruction_t  IW, ...
```

Dzieje się tak, ponieważ składnia SystemVerilog nie pozwala na umieszczenie instrukcji importu **import** między słowem kluczowym **module** a deklaracją portu modułu.

Deklaracje lokalne w module mają pierwszeństwo przed deklaracjami importu z użyciem symboli wieloznacznych (*). Narzędzia projektowe będą szukać komponentów projektu najpierw wśród deklaracji modułów lokalnych, następnie wśród elementów importowanych z określonymi nazwami, następnie w pakietach, które zostały zaimportowane z użyciem symboli wieloznacznych, a na końcu w zakresie deklaracji **$unit** (jednostki kompilacji omówiono w rozdziale 8.2).

8.1.3. Zalecenia dotyczące syntezy

Aby zadania i funkcje zdefiniowane w pakiecie mogły być syntetyzowane, muszą być automatyczne i nie mogą zawierać zmiennych statycznych. Gdy moduł odwołuje się do zadania lub funkcji zdefiniowanej w pakiecie, syntezator powiela to zadanie bądź funkcję w module i traktuje je tak, jakby były zadeklarowane w module. Dzięki temu moduły mogą działać równolegle i niezależnie od siebie.

Inaczej jest w przypadku zmiennych zadeklarowanych w pakiecie. Wszystkie moduły widzą tę zmienną i każdy z nich może zmieniać jej wartość. Jest to dopuszczalne w symulacji, ale nie jest niedopuszczalne w syntezie. Dlatego zmienne zadeklarowane w pakietach nie podlegają syntezie.

Wniosek: Aby synteza była możliwa, wartości zmiennych muszą być przesyłane między modułami przez porty.

8.1.4. Wbudowany pakiet std

Język SystemVerilog udostępnia dodatkowo *pakiet wbudowany std* (*the std build-in package*), który może zawierać typy systemowe (np. klasy), zmienne, zadania i funkcje. Zawartość pakietu wbudowanego std jest zdefiniowana w standardzie języka SystemVerilog, ale każdy kompilator może zdefiniować swój własny pakiet wbudowany. Użytkownicy nie mogą wstawiać własnych deklaracji do pakietu std.

Wbudowany pakiet std jest niejawnie importowany za pomocą symbolu wieloznacznego * w kontekście każdej jednostki kompilacji. Dlatego deklaracje w pakiecie wbudowanym są dostępne w dowolnym kontekście, chyba że zostaną nadpisane przez kod użytkownika.

Do deklaracji we wbudowanym pakiecie std można się jawnie odwoływać. Na przykład:

```
std :: sys_task ();              // wywołanie zadania sys_task z pakietu std
```

W przeciwieństwie do zadań i funkcji systemowych zadania i funkcje w pakiecie wbudowanym std nie mają prefiksu $. Zalecamy, aby czytelnik zapoznał się z dokumentacją dotyczącą wbudowanego pakietu std swojego kompilatora.

8.2. Jednostki kompilacji

Jednostka kompilacji (*compilation unit*) to wszystkie pliki źródłowe, które są kompilowane w tym samym czasie. Może ona zawierać jeden lub więcej modułów, które mogą znajdować się w jednym bądź kilku plikach. Jednostka kompilacji w języku SystemVerilog jest często określana jako **$unit**.

8.2.1. Deklaracje zewnętrznych jednostek kompilacji

Język SystemVerilog rozszerza zakres deklaracji elementów projektu w porównaniu z językiem Verilog, umożliwiając umieszczanie deklaracji poza granicami pakietów, modułów, interfejsów i bloków programowych. Te zewnętrzne deklaracje znajdują się w zakresie (kontekście) jednostki kompilacji i są widoczne dla wszystkich modułów, które kompilują się w tym samym czasie.

Kontekst jednostki kompilacji może zawierać:

- deklaracje jednostek czasu (niesyntetyzowane);
- deklaracje zmiennych (niesyntetyzowane);
- deklaracje sieci (niesyntetyzowane);
- deklaracje stałych;
- typy danych zdefiniowane przez użytkownika;
- definicje typów wyliczeniowych;
- definicje klas (niesyntetyzowane);
- definicje zadań i funkcji (syntetyzowane są tylko automatyczne zadania i funkcje, które nie zawierają zmiennych statycznych).

Poniższy przykład zawiera deklaracje zewnętrznych jednostek kompilacji.

Przykład 8.5. Deklaracje zewnętrzne w kontekście jednostki kompilacji (niesyntetyzowalne)

```
/******************* Deklaracje zewnętrzne *******************/
parameter VERSION = „1.2a”;          // stała zewnętrzna

reg resetN = 1;                      // zmienna zewnętrzna z aktywnym poziomem niskim

typedef struct packed {              // zewnętrzny typ użytkownika
        reg [31:0] address;
        reg [31:0] data;
        reg [ 7:0] opcode;
} instruction_word_t;

function automatic int log2 (input int n);    // funkcja zewnętrzna
        if (n <=1) return(1);
        log2 = 0;
        while (n > 1) begin
                n = n/2;
                log2++;
        end
        return(log2);
endfunction

/********************* definicja modułu *********************/
module register
        (output instruction_word_t q,        // używane są deklaracje zewnętrzne podczas
                                             // definiowania typów portów
        input instruction_word_t d,
        input wire clock );

        always @(posedge clock, negedge resetN)
                if (!resetN) q <= 0;         // użycie zmiennej zewnętrznej resetN
                else q <= d;
endmodule
```

Należy pamiętać, że deklaracje kontekstu jednostki kompilacji nie są globalne, lecz obowiązują tylko w obrębie danej jednostki kompilacji. Na przykład jeśli projekt składa się z dwóch jednostek kompilacji, w każdej z nich zadeklarowano zewnętrzną zmienną *reset*, odpowiada to dwóm różnym sygnałom resetowania *reset* w projekcie.

Twórcy języka SystemVerilog nie zalecają tworzenia zewnętrznych deklaracji w kontekście jednostki kompilacji, ponieważ może to prowadzić do błędów projektowych, jak w powyższym przykładzie. Ponadto jeśli deklaracje są rozproszone w wielu plikach, modyfikowanie deklaracji i poprawianie w nich błędów może być trudne.

8.2.2. Jawny dostęp do deklaracji zewnętrznych

Czasami, aby wyeliminować niejednoznaczność, konieczne jest wskazanie na deklarację zewnętrznego identyfikatora. Służy do tego specyfikator **$unit**, który jednoznacznie wskazuje, że dany identyfikator jest zewnętrzną deklaracją jednostki kompilacji. Na przykład:

```
bit b;                          // deklaracja zewnętrzna

task t;
        int b;                  // deklaracja lokalna
        b = 5 + $unit::b;       // $unit::b jest deklaracją zewnętrzną
endtask
```

8.2.3. Importowanie pakietów do jednostek kompilacji

Zamiast zewnętrznych deklaracji w jednostkach kompilacji zaleca się tworzenie deklaracji w nazwanych pakietach, a następnie importowanie pakietów do jednostek kompilacji. Ta ostatnia funkcja umożliwia np. deklarowanie typów portów modułu za pomocą typów zdefiniowanych przez użytkownika.

W punkcie 8.1 podano następujący przykład deklaracji portu modułu ALU:

```
module ALU
        (input definitions::instruction_t IW,
        input logic clock,
        output logic [31:0] result);
```

Powyższy sposób deklaracji portu IW wynika z faktu, że operator importu **import** nie może być wstawiony bezpośrednio po słowie kluczowym **module**.

W przypadku importowania pakietu w jednostce kompilacji kod może wyglądać tak, jak poniżej:

```
// importowanie elementów pakietu do jednostki kompilacji
import definitions::instruction_t;

module ALU
        (input instruction_t IW,
        input logic clock,
        output logic [31:0] result);
```

Pakiet w jednostce kompilacji można również zaimportować, używając symbolu wieloznacznego *. Na przykład:

```
import definitions::*;

module ALU
        (input instruction_t IW,
        input logic clock,
        output logic [31:0] result);
```

Należy pamiętać, że operator importu **import** w jednostce kompilacji musi poprzedzać moduły, w których używane są deklaracje pakietów.

8.2.4. Warunkowa kompilacja pakietów

Gdy jednostką kompilacji jest pojedynczy plik źródłowy, importowanie pakietów nie stanowi większego problemu. Wystarczy zaimportować pakiet na początku pliku, a deklaracje pakietów są widoczne we wszystkich modułach jednostki kompilacji.

Jeśli jednak projekt składa się z kilku plików, mogą one być kompilowane wszystkie razem (jak w Verilogu) lub w częściach (jak w SystemVerilog). Gdy pliki są kompilowane oddzielnie, zostanie utworzonych kilka odrębnych jednostek kompilacji. Jednak zaimportowanie pakietu w jednej jednostce kompilacji nie będzie widoczne w innej jednostce kompilacji.

Ten ostatni problem można rozwiązać, importując pakiet na początku każdego pliku jednostki kompilacji. Wiąże się to jednak z ponownym deklarowaniem elementów projektu. Rozwiązaniem tego problemu jest warunkowa kompilacja pakietów.

Istotą *kompilacji warunkowej pakietów* (*package conditional compilation*) jest użycie dyrektyw kompilacji **`ifndef**, **`define** i **`include**. Dyrektywa **`ifndef** służy do sprawdzania, czy w kodzie źródłowym nie ma flagi oznaczającej, że dana deklaracja nie została jeszcze wykonana. Jeśli ta flaga nie jest ustawiona, to wykonuje się ustawianie flagi, deklaracja pakietu jest wstawiana do kodu źródłowego i pakiet jest importowany do jednostki kompilacji; w przeciwnym razie nic nie jest wstawiane do kodu źródłowego. Dyrektywa **`define** jest używana do ustawiania flagi, a dyrektywa **`include** jest używana do dołączania niezbędnych opisów na początku każdego pliku źródłowego projektu. Na przykład:

Przykład 8.6. Pakiet z kompilacją warunkową

```
`ifndef DEFS_DONE                   // jeśli flaga DEFS_DONE nie jest ustawiona
        `define DEFS_DONE           // ustawienie flagi DEFS_DONE
        package definitions;        // definicja pakietu definitions

                parameter VERSION = "1.1";

                typedef enum {ADD, SUB, MUL} opcodes_t;
```

```
        typedef struct {
            logic [31:0] a, b;
            opcodes_t opcode;
            } instruction_t;

        function automatic [31:0] multiplier (input [31:0] a, b);
            return a * b;
        endfunction
    endpackage

    import definitions::*; // zaimportowanie pakietu do jednostki kompilacji
`endif
```

Załóżmy, że kod z przykładu 8.6 został zapisany w pliku o nazwie definitions.pkg. Następnie, aby włączyć ten kod do jednostek kompilacji, należy umieścić następujący wiersz na początku każdego pliku źródłowego projektu:

```
`include "definitions.pkg"
```

W przykładach 8.7 i 8.8 pokazano moduł ALU i jednostkę testową modułu ALU z wykorzystaniem warunkowej kompilacji pakietu z przykładu 8.6.

Przykład 8.7. Kod projektu zawierający plik z warunkową kompilacją pakietu

```
`include „definitions.pkg"          // plik include z warunkową kompilacją pakietu

module ALU
    (input instruction_t IW,
    input logic clock,
    output logic [31:0] result
    );

    always_comb begin
        case (IW.opcode)
            ADD : result = IW.a + IW.b;
            SUB : result = IW.a - IW.b;
            MUL : result = multiplier(IW.a, IW.b);
        endcase
    end
endmodule
```

Przykład 8.8. Kod jednostki testowej zawierający plik z warunkową kompilacją pakietu

```
`include „definitions.pkg”          // plik include z warunkową kompilacją pakietu

module test;
        instruction_t test_word;
        logic [31:0] alu_out;
        logic clock = 0;

        ALU dut (.IW(test_word), .result(alu_out), .clock(clock));

        always #10 clock = ~clock;

        initial begin
                @(negedge clock)
                test_word.a = 5;
                test_word.b = 7;
                test_word.opcode = ADD;
                @(negedge clock)
                $display(“alu_out = %d (expected 12)”, alu_out);
                $finish;
        end
endmodule
```

8.3. Deklaracje w blokach proceduralnych

Język Verilog pozwala na deklarowanie zmiennych lokalnych wewnątrz nazwanych bloków proceduralnych **begin...end** lub **fork...join**, których zakres jest ograniczony do obszaru wewnątrz bloku. W poniższym fragmencie kodu zadeklarowane są dwie zmienne i nie prowadzi to do błędu, ponieważ druga zmienna i jest zadeklarowana w bloku o nazwie loop.

```
module chip (input clock);
        integer i;                  // deklaracja na poziomie modułu

        always @(posedge clock)
        begin: loop                 // blok nazwany loop

                integer i;          // zmienna lokalna
                for (i=0; i<=127; i=i+1) begin
                ...
```

```
        end
    end
endmodule
```

Do zmiennych zadeklarowanych w nazwanych blokach proceduralnych można się odwoływać, używając nazwy hierarchicznej (pamiętając, że nazwy hierarchiczne nie są syntetyzowalne).

Język SystemVerilog rozszerza język Verilog o możliwość deklarowania zmiennych lokalnych także w nienazwanych blokach proceduralnych. Na przykład:

```
module chip (input clock);
    integer i;                  // deklaracja na poziomie modułu

    always @(posedge clock)
    begin                       // jednostka nienazwana

        integer i;              // zmienna lokalna
        for (i=0; i<=127; i=i+1) begin
        ...
        end
    end
endmodule
```

Jednak do zmiennych zadeklarowanych w nienazwanych blokach proceduralnych nie można się odwoływać, używając nazwy hierarchicznej.

8.4. Obszary widoczności identyfikatorów

Język SystemVerilog ma osiem *obszarów widoczności* (*zakresów*) identyfikatorów lub inaczej *przestrzeni nazw* (*name space*).

Dwie z nich są globalne dla całego projektu:

- *przestrzeń nazw definicji* (*definitions name space*);
- *przestrzeń nazw pakietów* (*package name space*).

Dwa zakresy są globalne dla jednostki kompilacji:

- *przestrzeń nazw zakresu jednostki kompilacji* (*compilation-unit scope name space*);
- *przestrzeń nazw makr tekstowych* (*text macro name space*).

Cztery zakresy są lokalne:

- *przestrzeń nazw modułów* (*module name space*);
- *przestrzeń nazw bloków* (*block name space*);

- *przestrzeń nazw portów* (*port name space*);
- *przestrzeń nazw atrybutów* (*attribute name space*).

Nazwy zadeklarowane w *zakresach globalnych* dla całego projektu są widoczne w jego wszystkich jednostkach kompilacji. Nie wolno ich ponownie deklarować w poszczególnych jednostkach kompilacji lub pakietach. *Zakres nazw definicji* obejmuje wszystkie identyfikatory zagnieżdżonych modułów, prymitywów, programów i interfejsów. *Zakres nazw pakietów* obejmuje identyfikatory wszystkich pakietów, które są zdefiniowane we wszystkich jednostkach kompilacji.

Nazwy zadeklarowane w *zakresie globalnym dla jednostki kompilacji* są widoczne w obrębie jednostki kompilacji, w której zostały zadeklarowane. Obiekty o tej samej nazwie mogą być zadeklarowane w innej jednostce kompilacji. *Zakres nazw kontekstu jednostki kompilacji* łączy definicje funkcji, zadań, kontrolerów, parametrów, nazwanych zdarzeń, deklaracji elementów sieciowych, zmiennych i typów użytkownika, które są zdefiniowane poza modułami, interfejsami, pakietami, kontrolerami, programami i prymitywami. *Zakres nazw makr tekstowych* (zdefiniowanych za pomocą poprzedzającego znaku ') łączy nazwy wszystkich makr w obrębie danej jednostki kompilacji. Nazwy makr tekstowych są definiowane w kolejności występowania w zbiorze plików wejściowych tworzących jednostkę kompilacji i mogą być zastępowane przez kolejne definicje.

Nazwy zadeklarowane w *zakresach lokalnych* są unikatowe w obrębie danego konstruktu zamkniętego. *Zakres nazw modułów* łączy definicje modułów, interfejsów, programów, kontrolerów, funkcji, zadań, nazwanych bloków, nazw instancji, parametrów, nazwanych zdarzeń, deklaracji elementów sieciowych, zmiennych i typów użytkownika zadeklarowanych w modułach, interfejsach, pakietach, programach, kontrolerach i prymitywach. *Zakres nazw bloków* obejmuje nazwy bloków, funkcji, zadań, parametrów, nazwanych zdarzeń, zmiennych i typów użytkownika zadeklarowanych w nazwanych i nienazwanych blokach, a także w określonych konstrukcjach **specify**, zadaniach i funkcjach. *Zakres nazw portów* jest wprowadzany przez moduły, interfejsy, prymitywy i programy. Zapewnia ona strukturalne połączenie między dwoma obiektami znajdującymi się w różnych przestrzeniach nazw. Przestrzeń nazw portów pokrywa się z przestrzenią nazw modułów i bloków. *Zakres nazw atrybutów* jest dołączany do elementu języka, w którym atrybut jest zdefiniowany, za pomocą konstrukcji (* i *).

8.5. Nazwy hierarchiczne i oddzielone kropkami

Każdy identyfikator w języku SystemVerilog ma unikatową nazwę hierarchiczną. Hierarchia nazw identyfikatorów jest strukturą drzewiastą, w której każdy moduł, zadanie, funkcja lub jednostka nazwanego bloku proceduralnego definiuje węzeł nowego poziomu hierarchii, a liście hierarchicznego drzewa nazw są identyfikatorami.

Nienazwane bloki generujące (*unnamed generate blocks*) tworzą gałęzie, które są widoczne tylko wewnątrz bloku proceduralnego. Korzeniem drzewa hierarchii nazw jest moduł najwyższego poziomu projektu. W przypadku wielu modułów najwyższego poziomu w opisie projektu tworzy się wiele hierarchii nazw.

Do każdego identyfikatora projektu można się jednoznacznie odwołać z dowolnego miejsca w projekcie za pomocą nazwy hierarchicznej. Hierarchiczna składnia nazw jest definiowana następująco:

[**$root**.] {*node_name*.} *identifier*

gdzie: **$root** odnosi się do korzenia drzewa hierarchicznego; *node_name* jest węzłem w ścieżce drzewa hierarchicznego; *identifier* jest identyfikatorem.

Innymi słowy, nazwa identyfikatora hierarchicznego to ścieżka w drzewie hierarchii, gdzie węzły drzewa są oddzielone kropkami. Nazwa hierarchiczna może zaczynać się od korzenia drzewa, w tym przypadku używane jest słowo kluczowe **$root**.

Ogólnie rzecz biorąc, nazwy hierarchiczne składają się z nazw instancji oddzielonych kropkami, przy czym nazwa instancji może być elementem tablicy. Na przykład:

```
$root.mymodule.u1          // nazwa bezwzględna
u1.struct1.field1          // u1 może być widoczne lokalnie lub powyżej
adder1[5].sum              // adder1[5] – nazwa elementu tablicy sumatorów
```

Pełna nazwa ścieżki dowolnego obiektu projektu zaczyna się od modułu najwyższego poziomu. Ta nazwa ścieżki może być używana na dowolnym poziomie hierarchii lub w hierarchii równoległej (w przypadku wielu modułów najwyższego poziomu). Pierwsza nazwa węzła w nazwie ścieżki może zaczynać się na poziomie, na którym ścieżka jest używana do dostępu do obiektów w dół drzewa hierarchii. Nazwy hierarchicznej nie można użyć w celu uzyskania dostępu do obiektów w zadaniach i funkcjach automatycznych oraz do obiektów w nienazwanych blokach generacyjnych.

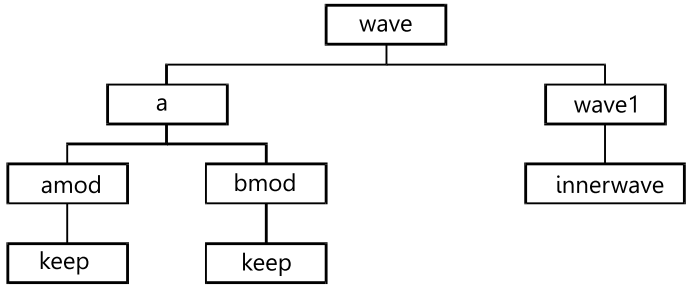
W przykładzie 8.9 pokazano, jak utworzyć drzewo hierarchiczne.

Przykład 8.9. Hierarchia instancji modułów i nazwanych bloków

```
module cct (stim1, stim2);
        input stim1, stim2;

        mod amod(stim1),bmod(stim2);      // tworzenie instancji modułu mod
endmodule

module mod (in);
        input in;
```

```
        always @(posedge in) begin: keep
                logic hold;
                hold = in;
        end
endmodule

module wave;
        logic stim1, stim2;

        cct a(stim1, stim2);                    // utwórz instancję modułu cct

        initial begin :wave1
                #100  fork: innerwave
                                reg hold;
                      join
                #150  begin
                                stim1 = 0;
                      end
        end
endmodule
```

Rysunek 8.1 przedstawia drzewo hierarchii nazw z przykładu 8.9.



RYS. 8.1. Drzewo hierarchii nazw z przykładu 8.9

ŹRÓDŁO: oprac. własne.

Nazwa hierarchiczna oraz selekcja elementu struktury, unii lub klasy mają tę samą formę składniową, czyli jest to sekwencja nazw składników oddzielonych kropkami. Są one nazywane *nazwami oddzielonymi kropkami* (*dotted names*). Kropki służą do określenia, czy dana nazwa jest *nazwą hierarchiczną* (*hierarchical name*), czy *wyborem elementu* lub *członka* (*member select*). Aspektem wyróżniającym *nazwę hierarchiczną* jest to, że pierwszy składnik nazwy musi odpowiadać nazwie zakresu, natomiast pierwszy składnik nazwy *wyboru członka* musi odpowiadać nazwie obiektu danych lub portu interfejsu.

To, czy nazwa obiektu jest nazwą hierarchiczną, czy wyborem członka, jest określane automatycznie przez kompilator na podstawie kodu projektu. Na przykład:

```
package p;
        struct { int x; } s1;
        struct { int x; } s2;
        function void f();
                int x;
        endfunction
endpackage

module m;
        import p::*;
        if (1) begin : s1
                initial begin
                        s1.x = 1;       // nazwa 1
                        s2.x = 1;       // nazwa 2
                        f.x = 1;        // nazwa 3
                        f2.x = 1;       // nazwa 4
                end
                int x;                  // zmienna lokalna
                some_module s2();
        end
endmodule
```

8.6. Odwoływanie się do nazw w górę hierarchii

Język SystemVerilog pozwala także na odwoływanie się do nazw obiektów projektu znajdujących się wyżej w drzewie hierarchii nazw. W tym celu stosuje się następującą składnię:

module_name.item_name,

gdzie *module_name* jest nazwą modułu, w którym znajduje się obiekt, a *item_name* jest nazwą obiektu.

Odwoływanie się do nazw w górę można również wykonać przy użyciu następującej składni:

scope_name.item_name,

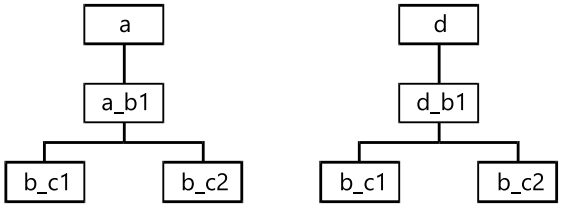
gdzie *scope_name* jest nazwą modułu, podprogramu (zadania lub funkcji), instancji interfejsu bądź jednostki generującej, a *item_name* jest nazwą obiektu.

W poniższym przykładzie znajdują się cztery moduły: a, b, c i d. Każdy z nich zawiera liczbę całkowitą i. Moduły położone na najwyższym poziomie w tym segmencie hierarchii modelu to a i d. Istnieją dwie kopie modułu b, ponieważ moduły a i d tworzą instancję modułu b. Są cztery kopie c.i, ponieważ każda z dwóch kopii b jest tworzona dwa razy.

Przykład 8.10. Odwołania do zmiennej *i* w drzewie hierarchii nazw

```
module a;                       // moduł a
        integer i;
        b a_b1();               // tworzenie instancji a_b1 modułu b
endmodule

module b;                       // moduł b
        integer i;
        c b_c1(),               // tworzenie dwóch instancji b_c1
          b_c2();               // i b_c2 modułu c
        initial                 // ścieżka w dół odnosi się do dwóch egzemplarzy i:
                #10 b_c1.i = 2; // a.a_b1.b_c1.i, d.d_b1.b_c1.i
endmodule

module c;                       // moduł c
        integer i;
        initial begin           // nazwa lokalna odnosi się do czterech instancji i:
                i = 1;          // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
                                // d.d_b1.b_c1.i, d.d_b1.b_c2.i
                b.i = 1;        // ścieżka rosnąca odnosi się do dwóch egzemplarzy i:
                                // a.a_b1.i, d.d_b1.i
        end
endmodule

module d;                       // moduł d
        integer i;
        b d_b1();               // tworzenie instancji d_b1 modułu b
        initial begin           // pełna nazwa ścieżki odnosi się do każdej kopii i
                a.i = 1; d.i = 5;
                a.a_b1.i = 2; d.d_b1.i = 6;
                a.a_b1.b_c1.i = 3; d.d_b1.b_c1.i = 7;
                a.a_b1.b_c2.i = 4; d.d_b1.b_c2.i = 8;
        end
endmodule
```

Rysunek 8.2 przedstawia drzewo hierarchii nazw z przykładu 8.6.



RYS. 8.2. Drzewo hierarchii nazw instancji modułu z przykładu 8.6

ŹRÓDŁO: oprac. własne.

Znajdowanie nazw zadań i funkcji jest oparte na podobnej zasadzie co wyszukiwanie nazw hierarchicznych w górę drzewa hierarchii.

8.7. Wiązanie kodu pomocniczego z kontekstami lub instancjami modułów

Czasami konieczne jest oddzielenie kodu do symulacji od kodu projektu do syntezy. Język SystemVerilog udostępnia konstrukcję wiązania **bind**, która jest używana do definiowania jednej lub więcej instancji modułu, interfejsu bądź programu weryfikacyjnego bez zmiany kodu projektu (docelowego). Dyrektywa **bind** może być zdefiniowana dla modułu, interfejsu albo zakresu (kontekstu) jednostki kompilacji.

Istnieją dwie formy składni **bind**. Pierwsza definiuje *kontekst obiektu docelowego* (*bind target scope*), do którego należy wstawić instancję wiązania. Druga forma składni wiązania jest używana do zdefiniowania *pojedynczej instancji obiektu docelowego*, do której ma zostać wstawiona instancja wiązania.

Przykład powiązania instancji programu weryfikacyjnego z modułem:

```
bind cpu cpu_program cpu_rules_1 (a, b, c);
```

gdzie: *cpu* jest nazwą modułu docelowego; *cpu_program* jest nazwą programu powiązanego z modułem; *cpu_rules_1* jest nazwą instancji programu. W każdej instancji modułu *cpu* zostanie utworzona instancja programu *cpu_rules_1*. Pierwsze trzy porty programu *cpu_program* są powiązane z obiektami a, b i c w module *cpu*.

Przykład powiązania instancji programu z konkretną instancją modułu:

```
bind cpu : cpu1 cpu_program cpu_rules_1 (a, b, c);
```

Ten przykład wiąże instancję *cpu_rules_1* z instancją *cpu1* modułu *cpu*.

Przykład łączenia instancji programu z wieloma instancjami modułu:

```
bind cpu : cpu1, cpu2, cpu3 cpu_program cpu_rules_1 (a, b, c);
```

W tym przykładzie instancja *cpu_rules_1* jest powiązana z instancjami *cpu1*, *cpu2* i *cpu3* modułu *cpu*.

Program połączony z modułem lub jego instancją staje się częścią *obiektu związanego* (*bound object*). Do nazw deklaracji w programie można się odwoływać za pomocą nazw hierarchicznych języka SystemVerilog.

Przykład powiązania instancji interfejsu z modułem:

```
interface range (input clk, enable, input var int minval, expr);
        property crange_en;
                @(posedge clk) enable |-> (minval <= expr);
        endproperty
        range_chk: assert property (crange_en);
endinterface

bind cr_unit range r1(c_clk,c_en,v_low,(in1&&in2));
```

W tym przykładzie instancja *r1* interfejsu *range* jest powiązana z modułem *cr_unit*.

8.8. Wnioski

Pakiet (*package*) jest częścią deklaracji projektu, która może być używana w różnych modułach. Pakiety są definiowane między słowami kluczowymi **package** i **endpackage**.

Pakiety mogą zawierać definicje parametrów **parameter**, parametrów lokalnych **localparam** i stałych **const**, typy zdefiniowane przez użytkownika, zadania i funkcje automatyczne oraz operatory importu **import** umożliwiające importowanie deklaracji z innych pakietów. Ponadto w skład pakietów mogą wchodzić niezsyntetyzowane deklaracje zmiennych globalnych oraz statyczne definicje zadań i funkcji.

Do elementów zadeklarowanych w pakiecie można się odwoływać za pomocą operacji *rozstrzygania kontekstu* (::), operatora importu **import**, symbolu wieloznacznego (*) oraz przez importowanie elementów pakietu do kontekstu *jednostki kompilacji*.

Aby zadania i funkcje zdefiniowane w pakiecie mogły być syntetyzowane, muszą być automatyczne i nie mogą zawierać zmiennych statycznych.

Zmienne zadeklarowane w pakietach nie są syntetyzowalne. W związku z tym, aby umożliwić syntezę, wartości zmiennych muszą być przekazywane między modułami za pośrednictwem ich portów.

Wbudowany pakiet std języka SystemVerilog zawiera typy i zmienne systemowe oraz zadania i funkcje. Jest on niejawnie importowany za pomocą symbolu wieloznacznego * w zakresie każdej jednostki kompilacji. Zadania i funkcje wbudowanego pakietu std nie mają prefiksu $.

Jednostka kompilacji (**$unit**) to wszystkie pliki źródłowe, które są kompilowane w tym samym czasie.

Kontekst jednostki kompilacji może zawierać następujące *deklaracje zewnętrzne*: deklaracje stałych, typy danych zdefiniowane przez użytkownika, definicje typów wyliczeniowych, definicje zadań i funkcji (syntetyzowane są tylko zadania automatyczne i funkcje niezawierające zmiennych statycznych), deklaracje jednostek czasu (niesyntetyzowane), deklaracje zmiennych (niesyntetyzowane), deklaracje sieci (niesyntetyzowane), definicje klas (niesyntetyzowane).

Deklaracje kontekstu jednostki kompilacji nie są globalne, mają zastosowanie tylko w obrębie jednostki kompilacji.

Zamiast tworzyć zewnętrzne deklaracje w jednostkach kompilacji, zaleca się tworzenie deklaracji w nazwanych pakietach, a następnie importowanie pakietów do jednostek kompilacji.

Problem wielu plików jednostek kompilacji oraz niebezpieczeństwo ponownego deklarowania elementów projektu rozwiązuje warunkowa kompilacja pakietów.

Język SystemVerilog rozszerza język Verilog o możliwość deklarowania zmiennych lokalnych w nienazwanych blokach. Do zmiennych zadeklarowanych w nienazwanych blokach nie można się odwoływać, używając nazwy hierarchicznej.

Język SystemVerilog ma osiem zakresów identyfikatorów lub *przestrzeni nazw* (*name space*). Zakres zmiennej nazywany jest *kontekstem* (*scope*).

Każdy identyfikator w SystemVerilog ma unikatową nazwę hierarchiczną. Hierarchia nazw identyfikatorów jest strukturą drzewiastą, w której każda instancja modułu, zadania, funkcji lub bloku proceduralnego definiuje węzeł nowego poziomu hierarchii, a liście hierarchicznego drzewa nazw są identyfikatorami. Korzeniem drzewa hierarchii nazw jest moduł projektu najwyższego poziomu.

Do każdego identyfikatora projektu można jednoznacznie odwołać się z dowolnego miejsca w projekcie, używając nazwy hierarchicznej. Słowo kluczowe **$root** odnosi się do korzenia drzewa hierarchii nazw.

Nazwy hierarchiczne składają się z nazw instancji oddzielonych kropkami, przy czym nazwa instancji może być elementem tablicy.

Nazwy komponentów w nazwie hierarchicznej są nazywane *nazwami oddzielonymi kropkami* (*dotted names*).

Język SystemVerilog pozwala na powiązania z nazwami obiektów projektu znajdującymi się wyżej w drzewie hierarchii nazw.

Konstrukcja **bind** jest używana do definiowania jednej lub więcej instancji modułu, interfejsu bądź programu weryfikacyjnego bez zmiany kodu projektu.

Program związany z modułem albo instancją modułu staje się częścią *obiektu związanego* (*bound object*).

9. Operatory i wyrażenia

W porównaniu z językiem Verilog operatory i wyrażenia używane w SystemVerilog wykorzystują nowe typy danych. Nowością są wyrażenia agregujące, w których można stosować struktury spakowane i tablice.

W języku SystemVerilog wprowadzono także szereg nowych operatorów. Umożliwiają one następujące operacje: przypisania arytmetycznego i bitowego w języku C, przypisania arytmetycznego i przesunięcia logicznego, równości z symbolami wieloznacznymi oraz inkrementacji i dekrementacji. Pierwotnymi operatorami języka SystemVerilog były: operator przynależności do zbioru **inside**, operator alokacji i operator wątku. Dodatkowo znacznie rozszerzono w nim także możliwości stosowania typów łańcuchowych w wyrażeniach.

Jednak nie wszystkie z tych nowych operacji są syntetyzowalne. Wyjaśnienia można znaleźć w dokumentacji używanego kompilatora.

Dodatkowo w rozdziale omówiono konstrukcję **let**, która w SystemVerilog ma zastąpić makra tekstowe (makrodefinicje) języka Verilog. Ich główną wadą jest to, że działają one globalnie na kodzie źródłowym całego projektu, co nie zawsze jest wygodne (lub wręcz niemożliwe) w dużym projekcie. Konstrukcja **let** działa lokalnie. Ponadto może być deklarowana w pakietach, co znacznie ułatwia tworzenie kodu źródłowego dla dużych projektów.

9.1. Wyrażenia, operatory i operandy

Wyrażenie (*expression*) w SystemVerilog to konstrukcja, która łączy *operandy* (*operands*) i *operatory* (*operators*) w celu uzyskania wyniku. Wyrażeniem jest również każdy operand bez operatora, np. wybór bitu. Wyrażenia łącznie z operatorami SystemVerilog mogą być używane wszędzie tam, gdzie wymagane jest uzyskanie jakiejś wartości.

Operandami w SystemVerilog mogą być:

- literał numeryczny;
- ciąg znaków (łańcuch);
- parametr, w tym parametry lokalne i parametry specyfikacji;
- element sieci;
- zmienna;
- struktura spakowana lub rozpakowana;

- element struktury;
- unia spakowana, rozpakowana lub oznaczona (tagged);
- element unii;
- tablica spakowana lub rozpakowana;
- wywołanie funkcji, funkcji systemowej lub metody zwracającej dowolny z powyższych obiektów.

Operandami mogą być także *wybór bitu* (*bit-select*) lub *wybór części* (*part-select*) parametru, sieci, zmiennej, struktury, spakowanej unii bądź tablicy.

9.2. Wyrażenia stałe

Niektóre instrukcje SystemVerilog wymagają *wyrażeń stałych* (*constant expressions*), czyli takich, które zwracają wartość stałą. Operandami wyrażenia stałego mogą być tylko stałe (liczby, łańcuchy, parametry, wywołania funkcji stałej), a także wybory bitów i części stałych. W wyrażeniach stałych można stosować dowolne operacje dozwolone w języku SystemVerilog.

W wyrażeniach stałych można używać funkcji systemowych, których argumentami są stałe. Są to funkcje systemowe służące do konwersji (*conversion*), funkcje matematyczne i przeznaczone dla wektorów bitowych. Ponadto w wyrażeniach stałych mogą być używane funkcje systemowe zapytań o dane (*data query*) i zapytań o tablice (*array query*), nawet jeśli ich argumenty nie są stałe.

9.3. Wyrażenia agregujące

Obiekty danych rozpakowanych struktur i tablic oraz *konstruktory* (*constructors*) rozpakowanych struktur i tablic mogą być używane jako *wyrażenia agregujące* (*aggregate expressions*). Jako takie można też wykorzystać wieloelementowe fragmenty rozpakowanej tablicy.

Wyrażenia agregujące można kopiować za pomocą instrukcji przeznaczenia i przenosić do podprogramów, można je również porównywać za pomocą operacji równości i nierówności.

Przykłady wyrażeń agregujących:

```
int [7:0] a, b;   // tablice 8-bitowe spakowane
int [15:0] c;     // tablica 16-bitowa spakowana
int s, t;

a = b;            // kopiowanie tablicy
```

```
c[7:0] = a;       // zapis części tablicy
c[15:8] = b;      // to samo

s = 7; t = 8;
c[s-:t] = a;      // tak samo jak c[7:0] = a;

s = 8;
c[s+:t] = b;      // tak samo jak c[15:8] = b;

c[0] = a[7];      // zapis jednego elementu tablicy

c = {a,b};        // konkatenacja tablic

c = a + b + 7;    // wyrażenie arytmetyczne z tablicami

s = (a==b);       // wyrażenie logiczne z tablicami
```

9.4. Operatory

Operatory (*operators*) określają działania, które mają być wykonywane w wyrażeniu na operandach. Symbole operatorów w SystemVerilog są w większości przypadków takie same jak w języku C (tabela 9.1).

TABELA 9.1. Operatory w języku SystemVerilog

Symbol operatora	Nazwa	Typy danych operandów
=	binarna operacja przypisania	każdy
+= -= /= *=	binarne operacje arytmetyczne przypisania	całkowity, **real**, **shortreal**
%=	binarna operacja przypisania modułu arytmetycznego	całkowity
&= |= ^=	binarne operacje przypisania operacji bitowych	całkowity
>>= <<=	binarne operacje przypisania przesunięcia logicznego	całkowity
>>>= <<<=	binarne operacje przypisania przesunięcia arytmetycznego	całkowity
?:	operacja warunkowa	każdy
+ -	jednoargumentowe operacje arytmetyczne	całkowity, **real**, **shortreal**
!	jednoargumentowa operacja negacji logicznej	całkowity, **real**, **shortreal**
~ & ~& | ~| ^ ~^ ^~	jednoargumentowe operacje redukcji logicznej	całkowity

Symbol operatora	Nazwa	Typy danych operandów
+ - * / **	binarne operacje arytmetyczne	całkowity, **real**, **shortreal**
%	binarna operacja arytmetyczna modulus	całkowity
& | ^ ^~ ~^	binarne operacje bitowe	całkowity
>> <<	binarne operacje przesunięcia logicznego	całkowity
>>> <<<	binarne operacje przesunięcia arytmetycznego	całkowity
&& || –> <–>	binarne operacje logiczne	całkowity, **real**, **shortreal**
< <= > >=	binarne operacje relacji	całkowity, **real**, **shortreal**
=== !==	binarne operacje tożsamości	każde, oprócz **real** i **shortreal**
== !=	binarne operacje równości	każdy
==? !=?	binarne operacje równości z symbolami wieloznacznymi	całkowity
++ --	jednoargumentowe operacje inkrementacji i dekrementacji	całkowity, **real**, **shortreal**
inside	binarna operacja przynależności do zbioru	singularny dla lewego operandu
dist	binarna operacja podziału binarnego	całkowity
{} {{}}	operacje konkatenacji i replikacji	całkowity
{<<{}} {>>{}}	operacje przesyłania strumieniowego	całkowity

ŹRÓDŁO: oprac. własne.

Typ całkowity to typ wartości, którą można przedstawić w postaci wektora bitów z dwoma lub czterema stanami dla każdego bitu. Większość operatorów dzieli się na jednoargumentowe (z jednym operandem) i dwuargumentowe (z dwoma operandami). Wyjątkiem są operatory warunkowe, konkatenacji, replikacji i operacje na wątkach.

Operator **dist** oraz operacje z operandami typu **real** i **shortreal** są używane tylko do celów symulacji.

9.4.1. Priorytet operacji

Wyrażenia języka SystemVerilog są obliczane w kolejności *priorytetów operacji* (*operation priority*). Nawiasy służą do zmiany kolejności operacji. Priorytet operacji SystemVerilog pokazano w tabeli 9.2, gdzie priorytet grup operacji maleje wraz ze wzrostem numeru grupy. Gdy priorytety są równe, większość operacji jest wykonywana

od lewej do prawej (asocjatywność operacji). Wyjątkami są operacje warunkowe (?:), implikacja logiczna (->) i równoważność logiczna (<->).

TABELA 9.2. Priorytet operacji języka SystemVerilog

Priorytet	Operacje	Asocjatywność
1	() [] :: .	od lewej do prawej
2	+ - ! ~ & ~& | ~| ^ ~^ ^~ ++ -- (jednoargumentowe)	niedostępne
3	**	od lewej do prawej
4	* / %	od lewej do prawej
5	+ - (dwuargumentowe)	od lewej do prawej
6	<< >> <<< >>>	od lewej do prawej
7	< <= > >= **inside dist**	od lewej do prawej
8	== != === !== ==? !=?	od lewej do prawej
9	& (dwuargumentowe)	od lewej do prawej
10	^ ~^ ^~ (dwuargumentowe)	od lewej do prawej
11	| (dwuargumentowe)	od lewej do prawej
12	&&	od lewej do prawej
13	||	od lewej do prawej
14	? : (operacja warunkowa)	od prawej do lewej
15	–> <–>	od prawej do lewej
16	= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>= := :/ <=	niedostępne
17	{} {{}}	od lewej do prawej

ŹRÓDŁO: oprac. własne.

9.4.2. Używanie literałów całkowitych w wyrażeniach

Literały całkowite mogą być używane jako operandy w wyrażeniach. Możliwe są następujące sposoby zapisu literału liczby całkowitej:

- jako liczba całkowita bez określania podstawy (bazy) i rozmiaru, np. 12;
- jako liczba całkowita z podstawą, ale bez rozmiaru, np. ‘d12 lub ‘sd12;
- w postaci liczby całkowitej z bazą i rozmiarem, np. 16’d12 lub 16’sd12.

W SystemVerilog ujemna liczba całkowita bez określenia podstawy jest inaczej interpretowana niż liczba całkowita z określeniem bazy. Ujemna liczba całkowita bez określenia bazy (np. -12) jest traktowana jako wartość ze znakiem w kodzie uzupełnienia do dwóch. Ujemna liczba całkowita z określeniem bazy bez znaku (np. -'d12) jest traktowana jako wartość bez znaku. Na reprezentację ujemnej liczby całkowitej ma również wpływ określenie jej wielkości w bitach. Na przykład:

```
int a;                  // a jest liczbą całkowitą o długości 32 bitów
a = -12/3;              // wynik -4; -12 jest traktowany jako uzupełnienie do dwóch
a = -‘d12/3;            /* wynik 1431655761; -d12 jest traktowany jako
                           wartość bez znaku 11...1100 */
a = -‘sd12/3;           /* wynik -4; -'sd12 jest wartością ze znakiem,
                           dlatego jest rozpatrywany w kodzie uzupełnienia do dwóch */
a = -4‘sd12/3;          /* wynik 1; -4'sd12 jest traktowany jako liczba binarna 1100
                           ze znakiem, tzn. -4, ale jest poprzedzony znakiem minus,
                           więc -(-4) = 4, a 4/3 = 1 */
```

9.4.3. Operacje z dwoma i czterema stanami

Operatory można stosować do wartości o dwóch i czterech stanach lub do mieszanych wartości o dwóch i czterech stanach. Jeśli operacje zostaną zastosowane do operandów o dwóch i czterech stanach, wynik będzie taki sam, jak gdyby wszystkie operandy były wartościami o czterech stanach. Jeśli wszystkie operandy są wartościami dwustanowymi, to wynik jest również wartością dwustanową. Wyjątkiem są operacje, które dają wynik x dla operandów o dwóch stanach, takie jak dzielenie przez zero. Na przykład:

```
int n = 8, zero = 0;
int res = 'b01xz | n;           // res jest równa 'b11xz, konwertowana do int, czyli 'b1100
int sum = n + n;                // sum jest równa 16, konwertowana do int, czyli 16
int sumx = 'x + n;              // sumx jest równa 'x', konwertowana do int, czyli 0
int div2 = n/zero + n;          // div2 jest równa 'x, konwertowana do int, czyli 0
integer div4 = n/zero + n;      // div4 równa się 'x
```

9.4.4. Użycie operatora przypisania w wyrażeniach

Operator przypisania (=) w tabeli 9.1 jest równoważny operatorowi proceduralnemu przypisania blokującego. Wyrażenie może zawierać blokującą operację przypisania, pod warunkiem że nie ma ono kontroli synchronizacji. Takie przypisania blokujące muszą być ujęte w nawiasy, aby uniknąć typowych błędów, takich jak użycie a = b zamiast a == b lub a |= b zamiast a != b. Na przykład:

```
if ((a=b)) b = (a+=1);  /* zmienna a jest przypisywana do zmiennej b,
                           jeśli wynik jest prawdziwy, to wykonana jest
                           operacja przypisania blokującego a = a + 1; */

a = (b = (c = 5));      /* zmiennej c przypisywana jest wartość 5,
                           wówczas zmiennej b zostanie przypisana wartość zmiennej c,
                           na koniec zmiennej a przypisywana jest wartość zmiennej b */
```

9.5. Opis operatorów

9.5.1. Operatory przypisania

Oprócz prostej *operacji przypisania* (*assignment operation*) (=) język SystemVerilog obejmuje operacje przypisania w języku C (+=, -=, *=, /=, %=), jak również specjalne operacje przypisania bitowego (&=, |=, ^=, <<=, >>=, <<<=, >>>=). Operacja przypisania jest semantycznie równoważna operatorowi proceduralnemu przypisania blokującego, z tą różnicą, że dowolne wyrażenie z indeksem po lewej stronie przypisania jest obliczane tylko raz. Na przykład:

```
a[i] += 2;          // tak samo jak a[i] = a[i] + 2;

a |= b;             // to samo, co a = a | b;

a <<= 2;            // tak samo jak a = a <<= 2;
```

9.5.2. Operatory inkrementacji i dekrementacji

Język SystemVerilog wspiera następujące *operatory przypisania inkrementacji i dekrementacji* (*increment and decrement assignment operations*) z języka C: ++i, --i, i++, a także i--. Operacje te nie wymagają nawiasów, gdy są stosowane w wyrażeniach. Zachowują się one podobnie jak operacje przypisania blokującego.

Kolejność operacji przypisania inkrementacji i dekrementacji w stosunku do innych operacji nie jest określona w wyrażeniu. Z tego powodu wynik takiego wyrażenia może być różny dla różnych kompilatorów. Na przykład:

```
i = 10;
j = i++ + (i = i - 1);
```

Po obliczeniu wyrażenia wartość zmiennej j może wynosić 18, 19 lub 20, w zależności od względnej kolejności operacji inkrementacji i przypisania.

Operacje inkrementacji i dekrementacji stosowane na operandach rzeczywistych powodują inkrementację lub dekrementację operandu o wartość 1.0.

9.5.3. Operatory arytmetyczne

Dwuargumentowe *operacje arytmetyczne* (*arithmetic operations*) przedstawiono w tabeli 9.3.

TABELA 9.3. Dwuargumentowe operacje arytmetyczne

Operacja	Opis
a + b	a plus b
a - b	a minus b
a * b	a pomnożone przez b
a / b	a podzielona przez b
a % b	a modulo b (reszta z dzielenia a przez b)
a ** b	a do potęgi b

ŹRÓDŁO: oprac. własne.

Przy dzieleniu całkowitym (/) odrzucana jest część ułamkowa. W przypadku dzielenia i operacji modulo (%), jeśli drugi operand jest zerem, wynikiem jest x. Wynik operacji modulo przyjmuje znak pierwszego operandu.

Jeśli jeden z operandów operacji arytmetycznej jest liczbą rzeczywistą, to wynik jest również liczbą rzeczywistą.

Wynik operacji potęgowania (**) jest niezdefiniowany (równy x), jeśli pierwszy operand jest zerem, a drugi operand jest liczbą ujemną, lub jeśli pierwszy operand jest liczbą ujemną, a drugi operand nie jest liczbą całkowitą. Wynikiem operacji potęgowania jest 1, jeśli drugi operand jest zerem.

Należy zauważyć, że często operatory potęgowania, dzielenia czy operacji modulo nie są wspierane przy wykorzystywaniu ich do syntezy. Aby dowiedzieć się, które operacje arytmetyczne są implementowane przez kompilator, należy zapoznać się z dokumentacją techniczną używanego kompilatora.

Przykłady operacji potęgowania i modulo są podane w tabeli 9.4.

TABELA 9.4. Przykłady operacji modulo i potęgowania

Wyrażenie	Wynik	Opis
10 % 3	1	10/3 daje resztę równą 1
11 % 3	2	11/3 daje resztę równą 2
12 % 3	0	12/3 nie daje żadnej reszty
–10 % 3	–1	Wynik przyjmuje znak pierwszego operandu
11 % –3	2	Wynik przyjmuje znak pierwszego operandu
–4'd12 % 3	1	-4'd12 jest postrzegana jako duża liczba dodatnia, która przy dzieleniu przez 3 daje resztę 1
3 ** 2	9	3 · 3
2 ** 3	8	2 · 2 · 2
2 ** 0	1	Każda wartość do potęgi zerowej jest równa 1
0 ** 0	1	Zero do potęgi zero również wynosi tu 1
2.0 ** –3'sb1	0.5	2.0 jest rzeczywiste, co daje rzeczywistą wartość odwrotności

Wyrażenie	Wynik	Opis
2 ** –3'sb1	0	2 ** -1 = 1/2. Wynik dzielenia liczby całkowitej jest obcinany do zera
0 ** –1	'x	0 ** -1 = 1/0. Dzielenie liczby całkowitej przez zero daje wynik 'x
9 ** 0.5	3.0	Rzeczywisty pierwiastek kwadratowy
9.0 ** (1/2)	1.0	Dzielenie całkowite powoduje obcięcie wykładnika do zera
–3.0 ** 2.0	9.0	Prawidłowo, ponieważ wartość rzeczywista 2.0 jest nadal wartością całkowitą

ŹRÓDŁO: oprac. własne.

Arytmetyczne operacje jednoargumentowe w SystemVerilog są reprezentowane przez jednoargumentowy plus +m i jednoargumentowy minus -m. Należy zauważyć, że +m oznacza to samo co m.

Jeśli w operacji arytmetycznej wartość jakiegoś bitu któregoś z operandów wynosi x lub z, to wynikiem operacji jest x.

9.5.4. Wyrażenia arytmetyczne ze znakiem i bez znaku

Sieci i zmienne mogą być jawnie deklarowane jako ze znakiem lub bez znaku. Typy danych **byte**, **shortint**, **int**, **integer** i **longint** są domyślnie ze znakiem. Inne typy danych są domyślnie bez znaku.

W języku SystemVerilog wartość przypisana do zmiennej lub sieci bez znaku jest traktowana jako wartość *bez znaku* (*unsigned*), a ta przypisana do zmiennej bądź sieci ze znakiem jest traktowana jako wartość ze znakiem. Wartości całkowite ze znakiem są przedstawiane jako uzupełnienie do dwóch. Wartości przypisane do zmiennych rzeczywistych wykorzystują reprezentację zmiennoprzecinkową. Transformacje pomiędzy wartościami ze znakiem i bez znaku zachowują tę samą reprezentację bitową, zmienia się tylko interpretacja.

W tabeli 9.5 przedstawiono sposób interpretacji wartości ze znakiem i bez znaku typów danych w operacjach arytmetycznych.

TABELA 9.5. Interpretacja typów danych w operacjach arytmetycznych

Typ danych	Interpretacja wartości
Sieć bez znaku	bez znaku
Sieć ze znakiem	ze znakiem, uzupełnienie do dwóch
Zmienna bez znaku	bez znaku
Zmienna ze znakiem	ze znakiem, uzupełnienie do dwóch
Zmienna rzeczywista	ze znakiem, zmiennoprzecinkowa

ŹRÓDŁO: oprac. własne.

W poniższym przykładzie pokazano różne sposoby dzielenia -12/3 przy użyciu zmiennych **integer** i **logic**.

```
integer intS;
var logic [15:0] U;
var logic signed [15:0] S;

intS = -4'd12;
U = intS / 3;          // wynik wyrażenia -4,
                       // intS jest typem danych całkowitych, U równa się 65 532

U = -4'd12;            // U równa się 65 524

intS = U / 3;          // U jest logicznym typem danych logic

intS = -4'd12 / 3;     // wynikiem wyrażenia jest 1 431 655 761
                       // -4'd12 jest w rzeczywistości 32-bitową liczbą
                       // z typem danych logic

U = -12 / 3;           // wynikiem wyrażenia jest -4, -12 jest w rzeczywistości
                       // typem danych integer, U równa się 65 532

S = -12 / 3;           // wynikiem tego wyrażenia jest -4. S jest typem logic ze znakiem

S = -4'sd12 / 3;       // wynikiem tego wyrażenia jest 1. -4'sd12 to w rzeczywistości 4,
                       // a wynik dzielenia całkowitego 4/3 = 1
```

9.5.5. Operatory relacji

Operatory relacyjne przedstawiono w tabeli 9.6.

TABELA 9.6. Operacje relacyjne

Operacja	Opis
a < b	a jest mniejsze od b
a > b	a jest większe niż b
a <= b	a jest mniejsze lub równe b
a >= b	a jest większe lub równe b

ŹRÓDŁO: oprac. własne.

Wynikiem operacji relacji jest jednobitowa (skalarna) wartość 1'b0, jeśli określona relacja jest fałszywa, lub jednobitowa wartość 1'b1, jeśli określona relacja jest prawdziwa. Jeśli dowolny bit dowolnego operandu w operacji relacyjnej ma wartość x lub z, to wynikiem jest jednobitowa wartość 1'bx.

Jeśli co najmniej jeden z operatorów operacji relacyjnej jest bez znaku, wyrażenie jest interpretowane jako porównanie wartości bez znaku. Mniejszy operand jest uzupełniany zerami do rozmiaru większego operandu.

Gdy oba operandy są ze znakiem, wyrażenie jest interpretowane jako porównanie wartości ze znakiem. Mniejszy operand jest rozwijany do rozmiaru większego operandu.

Jeśli operand jest liczbą rzeczywistą, drugi operand jest konwertowany na równoważną mu wartość rzeczywistą, a wyrażenie jest interpretowane jako porównanie liczb rzeczywistych.

9.5.6. Operatory równości

Operacje równości (*equality operations*) zostały przedstawione w tabeli 9.7.

TABELA 9.7. Operacje równości

Operacja	Opis
a === b	a równa się b, w tym x i z
a !== b	a nie jest równe b, w tym x i z
a == b	a jest równe b, wynik może nie być znany (x)
a != b	a nie jest równe b, wynik może nie być znany (x)

ŹRÓDŁO: oprac. własne.

Wynikiem operacji równości jest wartość skalarna 1, jeśli warunek jest spełniony, w przeciwnym wypadku zaś wartość skalarna 0. Operandy ze znakiem i bez znaku oraz rzeczywiste są interpretowane w taki sam sposób, jak w operacjach relacji.

Jeśli w logicznych operacjach równości (== i !=) jeden z operandów w którymkolwiek z bitów ma wartość x lub z, to wynikiem operacji jest wartość x (wartość nieznana).

Operacje == i !==, zwane również operacjami tożsamości, umożliwiają porównywanie wartości z czterema stanami, w tym wartości bitów x i z.

Wynikiem tych operacji jest zawsze wartość określona, tzn. 0 lub 1.

9.5.7. Operacje równości z symbolami wieloznacznymi

W porównaniu z językiem Verilog w SystemVerilog dodano dwie *operacje równości z symbolami wieloznacznymi* (*equal wildcard operations*): ==? i !=?. W tych operacjach wartości bitów x i z w prawym operandzie zachowują się jak symbole wieloznaczne. Przy takim porównaniu bit wieloznaczny w prawym operandzie odpowiada dowolnej wartości (0, 1, z lub x) w odpowiadającym mu bicie w lewym operandzie. Na przykład:

```
logic [3:0] a, b;

a = 4'b0011;
b = 4'b0xz1;

if (a == b) ...          // wynik porównania a i b jest prawdziwy (1'b1)
```

9.5.8. Operacje logiczne

Operacje logiczne (*logical operations*) języka SystemVerilog są przedstawione w tabeli 9.8.

TABELA 9.8. Operacje logiczne

Operacja	Zastosowanie	Opis
!	! a	negacja logiczna
&&	a && b	logiczne I (ORAZ) (AND)
||	a || b	logiczne LUB (OR)
->	a -> b	implikacja logiczna, taka sama jak (! a || b)
<->	a <-> b	równoważność logiczna, taka sama jak ((a -> b) && (b -> a)) lub ((! a && ! b) || (a && b))

ŹRÓDŁO: oprac. własne.

W porównaniu z językiem Verilog w SystemVerilog dodano dwie operacje: implikację logiczną (->) i równoważność logiczną (<->).

Operacje logiczne w wyrażeniach służą do logicznego łączenia innych wyrażeń. Aby zwiększyć czytelność kodu w wyrażeniach logicznych, zaleca się stosowanie nawiasów, mimo iż priorytet operacji logicznych jest niższy niż innych operacji. Na przykład następujące wyrażenia logiczne są równoważne:

```
a < size-1 && b != c && index != lastone
```

i

```
(a < size-1) && (b != c) && (index != lastone)
```

ale drugie wyrażenie jest bardziej zrozumiałe.

Aby zwiększyć czytelność kodu, zamiast **if** (!a) zaleca się stosowanie konstrukcji **if** (a == 0).

9.5.9. Operatory bitowe

Operatory bitowe (*bitwise operators*) w SystemVerilog są wymienione w tabeli 9.9. Operacje te odgrywają główną rolę w opisie projektów z funkcjami i wyrażeniami boolowskimi. Operacje logiczne algebry boolowskiej na wektorach i polach bitowych są wykonywane bitowo.

TABELA 9.9. Operacje bitowe

Symbol	Przykład	Opis
~	~m	inwersja każdego bitu wektora m, operacja jednoargumentowa
&	m & n	operacja logiczna AND każdego bitu wektora m z każdym bitem wektora n
|	m | n	operacja logiczna OR każdego bitu wektora m z każdym bitem wektora n
^	m ^ n	operacja logiczna XOR każdego bitu wektora m z każdym bitem wektora n
~^ или ^~	m ~^ n	operacja logiczna XNOR każdego bitu wektora m z każdym bitem wektora n
<<	m << n	przesunięcie w lewo o n pozycji zawartości wektora m, wartość po lewej stronie jest wypełniana zerami
>>	m >> n	przesunięcie w prawo o n pozycji zawartość wektora m, wartość po prawej stronie jest wypełniana zerami

ŹRÓDŁO: oprac. własne.

Wyniki operacji bitowych przedstawiono w tabeli 9.10. Przy obliczaniu wartości bitowych przyjmuje się, że wartość z jest równa x.

TABELA 9.10. Wyniki operacji bitowych

Operandy	Wyniki wyrażeń					
a	b	~a	a & b	a | b	a ^ b	a ~^ 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

Operandy	Wyniki wyrażeń					
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

ŹRÓDŁO: oprac. własne.

9.5.10. Operatory redukcji

Operatory redukcji (*reduction operators*), zwane też operacjami redukcji, przypominają operacje bitowe, ale ich działanie jest radykalnie inne. Operacje bitowe są stosowane po kolei do każdego bitu dwóch wektorów, a wynikiem jest wektor. Operacja redukcji jest najpierw stosowana do dwóch pierwszych bitów wektora. Jej kolejnymi operandami są wynik poprzedniego zastosowania tej operacji oraz następny bit wektora. Proces ten jest kolejno wykorzystywany do wszystkich bitów wektora. Wynikiem operacji redukcji jest jeden bit. Innymi słowy, operacje redukcji sprowadzają (redukują) wektor bitów do jednego bitu poprzez sekwencyjne zastosowanie odpowiedniej operacji logicznej do każdego bitu wektora.

Operacje redukcji w języku SystemVerilog są przedstawione w tabeli 9.11, a przykłady ich zastosowania w tabeli 9.12.

TABELA 9.11. Operacje redukcji

Symbol	Przykład	Opis
&	& m	redukuje bity wektora m przez AND
~&	~& m	redukuje bity wektora m przez NAND
|	| m	redukuje bity wektora m przez OR
~|	~| m	redukuje bity wektora m przez NOR
^	^ m	redukuje bity wektora m przez XOR
~^ lub ^~	~^ m	redukuje bity wektora m przez XNOR

ŹRÓDŁO: oprac. własne.

TABELA 9.12. Przykłady zastosowania operacji redukcji

Operand	Wyniki działań operacji redukcyjnych		
a	&a	|a	^a
00000000	0	0	0
11111111	1	1	0

Operand	Wyniki działań operacji redukcyjnych		
10101010	0	1	1
110011zz	0	1	x
1111111x	x	1	x

ŹRÓDŁO: oprac. własne.

Poniżej przedstawiono przykład zastosowania operacji redukcji do określenia, czy liczba jedynek wektora bitów jest parzysta i czy wszystkie bity mają wartość 1.

Przykład 9.1. Określanie w wektorze bitowym parzystej liczby jedynek i czy wszystkie bity mają wartość 1

```
module input_test
        (input [7:0] in,        // wejściowy wektor bitów
        output parity,          // prawda, jeśli liczba jednostek jest parzysta
all_ones);                      // prawda, jeśli wszystkie bity mają wartość 1

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

endmodule
```

Oto kolejny przykład zastosowania operacji bitowych i redukcji do realizacji funkcji równości w komparatorze.

Przykład 9.2. Implementacja funkcji równości komparatora

```
module comp_eq (
input [7:0] a, b,       // wektory wejściowe
output eq);             // funkcja równości

wire [7:0] im;          // zmienna pomocnicza

assign im = a ^ b;      // znajdowanie bitów, w których a i b nie są równe
assign eq = ~| im;      // jeśli w wektorze im jest co najmniej jedna jedynka,
                        // to a i b nie są równe
endmodule
```

9.5.11. Operacje przesunięcia

W SystemVerilog występują dwa typy operacji przesunięcia (*shift operations*): *operacje przesunięcia logicznego* (<< i >>) oraz *operacje przesunięcia arytmetycznego* (<<< i >>>). Operacje przesunięcia przesuwają w lewo (<< i <<<) lub w prawo (>> i >>>) wartość

pierwszego (lewego) operandu o liczbę bitów określoną przez drugi (prawy) operand. Puste pozycje bitów są wypełniane zerami, z wyjątkiem arytmetycznego przesunięcia w prawo wartości ze znakiem. W tym drugim przypadku bity są wypełniane wartością znaku. Znak wyniku jest określany przez lewy operand. Jeśli prawy operand ma wartość x lub z, to wynik jest nieznany (x). Prawy operand jest zawsze traktowany jako liczba bez znaku i nie ma wpływu na znak wyniku.

Przykładem zastosowania operacji przesunięcia jest algorytm mnożenia wartości przez liczbę 10.

Przykład 9.3. Mnożenie wartości przez 10

```
// Zastosowany wzór: A * 10 = A * (2 + 8) = A * 2 + A * 8
module mult_on_10 (input [7:0] A,          // wartość początkowa
                   output [11:0] Y);       // w wyniku otrzymujemy 4 bity więcej

      // niejawne użycie operatora przypisania:
      wire [8:0]    Ax2 = A << 1;  // A * 2
      wire [10:0]   Ax8 = A << 3;  // A * 8

      assign        Y = Ax2 + Ax8;       // jawne użycie operatora przypisania:
                                         // Y = (A<<1) + (A<<3)
endmodule
```

9.5.12. Operacja warunkowa

Operacja warunkowa (*conditional operation*) składa się z trzech operandów i ma następującą składnię:

cond_expression ? *expression1* : *expression2*,

gdzie *cond_expression* jest wyrażeniem warunkowym; *expression1* i *expression2* są wyrażeniami zwracanymi.

Operacja warunkowa oblicza wartość wyrażenia *cond_expression*. Jeśli wyrażenie warunkowe jest prawdziwe (wartość 1'b1), to operacja warunkowa zwraca wartość wyrażenia *expression1*; jeśli wyrażenie warunkowe jest fałszywe (wartość 1'b0), to operacja warunkowa zwraca wartość wyrażenia *expression2*. Operacja warunkowa może być stosowana wymiennie z operatorem proceduralnym **if-else**.

Jeśli wartość wyrażenia warunkowego jest nieznana (wartość x lub z), to obliczane są wartości wyrażeń *expression1* i *expression2*, które są porównywane pod względem równoważności logicznej (operacja ==). Jeśli to porównanie jest prawdziwe, operacja warunkowa zwraca wartość wyrażenia *expression1* lub wyrażenia *expression2*.

W przeciwnym razie operacja zwraca wynik oparty na typach danych wyrażeń *expression1* i *expression2*. W przypadku typów całkowitych wyniki wyrażeń *expression1* i *expression2* są wyrównywane według liczby bitów, a następnie wyniki są łączone bit po bicie zgodnie z tabelą 9.13.

TABELA 9.13. Wyniki łączenia bitów w operacji warunkowej

?:	**0**	**1**	**x**	**z**
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

ŹRÓDŁO: oprac. własne.

Należy zauważyć, że w języku SystemVerilog operacje warunkowe mogą być stosowane z wyrażeniami *expression1* i *expression2* typu rzeczywistego i agregującego.

Jako przykład można podać wykorzystanie operacji warunkowych do implementacji prostej jednostki arytmetyczno-logicznej (ALU).

Przykład 9.4. Wykorzystanie operacji warunkowych do opisania prostej jednostki ALU

```
module mini_ALU    (input [7:0] a, b,          // operandy
                    input [2:0] op,            // kod operacji
                    output [7:0] result);      // wynik

parameter     ADD=3'h0, SUB=3'h1, AND=3'h2,
              OR=3'h3, XOR=3'h4;

assign result = ((op == ADD) ? a+b : (
                 (op == SUB) ? a-b : (
                 (op == AND) ? a&b : (
                 (op == OR) ? a | b : (
                 (op == XOR) ? a^b : (a))))));
endmodule
```

9.5.13. Operator konkatenacji

Operator konkatenacji (*concatenation operator*) jest opisywany za pomocą nawiasów klamrowych { i } z przecinkami w środku w celu oddzielenia wyrażeń (operandów). Wynikiem operacji konkatenacji jest konkatenacja bitów wielu wyrażeń. Rozmiar każdego operandu musi być określony liczbą. Przykładem może być następująca operacja konkatenacji:

```
{a, b[3:0], w, 3'b101}
```

jest równoważne następującemu przykładowi:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}.
```

Konkatenacja jest interpretowana jako spakowany wektor bitów. Operacji konkatenacji można użyć po lewej stronie znaku równości. Na przykład:

```
wire a, b, cin, s, caut;
{caut, s} = a +b +cin;
```

Z konkatenacji można wybrać jeden lub więcej bitów, tak jak z regularnej tablicy spakowanej o zakresie [N-1:0], gdzie N jest rozmiarem wyniku operacji konkatenacji. Na przykład:

```
byte a, b ;
bit [1:0] c ;
c = {a + b}[1:0];                // dwa najmniej znaczące bity sumy a i b
```

Kilka przykładów użycia operacji konkatenacji:

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

9.5.14. Operator replikacji

Operator replikacji (*replication operator*), zwany także konkatenacją wielokrotną (*multiple concatenation*), jest opisany jako wyrażenie konkatenacji poprzedzone wyrażeniem stałym, zwanym *stałą replikacji* (*replication constant*), ujętym w nawiasy klamrowe.

Składnia operacji replikacji:

```
{N{a}},
```

gdzie N jest stałą replikacji, N ≥ 0; a jest wyrażeniem konkatenacyjnym.

Wynikiem operacji replikacji jest konkatenacja N wyrażeń a. Na przykład:

```
{4{w}}          // daje taką samą wartość jak {w, w, w, w}.
```

Replikację można zastosować w konkatenacji. Na przykład:

```
{b, {3{a, b}}}  // daje taką samą wartość jak {b, a, b, a, b, a, b, a, b}.
```

Stała replikacji może mieć wartość 0. Jest to przydatne w kodzie sparametryzowanym. Replikacja z zerową stałą replikacji jest uważana za replikację o zerowym rozmiarze i jest ignorowana. Powinna ona występować tylko w konkatenacji, w której co najmniej jeden z operandów konkatenacji ma rozmiar dodatni. Na przykład:

```
parameter P = 32;

assign b[31:0] = { {32-P{1'b1}}, a[P-1:0] };     // dopuszczalne dla wszystkich P
                                                 // od 1 do 32

assign c[31:0] = { {{32-P{1'b1}}}, a[P-1:0] };   // niedopuszczalne dla P = 32, ponieważ
                                                 //w konkatenacji pojawia się zerowa replikacja
```

9.5.15. Konkatenacja ciągów znaków (łańcuchów)

Język SystemVerilog umożliwia konkatenację obiektów danych typu **string**. Ogólnie rzecz biorąc, jeśli którykolwiek z operandów jest typu **string**, konkatenacja jest traktowana jako łańcuch, a wszystkie pozostałe argumenty są niejawnie konwertowane na typ danych **string**. Konkatenacja łańcuchów nie jest dozwolona po lewej stronie przypisania.

Przykład konkatenacji ciągów znaków:

```
string hello = “hello”;
string s;

s = { hello, “ “, “world” };
$display( “%s\n”, s );                    // wyświetla “hello world”

s = { s, “ and goodbye” };
$display( “%s\n”, s );                    // wyświetla “hello world and goodbye”
```

Operacji replikacji można również używać z obiektami danych typu **string**. W przypadku replikacji ciągów dozwolone jest stosowanie zmiennego mnożnika. Na przykład:

```
int n = 3;
string s = {n { “boo “ }};
$display( “%s\n”, s );                    // wyświetla “boo boo boo “.
```

W przeciwieństwie do konkatenacji bitowej wynik konkatenacji lub replikacji łańcuchów nie jest obcinany. Zamiast tego rozmiar zmiennej docelowej (łańcucha typu **string**) jest zwiększany, aby pomieścić łańcuch wynikowy.

9.5.16. Operacja ustalania przynależności do tablicy

Język SystemVerilog udostępnia oryginalną operację *ustalania przynależności* (*set membership*) jakiegoś elementu, która odpowiada na pytanie, czy element należy do danej tablicy. Składnia operacji przynależności do zbioru jest następująca

expression **inside** {*range_list*},

gdzie wyrażenie *expression* jest dowolnym wyrażeniem pojedynczym (*singular*), tzn. nie jest tablicą; *range_list* jest oddzieloną przecinkami listą wyrażeń lub zakresów.

Operacja **inside** sprawdza, czy wyrażenie *expression* należy do *range_list*, która jest skanowana do momentu znalezienia pasującego elementu. Jeśli zostanie znalezione dopasowanie, **inside** zwraca 1'b1, jeśli *range_list* zostanie przeskanowana do końca i nie zostanie znalezione dopasowanie, **inside** zwraca 1'b0. Na przykład:

```
int a, b, c;
if ( a inside {b, c} ) ...          // szukane jest dopasowanie a z b lub c
```

```
int array [$] = '{3,4,5};          // tablica dynamiczna z inicjalizacją
if ( ex inside {1, 2, array} ) ... // {1, 2, array } podobne do { 1, 2, 3, 4, 5}, szukane
                                   // jest dopasowanie ex do jednego z { 1, 2, 3, 4, 5}
```

W przypadku wyszukiwania dopasowań dla wyrażeń niecałkowitych operacja **inside** wykorzystuje operację równości (==). W przypadku wyrażeń całkowitych stosowana jest operacja równości z symbolami wieloznacznymi (==?), więc wartości x i z w zawartości elementu listy są traktowane jako wartości nieznane na danej pozycji bitowej. Jeśli zostanie użyty symbol wieloznaczny ?, wartości x i z w wyrażeniu *expression* nie są traktowane jako nieokreślone. Na przykład:

```
logic [2:0] val;
while ( val inside {3'b1?1} ) ...        // {3'b1?1} odpowiada 3'b101, 3'b111, 3'b1x1, 3'b1z1
```

Jeśli nie znaleziono dopasowania, ale niektóre porównania dają wynik x, operacja **inside** zwraca 1'bx. Wartość zwracana jest w rzeczywistości wynikiem operacji sumy logicznej OR wszystkich porównań elementów listy z wyrażeniem. Na przykład:

```
wire r;
assign r=3'bz11 inside {3'b1?1, 3'b011};     // r = 1'bx
```

Elementy listy *range_list* można określić za pomocą zakresu w postaci [low:high]. Dopasowanie jest dokonywane, jeśli wyrażenie *expression* znajduje się w zakresie. Na przykład:

```
bit ba = a inside { [16:23], [32:47] };  // to samo co ((a >= 16) && (a <= 23)) ||
                                         // ((a >= 32) && (a <= 47))
```

Należy zauważyć, że operacja **inside** nie jest syntetyzowalna w programie Quartus.

9.5.17. Operacje strumieniowe

Nowością w języku SystemVerilog, w porównaniu z Verilogiem, są również *operacje strumieniowe* (*streaming operations*). Pakują one typy zmiennych bitowych w sekwencję bitów w kolejności określonej przez użytkownika. Operacje strumieniowe stosowane po lewej stronie przypisania wykonują odwrotną czynność: rozpakowują strumień bitów do jednej lub więcej zmiennych.

Gdy określona kolejność strumienia bitów nie jest istotna dla konkatenacji strumienia bitów do określonego typu, można zastosować operację *konwersji strumieni bitów* omówioną w podrozdziale 5.2.3. Jeśli spakowane dane zawierają typy o czterech stanach, wynikiem operacji pakowania jest strumień o czterech stanach; w przeciwnym razie wynikiem operacji pakowania jest strumień o dwóch stanach.

Składnia operacji konkatenacji strumienia bitów wygląda następująco

{*stream_operator* [*slice_size*] *stream_concatenation***}**,

gdzie: *stream_operator* jest jedną z operacji << lub >>; *slice_size* – rozmiar fragmentu, typ prosty lub wyrażenie stałe; *stream_concatenation* – konkatenacja strumieni. W tym miejscu nawiasy klamrowe są obowiązkowe, a nawiasy kwadratowe oznaczają, że *slice_size* jest opcjonalny.

Operacja << określa, że bloki danych są przesyłane od prawej do lewej strony, a operacja >> – od lewej do prawej. Parameter *slice_size* ustala rozmiar każdego bloku danych w bitach. Jeśli *slice_size* nie określono, domyślnym rozmiarem bloku jest 1.

Konkatenacja strumieni *stream_concatenation* może być użyta zarówno w kontekście miejsca docelowego, jak i źródła i ma następujący format:

{*stream_expression* {, *stream_expression*}}

gdzie *stream_expression* jest wyrażeniem strumieniowym. Obowiązkowe są tu tylko zewnętrzne nawiasy klamrowe, a wewnętrzne nawiasy klamrowe służą do ewentualnego wielokrotnego powtarzania.

Konkatenacja strumienia *stream_concatenation* może być również użyta jako *stream_expression* w innej konkatenacji strumienia.

Operacje strumieniowe nie są syntetyzowalne. Są one zwykle stosowane przy symulacji złożonych projektów na poziomie transakcji.

9.6. Rozmiar wyrażeń bitowych

Rozmiar długości wyrażenia bitowego (*expression bit length*) zależy od operandów i kontekstu, w jakim wyrażenie jest używane. Może ono być *wyrażeniem określonym przez siebie* (*self-determined expression*) lub *wyrażeniem określonym przez kontekst* (*control-determined expression*). W wyrażeniu samodefiniowanym jego długość bitowa jest określana wyłącznie przez samo wyrażenie, np. reprezentujące wartość opóźnienia. W wyrażeniu zdefiniowanym w kontekście długość wyrażenia jest określana także przez fakt, że jest ono częścią innego wyrażenia. Na przykład rozmiar prawego wyrażenia w operacjach na miejscu docelowym zależy od samego wyrażenia i rozmiaru lewej strony.

W tabeli 9.14 przedstawiono rozmiary wyrażeń bitowych, gdzie i, j oraz k reprezentują wyrażenia operandów, a L(i) oznacza długość wyrażenia i.

TABELA 9.14. Wymiary wyrażeń bitowych

Wyrażenie	Długość w bitach	Opis
Niewymiarowa (unsized) liczba stała	taka sama jak liczba całkowita	
Wymiarowa liczba stała	jak podano	
i op j	max(L(i),L(j))	op – operacje binarne + - * / % & | ^ ^~ ~^.
op i	L(i)	op – operacje jednoargumentowe + - ~.
i op j	1 бит	op – operacje porównania == !== == != > >= < <=, operandy mają rozmiar max(L(i), L(j))
i op j	1 бит	op – operacje && || -> <->, wszystkie operandy są zdefiniowane samodzielnie
op i	1 бит	op – operacje & ~& | ~| ^ ~^ ^~ !, wszystkie operandy są zdefiniowane samodzielnie
i op j	L(i)	op – operacje >> << ** >>> << <<, j jest zdefiniowane samodzielnie
i ? j : k	max(L(j),L(k))	i jest zdefiniowane samodzielnie
{i,...,j}	L(i) + … + L(j)	wszystkie operandy są zdefiniowane samodzielnie
{i{j,..,k}}	i x (L(j) + … + L(k))	wszystkie operandy są zdefiniowane samodzielnie

ŹRÓDŁO: oprac. własne.

Przykład wyrażeń definiowanych samodzielnie i kontekstowo:

```
logic [3:0] a;
logic [5:0] b;
logic [15:0] c;

c = {a**b};                 // wyrażenie a ** b jest zdefiniowane samodzielnie
                            // poprzez operację konkatenacji
c = a**b;                   // rozmiar wyrażenia a**b jest zależny od rozmiaru c
```

9.7. Wyrażenia ze znakiem

Ogólnie rzecz biorąc, wyrażenie w SystemVerilog może być *ze znakiem* (*signed*) lub *bez znaku* (*unsigned*). Aby to określić, można użyć operacji konwersji oraz *funkcji systemowych* **$signed** i **$unsigned**. Na przykład:

```
logic [7:0] a, b;
logic signed [7:0] s;
```

```
a = $unsigned(-4);                  // a = 8'b11111100
b = $unsigned(-4'sd4);              // b = 8'b00001100
s = $signed (4'b1100);              // s = -4

a = unsigned'(-4);                  // a = 8'b11111100
s = signed'(4'b1100);               // s = -4

s = a + b;                          // dodawanie bez znaku
s = byte'(a) + byte'(b);            // dodawanie ze znakiem
s = signed'(a) + signed'(b);        // dodawanie ze znakiem
s = $signed(a) + $signed(b);        // dodawanie ze znakiem
```

Znak wyrażenia zależy od operandów i jest niezależny od lewej strony. Wszystkie liczby dziesiętne są liczbami ze znakiem. Liczby z określoną podstawą systemu są liczbami bez znaku, chyba że w określeniu podstawy użyto notacji s, np. 4›sd12.

Wyniki wyboru bitu lub części są zawsze bez znaku, nawet jeśli wybór części dotyczy całego wektora. Na przykład:

```
logic [15:0] a;
logic signed [7:0] b;

initial
a = b[7:0];              // b [7:0] jest bez znaku i dlatego jest rozszerzone o zera
```

Wyniki operacji konkatenacji i porównywania są zawsze bez znaku, niezależnie od operandów. Prawidłowe liczby przekształcone na liczby całkowite są ze znakiem. Znak i wielkość operandu samodefiniującego są określane przez sam operand i nie zależą od reszty wyrażenia.

W przypadku operandów samodefiniowanych obowiązują następujące zasady

- jeśli operand jest bez znaku, to wynik jest bez znaku, niezależnie od operacji;
- jeśli wszystkie operandy są ze znakiem, to wynik jest ze znakiem, niezależnie od operacji.

Podczas wykonywania operacji przypisania rozmiar prawej strony jest określany zgodnie z tabelą 9.14. W razie potrzeby rozmiar prawej strony jest zwiększany do wartości rozmiaru lewej strony. Jeśli wyrażenie po prawej stronie jest ze znakiem, uzupełnienie jest wykonywane bitem znaku. Jeśli wartość bitu znaku jest równa x lub z, to wynik jest uzupełniany wartością x lub z.

9.8. Wyrażenia literałów łańcuchowych

Przypomnijmy, że literały łańcuchowe mogą być przechowywane w wektorach bitowych i innych typach spakowanych. Język SystemVerilog ma również zmienne łańcuchowe, które mogą przechowywać literały łańcuchowe. Operandy literałów łańcuchowych są traktowane jako liczby składające się z sekwencji 8-bitowych kodów ASCII, po jednym kodzie na znak. Każda operacja SystemVerilog może obsługiwać operandy literałów łańcuchowych, tak jakby cały łańcuch był pojedynczą wartością liczbową.

Jeśli w momencie przypisywania wektor jest większy niż długość łańcucha, zawartość wektora po lewej stronie jest wypełniana zerami, tak jak w przypadku przypisywania wartości numerycznych bez znaku. Na przykład:

```
module string_test;
        bit [8*14:1] stringvar; // wektor bitów reprezentujący 14 znaków

        initial begin
                stringvar = „Hello world”;      // przypisanie 11-znakowego łańcucha znaków
                $display(“%s is stored as %h”, stringvar, stringvar);
                stringvar = {stringvar,”!!!”};          // dodanie trzech znaków
                $display(“%s is stored as %h”, stringvar, stringvar);
        end
endmodule
```

W wyniku symulacji powyższego kodu uzyskiwane są następujące łańcuchy:

```
„Hello world”,          zapamiętane jako    00000048656c6c6f20776f726c64
„Hello world!!!”,       zapamiętane jako    48656c6c6f20776f726c64212121
```

Język SystemVerilog obsługuje operacje kopiowania, konkatenacji i porównywania literałów łańcuchowych. Kopiowanie jest realizowane za pomocą operatora przypisania, konkatenacja – za pomocą operatora konkatenacji, a porównywanie – za pomocą operatora równości.

Aby uzyskać poprawne wyniki podczas manipulowania wektorami bitowymi i literałami łańcuchowymi, rozmiar wektora powinien wynosić 8*n, gdzie n jest liczbą znaków w łańcuchu. Gdy wektor jest większy niż literał łańcuchowy, najstarsze bity są wypełniane zerami, co może mieć wpływ na wyniki operacji porównywania i konkatenacji. Na przykład:

```
bit [8*10:1] s1, s2;
        initial begin
                s1 = „Hello”;                   // przypisanie łańcucha znaków
                s2 = „ world!”;                 // przypisanie łańcucha znaków
                if ({s1,s2} == „Hello world!”)  // porównanie konkatenacji z ciągiem znaków
```

```
                $display(„strings are equal”);
end
```

W tym przypadku operacja porównania daje fałszywy wynik, ponieważ zmienne s1 i s2 przyjmują w momencie przypisywania następujące wartości:

```
s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421
```

których konkatenacja nie jest taka sama jak ciąg znaków “Hello world!”.

Pusty łańcuch („”) jest równoważny znakowi ASCII NULL („\0”), który ma wartość zerową (0) inną niż łańcuch „0”.

9.9. Konstrukcja let

W języku Verilog do tworzenia kodu projektu można używać makr tekstowych, a także makr z argumentami. Język SystemVerilog wprowadza bardziej rozbudowany mechanizm zwany konstrukcją **let** (*let construct*), który w pewnych przypadkach upraszcza tworzenie kodu projektu.

Uogólniona składnia konstrukcji **let** to:

let *let_identifier* [(*let_port_list*)] = *expression*;

gdzie: *let_identifier* jest nazwą konstrukcji **let**; *let_port_list* jest listą portów konstrukcji **let**; *expression* jest wyrażeniem szablonu (*template expression*) lub ciałem konstrukcji **let** (let body).

Z kolei każdy element listy portów w konstrukcji **let** ma następujący format:

{ *attribute_instance* } *let_formal_type formal_port_identifier* { *variable_dimension* } [**=** *expression*]

gdzie: *attribute_instance* jest instancją atrybutu; *let_formal_type* jest typem elementu formalnego; *formal_port_identifier* jest nazwą portu formalnego; *variable_dimension* jest rozmiarem zmiennej; *expression* jest wyrażeniem określającym wartość domyślną elementu.

Przykład konstrukcji **let**:

```
let mult(x, y) = ($bits(x) + $bits(y))'(x * y);
```

Tutaj mult jest nazwą konstrukcji **let**; x i y są listami portów; (**$bits**(x) + **$bits**(y)) – określa rozmiar wyrażenia; '(x * y) jest wyrażeniem szablonu (wzorca).

Zauważmy też, że elementy *let_port_list* w wyrażeniu *expression* mogą być przekazywane zarówno przez pozycję, jak i przez nazwę, podobnie jak zmienne przekazywane do portów modułu.

Deklaracja konstruktu **let** definiuje wyrażenie szablonu skonfigurowane przez jego porty. Jest używana do dostosowywania (*customization*) kodu projektu i często zastępuje makra tekstowe. Konstrukcja **let** ma zasięg lokalny i dlatego jest bezpieczniejsza niż makra, które działają globalnie w obrębie całego kodu źródłowego projektu.

Deklaracje **let** mogą być dołączane do pakietów. Na przykład:

```
package def;
        let v_a(r, v, o) = |(r & v) || o;                    // deklaracja konstrukcji let
        ...
endpackage

module my_checker;
        import def::*;
        logic a, b;
        wire [1:0] req;
        wire [1:0] vld;
        logic ovr;
        ...
                if (v_a(.r(req), .v(vld), .o(ovr)))          // użycie konstrukcji let
                begin
                        ...
                end
        ...
endmodule
```

W tym przykładzie w pakiecie *def* zadeklarowano konstrukcję **let** o nazwie *v_a*, której portami są *r*, *v* i *o*. Ciało konstrukcji **let** jest reprezentowane przez wyrażenie |(r & v) || o. W module my_checker konstrukcja **let** jest używana do określenia wyniku wyrażenia instrukcji **if**, przy czym argumenty (*req*, *vld* i *ovr*) są przekazywane do portów konstrukcji **let** przez nazwę.

Konstrukcja **let** może być również używana do skracania nazw identyfikatorów i podwyrażeń. Na przykład:

```
task write_value;
        input logic [31:0] addr;
        input logic [31:0] value;
        ...
endtask
```

```
...
let addr = top.block1.unit1.base + top.block1.unit2.displ;  /* długie nazwy w wyrażeniu są
                                                              zastępowane nazwą skróconą addr */
...
write_value(addr, 0);                                        // krótka nazwa (addr) wyrażenia
```

Konstrukcje **let** mogą być deklarowane w modułach, interfejsach, pakietach, kontekstach jednostek kompilacji, jednostkach generujących, jednostkach procedur, zadaniach i funkcjach, a także w programach niesyntezowanych, kontrolerach i blokach synchronizacji.

Poniżej podano przykłady użycia konstrukcji **let**.

```
/* konstrukcje let z argumentami i bez argumentów */

module m;
        logic clk, a, b;
        logic p, q, r;

        // let z argumentami formalnymi x, y oraz dla y domyślną wartością b
        let eq(x, y = b) = x == y;

        // let bez argumentów
        let tmp = a && b;
        ...
        a1: assert property (@(posedge clk) eq(p,q));
        always_comb begin
                a2: assert (eq(r));      // r jest argumentem dla x,
                                         // a domyślną wartością dla y jest b
                a3: assert (tmp);        // użycie let bez argumentów
        end
endmodule : m
```

Efektywny kod po rozszerzeniu wyrażenia let:

```
module m;
        bit clk, a, b;
        logic p, q, r;
        // let eq(x, y = b) = x == y;
        // let tmp = a && b;
        ...
        a1: assert property (@(posedge clk) (m.p == m.q));
        always_comb begin
                a2: assert ((m.r == m.b));              // domyślną wartością dla y jest b
```

```
        a3: assert ((m.a && m.b));
    end
endmodule : m

/* Kontekstowe zakotwiczenie argumentów let */

module top;
    logic x = 1'b1;          // pierwsza zmienna x
    logic a, b;
    let y = x;
    ...
    always_comb begin
        // y jest powiązane z poprzednią definicją x w deklaratywnym
        // kontekście let
        bit x = 1'b0;    // druga zmienna x
        b = a | y;       // tutaj y odpowiada pierwszej zmiennej x
    end
endmodule : top
```

Efektywny kod po rozszerzeniu wyrażenia let:

```
module top;
    bit x = 1'b1;
    bit a;
    // let y = x;

    ...
    always_comb begin
        // y jest powiązane z poprzednią definicją x w deklaratywnym
        // kontekście let
        bit x = 1'b0;
        b = a | (top.x);
    end
endmodule : top
```

9.10. Wnioski

Wyrażenie (*expression*) w SystemVerilog to konstrukcja, która łączy *operandy* (*operands*) i *operatory* (*operators*) w celu uzyskania wyniku. Wyrażeniem jest również operand bez operatora, np. wybór bitu.

Operandami w SystemVerilog mogą być: literał numeryczny, literał łańcuchowy, parametr, sieć, zmienna, struktura, element struktury, unia, element unii, tablica oraz wywołanie funkcji lub metody.

Wyrażenie stałe (*constant expression*) to wyrażenie, które zwraca stałą. Jego operandami mogą być tylko stałe, wybór bitu i wybór części. W wyrażeniu stałym można również użyć funkcji systemowych, które zwracają stałe.

Wyrażenia zagregowane (*aggregate expressions*) są obiektami danych w postaci rozpakowanych struktur i tablic.

Operatory określają działania, które mają być wykonywane na operandach w wyrażeniu. Większość operatorów w SystemVerilog jest taka sama jak w języku C.

Nowe operacje, w porównaniu z językiem Verilog, to:

- operacje przypisania arytmetycznego (+= -= /= *= %=);
- operacje przypisania bitowego (&= |= ^=);
- operacje przypisania logicznego i przesunięcia arytmetycznego (>>= <<= >>>= <<=);
- operacje równości z symbolami wieloznacznymi (==? !=?);
- operacje inkrementacji i dekrementacji (++ --);
- implikacja logiczna (->);
- równoważność logiczna (<->);
- operacja ustalania przynależności do zbioru **inside**;
- operacja dystrybucji (**dist**);
- operacje strumieniowe ({<<{}} {>>{}}).

Operacja **dist** oraz operacje z operandami typu **real** i **shortreal** są używane tylko do celów symulacji.

Wyrażenia SystemVerilog są obliczane w kolejności priorytetów operatorów. Nawiasy okrągłe służą do zmiany kolejności wykonywania operacji.

Ujemna wartość liczby całkowitej bez określenia podstawy systemu w SystemVerilog jest interpretowana inaczej niż liczba całkowita z określeniem podstawy. Ujemna liczba całkowita bez określenia podstawy (np. -12) jest traktowana jako wartość ze znakiem w kodzie uzupełnienia do dwóch. Ujemna liczba całkowita ze specyfikatorem podstawy (np. -'d12) jest traktowana jako wartość bez znaku.

Jeśli operacje zostaną zastosowane do operandów o dwóch i czterech stanach, wynik będzie taki sam, jak gdyby wszystkie operandy były wartościami o czterech stanach. Jeśli wszystkie operandy są wartościami o dwóch stanach, to wynikiem jest wartość o dwóch stanach. Wyjątkiem są operacje, które dają wynik x dla operandów o dwóch stanach, np. dzielenie przez zero.

Operator przypisania (=) jest równoważny operatorowi proceduralnemu przypisania blokującego.

Przy syntezie na ogół nie są implementowane operacje potęgowania, dzielenia i operacje modulo.

Jeśli w operacji arytmetycznej wartość jakiegoś bitu któregoś z operandów wynosi x lub z, to wynikiem operacji jest x.

Wartości całkowite ze znakiem są przedstawiane jako uzupełnienie do dwóch. Transformacje między wartościami ze znakiem i bez znaku zachowują tę samą reprezentację bitową, zmienia się tylko interpretacja.

Wynikiem operacji relacji lub równości jest pojedyncza wartość bitowa 0 lub 1. Jeśli jakiś bit dowolnego operandu w operacji relacji ma wartość x albo z, to wynikiem jest jednobitowa wartość x.

W operacjach równości z symbolami wieloznacznymi (==? i !=?) wartości bitów x i z w prawym operandzie zachowują się jak symbole wieloznaczne. Przy porównywaniu bit wieloznaczny w prawym operandzie odpowiada dowolnej wartości (0, 1, z lub x) w odpowiadającym mu bicie w lewym operandzie.

W celu zwiększenia czytelności kodu zaleca się stosowanie nawiasów w wyrażeniach logicznych.

Operacje redukcji redukują wektor bitów do jednego bitu przez sekwencyjne zastosowanie odpowiedniej operacji logicznej do każdego bitu wektora.

Operacje przesunięcia przesuwają w lewo (<< i <<<) lub w prawo (>> i >>>) wartość pierwszego operandu o liczbę bitów określoną przez drugi operand. Puste pozycje bitów są wypełniane zerami, z wyjątkiem przesunięcia arytmetycznego w prawo, kiedy to bity są wypełniane wartością znaku.

Operacja warunkowa oblicza wartość wyrażenia *cond_expression*. Jeśli jego wartość jest prawdziwa, to operacja warunkowa zwraca wartość wyrażenia *expression1*, jeśli wartość wyrażenia *cond_expression* jest fałszywa, to operacja warunkowa zwraca wartość wyrażenia *expression2*. Operację warunkową można stosować wymiennie z operatorem proceduralnym **if-else**.

Wynikiem operacji konkatenacji jest połączenie bitów kilku wyrażeń. Konkatenacja jest interpretowana jako spakowany wektor bitów. Operatora konkatenacji można użyć po lewej stronie znaku równości. Przy operacji konkatenacji można zastosować selekcję bitów i selekcję części bitów.

Operator *replikacji* (*replication operator*) lub powtarzania umożliwia wykonanie wielu operacji konkatenacji na wyrażeniu bitowym. Wynikiem operacji replikacji jest konkatenacja N wyrażeń, gdzie N jest stałą replikacji.

Wynik konkatenacji lub replikacji łańcuchów nie jest obcinany. Zamiast tego rozmiar łańcucha docelowego jest zwiększany, aby pomieścić łańcuch wynikowy.

Operacja *określania przynależności do zbioru* **inside** odpowiada na pytanie, czy dany element należy do określonej tablicy.

Operacje strumieniowe polegają na spakowaniu zmiennych typu bitowego w sekwencję bitów w kolejności określonej przez użytkownika. Operacje strumieniowe stosowane po lewej stronie przypisania wykonują odwrotną czynność: rozpakowują strumień bitów do jednej lub więcej zmiennych.

Wyrażenie może być *wyrażeniem określonym przez siebie* (*self-determined expression*) lub *wyrażeniem określonym przez kontekst* (*control-determined expression*). W tym pierwszym długość bitów wyrażenia jest określana przez samo wyrażenie. W wyrażeniu kontekstowym długość wyrażenia zależy od jego długości oraz faktu, że jest ono

częścią innego wyrażenia, np. rozmiar prawego wyrażenia w operacji zależy od samego wyrażenia oraz rozmiaru lewej strony.

Wyrażenie w SystemVerilog może być *ze znakiem* (*signed*) lub *bez znaku* (*unsigned*). Aby to określić, można użyć operacji konwersji oraz funkcji systemowych **$signed** i **$unsigned**.

Znak wyrażenia jest zależny od operandów i nie jest zależny od lewej strony. Wszystkie liczby dziesiętne są liczbami ze znakiem. Liczby z określoną podstawą są liczbami bez znaku, chyba że w określeniu podstawy użyto notacji s, np. 4›sd12. Wyniki wyboru bitu lub części wektora są zawsze bez znaku, nawet jeśli wybór części dotyczy całego wektora. Wyniki operacji konkatenacji i porównywania są zawsze bez znaku, niezależnie od operandów. Liczby rzeczywiste przekształcone na liczby całkowite są ze znakiem. Istnieją również zasady służące do określania istnienia znaku operandów zdefiniowanych przez siebie oraz dla wyników operacji przypisania.

Operandy literałów łańcuchowych są traktowane jak liczby składające się z sekwencji 8-bitowych kodów ASCII, po jednym kodzie na znak. Każda operacja SystemVerilog może operować na operandach literałów łańcuchowych tak, jakby cały łańcuch był pojedynczą wartością liczbową.

Język SystemVerilog obsługuje operacje kopiowania, konkatenacji i porównywania literałów łańcuchowych.

Pusty łańcuch („”) jest równoważny znakowi ASCII NULL („\0”), który ma wartość zerową (0), w odróżnieniu od łańcucha „0”.

Konstrukcja **let** rozszerza możliwości makr tekstowych przy tworzeniu kodu projektu. Jej deklaracja definiuje wyrażenie szablonu (wzorca), które jest konfigurowane przez porty konstrukcji **let**. Konstrukcje **let** służą do samodzielnego konfigurowania kodu i w wielu przypadkach mogą zastąpić makra tekstowe. Konstrukcja **let** ma zasięg lokalny, więc jest bezpieczniejsza niż makra, które działają globalnie.

10. Operatory przypisania

Przydzielanie (przypisywanie) wartości do elementów sieci i zmiennych w SystemVerilog, podobnie jak polecenia przekazywania danych w językach programowania, jest najczęściej wykonywaną czynnością. Jednak w SystemVerilog przypisywanie wartości ma wiele cech, które znacznie różnią je od przypisywania wartości w językach programowania. Deweloper powinien bardzo dobrze rozumieć zawiłości przypisań języka SystemVerilog, dlatego temu tematowi poświęcony jest osobny rozdział.

Główny problem z przypisaniami wynika z natury podstawowych obiektów w SystemVerilog, czyli sieci i zmiennych. Sieci odpowiadają połączeniom przewodowym, a zmienne – rejestrom i przerzutnikom. Jednak zmienne SystemVerilog nie zawsze są implementowane jako rejestry lub przerzutniki.

Kolejnym problemem jest równoległość języka SystemVerilog. W językach programowania operacje są zwykle wykonywane sekwencyjnie, tak jak są zapisane w kodzie źródłowym. W SystemVerilog syntetyzowane operacje, w tym przypisania, mogą być wykonywane równolegle.

Różne języki opisu sprzętu radzą sobie z tymi problemami w różny sposób. W SystemVerilog wprowadzono pojęcie przypisania *ciągłego* i *proceduralnego*, aby odróżnić przypisywanie wartości do sieci i zmiennych. *Przypisania ciągłe* są głównie przeznaczone do symulowania przesyłania sygnałów w sieciach, natomiast *przypisania proceduralne* są przeznaczone do symulowania przesyłania danych między rejestrami.

Nowymi operatorami w SystemVerilog są operatory przypisania ciągłego z dodatkowymi operacjami działań, tak jak w języku C (np. += lub -=).

Równoległe i sekwencyjne wykonywanie operatorów w zadaniach w języku SystemVerilog jest obsługiwane przez *blokujące* i *nieblokujące* proceduralne operatory przypisania.

W normalnych warunkach syntezator ignoruje operatory kontroli czasu w instrukcjach przypisania, z wyjątkiem operatora listy czułości @. Jednak warto, aby projektant wiedział, jak funkcjonują operatory sterowania czasem przy operacjach przypisania.

W tym rozdziale omówiono również inicjalizację zmiennych oraz proceduralne przypisania ciągłe, które nie podlegają syntezie i są wykorzystywane tylko przy symulacji.

10.1. Przypisania w języku SystemVerilog

Przypisanie (*assignment*) to podstawowy mechanizm określania wartości elementów sieci i zmiennych. Istnieją dwie podstawowe formy przypisań:

- *przypisanie ciągłe* (*continuous assignment*), które przypisuje wartości do elementów sieci lub zmiennych;
- *przypisanie proceduralne* (*procedural assignment*), które przypisuje wartości do zmiennych.

Przypisanie ciągłe steruje sieciami lub zmiennymi w taki sam sposób, w jaki bramki sterują sieciami bądź wejściami rejestrów. Wyrażenie po prawej stronie można traktować jako układ kombinacyjny, który *w sposób ciągły steruje elementem sieci lub zmienną*. W odróżnieniu od tego przypisania proceduralne *umieszczają wartości w zmiennych*. Przypisanie proceduralne nie ma określonego czasu trwania, zamiast tego zmienna zachowuje swoją wartość aż do następnego przypisania proceduralnego do tej zmiennej.

Istnieją dwie dodatkowe formy przypisania, **assign-deassign** i **force-release**, które są nazywane *proceduralnymi przypisaniami ciągłymi* (*procedural continuous assignments*).

W SystemVerilog przypisania proceduralne są implementowane za pomocą operatorów *blokujących* (*blocking*) i *nieblokujących* (*nonblocking*). W instrukcji przypisania blokującego dwie części przypisania, lewa i prawa, są oddzielone znakiem równości (=), natomiast w instrukcji przypisania nieblokującego – parą znaków mniejszości i równości tworzących strzałkę (<=). Po prawej stronie przypisania może znajdować się dowolne wyrażenie, którego wartość można obliczyć. Lewa strona przypisania określa sieć lub zmienną, do której ma być przypisana wartość po prawej stronie. Lewa strona może przyjąć jedną z postaci przedstawionych w tabeli 10.1, w zależności od tego, czy przypisanie jest ciągłe czy proceduralne.

TABELA 10.1. Dopuszczalne formy lewej strony w wyrażeniach przypisania

Typ operatora	Lewa strona
Przypisanie ciągłe	• sieć lub zmienna (wektorowa bądź skalarna) • wybór bitu albo wybór części sieci wektorowej lub zmiennej spakowanej • konkatenacja lub zagnieżdżona konkatenacja dowolnej z powyższych postaci
Przypisanie proceduralne	• zmienna (wektorowa lub skalarna) • wybór bitu albo wybór części zmiennej spakowanej • tablica lub wybór elementu tablicy, fragmentu tablicy, słowa pamięci • konkatenacja bądź zagnieżdżona konkatenacja dowolnej z powyższych postaci

ŹRÓDŁO: oprac. własne.

Język SystemVerilog umożliwia również określenie wartości opóźnienia czasowego w instrukcji przypisania w następujący sposób:

```
#1ns r = a;                  // blokujący proceduralny operator przypisania
                             // z opóźnieniem czasowym
r = #1ns a;                  // blokujący proceduralny operator przypisania
                             // z opóźnieniem wewnętrznym
r <= #1ns a;                 // nieblokujący proceduralny operator przypisania
                             // z opóźnieniem wewnętrznym
assign #2.5ns sum = a + b;   // operator ciągłego przypisania z opóźnieniem czasowym
```

10.2. Przypisanie ciągłe

Przypisania ciągłe muszą określać wartości sieci lub zmiennych, zarówno wektorowych (spakowanych), jak i skalarnych. To przypisanie jest wykonywane za każdym razem, gdy zmienia się wartość prawej strony. Umożliwia modelowanie logiki kombinacyjnej bez określania relacji między bramkami. Zamiast tego w modelu określa się wyrażenie logiczne, które steruje siecią lub zmienną.

Istnieją dwie formy przypisań ciągłych: *przypisanie przy deklaracji sieci* (*net declaration assignments*) i *instrukcja (operator) przypisania ciągłego* (*continuous assign statements*) **assign**.

Uproszczona składnia polecenia przypisania ciągłego **assign** jest następująca:

assign (*delay*) *net_name* = *expression*;

gdzie: **assign** jest słowem kluczowym; *delay* jest wielkością opóźnienia czasowego; *net_name* jest nazwą zmiennej lub sieci po lewej stronie; *expression* jest wyrażeniem, którego wartość jest przypisywana do lewej strony. Ponadto w instrukcji przypisania **assign** można użyć słów kluczowych mocy logicznej (tylko dla sieci **trireg**) oraz **vectored**, **scaled** i **interconnect**.

Przypisanie ciągłe podczas deklaracji elementu sieci ma następującą składnię:

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

gdzie: *net_type* jest typem sieci; *strength* jest mocą logiczną (patrz podrozdział 3.2.4); [*size*] jest zakresem; *net_name* jest nazwą sieci; *expression* jest wyrażeniem przypisywanym.

W rzeczywistości składnia przypisania ciągłego podczas deklarowania sieci jest powtórzeniem operatora przypisania **assign**, tylko bez słowa kluczowego **assign**.

Przykłady przypisań ciągłych:

```
                                        // jawne przypisanie ciągłe
wire [31:0] mux_out;
assign mux_out = sel ? a : b;           // 32-bitowy multiplekser szyn a i b

                                        // niejawne przypisanie ciągłe
tri [0:15] # 2.7 buf_out = en ? in : 16'bz;   // trójstanowy bufor 16-bitowy
                                        // z opóźnieniem 2.7 jednostki czasu

// niejawne przypisanie ciągłe wskazujące moc logiczną
wire (strong1, pull0) [63:0] ALU_out = ALU_function(opcode, a, b);      // wywołanie funkcji
```

Operatory przypisania ciągłego **assign** mogą być używane do przypisywania wielu wartości do tej samej sieci, tzn. operator przypisania **assign** może być zastosowany do tej samej sieci wiele razy. Jednak zmienne mogą być sterowane tylko przez jeden ciągły operator przypisania **assign**. Na przykład dopuszczalne jest

```
module mult_assign_net
        (input wire a,b,s,
        output wand y);

        assign y = 1'b1;

        assign y = s ? a : b;    // wielokrotne przypisanie sieci wand

endmodule
```

i niedopuszczalne

```
module mult_assign_var
        (input wire a,b,s,
        output var logic y);

        assign y = 1'b1;

        assign y = s ? a : b;    // niedozwolone wielokrotne przypisanie zmiennej y

endmodule
```

Przypisanie ciągłe nie może również sterować częścią sieci zdefiniowaną przez użytkownika za pomocą słowa kluczowego **nettype**. Innymi słowy, lewa strona dla sieci zdefiniowanej przez użytkownika, ze zdefiniowanym typem sieci **nettype**, nie może zawierać operacji indeksowania ani wyboru. Na przykład:

```
nettype wand my_AND;
my_AND a[7:0];

assign a[0] = 1'b1;          // niedopuszczalne
```

10.3. Przypisania proceduralne

Przypisania proceduralne znajdują się wewnątrz procedur, takich jak **always**, **initial**, **task** i **function**. Przypisaniami proceduralnymi można sterować za pomocą operatorów **if** i **case**.

Prawą stroną przypisania proceduralnego może być dowolne wyrażenie, które oblicza jakąś wartość. Typ zmiennej po lewej stronie może jednak ograniczać wyrażenie dozwolone po prawej stronie. Lewą stroną przypisania proceduralnego może być:

- zmienna pojedyncza;
- zmienna zagregowana;
- wybór bitu, wybór części lub fragmentu spakowanej tablicy;
- fragment niespakowanej tablicy.

W języku SystemVerilog występują trzy rodzaje operatorów przypisań proceduralnych:

- *operator proceduralny przypisania blokującego* (*blocking*) (=);
- *operator proceduralny przypisania nieblokującego* (*nonblocking*) (<=);
- *operatory przypisania* (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=).

Proceduralne operatory przypisań blokujących i nieblokujących definiują różne *przepływy proceduralne* (*procedural flows*) w blokach sekwencyjnych **begin...end**.

Operacje przypisania są semantycznie równoważne operatorowi przypisania blokującego.

10.3.1. Operator przypisania blokującego (=)

Proceduralny operator przypisania blokującego (=) ma następujący format:

variable = *expression*;

gdzie *expression* jest pewnym wyrażeniem, a *variable* jest zmienną, do której zostanie przypisana wartość obliczonego wyrażenia.

Gdy symulator napotka w kodzie proceduralny operator blokujący, wyrażenie jest obliczane, a wynik jest przypisywany do zmiennej *variable*. W obrębie operacji ograniczonych konstrukcją **begin-end** wykonywanych sekwencyjnie kolejna operacja jest blokowana do czasu zakończenia procedury przypisywania. Stąd nazwa tego operatora proceduralnego – *blokujący* (*blocking*).

Przykłady operatorów blokujących:

```
// Pierwsze przypisanie zmienia m, a drugie zmienia n
module blocking_in_out (inout logic n,m);
always begin
                m = n;
                n = m;
        end
endmodule

// Inwersja wejść
module not_inputs(
input a,b,c,d,
        output logic w,x,y,z);

        always begin
                w =~a;
                x =~b;
                y =~c;
                z =~d;
        end
endmodule

// Łączenie wejść za pomocą bramek
module gating_outputs(
input a,b,c,d,
        output logic w,x,y,z);

        always begin
                w = a & b;
                x= b | c;
                y= c ^ d;
                z= d ~^ a;
        end
endmodule
```

```
// Szeregowe połączenie wejść i wyjść za pomocą bramek
module gating_in_out_1(
input a,b,c,d,
        output logic w,x,y,z);

        always begin
                w = a & b;
                x = w | c;
                y = x ^ d;
                z = y ~^ a;
        end
endmodule

// Inny przykład szeregowego łączenia bramek
module gating_in_out_2(
input a,b,c,d,
        output logic z);

        logic w,x,y;
        always begin
                w = a & b;
                x = w | c;
                y = x ^ d;
                z = y ~^ a;
        end
endmodule

// Podłączanie wielu wyjść
module connect_outs(
input a,b,c,d,
        output logic y,z);

        always begin
                y =!a;
                y =!b;
                z =!c;
                z =!d;
        end
endmodule
```

```
// Opis prostego układu kombinacyjnego
module comb_circuit(
input data,
        output logic c, d);

always @(*)
begin
                c = data;
                d = ~c & ~data;
end
endmodule
```

Czytelnik może wykonać zadanie określenia, jaki schemat kompilator zsyntetyzuje dla każdego przykładu. Następnie należy użyć dostępnego kompilatora i sprawdzić swoje przypuszczenia oraz wyjaśnić wyniki.

Oprócz znaku (=) operatora przypisania blokującego w składni można również używać symboli operacji przypisania (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=).

10.3.2. Operator przypisania nieblokującego (<=)

Proceduralny operator przypisania nieblokującego ma następujący format:

variable **<=** *expression*;

gdzie *expression* jest wyrażeniem, a *variable* jest zmienną, do której zostanie przypisana wartość obliczonego wyrażenia.

Gdy symulator napotka w kodzie operator przypisania nieblokującego, wyrażenie jest obliczane, ale przypisanie wynikowej wartości do zmiennej po lewej stronie jest odkładane do końca czasu symulacji. W grupie **begin-end** sekwencyjnie wykonywanych operatorów wykonanie kolejnego operatora nie jest blokowane, stąd nazwa *proceduralny nieblokujący operator przypisania* (*non-blocking*). Innymi słowy, następny operator zostanie wykonany bez czekania na zakończenie przypisania.

Przykład użycia nieblokujących operatorów przypisania:

```
module non_blocking_in_out
(inout logic n,m);

        always begin            // Oba przypisania zostaną obliczone przed zmianą n i m
                m <= n;
                n <= m;
        end
endmodule
```

W powyższym przykładzie oczekuje się, że dane z wyjścia dwukierunkowego *n* będą przesyłane do wyjścia dwukierunkowego *m* i jednocześnie z wyjścia *m* do wyjścia *n*. Ponieważ jednak fizycznie nie jest możliwe jednoczesne przesyłanie danych w dwóch kierunkach na tej samej linii, syntezator odmawia realizacji podanego opisu.

Zachęcamy czytelnika do zastąpienia operatora przypisania blokującego (=) operatorem przypisania nieblokującego (<=) w omawianych wcześniej przykładach i spróbowania określenia wyników syntezy, a następnie użycia dostępnego kompilatora i potwierdzenia swoich przypuszczeń oraz wyjaśnienia wyników.

10.3.3. Specyfika syntezy operatorów przypisań blokujących i nieblokujących

Jak już wcześniej wspomniano, operatory blokujące (=) i nieblokujące (<=) w blokach sekwencyjnych (**begin-end**) definiują różne *wątki proceduralne* (tzn. kolejność wykonywania instrukcji proceduralnych). Operator przypisania blokującego wstrzymuje wykonanie operatorów, które po nim następują, natomiast operator przypisania nieblokującego tego nie czyni. Z tego powodu operator nieblokujący (<=) jest używany, gdy kilka przypisań zmiennych musi zostać wykonanych w jednym i tym samym kroku czasowym bez uwzględniania kolejności i zależności przypisań od siebie.

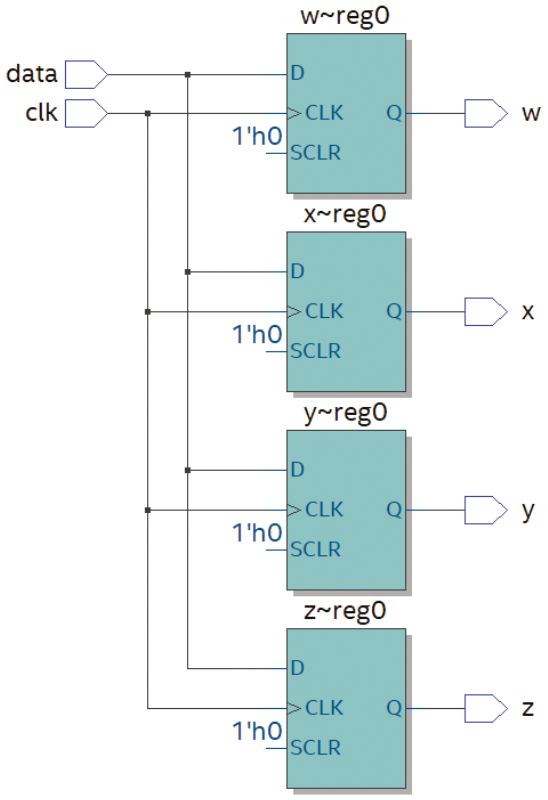
W praktyce operator przypisania blokującego (=) jest zwykle używany do opisywania układów (obwodów) kombinacyjnych, a operator przypisania nieblokującego (<=) – do opisywania układów sekwencyjnych sterowanych zboczem sygnału synchronizacji (zegara).

Rozważmy następujący przykład użycia operatora przypisania blokującego:

Przykład 10.1. Równoległa implementacja przerzutników z wykorzystaniem proceduralnych operatorów przypisania blokującego

```
module blocking_assignments(
input data, clk,
        output logic x, y, w, z);

always @(posedge clk)
begin
                x = data;
                y = x;
                w = y;
                z = w;
end
endmodule
```

Spróbujmy przewidzieć wynik syntezy na podstawie opisu podanego w przykładzie 10.1. Przede wszystkim możemy śmiało powiedzieć, że wartości zmiennych *x*, *y*, *w* i *z* zostaną wygenerowane na wyjściach przerzutników sterowanych narastającym zboczem sygnału zegara *clk*, ponieważ cała grupa instrukcji przypisania jest sterowana przez operator wrażliwości @(**posedge** clk). Pierwsza instrukcja przypisania przypisuje wejściową wartość *data* do zmiennej *x*, blokując wykonanie wszystkich kolejnych instrukcji przypisania. Druga instrukcja przypisania musi przypisać wartość zmiennej *x* do zmiennej *y*, ale wartość zmiennej *x* jest już znana i jest równa wartości zmiennej wejściowej *data*. Rozumując w ten sposób, można dojść do wniosku, że wszystkie zmienne *x*, *y*, *w* i *z* muszą być przypisane do tej samej zmiennej wejściowej *data*. W ten sposób układ opisany w module blocking_assignments jest realizowany za pomocą czterech przerzutników, których wyjścia tworzą wartości zmiennych *x*, *y*, *w* i *z*, a wejścia wszystkich przerzutników są określone przez wartość zmiennej wejściowej *data* (rysunek 10.1).



RYS. 10.1. Implementacja operatorów przypisania blokującego z przykładu 10.1
ŹRÓDŁO: oprac. własne.

W poniższym kodzie użyto operatora przypisania nieblokującego.

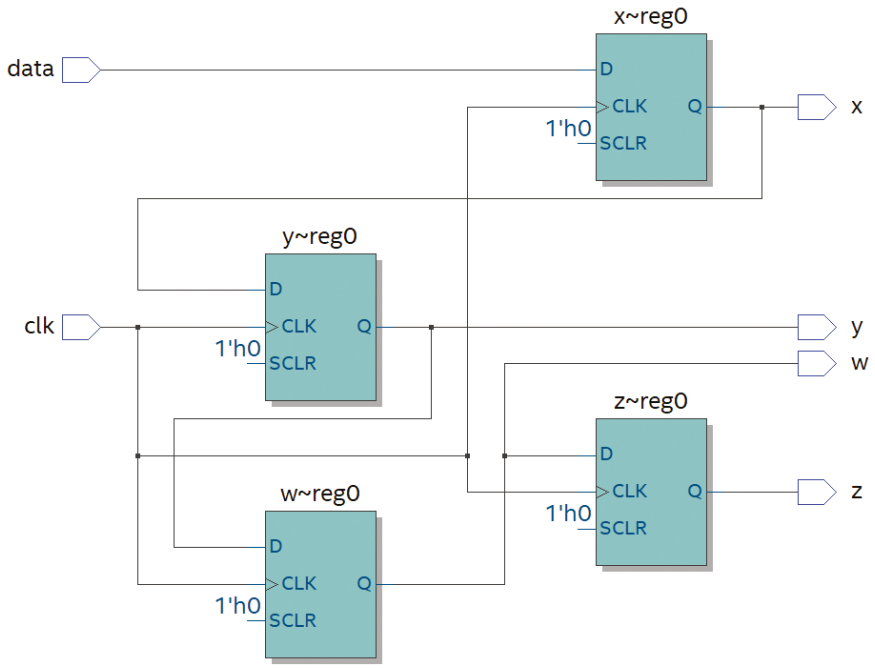
Przykład 10.2. Szeregowa implementacja przerzutników z wykorzystaniem proceduralnych operatorów przypisania nieblokującego

```
module non_blocking_assignments(
input data, clk,
        output logic x, y, w, z);

always @(posedge clk)
begin
                x <= data;
                y <= x;
                w <= y;
                z <= w;
end
endmodule
```

Spróbujmy przewidzieć wynik syntezy opisu podanego w przykładzie 10.2. Wartości zmiennych *x*, *y*, *w* i *z* zostaną wygenerowane na wyjściach przerzutników sterowanych narastającym zboczem sygnału zegara *clk*, ponieważ cała grupa instrukcji przypisania jest sterowana przez operator wrażliwości @(**posedge** clk). Pierwsza instrukcja przypisania przypisuje wejściową wartość *data* do zmiennej *x* bez blokowania wykonania wszystkich kolejnych instrukcji przypisania. Druga instrukcja przypisania musi przypisać wartość zmiennej *x* do zmiennej *y*, ale wartość zmiennej *x* nie jest jeszcze znana; poznamy ją dopiero w następnym cyklu zegara, po nadejściu narastającego zbocza sygnału synchronizacji *clk*. Dlatego wejście przerzutnika, którego wyjście określa wartość zmiennej *y*, powinno być połączone z jego wyjściem, który determinuje wartość zmiennej *x*. W ten sposób przerzutniki, których wyjścia określają wartości *x*, *y*, *w* i *z*, powinny być połączone szeregowo, jak pokazano na rys. 10.2.

Podsumujmy niektóre wnioski z zastosowań proceduralnych operatorów przypisania. Syntezator zwykle ignoruje operatory sterowania czasem w instrukcjach przypisania. Wyjątkiem jest operator listy czułości @, który sprawdza zbocze sygnału zegarowego. Gdy taki operator steruje blokiem proceduralnym zawierającym operatory przypisania blokującego, wynikiem będzie synteza równoległego układu elementów pamięci. Z kolei w przypadku zastosowania operatorów przypisania nieblokującego zostanie zsyntetyzowany schemat szeregowo połączonych elementów pamięci.

Aby uniknąć potencjalnych wyścigów w syntetyzowanym układzie i uzyskać wiarygodne dane przy symulacji z zerowym czasem opóźnienia, do opisu układów kombinacyjnych należy używać operatora przypisania blokującego (=), a do opisu układów sekwencyjnych – operatora przypisania nieblokującego (<=).



RYS. 10.2. Implementacja operatorów przypisania nieblokującego z przykładu 10.2

ŹRÓDŁO: oprac. własne.

10.4. Sterowanie czasem w proceduralnych operatorach przypisania

Podczas syntezy większość konstrukcji sterowania czasem w SystemVerilog jest ignorowana przez syntezator. Jednak projektant powinien wiedzieć, w jaki sposób sterowanie czasem jest realizowane w proceduralnych operatorach przypisania.

10.4.1. Sterowanie czasem w operatorach przypisania blokującego

Operatory sterowania czasem (**#**, **@**, **wait**) mogą występować zarówno przed operatorem przypisania, jak i przed wyrażeniem. W pierwszym przypadku proceduralny operator przypisania blokującego ma następujący format:

timing_control variable **=** *expression*;

gdzie: *timing_control* jest jednym z proceduralnych operatorów sterowania czasem **#**, **@** lub **wait**; *variable* – zmienna; *expression* – wyrażenie.

Gdy symulator napotka w kodzie taką konstrukcję, obliczenie wartości wyrażenia *expression* jest opóźniane o czas określony w operatorze *timing_control*. Jako przykład rozważmy następujący fragment kodu:

```
# 10                x = a;
@(posedge clk)      y = b;
```

Gdy symulator napotka pierwszy wiersz kodu, najpierw odczeka opóźnienie 10 jednostek czasu, a następnie przypisze wartość zmiennej *a* do zmiennej *x*. Gdy symulator natrafi na drugą linię kodu, poczeka na narastające zbocze sygnału *clk*, po czym przypisze wartość zmiennej *b* do zmiennej *y*. Wspólną cechą tych operatorów przypisania jest to, że albo czekają (blokują) przez określony czas opóźnienia, albo czekają na wystąpienie określonego zdarzenia przed wykonaniem przypisania. Innymi słowy, wartość wyrażenia po prawej stronie znaku równości nie jest obliczana, dopóki nie upłynie czas oczekiwania.

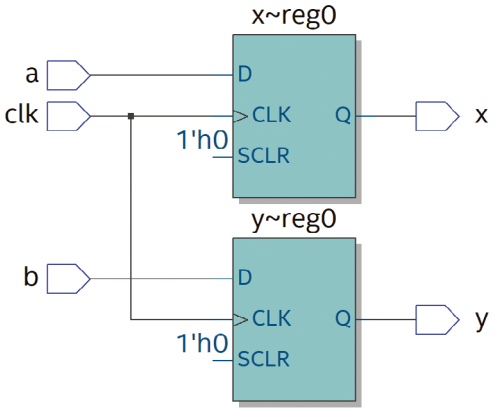
Poniższy przykład przedstawia implementację omawianego fragmentu kodu.

Przykład 10.3. Proceduralne operatory przypisania blokującego wraz z operatorami sterowania czasem

```
module blocking_st (
input a, b, clk,
      output logic x, y);

always begin
            # 10                x = a;
            @(posedge clk)      y = b;
end
endmodule
```

Układ zaimplementowany przez syntezator jest pokazany na rys. 10.3.

Spróbujmy wyjaśnić działanie syntezatora. Ignoruje on operator oczekiwania #10. Na pierwszy rzut oka może się wydawać, że wyjście *x* w układzie z rys. 10.3 powinno być połączone prostym obwodem z wejściem *a*. Należy jednak zauważyć, że blok instrukcji **always** jest wykonywany jako nieskończona pętla. Dlatego drugie i wszystkie kolejne przypisania zmiennej *x* do wartości wejścia *a* nie nastąpią przed nadejściem narastającego zbocza sygnału zegarowego *clk*. Stąd w układzie pomiędzy wejściem *a* a wyjściem *x* pojawia się przerzutnik sterowany narastającym zboczem zegara *clk*. Zgodnie z oczekiwaniami zmienna wyjściowa *y* jest generowana na wyjściu przerzutnika, którego wejściem danych jest zmienna *b*.



RYS. 10.3. Implementacja operatorów przypisania blokującego wraz z operatorami sterowania czasem z przykładu 10.3

ŹRÓDŁO: oprac. własne.

10.4.2. Opóźnienia wewnętrzne w operatorach przypisania blokującego

Gdy w proceduralnym operatorze przypisania operator sterowania czasem jest ustawiony przed wyrażeniem, to opóźnienie nazywane jest opóźnieniem *wewnętrznym* (*intra-assignment delay* lub *intra-assignment timing control*). Operator przypisania blokującego w przypadku zastosowania opóźnień wewnętrznych ma następujący format:

variable = *timing_control expression*;

Konstrukcja ta działa w następujący sposób. Gdy symulator napotka w kodzie proceduralny operator przypisania blokującego, obliczane jest wyrażenie *expression*. Przypisanie obliczonej wartości do zmiennej *variable* jest jednak wykonywane po upływie czasu określonego w instrukcji *timing_control*. W grupie **begin-end** sekwencyjnie wykonywanych poleceń wykonanie następnego polecenia jest blokowane do czasu zakończenia przypisania (po upływie czasu opóźnienia).

Jako przykład rozważmy następujący fragment kodu:

```
x = #10 a;
y = @(posedge clk) b;
```

Gdy symulator natrafi na pierwszy operator, najpierw zostanie obliczona wartość wyrażenia *a*, następnie zaś zostanie ona przypisana do pewnej zmiennej tymczasowej. Potem następuje okres oczekiwania wynoszący 10 jednostek czasu. W tym momencie wykonanie procesu, tzn. kolejnego operatora, zostanie zablokowane.

Po upłynięciu czasu oczekiwania wartość zmiennej tymczasowej zostanie przypisana do zmiennej *x*. Działanie pierwszego operatora można wyrazić w następującym fragmencie kodu:

```
begin
        aTemp = a;          /* obliczenie wyrażenia a i przypisanie wyniku
                               do zmiennej tymczasowej aTemp */
        # 10 x = aTemp;     /* oczekiwanie na 10 jednostek czasu i przypisanie
                               wartość aTemp do zmiennej x */
end
```

Gdy symulator napotka drugi operator, najpierw obliczy wartość wyrażenia *b* i przypisze tę wartość do pewnej zmiennej tymczasowej. Następnie czeka na narastające zbocze sygnału *clk*. Gdy to nastąpi, wartość *y* zostanie przypisana do zmiennej tymczasowej. Działanie drugiego operatora można wyrazić w następującym fragmencie kodu:

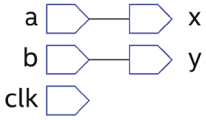
```
begin
        bTemp = b;                  /* obliczenie wyrażenia b i przypisanie wyniku
                                       do zmiennej tymczasowej bTemp */
        @(posedge clk) y = bTemp;   /* oczekiwanie na narastające zbocze sygnału clk
                                       i przypisanie wartości bTemp do zmiennej y */
end
```

Poniższy przykład przedstawia implementację powyższego fragmentu kodu.

Przykład 10.4. Proceduralne operatory przypisania blokującego wraz z opóźnieniami wewnętrznymi

```
module blocking_st_intra(
input a, b, clk,
        output logic x, y);

always begin
                x = # 10 a;
                y = @(posedge clk) b;
end
endmodule
```

Zsyntetyzowany obwód jest pokazany na rys. 10.4.



RYS. 10.4. Implementacja blokujących operatorów przypisania z opóźnieniami wewnętrznymi z przykładu 10.4

ŹRÓDŁO: oprac. własne.

Na podstawie rys. 10.4 można stwierdzić, że syntezator ignoruje wewnętrzne opóźnienia w przypadku operatorów przypisania blokującego.

10.4.3. Sterowanie czasem w operatorach przypisania nieblokującego

Operator przypisania nieblokującego, w przypadku gdy używany jest jeden z operatorów sterowania czasem (**#**, **@**, **wait**), ma następujący format:

timing_control variable **<=** *expression*;

gdzie: *timing_control* jest jednym z proceduralnych operatorów sterowania czasem **#**, **@** lub **wait**; *variable* – zmienna; *expression* – wyrażenie.

Gdy symulator napotka w kodzie taką konstrukcję, obliczenie wartości wyrażenia *expression* jest opóźniane o czas określony w operatorze *timing_control.*

Jako przykład rozważmy następujący fragment kodu:

```
# 10                x <= a;
@(posedge clk)      y <= a & b;
```

Gdy symulator napotka pierwszą linię kodu, najpierw wykona opóźnienie o 10 jednostek czasu, a dopiero potem do zmiennej *x* zostanie przypisana wartość zmiennej *a*. Nie spowoduje to zablokowania wykonania następnego polecenia. Gdy symulator wykona drugi wiersz kodu, najpierw zostanie obliczone wyrażenie *a* & *b*. Następnie symulator oczekuje na narastające zbocze sygnału *clk*, po czym do zmiennej *y* zostanie przypisana wartość obliczonego wcześniej wyrażenia *a* & *b*.

Poniższy przykład przedstawia implementację powyższego fragmentu kodu.

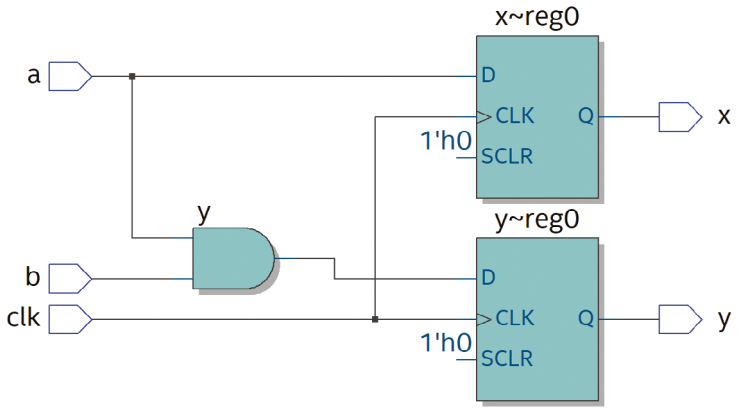
Przykład 10.5. Proceduralne operatory przypisania nieblokującego wraz z operatorami sterowania czasem

```
module non_blocking_st(
input a, b, clk,
```

```
        output logic x, y);

always begin
                # 10 x <= a;
                @(posedge clk) y <= a & b;
end
endmodule
```

Spróbujmy przewidzieć typ układu, który ma być zsyntetyzowany. Wartości zmiennych *x* i *y* będą generowane na wyjściach przerzutników, które są sterowane narastającym zboczem sygnału *clk*. Wejście przerzutnika, którego wyjście generuje wartość zmiennej *y*, będzie poprzedzone układem kombinacyjnym, jak opisano powyżej.

Implementacja modułu non_blocking_st została przedstawiona na rys. 10.5.



RYS. 10.5. Implementacja operatorów przypisania nieblokującego wraz z operatorami sterowania czasem z przykładu 10.5

ŹRÓDŁO: oprac. własne.

Jak widać na rys. 10.5, potwierdziły się nasze oczekiwania co do wyglądu zsyntetyzowanego schematu.

10.4.4. Opóźnienia wewnętrzne w operatorach przypisania nieblokującego

Operator przypisania nieblokującego w sytuacji, gdy używane są opóźnienia wewnętrzne, ma następujący format:

variable <= *timing_control expression*;

Konstrukcja ta działa w następujący sposób. Gdy symulator napotka w kodzie operator przypisania nieblokującego, wyrażenie *expression* jest obliczane, a obliczona wartość zmiennej *variable* jest przypisywana po czasie określonym w operatorze *timing_control*. W grupie poleceń ograniczonych konstrukcją **begin-end** następujących sekwencyjnie wykonanie kolejnego operatora *nie jest blokowane*. Należy pamiętać, że w taki sposób można symulować *opóźnienia transmisji danych*.

Jako przykład rozważmy następujący fragment kodu:

```
x <= #10 a;
y <= @(posedge clk) a & b;
```

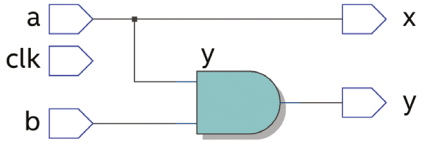
Gdy symulator napotka pierwszy operator, najpierw obliczy wartość wyrażenia *a* i przypisze tę wartość do pewnej zmiennej tymczasowej. Nie blokuje to wykonania procesu: następny operator kontynuuje pracę. Następnie po upływie 10 jednostek czasu wartość *x* zostanie przypisana do zmiennej tymczasowej.

Poniższy przykład przedstawia implementację powyższego fragmentu kodu.

Przykład 10.6. Proceduralne operatory przypisania nieblokującego z opóźnieniami wewnętrznymi

```
module non_blocking_st_intra(
input a, b, clk,
        output logic x, y);

always begin
                x <= # 10 a;
                y <= @(posedge clk) a & b;
end
endmodule
```

Implementacja modułu non_blocking_st_intra została przedstawiona na rys. 10.6.



RYS. 10.6. Przykład zastosowania operatorów przypisania nieblokującego z opóźnieniami wewnętrznymi

ŹRÓDŁO: oprac. własne.

Na podstawie rys. 10.6 można stwierdzić, że syntezator ignoruje wewnętrzne opóźnienia w przypadku operatorów przypisania nieblokującego.

10.5. Proceduralne przypisania ciągłe

W języku SystemVerilog proceduralne przypisania ciągłe są definiowane za pomocą par operatorów proceduralnych **assign-deassign** oraz **force-release**.

Składnia operatorów **assign** i **force**:

assign *variable* = *expression*;
force *variable* = *expression*;

gdzie *variable* jest zmienną, a *expression* jest wyrażeniem.

Jeśli jakakolwiek zmienna w wyrażeniu *expression* po prawej stronie zmieni wartość, wyrażenie zostanie przeliczone, a uzyskana w ten sposób wartość zostanie przypisana zmiennej *variable* po lewej stronie znaku równości.

Proceduralny operator przypisania ciągłego **assign** jest nadrzędny w stosunku do wszystkich proceduralnych przypisań zmiennych zdefiniowanych przez proceduralny operator przypisania (=).

Operator proceduralny **deassign** kończy proceduralne przypisanie ciągłe. Oznacza to, że wartość zmiennej pozostaje niezmieniona do momentu przypisania zmiennej nowej wartości za pomocą proceduralnego przypisania (=) lub proceduralnego przypisania ciągłego (**assign**).

Lewa strona instrukcji przypisania **assign** może być pojedynczą zmienną lub konkatenacją zmiennych i nie może być wyborem bitu lub wyborem części zmiennej.

W poniższym przykładzie przedstawiono zastosowanie operatorów proceduralnych **assign** i **deassign** do opisu działania przerzutnika D z wejściami *preset* i *clear*:

Przykład 10.7. Użycie operatorów proceduralnych **assign** i **deassign** w opisie behawioralnym przerzutnika D

```
module my_dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
logic q;

always @(clear or preset)
if (!clear)
assign q = 0;          // nieobsługiwane w programie Quartus
else if (!preset)
assign q = 1;
else
deassign q;
```

```
always @(posedge clock)
q = d;
endmodule
```

Para operatorów **force-release** jest podobna do operatorów **assign-deassign**, ale operator **force** może być stosowany zarówno do zmiennych, jak i do sieci. Lewa strona operatora **force** może być pojedynczą zmienną, wyborem bitowym lub częścią wektora elementów sieci.

Gdy do sieci zostanie zastosowany proceduralny operator **force**, zastępuje on wszystkie źródła sieciowe (wyjścia bramek, moduły i przypisania ciągłe) do momentu zastosowania do sieci proceduralnego operatora zwolnienia **release**. Po zwolnieniu sieci (za pomocą operatora **release**), do jej elementu natychmiast przypisywana jest wartość zdefiniowana w źródłach sieciowych.

Należy zauważyć, że proceduralne operatory ciągłe (pary **assign-deassign** i **force--release**) nie są obsługiwane przez narzędzia syntezy i są używane tylko przy symulacji.

10.6. Wnioski

Przypisanie (*assignment*) jest podstawowym mechanizmem określania wartości elementów sieci i zmiennych. Istnieją dwie podstawowe formy przypisań: *przypisanie ciągłe* (*continuous assignment*), które przypisuje wartości do sieci lub zmiennych, oraz *przypisanie proceduralne* (*procedural assignment*), które przypisuje wartości do zmiennych.

Przypisanie ciągłe steruje sieciami lub zmiennymi w taki sam sposób jak bramki. Wyrażenie po prawej stronie można traktować jako układ kombinacyjny, który w sposób ciągły steruje siecią bądź zmienną. Przypisania proceduralne umieszczają wartości w zmiennych. Zmienna zachowuje wartość przypisania aż do następnego proceduralnego przypisania tej zmiennej.

Istnieją dwie dodatkowe formy przypisania, **assign-deassign** i **force-release**, które są nazywane *proceduralnymi przypisaniami ciągłymi* (*procedural continuous assignments*). Proceduralne operatory przypisania ciągłego (pary **assign-deassign** i **force--release**) nie są obsługiwane przez narzędzia syntezy i są używane tylko w projektach symulacyjnych.

Lewą stroną przypisania ciągłego mogą być: sieć lub zmienna (wektorowa bądź skalarna); wybór bitu albo wybór części sieci wektorowej lub zmiennej spakowanej; konkatenacja lub zagnieżdżona konkatenacja dowolnej z powyższych lewych stron.

Lewą stroną przypisania proceduralnego mogą być: zmienna (wektorowa albo skalarna); wybór bitu lub wybór części zmiennej spakowanej; tablica bądź wybór elementu tablicy, fragmentu tablicy, słowa pamięci; konkatenacja albo zagnieżdżona konkatenacja dowolnej z powyższych lewych stron.

Przypisanie ciągłe jest wykonywane za każdym razem, gdy zmienia się wartość prawej strony. Umożliwia ono modelowanie logiki kombinacyjnej bez określania relacji między bramkami. Zamiast tego w modelu określa się wyrażenie logiczne, które steruje siecią lub zmienną.

Istnieją dwie formy przypisań ciągłych: przypisania przy deklaracji sieci i operator *przypisania ciągłego* **assign**.

Dzięki operatorowi przypisania ciągłego **assign** do tej samej sieci można przypisać wiele wartości. Zmienne mogą być kontrolowane tylko przez jeden operator przypisania ciągłego **assign**. Przypisanie ciągłe nie może również sterować częścią sieci zdefiniowaną przez użytkownika za pomocą słowa kluczowego **nettype**.

Przypisania proceduralne znajdują się w obrębie procedur, takich jak: **always**, **initial**, **task** i **function**. Przypisania proceduralne można kontrolować za pomocą operatorów **if** i **case**.

W języku SystemVerilog występują trzy rodzaje *proceduralnych operatorów przypisania*:

- *proceduralne operatory przypisania blokującego* (*blocking*) (=);
- *proceduralne operatory przypisania nieblokującego* (*nonblocking*) (<=);
- *operatory przypisania* (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=).

Operatory proceduralnych przypisań blokujących i nieblokujących definiują różne *przepływy proceduralne* (*procedural flows*) w blokach sekwencyjnych **begin...end**.

Operacje przypisania są semantycznie równoważne z operatorem przypisania blokującego.

Gdy symulator napotka w kodzie proceduralny operator przypisania blokującego, wyrażenie jest obliczane, a wynikowa wartość jest przypisywana do zmiennej *variable*. W grupie poleceń **begin-end** wykonywanych sekwencyjnie następne polecenie jest blokowane do momentu zakończenia procedury przypisania. Stąd nazwa tego proceduralnego operatora przypisania – *blokujący* (*blocking*).

Gdy symulator napotka w kodzie proceduralny operator przypisania nieblokującego, wyrażenie *expression* jest obliczane, ale przypisanie wynikowej wartości do zmiennej po lewej stronie jest odkładane do końca czasu symulacji. W grupie poleceń **begin-end** wykonywanych sekwencyjnie następny operator nie jest blokowany, stąd nazwa *proceduralny operator przypisania nieblokującego* (*non-blocking*).

Operator przypisania nieblokującego (<=) jest używany, gdy kilka przypisań zmiennych musi być wykonanych w jednym i tym samym interwale czasowym bez uwzględniania kolejności i zależności przypisań od siebie.

W praktyce operator przypisania blokującego (=) jest zwykle używany do opisywania układów kombinacyjnych, natomiast operator przypisania nieblokującego (<=) do opisywania układów sekwencyjnych sterowanych zboczem sygnału zegarowego.

Normalnie syntezator ignoruje operatory sterowania czasem w operatorach przypisania. Wyjątkiem jest operator listy czułości @, który sprawdza zbocze sygnału synchronizacji.

Blok proceduralny zawierający operatory przypisania blokującego jest realizowany przez układ równoległych elementów pamięci, a blok proceduralny zawierający operatory przypisania nieblokującego jest realizowany przez układ elementów pamięci połączonych szeregowo.

11. Procesy

Proces (*process*) w języku SystemVerilog to pewne ogólne pojęcie zbioru czynności związanych z przetwarzaniem danych, które dotyczy zarówno symulacji, jak i syntezy. W SystemVerilog procesy są tworzone przez procedury (**always** i **initial**), zadania i funkcje.

W języku SystemVerilog istnieje również pojęcie *wątku proceduralnego* (*procedural thread*). Wątki tworzą procedury, instrukcje w blokach **fork-join**, procesy dynamiczne i ciągłe instrukcje przypisania **assign**. Wątki te są sterowane poprzez sterowanie procesami.

Procesy w SystemVerilog są sterowane przez *opóźnienia* (*delays*) i *zdarzenia* (*events*). Opóźnienia są wykorzystywane tylko w symulacji i są ignorowane przy syntezie. Jednak konstrukcje związane ze zdarzeniami w SystemVerilog są w pełni syntezowalne. Zdarzenia obejmują zmiany poziomów sygnałów, jak również nadejście zboczy sygnałów. Zdarzenia sygnałowe są wykrywane przez operator *sterowania zdarzeniami* **@**, zwany też *operatorem czułości (wrażliwości)*, po którym w nawiasach znajdują się obserwowane sygnały na *liście czułości* (*events*). Problem polega na tym, że jeśli wartości jakiejś zmiennej generowanej w bloku proceduralnym nie są określone dla wszystkich możliwych wartości zmiennych wejściowych (sygnałów), to poprzednia wartość zmiennej musi być gdzieś przechowywana. W tym celu syntezator automatycznie wprowadza do układu *zatrzask* (*latch*). W rezultacie zamiast układu kombinacyjnego powstaje obwód z elementami pamięci.

Jednym z powodów wprowadzenia zatrzasków do układu jest fakt, że nie wszystkie sygnały wejściowe (zmienne) znajdują się na liście czułości. Język SystemVerilog używa konstrukcji **@*** i **@(*)** do wciągnięcia wszystkich zmiennych wejściowych na listę czułości, co pozwala uniknąć częstych błędów zdarzających się podczas modyfikacji projektu. Konstrukcje **@*** i **@(*)** nie zapobiegają jednak automatycznemu wprowadzeniu zatrzasków do układu ani nie pozwalają odpowiedzieć na pytanie, jaki układ zamierzał opisać projektant: kombinacyjny, zatrzaskowy czy sekwencyjny.

Aby rozwiązać ten problem, SystemVerilog dodaje do ogólnej procedury **always** wyspecjalizowane procedury **always_comb**, **always_latch** i **always_ff**, aby opisać odpowiednio *logikę kombinacyjną* (*combinational*), *zatrzaskową* (*latched*) i *sekwencyjną* (*sequential*).

Język SystemVerilog jest również rozszerzony o szereg konstrukcji upraszczających sterowanie procesami.

Zanim jednak powiemy o procedurach i sterowaniu procesami, przyjrzyjmy się ściśle związanym z nimi zagadnieniom sterowania czasem proceduralnym i grupowania operatorów w bloki proceduralne.

11.1. Sterowanie czasem proceduralnym

Sterowanie czasem proceduralnym, chociaż bardziej związane z symulacją, jest również szeroko stosowane podczas opisywania projektów do syntezy.

W SystemVerilog istnieją dwa rodzaje jawnego sterowania proceduralnym czasem podczas wykonywania operatorów proceduralnych: *sterowanie opóźnieniami* (*delay control*) i *sterowanie zdarzeniami* (*event control*). W tym pierwszym wyrażenie przed instrukcją proceduralną określa długość czasu pomiędzy napotkaniem instrukcji a momentem faktycznego wykonania operatora (patrz punkt 10.4). Z kolei sterowanie zdarzeniami pozwala na opóźnienie wykonania operatora do momentu wystąpienia zdarzenia.

Zdarzenie może być zmianą w sieci lub zmiennej, wówczas określa się je jako *zdarzenie niejawne* (*implicit event*). Może ono także mieć zadeklarowaną nazwę i być inicjowane przez inne procedury, wówczas jest to *zdarzenie jawne* (*explicit event*).

Zdarzenie polegające na zmianie wartości sieci lub zmiennej może być określone przez poziom sygnału (wysoki bądź niski) albo przez zbocze sygnału. W tym drugim przypadku rozróżnia się zbocze *pozytywne* (**posedge**) i *negatywne* (**negedge**), inaczej mówiąc, narastające i opadające.

Język SystemVerilog ma trzy operatory do proceduralnego sterowania czasem: *sterowanie opóźnieniem* **#**, *sterowanie zdarzeniem* **@** i *oczekiwanie* **wait**.

11.1.1. Operator sterowania opóźnieniami

Składnia operatora sterowania opóźnieniami jest następująca:

#*expression statement*;

gdzie *expression* to wyrażenie określające wartość opóźnienia w jednostkach czasu, a *statement* to operator, którego wykonanie jest opóźnione.

Operator # opóźnia wykonanie następnego operatora o wartość wyrażenia, która jest mierzona w jednostkach czasu. Operator # może być umieszczony przed dowolnym operatorem proceduralnym lub na prawo od znaku równości w operatorach przypisania. W tym drugim przypadku opóźnienie nazywane jest *opóźnieniem wewnętrznym* przypisania (*intra assignment* opóźnienia).

Jeśli wyrażenie opóźnienia zostało obliczone jako x lub z, to jest ono zerowe. Wyrażenie opóźnienia może mieć również wartość ujemną. W tym przypadku jest ono interpretowane jako uzupełnienie do dwóch bez znaku o tym samym rozmiarze co zmienna czasowa. W wyrażeniu opóźnienia dozwolone są parametry i zmienne. Na przykład:

```
#10 a = b;                  // opóźnia wykonanie przypisania o 10 jednostek czasu

a = #10 b;                  // wewnętrzne opóźnienie przypisania

parameter d = 10;           // definicja parametru d
#d a = b;                   // wartość opóźnienia jest określana przez parametr d

#((d+e)/2) a = b;           // opóźnienie jest średnią z d i e

#r r = r + 1;               // opóźnienie jest określone przez wartość zmiennej r
```

Opóźnienia dla operatorów przypisania zostały omówione bardziej szczegółowo w rozdziale 10.

11.1.2. Operator sterowania zdarzeniami @

Operator sterowania zdarzeniami @ pozwala na opóźnienie wykonania następnego operatora (bloku operatorów) do momentu wystąpienia co najmniej jednego z weryfikowalnych zdarzeń. Zdarzeniami tymi mogą być zmiany poziomu sygnału lub pojawienie się zbocza. Ponieważ operator @ wykrywa zdarzenia związane ze zmianami wartości sygnału, operator @ jest również nazywany *operatorem czułości* lub *wrażliwości* (*sensitivity statement*).

Wykonanie proceduralnego operatora może być zsynchronizowane albo ze zmianą wartości sieci lub zmiennej, albo z wystąpieniem jakiegoś zdarzenia. Synchronizowanie wykonania operatora ze zmianą w sieci bądź zmiennej nazywamy *wykrywaniem zdarzenia niejawnego* (*detecting an implicit event*).

Zdarzenie może być również oparte na kierunku zmiany wartości sygnału: w kierunku stanu wysokiego (**posedge**) lub w kierunku stanu niskiego (**negedge**). Zachowanie zdarzeń **posedge** i **negedge** przedstawiono w tabeli 11.1 i można je opisać w następujący sposób:

- **negedge** jest wykrywane przy przejściach z 1 do x, z lub 0 oraz z x lub z do 0;
- **posedge** jest wykrywane przy przejściach od 0 do x, z lub 1 oraz od x lub z do 1.

TABELA 11.1. Wykrywanie zdarzeń **posedge** i **negedge** w zależności od zmian wartości sygnału

Od	Do			
	0	**1**	**x**	**z**
0	brak zbocza	**posedge**	**posedge**	**posedge**
1	**negedge**	brak zbocza	**negedge**	**negedge**
x	**negedge**	**posedge**	brak zbocza	brak zbocza
z	**negedge**	**posedge**	brak zbocza	brak zbocza

ŹRÓDŁO: oprac. własne.

Zdarzenie **edge** oznacza zmianę sygnału na wartość 1 lub 0, tzn. występuje ono w momencie wykrycia zdarzenia **posedge** lub **negedge**.

Przy syntezie zdarzenia są zwykle wykrywane, gdy zmienia się wartość jakiegoś sygnału. Sygnał taki określa się jako *sygnał zegarowy* lub *sygnał synchronizujący*.

Przy symulacji do określenia wystąpienia zdarzenia mogą być używane wyrażenia. Zdarzenie to zmiana wartości wyrażenia. Jednak zmiana wartości dowolnego operandu w wyrażeniu bez zmiany wyniku wyrażenia nie jest zdarzeniem.

Na przykład:

```
@r a = b;                          // sterowane przez każdą zmianę wartości zmiennej r

@(posedge clock) a = b;            // sterowane przez narastające
                                   zbocze sygnału zegarowego

forever @(negedge clock) a = b;    // sterowane przez opadające
                                   zbocze sygnału zegarowego

forever @(edge clock) a = b;       // sterowane przez dowolne zbocze sygnału zegarowego
```

11.1.2.1. Lista czułości

W celu stworzenia *listy czułości* (*sensitivity list*) wiele zdarzeń można połączyć w jednym operatorze @ za pomocą logicznej operacji **or**. Aby tego dokonać, stosuje się operację **or** lub zdarzenia oddziela się przecinkami. Lista czułości w operatorze **@** jest używana do zatrzymania wykonania operatora proceduralnego przed wystąpieniem któregokolwiek ze zdarzeń z listy. Na przykład:

```
@(trig or enable) a = b;           // sterowanie przez sygnały trig lub enable

@(posedge clk_a or posedge clk_b or trig) a = b;    /* sterowanie przez zbocze sygnałów
clk_a, clk_b lub jakąkolwiek zmianą w sygnale trig */

always @(a, b, c, d, e)            // sterowanie przez sygnały a, b, c, d lub e
```

```
always @(posedge clk, negedge rstn)      /* sterowanie przez narastające zbocze sygnału
clk lub opadające zbocze sygnału rstn */

always @(a or b, c, d or e)              // sterowanie przez sygnały a, b, c, d lub e
```

11.1.2.2. Niejawna lista czułości @* lub @ (*)

Język SystemVerilog pozwala niejawnie opisać listę czułości operatora proceduralnego lub bloku dla wszystkich sygnałów, które mogą wywołać zdarzenie, czyli sygnałów, które są *wejściowymi* dla tego operatora. Służą do tego konstrukcje **@*** i **@ (*)**.

Konstrukcja @* automatycznie dodaje do listy czułości wszystkie identyfikatory sieci i zmiennych, które mogą wywołać zdarzenie. Są to sieci i zmienne w prawych częściach przypisań, w wywołaniach podprogramów, w wyrażeniach warunkowych operatora **case**, zmienne indeksowe w lewej części przypisania, zmienne w wyrażeniach elementów operatora **case**.

Przykłady niejawnych list czułości:

```
always @(*)                  // równoważnie @(a or b or c or d or f)
y = (a & b) | (c & d) | myfunction(f);

always @* begin              // równoważnie @(a or b or c or d or tmp1 or tmp2)
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end

always @* begin              // równoważnie @(b)
@(i) kid = b;                // i nie jest dodawane do @*
end

always @* begin              // równoważnie @(a or b or c or d)
x = a ^ b;
@*                           // równoważnie @(c or d)
x = c ^ d;
end

always @* begin              // równoważnie @(a or en)
y = 8'hff;
y[a] = !en;
end
```

Konstrukcje @* i @(*) są często używane przy opisie układów kombinacyjnych, dzięki czemu projektant nie martwi się o wypisanie wszystkich wejść bloku proceduralnego lub operatora. Zauważmy, że użycie procedury **always_comb** jest jeszcze lepszym rozwiązaniem niż użycie konstrukcji @* (procedura **always_comb** jest omówiona w 11.3.2.2).

11.1.2.3. Sterowanie zdarzeniami warunkowymi

W języku SystemVerilog element sterujący zdarzeniem *@event* może mieć kwalifikator **iff**, co oznacza „jeśli i tylko jeśli". Wyrażenie zdarzenia jest wyzwalane tylko wtedy, gdy wyrażenie po kwalifikatorze **iff** jest prawdziwe. Na przykład:

```
always @(a iff enable == 1)
y <= a;
```

W tym przykładzie zostanie wykonany operator y <= a; pod warunkiem, że wystąpiło zdarzenie a i sygnał *enable* ma wartość 1.

Kwalifikator **iff** ma pierwszeństwo przed **or**. Zauważmy, że kwalifikator **iff** nie jest obsługiwany w systemie Quartus.

11.1.3. Operator oczekiwania wait

Operator **wait** jest *wrażliwy na poziom* sygnałów (*level-sensitive*) w przeciwieństwie do operatora @, który jest *wrażliwy na zbocze* sygnałów (*edge-sensitive*). Składnia operatora **wait**:

wait (*expression*) *statement*;

Operator **wait** ocenia wyrażenie *expression* i jeśli jest ono fałszywe, blokuje wykonanie operatora *statement*, dopóki wyrażenie *expression* nie będzie prawdziwe. Na przykład:

```
begin
wait (!enable) #10 a = b;
#10 c = d;
end
```

Powyższy fragment kodu działa w następujący sposób. Jeżeli przy wejściu do bloku **begin-end** wartość sygnału *enable* wynosi 1, to operator **wait** opóźnia następne polecenie (#10 a = b;), aż sygnał *enable* zmieni się na 0. Jeżeli przy wejściu do bloku **begin-end** wartość *enable* wynosi już 0, to zadanie „a = b;" zostanie wykonane z opóźnieniem 10 jednostek czasu.

Kolejny przykład zastosowania operatora **wait**:

```
wire [7 : 0]  addr;              // szyna adresowa
wait (addr != 8'hFA)             // sprawdzenie wartości na szynie adresowej
     data = mem [addr];          // podłączenie komórki pamięci mem z adresem addr
                                 // do szyny data
```

W tym przykładzie odczyt z pamięci jest wykonywany tylko wtedy, gdy adres komórki pamięci nie jest równy wartości 8'hFA.

Zauważ, że operatory # i **wait** nie są syntetyzowalne i są ignorowane przy syntezie. Tylko operator sterowania zdarzeniami @ jest syntetyzowalny.

11.1.4. Sterowanie czasem proceduralnym wewnątrz przypisania

Wewnętrzne opóźnienie przypisania (*intra-assignment* opóźnienia) opóźnia przypisanie nowej wartości do lewej strony, ale wyrażenie po prawej stronie jest oceniane przed opóźnieniem, a nie po nim. Składnia wewnętrznego opóźnienia przypisania:

```
variable = delay_or_event_control expression
variable <= delay_or_event_control expression
```

gdzie: *variable* to zmienna; *delay_or_event_control* to sterowanie opóźnieniem lub zdarzeniami; *expression* to wyrażenie.

Wewnętrzne opóźnienie przypisania i sterowanie zdarzeniami mogą być stosowane zarówno do przypisań blokujących, jak i nieblokujących. Opóźnienia wewnętrzne w operatorach przypisania są omówione w rozdziale 10.

Sterowanie z kwalifikatorem powtórzenia **repeat** definiuje opóźnienie wewnątrz przypisania w zależności od określonej liczby zdarzeń. Na przykład:

```
repeat (3) @ (event_expression)     // trzykrotne wykonanie wyrażenia event_expression

repeat (-3) @ (event_expression)    // nie wykona event_expression

repeat (a) @ (event_expression)     /* z a równym -3 wykona event_expression,
                                       jeśli a jest zadeklarowane jako zmienna bez znaku,
                                       i nie wykona event_expression,
                                       jeśli a jest zadeklarowane jako zmienna ze znakiem */
```

Konstrukcja powtarzająca **repeat** jest wygodna w użyciu, gdy zdarzenia muszą być zsynchronizowane z licznikiem taktów zegara. W tabeli 11.2 przedstawiono zasady sterowania czasem proceduralnym wewnątrz przypisania.

TABELA 11.2. Różne sposoby sterowania czasem proceduralnym wewnątrz przypisania

Sterowanie czasem proceduralnym wewnątrz przypisania	
z konstrukcją wewnątrz przypisania	**bez konstrukcji wewnątrz przypisania**

a = #5 b;	**begin**
	temp = b;
	#5 a = temp;
	end

a = @(**posedge** clk) b;	**begin**
	temp = b;
	@(**posedge** clk) a = temp;
	end

a = **repeat**(3) @(**posedge** clk) b;	**begin**
	temp = b;
	@(**posedge** clk);
	@(**posedge** clk);
	@(**posedge** clk) a = temp;
	end

ŹRÓDŁO: oprac. własne.

Przyjrzyjmy się kilku przykładom zastosowania sterowania czasem proceduralnym wewnątrz przypisania.

Eliminacja warunków wystąpienia wyścigu. W poniższym kodzie możliwy jest warunek wystąpienia wyścigu, ponieważ instrukcje przypisania w bloku **fork-join** są wykonywane równolegle w tym samym momencie.

```
fork
    #5 a = b;
    #5 b = a;
join
```

W poniższym kodzie warunek wystąpienia wyścigu został wyeliminowany:

```
fork
    a = #5 b;
    b = #5 a;
join
```

ponieważ wewnętrzne opóźnienie przypisania powoduje, że wartości a i b są określane przed opóźnieniem, a przypisanie jest wykonywane po opóźnieniu.

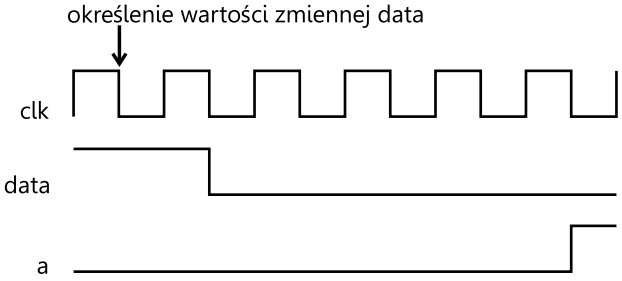
Obliczanie wyrażeń przed wystąpieniem zdarzenia. W poniższym przykładzie wyrażenia prawe są obliczane w momencie napotkania operatora przypisania, ale przypisania są opóźnione do momentu nadejścia narastającego zbocza sygnału zegarowego:

```
fork
        a = @(posedge clk) b;
        b = @(posedge clk) c;
join
```

Sterowanie powtarzalnością zdarzeń. Poniższy przykład pokazuje zarządzanie powtarzaniem zdarzeniami za pomocą wewnętrznego opóźnienia przypisania nieblokującego:

```
a <= repeat(5) @(posedge clk) data;
```

W tym przykładzie wartość zmiennej *data* jest obliczana w momencie wykrycia operatora przypisania, a zmienna *a* jest przypisywana do zmiennej *data* po pięciu powtórzeniach narastającego zbocza sygnału *clk*, jak pokazano na rys. 11.1.



RYS. 11.1. Sterowanie powtarzaniem zdarzeniami za pomocą dodatniego zbocza sygnału synchronizacji

ŹRÓDŁO: oprac. własne.

Poniższy przykład pokazuje bardziej złożone sterowanie, w którym obliczana jest suma powtórzeń zdarzeniami, a zdarzenia są wyzwalane przez różne zbocza różnych sygnałów:

```
a <= repeat (a+b) @(posedge phi1 or negedge phi2) data;
```

W tym przykładzie wartość *data* jest obliczana po wykryciu operatora przypisania. Po tym, jak liczba powtórzeń zbocza dodatnich *phi1* i ujemnych *phi2* będzie równa sumie *a+b*, zmiennej *a* przypisujemy wartość *data*. Nawet jeśli zdarzenia **posedge** phi1 i **negedge** phi2 wystąpiły w tym samym czasie symulacji, każde z nich jest wykrywane i liczone osobno.

11.2. Bloki proceduralne

Bloki proceduralne (*procedural blocks*) pozwalają na grupowanie operatorów proceduralnych w taki sposób, że są one traktowane jako jeden operator. W języku SystemVerilog istnieją dwa rodzaje bloków: *blok sekwencyjny* **begin-end** oraz *blok równoległy* **fork-join**.

Blok sekwencyjny jest ograniczony przez słowa kluczowe **begin** i **end**. Operatory w tym bloku są wykonywane w kolejności określonej w kodzie projektu.

Blok równoległy jest ograniczony przez słowa kluczowe **fork** i **join** oraz **join_any** lub **join_none**. Operatory w tym bloku są wykonywane jednocześnie, czyli równolegle.

11.2.1. Blok sekwencyjny begin-end

Sekwencyjny blok **begin-end** charakteryzuje się następującymi właściwościami:

- operatory są wykonywane sekwencyjnie, jeden po drugim;
- wartości opóźnień dla każdego operatora są przetwarzane w odniesieniu do czasu wykonania symulacji poprzedniego operatora;
- wyjście z bloku następuje po ostatnim operatorze.

Składnia bloku sekwencyjnego:

```
begin [: name]
        { block_item_declaration }
        { statement_or_null }
end [: name]
```

gdzie: *name* – nazwa bloku; *block_item_declaration* – deklaracja bloku; *statement_or_null* – operatory bloku.

Przyjrzyjmy się przykładom zastosowania bloku sekwencyjnego.

W poniższym przykładzie blok sekwencyjny pozwala, aby wynik dwóch następujących przypisań był deterministyczny.

```
begin
        a = b;
        c = a;          // c zachowuje wartość b
end
```

W powyższym przykładzie wartość zmiennej *b* jest najpierw przypisywana do zmiennej *a*, następnie zaś wartość zmiennej *a* jest przypisywana do zmiennej *c*.

W poniższym przykładzie w celu oddzielenia dwóch przypisań w czasie w bloku sekwencyjnym używane jest sterowanie zdarzeniem:

```
begin
        a = b;
        @(posedge clock) c = a;       // zadanie, opóźnione do dodatniego zbocza
                                      // sygnału zegarowego clock
end
```

Poniższy przykład pokazuje, jak można wykorzystać kombinację bloku szeregowego i sterowania opóźnieniem do ustawienia przebiegu w określonych punktach czasu.

```
parameter d = 50;             // deklaracja parametru d
logic [7:0] r;                // deklaracja 8-bitowej zmiennej r

begin                         // przebieg sygnału sterowany przez szeregowe opóźnienia
        #d r = 'h35;
        #d r = 'hE2;
        #d r = 'h00;
        #d r = 'hF7;
end
```

Zauważmy, że we wszystkich tych przykładach użyto operatora przypisania blokującego.

11.2.2. Bloki równoległe (fork-join, join_any i join_none)

Blok równoległy **fork-join** pozwala na tworzenie procesów równoległych z operatorów proceduralnych. Cechy charakterystyczne bloku równoległego to:

- operatory są wykonywane jednocześnie;
- należy uwzględnić wartości opóźnień dla każdego operatora w stosunku do czasu symulacji wejścia do bloku;
- sterowanie opóźnieniami może być wykorzystane do zapewnienia porządku czasowego zadań;
- sterowanie powoduje wyjście z bloku, gdy zostanie wykonany ostatni operator uporządkowany czasowo na podstawie typu słowa kluczowego **join**;
- użycie wywołań funkcji wewnątrz bloku jest ograniczone.

Składnia bloków równoległych:

```
fork [: name]
        { block_item_declaration }
        { statement_or_null }
join [: name]
```

gdzie: *name* – nazwa bloku; *block_item_declaration* – deklaracja bloku; *statement_or_null* – operatory bloku. Zamiast słowa kluczowego **join** można użyć słów kluczowych **join_any** lub **join_none**.

Każdy operator bloku równoległego jest wykonywany jako proces równoległy. Elementy sterowania czasem proceduralnym w bloku **fork-join** nie są sekwencyjnie uporządkowane w czasie, jak w sekwencyjnym bloku **begin-end**. Poniższy przykład opisuje przebieg z poprzedniego przykładu, przy czym zamiast szeregowego bloku **begin-end** zastosowano równoległy blok **fork-join**.

```
fork
        #50 r = 'h35;
        #100 r = 'hE2;
        #150 r = 'h00;
        #200 r = 'hF7;
join
```

Blok równoległy może zablokować wywołujący go proces macierzysty (nadrzędny). Charakter blokowania tego procesu określają słowa kluczowe **join**, **join_any** i **join_none**. Tabela 11.3 pokazuje, w jaki sposób każde z tych słów kluczowych wpływa na kontynuację procesu nadrzędnego.

TABELA 11.3. Opcje blokowania procesu nadrzędnego za pomocą bloku **fork-join**

Opcja	Opis
join	Proces macierzysty jest zablokowany, dopóki nie zakończą się wszystkie procesy utworzone przez blok **fork**
join_any	Proces macierzysty jest zablokowany do czasu zakończenia dowolnego z procesów utworzonych przez blok **fork**
join_none	Proces macierzysty kontynuuje pracę jednocześnie ze wszystkimi procesami utworzonymi przez blok **fork**

ŹRÓDŁO: oprac. własne.

Umieszczenie bloku **begin-end** wewnątrz bloku **fork-join** powoduje, że cały blok jest wykonywany jako jeden proces, a każdy operator jest wykonywany sekwencyjnie. Na przykład:

```
fork
        begin                   // jeden proces z dwoma operatorami
                statement1;
                statement2;
        end
join
```

W poniższym przykładzie dochodzi do dwóch procesów. Pierwszy czeka przez 20 ns, a drugi czeka na wystąpienie określonego zdarzenia eventA. Ponieważ podano słowo kluczowe **join**, proces nadrzędny musi zostać zablokowany do momentu zakończenia obu procesów, czyli do czasu, gdy upłynie 20 ns i zostanie wywołane zdarzenie eventA.

```
fork
        begin
                $display( “First Block\n” );
                # 20ns;
        end
        begin
                $display( “Second Block\n” );
                @eventA;
        end
join
```

Operator powrotu **return** w kontekście bloku **fork-join** jest niedopuszczalny i spowoduje błąd w czasie kompilacji. Na przykład:

```
task wait_20;
        fork
                # 20;
                return;                 // Niedopuszczalne: powrót nie jest możliwy, ponieważ
                                        // zadanie funkcjonuje w innym procesie
        join_none
endtask
```

Zmienne zadeklarowane w bloku **fork_join** są inicjalizowane wartościami ustawionymi w momencie ich zadeklarowania, przy każdym wejściu do bloku, zanim zostaną utworzone jakiekolwiek procesy. Na przykład:

```
initial
        for( int j = 1; j <= 3; ++j )
                fork
                        automatic int k = j;            // lokalna kopia k dla każdej wartości j
                        #k $write( „%0d”, k );
                        begin
                                automatic int m = j;    // wartość m jest nieokreślona
                                ...
                        end
                join_none
```

Należy pamiętać, że bloki **fork-join** nie są syntetyzowalne i są używane tylko podczas symulacji projektów. Sekwencyjne i równoległe wykonywanie operatorów w syntezie opisywane jest za pomocą blokujących (=) i nieblokujących (<=) operatorów przypisania w bloku **begin-end**.

11.2.3. Czas rozpoczęcia i zakończenia bloków proceduralnych

Zarówno bloki sekwencyjne, jak i równoległe mają *czas rozpoczęcia* (*start time*) i *czas zakończenia* (*finish time*). Dla bloków sekwencyjnych czas rozpoczęcia to czas wykonania pierwszego operatora, a czas zakończenia to czas wykonania ostatniego operatora. W przypadku bloków równoległych czas rozpoczęcia jest taki sam dla wszystkich operatorów, a czas zakończenia zależy od typu zastosowanej konstrukcji **join** (tabela 11.3).

Bloki szeregowe i równoległe mogą być wbudowane w siebie, co pozwala na łatwe wyrażenie złożonych struktur sterowania z dużą dokładnością. Gdy bloki są wbudowane jeden w drugi, ważne są ich czasy rozpoczęcia i zakończenia. Wykonanie musi trwać do momentu osiągnięcia czasu zakończenia bloku, czyli do jego całkowitego zakończenia.

Poniższy przykład pokazuje operatory z przykładu w podrozdziale 11.2.2 zapisane w odwrotnej kolejności, które wytwarzają ten sam przebieg.

```
fork                        // jednostka równoległa
      #200 r = 'hF7;
      #150 r = 'h00;
      #100 r = 'hE2;
      #50 r = 'h35;
join
```

Gdy przypisanie musi nastąpić po dwóch oddzielnych zdarzeniach, mamy tzw. *łączenie zdarzeń* (*joining of events*), wówczas przydatny może być blok **fork-join**.

```
begin                       // sekwencyjny blok
      fork                  // równoległy blok
            @Aevent;        // pierwsze zdarzenie
            @Bevent;        // drugie zdarzenie
      join
      a = b;
end
```

Poniższy przykład pokazuje dwa sekwencyjne bloki, z których każdy zaczyna wykonywać się w momencie wystąpienia określonego zdarzenia sterującego. Ponieważ kontrola zdarzeniami (@enable_a i @enable_b) znajduje się w bloku **fork-join**, są one wykonywane równolegle, a zatem bloki sekwencyjne będą także wykonywane równolegle.

```
fork                        // równoległy blok
    @enable_a
    begin                   // pierwszy blok sekwencyjny
        #ta wa = 0;
        #ta wa = 1;
        #ta wa = 0;
    end

    @enable_b
    begin                   // drugi blok sekwencyjny
        #tb wb = 1;
        #tb wb = 0;
        #tb wb = 1;
    end
join
```

11.2.4. Nazwy bloków proceduralnych

Bloki sekwencyjne i równoległe mogą być nazywane poprzez dodanie nazwy bloku po słowach kluczowych **begin** lub **fork**. Nazwany blok tworzy nowy obszar w hierarchii. Czynność ta służy do następujących celów:

- pozwala na hierarchiczne odwoływanie się do zmiennych lokalnych, parametrów i zdarzeń zadeklarowanych wewnątrz bloku nazwanego;
- pozwala na odwoływanie się do bloku w instrukcjach wyłączających **disable**.

Odpowiednia nazwa bloku może być zdefiniowana po słowie kluczowym **end**, **join**, **join_any** lub **join_none**, poprzedzonym dwukropkiem. Jeśli nazwa na końcu bloku różni się od nazwy bloku na początku, kompilator generuje komunikat o błędzie.

Przykład bloku nazwanego:

```
begin : block A
...
end : block A
```

Dla lepszej czytelności i dokumentacji kodu zaleca się, aby zawsze umieszczać nazwę na końcu nazwanych bloków proceduralnych.

W języku SystemVerilog nazwy oznaczające koniec danego bloku mogą być również umieszczone po następujących słowach kluczowych: **endchecker**, **endclass**, **endclocking**, **endconfig**, **endfunction**, **endgroup**, **endinterface**, **endmodule**, **endpackage**, **endprimitive**, **endprogram**, **endproperty**, **endsequence** i **endtask**.

Nazwa końcowa bloku generacji może być też umieszczona po słowie kluczowym **end** bloku generacji.

11.2.6. Etykiety operatorów proceduralnych

Operatory proceduralne, czyli takie, które są umieszczone wewnątrz bloków **begin-end** lub **fork-join**, mogą mieć etykiety. W SystemVerilog bloki **begin-end** i **fork-join** są uważane za operatory i mogą mieć również etykiety. Nazwa etykiety jest zapisywana przed operatorem i oddzielona od niego dwukropkiem:

label : *statement*;

gdzie *label* to nazwa etykiety, a *statement* to operator proceduralny lub blok proceduralny.

Jeśli blok proceduralny ma etykietę, to jest ona nazwą bloku i może być podana po słowach kluczowych **end**, **join**, **join_any** lub **join_none**. Na przykład:

```
labelA : begin
...
end labelA
```

Zabronione jest jednoczesne określenie etykiety i nazwy bloku proceduralnego po słowach kluczowych **begin** lub **fork**.

Ogólnie rzecz biorąc, etykieta operatora tworzy nazwany blok **begin-end** wokół operatora i tworzy nowy obszar hierarchii. Na przykład:

```
label_B : a = b;
```

jest równoważne:

```
begin : labelB
        a = b;
end : labelB
```

Zauważmy, że operatory inicjalizacji w pętlach **for** i **foreach** również mogą mieć etykiety. Taka etykieta nazywana jest *niejawnym blokiem utworzonym przez pętlę* (*implicit block created by the loop*). Etykiety mogą być również określone przed blokiem generacji **begin-end** oraz przed *twierdzeniem* lub *zapewnieniem równoległym* (*concurrent assertion*).

Oznaczony operator można wyłączyć za pomocą polecenia **disable**. Przypomnijmy, że operator **disable** działa tylko przy symulacji i jest ignorowany podczas syntezy.

11.3. Procedury strukturalne

W SystemVerilog procesy są bezpośrednio związane z *procedurami strukturalnymi* (*structured procedures*). Innymi słowy, proces jest pewnym rodzajem procedury strukturalnej. Do procedur strukturalnych w SystemVerilog należą:

- procedury **initial**;
- procedury **always** i jej pochodne **always_comb**, **always_latch** i **always_ff**;
- procedury **final**;
- zadania;
- funkcje.

Każda procedura strukturalna generuje proces. Operator **wait** i jego pochodne (**wait fork** i **wait_order**) umożliwiają blokowanie wykonywania procesów. Procesy mogą być również zakończone za pomocą operatorów **disable** i **disable fork**.

Składnia procedur **initial** i **always**:

initial *statement*

always_keyword statement

gdzie: **initial** to słowo kluczowe; *statement* to operator proceduralny; *always_keyword* to jedno ze słów kluczowych **always**, **always_comb**, **always_latch** lub **always_ff**. Przypomnijmy, że zamiast operatora proceduralnego można użyć bloku proceduralnego **begin-end** lub **fork-join**.

Procedury **initial** i **always** są aktywowane na początku symulacji. Procedura **initial** jest wykonywana tylko raz i kończy się po zakończeniu działania operatora **initial** lub bloku **initial**. W przeciwieństwie do procedury **initial** procedura **always** jest wykonywana wielokrotnie (jako pętla nieskończona) i kończy się po zakończeniu symulacji.

Zazwyczaj procedury **initial** i **always** są używane razem z blokami proceduralnymi **begin-end** lub **fork-join**. W tym przypadku mówimy o *blokach* **initial** lub *blokach* **always**.

W języku SystemVerilog nie ma ograniczeń na liczbę procedur **initial** i **always** w projekcie, a także na kolejność ich wykonywania, tzn. zaczynają one działać jednocześnie i równolegle. Można to porównać do obwodu elektrycznego, gdzie po włączeniu zasilania wszystkie elementy obwodu zaczynają pracować jednocześnie i równolegle.

Procedury **final** są uruchamiane na koniec czasu symulacji i są wykonywane tylko raz.

Zadania i funkcje są uruchamiane z jednego lub kilku miejsc w innych procedurach.

Oprócz procedur strukturalnych język SystemVerilog zawiera następujące *konteksty proceduralne* (*procedural contexts*): *wyrażenia punktu pokrycia* (*coverage point*

expressions), *elementy dopasowania sekwencji zapewnień* (*assertion sequence match items*) oraz *bloki akcji* (*action blocks*).

Język SystemVerilog udostępnia następujące sposoby sterowania strumieniami w ramach procedury:

- operatory wyboru (**if-else**, **case**), operatory pętli i operatory przejść;
- wywołania podprogramów (zadań i funkcji);
- bloki sekwencyjne (**begin-end**) i równoległe (**fork-join**);
- proceduralne operatory sterowania czasem (**#**, **@**, **wait**);
- operatory sterowania procesami (**wait fork**, **wait_order**, **disable**, **disable fork**).

11.3.1. Procedura initial

Procedura **initial** jest wykonywana tylko raz i kończy się po zakończeniu wykonania ostatniego operatora. Poniższy przykład ilustruje użycie procedury **initial** do inicjalizacji jednoportowej pamięci RAM [5].

Przykład 11.1. Jednoportowa pamięć RAM z inicjalizacją wszystkich słów zerami

```
module ram_initial
        #(parameter SIZE=8, ADDR_SIZE=5)
        (input [SIZE-1:0] data_in,                  // szyna danych wejściowych
        input [ADDR_SIZE-1:0] addr,                 // szyna adresowa
        input clk, wr,                              // sygnały sterujące
        output logic [SIZE-1:0] data_out);          // szyna wyjściowa danych

        logic [SIZE-1:0] mem [2**ADDR_SIZE-1:0];    // macierz pamięci

        initial                                     // inicjalizacja pamięci
                for (int i=0; i<ADDR_SIZE; i++)
                        mem[i] = 0;

        always @(posedge clk)
                if (wr)         mem[addr] = data_in;    // zapisywanie
                else            data_out = mem[addr];   // odczyt
endmodule
```

Zmienne mogą być inicjalizowane w ten sam sposób. Na przykład:

```
logic a;
initial a = 0;
```

Innym typowym zastosowaniem procedury **initial** jest specyfikacja opisów przebiegów, które wykonywane są jednorazowo w celu dostarczenia danych początkowych dla głównej części symulowanego obwodu.

```
initial begin
inputs = 'b000000;          // inicjalizacja w czasie zerowym
#10 inputs = 'b011001;      // pierwszy szablon
#10 inputs = 'b011011;      // drugi szablon
#10 inputs = 'b011000;      // trzeci szablon
#10 inputs = 'b001000;      // ostatni szablon
end
```

11.3.2. Procedury always

W języku SystemVerilog występują cztery formy procedur **always**: **always**, **always_comb**, **always_latch** i **always_ff**. Wszystkie powtarzane są w sposób ciągły przez czas trwania symulacji.

W Verilogu istnieje tylko jedna procedura **always**. To, jaki typ logiki (kombinacyjny, sekwencyjny czy zatrzaskowy) zostanie zaimplementowany na jej podstawie, jest wydedukowane przez kompilator automatycznie z kodu projektu. Z jednej strony nakłada to dodatkowe obciążenie na narzędzia projektowe, a z drugiej strony może być źródłem trudnych do znalezienia błędów. Na przykład zamiast obwodu kombinacyjnego może zostać zaimplementowany obwód z wyjściem zatrzaskowym, czyli obwód z pamięcią.

Aby uniknąć takich błędów, do języka SystemVerilog dodano procedury **always_comb**, **always_latch** i **always_ff**, które służą do wyłącznej implementacji odpowiednio: logiki kombinacyjnej, zatrzaskowej i sekwencyjnej. Te specjalne procedury nie tylko upraszczają pracę narzędzi programistycznych w określaniu logiki danego typu z kodu źródłowego, lecz także dodatkowo dokumentują kod projektu poprzez jednoznaczne wskazanie intencji twórcy.

11.3.2.1. Procedura ogólnego przeznaczenia **always**

Procedura ogólnego przeznaczenia **always** jest reprezentowana przez słowo kluczowe **always**. Używa się go do opisu powtarzalnych zachowań, np. do opisu generatorów sygnałów zegarowych. Procedura ta, podobnie jak w Verilogu, może być użyta do opisu dowolnego rodzaju logiki: kombinacyjnej, sekwencyjnej lub zatrzaskowej (z zatrzaskami na wyjściach). Na bardziej abstrakcyjnych poziomach symulacji blok proceduralny **always** może zostać użyty do symulacji algorytmów, bez jawnego reprezentowania szczegółów implementacji ich zachowania, np. w metodyce projektowania ASMD-FSMD [5]. Procedura **always** jest również szeroko stosowana w modułach testowych do opisu sygnałów taktowania i innych zadań weryfikacyjnych, które muszą być powtarzane podczas całego procesu symulacji.

Zazwyczaj procedura **always** jest stosowana w połączeniu z operatorami sterowania jej czasu. Blok **always** jest nieskończoną pętlą, która cyklicznie wykonuje zawarte wewnątrz niego operatory. Blok **always** może zawierać dowolną liczbę elementów sterowania czasem lub zdarzeniami (operatorów **#**, **@** i **wait**), które mogą być określone w dowolnym miejscu bloku.

W przypadku syntezy procedura **always** jest często używana w połączeniu z instrukcją sterowania zdarzeniami **@**. W tym przypadku składnia procedury **always** jest następująca:

always @ (*sensitivity_list*) *statement*

gdzie *sensitivity_list* – lista czułości, a *statement* – operator proceduralny (albo blok proceduralny).

Lista czułości to lista sygnałów oddzielonych przecinkami lub słowami kluczowymi **or**, których zmiana wartości dowolnego z nich wznawia wykonanie operatora proceduralnego. Efekt powyższej listy czułości jest następujący: wykonanie operatora *statement* (lub grupy operatorów) zostanie wstrzymane do momentu, gdy któryś z sygnałów wymienionych na liście czułości zmieni swój stan.

Rozważmy, jak procedura ogólnego przeznaczenia **always** działa podczas implementacji różnych typów logiki, ponieważ jest ona również dostępna w języku SystemVerilog wraz z procedurami **always_comb**, **always_latch** i **always_ff**.

Kompilator wyprowadzi logikę kombinacyjną (tzn. podczas implementacji zostanie zsyntetyzowany układ kombinacyjny) z kodu projektu zawierającego blok proceduralny **always**, jeśli spełnione zostaną następujące warunki:

- lista czułości bloku **always** nie zawiera kwalifikatorów **posedge** i **endedge**;
- lista czułości bloku **always** zawiera wszystkie wejścia do danego bloku proceduralnego – są nimi sygnały, które otrzymują swoją wartość poza blokiem proceduralnym;
- wartości wszystkich zmiennych, które są zapisywane w bloku proceduralnym, muszą być określone dla wszystkich możliwych wartości wejść bloku;
- blok proceduralny nie może zawierać innych operatorów sterowania zdarzeniami;
- zmienne, do których przypisano wartości w bloku proceduralnym, nie mogą mieć przypisanych wartości w innych blokach proceduralnych.

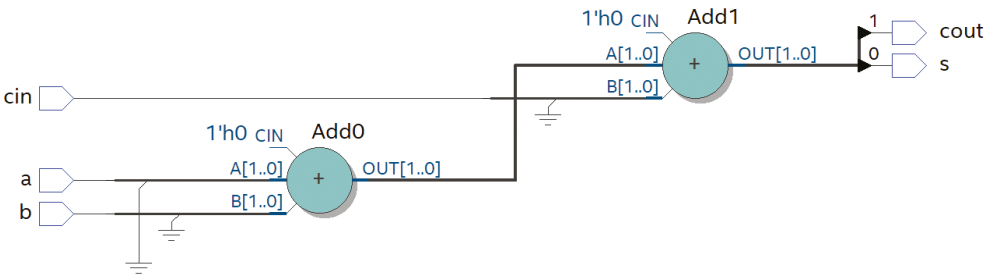
Zwróćmy uwagę na ostatni warunek, który musi być spełniony dla całego projektu. Poniższy przykład opisuje logikę kombinacyjną.

Przykład 11.2. Wariant sumatora z listą czułości

```
module adder_1_bit
(input a, b, cin,
      output logic s, cout);
```

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

Powyższy przykład opisuje sumator jednobitowy. Wejściem do bloku proceduralnego **always** są sygnały *a*, *b* (bity słowa) oraz *cin* (przeniesienie z poprzedniej cyfry). Zmienne, do których zapisywane są dane w bloku proceduralnym, to suma *s* oraz sygnał przeniesienia *cout*. Ponieważ blok **always** zawiera tylko jeden operator blokujący (=), nawiasy w bloku **begin-end** nie są potrzebne. Wyniki syntezy modułu przedstawiono na rys. 11.2.



RYS. 11.2. Widok projektu adder_1_bit na poziomie RTL

ŹRÓDŁO: oprac. własne.

Na rys. 11.2 widzimy, że zaimplementowano układ kombinacyjny składający się z dwóch prymitywów jednobitowych sumatorów systemu Quartus.

Aby logika zatrzaskowa z kodu projektu mogła być wydedukowana przez narzędzia projektowe (układ kombinacyjny z zatrzaskami na wyjściach), blok proceduralny **always** musi spełniać wszystkie warunki dedukcji logiki kombinacyjnej, z wyjątkiem tego, że wartości zmiennych mogą nie być zdefiniowane dla wszystkich możliwych wartości wejść.

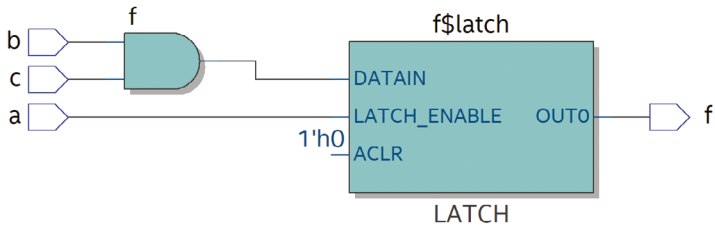
Poniższy przykład opisuje logikę zatrzaskową.

Przykład 11.3. Opis układu kombinacyjnego z zatrzaskami na wyjściach

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

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

W projekcie circ_latch wartość wyjściowa *f* nie jest określona dla $a = 0$, więc z kodu projektu zostanie wyprowadzona przez syntezator logika zatrzaskowa, a zsyntetyzowany układ będzie zawierał zatrzaski na wyjściu *f*, aby utrzymać poprzednią wartość zmiennej *f* przy $a = 0$.

Implementacja projektu circ_latch została przedstawiona na rys. 11.3.



RYS. 11.3. Widok projektu circ_latch na poziomie RTL

ŹRÓDŁO: oprac. własne.

Na rys. 11.3 widzimy, że wyjście *f* jest wyjściem zatrzasku (transparentnego przerzutnika typu D).

Aby z kodu projektu wywnioskować logikę sekwencyjną (w implementacji zostanie zsyntetyzowany układ sekwencyjny), blok proceduralny **always** musi spełniać następujące warunki:

- wszystkim sygnałom na liście czułości muszą towarzyszyć kwalifikatory **posedge** lub **negedge**;
- blok proceduralny nie może zawierać innych operatorów sterowania zdarzeniami;
- zmienne, do których przypisano wartości w bloku proceduralnym, nie mogą mieć przypisanych wartości w innych blokach proceduralnych.

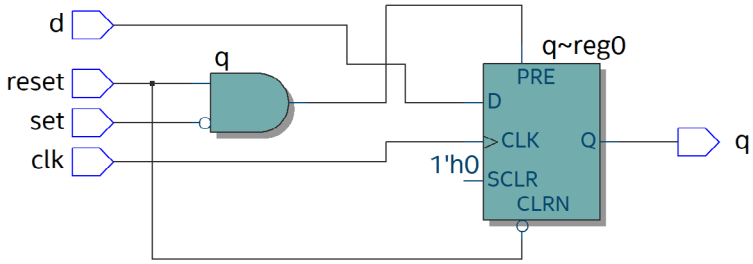
Zauważmy, że nie ma żadnych ograniczeń na wejścia bloku proceduralnego ani na wartości przypisywane zmiennym w ramach bloku.

Poniższy przykład opisuje logikę sekwencyjną.

Przykład 11.4. Przerzutnik typu D z sygnałami sterującymi

```
module my_dff_with_control_sygnals
(input d,                          // wejście danych
input clk, reset, set,             //sygnały sterujące
output logic q);                   // wyjście

      always @(posedge clk, negedge reset, negedge set)      // lista czułości
            if (~reset)     q <= 1'b0;      // zerowanie, gdy reset jest niski
            else if (~set)  q <= 1'b1;      // ustawienie, gdy set jest niski
            else            q <= d;         // przypisywanie danych
endmodule
```

Wyniki syntezy projektu my_dff_with_control_sygnals przedstawiono na rys. 11.4.



RYS. 11.4. Widok projektu my_dff_with_control_sygnals na poziomie RTL

ŹRÓDŁO: oprac. własne.

W liście czułości bloku **always** narzędzia syntezy przydzielają jeden sygnał dla logiki sekwencyjnej, który nie bierze udziału w sprawdzeniach wewnątrz bloku proceduralnego, tzn. w wyrażeniach logicznych operatora **if** i **case**. Ten sygnał jest traktowany jako *sygnał zegarowy* (*clock*) i jest on podłączany do wejścia zegarowego przerzutnika lub rejestru podczas realizacji. W przykładzie 11.3 sygnałem tym jest sygnał *clk*.

11.3.2.2. Procedura logiki kombinacyjnej always_comb

Język SystemVerilog zapewnia specjalną procedurę **always_comb** do opisu logiki kombinacyjnej. Na przykład:

```
always_comb
    if (!mode)      y = a + b;
    else            y = a - b;
```

W przeciwieństwie do procedury ogólnego przeznaczenia **always** procedura **always_comb** nie zawiera listy czułości, która jest automatycznie generowana przez kompilator. Lista ta obejmuje wszystkie sygnały wejściowe bloku proceduralnego, a także sygnały odczytywane przez funkcje, które są wywoływane z bloku proceduralnego **always_comb**.

Blok proceduralny **always_comb** wymaga również, aby zmienne po lewej stronie przypisania nie mogły być zmieniane przez żaden inny blok proceduralny (jeden z warunków syntezy logiki kombinacyjnej z procedurą **always**). Jest to jednak dozwolone, gdy pewne bity zmiennej są przypisane w jednej procedurze **always_comb**, a inne bity są przypisane w innej procedurze **always_comb**.

Jeśli blok proceduralny **always_comb** nie opisuje logiki kombinacyjnej, narzędzie programowe wydaje ostrzeżenie. Na przykład:

```
always_comb
    if (en)  y = a;
```

W tym przypadku sprzętowa implementacja zmiennej *y* wymaga zatrzasku do zapamiętania wartości, gdy *en* = 0.

W języku Verilog lista czułości może być opisana konstrukcją @* lub @(*) określającą wszystkie wejścia bloku proceduralnego. W języku SystemVerilog istnieją jednak istotne różnice pomiędzy procedurami **always @(*)** i **always_comb**.

Konstrukcja @* nie wymaga, aby zawartość bloku proceduralnego ogólnego przeznaczenia **always** spełniała warunki syntezy logiki kombinacyjnej. Wyspecjalizowany blok proceduralny **always_comb** nie tylko generuje listę czułości logiki kombinacyjnej, lecz także uniemożliwia innym blokom proceduralnym zapisywanie do zmiennych bloku danego proceduralnego, aby zapewnić prawdziwe zachowanie kombinacyjne.

Konstrukcja @* efektywnie tworzy listę czułości dla wszystkich wejść bloku proceduralnego. Jeśli jednak blok proceduralny wywołuje zadanie lub funkcję, to konstrukcja @* uwzględnia w liście czułości tylko argumenty wywołania zadania bądź funkcji. Konstrukcja @* nie tworzy listy czułości dla sygnałów odczytywanych z funkcji albo zadań wywoływanych przez blok proceduralny. Może to spowodować, że lista czułości będzie niekompletna.

Na przykład niech będzie zdefiniowana następująca funkcja:

```
function decode;                    // funkcja nie ma wejść
        begin
                case (sel)
                        2'b01: decode = d | e;
                        2'b10: decode = d & e;
                        default: decode = c;
                endcase
        end
endfunction
```

Zauważmy, że w tym przypadku zmienne *sel*, *c*, *d* i *e* nie są argumentami funkcji (w Verilogu i SystemVerilog jest to dozwolone). Następujący blok proceduralny

```
always @* begin
        a1 = data << 1;
        b1 = decode();
        ...
end
```

wygeneruje listę czułości @(data). Podczas gdy blok proceduralny

```
always_comb begin
        a2 = data << 1;
        b2 = decode();
        ...
end
```

wygeneruje kompletną listę czułości @(data, sel, c, d, e).

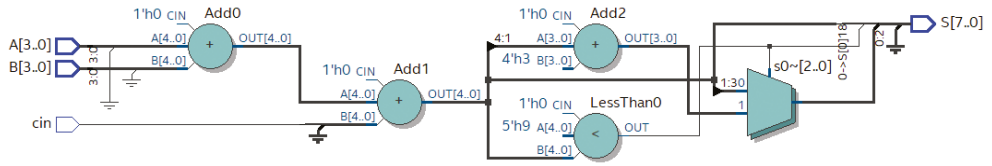
Jeśli opis procedury **always_comb** nie odpowiada zachowaniu logiki kombinacyjnej, narzędzie projektowe wygeneruje komunikat o błędzie.

Jako przykład rozważmy użycie procedury **always_comb** do opisu dwucyfrowego sumatora w kodzie BCD [4].

Przykład 11.5. Sumator dwóch cyfr w kodzie BCD

```
module adder_BCD_2_digits(
        input [3:0] A, B,           // cyfry wejściowe
        input cin,                  // przeniesienie z poprzedniej cyfry
        output [7:0] S);            // wynik

        logic [3:0] s0,  s1;        // dwie cyfry sumy w kodzie BCD
        wire [4:0] T;               // zmienna pomocnicza

        assign T = A + B + cin;     // obliczanie sumy w kodzie binarnym

always_comb                         // opis logiki kombinacyjnej
                if (T > 9) begin
                        s0 = T + 4'b0110;       // dodawanie poprawki
                        s1 = 4'b0001;
                end
                else begin
                        s0 = T;
                        s1 = 4'b0000;
                end

        assign S = {s1,s0};         // generowanie wyniku
endmodule
```

Wyniki syntezy projektu adder_BCD_2_digits przedstawiono na rys. 11.5.



RYS. 11.5. **Widok projektu adder_BCD_2_digits na poziomie RTL**

ŹRÓDŁO: **oprac. własne.**

11.3.2.3. Procedura logiki zatrzaskowej always_latch

Język SystemVerilog udostępnia specjalną procedurę **always_latch** do opisu *logiki zatrzaskowej* (*latched logic*), czyli logiki z zatrzaskami na wyjściach. Na przykład:

```
always_latch
if(ck) q <= d;
```

Reguły semantyczne procedury **always_latch** są identyczne jak reguły procedury **always_comb**. Kompilator generuje komunikat o błędzie, jeśli opis procedury **always_latch** nie jest zgodny z logiką zatrzaskową. W szczególności dotyczy to tworzenia listy czułości. Zmienne, których wartości są przypisane w bloku proceduralnym, nie mogą mieć przypisanych wartości w innych blokach proceduralnych. Natomiast dla bloku **always_latch** dopuszcza się, że wartości zmiennych wyjściowych bloku nie są określone dla wszystkich możliwych wartości zmiennych wejściowych. W tym przypadku odpowiednie wyjścia będą miały *zatrzaski* (*latchs*), czyli przezroczyste przerzutniki D.

Kompilator wygeneruje komunikat o błędzie, jeśli opis procedury **always_latch** nie pasuje do logiki zatrzaskowej.

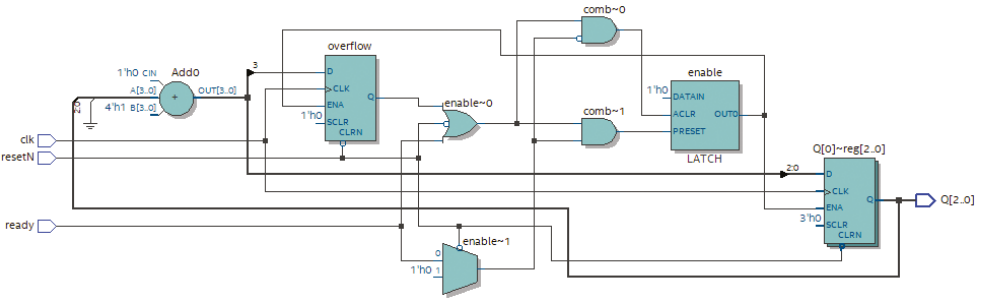
Jako przykład zastosowania logiki zatrzaskowej rozważmy licznik counter_ready, który zaczyna liczyć w momencie ustawienia sygnału *ready*. Wejście *ready* ma wysoki poziom tylko przez krótki czas. Dlatego w projekcie sygnał *ready* jest zatrzaskiwany jako wewnętrzny sygnał zezwolenia *enable*. Zatrzask utrzymuje sygnał *enable* w stanie wysokim aż do osiągnięcia przez licznik pełnej wartości, po czym zeruje *enable*, uniemożliwiając ponowne działanie licznika aż do następnego ustawienia sygnału wejściowego *ready*.

Przykład 11.6. Licznik z sygnałem startu liczenia *ready*

```
module counter_ready
        #(parameter N=3)
        (input clk, ready, resetN,          // sygnały sterujące
        output logic [N-1:0] Q);            // wyjście licznika

        logic enable;                       // wewnętrzny sygnał zezwolenia dla licznika
        logic overflow;                     // wewnętrzna flaga przepełnienia licznika

        always_latch begin                  // zablokowanie wejścia ready
                if (!resetN)
                        enable <= 0;
                else if (ready)
                        enable <= 1;
                else if (overflow)
                        enable <= 0;
        end
```

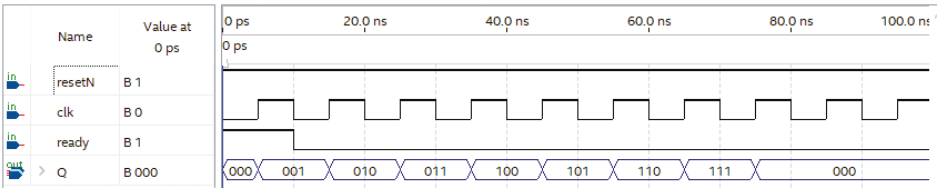
```
    always @(posedge clk, negedge resetN) begin // N-bitowy licznik
        if (!resetN)
            {overflow,Q} <= 0;
        else if (enable)
            {overflow,Q} <= Q + 1'b1;
    end
endmodule
```

Wyniki syntezy i symulacji projektu counter_ready przedstawiono odpowiednio na rys. 11.6 i 11.7.



RYS. 11.6. Widok projektu counter_ready na poziomie RTL

ŹRÓDŁO: oprac. własne.



RYS. 11.7. Wyniki symulacji projektu counter_ready

ŹRÓDŁO: oprac. własne.

Z rys. 11.6 wynika, że sygnał ready steruje (poprzez logikę kombinacyjną) zatrzaskiem *enable*, który z kolei steruje rejestrem wyjściowym licznika. Wyniki symulacji (rys. 11.7) są również w pełni zgodne z oczekiwanymi funkcjonalnościami licznika.

11.3.2.4. Procedura logiki sekwencyjnej always_ff

Specjalna procedura **always_ff** jest przeznaczona do opisu zachowania logiki sekwencyjnej. Na przykład:

```
always_ff @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;
    else q <= d;
```

Blok proceduralny **always_ff** musi mieć zdefiniowaną listę czułości, aby narzędzie projektowe mogło wydzielić sygnał zegarowy w implementowanym układzie. W bloku proceduralnym musi być tylko jeden taki sygnał zegarowy. Procedura **always_ff** wymaga również, aby każdy sygnał z listy czułości był zdefiniowany z kwalifikatorami **posedge** lub **negedge**. Jest to jednym z wymagań listy czułości przy syntezie logiki sekwencyjnej. Blok proceduralny **always_ff** zabrania też stosowania elementów sterowania w dowolnym miejscu poza początkiem bloku proceduralnego. Wartości zmiennych po lewej stronie przypisań w procedurze **always_ff** nie mogą być przypisywane przez żaden inny proces.

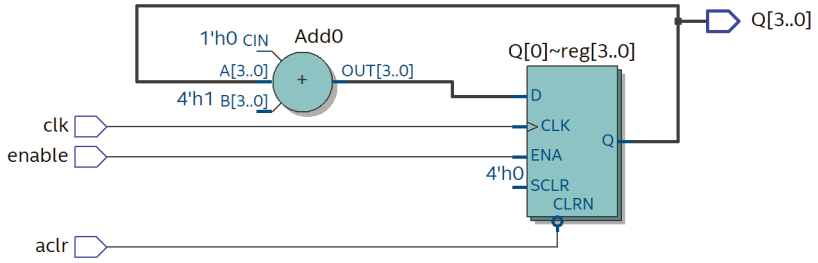
Kompilator wykonuje dodatkowe kontrole i generuje komunikat o błędzie, jeśli opis procedury **always_ff** nie reprezentuje logiki sekwencyjnej.

Jako przykład zastosowania procedury **always_ff** przyjrzyjmy się opisowi licznika binarnego.

Przykład 11.7. Licznik binarny z wejściem zezwolenia liczenia i asynchronicznym zerowaniem

```
module counter_N_bits
        #(parameter N=4)
        (input clk,aclr,enable,                    //sygnały sterujące
        output logic [N-1:0] Q);                   // wyjście licznika

        always_ff @(posedge clk, negedge aclr)
                if (!aclr)              Q <= {N{1'b0}};
                else    if (enable)     Q <= Q + 1'b1;
                else                    Q <= Q;
endmodule
```

Wyniki syntezy projektu counter_N_bits przedstawiono na rys. 11.8.



RYS. 11.8. Widok projektu licznika_N_bits na poziomie RTL

ŹRÓDŁO: oprac. własne.

11.3.3. Procedura final

Procedura **final** jest podobna do procedury **initial** poza tym, że występuje na końcu czasu symulacji i jest wykonywana bez opóźnienia. Procedura **final** służy zwykle do wyświetlania informacji statystycznych uzyskanych podczas symulacji.

Procedura **final** jest wykonywana, gdy symulacja kończy się z powodu jawnego lub ukrytego wywołania funkcji systemowej **$finish**. Na przykład:

```
final begin
        $display("Number of cycles executed %d",$time/period);
        $display("Final PC = %h",PC);
end
```

11.4. Sterowanie procesami

W języku SystemVerilog istnieje pojęcie *wątku proceduralnego* (*procedural thread*). Wątki tworzone są przez:

- każdą procedurę **initial**;
- każdą procedurę **always**, **always_comb**, **always_latch** i **always_ff**;
- każdą procedurę **final**;
- każdy operator równoległy w grupie operatorów **fork-join** (**join_any** lub **join_none**);
- każdy proces dynamiczny.

Dodatkowo każde przypisanie ciągłe **assign** można również uznać za wątek.

Język SystemVerilog dostarcza konstrukcji, które pozwalają jednemu procesowi zakończyć inne procesy lub czekać na zakończenie innych procesów.

Konstrukcja **wait fork** oczekuje na zakończenie procesu, a konstrukcja **disable** zatrzymuje wszystkie działania w obrębie nazwanego bloku lub zadania, niezależnie od relacji *rodzic–potomek* (*parent–child*). Na przykład proces potomka może zakończyć proces rodzica lub proces może zakończyć inny całkowicie niepowiązany proces. Konstrukcja **disable fork** zatrzymuje również wykonywanie procesów, ale w odniesieniu do relacji rodzic–potomek.

Operatory sterowania procesem mają następującą składnię:

```
wait (expression) statement_or_null;
wait fork;
wait_order (identifier {, identifier}) action_block;
disable task_identifier;
disable  block_identifier;
disable fork;
```

gdzie: *expression* to wyrażenie definiujące warunek oczekiwania; *statement_or_null* to operator proceduralny; *identifier* to identyfikator definiujący zdarzenie; *action_block* to wykonywany blok; *task_identifier* to nazwa zadania; *block_identifier* to nazwa nazwanego bloku. W tej składni wszystkie identyfikatory mogą mieć nazwy hierarchiczne.

Operator **wait** został omówiony w 11.1.3.

11.4.1. Operator wait fork

Operator **wait fork** blokuje proces przed wykonaniem, dopóki wszystkie jego bezpośrednie potomne procesy nie zakończą działania. Na przykład operator **wait fork** pozwala blokowi programowemu, przed wyjściem z niego, zaczekać na zakończenie wszystkich wątków współbieżnych.

W poniższym przykładzie dwa procesy potomków (*child1* i *child2*) są tworzone przed wywołaniem zadania *do_test*, które tworzy trzy kolejne procesy potomków (*child3*, *child4* i *child5*) oraz dwa procesy potomków (*decendant1* i *decendant2*). Następnie *do_sequence* tworzy dwa kolejne procesy potomków (*child6* i *child7*). Operator **wait fork** blokuje wątek *do_test*, aż wszystkie siedem procesów potomków zakończy się przed powrotem do procesu wywołującego. Zauważmy, że operator **wait fork** nie zależy bezpośrednio od potomków generowanych przez proces *child5*.

```
initial begin : test                    // blok inicjalizacji
        fork                            // uruchomienie dwóch równoległych procesów
                child1();
                child2();
        join_none
        do_test();                      // wywołanie zadania do_test
end : test

task do_test();                         // zadanie do_test
        fork
                child3();
                child4();
                fork : child5           // zagnieżdżony blok fork-join_none
                                        // jest procesem potomnym
                        descendant1();
                        descendant2();
                join_none
        join_none
        do_sequence();                  // wywołanie funkcji do_sequence
        wait fork;                      // blokowanie do zakończenia child1...child7
endtask
```

```
function void do_sequence();        // funkcja do_sequence
        fork
                child6();
                child7();
        join_none
endfunction
```

11.4.2. Operator wait_order

Operator **wait_order** wstrzymuje proces do momentu, gdy wszystkie określone zdarzenia zostaną wywołane w określonej kolejności (od lewej do prawej) lub któreś z nich zostanie nieprawidłowo wywołane, co doprowadzi do błędu.

Operator **wait_order** może zawierać konstrukcję **else**, która określa działanie, jakie należy podjąć w przypadku błędu. Na przykład:

```
bit success;

wait_order( a, b, c ) success = 1; else success = 0;
```

W tym przykładzie stan zakończenia jest przechowywany w zmiennej *success*, nie powodując błędu.

11.4.3. Operator disable

Operator **disable** umożliwia zakończenie działania związanego z równolegle aktywnymi procesami, przy zachowaniu strukturalnej natury opisów proceduralnych. Zapewnia mechanizm zakończenia zadania przed wykonaniem wszystkich jego operacji, przerwania operatora pętli lub pominięcia operatora w celu kontynuowania kolejnej iteracji operatora pętli. Jest przydatny do obsługi sytuacji specjalnych, takich jak przerwania sprzętowe i globalne zerowanie. Operator **disable** może być również użyty do zakończenia oznaczonego operatora, w tym *odroczonego zapewnienia* (*deferred assertion*) lub *proceduralnego współbieżnego zapewnienia* (*procedural concurrent assertion*).

Operator **disable** kończy zadanie lub blok, tzn. wszystkie działania wykonywane w ramach określonego bloku bądź zadania zostają zakończone. Wykonanie jest wznawiane, poczynając od operatora występującego po bloku albo po operatorze wywołania zadania. Jeśli jedno zadanie wywołuje inne zadanie, a kolejne wywołuje następne, wyłączenie zadania w łańcuchu spowoduje wyłączenie wszystkich zadań w dół łańcucha. Jeśli zadanie jest wywoływane więcej niż raz, jego wyłączenie spowoduje wyłączenie wszystkich instancji zadania.

Operator wyłączenia **disable** może być używany w blokach i zadaniach, aby wyłączyć zawierający je określony blok lub zadanie. Może być użyty do wyłączenia

nazwanych bloków wewnątrz funkcji, ale nie może być użyty do wyłączenia funkcji. W przypadkach, gdy operator **disable** wewnątrz funkcji wyłącza blok lub zadanie, zachowanie procesu wywołującego funkcję jest niezdefiniowane. Wyłączenie zadania automatycznego lub bloku w ramach zadania automatycznego jest wykonywane dla wszystkich współbieżnych instancji zadania.

Przyjrzyjmy się kilku przykładom użycia operatora **disable**.

Poniższy przykład ilustruje wyłączenie bloku.

```
begin : block_name
        a = b;
        disable block_name;
        regc = a;                       // to przypisanie nigdy nie zostanie wykonane
end
```

Poniższy przykład pokazuje, jak operator **disable** jest używany w nazwanym bloku podobnym do operatora goto w języku C. Operator wykonywany po operatorze **disable** następuje po nazwanym bloku.

```
begin : block_name
        ...
        ...
        if (a == 0)
                disable block_name;
        ...
end                             // koniec nazwanego bloku
                                // kontynuacja po nazwanym bloku
...
```

Poniższy przykład ilustruje użycie konstrukcji **disable** do zatrzymania wykonywania bloku nazwanego, który nie zawiera operatora **disable**. Jeśli blok jest aktualnie wykonywany, powoduje to przełączenie sterowania na instrukcję znajdującą się bezpośrednio po nim. Jeśli blok jest ciałem pętli, działa jak operator **continue** w języku C. Jeśli blok nie jest aktualnie wykonywany, to operator **disable** nie ma wpływu na przebieg.

```
module m (...);
        always
                begin : always1
                        ...
                        t1: task1( );           // wywołanie zadania
                        ...
                end
        ...
```

```
always
        begin
                ...
        disable m.always1;    // wyjście z bloku always1, który wyjdzie
                              // z zadania task1, jeśli jest ono aktualnie wykonywane
        end
endmodule
```

Poniższy przykład pokazuje operator **disable**, który jest używany jako wcześniejszy powrót z zadania. Język SystemVerilog ma również operator powrotu **return** z zadania, kończący wykonywanie procesu, w którym wykonywany jest powrót. Jednak zadanie, które jest wyłączane za pomocą operatora **disable**, nie zachowuje się analogiczne jak w przypadku użycia operatora **return**. Jeśli w zadaniu zostanie zastosowane polecenie **disable**, wszystkie aktywne wykonania zadania zostaną przerwane.

```
task proc_a;
        begin
                ...
                ...
                if (a == 0)
                        disable proc_a;        // powrót, jeśli prawda
                ...
                ...
        end
endtask
```

Poniższy przykład pokazuje operator **disable** używany w taki sam sposób jak operatory **continue** i **break** w języku C.

```
begin: break_for
        for (i=0; i<n; i=i+1)
                begin: cont_for
                        ...
                        if (a==0) disable cont_for;    // przechodzimy do następnej pętli
                                                       // operatora for,
                                                       // analogicznie do polecenia continue w C
                        ...
                        if (a==b) disable break_for;   // wyjście z pętli for,
                                                       // odpowiednik polecenia break w C
                        ...
                end                                    // koniec cont_for
end                                                    // koniec break_for
```

11.4.4. Operator disable fork

Operator **disable fork** kończy wszystkie aktywne procesy potomne procesu wywołującego, jak również procesy potomne procesu potomnego. Innymi słowy, jeśli którykolwiek z procesów potomnych ma potomków, operator **disable fork** zakończy je również.

W poniższym przykładzie zadanie *get_first* tworzy trzy wersje zadania oczekujące na konkretne urządzenie (1, 7 lub 13). Zadanie *wait_device* czeka na gotowość konkretnego urządzenia, a następnie zwraca jego adr. Gdy pierwsze urządzenie stanie się dostępne, zadanie *get_first* wznawia wykonywanie i przystępuje do zakończenia pozostałych procesów *wait_device*.

```
task get_first (output int adr);
        fork                        // równoległe uruchamianie trzech zadań
                wait_device (1, adr );
                wait_device (7, adr );
                wait_device (13, adr );
        join_any                    // aż jedno z zadań zostanie wykonane

        disable fork;               // zakończenie pozostałych zadań
endtask
```

Operator **disable fork** różni się od disable tym, że bierze pod uwagę dynamiczną relację procesów rodzic–potomek, podczas gdy **disable** używa statycznych informacji syntaktycznych wyłączanego bloku. Tak więc operator **disable** kończy wszystkie procesy uruchomione w danym bloku, niezależnie od tego, czy zostały one wywołane przez wątek wywołujący, czy też nie, natomiast operator **disable fork** kończy tylko procesy wywołane przez wątek wywołujący.

Operatory **disable** i **disable fork** nie są syntetyzowalne i są ignorowane podczas syntezy.

11.4.5. Precyzyjne sterowanie procesami

W języku SystemVerilog proces jest klasą wbudowaną, która pozwala jednemu procesowi na dostęp i sterowanie innego procesu po jego uruchomieniu. Użytkownicy mogą deklarować zmienne typu process i bezpiecznie przekazywać je przez zadania lub włączać do innych obiektów. Prototyp klasy *process*:

```
class process;
        typedef enum { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED } state;

        static function process self();
        function state status();
```

```
        function void kill();
        task await();
        function void suspend();
        function void resume();
        function void srandom( int seed );
        function string get_randstate();
        function void set_randstate( string state );
endclass
```

Obiekty typu *process* są tworzone automatycznie przez kompilator w zachodzących procesach. Użytkownicy nie mogą tworzyć obiektów typu *process*, mogą jednak wykorzystać funkcje klasy *process* do precyzyjnego sterowania procesami. Poniżej opisano funkcje do zarządzania procesami.

Funkcja **self()** zwraca *uchwyt* (*deskryptor*) (*handle*) bieżącego procesu, czyli uchwyt procesu, który wywołuje **self()**.

Funkcja **status()** zwraca stan procesu:

- FINISHED – proces został zakończony normalnie;
- RUNNIND – proces jest aktualnie uruchomiony (nie w operatorze blokującym);
- WAITING – proces aktualnie oczekuje w operatorze blokującym (*blocking statement*);
- SUSPENDED – proces jest obecnie zatrzymany i czeka na wznowienie;
- KILLED – proces zakończony.

Funkcja **kill()** kończy działanie procesu i wszystkich jego podprocesów, czyli procesów powstałych przez wywołanie instrukcji **fork**.

Zadanie **await()** pozwala jednemu procesowi czekać na zakończenie innego procesu.

Funkcja **suspend()** pozwala procesowi zawiesić wykonywanie własnego lub innego procesu.

Funkcja **resume()** uruchamia ponownie zawieszony wcześniej proces.

W poniższym przykładzie uruchamiana jest dowolna liczba procesów, określona przez argument *N*. Następnie zadanie czeka na rozpoczęcie ostatniego procesu, a potem na zakończenie pierwszego procesu. W tym momencie proces nadrzędny bezwarunkowo kończy wszelkie procesy, które nie zostały jeszcze zakończone.

```
task automatic do_n_way( int N );
        process job[] = new [N];        // uruchomiono N procesów

        foreach (job[j])                // definiowane są deskryptory procesów
                fork
                        automatic int k = j;
                        begin job[k] = process::self(); ... ; end
                join_none
```

```
        foreach (job[j])                // oczekuje się uruchomienia wszystkich procesów
                wait( job[j] != null );

        job[1].await();                 // oczekuje się zakończenia pierwszego procesu

        foreach (job[j]) begin          // zakończenie wszystkich procesów
                if ( job[j].status != process::FINISHED )
                job[j].kill();
        end
endtask
```

11.5. Wnioski

Proces w języku SystemVerilog to pewne ogólne pojęcie dotyczące przetwarzania danych, które odnoszą się zarówno do symulacji, jak i do syntezy. W SystemVerilog procesy są tworzone przez procedury, zadania i funkcje.

W SystemVerilog istnieją dwa rodzaje jawnego sterowania czasem w procedurach podczas wykonywania operacji: *sterowanie opóźnieniem* (*delay control*) i *sterowanie zdarzeniami* (*event control*). W tym pierwszym wyrażenie przed instrukcją proceduralną określa długość czasu pomiędzy napotkaniem operatora a momentem jego faktycznego wykonania. Sterowanie zdarzeniami pozwala zaś na opóźnienie wykonania operatora proceduralnego do momentu wystąpienia zdarzenia.

Zdarzenie jawne (*explicit event*) ma zadeklarowaną nazwę i jest inicjalizowane przez inne procedury. *Zdarzenie niejawne* (*implicit event*) to zmiana wartości sieci lub zmiennej. Może być ono zdefiniowane przez poziom sygnału (wysoki lub niski), jak również przez zbocze sygnału. W tym drugim przypadku rozróżnia się *narastające* i *opadające* zbocze sygnału.

W języku SystemVerilog istnieją trzy operatory do sterowania czasem proceduralnym: sterowanie opóźnieniami #, sterowanie zdarzeniami @ i oczekiwanie **wait**.

Operator # opóźnia wykonanie następnego operatora o wartość wyrażenia, która jest mierzona w jednostkach czasu. Operator ten może być umieszczony przed dowolnym operatorem proceduralnym, jak też na prawo od znaku równości w operatorach przypisania – wówczas opóźnienie to nazywane jest *wewnętrznym opóźnieniem przypisania* (*intra-assignment delay*). Powoduje ono opóźnione przypisanie wcześniej obliczonej nowej wartości do lewej strony.

Operator sterowania zdarzeniami (operator czułości) @ pozwala na opóźnienie wykonania kolejnego operatora (grupy operatorów), dopóki nie wystąpi co najmniej jedno ze sprawdzanych zdarzeń. Zdarzeniami tymi mogą być zmiany poziomu sygnału lub pojawienie się zbocza sygnału.

Synchronizowanie wykonywania operatorów ze zmianami w układach i wartości zmiennych nazywamy *wykrywaniem zdarzenia niejawnego* (*detecting an implicit event*).

Zdarzenie może być oparte na kierunku zmiany wartości sygnału: do wartości 1 (**posedge**) lub do wartości 0 (**negedge**).

Lista czułości (*sensitivity list*) w operatorze **@** służy do opóźnienia wykonania operatora proceduralnego w przypadku wystąpienia dowolnego zdarzenia z tej listy.

Zmienne wejściowe dla operatora lub bloku są zmiennymi, które mogą wywołać zdarzenie dla danego operatora proceduralnego bądź bloku.

Konstrukcje **@*** lub **@ (*)** mówią kompilatorowi, aby automatycznie generował niejawną listę czułości ze wszystkimi wejściami danego operatora proceduralnego. Konstrukcje te są często używane przy opisie układów kombinacyjnych. Jednak użycie procedury **always_comb** jest bezpieczniejsze niż użycie konstrukcji **@***.

Operator **wait** pozwala opóźnić wykonanie następnego operatora proceduralnego, aż wartość wyrażenia stanie się prawdziwa.

Operatory **#** i **wait** nie są syntetyzowalne i są ignorowane podczas syntezy. Tylko operator sterowania zdarzeniami **@** jest syntetyzowalny.

Sterowanie z kwalifikatorem **repeat** określa opóźnienie w ramach zadania w zależności od wystąpienia określonej liczby zdarzeń.

Bloki proceduralne pozwalają na grupowanie operatorów proceduralnych w taki sposób, że są one traktowane składniowo jako jeden operator. W języku SystemVerilog istnieją dwa rodzaje bloków proceduralnych: blok sekwencyjny **begin-end** oraz blok równoległy **fork-join**. Operatory w bloku sekwencyjnym są wykonywane sekwencyjnie w kolejności określonej w kodzie projektu. Operatory w bloku równoległym są wykonywane jednocześnie, czyli równolegle.

Blok równoległy może zablokować proces macierzysty, który go wywołał. Charakter blokowania procesu nadrzędnego przez blok równoległy określają słowa kluczowe **join**, **join_any** i **join_none**.

Wszystkie bloki równoległe typu **fork-join** nie są syntetyzowalne i są używane tylko podczas symulacji projektów.

Nazwany blok proceduralny **begin** lub **fork** tworzy nowy obszar hierarchii. Nadawanie nazw blokom umożliwia hierarchiczne odwoływanie się do zmiennych lokalnych, parametrów i zdarzeń zadeklarowanych wewnątrz nazwanego bloku, a także umożliwia odwoływanie się do bloku w operatorach wyłączających **disable**.

W SystemVerilog operatory proceduralne i bloki mogą mieć *etykiety* (*labels*).

Jeśli blok proceduralny ma etykietę, to jest ona nazwą bloku i może być podana po słowach kluczowych **end**, **join**, **join_any** lub **join_none**. Nie jest jednak dozwolone określenie zarówno etykiety, jak i nazwy bloku proceduralnego po słowach kluczowych **begin** lub **fork**.

Operatory inicjalizacyjne w pętlach **for** i **foreach** również mogą mieć etykiety. Taka etykieta nazywana jest *niejawnym blokiem tworzonym przez pętlę*.

Operator z etykietą można wyłączyć za pomocą operatora **disable**. Jest on ważny tylko przy symulacji i jest ignorowany przy syntezie.

Każda procedura strukturalna generuje proces. Do procedur strukturalnych w SystemVerilog należą: procedura **initial**, procedura **always** i jej pochodne: **always_comb**, **always_latch** i **always_ff**, procedura **final**, zadania i funkcje.

Operator **wait** i jego pochodne (**wait fork** i **wait_order**) pozwalają blokować procesy, które mogą być również zakończone za pomocą operatorów **disable** i **disable fork**.

Procedura **initial** wykonywana jest tylko raz. Procedura **always** jest wykonywana jako pętla nieskończona.

Zazwyczaj procedury **initial** i **always** są używane razem z blokami procedur **begin-end** lub **fork-join**. W tym przypadku są one określane jako *bloki* **initial** lub *bloki* **always**.

Podczas symulacji wszystkie procedury **always** i **initial** w projekcie zaczynają działać jednocześnie i równolegle.

Procedura **final** jest uruchamiana na końcu czasu symulacji i jest wykonywana tylko raz.

Zadania i funkcje są uruchamiane z jednego lub kilku miejsc w innych procedurach.

Procedura **initial** służy do inicjalizacji zmiennych i pamięci oraz do dostarczenia danych początkowych do symulacji.

Oprócz procedury ogólnego przeznaczenia **always** SystemVerilog ma wyspecjalizowane procedury **always_comb**, **always_latch** i **always_ff** służące do implementacji odpowiednio logiki kombinacyjnej, zatrzaskowej i sekwencyjnej.

Procedura ogólnego przeznaczenia **always** może być użyta do opisu dowolnego rodzaju logiki: kombinacyjnej, zatrzaskowej lub sekwencyjnej. Na bardziej abstrakcyjnych poziomach symulacji blok proceduralny **always** może być użyty do symulacji algorytmów bez wyraźnego pokazywania szczegółów implementacji.

Blok **always** może zawierać dowolną liczbę elementów sterowania czasem lub zdarzeniami (operatorów **#**, **@** i **wait**), które mogą być określone w dowolnym miejscu bloku.

Jednym z warunków rozpoznania w bloku **always** układu kombinacyjnego jest to, aby wartości wszystkich zmiennych, do których zapisuje się w bloku proceduralnym, były określone dla wszystkich możliwych wartości wejść bloku. Podczas wnioskowaniu z bloku **always** logiki zatrzaskowej warunek ten nie jest konieczny.

W liście czułości bloku **always** dla logiki sekwencyjnej narzędzia syntezy przydzielają jeden sygnał, który nie bierze udziału w sprawdzeniach wewnątrz bloku proceduralnego. Przyjmuje się, że ten sygnał jest sygnałem zegarowym i jest podłączony do wejścia zegarowego przerzutnika lub rejestru podczas realizacji sprzętowej.

W przypadku zastosowania procedury **always_comb** narzędzia syntezy automatycznie sprawdzają warunki implementacji logiki kombinacyjnej.

Dla procedury **always_latch** dopuszczalna jest sytuacja, gdy nie wszystkie możliwe kombinacje zmiennych wejściowych są przypisane do zmiennych wyjściowych bloku.

Przy zastosowaniu procedury **always_ff** narzędzia syntezy automatycznie sprawdzają warunki realizacji logiki sekwencyjnej.

Procedura **final** jest wywoływana na końcu czasu symulacji i jest wykonywana bez opóźnienia. Zazwyczaj służy ona do wyświetlania informacji statystycznych dotyczących symulacji.

W języku SystemVerilog występuje pojęcie *wątku proceduralnego* (*procedural thread*). *Wątki* (*threads*) tworzą procedury, operatory równoległe w **fork-join**, procesy dynamiczne i przypisania ciągłe. Sterowanie wątkami odbywa się poprzez sterowanie procesami.

Język SystemVerilog udostępnia następujące mechanizmy sterowania wątkami:

- operatory wyboru (**if-else**, **case**), operatory pętli i operatory przejść;
- wywołania podprogramów (zadań i funkcji);
- bloki sekwencyjne (**begin-end**) i równoległe (**fork-join**);
- proceduralne operatory sterowania czasem **(#**, **@**, **wait**);
- operatory sterowania procesami (**wait fork**, **wait_order**, **disable**, **disable fork**).

Operator **wait** blokuje wykonanie operatora proceduralnego do momentu, aż wyrażenie stanie się prawdziwe.

Operator **wait fork** blokuje wykonanie procesu do czasu, aż wszystkie procesy potomne zakończą swoje działanie.

Operator **wait_order** blokuje proces do momentu, gdy wszystkie określone zdarzenia zostaną zainicjowane w określonej kolejności (od lewej do prawej) lub któreś z nich zostanie zainicjowane nieprawidłowo, co spowoduje błąd. Operator **wait_order** może zawierać konstrukcję **else**, która określa działanie, jakie należy podjąć w przypadku błędu.

Operator **disable** zapewnia możliwość zakończenia zadania przed wykonaniem wszystkich jego operatorów, przerwania operatora pętli lub pominięcia operatora w celu kontynuowania pracy z kolejną iteracją operatora pętli, przerwania wykonywania nazwanego bloku lub operatora oznaczonego. Operator **disable** nie może być użyty do wyłączenia funkcji.

Operator **disable fork** kończy wszystkie aktywne procesy potomne procesu wywołującego, jak również procesy potomne jego potomków.

Operatory **disable** i **disable fork** nie są syntetyzowalne i są ignorowane podczas syntezy.

W języku SystemVerilog proces jest klasą wbudowaną, która pozwala jednemu procesowi na dostęp i kontrolę innego procesu po jego uruchomieniu. Użytkownicy mogą deklarować zmienne typu *process* i bezpiecznie przekazywać je przez zadania lub dołączać do innych obiektów.

12. Operatory proceduralne

W SystemVerilog operatory proceduralne mogą być umieszczane w blokach proceduralnych, zadaniach i funkcjach. Operatory proceduralne służą do opisu projektu w sposób *behawioralny* lub *algorytmiczny*, jak w językach programowania.

Operatorami proceduralnymi w SystemVerilog są:

- operatory wyboru (**if**, **case**);
- operatory pętli (**for**, **repeat**, **foreach**, **while**, **do-while**, **forever**);
- operatory przejścia (**break**, **continue**, **return**);
- bloki sekwencyjne (**begin-end**) i równoległe (**fork-join**);
- operatory kontroli czasu (sterowania czasem) (**#**, **@**, **wait**);
- operatory sterowania procesami (**wait**, **disable**);
- przypisania proceduralne (=, <=, **assign**, **force**, szablony przypisania);
- wywołania zadań i funkcji.

Głównymi operatorami proceduralnymi przy opisie algorytmów są operator warunkowy **if**, operator wyboru **case** oraz operatory pętli. Przy opisie schematów kombinacyjnych często zapisuje się sekwencyjnie kilka operatorów **if** jako łańcuch **if-else-if**. Problem polega na określeniu, w jakiej kolejności sprawdzać wyrażenia logiczne każdego operatora **if**: w kolejności zapisu, w kolejności arbitralnej czy równolegle.

W przypadku sprawdzania wyrażeń logicznych w kolejności zapisu implementacja sprzętowa wymaga wprowadzenia do układu kodera priorytetowego, co zwiększa koszt implementacji i opóźnienie sygnałów. Arbitralna kolejność sprawdzania wyrażeń logicznych pozwala narzędziu projektowemu na optymalizację obwodu poprzez zmianę tej kolejności. Równoległe sprawdzanie wyrażeń logicznych daje jeszcze więcej możliwości optymalizacji układu.

Język SystemVerilog dostarcza kwalifikatorów **unique** (**unique0**) i **priority** do określenia kolejności sprawdzania wyrażeń logicznych w łańcuchach **if-else-if**. Kwalifikator **unique** mówi narzędziu projektowemu, że każde wyrażenie logiczne jest unikatowe i że suma wszystkich wyrażeń logicznych jest kompletna. Dzięki temu narzędzie programowe może równolegle sprawdzać wszystkie warunki logiczne i realizować układ kombinacyjny bez zatrzasków. Natomiast kwalifikator priorytetu **priority** mówi programowi narzędziowemu, aby sprawdzał wyrażenia logiczne w określonej kolejności, zgodnie z zapisem w kodzie źródłowym, oraz aby realizował układ kombinacyjny również bez zatrzasków. W przypadku naruszenia warunków opisu

układu kombinacyjnego zarówno przy zastosowaniu kwalifikatora **unique**, jak i kwalifikatora **priority** narzędzie programowe generuje komunikat o błędzie, co pozwala na wykrycie nieprawidłowości w opisie na wczesnych etapach projektowania.

Podobne problemy pojawiają się też przy opisie układów kombinacyjnych z wykorzystaniem operatorów **case**. Język SystemVerilog umożliwia stosowanie kwalifikatorów **unique** (**unique0**) i **priority** także do operatorów **case**.

Przy porównywaniu wyrażeń jeśli pojedynczy bit wyrażenia jest równy z lub x, to wartość wyrażenia uważa się za nieznaną (wartość x). Jednak w pewnych przypadkach konieczne jest porównywanie wyrażeń z wartościami z i x bit po bicie. W tym celu w języku SystemVerilog istnieją operatory **casez** i **casex**, tak jak w Verilogu.

Oprócz kwalifikatorów **unique** i **priority** w konstrukcjach **case** można stosować atrybuty *full_case* i *parallel_case*. Ich zastosowanie pokrywa się z ich działaniem w Verilogu: atrybut *full_case* pozwala nie ustawiać zatrzasków na wyjściach schematu kombinacyjnego, a *parallel_case* na równoległą realizację wszystkich gałęzi operatora wyboru **case**.

Nowością w języku SystemVerilog jest również kwalifikator **inside**, który pozwala na wyszukiwanie elementów w zakresie.

Język Verilog ma dość ograniczone możliwości operatorów pętli. SystemVerilog rozwiązuje ten problem, znacznie rozszerzając możliwości operatora **for** oraz dodając dwa nowe operatory: podobny do istniejącego w języku C operator **do-while** oraz operator **foreach** do pracy z elementami tablic.

Innym problemem związanym z językiem Verilog jest brak operatorów podobnych do tych z języka C, służących do kończenia iteracji pętli **continue** i kończenia pętli **break**. Język SystemVerilog dodaje te operatory, a także operator **return** do wychodzenia z zadań i funkcji.

12.1. Operator warunkowy if-else

Operator **if-else** pozwala na wykonanie pewnych fragmentów kodu, gdy spełnione są określone warunki. Format operatora **if-else**:

if (*expression*) *statement1*
[**else** *statement2*]

gdzie: **if** i **else** to słowa kluczowe; *expression* to wyrażenie, które ma być sprawdzone; *statement1* i *statement2* to pewne operatory. Klauzula **else**, po której następuje *statement2*, jest opcjonalna. Zamiast operatorów *statement1* i *statement2* można użyć bloku proceduralnego **begin-end** (czyli grupy operatorów) oraz innego operatora **if-else**.

Działanie operatora **if-else** jest następujące. Obliczane jest wyrażenie *expression*, jeśli ma ono wartość niezerową, to wykonywany jest operator *statement1*; jeśli

wyrażenie *expression* ma wartość zerową, wartość x lub z, to wykonywany jest operator *statement2* (w obecności konstrukcji **else** *statement2*).

Ponieważ wyrażenie *expression* jest sprawdzane pod kątem wartości zerowej, następujące dwa fragmenty są równoważne:

if (*expression*) oraz **if** (*expression* != 0).

Jeśli użyjemy operatora **if-else** zamiast operatora *statement1* lub *statement2*, może powstać wątpliwość co do tego, które **if** odnosi się do słowa kluczowego **else**. Zasadą jest tutaj, że **else** zawsze odnosi się do najbliższego poprzedzającego go **if**. W kodzie można to wyrazić za pomocą wcięć. Na przykład:

```
if (index > 0)
        if (a > b)
                result = a;
        else                    // else odnosi się do poprzedniego if
                result = b;
```

Nawiasy proceduralne **begin-end** są używane do wymuszenia prawidłowej relacji między **if** i **else**. Na przykład:

```
if (index > 0)
begin
        if (a > b)
                result = a;
        else
                result = b;
end
```

12.1.1. Konstrukcja if-else-if

Gdy w formacie operatora **if-else** zamiast operatora *statement2* używany jest inny operator **if-else**, mówimy o łańcuchu operatorów **if-else**, nazywając go *konstrukcją **if-else-if***. Łączy ją wiele operatorów **if**. Na przykład:

```
if (expression1)        statement1;
else if (expression2)   statement2;
else if (expression3)   statement3;
else                    statement4;
```

Wyrażenia w konstrukcji **if-else-if** są obliczane w takiej kolejności, jak je zapisano w kodzie źródłowym. Jeśli wyrażenie jest prawdziwe, to zostanie wykonane odpowiednie polecenie i łańcuch zostanie zakończony. Zauważmy, że każdy operator

w konstrukcji **if-else-if** może być oddzielnym operatorem lub blokiem operatorów **begin-end**.

Poniżej przedstawiono przykłady użycia operatora **if-else**.

Przykład 12.1. Jednostka arytmetyczna realizująca dodawanie lub mnożenie

```
module adder_mult
        #(parameter N=8)            // N jest szerokością słów
        (input [N-1:0] a, b,        // słowa wejściowe
input add_sub,                      // wejście sterujące
        output logic [N-1:0] y);    // wynik

        always @(*)
        begin
                if (add_sub ==0)    y = a + b;
                else                y = a * b;
        end
endmodule
```

Przykład 12.2. Parametryzowany moduł multipleksera dwuwejściowego

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

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

Ogólnie rzecz biorąc, operator **if** może być zastąpiony operacją warunkową, jak w poniższym przykładzie.

Przykład 12.3. Użycie operacji warunkowej zamiast operatora **if**

```
module mux_N_conditional
        #(parameter N=8)            // N jest szerokością 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
```

Ostatni operator w łańcuchu **if-else-if** (po słowie kluczowym **else**) odpowiada przypadkowi, gdy żadne z wyrażeń nie jest prawdziwe, co odpowiada pewnemu domyślnemu działaniu. Zauważmy, że w łańcuchu **if-else-if** może brakować ostatniego operatora (po słowie kluczowym **else**). Na przykład:

Przykład 12.4. Użycie łańcucha **if-else-if** przy opisie prostego ALU

```
module alu_if_else_if
    #(parameter N=4)
    (input logic [N-1:0] a,b,          // operandy
    input [2:0] op,                    // kod operacji
    output logic [N-1:0] y);           // wynik

    always @(*)
        if (op==3'b000) y = a + b;
        else if (op==3'b001) y = a - b;
        else if (op==3'b010) y = a & b;
        else if (op==3'b011) y = a | b;
        else if (op==3'b100) y = a ^ b;
endmodule
```

Realizując przykład 12.4, syntezator automatycznie umieści zatrzaski na wyjściach układu, ponieważ wartość zmiennej wyjściowej *y* jest niezdefiniowana dla trzech kombinacji wejść *op*.

Ostatni operator w konstrukcji **if-else-if** może być wykorzystany do wykrywania błędów, np. przypisując wartość x do zmiennej we wszystkich bitach.

Przykład 12.5. Użycie ostatniego **else** do wykrywania błędów

```
module alu_if_else_if_else
    #(parameter N=4)
    (input logic [N-1:0] a,b,
    input [2:0] op,
    output logic [N-1:0] y);

    always @(*)
        if (op==3'b000) y = a + b;
        else if (op==3'b001) y = a - b;
        else if (op==3'b010) y = a & b;
        else if (op==3'b011) y = a | b;
        else if (op==3'b100) y = a ^ b;
```

```
        else y = {N{1'bx}};
endmodule
```

Przykład 12.5 jest realizowany jako układ kombinacyjny, zgodnie z zamierzeniami projektanta.

Wniosek: Aby zaimplementować układ kombinacyjny, w łańcuchu operatorów **if-else-if** każda instrukcja **if** musi mieć swoją instrukcję **else**.

12.1.2. Kwalifikatory unique, unique0 i priority

W SystemVerilog kwalifikatory **unique**, **unique0** i **priority** mogą być używane przed instrukcją **if-else**.

Podczas realizacji operatora **if-else-if** wartości wyrażeń są sprawdzane sekwencyjnie, zgodnie z tym, co napisano w kodzie. Jeżeli jedno z wyrażeń jest prawdziwe, to zostaje wykonany odpowiedni operator, kolejne warunki nie są sprawdzane i łańcuch operatora **if-else** zostaje zakończony. Aby zaimplementować takie zachowanie w sprzęcie, narzędzie syntezy musi wprowadzić do obwodu koder priorytetowy, co wymaga pewnych kosztów sprzętowych.

Jeśli wiadomo, że każdy warunek jest unikatowy i nigdy dwa wyrażenia nie mogą być prawdziwe w tym samym czasie, sprawdzanie wyrażeń w konstrukcji **if-else-if** może być wykonane równolegle bez użycia kodera priorytetowego. Pozwala to na zmniejszenie kosztów sprzętowych i zwiększenie wydajności sprzętowej implementacji powyższej konstrukcji.

Kwalifikator **unique** przed słowem kluczowym **if** wskazuje kompilatorowi, że wyrażenia w konstrukcji **if-else-if** są unikatowe i dwa wyrażenia nigdy nie mogą być jednocześnie prawdziwe. Układ z kwalifikatorem **unique** jest implementowany jako logika kombinacyjna bez zatrzasków.

Jeśli kwalifikator **unique** jest określony w konstrukcji **if-else-if**, kompilator sprawdza wszystkie wyrażenia pod kątem unikalności. Jeśli ten warunek zostanie naruszony, pojawi się komunikat o błędzie. Na przykład zostanie wydane ostrzeżenie dla następującego fragmentu kodu, w którym nakładają się warunki.

```
logic [2:0] sel;

always_comb begin
        unique if (sel[0])          mux_out = a;
                else if (sel[1]) mux_out = b;
                else if (sel[2])          mux_out = c;
end
```

Tak więc użycie kwalifikatora **unique** w konstrukcji **if-else-if** może zwiększyć niezawodność projektu, a także zmniejszyć koszt realizacji i zwiększyć szybkość projektu.

Przykład 12.6. Użycie kwalifikatora **unique** w projekcie ALU

```
module alu_if_else_if_unique
        #(parameter N=4)
        (input logic [N-1:0] a,b,
        input [2:0] op,
        output logic [N-1:0] y);

        always @(*)
                unique if (op==3'b000) y = a + b;
                else if (op==3'b001) y = a - b;
                else if (op==3'b010) y = a & b;
                else if (op==3'b011) y = a | b;
                else if (op==3'b100) y = a ^ b;
endmodule
```

Podczas używania kwalifikatora **unique0** w konstrukcji **if-else-if** kompilator nie generuje komunikatu o błędzie w przypadku naruszenia warunku unikalności wyrażeń (kwalifikator **unique0** nie jest podtrzymywany w Quartusie).

Kwalifikator priorytetu **priority** w konstrukcji **if-else-if** informuje narzędzie projektowe, że wyrażenia powinny być oceniane w kolejności określonej w kodzie. Kwalifikator priorytetu zapewnia, że zachowanie narzędzi programowych jest spójne, tzn. że modele do syntezy i do symulacji nie różnią się od siebie.

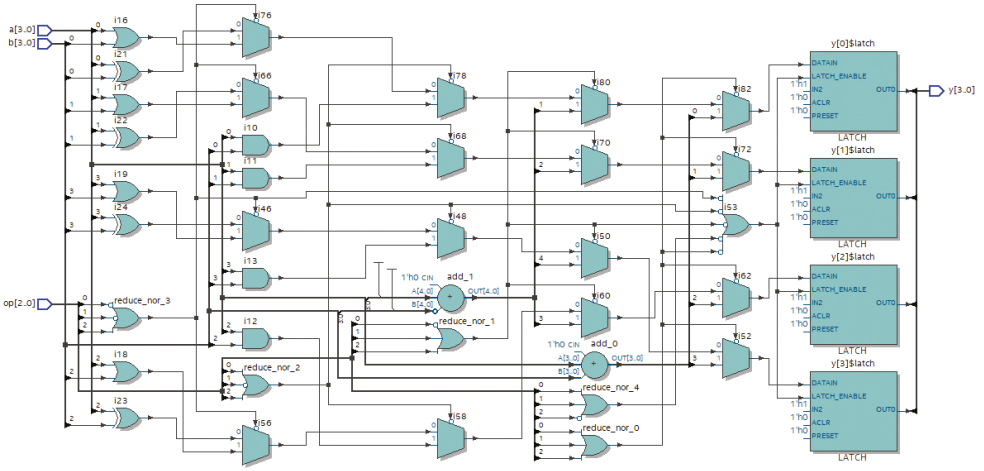
Gdy używamy kwalifikatora priorytetu **priority** i żadne z wyrażeń w konstrukcji **if-else-if** nie jest prawdziwe, kompilator wygeneruje komunikat ostrzegawczy, ale logika kombinacyjna bez zatrzasków zostanie zaimplementowana.

Przykład 12.7. Użycie kwalifikatora **priority** w projekcie ALU

```
module alu_if_else_if_priority
        #(parameter N=4)
        (input logic [N-1:0] a,b,
        input [2:0] op,
        output logic [N-1:0] y);

        always @(*)
                priority if (op==3'b000) y = a + b;
                else if (op==3'b001) y = a - b;
                else if (op==3'b010) y = a & b;
                else if (op==3'b011) y = a | b;
                else if (op==3'b100) y = a ^ b;
endmodule
```

12.1.3. Cechy charakterystyczne syntezy kwalifikatorów unique, unique0 i priority w konstrukcji if-else-if

Przyjrzyjmy się wykorzystaniu przy syntezie kwalifikatorów **unique**, **unique0** i **priority**. Wróćmy do konstrukcji prostego ALU alu_if_else_if podanej w przykładzie 12.4. W łańcuchu operatora **if-else-if** brakuje ostatniego słowa kluczowego **else**, które zapewnia działanie domyślne, tj. gdy żaden z warunków nie jest spełniony. Naturalne jest założenie, że kompilator ustawi zatrzaski na wyjściu schematu kombinacyjnego. Wynik syntezy projektu alu_if_else_if przedstawiono na rys. 12.1.



RYS. 12.1. Implementacja ALU bez ostatniego słowa kluczowego **else**

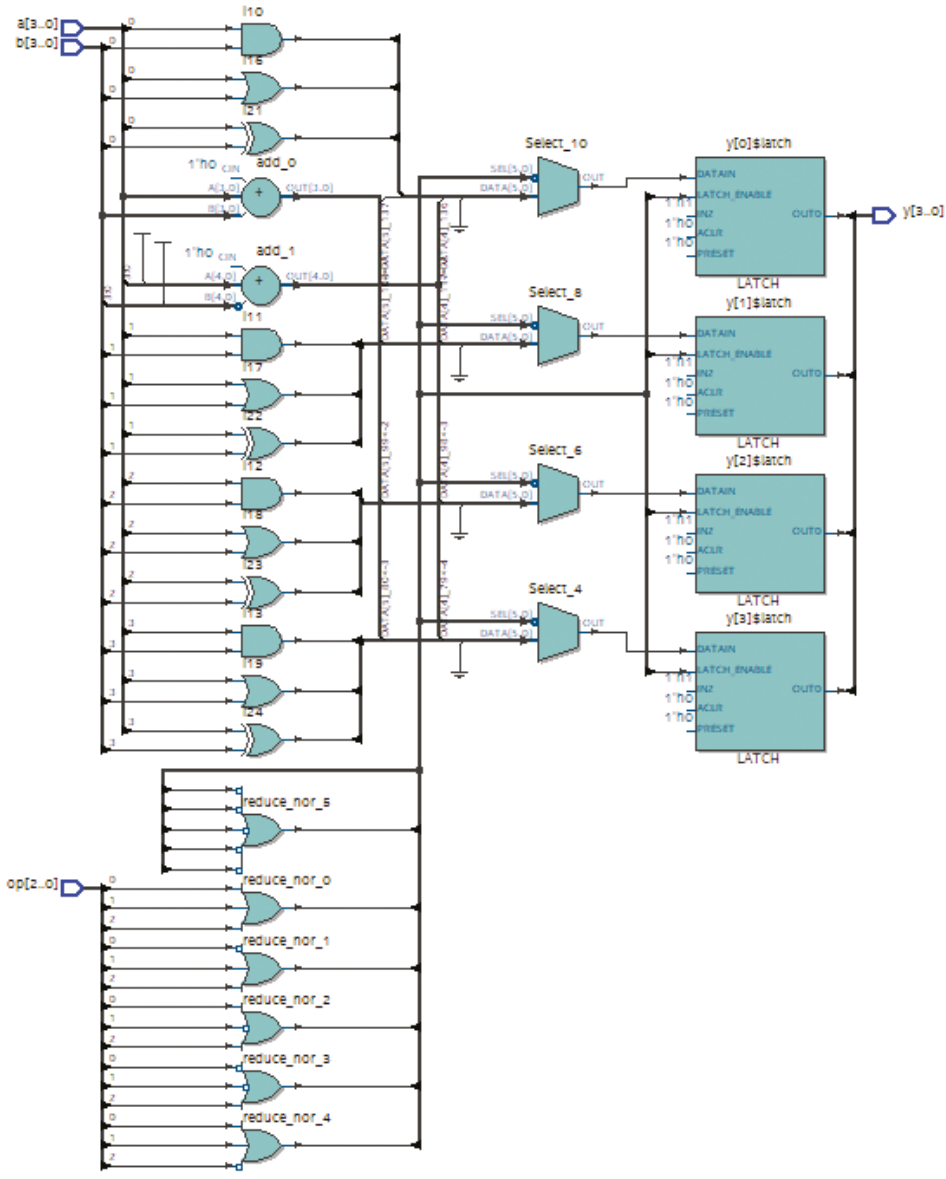
ŹRÓDŁO: oprac. własne.

Zastosowanie kwalifikatora **unique** dla naszego przykładu pozwala nam na równoległe wykonywanie operacji ALU. Wynik syntezy projektu alu_if_else_if_unique z przykładu 12.6 przedstawiono na rys. 12.2.

Część kombinacyjna schematu z rys. 12.2 ma krótszą ścieżkę krytyczną w porównaniu z rys. 12.1. Można więc założyć, że schemat z rys. 12.2 będzie miał większą wydajność.

Wynik syntezy projektu alu_if_else_if_priority z przykładu 12.7 jest dokładnie taki sam jak obwód z rys. 12.1, na którym widać, że syntezator umieścił zatrzaski na wyjściu obwodu. Tak więc w systemie Quartus kwalifikator priorytetu **priority** nie pozwala nam uniknąć umieszczania zatrzasków na wyjściu obwodu kombinacyjnego.

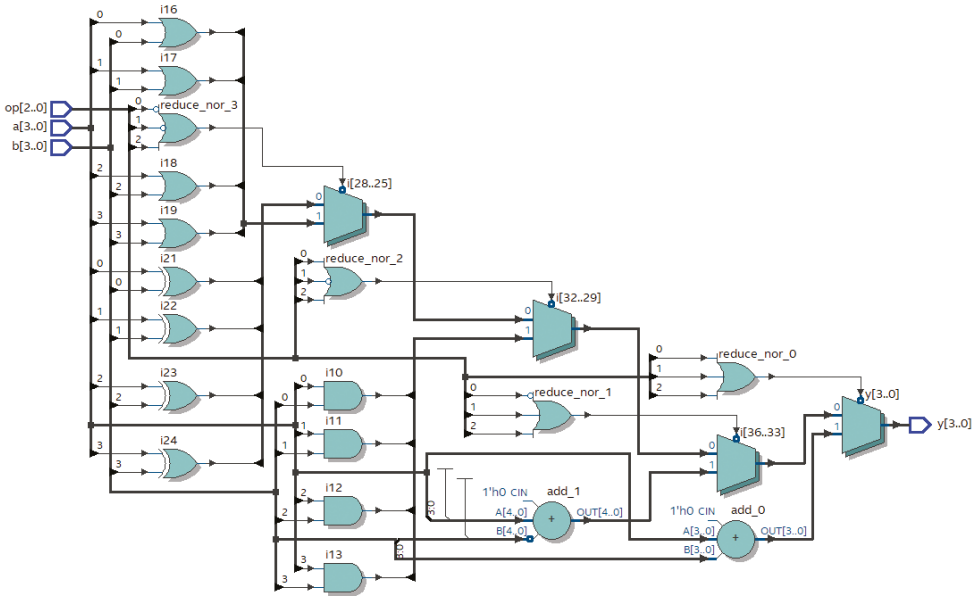
W praktyce wartość domyślna w łańcuchach operatora **if-else-if** po słowie kluczowym **else** jest często wykorzystywana do sygnalizowania błędu, np. przez przypisanie nieznanej wartości na wszystkich bitach, jak w projekcie alu_if_else_if_else z przykładu 12.5. Wynik syntezy projektu alu_if_else_if_else pokazano na rys. 12.3.



RYS. 12.2. Wynik zastosowania kwalifikatora **unique** w projekcie ALU

ŹRÓDŁO: oprac. własne.

Zauważmy, że układ z rys. 12.3 nie ma zatrzasków wyjściowych i jest znacznie prostszy od układu z rys. 12.1. Zauważmy też, że koszt implementacji (liczba użytych elementów logicznych FPGA) projektu alu_if_else jest mniejszy w porównaniu z innymi projektami ALU z naszego przykładu.



RYS. 12.3. Realizacja projektu alu_if_else_if_else z wykorzystaniem wartości domyślnej do wskazania błędu

ŹRÓDŁO: oprac. własne.

Wniosek: kwalifikatory **unique**, **unique0** i **priority** są potężnymi narzędziami w języku SystemVerilog do projektowania układów kombinacyjnych. Jednak nie wszystkie narzędzia projektowe je obsługują.

12.2. Operator case

Operator **case** pozwala na wykonanie różnych fragmentów kodu w zależności od wartości wyliczonego wyrażenia. Format instrukcji case:

case (*expression*)
item_expression,...,*item_expression* : *statement*;
...
item_expression,...,*item_expression* : *statement*;
[**default** : *statement*;]
endcase

gdzie: **case**, **default** i **endcase** są słowami kluczowymi; *expression* jest wyrażeniem operatora **case**; *item_expression* jest wyrażeniem elementu operatora **case**; *statement* jest pewnym operatorem, może być też blokiem **begin-end**. Konstrukcja **default** jest opcjonalna.

Działanie operatora **case** jest następujące. Najpierw obliczane jest wyrażenie *expression*. Otrzymana wartość jest kolejno porównywana z każdym wyrażeniem elementów instrukcji. Jeśli istnieje dopasowanie, to wykonywany jest operator, który występuje bezpośrednio po podanym wyrażeniu *item_expression*, a następnie sekwencyjne poszukiwanie jest zakończone. Jeśli żadna z wartości wyrażenia *item_expression* nie pasuje do wartości wyrażenia *expression*, to wykonywany jest operator następujący po słowie kluczowym **default**. Jeśli nie ma konstrukcji **default** i żadne z porównań nie pasuje, to nie jest wykonywany żaden z operatorów *statement*. Wartość *item_expression* jest również nazywana *elementem stałym* (*constant element*).

Zauważmy, że pojedynczy operator *statement* może być poprzedzony wieloma elementami *item_expression*. Wyrażenie *expression* jest dopasowane do *item_expression* tylko wtedy, gdy każdy bit jest dokładnie równy 0, 1, x lub z. Długość wyrażenia *expression* i wszystkich wyrażeń *item_expression* musi być równa. Ostatnia cecha odróżnia operator **case** od konstrukcji **if-else-if**, gdzie wyrażenia mogą być bardziej ogólne.

Rozważmy możliwości operatorów **case** i **if-else** podczas realizacji funkcji boolowskiej, opisanej następującym wyrażeniem:

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

Przykład 12.8. Opis funkcji boolowskiej *y* za pomocą operatora **if-else**

```
module sm_IF
(input a,b,c,
        output logic y);

wire [2:0] w={a,b,c};   // wektor pomocniczy łączący sygnały wejściowe
        always @(*) begin
                if (w==3`d0 || w==3`d5 || w==3`d6) y = 1`b0;
                else y = 1`b1;
        end
endmodule
```

Przyjrzyjmy się teraz różnym sposobom opisu tej samej funkcji *y* za pomocą operatora **case**.

Przykład 12.9. Opis funkcji *y* za pomocą operatora **case** zgodnie z tablicą prawdy

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

```
        always @(*)
          case ({a, b, c})                  // użycie operacji konkatenacji
                3'b000: y = 1'b0;           // wszystkie możliwe kombinacje wejść
                                            // i odpowiadające im wartości wyjściowe
                3'b001: y = 1'b1;
                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
```

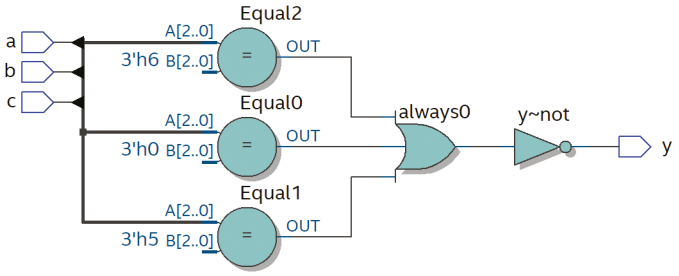
Przykład 12.10. Opis funkcji *y* przy użyciu konstrukcji **default**

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

        always @(*)
          case ({a, b, c})
                3'b000: y = 1'b0;           // analizowane są tylko zerowe wartości wyjściowe
                3'b101: y = 1'b0;
                3'b110: y = 1'b0;
                default: y = 1'b1;          // wartość wyjściowa 1 jest traktowana jako domyślna
          endcase
endmodule
```

Przykład 12.11. Wariant poprzedniego przykładu, gdy jeden operator poprzedzony jest kilkoma wyrażeniami elementów operatora **case**

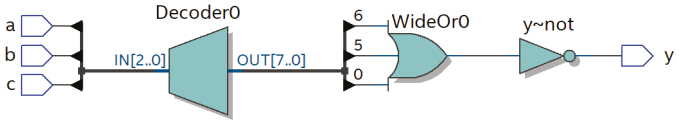
```
module sm_Case3
(input a,b,c,
        output logic y);

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

Wyniki syntezy projektu sm_IF z przykładu 12.8 z wykorzystaniem operatora **if--else** przedstawiono na rys. 12.4. Wyniki syntezy wszystkich projektów z przykładów 12.9–12.11 z wykorzystaniem operatora **case** są takie same (rys. 12.5).



RYS. 12.4. Implementacja funkcji boolowskiej z wykorzystaniem operatora **if-else**

ŹRÓDŁO: oprac. własne.



RYS. 12.5. Implementacja funkcji boolowskiej przy użyciu operatora **case**

ŹRÓDŁO: oprac. własne.

Elementy stałe operatora case mogą używać wartości x i z, ale są one ignorowane podczas syntezy. Do porównywania wyrażeń, których wartości zawierają x i z, język SystemVerilog dostarcza odpowiednio operatorów **casex** i **casez**.

12.2.1. Operatory casez i casex

Format instrukcji **casez** i **casex** jest dokładnie taki sam jak format operatora **case**, z tym że zamiast słowa kluczowego **case** używa się odpowiednio słów kluczowych **casez** i **casex**. Operatory **casez** i **casex** kończą się tym samym słowem kluczowym **endcase**.

Operator **casez** umożliwia wykorzystanie wartości logicznej z do reprezentowania *niezdefiniowanych* (*don't care*) wartości bitowych w obliczonym wyrażeniu, jak również w elementach stałych. Operator **casex** jest podobny do operatora **casez**, ale pozwala na użycie dwóch wartości logicznych z i x do oznaczenia niezdefiniowanych wartości bitowych. Dodatkowo elementy stałe operatorów **casez** i **casex** mogą używać znaku zapytania (?), aby wskazać niezdefiniowane lub nieznane wartości.

Jeśli jedynie wyjście funkcji boolowskiej może mieć wartość niezdefiniowaną, to taką funkcję można opisać za pomocą zwykłego operatora **case**.

Przykład 12.12. Użycie niezdefiniowanych wartości wyjść w operatorze **case**

```
module sm_Case4 (input a,b,c,
                 output logic f);

      always @(*)
        case ({a, b, c})                  // operator casez lub casex nie jest tu wymagany
              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 mieć niezdefiniowaną wartość
         endcase
endmodule
```

Jeśli jednak niezdefiniowana wartość występuje w elementach stałych, wówczas należy użyć operatora **casex**.

Przykład 12.13. Użycie niezdefiniowanych wartości wejść w operatorze **casex**

```
module decoder       (input [7:0] in,
output [1:0] logic y);
      always begin
            casex (in)       // wektor wejściowy, którego wartość jest sprawdzana
                  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
```

Inny przykład opisu funkcji logicznej, w którym znak zapytania jest używany do oznaczenia nieznanej wartości, pokazano poniżej.

Przykład 12.14. Użycie operatora **casex** do implementacji układu kombinacyjnego

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

12.2.2. Wyrażenia stałe w operatorach case

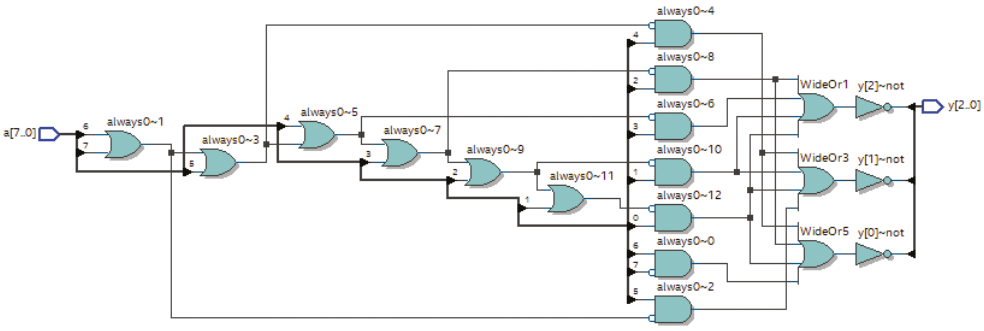
Wyrażenia stałe mogą być używane w operatorach **case** zamiast *expression* i *item_expression*.

Poniższy przykład opisuje koder priorytetowy. Stała 1 jest używana jako wyrażenie *expression*, a elementy stałe są odpowiednimi bitami słowa wejściowego *a*.

Przykład 12.15. Koder priorytetowy z 8 wejściami i 3 wyjściami

```
module pr_encoder_8_3
        (input [7 : 0] a,
         output logic [2 : 0] y);
         always_comb
                case (1)
                        a[7] : y = 7;
                        a[6] : y = 6;
                        a[5] : y = 5;
                        a[4] : y = 4;
                        a[3] : y = 3;
                        a[2] : y = 2;
                        a[1] : y = 1;
                        a[0] : y = 0;
                        default : y = 3'bx;
                endcase
endmodule
```

Wyniki syntezy projektu pr_encoder_8_3 przedstawiono na rys. 12.6.



RYS. 12.6. Wyniki syntezy projektu pr_encoder_8_3

ŹRÓDŁO: oprac. własne.

12.2.3. Kwalifikatory unique, unique0 i priority w operatorach case

Kwalifikatory **unique**, **unique0** i **priority** mogą być zapisane przed operatorami **case**, jak również przed operatorami **if-else-if**.

W operatorze **case**, jeśli wartości wyrażenia *expression* i pewnego stałego elementu *item_expression* są zbieżne, wykonywany jest odpowiedni operator, kolejne elementy *item_expression* nie są sprawdzane, a operator **case** jest zakończony. Do sprzętowej realizacji takiego zachowania służy koder priorytetowy.

Kwalifikator **unique** mówi kompilatorowi, że wyrażenia *item_expression* mogą być obliczane i porównywane w dowolnej kolejności, ponieważ są unikatowe. Kwalifikator **unique** mówi również kompilatorowi, że zestaw wyrażeń *item_expression* jest kompletny, czyli że wartość wyrażenia odpowiada jednej i tylko jednej wartości *item_expression*. Pozwala to nie wprowadzać do schematu kodera priorytetowego, po to aby zaimplementować operator **case** i cały układ w postaci logiki kombinacyjnej.

Dodatkowo gdy użyto kwalifikatora **unique**, kompilator sprawdza, czy wszystkie wyrażenia *item_expression* są unikatowe i się nie pokrywają. Jeśli te warunki zostaną naruszone, kompilator generuje komunikat o błędzie.

Kwalifikator **unique** operatora **case** jest często używany w połączeniu z procedurą **always_comb** do implementacji logiki kombinacyjnej. Na przykład:

```
always_comb
    unique case (opcode)
        2'b00: y = a + b;
        2'b01: y = a - b;
        2'b10: y = a * b;
        2'b11: y = a / b;
    endcase
```

Kwalifikator **unique0** jest podobny do kwalifikatora **unique**, z tym że kompilator nie generuje komunikatu o błędzie w przypadku naruszenia warunku unikalności dla wyrażeń *item_expression*.

Kwalifikator priorytetu **priority** w konstrukcji **case** mówi kompilatorowi, że:

- co najmniej jeden element wyrażenia *item_expression* powinien odpowiadać wyrażeniu *expression*;
- jeżeli więcej niż jeden *item_expression* pasuje do wyrażenia *expression*, to zostanie wzięta pierwsza pasująca gałąź.

Jeśli powyższe warunki nie są spełnione, kompilator generuje komunikat o błędzie.

Innymi słowy, używając kwalifikatora priorytetu **priority**, projektant mówi kompilatorowi, że wyrażenie *expression* może pasować do dwóch lub więcej wyrażeń *item_expression*, i kolejność wyboru elementów **case** jest ważna. Podczas korzystania z kwalifikatora priorytetu **priority** logika kombinacyjna jest implementowana bez zatrzasków wyjściowych, ponieważ co najmniej jeden element wyrażenia *item_expression* musi pasować do wyrażenia *expression*.

Poniższy przykład określa priorytet, w jakim dekodowane są żądania przerwania *irq0* do *irq3*, przy czym *irq0* ma najwyższy priorytet.

```
always_comb
    priority case (1'b1)
        irq0: irq = 4'b0001;
        irq1: irq = 4'b0010;
        irq2: irq = 4'b0100;
        irq3: irq = 4'b1000;
    endcase
```

12.2.4. Właściwości syntezy kwalifikatorów unique, unique0 i priority w operatorze case

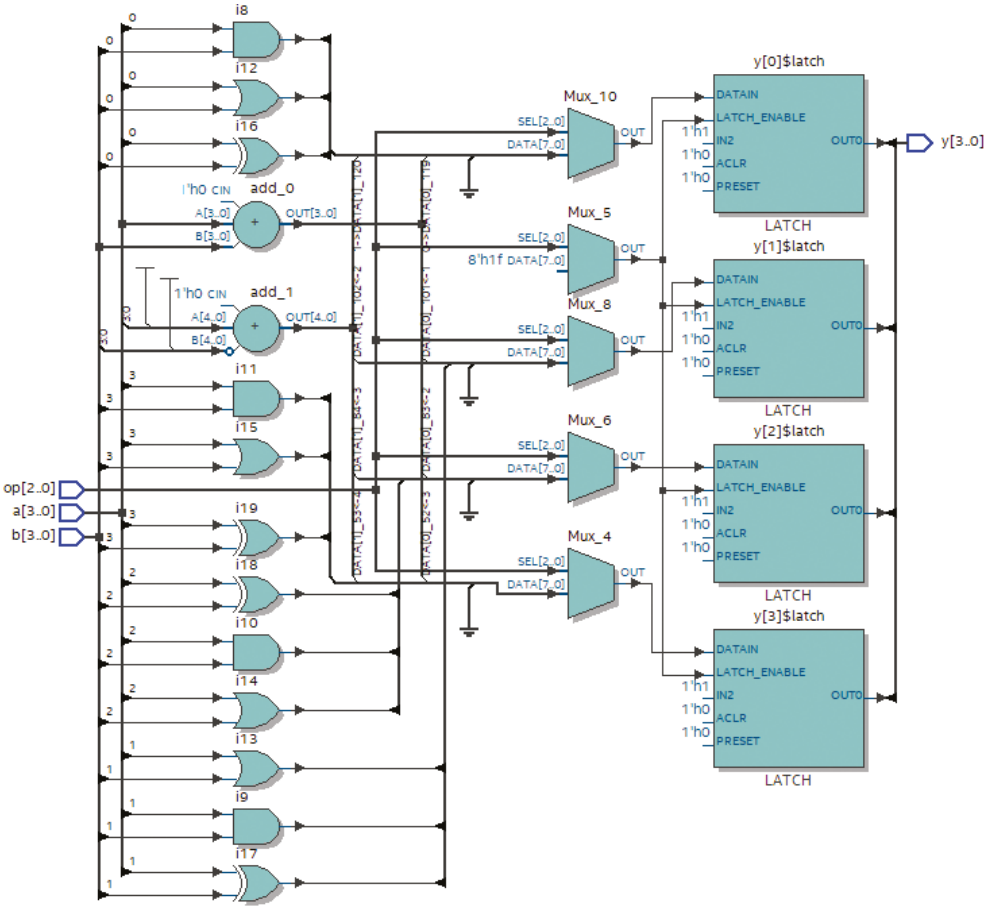
Rozważmy użycie kwalifikatorów **unique**, **unique0** i **priority** w operatorze **case** podczas implementacji prostego ALU.

Przykład 12.16. Proste ALU z wykorzystaniem operatora **case**

```
module alu_case
    #(parameter N=4)
    (input logic [N-1:0] a,b,
    input [2:0] op,
    output logic [N-1:0] y);
```

```
        always @(*)
            case (op)
                3'b000: y = a + b;
                3'b001: y = a - b;
                3'b010: y = a & b;
                3'b011: y = a | b;
                3'b100: y = a ^ b;
            endcase
endmodule
```

Wyniki syntezy projektu alu_case przedstawiono na rys. 12.7.



RYS. 12.7. Implementacja ALU z wykorzystaniem operatora **case**

ŹRÓDŁO: oprac. własne.

Na rys. 12.7 widzimy, że na wyjściach ALU znajdują się zatrzaski, ponieważ wartości wyjść nie są określone dla wszystkich możliwych kombinacji wejść.

Poniższy projekt alu_case_unique używa kwalifikatora **unique** przed operatorem **case**.

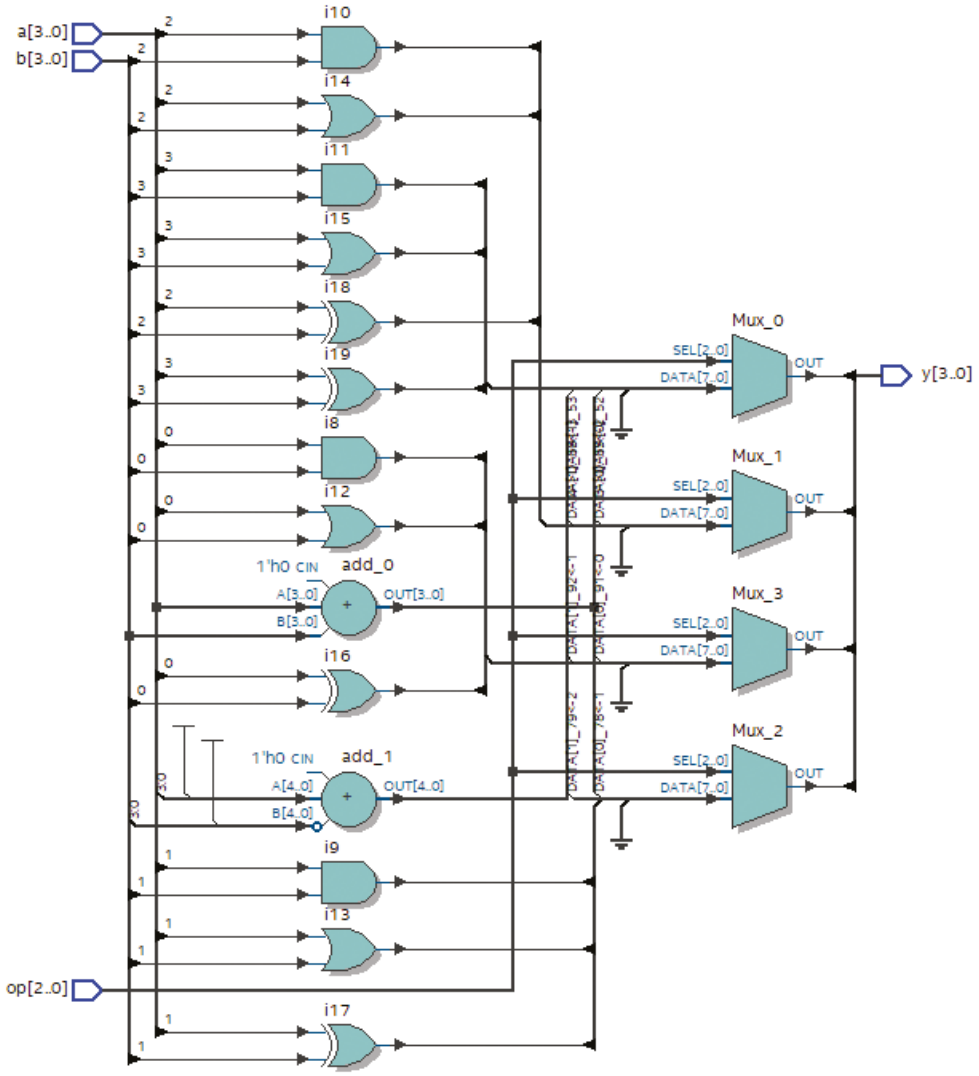
Przykład 12.17. Użycie kwalifikatora **unique** z operatorem **case** w projekcie ALU

```
module alu_case_unique
        #(parameter N=4)
        (input logic [N-1:0] a,b,
        input [2:0] op,
        output logic [N-1:0] y);

        always @(*)
                unique case (op)
                        3'b000: y = a + b;
                        3'b001: y = a - b;
                        3'b010: y = a & b;
                        3'b011: y = a | b;
                        3'b100: y = a ^ b;
                endcase
endmodule
```

Kompilator podczas syntezy projektu alu_case_unique nie ustawia zatrzasków na wyjściach (rys. 12.8), ale zgłasza, że deklaracja **unique** operatora **case** nie jest kompletna, czyli wszystko działa tak, jak się spodziewaliśmy.

Ustawienie kwalifikatora **unique0** przed operatorem **case** w Quartusie powoduje błąd kompilacji. Ustawienie kwalifikatora priorytetu **priority** przed operatorem **case** odpowiada schematowi z rys. 12.7 bez kwalifikatorów.

Jednak najlepszym rozwiązaniem jest jawne określenie wartości domyślnych dla wyjść za pomocą słowa kluczowego **default**.



RYS. 12.8. Wynik syntezy projektu alu_case_unique

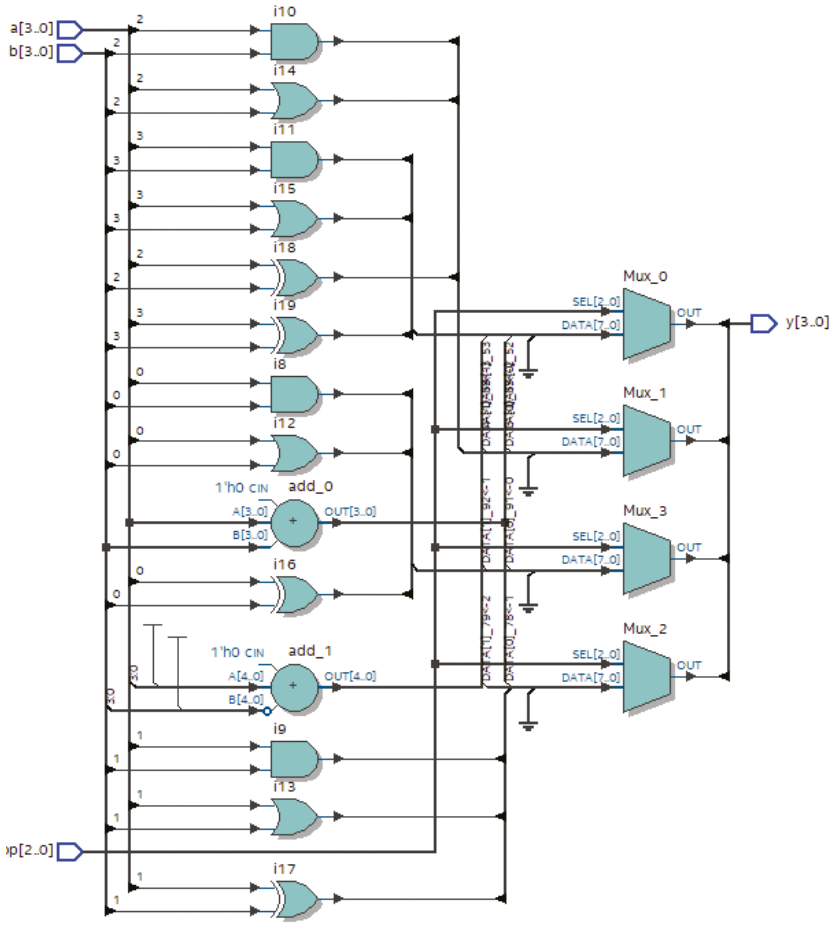
ŹRÓDŁO: oprac. własne.

Przykład 12.18. Użycie konstrukcji **default** w projekcie ALU

```
module alu_case_default
        #(parameter N=4)
        (input logic [N-1:0] a, b,
        input [2:0] op,
        output logic [N-1:0] y);
```

```
    always @(*)
        priority case (op)
            3'b000: y = a + b;
            3'b001: y = a - b;
            3'b010: y = a & b;
            3'b011: y = a | b;
            3'b100: y = a ^ b;
            default: y = {N{1'bx}};
        endcase
endmodule
```

Implementacja projektu alu_case_default (rys. 12.9) nie powoduje wyświetlenia żadnych komunikatów kompilatora.



RYS. 12.9. Wynik syntezy projektu alu_case_default

ŹRÓDŁO: oprac. własne.

12.2.5. Atrybuty syntezy full_case i parallel_case

Zastosowanie języków Verilog i SystemVerilog w praktyce inżynierskiej pokazało, że często poszczególne elementy języka wymagają doprecyzowania przy ich implementacji w kompilatorach, syntezatorach, symulatorach itp. *Atrybuty syntezy* (*synthesis attributes*) są używane do przekazywania takich informacji wyjaśniających. Innymi słowy, atrybuty języka SystemVerilog określają specyficzne właściwości operatorów i konstrukcji zaimplementowanych w konkretnym oprogramowaniu, takim jak kompilator języka SystemVerilog w pakiecie Quartus. Format atrybutów języka SystemVerilog jest następujący:

(* *attribute, attribute,...* ****)**

gdzie para znaków (* – początek listy atrybutów, a *) – koniec listy atrybutów; *attribute* – atrybut do zdefiniowania.

Atrybuty SystemVerilog mogą występować jako prefiksy deklaracji, instancji modułów, operatorów, portów itp. Mogą być również umieszczane jako przyrostki do operacji lub wywołań funkcji. Atrybuty mogą mieć wartości. Jeśli nie określono wartości atrybutu, wartością domyślną jest 1. Możliwe jest też zdefiniowanie kilku atrybutów jednocześnie. W takim przypadku są one oddzielone przecinkami na liście atrybutów.

Przykłady wykorzystania atrybutów:

```
(* INIT="1" *) logic Q;              // użycie atrybutu przy deklarowaniu zmiennej

sum = a + (* ripple_adder *) b;      // użycie atrybutu w wyrażeniu
                                     // jako przyrostek do operacji dodawania
```

Atrybuty jako możliwe jednostki konstrukcyjne zostały po raz pierwszy wprowadzone w języku Verilog-2001. Standard języka SystemVerilog nie definiuje jednak konkretnych atrybutów. Lista atrybutów języka jest definiowana przez twórców oprogramowania (kompilatory, syntezatory, symulatory itp.), w których język SystemVerilog jest obsługiwany.

Chociaż konkretne atrybuty języka SystemVerilog są definiowane przez twórców narzędzi do projektowania, wszystkie jego kompilatory obsługują dwa atrybuty: *full_case* i *parallel_case*.

12.2.6. Atrybut full_case

Zastosowanie operatora **case** przy opisie złożonych układów kombinacyjnych może prowadzić do sytuacji, w której wartości wyjść układu nie są określone dla wszystkich zbiorów wartości wejściowych. Z taką sytuacją mamy często do czynienia, gdy **case** nie

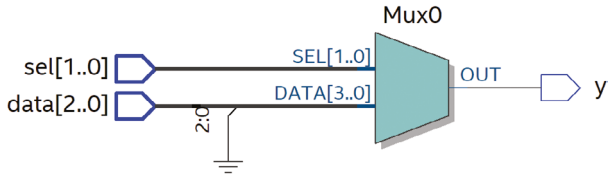
zawiera klauzuli domyślności **default**. Jeśli jednak dla wszystkich możliwych zestawów wejść nie zostaną określone wartości wyjścia układu, syntezator automatycznie ustawi zatrzaski na wyjściu układu (podobną sytuację obserwujemy na rys. 12.7 podczas realizacji przykładu 12.6). Atrybut *full_case* jest używany w celu uniknięcia automatycznego ustawiania zatrzasków na wyjściach układów kombinacyjnych.

Atrybut *full_case* mówi syntezatorowi, by traktował wartości wyjść układu jako niezdefiniowane (*don't care*) dla zestawów wejściowych, które nie są zdefiniowane w deklaracji operatora **case**. Rozważmy użycie atrybutu *full_case* przy opisie następującego układu kombinacyjnego.

Przykład 12.19. Użycie atrybutu *full_case* w opisie układu kombinacyjnego

```
module use_full_case (input [2:0] data,
                      input [1:0] sel,
                      output logic y);
      always @ (*)
            (* full_case *)              // zastosowanie atrybutu full_case
                                         // do operatora case
            case (sel)
                  2'b00 : y = data[0];
                  2'b01 : y = data[1];
                  2'b10 : y = data[2];
                  // brak kombinacji wejść 2'b11
            endcase
endmodule
```

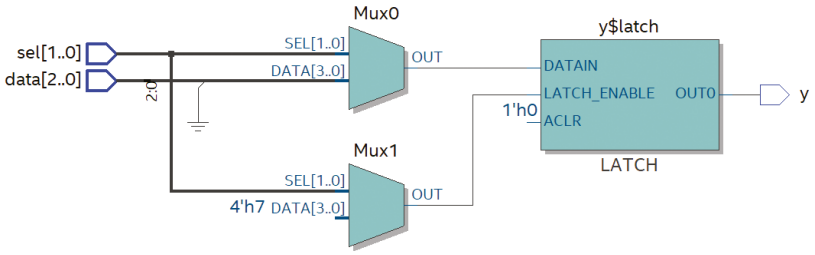
Wynik syntezy projektu use_full_case przedstawiono na rys. 12.10.



RYS. 12.10. **Wynik syntezy projektu use_full_case**

ŹRÓDŁO: **oprac. własne.**

Implementacja tego samego przykładu bez atrybutu *full_case* wymaga dwukrotnie większej liczby elementów logicznych i została przedstawiona na rys. 12.11.



RYS. 12.11. Wynik syntezy projektu use_full_case bez atrybutu full_case

ŹRÓDŁO: oprac. własne.

12.2.7. Atrybut parallel_case

Przypomnijmy sobie, jak działa operator case: obliczane jest wyrażenie *expression*; wartość ta jest kolejno porównywana z wyrażeniami *item_expression* w takiej kolejności, w jakiej są one zapisane w kodzie źródłowym; jeśli jest dopasowanie, wykonywany jest operator następujący po danym *item_expression*; na koniec wykonywane jest wyjście z operatora **case**. Pozostałe wyrażenia *item_expression* nie są sprawdzane. Rozważany algorytm realizacji operatora **case** wymaga obecności w układzie kodera priorytetowego, który zakończy działanie operatora **case**, jeśli wartość wyrażenia *expression* będzie pasować do jednego z wyrażeń *item_expression*.

Istnieją dwa powody zmiany strategii realizacji operatora **case**.

Po pierwsze, jeśli wiadomo, że wartość wyrażenia *expression* pasuje do jednego z wyrażeń *item_expression* i nie może być ona dopasowana przez inne wyrażenia *item_expression*. Przykładem może być opis pełnej tablicy prawdy systemu funkcji boolowskich za pomocą operatora **case**. W tym przypadku dla zwiększenia szybkości działania układu, a czasem dla zmniejszenia kosztów realizacji sensowne jest pozbycie się kodera priorytetowego w realizacji układu operatora **case**.

Po drugie, wyrażenia *item_expression* operatorów **casex** i **casez** mogą używać niezdefiniowanych wartości bitowych (oznaczonych znakiem „?”). Wówczas ta sama wartość wyrażenia *expression* może się pokrywać z kilkoma wyrażeniami *item_expression*. Czasami wymagane jest, aby operatory zarządzane przez takie wyrażenia funkcjonowały równolegle. W takiej sytuacji jest konieczne równoległe sprawdzenie wartości wyrażenia *expression* z wszystkimi wyrażeniami *item_expression* i wykonanie odpowiednich operatorów w przypadku zbieżności.

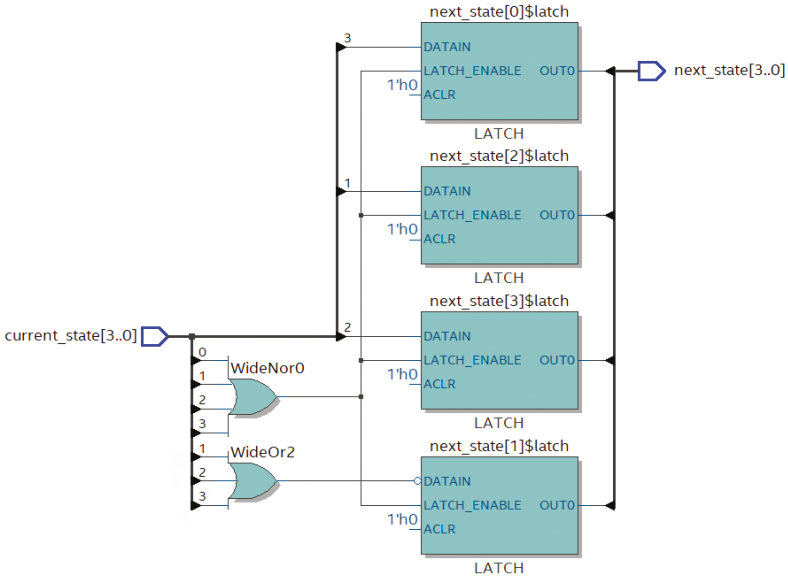
Do rozwiązania podanych problemów służy atrybut *parallel_case*. Mówi on syntezatorowi, żeby zamiast implementować koder priorytetowy, ma wykonać równoległą implementację instrukcji dla wszystkich wyrażeń *item_expression* instrukcji **case**.

Jako przykład rozważmy użycie atrybutu *parallel_case* w implementacji funkcji przejścia automatu skończonego w przypadku kodowania stanów automatu w kodzie unarnym (one-hot).

Przykład 12.20. Użycie atrybutu *parallel_case* do implementacji funkcji przejścia automatu skończonego

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

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

      always @(*)
      (* parallel_case *)                     // zastosowanie atrybutu parallel_case
      case (1'b1)                             // jedynka jest poszukiwana w bitach kodu
            current_state[0] : next_state = s2;  // ponieważ kod jest unarny,
                                                 // tylko jeden z operatorów zostanie wykonany
current_state[1] : next_state = s3;
            current_state[2] : next_state = s4;
            current_state[3] : next_state = s1;
      endcase
endmodule
```

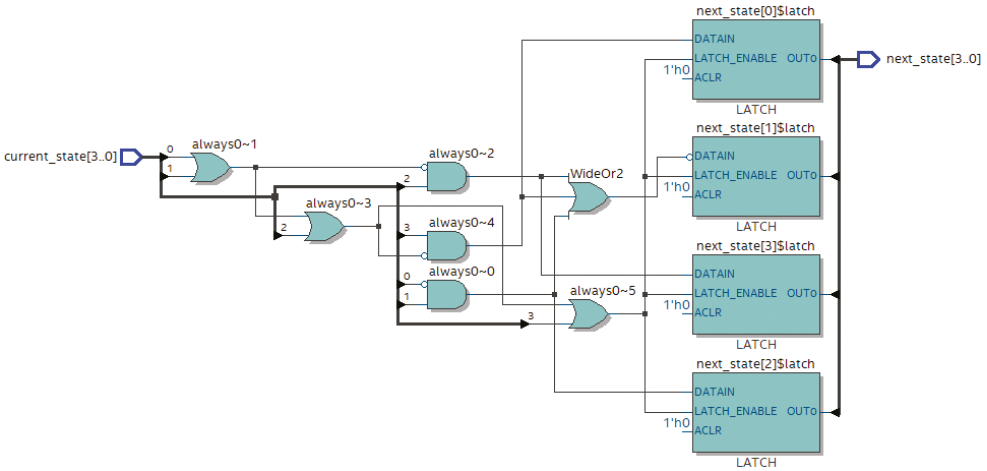
Realizacja przykładu 2 wymaga 6 elementów logicznych i została przedstawiona na rys. 12.12.



RYS. 12.12. Wynik syntezy projektu one_hot_assign1

ŹRÓDŁO: oprac. własne.

Implementacja tego samego przykładu bez atrybutu *parallel_case* wymaga 9 elementów logicznych i została przedstawiona na rys. 12.13.



RYS. 12.13. Wynik syntezy projektu one_hot_assign1 bez atrybutu *parallel_case*

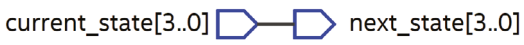
ŹRÓDŁO: oprac. własne.

Atrybuty *full_case* i *parallel_case* mogą być używane razem.

Przykład 12.21. Wspólne użycie atrybutów *full_case* i *parallel_case* do realizacji funkcji przejść automatu skończonego

```
module one_hot_assign2 (
input [3:0] current_state,                    // aktualny stan automatu skończonego
      output logic [3:0] next_state);         // następny stan automatu skończonego

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

      always @(*)
      (* parallel_case, full_case *)                  // zastosowanie atrybutów parallel_case
                                                      // i full_case
      case (1'b1)
            current_state[0] : next_state = s2;
            current_state[1] : next_state = s3;
            current_state[2] : next_state = s4;
            current_state[3] : next_state = s1;
      endcase
endmodule
```

Realizacja projektu one_hot_assign2 nie wymaga w ogóle elementów logicznych i została przedstawiona na rys. 12.14.



RYS. 12.14. Wynik syntezy projektu one_hot_assign2 przy jednoczesnym użyciu atrybutów *full_case* i *parallel_case*

ŹRÓDŁO: oprac. własne.

12.2.8. Porównanie kwalifikatorów unique i priority z atrybutami full_case i parallel_case oraz konstrukcjami else i default

Kwalifikator **unique** jest równoważny użyciu przy syntezie dwóch atrybutów *full_case* i *parallel_case* w deklaracji **case**, a kwalifikator **priority** jest równoważny użyciu atrybutu *full_case*. Jednak powyższe kwalifikatory dają projektantowi więcej możliwości niż atrybuty *full_case* i *parallel_case*.

Po pierwsze, kwalifikatory **unique** i **priority** zmniejszają ryzyko niespójności między narzędziami programistycznymi, ponieważ atrybuty nie są ściśle zdefiniowane w standardzie języków SystemVerilog i Verilog oraz są jedynie zaleceniami dla twórców oprogramowania, natomiast kwalifikatory są częścią języka i są ściśle zdefiniowane w standardzie języka SystemVerilog.

Po drugie, kwalifikatory te zapewniają dodatkowe kontrole semantyczne, zarówno przy syntezie, jak i symulacji, które mogą ujawnić potencjalne problemy projektowe już na wczesnych etapach projektowania.

Nadal pozostaje pytanie, jak skuteczne jest stosowanie kwalifikatorów **unique** i **priority** w operatorach **if** i **case** oraz atrybutów *full_case* i *parallel_case* w konstrukcji **case**. W tym celu zbadano następujące proste projekty ALU:

- alu_if_else_if – projekt z przykładu 12.4;
- alu_if_else_if_else – projekt z przykładu 12.5;
- alu_if_else_if_unique – projekt z przykładu 12.6;
- alu_if_else_if_priority – projekt z przykładu 12.7;
- alu_case – projekt z przykładu 12.16;
- alu_case_default – projekt z przykładu 12.18;
- alu_case_unique – projekt z przykładu 12.17;
- alu_case_ priority – projekt z przykładu 12.16 z użyciem kwalifikatora **priority**;
- alu_full_case – projekt z przykładu 12.16 z wykorzystaniem atrybutu *full_case*;
- alu_parallel_case – projekt z przykładu 12.16 z wykorzystaniem atrybutu *parallel_case*;
- alu_full_parallel_case – projekt z przykładu 12.16 wykorzystujący jednocześnie atrybuty *full_case* i *parallel_case*.

Badania eksperymentalne przeprowadzono z implementacją ALU przy użyciu systemu Quartus Prime Pro w wersji 21.1 na FPGA rodziny Cyclone 10 GX. Wyniki przedstawiono w tabeli 12.1, gdzie: LE – *koszt implementacji*, liczba użytych elementów logicznych (*adaptive look-up tables* – ALUT); Dmax – maksymalna długość ścieżki krytycznej elementów ALUT; Dav – średnia długość ścieżki krytycznej elementów ALUT; Fmax – maksymalna częstotliwość pracy w megahercach (MHz); Best – wartość najlepsza danego parametru.

TABELA 12.1. Badanie zastosowania kwalifikatorów i atrybutów w implementacji prostego ALU

Projekt	LE	Dmax	Dav	Fmax	Uwagi
alu_if_else_if	15	3,40	2,46	125	z zatrzaskami
alu_if_else_if_unique	15	3,40	2,46	–	z zatrzaskami
alu_if_else_if_priority	15	3,40	2,46	–	z zatrzaskami
alu_if_else_if_else	10	3,40	2,93	344	bez zatrzasków
alu_case	14	2,40	1,79	75	z zatrzaskami
alu_case_default	9	2,40	2,14	337	bez zatrzasków
alu_case_unique	9	2,40	2,14	337	bez zatrzasków
alu_case_ priority	14	2,40	1,79	337	z zatrzaskami
alu_full_case	9	2,40	2,14	337	bez zatrzasków
alu_parallel_case	14	2,40	1,79	75	z zatrzaskami
alu_full_parallel_case	9	2,40	2,14	337	bez zatrzasków
Best	9	2,40	1,79	344	

ŹRÓDŁO: oprac. własne.

Uwaga. Maksymalna częstotliwość pracy została określona przy użyciu programu Quartus Prime w wersji 18.1 dla FPGA rodziny Cyclone IV E.

Z analizy tabeli 12.1 wynika, że w przypadku zastosowania operatora **if-else-if** bez ostatniego słowa kluczowego **else** w kodzie ALU (projekt alu_if_else_if) syntezator wprowadza do układu zatrzaski, przy czym koszt implementacji wynosi 15, a prędkość 125 MHz. Zastosowanie kwalifikatorów **unique** (projekt alu_if_else_if_unique) i **priority** (projekt alu_if_else_if_priority) nie zmienia wyników syntezy. Wprowadzenie konstrukcji **else**, która odwołuje się do ostatniego **if** i generuje na wyjściu niezdefiniowaną wartość (projekt alu_if_else_if_else), pozwala na obniżenie kosztu implementacji do 10 i zwiększenie prędkości do 344 MHz.

Jeśli kod ALU używa operatora **case** bez użycia konstrukcji **default** (projekt alu_case), syntezator wprowadza do schematu również zatrzaski, przy czym koszt implementacji wynosi 14, a prędkość 75 MHz. Wprowadzenie konstruktu **default** (projekt alu_case_default) zmniejsza koszt implementacji do 9 i zwiększa prędkość do 337 MHz. Użycie kwalifikatora **unique** (projekt alu_case_unique) w kodzie ALU odpowiada

projektowi alu_case_default, a użycie kwalifikatora **priority** (projekt alu_case_ priority) odpowiada projektowi alu_case. Z kolei użycie atrybutu *full_case* odpowiada projektowi alu_case_default, a atrybutu *parallel_case* – projektowi alu_case. Jednoczesne użycie atrybutów *full_case* i *parallel_case* odpowiada projektowi alu_case_default.

Zauważmy, że długość ścieżki krytycznej, maksymalna Dmax i średnia Dav, nie wpływają na wydajność projektów ALU.

Wnioski. Syntezator Quartus ignoruje kwalifikatory **unique** i **priority** w instrukcjach **if**, ale uwzględnia je w instrukcjach **case**. Atrybuty *full_case* i *parallel_case* są w pełni obsługiwane przez syntezator Quartus.

Podsumujmy kilka wyników. Język SystemVerilog zapewnia następujące cechy dotyczące stosowania operatorów **if** i **case**:

- użycie w łańcuchu operatorów **if-else-if** konstrukcji **else** odnoszącej się do ostatniego **if** w celu określenia wartości wyjść w przypadku, gdy nie powiedzie się żadne z wyrażeń logicznych;
- użycie **default** w operatorze **case** w tym samym celu;
- użycie atrybutów *full_case* i *parallel_case* w operatorze **case**;
- użycie kwalifikatorów **unique** i **priority** w operatorach **if** i **case**.

Do projektanta należy wybór, którą z powyższych możliwości wykorzystać w konkretnym przypadku. Zauważmy, że nie wszystkie syntezatory obsługują kwalifikatory **unique** i **priority**.

12.2.9. Kwalifikator inside operatora case

Czasami konieczne jest ustalenie, czy wartość wyrażenia *expression* operatora **case** należy do jakiegoś zakresu wartości. Rozwiązaniem jest wypisanie wszystkich elementów *item_expression*, które odpowiadają temu zakresowi, oddzielonych przecinkami. Jednak język SystemVerilog zapewnia bardziej eleganckie rozwiązanie poprzez użycie kwalifikatora **inside**.

Gdy kwalifikator **inside** jest używany z operatorem **case**, jest on zapisywany po wyrażeniu *expression* i pozwala określić dopasowanie do jednego z elementów zakresu, który reprezentuje *item_expression*. Na przykład:

```
logic [2:0] s;
case (s) inside
      3'b0?0 : y = a;          // odpowiada 'b000, 'b010, 'b0x0 i 'b0z0
      [4:7] : y = b;           // odpowiada 'b100, 'b101, 'b110 i 'b111
```

Kwalifikator **inside** może być również łączony z kwalifikatorami **unique** i **priority**.

12.3. Operatory pętli

Język SystemVerilog dostarcza sześć operatorów do opisu pętli: **for**, **repeat**, **foreach**, **while**, **do-while** oraz **forever**. Nowością w stosunku do języka Verilog jest wprowadzenie operatorów **foreach** i **do-while**.

12.3.1. Pętla for

Składnia instrukcji pętli **for**:

for ([*for_initialization*]; [*expression*]; [*for_stop*])
    statement_or_null

gdzie: *for_initialization* jest listą operatorów inicjalizacji zmiennych pętli; *expression* jest wyrażeniem warunku zakończenia pętli; *for_stop* jest listą operatorów wykonywanych na końcu każdej pętli. Zauważmy, że każdy z elementów *for_initialization*, *expression* i *for_stop* może być pominięty, ale nie wszystkie razem.

Działanie operatora **for** jest następujące:

a) wykonywane są operatory inicjalizacji zmiennych z listy *for_initialization* w pętli;
b) obliczane jest wyrażenie *expression*; jeśli wynik jest fałszywy, to operator pętli **for** zostaje zakończony; w przeciwnym razie (jeśli wyrażenie *expression* jest prawdziwe lub pominięte) wykonywany jest operator *statement_or_null* i następuje krok c;
c) wykonywane są instrukcje *for_stop*, które zwykle zmieniają wartości zmiennych sterujących pętlą.

W SystemVerilog zmienne sterujące pętlą **for** mogą być zadeklarowane albo przed nią, albo w pętli **for** jako część konstrukcji *for_initialisation*. W tym przypadku ta sama zmienna może być użyta w dwóch różnych pętlach jednocześnie. Na przykład:

```
module m;

    initial begin
        for (int i = 0; i <= 255; i++)
        ...
    end

    initial begin
        loop2: for (int i = 15; i >= 0; i--)
        ...
    end
endmodule
```

Zmienne zadeklarowane w pętli **for** są zmiennymi automatycznymi. Są tworzone i inicjalizowane w momencie wywołania pętli **for**, a niszczone w momencie wyjścia z pętli. Zmienne w pętli **for** poza nią nie istnieją i nie mogą być używane. Jeśli zmienna musi zostać przywołana poza pętlą, musi być poza nią zadeklarowana. Na przykład:

```
always_comb begin                   // szukanie pierwszej jedynki w wektorze danych
        int i;                      // lokalna zmienna bloku
        for (i=0; i<=63; i++) begin
                if (data[i]) break; // zakończenie pętli if
        end
        if (i > 7)                  // użycie zmiennej w pętli
                                    // i ma wartość liczby faktycznie wykonanych iteracji
        ...
end
```

Lista operator***ów*** *for_initialization* może być jedną deklaracją lub listą deklaracji oddzielonych przecinkami. Lista operatorów *for_stop* może być również jednym operatorem przypisania, wyrażeniem inkrementacji bądź dekrementacji albo wywołaniem funkcji lub kilkoma z nich rozdzielonymi przecinkami. Na przykład:

```
for (int i = 0, done = 0, j = 0; j * i < 125; j++, i++)
begin
        done = 1;
        v = v + a[j] + b[i];
end
```

Podczas inicjalizacji operatora **for** należy zadeklarować lokalnie wszystkie zmienne sterujące albo też nie deklarować żadnej z nich. W drugiej pętli powyższego przykładu zmienne i, done oraz j są zadeklarowane lokalnie.

Każda zmienna w pętli może być zadeklarowana jako inny typ. Na przykład:

```
for (int i=1, byte j=0; i*j < 128; i++, j+=3) ...
```

Poniższy kod byłby nieprawidłowy, ponieważ zmienna *y* jest tutaj zadeklarowana lokalnie, podczas gdy zmienna *x* nie została zadeklarowana lokalnie:

```
for (x = 0, int y = 0; ...) ...
```

W konstrukcji inicjalizacyjnej *for_initialisation*, która deklaruje wiele zmiennych lokalnych, wyrażenie inicjalizacyjne zmiennej lokalnej może używać wcześniej zadeklarowanych zmiennych lokalnych, np.:

```
for (int i = 0, j = i+offset; i < N; i++, j++) ...
```

Do zmiennych lokalnych zadeklarowanych jako część pętli **for** nie można się odwoływać hierarchicznie.

12.3.2. Pętla repeat

Format operatora powtórzenia **repeat**:

repeat (*number*) *statement*

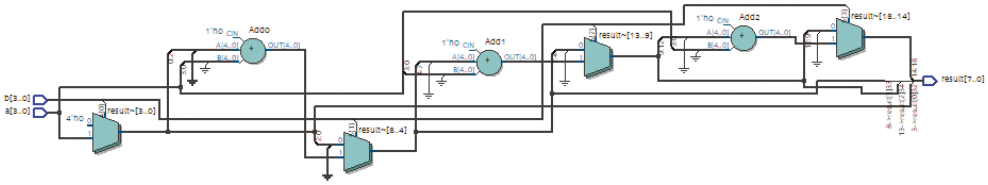
gdzie *number* jest liczbą powtórzeń pętli. Konstrukcja *number* w operatorze **repeat** może być liczbą lub wyrażeniem, którego wartość jest obliczana raz, na początku operatora **repeat**, i nie zmienia się w trakcie wykonywania pętli.

Działanie operatora **repeat** jest następujące. Operator *statement* jest wykonywany tyle razy, ile wynosi liczba lub wyrażenie *number*. Jako przykład zastosowania operatora **repeat** przyjrzyjmy się opisowi układu mnożącego.

Przykład 12.22. Użycie operatora **repeat** w jednostce mnożącej

```
module my_mult
        #(parameter N = 4)
        (input logic [N-1:0] a, b,              // operandy
        output logic [2*N-1:0] result);         // wynik

        always_comb                             // opisuje logikę kombinacyjną
        begin: mult                             // blok nazwany
                logic [2*N-1:0] shift_a;        // deklaracja zmiennych lokalnych
                logic [N-1:0] shift_b;

                shift_a = a;                            // inicjalizacja zmiennych
                shift_b = b;
                result = 0;
                repeat (N) begin                        // pętla repeat jest wykonywana N razy
                        if (shift_b[0])                 // sprawdzamy najmłodszy bit mnożnika
                                result = result + shift_a;
                        shift_a = shift_a << 1;         // przesunięcie mnożnika w lewo
                        shift_b = shift_b >> 1;         // przesunięcie mnożnika w prawo
                end
        end
endmodule
```

Wynik syntezy projektu my_mult przedstawiono na rys. 12.15.



RYS. 12.15. Implementacja układu mnożącego z wykorzystaniem operatora pętli **repeat**
ŹRÓDŁO: oprac. własne.

Operator **repeat** jest syntetyzowalny tylko wtedy, gdy liczba powtórzeń *number* jest stałą.

12.3.3. Pętla foreach

Operator pętli **foreach** iteruje po elementach tablicy. Przypomnijmy, że został on krótko omówiony w podrozdziale 7.1.13. Składnia operatora **foreach**:

foreach (*array_identifier* [*list_loop_variables*]) *statement*;

gdzie: *array_identifier* to nazwa tablicy; *list_loop_variables* to lista zmiennych pętli; *statement* to operator wykonywany w pętli. Każda zmienna pętli na liście *list_loop_variables* odpowiada jednemu z wymiarów tablicy.

Działanie operatora **foreach** jest podobne do operatora **repeat**, który wykorzystuje granice tablicy zamiast wyrażenia do wskazania liczby powtórzeń. Na przykład:

```
string words [2] = '{ „hello", „world" };      // tablica dwóch ciągów znaków
int a [1:8] [1:3];                             // rozpakowana dwuwymiarowa
                                               // tablica liczb całkowitych
foreach( words [ j ] )
        $display( j , words[j] );          // wyświetlanie każdego indeksu i wierszy tablicy słów

foreach ( a[ k, m ] )
        a[k][m] = k * m;                   // inicjalizacja elementów tablicy a
```

W tablicach wielowymiarowych numery indeksów następują od lewej do prawej, poczynając od 1, i odpowiadają wymiarom tablicy. Przypomnijmy, że w tablicach wielowymiarowych najpierw wymienia się indeksy dla wymiarów rozpakowanych, a następnie dla pomiarów spakowanych.

Liczba zmiennych pętli nie może przekraczać liczby wymiarów zmiennej tablicowej. Zmienne pętli mogą być pominięte, aby pokazać brak iteracji na tym wymiarze tablicy, końcowe przecinki w liście również mogą być pominięte. Na przykład:

```
//      1 2 3         3   4       1   2          -> wymiary
int A [2][3][4]; bit [3:0][2:1] B [5:1][4];

foreach  ( A [ i, j, k ] ) ...
foreach ( B [ q, r, , s ] ) ...
```

Pierwsza pętla **foreach** iteruje i od 0 do 1, j od 0 do 2 i k od 0 do 3. Druga pętla **foreach** iteruje q od 5 do 1, r od 0 do 3 i s od 2 do 1 (iteracja nad trzecim indeksem jest pomijana).

12.3.4. Pętla while

Składnia operatora pętli **while**:

while (*expression*) *statement_or_null*;

Operator **while** wielokrotnie wykonuje instrukcję *statement_or_null*, jak długo wyrażenie *expression* jest prawdziwe. Jeśli wyrażenie nie jest prawdziwe na początku operatora **while**, to operator *statement_or_null* nie jest w ogóle wykonywany.

Poniższy przykład zlicza liczbę wartości logicznych 1 w zmiennej data:

```
begin : count1s
        logic [7:0] t;
        count = 0;
        t = data;
        while (t) begin
                if (t[0]) count++;
                t >>= 1;
        end
end
```

12.3.5. Pętla do-while

Składnia operatora pętli **do-while**:

do *statement_or_null* **while** (*expression*);

Pętla **do-while** różni się od **while** tym, że sprawdza ona swoje wyrażenie *expression* na końcu pętli. Tak więc operator *statement_or_null* w pętli **do-while** jest wykonywany co najmniej raz. Na przykład poniższy kod wyświetla tablicę łańcuchów:

```
string s;
if ( map.first( s ) )
        do
                $display( "%s : %d\n", s, map[ s ] );
        while ( map.next( s ) );
```

12.3.6. Pętla forever

Składnia operatora pętli **forever**:

forever *statement_or_null*;

Operator pętli **forever** wielokrotnie wykonuje instrukcję *statement_or_null*. Aby uniknąć nieskończonej pętli z zerowym opóźnieniem, operator **forever** powinien być używany tylko w połączeniu z elementami sterowania czasem lub operatorem **disable**. Na przykład:

```
initial begin
        clock1 <= 0;
        clock2 <= 0;
        fork            // generowanie sygnałów zegarowych
                forever #10 clock1 = ~clock1;
                #5 forever #10 clock2 = ~clock2;
        join
end
```

12.4. Operatory przejścia

W języku Verilog brakuje operatorów **break**, **continue** i **return**, podobnych do tych z języka C. W tym celu w Verilogu można wykorzystać operator **disable**. Przykłady jego użycia jako operatora **break**, **continue**, **return** i **goto** podano w podrozdziale 11.4.3. Przypomnijmy, że **disable** jest przeznaczony do zakończenia wykonywania sekwencji poleceń w bloku nazwanym.

Używanie tego samego operatora **disable** do przerywania iteracji pętli, kończenia pętli oraz kończenia zadań i funkcji nie zawsze jest wygodne, dlatego język SystemVerilog udostępnia instrukcje **break**, **continue** i **return**.

Składnia operatorów przejścia:

continue;
break;
return [*expression*];

Operatory **continue** i **break** są używane tylko w pętlach. Operator **continue** przechodzi do końca pętli i przekazuje sterowanie do następnej iteracji pętli. Operator **break** kończy pętlę, a sterowanie jest przekazywane do następnego operatora po operatorze pętli. Polecenia **continue** i **break** nie mogą być używane wewnątrz bloku **fork-join**.

Operator **return** jest używany tylko w podprogramie, czyli w funkcji lub zadaniu. Umożliwia on wyjście z funkcji bądź zadania. Jeśli funkcja zwraca wartość, typ wyrażenia musi odpowiadać typowi wartości zwracanej przez funkcję.

Należy zauważyć, że w języku SystemVerilog nie ma operatora przejścia **goto**.

12.4.1. Operator continue

Operator **continue** kończy bieżącą pętlę operatora **for** i przekazuje sterowanie do wykonania kolejnych pętli operatora **for**. Na przykład:

```
logic [15:0] array [0:255];

always_comb begin
        for (int i = 0; i <= 255; i++)
        begin : loop
                if (array[i] == 0)          continue;       // pomijanie elementów zerowych
                transform_function(array[i]);   // wywołanie funkcji do przetworzenia elementu
        end
end
```

12.4.2. Operator break

Operator **break** przerywa wykonywanie operatora **for** i przekazuje sterowanie do operatora, który następuje po operatorze **for**. Na przykład:

```
// znalezienie pierwszego ustawionego bitu w zakresie wektora bitów
always_comb begin
        first_bit = 0;
        for (int i=0; i<=63; i=i+1)
        begin
                if (i < start_range) continue;          // przechodzimy do następnego cyklu
                if (i > end_range) break;               // wyjście z pętli for
                if ( data[i] )
```

```
            begin
                first_bit = i;
                break;              // wyjście z pętli for
            end
        end
        ... // przetwarzanie pierwszego ustawionego bitu
end
```

12.4.3. Operator return

Operator powrotu **return** kończy zadanie lub funkcję, nie dochodząc do końca zadania albo funkcji. Na przykład:

```
task add_up_to_max
        (input [ 5:0] max,
        output [63:0] result);

        result = 1;
        if (max == 0) return;           // zakończenie zadania
        for (int i=1; i<=63; i=i+1)
        begin
            result = result + result;
            if (i == max) return;       // zakończenie zadania
        end
endtask
```

Operator **disable** może być również użyty do zakończenia zadania, ale nie może być użyty do zakończenia funkcji, w przeciwieństwie do operatora **return**.

Przykład 12.23. Funkcja logarytmiczna o podstawie 2

```
function automatic int log2 (input int n);
        if (n <=1) return 1;        // wcześniejsze wyjście dla argumentów ujemnych
        log2 = 0;                   // inicjalizacja wartości funkcji
        while (n > 1)               // główny cykl obliczeniowy
        begin
            n = n/2;                // dzielenie argumentu przez 2
            log2++;                 // zwiększenie wartości funkcji o 1
        end
        return log2;                // zakończenie funkcji ze zwróceniem wartości funkcji
endfunction
```

12.5. Wnioski

Operatorami proceduralnymi w języku SystemVerilog są: operatory wyboru (**if**, **case**); operatory pętli (**for**, **repeat**, **foreach**, **while**, **do-while**, **forever**); operatory przejścia (**break**, **continue**, **return**); bloki sekwencyjne (**begin-end**) i równoległe (**fork-join**); operatory sterowania czasem (**#**, **@**, **wait**); operatory sterowania procesem (**wait**, **disable**); przypisania proceduralne (=, <=, **assign**, **force**, szablony przypisania); wywołania zadań i funkcji.

W języku SystemVerilog operatory proceduralne zapisywane są w blokach proceduralnych, zadaniach i funkcjach i służą do opisu projektu w sposób *algorytmiczny* lub *behawioralny*.

Operator **if-else** pozwala na wykonanie pewnych fragmentów kodu, gdy spełnione są określone warunki. Jego działanie jest podobne do działania operatora **if** w języku programowania C.

Łańcuch operatora **if-else** nazywany jest konstrukcją ***if-else-if***. Wyrażenia w tej konstrukcji są obliczane w kolejności. Jeżeli wyrażenie jest prawdziwe, to wykonywany jest odpowiedni operator i łańcuch się kończy.

Ostatni operator w łańcuchu **if-else-if** po słowie kluczowym **else** odpowiada przypadkowi, gdy żadne z wyrażeń nie jest prawdziwe, co odpowiada pewnej domyślnej akcji, np. wykryciu błędów. Ten operator w łańcuchu **if-else-if** jest opcjonalny.

Do realizacji układów kombinacyjnych w łańcuchu operatora **if-else-if** każdy operator **if** musi mieć swoją klauzulę **else**.

Kwalifikator **unique** przed słowem kluczowym **if** mówi kompilatorowi, że wyrażenia w konstrukcji **if-else-if** są unikatowe i nigdy dwa wyrażenia nie mogą być jednocześnie prawdziwe. Jeśli ten warunek unikalności zostanie naruszony, nastąpi komunikat o błędzie. Układ z kwalifikatorem **unique** jest implementowany jako logika kombinacyjna bez zatrzasków.

Użycie kwalifikatora **unique** w konstrukcji **if-else-if** wskazuje kompilatorowi, by nie wprowadzał do schematu kodera priorytetowego, co zwiększa niezawodność i wydajność projektu oraz zmniejsza koszty implementacji.

Podczas używania kwalifikatora **unique0** w konstrukcji **if-else-if** kompilator nie generuje komunikatu o błędzie, jeśli warunek unikalności wyrażenia zostanie naruszony.

Kwalifikator priorytetu **priority** w konstrukcji **if-else-if** informuje narzędzie projektowe, że wyrażenia muszą być oceniane w kolejności określonej w kodzie, czyli ważna jest kolejność decyzji.

Jeśli użyto kwalifikatora priorytetu **priority** i żadne z wyrażeń w konstrukcji **if-else-if** nie jest prawdziwe, kompilator wygeneruje komunikat ostrzegawczy, ale logika kombinacyjna bez zatrzasków zostanie zaimplementowana.

Operator **case** pozwala na wykonanie różnych fragmentów kodu w zależności od wartości obliczonego wyrażenia.

Operatory **casex** i **casez** służą do porównywania wyrażeń, których wartości zawierają wartości x i z.

Zastosowanie kwalifikatora **unique** w konstrukcji **case** mówi kompilatorowi, że jedna i tylko jedna stała *item_expression* odpowiada wartości wyrażenia *expression*. Dzięki temu można uniknąć wprowadzania do schematu kodera priorytetowego i zaimplementować schemat jako logikę kombinacyjną. Gdy określony jest kwalifikator **unique**, kompilator sprawdza, czy wszystkie wyrażenia *item_expression* są unikatowe i się nie pokrywają. Jeśli te warunki zostaną naruszone, kompilator podaje komunikat o błędzie.

Kwalifikator **unique0** jest podobny do kwalifikatora **unique**, z tą różnicą, że kompilator nie generuje komunikatu o błędzie, jeśli warunek unikalności dla wyrażeń *item_expression* zostanie naruszony.

Kwalifikator priorytetu **priority** w konstrukcji **case** mówi kompilatorowi, że przynajmniej jeden element *item_expression* musi pasować do wyrażenia *expression*; jeśli pasuje więcej niż jeden element, to musi zostać wykonana pierwsza pasująca gałąź. Jeśli wyrażenie *expression* nie pasuje do żadnej ze stałych, kompilator generuje komunikat o błędzie. Przy zastosowaniu kwalifikatora priorytetu **priority** logika kombinacyjna jest implementowana bez zatrzasków wyjściowych.

Atrybuty języka SystemVerilog określają specyficzne właściwości operatorów i konstruktów zaimplementowanych w konkretnym oprogramowaniu, takim jak kompilator języka SystemVerilog pakietu Quartus.

Atrybuty języka mogą być przypisane jako prefiksy do deklaracji, instancji modułów, operatorów, portów, a także jako przyrostki do operacji lub wywołań funkcji. Atrybuty mogą mieć wartości, domyślną wartością jest 1. Można określić wiele atrybutów jednocześnie, oddzielając je przecinkami. Atrybuty są definiowane przez twórców oprogramowania projektowego, ale wszystkie kompilatory języka SystemVerilog obsługują dwa atrybuty: *full_case* i *parallel_case*.

Atrybut *full_case* mówi syntezatorowi, aby traktował wartości wyjściowe układów jako niezdefiniowane (*don't care*) dla wektorów wejściowych, które nie są zdefiniowane w deklaracji **case**. Pozwala to uniknąć automatycznego ustawiania zatrzasków na wyjściach układów kombinacyjnych.

Atrybut *parallel_case* wskazuje syntezatorowi konieczność zapewnienia równoległej implementacji operatorów dla wszystkich wyrażeń *item_expression* operatora **case** zamiast wykorzystania kodera priorytetowego.

Atrybuty *full_case* i *parallel_case* mogą być używane razem.

Kwalifikator **unique** jest równoważny przy syntezie użycia jednocześnie atrybutów *full_case* i *parallel_case* w operatorze **case**, a kwalifikator **priority** jest równoważny użyciu atrybutu *full_case*.

Kwalifikatory **unique** i **priority** dają projektantowi więcej możliwości niż atrybuty *full_case* i *parallel_case*.

Język SystemVerilog zapewnia następujące możliwości dotyczące stosowania operatorów **if** i **case**:

- zastosowanie w łańcuchu operatorów **if-else-if** konstrukcji **else** związanej z ostatnim **if** w celu określenia wartości wyjść w przypadku, gdy żadne z wyrażeń logicznych nie jest prawdziwe;

- użycie **default** w operatorze **case** w tym samym celu;
- użycie atrybutów *full_case* i *parallel_case* w deklaracji **case**;
- użycie kwalifikatorów **unique** i **priority** w konstrukcjach **if** i **case**.

Kwalifikator **inside** operatora **case** pozwala na określenie dopasowania do jednego z elementów zakresu, który reprezentuje *item_expression*.

Język SystemVerilog dostarcza sześć operatorów do opisu pętli: **forever**, **repeat**, **while**, **for**, **do-while** i **foreach**.

W SystemVerilog zmienne sterujące pętlą **for** mogą być zadeklarowane albo przed nią, albo w pętli **for** jako część konstrukcji *for_initialisation*. W tym przypadku ta sama zmienna może być użyta jednocześnie w dwóch różnych pętlach.

W pętli **for** może być kilka operatorów inicjalizacyjnych, a także kilka operatorów przypisania kroku pętli, wyrażeń inkrementacji lub dekrementacji oraz wywołań funkcji. Każda zmienna pętli może być zadeklarowana z innym typem danych.

Operator pętli **repeat** wykonuje operator *statement* tyle razy, ile wynosi podana liczba lub wyrażenie.

Operator pętli **foreach** iteruje po elementach tablicy. Można pominąć zmienne pętli, aby określić, że nie jest wymagana iteracja na danym wymiarze tablicy.

Operator **while** jest wielokrotnie wykonywany tak długo, jak długo wyrażenie jest prawdziwe. Jeśli wyrażenie nie jest prawdziwe na początku pętli **while**, to operator *statement* nie jest w ogóle wykonywany.

Operator **do-while** sprawdza wyrażenie kontrolne na końcu pętli, więc operator *statement* jest wykonywany co najmniej raz w pętli **do-while**.

Operator pętli **forever** wykonuje operator *statement* wielokrotnie. Aby uniknąć nieskończonej pętli z zerowym opóźnieniem, operator **forever** powinien być używany tylko w połączeniu z elementami do sterowania czasem lub operatorem **disable**.

Język SystemVerilog zapewnia podobne do istniejących w języku C operatory **break**, **continue** i **return**. Język SystemVerilog nie dostarcza operatora **goto** w przeciwieństwie do języka C.

Operator **continue** przeskakuje na koniec pętli i przekazuje sterowanie do następnej iteracji pętli. Operator **break** kończy pętlę, a sterowanie jest przekazywane do operatora występującego za operatorem pętli. Instrukcje **continue** i **break** nie mogą być używane wewnątrz bloku **fork-join**.

Operator **return** zapewnia wyjście z funkcji lub zadania, przy czym może on zwrócić wartość wyrażenia.

13. Zadania i funkcje

Ograniczone możliwości zadań i funkcji w Verilogu są często istotnym wyzwaniem przy tworzeniu dużych projektów. Na przykład w Verilogu zadania i funkcje mogą być zadeklarowane tylko w module, który ich używa; nie można przekazywać argumentów do funkcji, używając nazw, tak jak w instancjach modułów; nie można określić wartości domyślnych dla argumentów zadań i funkcji; nie można przekazywać tablic jako argumentów funkcji; nie jest możliwe przekazanie argumentów przez referencję; funkcje nie mogą mieć innych wartości wyjściowych niż ta zwracana poprzez nazwę funkcji; na zadania i funkcje nie można rezerwować miejsca w dużym projekcie, a następnie precyzować ich podczas rozbudowy projektu; zadania i funkcje nie mogą mieć innych punktów wyjścia niż zakończenie określonej sekwencji operatorów itp.

Język SystemVerilog mocno rozszerza możliwości zadań i funkcji w porównaniu z językiem Verilog, co znacznie upraszcza tworzenie dużych i złożonych projektów.

Przede wszystkim zadania i funkcje, zwane w języku SystemVerilog *podprogramami* (*subroutines*), mogą być deklarowane nie tylko w obrębie modułów, lecz także w pakietach, co pozwala na wykorzystanie podprogramów zadeklarowanych w pakiecie w dowolnym module projektu. W języku SystemVerilog usunięto ograniczenie mówiące o tym, że wszystkie instrukcje podprogramów muszą być grupowane za pomocą bloku **begin...end**. Podprogramy w SystemVerilog mogą być statyczne (domyślnie) lub automatyczne.

Porty podprogramów w SystemVerilog nazywane są *argumentami formalnymi* (*formal arguments*). W podprogramach mogą być one przekazywane nie tylko przez wartość, lecz także przez referencję (jak w językach programowania), co znacznie zmniejsza liczbę kopiowanych argumentów. Jeśli argumenty są przekazywane przez wartość, mogą być przekazywane pozycyjnie lub przez nazwę (jak w instancjach modułów).

Formalnymi argumentami dla podprogramów w SystemVerilog mogą być zarówno sieci i zmienne, jak i tablice, struktury i unie. Podprogramy bez argumentów formalnych są także dozwolone w SystemVerilog.

Aby zwiększyć niezawodność projektów, SystemVerilog wprowadza pojęcie *zmiennej tylko do odczytu* (*read-only variable*), której wartość nie może być zmieniona w zadaniu lub funkcji, co zapobiega przypadkowym zmianom wartości zmiennej w podprogramie.

Każdemu argumentowi formalnemu można nadać wartość domyślną, co pomaga zmniejszyć liczbę wywołań do podprogramu o dużej liczbie argumentów.

Podprogramy mogą nie mieć w ogóle operatorów, czyli być *puste*. Rezerwują wówczas miejsce w kodzie złożonego projektu na najwyższych poziomach, ich zawartość jest następnie wypełniana w trakcie projektowania.

Oprócz tradycyjnego wyjścia z podprogramu po wykonaniu wszystkich instrukcji język SystemVerilog dostarcza instrukcję **return** służącą do zakończenia podprogramu. Jej użycie pozwala na posiadanie przez podprogram wielu punktów wyjścia.

Dzięki językowi SystemVerilog funkcje mają argumenty typu **output** i **inout**. Mogą też mieć kilka wartości zwracanych: wartość związaną z nazwą funkcji, a także wartości argumentów **output** i **inout**.

Język SystemVerilog dopuszcza funkcje typu **void**, które nie zwracają wartości. W rzeczywistości jednak mogą je zwracać poprzez argumenty typu **output** i **inout**. Funkcje typu **void** są wywoływane za pomocą instrukcji, podobnie jak zadania, i pozwalają ominąć ograniczenie języka Verilog, które zabrania wywoływania zadań z funkcji. Język SystemVerilog umożliwia również stosowanie funkcji rekurencyjnych i stałych.

Można stwierdzić, że język SystemVerilog zaciera różnice między zadaniami a funkcjami, rozszerzając możliwości tych ostatnich.

13.1. Informacje ogólne

Zadania i funkcje w SystemVerilog nazywane są *podprogramami* (*subroutines*). Są to jednostki strukturalne SystemVerilog, które pozwalają ustrukturyzować kod źródłowy, aby uczynić go bardziej czytelnym i zwartym. Ponieważ zadania i funkcje mogą być wywoływane wielokrotnie, można w ten sposób skrócić kod źródłowy projektu. Mogą być również używane do ukrywania zmiennych lokalnych. Podobnie jak w przypadku języków programowania, zadania i funkcje są elementami programowania strukturalnego.

Zadania języka SystemVerilog w przybliżeniu przypominają procedury w językach programowania, natomiast funkcje języka SystemVerilog przypominają funkcje języków programowania. Nie należy jednak zapominać, że główną jednostką strukturalną Veriloga jest moduł, a zadania i funkcje są wykorzystywane jako elementy pomocnicze w opisie modułów. W SystemVerilog istnieje również pojęcie *funkcji stałej* (*constant function*).

Przy opisie podprogramów pojedynczy operator proceduralny wszędzie może być zastąpiony blokiem proceduralnym, czyli grupą operatorów umieszczonych w nawiasach operatorowych **begin-end**.

Główna różnica między zadaniem a funkcją polega na tym, że ta ostatnia jest wykonywana w czasie zerowym i nie może zawierać operatorów do zarządzania czasem. Z tego powodu nie może wywołać zadania. Zadanie może zarządzać czasem, więc

może wywoływać inne zadania, a także funkcje. Ponadto zadania są wywoływane za pomocą instrukcji wywołania, która przypomina instrukcję tworzenia instancji modułu, a funkcja zwracająca wartość może być używana w wyrażeniach jako zwykły operand.

Zadania i funkcje mogą być zadeklarowane wewnątrz modułu – wówczas są one elementami lokalnymi w odniesieniu do tego modułu. Zadania i funkcje automatyczne mogą być też deklarowane w pakietach i w takim przypadku stają się one dostępne w dowolnym module w projekcie.

13.2. Zadania i funkcje automatyczne i statyczne

Pojęcie zadań i funkcji automatycznych i statycznych pochodzi z języków programowania i jest związane z działaniem symulatora. Wszystkie zadania i funkcje są domyślnie statyczne. Oznacza to, że dla wszystkich zmiennych (wejściowych, wyjściowych, lokalnych) podczas symulacji zadań i funkcji w pamięci komputera przydzielany jest jednokrotnie statyczny obszar pamięci. Przy każdym dostępie do zadania lub funkcji wartości zmiennych są zapisywane do tego obszaru.

Problemy z pamięcią statyczną pojawiają się przy symulacji procesów równoległych, gdy możliwe jest jednoczesne wykonanie kilku instancji jednego i tego samego zadania albo funkcji. W tym przypadku wartości zmiennych dwóch lub więcej współbieżnie pracujących instancji zadania bądź funkcji będą zapisywane w tej samej przestrzeni pamięci.

Przy zadaniach i funkcjach automatycznych pamięć dla zmiennych jest przydzielana każdorazowo przy wywołaniu zadania lub funkcji i zwalniana po zakończeniu ich pracy. Zastosowanie pamięci automatycznej pozwala na jednoczesne wykonywanie kilku instancji danego zadania albo funkcji. Z punktu widzenia działania symulatora oznacza to, że możliwa staje się symulacja procesów równoległych.

Tak więc jeśli w projekcie dozwolone jest równoległe działanie różnych zadań i wywołań funkcji, to muszą być one zadeklarowane jako automatyczne. We wszystkich innych przypadkach zadania i funkcje mogą być statyczne.

W SystemVerilog zadania i funkcje automatyczne mogą być zadeklarowane jawnie za pomocą słowa kluczowego **automatic** lub *niejawnie* podczas ich definiowania w automatycznym module, interfejsie, programie lub pakiecie. Zadania i funkcje zdefiniowane w klasie są zawsze automatyczne.

Zadania i funkcje automatyczne mogą być wywoływane przy użyciu nazw hierarchicznych, ale nie jest możliwy dostęp do elementów zadania albo funkcji automatycznej przy użyciu nazwy hierarchicznej.

Konkretne zmienne lokalne mogą być zadeklarowane jako automatyczne (**automatic**) w zadaniu lub funkcji statycznej bądź jako statyczne (**static**) w zadaniu lub funkcji automatycznej.

Wszystkie funkcje rekurencyjne muszą być automatyczne.

Przykład 13.1. Funkcja rekurencyjna do obliczania silni

```
function automatic integer factorial (input [31:0] n);
        if (n >= 2)
                factorial = factorial (n - 1) * n;
        else
                factorial = 1;
endfunction: factorial
```

13.3. Zadania

13.3.1. Deklaracja zadań

Zadania SystemVerilog przypominają podprogramy lub procedury w językach programowania. Różnica polega na tym, że języki programowania używają listy parametrów do przekazywania wartości argumentów do podprogramów i zwracania wyników, natomiast zadania SystemVerilog używają listy portów, przez które pewne wartości są przekazywane i zwracane.

Zadania mogą być wywoływane z bloków procedur **initial** i **always**, a także z innych zadań. Mogą mieć dowolną liczbę portów wejściowych, wyjściowych i dwukierunkowych lub nie mieć ich wcale (jak moduły). Mogą też zawierać operatory zarządzania czasem: **#**, **@** lub **wait**.

Deklaracja zadania ma następujący format (styl ANSI):

```
task [automatic] task_name ( port_declaration_list );
        local_variable_declarations
        { statement_or_null }
endtask [: task_name ]
```

gdzie: **task**, **automatic** i **endtask** to słowa kluczowe; *task_name* to nazwa zadania; *port_declaration_list* to lista deklaracji portów; *local_variable_declarations* to deklaracje zmiennych lokalnych; *statement_or_null* to instrukcje proceduralne składające się na ciało zadania. Słowo kluczowe **automatic** i nazwa zadania po słowie kluczowym **endtask** są opcjonalne.

Deklaracja zadania w starym stylu (non-ANSI) ma następujący format:

```
task [automatic] task_name (port_name_list);
        port_declaration_list
        local_variable_declarations
        { statement_or_null }
endtask [: task_name ]
```

gdzie *port_name_list* jest listą nazw portów zadania.

Opcjonalna konstrukcja **automatic** definiuje zadanie z automatycznym przydziałem pamięci, pozwalając na jednoczesne działanie wielu instancji tego samego zadania.

Porty w deklaracji zadania nazywane są *argumentami formalnymi* (*formal arguments*). Dla każdego z nich kierunek i typ danych jest określony w deklaracji.

Każdy argument formalny ma jeden z następujących kierunków:

```
input     // kopiowanie wartości na początku
output    // kopiowanie wartości na końcu
inout     // kopiowanie wartości na początku i na końcu
ref       // odniesienie (referencja) do zmiennej
```

Jeśli nie podano kierunku argumentu, domyślnie wybierane jest wejście **input**. Po określeniu kierunku kolejne argumenty domyślnie mają ten sam kierunek przekazywania.

Każdy argument formalny ma typ danych, który może być jawnie zadeklarowany lub odziedziczony po poprzednim argumencie. Jeśli typ danych nie jest jawnie zadeklarowany, domyślnym typem danych jest **logic**.

Jako formalny argument zadania może być określona tablica. Na przykład:

```
task mytask4 (input [3:0][7:0] a, b[3:0],
              output [3:0][7:0] y[1:0]);
...
endtask
```

W powyższym przykładzie tablica b jest zadeklarowana jako **input** [3:0] [7:0] b [3:0].

Między deklaracją zadania a **endtask** można wpisać kilka operatorów. Są one wykonywane sekwencyjnie, tak jakby były zamknięte w bloku **begin...end**. Dopuszczalny jest również brak operatorów (w przypadku zadania *pustego*).

Zadanie zostaje zakończone po osiągnięciu słowa kluczowego **endtask**. Aby zakończyć zadanie, przed słowem kluczowym **endtask** można użyć operatora **return**.

13.3.2. Wywołanie zadań

Wywołanie zadania w SystemVerilog jest również nazywane *uaktywnieniem zadania* (*task enable*). Wywołanie (aktywacja) zadania ma następujący uogólniony format:

task_name (*task_call_arguments*);

gdzie *task_name* jest nazwą zadania, a *task_call_arguments* są oddzielonymi przecinkami argumentami wywołania zadania.

Poniższy przykład ilustruje stary styl definicji zadania z pięcioma argumentami:

```
task my_task;
        input a, b;
        inout c;
        output d, e;
        . . .                   // operatory zadania
        c = a;                  // przypisania, które inicjalizują wynik (wyjścia zadania)
        d = b;
        e = c;
endtask
```

Korzystając z nowego stylu, deklaracja zadania ma następujący wygląd:

```
task my_task (input a, b, inout c, output d, e);
        . . .                   // operatory zadania
        c = a;                  // przypisania, które inicjalizują wynik
        d = b;
        e = c;
endtask
```

Poniższy operator wywołuje zadanie:

```
initial
my_task (v, w, x, y, z);
```

Argumenty wywołania zadania (v, w, x, y i z) odpowiadają argumentom (a, b, c, d i e) zdefiniowanym przez zadanie. Podczas wywołania argumenty typu **input** i **inout** (a, b i c) otrzymują wartości przekazane w zmiennych v, w i x. Tak więc wykonanie połączenia faktycznie skutkuje następującymi przypisaniami:

```
a = v;
b = w;
c = x;
```

Podczas wykonywania zadania obliczone wartości wyników są umieszczane w argumentach formalnych c, d i e. Po zakończeniu zadania, aby zwrócić obliczone wartości do procesu wywołującego, wykonywane są następujące zadania:

```
x = c;
y = d;
z = e;
```

13.4. Funkcje

Głównym celem działania funkcji jest zwrócenie wartości, która ma być użyta w wyrażeniu. Funkcja typu **void** służy do zdefiniowania podprogramu, który wykonuje się i zwraca pewną wartość w ramach jednego kroku czasowego.

Zakłada się, że funkcje zawsze wykonują się w zerowym czasie symulacji i nie zawieszają procesu, który je uruchamia. Nakłada to następujące ograniczenia na wykorzystanie funkcji:

- funkcja nie może zawierać operatorów zarządzania czasem, tj. żadnych #, ##, @, **fork-join**, **fork-join_any**, **wait**, **wait_order** lub **expect**;
- funkcja nie może wywoływać zadań niezależnie od tego, czy zadania te zawierają operatory zarządzania czasem.

Funkcje mogą jednak wykorzystywać metody *precyzyjnego sterowania procesami* (patrz podrozdział 11.4.5) do wstrzymywania własnego lub innego procesu.

13.4.1. Deklaracja funkcji

Ogólna składnia deklaracji funkcji w nowym stylu ANSI jest następująca:

function [automatic] *function_data_type function_name* (*port_declaration_list*);
    local_variable_declarations
    { *statement_or_null* }
endfunction [: *function_name*]

gdzie: *function_data_type* to typ wartości zwracanej lub słowo kluczowe **void**; *function_name* to nazwa funkcji; *port_declaration_list* to lista deklaracji portów; *local_variable_declarations* to deklaracje zmiennych lokalnych; *statement_or_null* to zbiór operatorów proceduralnych.

Deklaracja funkcji w starym stylu (non-ANSI) ma następujący format:

function [automatic] *function_data_type function_name* (*port_name_list*);
    port_declaration_list
    local_variable_declarations
    { *statement_or_null* }
endfunction [: *function_name*]

gdzie *port_name_list* jest listą nazw portów funkcji.

Należy pamiętać, że funkcje w języku SystemVerilog mogą mieć takie same argumenty formalne jak zadania, tzn. kierunki przekazywania argumentów funkcji mogą być określone jako: **input**, **output**, **inout** lub **ref**. Jeśli nie zostanie określony kierunek argumentu formalnego, domyślnie przyjmuje się **input**.

Każdy argument formalny ma typ danych, który może być jawnie zadeklarowany lub odziedziczony po poprzednim argumencie. Jeśli typ danych nie jest jawnie zadeklarowany, domyślnym typem danych jest **logic**. W przeciwnym razie typ danych jest dziedziczony z poprzedniego argumentu. Ponadto także tablica może być określona jako argument formalny funkcji. Na przykład:

```
function [3:0][7:0] myfunc(input [3:0][7:0] a, b[3:0]);
        ...
endfunction
```

Między nagłówkiem funkcji a słowem kluczowym **endfunction** może być umieszczonych kilka operatorów. Są one wykonywane sekwencyjnie, tak jakby były zamknięte w bloku **begin-end**. Dopuszczalny jest również brak operatorów (dla funkcji *pustej*), w takim przypadku funkcja zwraca aktualną wartość zmiennej niejawnej, która ma taką samą nazwę jak funkcja.

13.4.2. Wartości zwracane

Definicja funkcji niejawnie deklaruje zmienną wewnętrzną o tej samej nazwie co funkcja. Zmienna ta ma taki sam typ jak wartość zwracana funkcji. Wartości te można określić na dwa sposoby: poprzez użycie operatora **return** lub przypisanie wartości do zmiennej wewnętrznej o tej samej nazwie co funkcja. Na przykład:

```
function [15:0] myfunc1 (input [7:0] x,y);
        myfunc1 = x * y - 1;            // wartość zwracana, przypisanie do nazwy funkcji
endfunction

function [15:0] myfunc2 (input [7:0] x,y);
        return x * y - 1;               // zwracana wartość jest określana
                                        // za pomocą instrukcji return
endfunction
```

Operator **return** ma pierwszeństwo przed przypisaniem wartości do nazwy funkcji, nadpisuje on wszelkie wartości przypisane do nazwy funkcji.

Funkcje SystemVerilog mogą zwracać strukturę lub unię. Wówczas nazwa hierarchiczna używana wewnątrz funkcji i rozpoczynająca się od nazwy funkcji jest interpretowana jako człon wartości zwracanej.

W SystemVerilog, inaczej niż w Verilogu, funkcja może nie mieć wartości zwracanej. W tym przypadku typem wartości zwracanej jest **void**. Funkcje typu **void** są wywoływane jako zadania za pomocą operatora wywołania. Na przykład:

```
function void myprint (int a);
        ...
endfunction

myprint(a);                    // wywołaj funkcję myprint jako operator
```

Funkcje zwracające wartości mogą być użyte w przypisaniu lub wyrażeniu. Na przykład:

```
a = b + myfunc1 (c, d);        // użycie funkcji w wyrażeniu jako operandu
```

W języku SystemVerilog funkcja zwracająca wartość może być również użyta jako operator wywołania. Wartość zwracana w tym przypadku jest odrzucana przez użycie operacji konwersji typu, aby rzutować wartość zwracaną na typ **void**. Na przykład:

```
void '(some_function ());
```

13.4.3. Argumenty funkcji typu output i inout

Język SystemVerilog rozszerza możliwości języka Verilog o możliwość posiadania przez funkcje argumentów typu **output** i **inout**. Dzięki temu funkcja może mieć dowolną liczbę wyjść oprócz wartości zwracanej przez nazwę funkcji.

W poniższym przykładzie funkcja *add* oprócz sumy zwraca flagę *overflow*, która wskazuje na przepełnienie podczas wykonywania operacji dodawania.

Przykład 13.2. Funkcja sumy zwracająca flagę przepełnienia na liście argumentów

```
function [7:0] add
        ( input [7:0] a, b,
          output overflow );
        { overflow, add } = a + b;
endfunction
```

Aby zapobiec niepożądanym efektom, SystemVerilog ogranicza miejsca, z których można wywołać funkcje z argumentami **output** lub **inout**. Nie mogą one być wywołane z:

- wyrażenia zdarzenia;

- wyrażenia w ramach proceduralnego przypisania ciągłego;
- wyrażenia, które nie znajduje się wewnątrz operatora proceduralnego.

13.4.4. Funkcje typu void

W języku SystemVerilog typem wartości zwracanej przez funkcję może być typ **void**. Oznacza to, że z nazwą funkcji nie jest związana żadna wartość zwracana. Jednocześnie funkcja **void** może zwracać wartości za pomocą argumentów typu **output** i **inout**.

Funkcje typu **void**, podobnie jak zadania, są wywoływane za pomocą operatorów wywołania. Zamiast wywoływać zadania, funkcje mogą wywoływać funkcje typu **void** i zwracać wymagane wartości za pomocą argumentów typów **output** i **inout**.

W ten sposób można ominąć ograniczenie polegające na tym, że funkcje nie mogą wywoływać zadań (w tym przypadku funkcje typu **void** odgrywają rolę zadań).

Jednak funkcje **void** nadal mają ograniczenia dotyczące używania deklaracji zarządzania czasem, zdarzeń i przypisań nieblokujących.

W poniższym przykładzie funkcja *fill_packet* typu **void** zwraca strukturę, w której określa wartości na szynie danych, definiuje bity kontroli parzystości *check* i ustawia flagę *valid*.

Przykład 13.3. Funkcja **void** zwracająca strukturę

```
typedef struct {                    // deklaracja struktury w pakiecie packet_t
        logic valid;                // flaga wskazuje na obliczenie bitu parzystości
        logic [ 7:0] check;         // bity parzystości
        logic [63:0] data;          // szyna danych data
} packet_t;

function void fill_packet (         // deklaracja funkcji fill_packet
        input logic [63:0] data_in, // wejściowa szyna
        output packet_t data_out ); // wyjściem funkcji jest struktura, zadeklarowana
                                    // w pakiecie packet_t
        data_out.data = data_in;    // określenie wartości na szynie danych data

        for (int i=0; i<=7; i++)    // obliczenie bitu parzystości
                data_out.check[i] = ^data_in[(8*i)+:8];

        data_out.valid = 1;         // ustawienie flagi valid
endfunction
```

13.4.5. Funkcje rekurencyjne

Funkcja *rekurencyjna* (*recursive*) to funkcja, która wywołuje samą siebie. Język SystemVerilog pozwala na stosowanie funkcji rekurencyjnych zarówno w syntezie, jak i symulacji. Wcześniej, w przykładzie 13.1, rozważaliśmy funkcję rekurencyjną, która oblicza silnię liczby.

Aby funkcja mogła wywoływać samą siebie, musi być zadeklarowana jako automatyczna, tzn. funkcje rekurencyjne muszą być zadeklarowane z kwalifikatorem **automatic**. Każda funkcja rekurencyjna musi zawierać warunek zakończenia, który powinien być określony w jej ciele (inaczej otrzymamy pętlę nieskończoną).

Do implementacji funkcji rekurencyjnych użyteczny jest operator **return**, który w niektórych przypadkach upraszcza weryfikację wyjścia funkcji.

Jako przykład rozważmy funkcję log2 obliczającą logarytm przy podstawie 2.

Przykład 13.4. Funkcja rekurencyjna obliczająca logarytm o podstawie 2 bez operatora **return**

```
function automatic int log2 (input int n);
        if (n <=1)                      // wyznaczanie wartości funkcji, gdy n <=1
                log2 = 1;
        else begin                      // wyznaczanie wartości funkcji, gdy n > 1
                log2 = 0;
                while (n > 1) begin
                        n = n/2;
                        log2 = log2+1;  // wywołanie funkcji rekurencyjnej
                end
        end
endfunction
```

Użycie operatora **return** znacznie upraszcza poniższy kod.

Przykład 13.5. Funkcja rekurencyjna obliczająca logarytm o podstawie 2 z operatorem **return**

```
function automatic int log2 (input int n);
        if (n <=1) return 1;            // wyznaczanie wartości funkcji, gdy n <=1
        log2 = 0;                       // wyznaczanie wartości funkcji, gdy n > 1
        while (n > 1) begin
                n = n/2;
                log2++;                 // wywołanie funkcji rekurencyjnej
        end
endfunction
```

13.5. Funkcje stałe

Funkcje stałe w SystemVerilog są podobne do funkcji preprocesora w języku programowania. Innymi słowy, funkcja stała musi pozwolić na jej wykonanie (i zwrócić jakąś wartość) przed uruchomieniem symulatora. Stąd szereg ograniczeń w deklarowaniu i używaniu funkcji stałych w porównaniu z normalnymi funkcjami języka SystemVerilog:

- funkcja stała nie może mieć argumentów typu **output**, **inout** lub **ref**;
- funkcja **void** nie może być funkcją stałą;
- funkcja stała nie może zawierać konstrukcji **fork** ani odniesień hierarchicznych;
- funkcja stała może wywoływać tylko funkcję stałą;
- funkcja stała może wywołać dowolną funkcję systemową, która jest dozwolona w *wyrażeniach stałych* (*constant expressions*);
- wszystkie wartości parametrów używane w funkcji muszą być zdefiniowane przed wywołaniem funkcji stałej;
- funkcje stałe mogą odwoływać się do parametrów zdefiniowanych w pakietach i jednostce kompilacji;
- wszystkie identyfikatory w funkcji stałej są deklarowane lokalnie;
- funkcje stałe nie powinny być deklarowane wewnątrz jednostki generującej;
- funkcje stałe nie powinny używać funkcji stałej w kontekście, który wymaga stałego wyrażenia;
- funkcja stała nie może mieć domyślnych wartości argumentów.

Jako przykład rozważmy użycie stałej funkcji clogb2, która zwraca najmniejszą liczbę całkowitą większą lub równą log2n. Funkcja clogb2 służy do określenia szerokości magistrali adresu pamięci, mając daną liczbę słów pamięci (głębokość pamięci).

Przykład 13.6. Jednoportowa pamięć synchroniczna z wykorzystaniem funkcji stałej *clogb2*

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

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

        /* funkcja stała clogb2, która wykorzystuje rozmiar przestrzeni adresowej ram_depth
        oblicza odpowiednią 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 macierzy pamięci

    always @(posedge clk) begin                  // pamięć synchroniczna
        if (r_w) data = RAM_mem[address];        // odczyt z pamięci
        else RAM_mem[address] = data;            // zapis do pamięci
    end
endmodule
```

13.6. Wywołania podprogramów i przekazywanie argumentów

W języku SystemVerilog zadania i funkcje **void** są wywoływane za pomocą operatorów wywołania. Uogólniona składnia operatora wywołania podprogramu jest następująca:

subroutine_name (*argument_list*);

gdzie *subroutine_name* jest nazwą podprogramu, a *argument_list* jest listą argumentów.

Jeśli argument w podprogramie jest zadeklarowany jako wejście **input**, to odpowiadające mu wyrażenie w wywołaniu podprogramu może być dowolne. Jeśli argument w podprogramie jest zadeklarowany jako **output** lub **inout**, to odpowiadająca mu zmienna w wywołaniu podprogramu może być użyta tylko w lewej stronie przypisania proceduralnego.

Wywołanie podprogramu przekazuje wartości wejściowe z wyrażeń wymienionych w argumentach wywołania. Powrót z podprogramu przekazuje wartości z argumentów typu **output** i **inout** do odpowiednich zmiennych użytych przy wywołaniu podprogramu.

Język SystemVerilog zapewnia dwa sposoby przekazywania argumentów do zadań i funkcji: *przez wartość* (*by value*) i *przez referencję* (*by reference*).

Podczas wywoływania zadania lub funkcji argumenty mogą być powiązane ze zmiennymi i wyrażeniami zarówno *przez nazwę* (*by name*), jak i *przez pozycję* (*by position*). Dodatkowo argumenty zadań bądź funkcji mogą mieć wartości domyślne, co zmniejsza liczbę wartości przekazywanych w wywołaniu.

13.6.1. Przekazywanie argumentów przez wartość

Tradycyjnie argumenty są przekazywane do podprogramów przez wartość. Wartość każdego argumentu jest kopiowana do obszaru podprogramu. Podprogramy automatyczne przechowują lokalne kopie argumentów na swoim stosie. W przypadku dużych argumentów takie kopiowanie może nie być pożądane.

Na przykład wywołanie następującej funkcji *crc*, która oblicza sumę kontrolną pakietu *packet* typu tablicowego, kopiuje 1000 bajtów przy każdym jej wywołaniu.

```
function automatic int crc( byte packet [1000:1] );
        for( int j= 1; j <= 1000; j++ ) begin
                crc ^= packet[j];
        end
endfunction
```

W takich przypadkach język SystemVerilog zapewnia możliwość przekazywania wartości argumentów przez referencję.

13.6.2. Przekazywanie argumentów przez referencję

Przy przekazywaniu argumentów przez referencje (odniesienie) wartości argumentów nie są kopiowane do obszaru podprogramu – przekazywane są do niego tylko referencje do oryginalnych argumentów. Argumenty przekazywane przez referencję muszą odpowiadać równoważnym typom danych.

Aby określić argument przekazywany przez referencję, jego deklarację poprzedza słowo kluczowe **ref**. Uogólniona składnia dla argumentu przekazywanego przez referencję jest następująca:

subroutine (**ref** *type argument*);

gdzie: *subroutine* jest nazwą podprogramu; **ref** jest słowem kluczowym; *type* jest typem danych argumentu; *argument* jest nazwą argumentu.

Poprzedni przykład można zapisać w następujący sposób:

```
function automatic int crc( ref byte packet [1000:1] );
        for( int j= 1; j <= 1000; j++ ) begin
                crc ^= packet[j];
        end
endfunction
```

Zauważmy, że wywołanie podprogramu z argumentem przekazanym przez referencję lub przez wartość pozostaje bez zmian. Na przykład wywołanie dowolnej wersji funkcji *crc* pozostaje takie same:

```
byte packet1[1000:1];
int k = crc( packet1 );          // wywołanie jest takie same
```

Zmiany dokonane w argumencie przez podprogram są widoczne poza podprogramem tuż przed jego zakończeniem.

Przez referencję do podprogramu można przekazać:

- zmienne;
- elementy rozpakowanej tablicy lub rozpakowanej struktury;
- właściwości klasy.

Przez referencję do podprogramu nie można przekazywać sieci, wyboru bitów sieci lub wyboru części sieci. Argumenty nie mogą być przekazywane przez referencję do podprogramów statycznych.

Zmienna przekazana przez referencję jest zmienną automatyczną, dlatego kwalifikator **ref** nie może być używany w kontekstach, które są zabronione dla zmiennych automatycznych.

Kwalifikator **ref** nie może być łączony z kwalifikatorami **input**, **output** lub **inout**. Na przykład następująca deklaracja spowoduje błąd kompilacji:

```
task automatic incr( ref input int a );          // błąd: ref i input nie mogą istnieć razem
```

Aby chronić argumenty przekazywane przez referencję przed zmianą przez podprogram, wraz z **ref używany jest kwalifikator const**, który wskazuje, że argument jest *zmienną tylko do odczytu*. Na przykład:

```
task automatic show ( const ref byte data [] );
      for ( int j = 0; j < data.size ; j++ )
            $display( data[j] );      // dane mogą być odczytywane, ale nie zapisywane
endtask
```

Próba zmiany argumentu zadeklarowanego jako **const ref** w podprogramie spowoduje błąd kompilacji.

13.6.3. Wartości domyślne argumentów

Język SystemVerilog pozwala na określenie wartości domyślnej dla każdego argumentu w deklaracji podprogramu. Składnia deklaracji argumentu domyślnego w podprogramie jest następująca:

subroutine([*direction*] [*type*] *argument* = *expression*);

gdzie: *subroutine* jest nazwą podprogramu; *direction* jest kierunkiem określonym przez jedno ze słów kluczowych **input**, **output**, **inout** lub **ref**; *type* jest typem danych; *argument* jest nazwą argumentu; *expression* jest wyrażeniem, którego wartość jest domyślnie przypisana do argumentu.

Jeśli użyto argumentu domyślnego, wyrażenie *expression* jest obliczane przy każdym wywołaniu podprogramu.

Jeśli wartość domyślna nie jest używana, wyrażenie domyślne nie jest obliczane. Używanie wartości domyślnych jest dozwolone tylko w przypadku deklaracji ANSI (nowy styl).

Przy wywoływaniu podprogramu można pominąć argumenty o wartościach domyślnych. Na przykład:

```
task read(int j = 0, int k, int data = 1 );
...
endtask
```

Powyższy przykład zawiera deklarację zadania read() z dwoma domyślnymi argumentami, *j* i *data*. Zadanie może być następnie wywołane z różnymi domyślnymi argumentami:

```
read( , 5 );          // równoważne read (0, 5, 1);
read( 2, 5 );         // równoważne read (2, 5, 1);
read( , 5, );         // równoważne read (0, 5, 1);
read( , 5, 7 );       // równoważne read (0, 5, 7);
read( 1, 5, 2 );      // równoważne read (1, 5, 2);
read( );              // błąd; k nie ma wartości domyślnej
read( 1, , 7 );       // błąd; k nie ma wartości domyślnej
```

13.6.4. Przekazywanie argumentów przez nazwę

Podobnie jak instancje modułów, z wyjątkiem przekazywania argumentów *przez pozycję* (*by position*), argumenty funkcji mogą być również przekazywane *przez nazwę* (*by name*). Dzięki temu odbywa się to w dowolnej kolejności. Na przykład:

```
function int fun( int j = 1, string s = "no" );
        ...
endfunction
```

Funkcję fun można wywołać w następujący sposób:

```
fun( .j(2), .s(„yes") );      // równoważne fun( 2, „tak" );
fun( .s(„yes") );             // równoważne fun( 1, „tak" );
fun( , „yes" );               // równoważne fun( 1, „tak" );
fun( .j(2) );                 // równoważne fun( 2, „nie" );
fun( .s(„yes"), .j(2) );      // równoważne fun( 2, „tak" );
fun( .s(), .j() );            // równoważne fun( 1, „nie" );
```

```
fun( 2 );                    // równoważne fun( 2, „nie” );
fun( );                      // równoważne fun( 1, „nie” );
```

Jeśli w tym samym wywołaniu podprogramu chcemy podać argumenty zarówno przez pozycję, jak i przez nazwę, to wszystkie argumenty podawane przez pozycję muszą poprzedzać argumenty podawane przez nazwę. Na przykład:

```
fun( .s(„yes”), 2 );         // niedopuszczalne
fun( 2, .s(„yes”) );         // dopuszczalne
```

13.6.5. Przekazywanie tablic, struktur i unii jako argumentów

W SystemVerilog argumenty podprogramów mogą być spakowanymi lub rozpakowanymi tablicami bądź strukturami oraz rozpakowanymi i oznaczonymi uniami. Gdy spakowane tablice są przekazywane jako argumenty, są one traktowane jako wektory: jeśli rozmiar spakowanej tablicy nie odpowiada rozmiarowi jej argumentu, wektor jest obcinany lub rozszerzany zgodnie z zasadami przypisywania wektorów. W przypadku tablic rozpakowanych argument musi dokładnie odpowiadać typowi elementów tablicy: argumenty wywołujący i formalny muszą mieć tę samą liczbę wymiarów tablicy, te same rozmiary zakresów i ten sam rozmiar każdego elementu.

Na przykład poniższy przykład pokazuje użycie rozpakowanej tablicy i rozpakowanej struktury jako formalnego argumentu dla funkcji, która oblicza sumę kontrolną danych.

```
typedef struct {                    // deklaracja struktury
        logic valid;                // flaga
        logic [ 7:0] check;         // suma kontrolna
        logic [63:0] data;          // dane
} packet_t;                         // jako typ użytkownika packet_t

function void fill_packet (         // deklaracja funkcji void
        input logic [7:0] data_in [0:7],   // tablica jako argument wejściowy
        output packet_t data_out );        // struktura jako argument wyjściowy

        for (int i=0; i<=7; i++) begin
                data_out.data[(8*i)+:8] = data_in[i];  // przepisanie danych
                data_out.check[i] = ^data_in[i];       // obliczenie sumy kontrolnej
        end
        data_out.valid = 1;         // ustawienie flagi
endfunction
```

13.7. Puste zadania i funkcje

Język Verilog wymaga, aby zadania i funkcje zawierały przynajmniej jeden operator (który może być pustą grupą operatorów pomiędzy słowami **begin...end**). W SystemVerilog zadania i funkcje mogą być całkowicie puste – bez operatorów lub grup operatorów. Funkcja pusta zwraca aktualną wartość zmiennej niejawnej reprezentującej nazwę funkcji.

Puste zadanie albo funkcja to symbol zastępczy, który opisuje zarezerwowane miejsce (*placeholder*) dla tworzonego kodu. W projektowaniu odgórnym (*top-down*) utworzenie pustego zadania lub funkcji może służyć do wskazania miejsca, w którym funkcjonalność zostanie szczegółowo określona w dalszej części procesu projektowania.

13.8. Zadania i funkcje parametryzowane

Język SystemVerilog umożliwia tworzenie zadań i funkcji sparametryzowanych znanych również jako *podprogramy sparametryzowane*. Pozwalają one użytkownikowi zdefiniować implementację na warunkach ogólnych. Przy zastosowaniu sparametryzowanego podprogramu możliwe jest podanie parametrów, które określają jego zachowanie. Pozwala to na napisanie i utrzymanie tylko jednej definicji podprogramu zamiast kilku podprogramów z różnymi rozmiarami tablic, typami danych i zmiennymi szerokościami wektorów.

Sposobem na implementację sparametryzowanych podprogramów jest użycie metod statycznych w sparametryzowanych klasach. Ponieważ klasy nie są obsługiwane przy syntezie, zadania i funkcje sparametryzowane również nie są obsługiwane przez narzędzia do syntezy.

13.9. Porównanie zadań i funkcji

W SystemVerilog nie ma tak ścisłego podziału na zadania i funkcje, jaki jest spotykany w Verilogu czy w językach programowania. Różnice między zadaniami i funkcjami w SystemVerilog zostały zatarte poprzez rozszerzenie możliwości funkcji:

- funkcje mogą mieć więcej niż jedną wartość zwracaną dzięki argumentom typu **output** i **inout**;
- funkcje **void** mogą być wywoływane za pomocą operatora wywołania, podobnie jak zadania;
- funkcje mogą się odnosić do funkcji **void**, które służą jako zadania;
- funkcje **void** nie mogą mieć żadnych argumentów.

Jednocześnie w języku SystemVerilog zachowano rozróżnienie między zadaniami i funkcjami, co przedstawiono w tabeli 13.1.

TABELA 13.1. Porównanie funkcji i zadań

Kategoria	Funkcja	Zadanie
Wywołanie (aktywacja)	Możliwe jest wywołanie funkcji wewnątrz wyrażenia, a wartość zwrotna może być użyta w tym wyrażeniu. Możliwe jest wywołanie funkcji w operatorach przypisania ciągłego **assign** oraz operatorach proceduralnych. Funkcja **void** może być wywołana jako oddzielny operator proceduralny	Wykonywane jako osobny operator proceduralny. Odwołanie się do zadania nie może być wykonane w operatorach przypisania ciągłego
Odwoływanie się do innych zadań i funkcji	W ciele funkcji można odwoływać się do innych funkcji, w tym do siebie (rekurencyjnie). Nie można jednak odwoływać się do zadań. Funkcje mogą odwoływać się do funkcji typu **void** jako do zadań za pomocą oddzielnego operatora proceduralnego	Zadania mogą odwoływać się do funkcji, jak również do innych zadań
Wejścia i wyjścia	Funkcja, podobnie jak zadanie, może mieć dowolną liczbę argumentów formalnych, takich typów jak **input**, **output**, **inout** i **ref**, a także nie mieć żadnych argumentów	Zadanie może mieć dowolną liczbę argumentów formalnych, takich jak **input**, **output**, **inout** i **ref**, lub w ogóle nie mieć żadnych argumentów
Proceduralne zarządzanie czasem	Funkcje nie mogą zarządzać czasem proceduralnym, tzn. nie mogą zawierać operatorów **#**, **@** i **wait**	Zadania mogą zarządzać czasem proceduralnym, tzn. mogą zawierać operatory **#**, **@** i **wait**
Operatory przypisania nieblokującego	Funkcje nie mogą zawierać operatora przypisania nieblokującego „<=”	Zadania mogą zawierać operator przypisania nieblokującego „<=”

ŹRÓDŁO: oprac. własne.

Podsumowując porównanie zadań i funkcji, można stwierdzić, że funkcje są wygodniejsze przy opisie układów kombinacyjnych, a zadania przy opisie układów sekwencyjnych.

13.10. Zadania i funkcje systemowe

Standard języka SystemVerilog opisuje wiele zadań i funkcji systemowych. Jednak większość z nich jest obsługiwana tylko podczas symulacji. Każdy konkretny kompilator języka SystemVerilog obsługuje swój własny podzbiór zadań i funkcji systemowych.

Dlatego przed ich zastosowaniem (do syntezy) należy zapoznać się z odpowiednią dokumentacją używanego oprogramowania, aby dowiedzieć się, jakie systemowe zadania i funkcje są obsługiwane przez ten konkretny kompilator.

Wszystkie systemowe zadania i funkcje podzielone są na grupy według wykonywanych czynności i celów wykorzystania. Dokonajmy krótkiego przeglądu niektórych zadań i funkcji systemowych, które są obsługiwane przez kompilator Quartus.

Funkcje systemowe do pracy z tablicami są omówione w podrozdziale 7.4.1.

Funkcja systemowa określająca rozmiar wyrażeń **$bits** została omówiona w podrozdziale 7.4.2.

13.10.1. Funkcje do obsługi wyrażeń bitowych

Język SystemVerilog udostępnia funkcję systemową **$countbits** do obsługi wektorów bitowych. Składnia funkcji **$countbits**:

$countbits (*expression*, *control_bit_list*)

gdzie *expression* jest wyrażeniem, a *control_bit_list* jest listą bitów sterujących.

Funkcja **$countbits** liczy liczbę bitów w wektorze bitowym, które mają określone wartości: 0, 1, x lub z. Funkcja **$countbits** zwraca wartość typu **int** równą liczbie bitów w wyrażeniu *expression*, których wartości odpowiadają liście bitów sterujących. Na przykład:

```
$countbits (expression, '1)          // zwraca liczbę bitów w expression,
                                     // które mają wartość 1
$countbits (expression, '1, '0)      // zwraca liczbę bitów w expression,
                                     // które mają wartość 1 lub 0
$countbits (expression, 'x, 'z)      // zwraca liczbę bitów w expression,
                                     // które mają wartości x lub z.
```

Dla ułatwienia korzystania z funkcji **$countbits** w języku SystemVerilog przewidziano następujące funkcje systemowe:

- **$countones** (*expression*) jest równoważne **$countbits** (*expression*, ‘1);
- **$onehot** (*expression*) zwraca 1’b1 (true), jeśli **$countbits** (*expression*, ‘1) == 1, w przeciwnym razie zwraca 1’b0 (false);
- **$onehot0** (*expression*) zwraca 1’b1, jeśli **$countbits** (*expression*, ‘1) <= 1, w przeciwnym razie zwraca 1’b0;
- **$isunknown** (*expression*) zwraca 1’b1, jeśli **$countbits** (*expression*, ‘x, ‘z) != 0, w przeciwnym razie zwraca 1’b0.

Funkcje te mogą być używane jako wyrażenia stałe, jeśli wszystkie ich argumenty są również wyrażeniami stałymi.

13.11. Wnioski

Zadania i funkcje są jednostkami strukturalnymi języka SystemVerilog i nazywane są *podprogramami* (*subroutines*). Mogą być używane do redukcji kodu źródłowego projektu oraz do uczynienia go bardziej czytelnym, a także do ukrywania zmiennych lokalnych.

W języku SystemVerilog istnieje również pojęcie funkcji stałej, która pracuje na stałych i zwraca wartość stałą.

Funkcja w SystemVerilog jest wykonywana w czasie zerowym i nie może zawierać operatorów zarządzania czasem. Dlatego funkcje nie mogą wywoływać zadań. Zadanie może kontrolować czas, może wywoływać inne zadania, a także funkcje. Zadania są wywoływane za pomocą instrukcji wywołania, która przypomina instrukcję tworzenia instancji modułu, a funkcja może być używana w wyrażeniach jako zwykły operand.

Wszystkie zadania i funkcje są domyślnie statyczne. W zadaniach i funkcjach statycznych symulator przydziela pamięć raz dla wszystkich zmiennych (wejściowych, wyjściowych i lokalnych). W zadaniach i funkcjach automatycznych pamięć dla zmiennych jest przydzielana przy każdym wywołaniu zadania lub funkcji i zwalniana po zakończeniu zadania bądź funkcji.

Jeśli w projekcie dozwolone jest równoległe uruchamianie różnych zadań i wywołań funkcji, muszą być one zadeklarowane jako automatyczne. We wszystkich innych przypadkach zadania i funkcje mogą być statyczne. Wszystkie funkcje rekurencyjne muszą być automatyczne.

Zmienne lokalne mogą być zadeklarowane jako automatyczne (**automatic**) w zadaniu lub funkcji statycznej albo jako statyczne (**static**) w zadaniu bądź funkcji automatycznej.

Zadania i funkcje mogą być zadeklarowane wewnątrz modułu i wówczas są elementami lokalnymi w odniesieniu do tego modułu. Automatyczne zadania i funkcje mogą być również zadeklarowane w pakietach, w takim przypadku stają się dostępne w dowolnym module projektu.

Zadania mogą być wywoływane z bloków procedur **initial** i **always**, a także z innych zadań. Mogą mieć dowolną liczbę portów wejściowych, wyjściowych i dwukierunkowych lub nie mieć ich wcale. Mogą zawierać operatory zarządzania czasem: **#**, **@** lub **wait**.

Porty w deklaracji zadania nazywane są argumentami formalnymi. Dla każdego argumentu formalnego kierunek i typ danych są określone w deklaracji. Domyślnie typ danych to **logic**, a kierunek to **input**.

Operatory w zadaniach i funkcjach są wykonywane sekwencyjnie, tak jakby były zamknięte w bloku **begin...end**.

Zadanie jest zakończone po osiągnięciu słowa kluczowego **endtask** lub po wywołaniu operatora **return**.

Głównym celem funkcji jest zwrócenie wartości, która ma być użyta w wyrażeniu. Zamiast zadania można też użyć funkcji **void** do zdefiniowania podprogramu, który jest wykonywany i zwraca jakąś wartość w ciągu jednego kroku czasowego.

Funkcja jest zawsze wykonywana w zerowym czasie symulacji i nie zawiesza procesu, który ją uruchamia. Dlatego funkcje nie mogą zawierać operatorów zarządzania czasem i wywoływać zadań.

Funkcje w SystemVerilog mogą mieć takie same argumenty formalne jak zadania. Tablica może być określona jako argument formalny funkcji.

Funkcja może nie mieć w ogóle operatorów. Zwraca wówczas aktualną wartość zmiennej niejawnej, która ma taką samą nazwę jak funkcja.

Wartości zwracane przez funkcję można określić na dwa sposoby: za pomocą operatora **return** lub przypisując wartość do zmiennej wewnętrznej o tej samej nazwie co funkcja. Operator **return** ma pierwszeństwo przed przypisaniem wartości do nazwy funkcji. Obiektem zwracanym przez funkcję może być struktura lub unia.

Funkcje typu **void** nie mają wartości zwrotnej i mogą być wywoływane jako zadania z innych funkcji za pomocą operatora wywołania. Pozwala to obejść ograniczenie polegające na tym, że funkcje nie mogą wywoływać zadań.

W SystemVerilog funkcja, która zwraca wartość, może być również używana jako operator wywołania. W tym przypadku wartość zwracana jest odrzucana przez zastosowanie operacji rzutowania na typ **void**.

W języku SystemVerilog funkcje mogą mieć argumenty typu **output** i **inout**. Dzięki temu funkcja może mieć dowolną liczbę wyjść oprócz wartości zwracanej przez nazwę funkcji.

Funkcja *rekurencyjna* to funkcja, która wywołuje samą siebie. Musi być zadeklarowana jako automatyczna. Musi też zawierać warunek jej zakończenia, czyli wyjścia z funkcji (inaczej otrzymamy nieskończoną pętlę). Do implementacji funkcji rekurencyjnych w języku SystemVerilog wygodnie jest zastosować operator **return**.

Funkcje stałe języka SystemVerilog są podobne do funkcji preprocesora w języku programowania. Stąd szereg ograniczeń w deklarowaniu i używaniu funkcji stałych, w porównaniu ze zwykłymi funkcjami języka SystemVerilog.

Język SystemVerilog zapewnia dwa sposoby przekazywania argumentów do zadań i funkcji: *przez wartość* (*by value*) i *przez referencję* (*by reference*). Podczas wywoływania zadania lub funkcji argumenty mogą być powiązane ze zmiennymi i wyrażeniami *przez nazwę* (*by name*) bądź *przez pozycję* (*by position*). Argumenty zadań albo funkcji w języku SystemVerilog mogą mieć wartości domyślne.

Aby określić argument przekazywany przez referencję, używane jest słowo kluczowe **ref**.

Wywołanie podprogramu z argumentem przekazanym przez referencję lub przez wartość pozostaje niezmienione.

Zmienne, elementy rozpakowanej tablicy lub rozpakowanej struktury oraz właściwości klasy mogą być przekazywane do podprogramu przez referencję. W ten sposób

nie można natomiast przekazywać sieci, wyboru bitów sieci lub wyboru części sieci. Również argumenty nie mogą być przekazywane przez referencję statycznych podprogramów.

Kwalifikator **ref** nie może być łączony z kwalifikatorami **input**, **output** lub **inout**.

Kwalifikator **const** wraz z **ref** wskazuje, że argument jest zmienną tylko do odczytu.

W SystemVerilog argumenty podprogramu mogą być spakowanymi lub rozpakowanymi tablicami oraz strukturami, a także rozpakowanymi i oznaczonymi uniami.

W SystemVerilog zadania i funkcje mogą być całkowicie puste, bez operatorów lub grup operatorów. Puste zadanie bądź funkcja to swego rodzaju „wypełniacz" dla tworzonego kodu.

Różnice między zadaniami i funkcjami w języku SystemVerilog zostały zatarte przez rozszerzenie możliwości funkcji.

Funkcje są bardziej odpowiednie do opisu układów kombinacyjnych, a zadania do schematów sekwencyjnych.

Standard języka SystemVerilog opisuje wiele zadań i funkcji systemowych. Jednak większość z nich jest obsługiwana tylko przy symulacji.

Do syntezy można wykorzystywać funkcje systemowe do określania rozmiaru wyrażeń **$bits**, funkcje do pracy z tablicami oraz funkcje do pracy z wyrażeniami bitowymi.

14. Interfejsy

Podczas tworzenia złożonych projektów, gdy moduły mają dużą liczbę portów, projektanci stają przed problemem redundancji w opisie takich projektów w języku Verilog. Na przykład system mikroprocesorowy, w którym magistrala główna składa się z kilku szyn i dużej liczby sygnałów sterujących, wymaga opisania jako zbiór portów w module procesora, a także w instancjach modułów wszystkich elementów podłączonych do tej magistrali. Uwaga projektanta często skupia się więc nie na opisie funkcjonalności układu, ale na sprawdzeniu, czy porty są podłączone do wymaganych sygnałów.

Aby rozwiązać ten problem, język SystemVerilog wprowadza pojęcie *interfejsu* (*interface*). Umożliwia on zastąpienie grupy nazw portów jedną nazwą i późniejszą pracę z tą grupą jak z pojedynczym portem.

Problemem jest również określenie kierunku sygnałów interfejsu, np. dla niektórych modułów pewne sygnały mogą być używane jako wejściowe, a dla innych te same sygnały mogą być używane jako wyjściowe. W SystemVerilog ten problem jest rozwiązany za pomocą tzw. *modportów* (*modports*). Pozwalają one też na ograniczenie dostępu poszczególnych modułów do określonych sygnałów interfejsu, co przyczynia się do zwiększenia niezawodności projektu.

Często interfejsy, zwłaszcza w telekomunikacji, mają własne funkcjonalności. Na przykład interfejsy odpowiadające protokołom transmisji danych muszą generować sygnały sterujące, przesyłać dane, odbierać odpowiedź, sprawdzać sumę kontrolną itd. Takie funkcjonalności są zwykle specyficzne tylko dla tego interfejsu i nie mają nic wspólnego z funkcjonalnościami podstawowych elementów systemu.

Aby rozwiązać ten problem, w interfejsach języka SystemVerilog dopuszcza się opisywanie ich funkcjonalności za pomocą operatorów proceduralnych, bloków proceduralnych, zadań i funkcji. Dodatkowo metody przetwarzania danych opisane w interfejsach mogą być eksportowane do modułów projektu.

W SystemVerilog możliwe jest połączenie jednego interfejsu z drugim. Pozwala to na stworzenie drzewiastej struktury wzajemnych powiązań złożonego systemu.

Dodatkowo interfejsy mogą mieć własne porty i sygnały wewnętrzne, np. do testowania funkcjonalności interfejsu.

W interfejsach można również używać parametrów, którymi mogą być nie tylko stałe, lecz także typy danych.

14.1. Informacje ogólne

W języku SystemVerilog wprowadzono interfejsy grupujące sygnały służące do komunikacji między cyfrowymi blokami systemu. Pozwala to na odgórne projektowanie złożonych projektów, płynne przechodzenie od poziomów abstrakcyjnego i strukturalnego do poziomu RTL. Interfejsy ułatwiają również ponowne wykorzystanie projektów. Są to potężne narzędzia do projektowania złożonych projektów, co stanowi o znacznej przewadze języka SystemVerilog nad językiem Verilog.

Na najniższym poziomie (uproszczonym) interfejs to nazwany pakiet układów i zmiennych. Zazwyczaj duża część kodu projektu to lista portów i lista połączeń portów. Interfejsy pozwalają na zastąpienie grupy nazw portów jedną nazwą, a tym samym istotnie poprawiają czytelność projektu i zwiększają możliwości jego modyfikacji poprzez skupienie uwagi projektanta na funkcjonalności projektu, a nie na sprawdzaniu, czy porty są prawidłowo podłączone do modułów.

Te same sygnały interfejsu w różnych modułach mogą mieć inny kierunek. Na przykład w modułach typu *master* sygnały mają jeden kierunek, natomiast w modułach typu *slave* mają kierunek przeciwny. Do wskazania kierunku portów w określonych modułach są używane *modporty* (*modports*), które są deklarowane razem z interfejsami.

Dodatkowo interfejsy pozwalają na enkapsulację funkcjonalności do implementacji protokołów komunikacyjnych i ich weryfikacji, upodabniając interfejsy do szablonów klas. Interfejs może mieć parametry i stałe. W interfejsach mogą być deklarowane typy elementów, które mogą również być w nich przekazywane jako parametry. Interfejsy mogą zawierać bloki procedur (takie jak **always** i **initial**), operatory przypisania ciągłego, zadania i funkcje służące do implementacji funkcjonalności. Moduły projektu połączone przez interfejs mogą wywoływać zadania i funkcje z tego interfejsu, aby kontrolować połączenie.

Ponieważ funkcjonalność interfejsu jest odizolowana od modułów projektu, interfejs może być zastąpiony innym interfejsem o tych samych elementach, ale na innym poziomie abstrakcji. Moduły podłączone do interfejsu nie muszą być zmieniane.

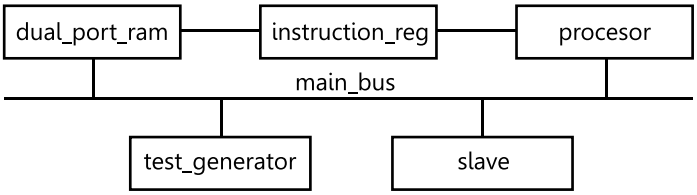
Funkcjonalność interfejsów pozwala na dołączenie do nich własnego narzędzia do weryfikacji protokołów, np. w celu automatycznego sprawdzenia, czy moduły połączone przez interfejs są zgodne z określonym protokołem. Z interfejsem można też zintegrować inne aplikacje weryfikacyjne, np. raport pokrycia funkcjonalnego czy weryfikację protokołu.

Ponadto podprogramy (zadania i funkcje) interfejsów mogą być zdefiniowane w jednym module i wywoływane w innym module za pomocą konstrukcji **export** i **import**. Interfejsy mogą być także wykorzystywane do symulacji komunikacji rozgłoszeniowej za pomocą zadań **forkjoin**, które mogą być zdefiniowane w kilku modułach i wykonywane jednocześnie.

14.2. Deklaracja interfejsów

14.2.1. Przykład systemu cyfrowego bez interfejsów

Na rys. 14.1 przedstawiono przykład pewnego systemu mikroprocesorowego ze wspólną magistralą *main_bus*, procesorem głównym *processor*, urządzeniem *slave*, pamięcią rozkazów *dual_port_ram*, generatorem testów *test_generator* i rejestrem instrukcji *instruction_reg*.



RYS. 14.1. Schemat blokowy systemu mikroprocesorowego

ŹRÓDŁO: oprac. własne.

Zauważmy, że rejestr instrukcji *instruction_reg* nie jest połączony z pamięcią i procesorem przez wspólną magistralę *main_bus*, lecz przez osobne połączenia.

Poniżej znajduje się kod w języku Verilog, opisujący prosty system cyfrowy, bez użycia koncepcji interfejsów języka SystemVerilog.

Przykład 14.1. Opis prostego systemu cyfrowego bez użycia koncepcji interfejsów

```
// moduł najwyższego poziomu
module top (input wire clock, resetN, test_mode);
        wire [15:0] data, address, program_address, jump_address;
        wire [ 7:0] instruction, next_instruction;
        wire [ 3:0] slave_instruction;
        wire slave_request, slave_ready;
        wire bus_request, bus_grant;
        wire mem_read, mem_write;
        wire data_ready;

// opis elementów systemu
        processor proc1 (
        // porty szyny main_bus
        .data(data),
        .address(address),
        .slave_instruction(slave_instruction),
        .slave_request(slave_request),
```

```
.bus_grant(bus_grant),
.mem_read(mem_read),
.mem_write(mem_write),
.bus_request(bus_request),
.slave_ready(slave_ready),
// inne porty
.jump_address(jump_address),
.instruction(instruction),
.clock(clock),
.resetN(resetN),
.test_mode(test_mode)
);

slave slave1 (
// porty szyny main_bus
.data(data),
.address(address),
.bus_request(bus_request),
.slave_ready(slave_ready),
.mem_read(mem_read),
.mem_write(mem_write),
.slave_instruction(slave_instruction),
.slave_request(slave_request),
.bus_grant(bus_grant),
.data_ready(data_ready),
// inne porty
.clock(clock),
.resetN(resetN)
);

dual_port_ram ram (
// porty szyny main_bus
.data(data),
.data_ready(data_ready),
.address(address),
.mem_read(mem_read),
.mem_write(mem_write),
// inne porty
.program_address(program_address),
.data_b(next_instruction)
);

test_generator test_gen(
```

```
        // porty szyny main_bus
        .data(data),
        .address(address),
        .mem_read(mem_read),
        .mem_write(mem_write),
        // inne porty
        .clock(clock),
        .resetN(resetN),
        .test_mode(test_mode)
        );

        instruction_reg ir (
        .program_address(program_address),
        .instruction(instruction),
        .jump_address(jump_address),
        .next_instruction(next_instruction),
        .clock(clock),
        .resetN(resetN)
        );
endmodule

// definicja modułów
module processor (
        // porty szyny main_bus
        inout wire [15:0] data,
        output reg [15:0] address,
        output reg [ 3:0] slave_instruction,
        output reg slave_request,
        output reg bus_grant,
        output wire mem_read,
        output wire mem_write,
        input wire bus_request,
        input wire slave_ready,
        // inne porty
        output reg [15:0] jump_address,
        input wire [ 7:0] instruction,
        input wire clock,
        input wire resetN,
        input wire test_mode
        );
        // kod funkcjonalności modułu
endmodule
```

```
module slave (
        // porty szyny main_bus
        inout wire [15:0] data,
        inout wire [15:0] address,
        output reg bus_request,
        output reg slave_ready,
        output wire mem_read,
        output wire mem_write,
        input wire [ 3:0] slave_instruction,
        input wire slave_request,
        input wire bus_grant,
        input wire data_ready,
        // inne porty
        input wire clock,
        input wire resetN
        );
        // kod funkcjonalności modułu
endmodule

module dual_port_ram (
// porty szyny main_bus
        inout wire [15:0] data,
        output wire data_ready,
        input wire [15:0] address,
        input tri0 mem_read,
        input tri0 mem_write,
        // inne porty
        input wire [15:0] program_address,
        output reg [ 7:0] data_b
        );
        // kod funkcjonalności modułu
endmodule

module test_generator (
        // porty szyny main_bus
        output wire [15:0] data,
        output reg [15:0] address,
        output reg mem_read,
        output reg mem_write,
        // inne porty
        input wire clock,
        input wire resetN,
        input wire test_mode
        );
```

```
        // kod funkcjonalności modułu
endmodule

module instruction_reg (
        output reg [15:0] program_address,
        output reg [ 7:0] instruction,
        input wire [15:0] jump_address,
        input wire [ 7:0] next_instruction,
        input wire clock,
        input wire resetN
        );
        // kod funkcjonalności modułu
endmodule
```

Redundancja kodu jest natychmiast widoczna podczas projektowania takich dużych i złożonych projektów w Verilogu. W takim przypadku projektanci poświęcają dużo czasu na ręczne sprawdzanie, czy porty instancji modułów są prawidłowo połączone. Stąd możemy wnioskować, że język Verilog ma ograniczone możliwości przy opisie dużych, złożonych projektów.

14.2.2. Grupowanie sygnałów za pomocą interfejsów

Najłatwiejszym sposobem wykorzystania interfejsów jest grupowanie sygnałów magistrali systemowej tak, aby grupa sygnałów mogła być przekazywana między modułami jako jeden port, bez konieczności dublowania wszystkich sygnałów. Deklaracja interfejsu ma następującą składnię:

interface *interface_name*
        …
        interface_items
        …
endinterface [: *interface_name*]

gdzie: *interface_name* jest nazwą interfejsu; *interface_items* to elementy interfejsu; **interface** i **endinterface** to słowa kluczowe.

Poniższy kod deklaruje interfejs *main_bus* w celu uproszczenia opisu naszego przykładowego układu cyfrowego. W module *top* najwyższego poziomu tworzona jest instancja magistrali *bus* interfejsu *main_bus*. Interfejs *bus* typu *main_bus* jest zadeklarowany na liście portów każdego modułu w naszym systemie cyfrowym. Instancja interfejsu *bus* jest następnie przekazywana przez nazwę, jako zwykły port, do wszystkich instancji modułów, które są tworzone w module *top*.

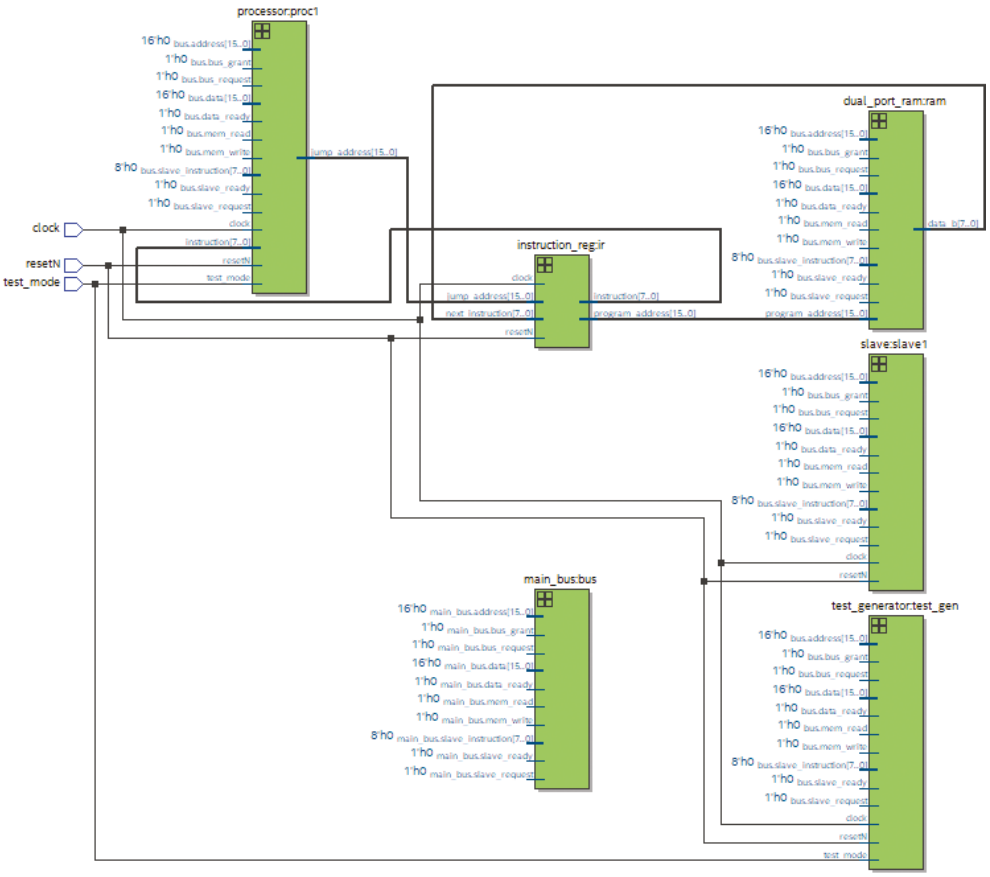
Przykład 14.2. Zastosowanie interfejsu do grupowania wspólnych sygnałów szyny *main_bus*

```
// deklaracja interfejsu
interface main_bus;
        wire [15:0] data;
        wire [15:0] address;
        logic [ 7:0] slave_instruction;
        logic slave_request;
        logic bus_grant;
        logic bus_request;
        logic slave_ready;
        logic data_ready;
        logic mem_read;
        logic mem_write;
endinterface

// moduł najwyższego poziomu
module top (input logic clock, resetN, test_mode);
        logic [15:0] program_address, jump_address;
        logic [ 7:0] instruction, next_instruction;

        main_bus bus ( );               // instancja interfejsu

        // opis elementów systemu
        processor proc1 (
        // porty szyny main_bus
        .bus(bus),                      // połączenie interfejsu
        // inne porty
        .jump_address(jump_address),
        .instruction(instruction),
        .clock(clock),
        .resetN(resetN),
        .test_mode(test_mode)
        );

        slave slave1 (
        // porty szyny main_bus
        .bus(bus),                      // połączenie interfejsu
        // inne porty
        .clock(clock),
        .resetN(resetN)
        );
```

```
        dual_port_ram ram (
        // porty szyny main_bus
        .bus(bus),                  // połączenie interfejsu
        // inne porty
        .program_address(program_address),
        .data_b(next_instruction)
        );

        test_generator test_gen(
        // porty szyny main_bus
        .bus(bus),                  // połączenie interfejsu
        // inne porty
        .clock(clock),
        .resetN(resetN),
        .test_mode(test_mode)
        );

        instruction_reg ir (
        .program_address(program_address),
        .instruction(instruction),
        .jump_address(jump_address),
        .next_instruction(next_instruction),
        .clock(clock),
        .resetN(resetN)
        );
endmodule

// opis modułów
module processor (
        main_bus bus,               // port połączenia z interfejsem main_bus
        // inne porty
        output logic [15:0] jump_address,
        input logic [ 7:0] instruction,
        input logic clock,
        input logic resetN,
        input logic test_mode
        );
        // kod funkcjonalności modułu
endmodule

module slave (
        main_bus bus,               // port połączenia z interfejsem main_bus
        // inne porty
```

```
        input logic clock,
        input logic resetN
        );
        // kod funkcjonalności modułu
endmodule

module dual_port_ram (
        main_bus bus,                   // port połączenia z interfejsem main_bus
        // inne porty
        input logic [15:0] program_address,
        output logic [ 7:0] data_b
        );
        // kod funkcjonalności modułu
endmodule

module test_generator (
        main_bus bus,                   // port połączenia z interfejsem main_bus
        // inne porty
        input logic clock,
        input logic resetN,
        input logic test_mode
        );
        // kod funkcjonalności modułu
endmodule

module instruction_reg (
        output logic [15:0] program_address,
        output logic [ 7:0] instruction,
        input logic [15:0] jump_address,
        input logic [ 7:0] next_instruction,
        input logic clock,
        input logic resetN
        );
        // kod funkcjonalności modułu
endmodule
```

Wynik syntezy projektu z przykładu 14.2 przedstawiono na rys. 14.2.



RYS. 14.2. Wynik syntezy przykładu 14.2 przy zastosowaniu interfejsu do grupowania sygnałów wspólnej magistrali

ŹRÓDŁO: oprac. własne.

Na rys. 14.2 widzimy, że interfejs jest przedstawiony jako osobny blok, który nie jest do niczego podłączony, a w każdym module zgrupowane sygnały również pozostają niepodłączone. Tak więc proste zastosowanie interfejsów do grupowania sygnałów, przy jednoczesnej redukcji kodu źródłowego, pozostawia otwartą kwestię połączenia i kierunku zgrupowanych sygnałów. Ten ostatni problem rozwiązuje się poprzez definiowanie *modportów* (*modports*) w interfejsach, co omówiono poniżej.

14.2.3. Porty interfejsów

W języku SystemVerilog interfejsy mogą, podobnie jak moduły, mieć własne porty. Pozwala to na przesyłanie sygnałów zewnętrznych do interfejsu, takich jak sygnał zegara i reset. Interfejsy mogą również zawierać deklaracje dowolnych typów sieci i zmiennych, a także typów użytkownika.

Składnia interfejsu z portami jest następująca:

```
interface  interface_name ( port_list );
        …
        interface_items;
        …
endinterface [ : interface_name ]
```

gdzie *port_list* jest listą portów interfejsu, która ma dokładnie taki sam format jak lista portów modułu.

Poniższy kod pokazuje użycie interfejsu z portami dla naszego przykładowego układu cyfrowego.

Przykład 14.3. Wykorzystanie interfejsu z własnymi portami dla sygnałów zewnętrznych

```
// definicja interfejsu z portami dla sygnałów zewnętrznych
interface main_bus (input logic clock, resetN, test_mode);
        wire [15:0] data;
        wire [15:0] address;
        logic [ 7:0] slave_instruction;
        logic slave_request;
        logic bus_grant;
        logic bus_request;
        logic slave_ready;
        logic data_ready;
        logic mem_read;
        logic mem_write;
endinterface

// moduł najwyższego poziomu
module top (input logic clock, resetN, test_mode);
        logic [15:0] program_address, jump_address;
        logic [ 7:0] instruction, next_instruction;

        // instancja interfejsu
        main_bus bus (
        .clock(clock),                  // przekazane do interfejsu
        .resetN(resetN),                // sygnały zewnętrzne
        .test_mode(test_mode)
        );
```

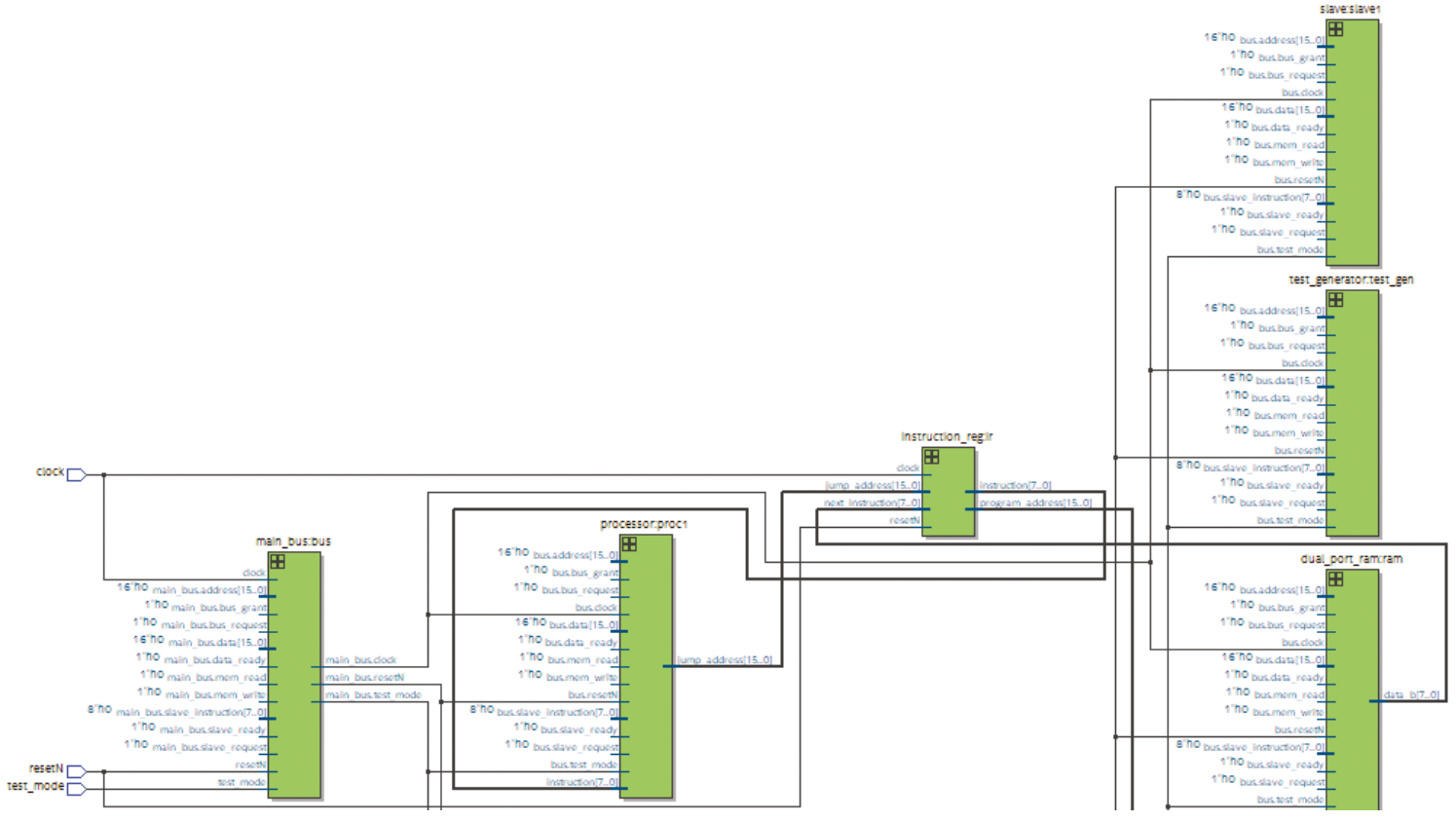
```
        // opis elementów systemu
        processor proc1 (.bus(bus),                 // połączenie interfejsu
        // inne porty
        .jump_address(jump_address),
        .instruction(instruction)
        );

        slave slave1 (.bus(bus));                   // połączenie interfejsu

        dual_port_ram ram (.bus(bus),               // połączenie interfejsu
        // inne porty
        .program_address(program_address),
        .data_b(next_instruction)
        );

        test_generator test_gen(.bus(bus));             // połączenie interfejsu

        instruction_reg ir (
        .program_address(program_address),
        .instruction(instruction),
        .jump_address(jump_address),
        .next_instruction(next_instruction),
        .clock(clock),
        .resetN(resetN)
        );
endmodule

// opis modułów
module processor (
        main_bus bus,               // port połączenia z interfejsem main_bus
        // inne porty
        output logic [15:0] jump_address,
        input logic [ 7:0] instruction
        );
        // kod funkcjonalności modułu
endmodule

module slave (
        main_bus bus);              // port połączenia z interfejsem main_bus
        // kod funkcjonalności modułu
endmodule
```

```
module dual_port_ram (
        main_bus bus,                   // port połączenia z interfejsem main_bus
        // inne porty
        input logic [15:0] program_address,
        output logic [ 7:0] data_b
        );
        // kod funkcjonalności modułu
endmodule

module test_generator (
        main_bus bus);                  // port połączenia z interfejsem main_bus
        // kod funkcjonalności modułu
endmodule

module instruction_reg (
        output logic [15:0] program_address,
        output logic [ 7:0] instruction,
        input logic [15:0] jump_address,
        input logic [ 7:0] next_instruction,
        input logic clock,
        input logic resetN
        );
// kod funkcjonalności modułu
endmodule
```

W powyższym kodzie lista portów interfejsu zawiera trzy sygnały wejściowe *clock*, *resetN* i *test_mode*. Gdy w module górnym tworzona jest instancja magistrali interfejsu *main_bus*, sygnały o takich samych nazwach *clock*, *resetN* i *test_mode* są przekazywane do interfejsu jako sygnały zewnętrzne.

Generalnie możliwe jest stworzenie wielu instancji interfejsu main_bus, a do każdej z nich można przekazywać różne sygnały zewnętrzne. Zauważmy też, że w opisie modułu nie ma deklaracji dla sygnałów *clock*, *resetN* i *test_mode*, ponieważ są one już zdefiniowane w instancji *bus* interfejsu main_bus, która jest przekazywana do każdego modułu jako zwykły port. Wyniki syntezy projektu z przykładu 14.3 przedstawiono na rys. 14.3.

Na rys. 14.3 widzimy, że interfejs nie jest osobną jednostką, ma wejścia *clock*, *resetN* i *test_mode* oraz wyjścia *main_bus.clock*, *main-bus.resetN* i *main-bus.test_mode*; wyjścia interfejsu są połączone z wejściami wszystkich modułów oprócz *instruction_reg*. Jednak sygnały zgrupowane z interfejsem nadal nie mają połączeń. W rzeczywistości mają odpowiednie połączenia w bazie danych projektu, ale nie są one pokazane na rys. 14.3.



RYS. 14.3 Wynik syntezy z przykładu 14.3 w przypadku interfejsu z portami dla sygnałów zewnętrznych

ŹRÓDŁO: oprac. własne.

14.2.4. Stosowanie notacji .name i .*

W przypadku stosowania interfejsów w celu uproszczenia opisu portów modułów można stosować notację **.name** i **.***. Na przykład moduł *top* w naszym systemie cyfrowym przy wykorzystaniu notacji .* wyglądałby następująco.

Przykład 14.4. Użycie konstrukcji .* do opisu modułów z interfejsami

```
// moduł top
module top (input logic clock, resetN, test_mode);

        logic [15:0] program_address, jump_address;
        logic [ 7:0] instruction, next_instruction, data_b;

        main_bus bus ( .* );                    // instancja interfejsu

        // opis elementów systemu
        processor proc1 ( .* );
        slave slave1 ( .* );
        instruction_reg ir ( .* );
        test_generator test_gen ( .* );
        dual_port_ram ram ( .*, .data_b(next_instruction) );
endmodule
```

Opis interfejsu i pozostałych modułów naszego systemu pozostaje bez zmian. Ponieważ notacja .* skraca tylko opis, wyniki syntezy przykładu 14.4 nie ulegają zmianie i są takie same jak na rys. 14.3.

14.2.5. Interfejsy globalne i lokalne

Nazwy interfejsów w kodzie projektu mogą być zadeklarowane jako globalne lub lokalne.

Jeśli interfejs jest zadeklarowany oddzielnie od definicji modułów za pomocą słów kluczowych **interface** i **endinterface**, to jego nazwa jest globalna i będzie w globalnym kontekście definicji nazw modułów. Dzięki temu definicja interfejsu może być użyta jako port dla dowolnego modułu w dowolnym punkcie hierarchii projektu.

Jeśli interfejs jest zadeklarowany wewnątrz modułu, nazwa interfejsu jest lokalna dla tego modułu. Tylko moduł, w którym interfejs jest lokalnie zadeklarowany, może stworzyć jego instancję. Dzięki temu można ograniczyć stosowanie interfejsu tylko do jednej części hierarchii projektu, np. tylko w obrębie modelu bloku IP (*intellectual property*).

14.3. Użycie interfejsów jako portów modułu

Porty modułów, przez które przesyłane są interfejsy, nazywane są *portami interfejsów* (*interface ports*).

14.3.1. Porty interfejsu zadeklarowane w sposób jawny

Porty interfejsów w modułach mogą być zadeklarowane jawnie. W tym celu zamiast kierunku portu (**input**, **output**, **inout** lub **ref**) określa się typ interfejsu przy pomocy następującej składni:

module *module_name* (*interface_name port_name*);

gdzie: *module_name* jest nazwą modułu; *interface_name* jest nazwą interfejsu; *port_name* jest nazwą portu.

Na przykład:

```
interface chip_bus;                  // deklaracja interfejsu
...
endinterface

module cache (chip_bus pins,         // port interfejsu
      input clock);                  // normalny port
...
endmodule
```

Jawnie zadeklarowany port interfejsu może być podłączony tylko do interfejsu o tej samej nazwie.

14.3.2. Uniwersalne porty interfejsów

Port interfejsu w module może być zadeklarowany za pomocą słowa kluczowego **interface** w postaci:

module *module_name* (**interface** *port_name*);

gdzie *module_name* jest nazwą modułu, a *port_name* jest nazwą portu.

W tym przypadku port ten nazywany jest *portem interfejsu ogólnego* (*generic interface port*). Może się on łączyć z różnymi interfejsami, np.:

```
module RAM (interface pins,          // uniwersalny port interfejsu
        input clock);
...
endmodule
```

W tym przykładzie port *pins* modułu RAM może być podłączony do różnych interfejsów.

14.4. Tworzenie instancji interfejsów i podłączanie interfejsów do portów modułu

14.4.1. Tworzenie instancji interfejsu

Przed użyciem interfejsu w projekcie, podobnie jak w przypadku modułów, należy utworzyć instancję interfejsu. Jest on tworzony w taki sam sposób jak instancje modułów. Na przykład:

```
main_bus  bus1 ();                   // tworzenie instancji interfejsu bez portów

main_bus  bus2                       // tworzenie instancji interfejsu przy przekazywaniu
        ( clock, resetN, test_mode ); // sygnałów do portów interfejsu według pozycji

main_bus  bus3 (                     // tworzenie instancji interfejsu przy przekazywaniu
                                     // sygnałów do portów interfejsu według nazwy
        .clock (clock),
        .resetN (resetN),
        .test_mode (test_mode));

main_bus  bus4 (.*);                 // tworzenie instancji interfejsu z użyciem notacji .*
```

Zauważmy, że jeśli interfejs został zadeklarowany z portami, to przy tworzeniu instancji interfejsu, w przeciwieństwie do modułów, obowiązkowe jest określenie sygnałów podłączonych do portów interfejsu.

14.4.2. Podłączanie interfejsów do portów modułu

Podłączenie interfejsów do portów modułu jest określane podczas tworzenia instancji modułu. Jeśli port interfejsu modułu jest jawnie zadeklarowany z nazwą interfejsu, to może połączyć się tylko z instancją określonego typu interfejsu. Na przykład:

```
main_bus bus ();                    // tworzenie instancji interfejsu

processor proc1 (bus, … );          // tworzenie instancji modułu
                                    // podłączenie portu interfejsu według pozycji

processor proc2 ( .bus (bus), … );  // tworzenie instancji modułu
                                    // połączenie portu interfejsu według nazwy

processor proc3 ( .* );             // tworzenie instancji modułu
                                    // podłączenie portu interfejsu za pomocą notacji .*
```

Przypomnijmy, że jeśli port interfejsu modułu jest zadeklarowany za pomocą słowa kluczowego **interface** jako uniwersalny, może on połączyć się z instancją dowolnego typu interfejsu.

14.4.3. Podłączanie interfejsów do innych interfejsów

Port interfejsu może być również zdefiniowany jako interfejs. Dzięki temu jeden interfejs może być połączony z innym interfejsem. Na przykład magistrala główna projektu może mieć jedną lub więcej magistral podrzędnych. Zarówno magistrala główna, jak i jej elementy podrzędne mogą być opisane jako interfejsy. Interfejsy podrzędne można opisać jako porty na interfejsie głównym. Na przykład:

```
// interfejs magistrali głównej
interface main_bus ( interface adder, interface data, interface control );
        …
endinterface

interface addr_bus;                 // interfejs magistrali adresowej
        wire [ 15 : 0 ] address;
        …
endinterface

interface data_bus;                 // interfejs magistrali danych
        wire [ 7 : 0 ] data;
        …
endinterface

interface control_bus;              // interfejs sygnałów sterujących
        logic slave_ready;
        logic data_ready;
        logic mem_ready;
```

```
        logic mem_write;
endinterface
…
module top (…);
…
        addr_bus addr ();               // tworzenie instancji addr interfejsu addr_bus
        data_bus data ();               // tworzenie instancji data interfejsu data_bus
        control_bus control ();         // tworzenie instancji control control_bus
        main_bus bus (                  // tworzenie instancji bus interfejsu main_bus
                .addr ( addr ),
                .data ( data ),
                .control ( control )
        );
        …
endmodule
```

W tym przykładzie magistrala główna *bus* ma trzy elementy podrzędne: *addr*, *data* i *control*.

14.5. Odnoszenie się do sygnałów w ramach interfejsu

W module, który ma port interfejsu, można odnosić się do sygnałów w ramach interfejsu za pomocą następującej składni:

port_name.signal_name

gdzie *port_name* jest nazwą portu, a *signal_name* jest nazwą sygnału w ramach interfejsu.

Na przykład:

```
// deklaracja interfejsu
interface main_bus ( input logic clock, resetN, test_mode );
        wire [15 : 0] data;
        wire [15 : 0] address;
        logic  mem_write;
endinterface

// deklaracja modułu
module slave (
        main_bus  bus,                  // przekazanie interfejsu przez port modułu
```

```
        …                               // inne porty
        );
        …
        assign  bus.address = … ;       // odniesienie do sygnałów w interfejsie
        assign  bus.data = … ;
        always_ff @ ( posedge bus.clock, negedge bus.resetN ) begin : FSM
        …
        end
endmodule
```

W powyższym przykładzie moduł *slave* przypisuje wartości sygnałów *address* i *data* interfejsu *bus*.

14.6. Modporty interfejsów

Każdy konkretny moduł podłączony do interfejsu może wymagać innego kierunku przesyłania sygnałów zadeklarowanych w interfejsie. Na przykład dla urządzenia podrzędnego sygnał magistrali *interrupt_request* może być sygnałem wyjściowym, a dla procesora nadrzędnego może być sygnałem wejściowym.

Modporty (*modports*) służą do określenia kierunku sygnałów w interfejsach. Słowo *modports* jest skrótem od słów *module ports*.

14.6.1. Definicje modportów

Modporty w interfejsie definiuje się za pomocą następującej składni:

modport *modport_name* ({ *direction signal_name* });

gdzie: **modport** jest słowem kluczowym; *modport_name* jest nazwą modportu; *direction* jest kierunkiem sygnału określonym przez słowo kluczowe **input**, **output**, **inout** lub **ref**; *signal_name* jest nazwą sygnału. Na przykład:

```
interface chip_bus (input logic clock, resetN);     // definicja interfejsu
        logic interrupt_request, grant, ready;      // deklaracja sygnałów interfejsu
        logic [31:0] address;
        wire [63:0] data;

        modport master                              // definicja modportu master
                (input interrupt_request,
                input address,
```

```
        output grant, ready,
        inout data,
        input clock, resetN);

    modport slave                          // definicja modportu slave
        (output interrupt_request,
        output address,
        input grant, ready,
        inout data,
        input clock, resetN);
endinterface
```

Zauważmy, że definicje modportów nie zawierają rozmiarów wektorów ani typów sygnałów. Deklaracja modportu określa jedynie kierunek sygnałów (widziany przez odpowiedni moduł): **input**, **output**, **inout** lub **ref**.

14.6.2. Dwa sposoby dołączenia interfejsów z modportami do instancji modułów

W języku SystemVerilog istnieją dwa sposoby na określenie, który modport interfejsu powinien być używany przez daną instancję modułu:

- przez określenie modportu podczas tworzenia instancji modułu;
- przez określenie modportu podczas deklaracji portu modułu.

Oba te style są syntetyzowalne.

14.6.2.1. Wybór modportu w instancji modułu

Konkretny interfejs modport można określić podczas tworzenia instancji modułu za pomocą następującej składni:

interface_instance_name.modport_name

gdzie *interface_instance_name* jest nazwą instancji interfejsu, a *modport_name* jest nazwą modportu. Na przykład:

```
chip_bus bus;                  // instancja bus interfejsu chip_bus

// tworzenie instancji i1 modułu primary
primary i1 ( bus.master );     // używany jest modport master instancji interfejsu bus
```

W poniższym przykładzie modport *master* służy do podłączenia interfejsu *chip_bus* do instancji *i1* modułu *primary*, a modport *slave* do instancji *i2* modułu *secondary*.

```
interface chip_bus (input logic clock, resetN);
        modport master (...);
        modport slave (...);
endinterface

module primary (interface pins);     // uniwersalny port interfejsu
        ...
endmodule

module secondary (chip_bus pins); // zdefiniowany port interfejsu
        ...
endmodule

module chip (input logic clock, resetN);
        chip_bus bus (clock, resetN);          // instancja interfejsu

        primary i1 (bus.master);               // używanie modportu master

        secondary i2 (bus.slave);              // używanie modportu slave
endmodule
```

14.6.2.2. Wybór modportu w deklaracji portu modułu

Konkretny interfejs modport może być określony podczas deklarowania portu modułu przy użyciu następującej składni:

interface_name.modport_name

gdzie *interface_name* jest nazwą interfejsu, a *modport_name* jest nazwą modportu.

Na przykład:

```
module secondary (chip_bus.slave pins);   // port interfejsu z nazwą chip_bus
                                          // i nazwa modportu slave
        ...
endmodule
```

Poniżej znajduje się bardziej kompletny przykład pokazujący, jak połączyć modporty z instancjami modułów, gdy modport jest określony w deklaracji portu modułu.

```
interface chip_bus (input logic clock, resetN);
        modport master (...);
        modport slave (...);
endinterface
```

```
module primary (chip_bus.master pins);     // wykorzystanie modportu master
        ...
endmodule

module secondary (chip_bus.slave pins);    // wykorzystanie modportu slave
        ...
endmodule

module chip (input logic clock, resetN);
        chip_bus bus (clock, resetN);      // instancja interfejsu

        primary i1 (bus);                  // zostanie użyty modport master

        secondary i2 (bus);                // zostanie użyty modport slave
endmodule
```

Zauważmy, że moduły mogą łączyć się z interfejsem bez określania konkretnego modportu. W tym przypadku zakłada się, że wszystkie węzły w interfejsie mają kierunek **inout**, a wszystkie zmienne w interfejsie są typu **ref**. Na przykład:

```
module test (chip_bus pins);          // podłączenie modułu do interfejsu bez podawania
                                      // modportu
```

14.6.3. Ograniczenie dostępu do sygnałów interfejsu przez modporty

W interfejsach złożonych może być wymagane, aby każdy moduł podłączony do interfejsu widział tylko jego ograniczony zestaw sygnałów. Wymagania te można zrealizować za pomocą modportów, ponieważ każdy moduł ma dostęp tylko do sygnałów wymienionych w definicji jego modportu. Na przykład interfejs zawiera sygnał *test_clock*, który może być używany tylko przez moduły podłączone do interfejsu przez modport *master*, natomiast sygnał *test_clock* nie może być używany przez moduły podłączone przez modport *slave*.

Zauważmy, że do każdego obiektu w interfejsie można uzyskać dostęp za pomocą pełnej nazwy ścieżki hierarchicznej. Jednak pełne hierarchiczne ścieżki obiektów interfejsu nie są możliwe do zsyntetyzowania i mogą być wykorzystywane jedynie do celów symulacji.

Interfejsy mogą zawierać wewnętrzne sygnały, które nie są widoczne w żadnym ze zdefiniowanych modportów. Zazwyczaj takie sygnały są wykorzystywane do weryfikacji protokołów komunikacyjnych.

Jeśli moduł jest podłączony do interfejsu bez określenia modportu, moduł ma dostęp do wszystkich sygnałów interfejsu.

W poniższym przykładzie naszego systemu mikroprocesorowego interfejs *main_bus* deklaruje trzy modporty *master*, *slave* i *mem*, które definiują różne zestawy sygnałów używanych odpowiednio przez moduły *processor*, *slave* i *dual_port_ram*. Instancja modułu *test_generator* łączy się z interfejsem bez określania modportu, więc wszystkie sygnały interfejsu są dla niej dostępne.

Przykład 14.5. Opis interfejsu z modportami

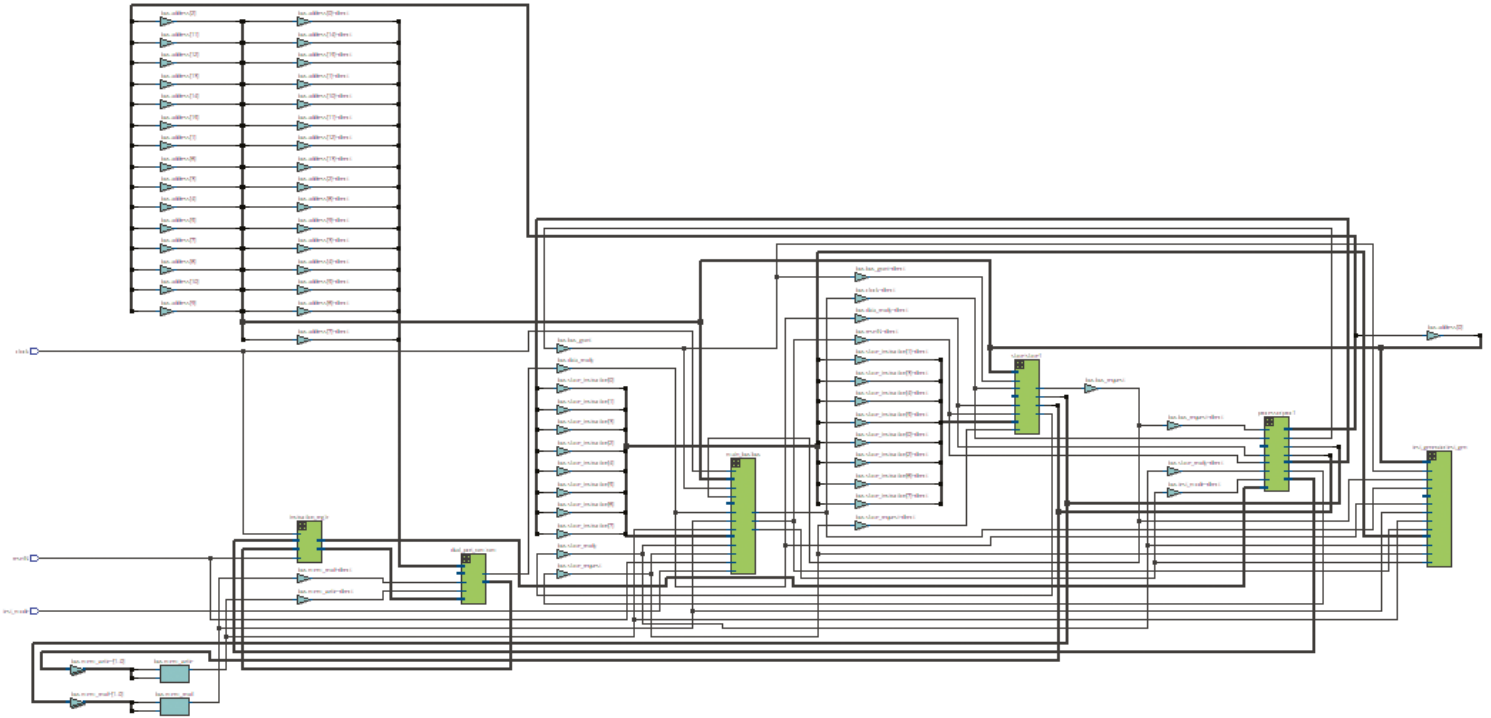
```
interface main_bus (input logic clock, resetN, test_mode);  // deklaracja interfejsu
        wire [15:0] data;
        wire [15:0] address;
        logic [ 7:0] slave_instruction;
        logic slave_request;
        logic bus_grant;
        logic bus_request;
        logic slave_ready;
        logic data_ready;
        logic mem_read;
        logic mem_write;

        modport master (                    // deklaracja modportu master
                inout data,
                output address,
                output slave_instruction,
                output slave_request,
                output bus_grant,
                output mem_read,
                output mem_write,
                input bus_request,
                input slave_ready,
                input data_ready,
                input clock,
                input resetN,
                input test_mode
        );

        modport slave (                     // deklaracja modportu slave
                inout data,
                inout address,
                output mem_read,
                output mem_write,
                output bus_request,
                output slave_ready,
```

```
            input slave_instruction,
            input slave_request,
            input bus_grant,
            input data_ready,
            input clock,
            input resetN
        );

        modport mem (                       // deklaracja modportu mem
            inout data,
            output data_ready,
            input address,
            input mem_read,
            input mem_write
        );
endinterface

// moduł najwyższego poziomu
module top (input logic clock, resetN, test_mode);

        logic [15:0] program_address, jump_address;
        logic [ 7:0] instruction, next_instruction, data_b;

        main_bus bus ( .* );            // instancja interfejsu

        // opis elementów systemu
        processor proc1 (.bus(bus.master), .* );
        slave slave1 (.bus(bus.slave), .* );
        instruction_reg ir ( .* );
        test_generator test_gen (.bus(bus), .* );
        dual_port_ram ram (.bus(bus.mem), .* , .data_b(next_instruction) );
endmodule

// opis modułów
module processor (
        main_bus bus,                   // port połączenia z interfejsem main_bus
        // inne porty
        output logic [15:0] jump_address,
        input logic [7:0] instruction
        );
        // kod funkcjonalności modułu
endmodule
```

```
module slave (
        main_bus bus                    // port połączenia z interfejsem main_bus
        );
        // kod funkcjonalności modułu
endmodule

module dual_port_ram (
        main_bus bus,                   // port połączenia z interfejsem main_bus
        // inne porty
        input logic [15:0] program_address,
        output logic [ 7:0] data_b
        );
        // kod funkcjonalności modułu
endmodule

module test_generator (
        main_bus bus                    // port połączenia z interfejsem main_bus
        );
        // kod funkcjonalności modułu
endmodule

module instruction_reg (
        output logic [15:0] program_address,
        output logic [ 7:0] instruction,
        input logic [15:0] jump_address,
        input logic [ 7:0] next_instruction,
        input logic clock,
        input logic resetN
        );
        // kod funkcjonalności modułu
endmodule
```

Wyniki syntezy powyższego kodu przedstawiono na rys. 14.4.

Na rys. 14.4 widzimy, że wszystkie elementy projektu mają przypisane wejścia i wyjścia.



RYS. 14.4. Wyniki syntezy przykładu 14.5 w przypadku zastosowania interfejsu z modportami

ŹRÓDŁO: oprac. własne.

14.6.4. Porównanie różnych sposobów opisu interfejsów

Tabela 14.1 przedstawia liczbę linii kodu dla różnych sposobów opisu prostego systemu mikroprocesorowego z rys. 14.1 za pomocą interfejsów.

TABELA 14.1. Liczba linii kodu prostego systemu mikroprocesorowego dla różnych metod opisu interfejsu

Przykład	Łączna liczba	Dla modułu top	Uwagi
Przykład 14.1.	126	65	opis bezpośredni bez użycia interfejsu
Przykład 14.2.	86	37	z użyciem interfejsu
Przykład 14.3.	67	27	interfejs z portami dla sygnałów zewnętrznych
Przykład 14.4.	51	10	z użyciem notacji .name i .*
Przykład 14.5.	84	10	z użyciem modportów

ŹRÓDŁO: oprac. własne.

Z tabeli 14.1 wynika, że zastosowanie interfejsów **.name** i **.*** może znacznie zmniejszyć liczbę linii kodu w opisie projektu. Na przykład dla modułu górnego *top* liczba linii kodu zmniejsza się o 65/10 = 6,5 razy. Chociaż użycie modportów zwiększa liczbę linii kodu, to jednak stwarza nowe możliwości dostępu różnych modułów do sygnałów interfejsu.

14.7. Zastosowanie zadań i funkcji w interfejsach

Oprócz definicji sieci i zmiennych interfejsy mogą zawierać zadania i funkcje realizujące ich funkcjonalności. Na przykład zgodnie z protokołem main_bus główny procesor musi zawierać kod procedury do ustawiania i zerowania swoich sygnałów potwierdzenia komunikacji oraz do monitorowania sygnałów potwierdzenia komunikacji urządzeń podrzędnych. Urządzenia slave muszą również mieć podobną funkcjonalność. Opis protokołu magistrali w każdym urządzeniu powoduje powielanie kodu.

W przypadku modyfikacji protokołu magistrali należy dokonać zmian w każdym module. Z tego powodu w języku SystemVerilog możliwe jest wbudowanie funkcjonalności interfejsu bezpośrednio w sam interfejs w postaci zadań i funkcji.

14.7.1. Metody interfejsu

Zadania i funkcje zawarte w interfejsie nazywane są *metodami interfejsu* (*interface methods*). Są one opisane przy użyciu tej samej składni, której używają moduły do opisu zadań i funkcji. Metody interfejsu mogą obsługiwać dowolne wewnętrzne sygnały.

Metody interfejsu są opisane wewnątrz niego i mogą być używane w modułach, do których są przekazywane przez porty modułu.

W poniższym przykładzie zadeklarowano interfejs *simple_bus* ze zdefiniowanymi dwoma zadaniami: *masterRead* i *slaveRead*. Instancja interfejsu *sb_intf* jest przekazywana do modułów *memMod* i *cpuMod* przez porty interfejsu uniwersalnego. Metoda interfejsu *slaveRead* jest wywoływana w module *memMod*, a metoda *masterRead* w module *cpuMod*.

```
interface simple_bus (input logic clk);              // deklaracja interfejsu
        logic req, gnt;
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;

        task masterRead(input logic [7:0] raddr);    // metoda MasterRead
                // ...
        endtask: masterRead

        task slaveRead;                              // metoda slaveRead
                // ...
        endtask: slaveRead

endinterface: simple_bus

module top;
        logic clk = 0;

        simple_bus sb_intf(clk);                     // instancja interfejsu

        memMod mem(sb_intf);
        cpuMod cpu(sb_intf);
endmodule

module memMod(interface a);                          // uniwersalny port interfejsu
        logic avail;

        always @(posedge a.clk)                      // sygnał clk interfejsu
                a.gnt <= a.req & avail               // sygnały interfejsu gnt i req

        always @(a.start)
                a.slaveRead;                         // wywołanie metody slaveRead interfejsu
endmodule
```

```
module cpuMod(interface b);
        enum {read, write} instr;
        logic [7:0] raddr;

        always @(posedge b.clk)
                if (instr == read)
                        b.masterRead(raddr); // wywołanie metody masterRead interfejsu
                ...
endmodule
```

14.7.2. Importowanie metod interfejsów do modportów

Modporty interfejsów mogą importować metody interfejsów za pomocą słowa kluczowego **import**. Istnieją dwa sposoby importowania metod interfejsu:

- używając nazwy zadania lub funkcji;
- wykorzystując *prototypy* (*prototype*) zadania lub funkcji.

Obie metody są równoważne i możliwe do syntezy. Druga metoda daje nieco więcej informacji o jej argumentach i może być wykorzystana do poprawy dokumentacji kodu.

Gdy używana jest nazwa metody interfejsu, składnia dla modportu jest następująca:

modport (**import** *method_name*);

gdzie *method_name* jest nazwą metody interfejsu (funkcji lub zadania).

Na przykład:

```
modport in (import Read,
        import parity_gen,
        input clock, resetN );
```

Składnia modportów w przypadku użycia prototypu metody interfejsu wygląda jak poniżej:

modport (**import task** *task_name* (*formal_arguments*));
modport (**import function** *function_name* (*formal_arguments*));

gdzie: *task_name* to nazwa zadania; *function_name* to nazwa funkcji; *formal_arguments* to argumenty formalne.

Na przykład:

```
// deklaracja modportu
modport in    (import task Read (input [63:0] data,              // importowanie zadania
                        output [31:0] address),
               import function parity_gen (input [63:0] data),   // importowanie funkcji
               input clock, resetN);                             // inne sygnały
```

W poniższym przykładzie modport *slave* interfejsu *simple_bus* importuje metody *slaveRead* i *slaveWrite*, a modport *master* importuje metody *masterRead* i *masterWrite*.

```
interface simple_bus (input logic clk);              // definicja interfejsu
        logic req, gnt;
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;

        modport slave         (input req, addr, mode, start, clk,
                output gnt, rdy,
                ref data,
                // import metod do modportu slave
                import slaveRead,
                        slaveWrite);

        modport master        (input gnt, rdy, clk,
                output req, addr, mode, start,
                ref data,
                // import metod do modportu master
                import masterRead,
                        masterWrite);

        // opis metod interfejsu
        task masterRead(input logic [7:0] raddr);        // metoda MasterRead
                // ...
        endtask

        task slaveRead;                                  // metoda slaveRead
                // ...
        endtask

        task masterWrite(input logic [7:0] waddr);       // metoda MasterWrite
                //...
        endtask

        task slaveWrite;                                 // metoda slaveWrite
```

```
            //...
        endtask

endinterface: simple_bus

module memMod(interface a);             // uniwersalny port interfejsu
        logic avail;

        always @(posedge a.clk)         // sygnał clk interfejsu
            a.gnt <= a.req & avail;     // sygnały interfejsu gnt i req

        always @(a.start)
            if (a.mode[0] == 1'b0)
                a.slaveRead;            // wywołanie metody interfejsu slaveRead
            else
            a.slaveWrite;               // wywołanie metody interfejsu slaveWrite
endmodule

module cpuMod(interface b);             // uniwersalny port interfejsu
        enum {read, write} instr;
        logic [7:0] raddr = $random();

        always @(posedge b.clk)         // sygnał clk interfejsu
            if (instr == read)
                b.masterRead(raddr); // wywołanie metody interfejsu MasterRead
            // ...
            else
                b.masterWrite(raddr);// wywołanie metody interfejsu MasterWrite
endmodule

module omniMod( interface c);           // uniwersalny port interfejsu
        //...
endmodule: omniMod

module top;
        logic clk = 0;

        simple_bus sb_intf(clk);        // instancja interfejsu

        memMod mem(sb_intf.slave);      // ma dostęp tylko do zadań slave

        cpuMod cpu(sb_intf.master);     // ma dostęp tylko do zadań master

        omniMod omni(sb_intf);          // ma dostęp do wszystkich zadań master i slave
endmodule
```

14.7.3. Eksportowanie metod interfejsów

Eksportowanie metod interfejsu pozwala na zdefiniowanie zadania lub funkcji w jednym module i wywołanie ich w innym module przez użycie modportów do kontroli dostępu do metod interfejsu.

W poniższym przykładzie modport *slave* interfejsu *simple_bus* eksportuje metody *Read* i *Write*, a modport master importuje metody *Read* i *Write*. Zadania *Read* i *Write* są zadeklarowane w module *memMod* i przekazywane (eksportowane) przez uniwersalny port interfejsu. Moduł *cpuMod* wywołuje metody *Read* i *Write*, które są importowane przez port interfejsu uniwersalnego.

```
interface simple_bus (input logic clk);              // definicja interfejsu
        logic req, gnt;
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;

        modport slave       ( input req, addr, mode, start, clk,
                              output gnt, rdy,
                              ref data,
                              // eksport z modułu za pomocą modportu slave
                              export Read,
                                     Write);

        modport master      (input gnt, rdy, clk,
                              output req, addr, mode, start,
                              ref data,
                              // pełny prototyp zadania wymagany do importu
                              import task Read(input logic [7:0] raddr),
                                     task Write(input logic [7:0] waddr));
endinterface: simple_bus

module memMod(interface a);                  // uniwersalny port interfejsu
        logic avail;

        task a.Read;                         // metoda Read
                avail = 0;
                ...
                avail = 1;
        endtask

        task a.Write;                        // metoda Write
                avail = 0;
```

```
            ...
            avail = 1;
        endtask
endmodule

module cpuMod(interface b);                 // uniwersalny port interfejsu
        enum {read, write} instr;
        logic [7:0] raddr;

        always @(posedge b.clk)
            if (instr == read)
                b.Read(raddr);              // import metody Read przez interfejs b
            else
                b.Write(raddr);             // import metody Write przez interfejs b
endmodule

module top;
        logic clk = 0;

        simple_bus sb_intf(clk);            // instancja interfejsu

        memMod mem(sb_intf.slave);          // eksportuje zadania Read i Write
        cpuMod cpu(sb_intf.master);         // importuje zadania Read i Write
endmodule
```

Metody interfejsu mogą być również eksportowane bez użycia modportów. W takim przypadku prototyp metody ze słowem kluczowym **extern** nie jest deklarowany w modportach, ale w ciele interfejsu. Na przykład:

```
interface chip_bus (input logic clock, resetN);
        logic request, grant, ready;
        logic [63:0] address, data;

        extern function check(input logic parity,                   // prototyp metody check
                              input logic [63:0] data);

        modport master (output request, ...);                       // modport master

        modport slave (input request, ...                           // modport slave
            import function check (input logic parity,              // import metody check
                                   input logic [63:0] data) );
endinterface
```

```
module CPU (chip_bus.master io);
        function check (input logic parity, input logic [63:0] data); // opis metody check
                ...
        endfunction
        ...
endmodule
```

Zauważmy, że eksport metod interfejsu nie podlega syntezie i jest używany tylko do celów symulacji.

14.8. Zastosowanie w interfejsach bloków proceduralnych

Oprócz zadań i funkcji interfejsy mogą zawierać bloki proceduralne i przypisania ciągłe. Dzięki temu mogą opisywać funkcjonalności za pomocą bloków proceduralnych **always**, **always_comb**, **always_ff**, **always_latch**, **initial** lub **final** i operatorów przypisania **assign**. Interfejs może również zawierać bloki programowe **program** do weryfikacji protokołów.

Jednym z zastosowań użycia deklaracji procedur i bloków w interfejsie jest stworzenie w nim narzędzia do weryfikacji protokołów. Za każdym razem, gdy wartości są przekazywane przez interfejs, wbudowane walidatory protokołu mogą sprawdzić, czy wszystkie wymagania protokołu zostały spełnione.

14.9. Interfejsy parametryzowane

Interfejsy w języku SystemVerilog, podobnie jak moduły, mogą mieć parametry i operatory **generate**. Pozwala to na tworzenie *interfejsów parametryzowanych* (*parameterized interfaces*), które mogą dostosować rozmiary wektorów i typy danych podczas deklarowania każdej konkretnej instancji interfejsu. Przypomnijmy, że w języku SystemVerilog jako parametry mogą być używane nie tylko wyrażenia stałe, lecz także typy danych.

Operatory **generate** mogą być użyte do dostosowania funkcjonalności interfejsu w zależności od podanych parametrów.

W poniższym przykładzie używamy modportów *int_io* i *fp_io* do przekazywania danych całkowitych lub rzeczywistych przez ten sam interfejs. Parametrem jest typ danych, więc każda instancja interfejsu może być skonfigurowana do korzystania z danych całkowitych lub rzeczywistych.

```
interface math_bus #(parameter type DTYPE = int)    // parametryzowany typ DTYPE
                                                    // domyślnie int
        (input logic clock);                        // porty interfejsu
```

```
        DTYPE a, b, result;                              // zmienne parametryzowane
        ...
        task Read (output DTYPE a, b);       // metoda z wyjściem typu parametryzowanego
                ...  // wykonywanie handshake, aby uzyskać wartości a i b
        endtask

        modport int_io (import Read,
                        input clock,
                        output result);

        modport fp_io (import Read,
                        input clock,
                        output result);
endinterface

module top (input logic clock, resetN);
        // tworzenie instancji interfejsu
        math_bus bus_a(clock);       // domyślnym typem danych jest int
        math_bus (#.DTYPE(real)) bus_b(clock);    // używany typ danych to real

        // połączenie interfejsu przy użyciu typu danych int
        integer_math_unit i1 (bus_a.int_io);

        // połączenie interfejsu przy użyciu typu danych real
        floating_point_unit i2 (bus_b.fp_io);

endmodule
```

14.10. Konstrukcje interfejsów niepodlegające syntezie

Interfejsy SystemVerilog mają szereg innych konstrukcji, które nie są syntezowalne, ale znacznie zwiększają ich możliwości w zakresie symulacji systemów cyfrowych. Do tych konstrukcji należą:

- wyrażenia modportów;
- bloki zegarowe w interfejsach;
- bloki specyfikacji w interfejsach;
- interfejsy wirtualne;
- modporty wirtualne.

Konstrukcje interfejsów niesyntezowalnych nie są przedmiotem rozważań w tej książce.

14.11. Wnioski

Interfejsy SystemVerilog przeznaczone są do grupowania sygnałów komunikacyjnych przesyłanych między jednostkami systemu cyfrowego. W uproszczonej formie interfejs to nazwany pakiet węzłów i zmiennych. Interfejsy umożliwiają zastąpienie grupy nazw portów jedną nazwą, dzięki czemu grupa sygnałów może być przekazywana między jednostkami jako pojedynczy port.

Dodatkowo interfejsy pozwalają na grupowanie funkcjonalności w celu implementacji protokołów komunikacyjnych i ich weryfikacji.

Interfejs może mieć parametry i stałe. Typy elementów mogą być deklarowane w interfejsach, a także przekazywane w nich jako parametry.

Interfejsy mogą zawierać bloki proceduralne (**always** i **initial**), operatory przypisania ciągłego oraz zadania i funkcje realizujące funkcjonalności interfejsu.

Te same sygnały interfejsu mogą mieć różne kierunki w różnych modułach. Do wskazania kierunku portów w określonych modułach są używane *modporty* (*modports*).

Podprogramy (zadania i funkcje) interfejsów mogą być zdefiniowane w jednym module i wywoływane w innym przy użyciu konstrukcji **export** i **import**.

Interfejsy, podobnie jak moduły, mogą mieć własne porty. Dzięki temu do interfejsu mogą być przekazywane sygnały, które w stosunku do niego są zewnętrzne. Interfejsy mogą również zawierać deklaracje dowolnych typów sieci i zmiennych, a także typów użytkownika.

W przypadku stosowania interfejsów w celu uproszczenia opisu portów modułów można stosować notacje **.name** oraz **.***.

Nazwy interfejsów mogą być zadeklarowane w kodzie projektu jako globalne lub lokalne.

Porty modułu, przez które przesyłane są interfejsy, nazywane są *portami interfejsu* (*interface ports*).

Porty interfejsów w modułach mogą być zadeklarowane jawnie. W tym celu określa się typ interfejsu zamiast kierunku portu (**input**, **output**, **inout** lub **ref**). Jawnie zadeklarowany port interfejsu może być podłączony tylko do interfejsu o tej samej nazwie.

Uniwersalny port interfejsu w module jest deklarowany za pomocą słowa kluczowego **interface** i może łączyć się z różnymi interfejsami.

Przed użyciem interfejsu w projekcie należy utworzyć instancję interfejsu. Jest ona tworzona w taki sam sposób jak instancje modułów.

Jeśli interfejs został zadeklarowany z portami, to podczas tworzenia instancji interfejsu, w przeciwieństwie do modułów, obowiązkowe jest określenie sygnałów podłączonych do portów interfejsu.

Połączenia interfejsu z portami modułu są określane podczas tworzenia instancji modułu.

Jeśli port interfejsu modułu jest jawnie zadeklarowany z nazwą interfejsu, port ten może być podłączony tylko do instancji interfejsu określonego typu.

Jeśli port interfejsu modułu jest zadeklarowany ze słowem kluczowym **interface** jako uniwersalny, może on być połączony z instancją interfejsu dowolnego typu.

Port interfejsu może być również zdefiniowany jako interfejs. Dzięki temu jeden interfejs może połączyć się z innym interfejsem.

W module, który ma port interfejsu, możliwe jest odniesienie się do sygnałów w ramach interfejsu. Odbywa się to poprzez podanie nazwy portu interfejsu i określenie nazwy sygnału z kropką.

Modporty (*modports*) w interfejsach służą do określenia kierunku sygnałów. Definicje modportów nie zawierają rozmiarów wektorów i typów sygnałów. Deklaracja modportu określa jedynie kierunek sygnałów: **input**, **output**, **inout** lub **ref**.

To, który modport interfejsu powinien być używany przez daną instancję modułu, jest określane przez podanie nazwy modportu podczas tworzenia instancji modułu lub deklarowania jego portu.

Moduły mogą łączyć się z interfejsem bez określania konkretnego modportu. W tym przypadku przyjmuje się, że wszystkie węzły w interfejsie mają kierunek **inout**, a wszystkie zmienne w interfejsie są typu **ref**.

Modporty pozwalają ograniczyć dostęp poszczególnych modułów do sygnałów interfejsu. Jeśli moduł jest podłączony do interfejsu bez określenia modportu, moduł ma dostęp do wszystkich sygnałów interfejsu.

Interfejsy mogą zawierać sygnały wewnętrzne, które nie są widoczne w żadnym ze zdefiniowanych modportów.

Język SystemVerilog pozwala na zawarcie funkcjonalności interfejsu bezpośrednio w samym interfejsie w postaci zadań i funkcji. Zadania i funkcje zawarte w interfejsie nazywane są *metodami interfejsu* (*interface methods*), które mogą pracować z dowolnymi sygnałami wewnętrznymi interfejsu.

Oprócz zadań i funkcji interfejsy mogą zawierać bloki proceduralne i przypisania ciągłe, jak też bloki programowe (**program**) do weryfikacji protokołu.

Metody interfejsu są opisane w ramach interfejsu i mogą być stosowane w modułach, do których są przekazywane przez porty modułu.

Modporty interfejsów mogą importować metody interfejsów za pomocą słowa kluczowego **import**. Istnieją dwa sposoby ich importowania: przy użyciu nazwy zadania lub funkcji oraz przy użyciu prototypu zadania bądź funkcji.

Poprzez eksportowanie metod interfejsu można zdefiniować zadanie albo funkcję w jednym module i wywołać ją w innym module, używając modportów jako kontrolerów dostępu do metod interfejsu.

Metody interfejsu mogą być również eksportowane bez użycia modportów. W tym przypadku prototypowa metoda ze słowem kluczowym **extern** jest zadeklarowana w ciele interfejsu zamiast modportów. Eksport metod interfejsu nie podlega syntezie i służy jedynie do celów symulacji.

Interfejsy w SystemVerilog, podobnie jak moduły, mogą mieć parametry i operatory **generate**, a jako ich parametry mogą być używane nie tylko wyrażenia stałe, lecz także typy danych.

15. Generowanie kodu

Załóżmy, że chcemy opisać sumator z szeregowym przeniesieniem z elementów jednobitowych. Jeśli rozmiar słów wejściowych jest mały, nie jest to problemem. Jednak gdy rozmiar słowa wynosi 1024 bity lub więcej, pisanie kodu źródłowego staje się bardzo czasochłonne. Jest to częsty problem podczas pracy z magistralami (szynami) o dużych rozmiarach.

Innym problemem, często spotykanym podczas pisania kodu źródłowego, jest wybór odpowiedniego fragmentu z kilku możliwych. Na przykład konieczne jest uwzględnienie najbardziej pasującej metody przetwarzania danych w zależności od wielkości operandów.

Aby rozwiązać te problemy, SystemVerilog dostarcza mechanizmu automatycznego generowania kodu projektu za pomocą *konstrukcji generacji* (*generate constructs*).

Konstrukcje te są przeznaczone do tworzenia fragmentów kodu języka SystemVerilog. W nawiązaniu do języków programowania konstrukcje generacji działają jak preprocesor do przetwarzania kodu źródłowego przed jego kompilacją – wstępnie przetwarzają kod źródłowy przed jego wykonaniem za pomocą narzędzia projektowego (syntezatora lub symulatora). Konstrukcje generacji pozwalają na tworzenie sekwencji powtarzających się fragmentów kodu, które nieznacznie różnią się od siebie; na wybieranie fragmentów kodu, jeśli spełnione są określone warunki itp. Ponieważ konstrukcje generacji dokonują konwersji kodu źródłowego przed jego wykonaniem, wszystkie używane w nich wyrażenia muszą być stałe.

W językach programowania operatory preprocesora i operatory języka często znacznie się różnią (np. Assembler, C). W SystemVerilog składnia operatorów używanych w konstrukcjach generacji pokrywa się ze składnią operatorów samego języka SystemVerilog. Należą do nich operatory **for**, **if** i **case**.

15.1. Składnia konstrukcji generate

Konstrukcje generacji są zdefiniowane wewnątrz modułu i używane do generowania dla niego kodu. Uogólniona składnia konstrukcji generacji jest następująca:

genvar *genvar_name*, ... ;
generate

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

gdzie: **genvar**, **generate** i **endgenerate** są słowami kluczowymi; *genvar_name* jest zmienną typu **genvar**; *generate_items* są elementami bloku generacji.

Zmienna typu **genvar** musi być dodatnią wartością całkowitą. Jest ona używana podczas generowania fragmentów kodu do kompilacji na podstawie pewnego fragmentu kodu źródłowego (podobnie jak iterator w instrukcji **for**). Wykorzystuje się ją tylko wewnątrz konstrukcji generacji, może mieć zdefiniowane wartości jedynie w czasie generowania kodu i nie istnieje w czasie wykonania. Zmienne typu **genvar** mogą być deklarowane zarówno wewnątrz, jak i na zewnątrz konstrukcji generacji. W tym drugim przypadku mogą być używane także w innych konstrukcjach generacji.

Elementami konstrukcji generacji (*generate_items*) mogą być:

- operator przypisania do zmiennej typu **genvar**;
- deklaracje sieci i zmiennych;
- instancje modułów i prymitywów;
- bloki proceduralne;
- deklaracje zadań i funkcji;
- operatory **if**-**else**, **case** i **for**.

Format deklaracji przypisania zmiennej typu **genvar**:

```
genvar_name = constant_expression;
```

gdzie *genvar_name* jest zmienną typu **genvar**, a *constant_expression* jest wyrażeniem stałym (tj. wyrażeniem zwracającym stałą).

Słowa kluczowe **generate** i **endgenerate** definiują region generowania w module, w którym mogą się pojawić konstrukcje generacji. Słowa te są opcjonalne. W takim przypadku region generowania jest definiowany poprzez użycie operatora **for**, **if** lub **case** poza blokami proceduralnymi.

Konstrukcje generacji mogą zawierać *bloki generacji* (*generate blocks*), które mają następujący format:

```
begin [ : generate_block_name ]
    {generate_item }
end [ : generate_block_name ]
```

gdzie *generate_block_name* jest nazwą bloku generacji, a *generate_item* jest elementem konstrukcji generacji.

Przetwarzanie bloków generacji przez kompilator skutkuje pustymi lub wielokrotnymi instancjami bloków generacji. Są one podobne do instancji modułu, tworzą nowy poziom w hierarchii nazw projektów. W ten sposób powstają instancje elementów generacji opisanych w ramach bloku generacji. Konstrukcje te zachowują się tak, jakby zostały stworzone w wyniku utworzenia instancji modułu. Do nazw obiektów w instancji nazwanego bloku generacji można się odwoływać hierarchicznie. Parametry zadeklarowane w blokach generacji są traktowane jak parametry lokalne (**localparam**).

Wszystkie konstrukcje języka SystemVerilog są podzielone na dwa typy: *konstrukcje generacji pętli* (*loop generate constructs*) i *konstrukcje generacji warunkowej* (*conditional generate constructs*). Te pierwsze pozwalają na tworzenie wielu instancji jednego bloku generacji, drugie natomiast tworzą (lub nie) jeden blok generacji ze zbioru alternatywnych bloków generacji.

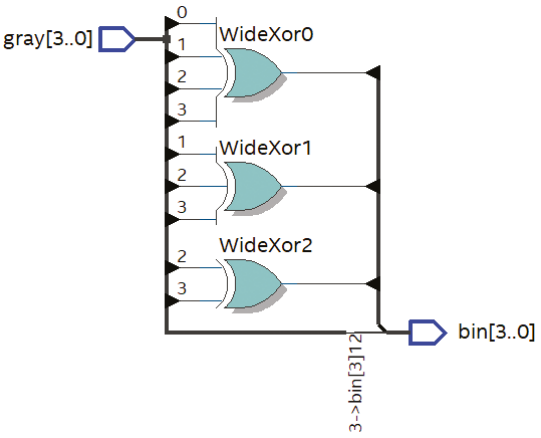
15.2. Konstrukcje generacji pętli

Konstrukcje generacji pętli pozwalają na tworzenie wielu instancji bloku generacji przy użyciu operatora **for**. Jego składnia do generacji kodu jest podobna do składni proceduralnego operatora **for**. Różnica polega na tym, że zmienna pętli musi być zadeklarowana jako zmienna typu **genvar**. Jest ona używana przez kompilator jako liczba całkowita, nie powinna być używana poza konstrukcją generacji pętli, nie istnieje też podczas symulacji.

Przykład 15.1. Użycie operatora generacji **for** do opisu konwertera kodu Graya na kod binarny

```
module gray_bin # (parameter N = 4)
        (input [N-1:0] gray,                    // wartością wejściową jest kod Graya
        output [N-1:0] bin);                    // wartość wyjściowa to kod binarny

        genvar i;                               // i jest zmienną generacji

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

Wyniki syntezy projektu gray_bin dla N = 4 przedstawiono na rys. 15.1.



RYS. 15.1. Wyniki syntezy konwertera kodu Graya na kod binarny z wykorzystaniem operatora generacji **for**

ŹRÓDŁO: oprac. własne.

Dopuszczalne są również wielopoziomowe pętle generacji. Na przykład:

```
parameter N = 2;
genvar i, j, k, m;                          // zmienne generacji
generate                                    // region generacji
    for (i=0; i<N; i=i+1) begin:B1                          // pętla generacji
        ...
        for (j=0; j<N; j=j+1) begin:B2                      // pętla generacji
            ...
            for (k=0; k<N; k=k+1) begin:B3                  // pętla generacji
                ...
            end
        end
        if (i>0) begin:B4                                   // blok generacji
            for (m=0; m<N; m=m+1) begin:B5                  // pętla generacji
                ...
            end
        end
    end
endgenerate
```

15.3. Konstrukcje generacji warunkowej

Konstrukcje generacji warunkowej oparte na wyrażeniach stałych pozwalają na wybranie nie więcej niż jednego bloku generacji ze zbioru alternatywnych bloków generacji. Do implementacji konstrukcji generacji warunkowej służą operatory **if** i **case**, których składnia jest podobna do składni odpowiednich operatorów proceduralnych.

Ponieważ tworzony jest co najwyżej jeden blok generacji alternatywnej, można mieć kilka bloków o tej samej nazwie w jednej konstrukcji generacji warunkowej. Jeśli blok generacyjny wybrany do utworzenia instancji ma nazwę, to deklaruje ona instancję bloku generacji i jest nazwą tworzonego kontekstu. Konstrukcje generacji warunkowej pozwalają modułowi zawierać instancję samego siebie.

Najczęstszym sposobem jest stworzenie schematu **if-else-if** z dowolną liczbą elementów **else-if**, z których wszystkie mogą mieć bloki o tej samej nazwie, ponieważ tylko jeden blok zostanie wybrany do utworzenia instancji. Na przykład:

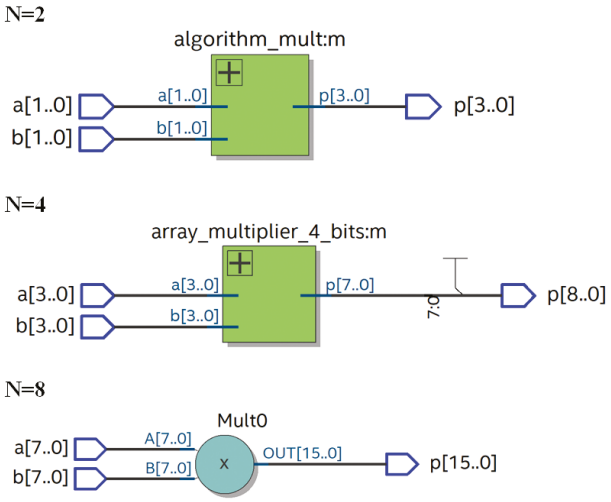
```
module test;
        parameter p = 0, q = 0;
        wire a, b, c;

        // słowa kluczowe generate i endgenerate są tutaj pominięte
        if (p == 1)
                if (q == 0)
                        begin : u1              // tworzona jest instancja and o nazwie test.u1.g1
                                and g1(a, b, c);
                        end
                        else if (q == 2)
                                begin : u1      // tworzona instancja or o nazwie test.u1.g1
                                        or g1(a, b, c);
                                end
                        else ;
                else if (p == 2)
                        case (q)
                                0, 1, 2:
                                        begin : u1      // tworzona jest instancja xor
                                                        // o nazwie test.u1.g1
                                                xor g1(a, b, c);
                                        end
                                default:
                                        begin : u1      // tworzona jest instancja xnor
                                                        // o nazwie test.u1.g1
                                                xnor g1(a, b, c);
                                        end
                        endcase
endmodule
```

Zauważmy, że w powyższym przykładzie konstrukcję generacji definiuje się, umieszczając operator **if** poza blokami procedur, przy czym wszystkie bloki generacji mają tę samą nazwę *u1*. Konstrukcja generacji wybierze nie więcej niż jeden z bloków generacji o nazwie *u1*. Hierarchiczną nazwą instancji bramki w tym bloku będzie *test.u1.g1*.

Przykład 15.2. Użycie operatora generacji **if** do wyboru metody mnożenia w zależności od wielkości operandów

```
module multiplier #(parameter N=8)
        (input [N-1:0] a, b,                    // liczby do pomnożenia
        output [2*N-1:0] p);                    // wynik

        generate
                if (N < 4)                      // wybór metody mnożenia dla małych liczb
                        algorithm_mult #(N) m(a,b,p);
                else if (N == 4)                // wybór metody mnożenia przy N = 4
                        array_multiplier_4_bits m(a,b,p);
                else                            // wybór metody mnożenia dla pozostałych liczb
                        assign p = a * b;
        endgenerate
endmodule
```

W powyższym przykładzie zastosowano różne algorytmy mnożenia liczb binarnych, w zależności od wielkości argumentów N. Dla $N < 4$ stosowana jest algorytmiczna metoda mnożenia, która odpowiada prostemu algorytmowi szkolnemu, dla $N = 4$ stosowany jest macierzowy układ mnożący, a dla $N > 4$ metoda mnożenia zaimplementowana w stosowanym kompilatorze. Wyniki syntezy z przykładu 15.2 dla różnych wartości N przedstawiono na rys. 15.2.



RYS. 15.2. Wyniki syntezy układu mnożącego przy zastosowaniu różnych metod mnożenia w zależności od wielkości argumentów

ŹRÓDŁO: oprac. własne.

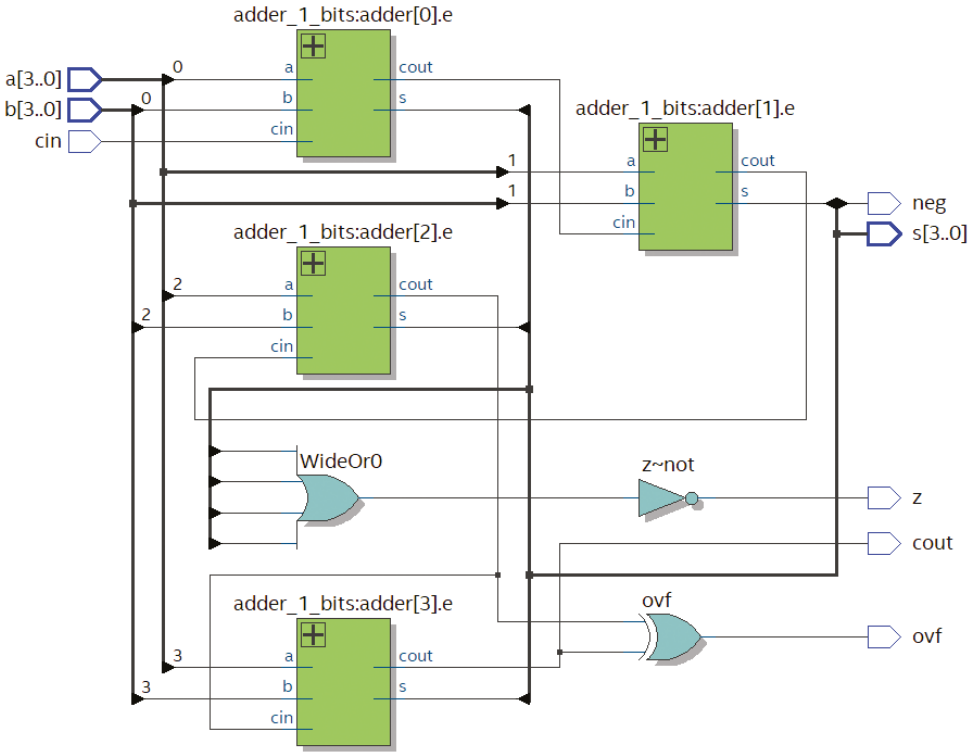
Inny przykład *sumatora z przeniesieniem szeregowym* (*ripple-carry adder* – RCA) pokazuje wspólne użycie operatorów generacji **for** i **case**.

Przykład 15.3. Użycie operatorów generacji **for** i **case** w sumatorze z szeregowym przeniesieniem

```
module RCA_adder #(parameter N=4)
        (input [N-1:0] a, b,                    // słowa wejściowe
        input cin,                              // przeniesienie wejściowe
        output [N-1:0] s,                       // suma
        output cout, neg, ovf,z);               // flagi
        wire [N-1:0] c;

        genvar i;                               //zmienna generacji

        generate                                // konstrukcja generacji
                for (i=0; i<N; i=i+1)
begin: adder                                    // blok generacji
                        case(i)
                                0:      adder_1_bits e(a[i],b[i],cin,s[i],c[i]);      // bit zerowy
                                N-1: begin                                            // ostatni bit
                                        adder_1_bits e(a[i],b[i],c[i-1],s[i],cout);
                                        assign neg = s[i];              // wynik jest negatywny
                                        assign ovf = cout ^ c[i-1];     // nadmiar
                                        assign z = ~(|s);               // wynik jest równy zero
                                    end
                                default:                                              // pozostałe bity
                                        adder_1_bits e(a[i],b[i],c[i-1],s[i],c[i]);
                        endcase
                end
        endgenerate
endmodule
```

Wyniki syntezy projektu RCA_adder przedstawiono na rys. 15.3.

Zwróćmy uwagę na nazwy jednobitowych sumatorów na schemacie z rys. 15.3. Są one nazwą hierarchiczną składającą się z nazwy modułu, nazwy bloku **begin-end**, operatora pętli **for** z numerem iteracji (adder[i]), w którym ta instancja jest generowana jako indeks, oraz nazwy wygenerowanej instancji (e).



RYS. 15.3. **Wyniki syntezy sumatora z przeniesieniem szeregowym**

ŹRÓDŁO: **oprac. własne.**

Zauważmy, że w deklaracjach **for** i **case** nazwy instancji modułów mogą się pokrywać, ponieważ nazwy generowanych instancji są zawsze różne.

W systemie Quartus słowa kluczowe **generate** i **endgenerate** dla konstrukcji generacji są obowiązkowe. Innymi słowy, użycie operatora **for**, **if** i **case** poza blokami proceduralnymi powoduje błąd kompilacji. W systemie Quartus nie można również zadeklarować zmiennej **genvar** wewnątrz operatora **for**.

15.4. Nazwy zewnętrzne nienazwanych bloków generacji

Chociaż *nienazwany* (*unnamed*) *blok generacji* nie ma nazwy, która może być używana w hierarchii, musi mieć nazwę, przez którą może być wywoływany przez zewnętrzne interfejsy. W tym celu każdemu nienazwanemu blokowi generacji nadaje się nazwę w następujący sposób.

Każdej konstrukcji generacji w danym kontekście (*scope*) przypisany jest numer. Dla konstrukcji, która pojawia się jako pierwsza w kodzie, przypisana jest liczba 1 i będzie ona zwiększana o 1 dla każdej kolejnej konstrukcji generacji w danym kontekście. Wszystkie nienazwane bloki generacji będą miały nazwę „genblk<n>", gdzie <n> jest numerem przypisanym przez zewnętrzną konstrukcję generacji. Jeśli taka nazwa koliduje z jawnie zadeklarowaną nazwą, przed numerem dodawane są wiodące zera, aby wyeliminować konflikt. Na przykład:

```
module top;

    parameter genblk2 = 0;
    genvar i;

    // Następny blok generacji jest niejawnie nazwany genblk1
    if (genblk2) logic a;          // top.genblk1.a
    else logic b;                  // top.genblk1.b

    // Następny blok generacyjny jest niejawnie nazwany genblk02,
    // ponieważ genblk2 jest już zadeklarowanym identyfikatorem
    if (genblk2) logic a;          // top.genblk02.a
    else logic b;                  // top.genblk02.b

    // Następny blok generacji miałby nazwę genblk3, ale wyraźnie nazwany g1
    for (i = 0; i < 1; i = i + 1) begin : g1          //nazwa bloku
        // Następny blok generacji jest niejawnie nazwany genblk1
        // jako pierwszy blok zagnieżdżony w g1
        if (1) logic a;                               // top.g1[0].genblk1.a
    end

    // Poniższy blok generacji jest niejawnie nazwany genblk4, ponieważ
    // należy do czwartej konstrukcji generacji w obszarze „top"
    for (i = 0; i < 1; i = i + 1)
        // Następny blok generacji jest niejawnie nazwany genblk1
        // jako pierwszy zagnieżdżony blok generacji w bloku genblk4
        if (1) logic a;          // top.genblk4[0].genblk1.a

    // Następny blok generacji jest niejawnie nazwany genblk5
    if (1) logic a;              // top.genblk5.a

endmodule
```

15.5. Wnioski

Język SystemVerilog zapewnia mechanizm automatycznej generacji kodu projektu za pomocą *konstrukcji generacji* (*generate constructs*).

Konstrukcje generacji są przeznaczone do generacji fragmentów kodu języka SystemVerilog. Podobnie jak preprocesor w językach programowania, służą one do przetwarzania kodu źródłowego przed jego kompilacją.

Konstrukcje generacji pozwalają na tworzenie sekwencji powtarzających się fragmentów kodu, które nieznacznie różnią się od siebie, a także na wybieranie fragmentów kodu do realizacji, jeśli spełnione są określone warunki.

Konstrukcje generacji są zdefiniowane wewnątrz modułu i używane do generacji kodu dla niego.

Zmienne typu **genvar** są używane w operatorach pętli **for** podczas generacji fragmentu kodu źródłowego.

Zmienne typu **genvar** mogą być deklarowane zarówno wewnątrz, jak i na zewnątrz operatora generacji, w tym drugim przypadku mogą być używane również w innych operatorach generacji.

Elementami konstrukcji generacji (*generate_items*) mogą być: operator przypisania do zmiennej typu **genvar**; deklaracje sieci i zmiennych; instancje modułów i prymitywów; bloki procedur; deklaracje zadań i funkcji; operatory generacji **if-else**, **case** i **for**.

Wszystkie wyrażenia używane w konstrukcjach generacji muszą być stałe.

Konstrukcje generacji mogą zawierać bloki generacji zdefiniowane przez słowa kluczowe **begin**-**end**.

Słowa kluczowe **generate** i **endgenerate** definiują region generacji w module, w którym mogą pojawić się konstrukcje generacji. Słowa te mogą być opcjonalne. W przypadku ich braku region generacji jest definiowany poprzez użycie operatora **for**, **if** lub **case** poza blokami proceduralnymi. W systemie Quartus słowa kluczowe **generate** i **endgenerate** dla konstrukcji generacji są obowiązkowe.

Instancja bloku generacji tworzy nowy poziom hierarchii nazw projektów. W ten sposób powstają instancje elementów generacji opisanych w ramach bloku generacji. Konstrukcje te zachowują się tak, jakby powstały w wyniku utworzenia instancji modułu. Do nazw obiektów w utworzonej instancji nazwanego bloku generacji można się odwoływać hierarchicznie. Parametry zadeklarowane w konstrukcjach generacji są traktowane jako parametry lokalne.

Istnieją dwa rodzaje konstrukcji generacji: *konstrukcje generacji pętli* i *konstrukcje generacji warunkowej*. Te pierwsze pozwalają na tworzenie wielu instancji jednego bloku generacji, drugie natomiast tworzą (lub nie) jeden blok generacji ze zbioru alternatywnych bloków generacji.

Konstrukcje generacji pętli pozwalają na tworzenie wielu instancji bloku generacji przy użyciu operatora pętli **for**, który wykorzystuje zmienną typu **genvar**. Składnia operatora **for** do generacji kodu jest podobna do składni proceduralnego operatora **for**. Dopuszczalne są również wielopoziomowe pętle generacji.

Konstrukcje generacji warunkowej na podstawie wyrażeń stałych wybierają nie więcej niż jeden blok generacji ze zbioru alternatywnych bloków generacji. Do realizacji konstrukcji generacji warunkowej używa się operatorów **if** i **case**, których składnia jest podobna do składni odpowiednich operatorów proceduralnych.

W jednej konstrukcji generacji warunkowej dozwolone jest kilka bloków o tej samej nazwie.

Konstrukcje generacji warunkowej pozwalają modułowi zawierać instancję samego siebie.

Wszystkie nienazwane bloki generacji mają automatycznie przypisywane przez kompilator nazwy „genblk<n>", gdzie <n> jest numerem przypisanym przez zewnętrzną konstrukcję generacji. Jeśli taka nazwa koliduje z jawnie zadeklarowaną nazwą, przed numerem dodawane są wiodące zera, aby wyeliminować konflikt.

16. Dyrektywy kompilacji

Dyrektywy kompilacji (*compilation directives*) są poleceniami kompilatora i mówią mu, jak interpretować kod źródłowy projektu napisanego w SystemVerilog. Dyrektywy kompilatora mają wpływ na proces tworzenia kodu wynikowego projektu. Niektóre dyrektywy języka SystemVerilog pokrywają się z tymi obecnymi w większości języków programowania (dyrektywy dotyczące definicji makr, kompilacji warunkowej, dołączania plików), inne są specyficzne tylko dla tego języka (dyrektywy definiujące wartości jednostek czasu, domyślny typ obwodu, wartości logiczne dla niepodłączonych wejść itp.)

Wszystkie dyrektywy kompilacji języka SystemVerilog poprzedzone są znakiem (`) zwanym *akcentem* (*grave accent*) lub *odwróconym apostrofem* (kod ASCII 0x60). Zauważmy, że znak akcentu (`) jest inny niż znak apostrofu ('), który ma kod ASCII 0x27.

Zakres dyrektywy kompilacji zaczyna się w miejscu, w którym jest ona napotkana w kodzie źródłowym, i rozciąga się na wszystkie pliki bieżącej jednostki kompilacji aż do punktu, w którym inna dyrektywa jej nie zastąpi lub do zakończenia jednostki kompilacji.

Dyrektywy języka SystemVerilog pozwalają:

- powrócić do domyślnych wartości dyrektyw kompilacji;
- określić wartość jednostki czasu;
- pracować z makrodefinicjami;
- wykonywać warunkową kompilację projektu;
- określić domyślny typ sieci;
- określić wartości logiczne dla niepodłączonych wejść;
- definiować moduły komórkowe;
- pracować z pragmami;
- zawierać pliki kodu źródłowego;
- określać numery linii kodu źródłowego;
- określać nazwę pliku i numer linii w pliku, w którym wystąpił błąd;
- określać listę słów kluczowych dla danej wersji SystemVerilog.

16.1. Powrót do domyślnych wartości dyrektyw kompilacji

Wiele dyrektyw języka SystemVerilog ma wartości. Czasami konieczny jest powrót do ich wartości domyślnych. Dyrektywa **`resetall** zwraca wartość domyślną dla tych dyrektyw kompilatora, które mają wartości domyślne.

Format dyrektywy **`resetall**:

`resetall

Dyrektywa **`resetall** nie ma parametrów. Zaleca się, by była ona ustawiona na początku każdego źródłowego pliku tekstowego.

16.2. Definiowanie wartości jednostki czasu

Jednostki czasu (*time units*) są szeroko stosowane w różnych konstrukcjach języka SystemVerilog podczas opisywania projektów do symulacji. Wartość (czas trwania) pojedynczej jednostki czasu definiuje się za pomocą dyrektywy **`timescale**, która ma następujący format:

`timescale *time_unit base / precision base*

gdzie: *time_unit* jest wielkością opóźnienia dla pojedynczej jednostki czasu, może wynosić 1, 10 lub 100; *base* jest bazową jednostką czasu, może być *s* (sekundy), *ms* (milisekundy), *us* (mikrosekundy), *ns* (nanosekundy), *ps* (pikosekundy) i *fs* (femtosekundy); *precision* jest precyzyjną jednostką czasu, określa bazę jednostki czasu do reprezentowania czasu po przecinku; *base* po *precision* jest bazą jednostki czasu do reprezentowania czasu precyzyjnego.

Przykład użycia dyrektywy **`timescale**:

`timescale 1ns / 10ps

definiuje jednostki czasu o czasie trwania 1 nanosekundy z dokładnością (precyzją) do 10 pikosekund (0,01 nanosekundy) i dwóch pozycji po przecinku.

Zauważmy, że dyrektywa **`timescale** nie ma wartości domyślnej, więc musi być zawsze zdefiniowana w projektach, gdzie używane są jednostki czasu.

Dyrektywa **`timescale** pozwala uniezależnić kod projektu od konkretnych wartości jednostek czasu. Nie jest syntetyzowalna i jest używana tylko w symulacjach.

16.3. Makrodefinicje

W niektórych przypadkach może być wygodne zastąpienie niektórych prostych, często powtarzanych fragmentów kodu jedną krótką konstrukcją. W SystemVerilog do tego celu służą *makrodefinicje* (*macro definitions*). Mają one zastąpić jeden ciąg tekstowy w kodzie źródłowym innym i są szeroko stosowane w językach programowania. W SystemVerilog definicje makr definiuje się za pomocą dyrektywy **`define**, która ma następujące dwa formaty:

`define *macro_name text_string*

i

`define *macro_name (arguments) text_string (arguments)*

gdzie: *macro_name* to nazwa makrodefinicji; w późniejszym użyciu w kodzie źródłowym nazwa makra powinna być poprzedzona znakiem apostrofu odwróconego (`); *text_string* to łańcuch tekstowy, którym zostanie zastąpiona nazwa makrodefinicji w kodzie źródłowym; *arguments* to lista argumentów przekazywanych w makrodefinicji.

Pierwszy format służy do definiowania nazw stałych, a drugi do definiowania makr wykonujących proste funkcje. Przy opisie definicji makr mogą być używane komentarze.

Jeśli do opisu definicji makra wymagany jest więcej niż jeden wiersz, znak nowej linii poprzedzony jest odwrotnym ukośnikiem (\).

W makrach lewy nawias musi bezpośrednio następować po tekstowej nazwie makra, bez spacji pomiędzy nimi.

Formalne argumenty makr mogą mieć wartość domyślną. Jest ona określana przez dodanie znaku równości (=) po nazwie argumentu formalnego, a następnie tekstu domyślnego. Tekst domyślny jest zastępowany argumentem formalnym, jeśli nie podano odpowiadającego mu argumentu rzeczywistego.

Tekst domyślny może być jawnie określony jako pusty przez dodanie tokena = po formalnej nazwie argumentu, po którym następuje przecinek (lub prawy nawias, jeśli jest to ostatni argument na liście argumentów).

Argumenty rzeczywiste i wartości domyślne nie mogą zawierać przecinka ani prawego nawiasu poza pasującymi parami lewych i prawych nawiasów (), nawiasów kwadratowych [], nawiasów klamrowych {}, podwójnych cudzysłowów „” lub ukrytego identyfikatora.

Argument rzeczywisty może być pustym lub białym znakiem. W tym przypadku argument formalny jest zastępowany argumentem domyślnym lub niczym, jeśli nie określono wartości argumentu domyślnego.

Po zdefiniowaniu makra tekstowego można je wykorzystać w opisie źródła, używając znaku (`), po którym następuje nazwa makra. Kompilator zastępuje nazwę makra zdefiniowanym wcześniej łańcuchem *text_string* i wszelkie kolejne argumenty.

Przykłady makrodefinicji:

```
`define cycle 20                              // okres zegara jako cycle
 always  # (`cycle/2) clk = ~clk;             // generowanie sygnałów zegara clk
                                              // z okresem cycle

`define NAND (delay) nand # (delay)           // zapis prymitywu nand wielkimi literami

`NAND (3)          i1 (y, a, b);              // definicja instancji i1 prymitywu nand
`NAND (3:4:5)      i2 (o, c, d);              // definiowanie instancji i2 prymitywu nand
```

Przykłady definicji makr z wartościami domyślnymi:

```
`define MACRO1(a=5,b=`B`,c) $display(a,,b,,c);       // definicja makra MACRO1

`MACRO1( , 2, 3 )      // argument a jest pominięty, zastąpiony wartością domyślną 5
                       // rozciąga się na $display(5,,2,,3);
`MACRO1( 1 , , 3 )     // argument b pominięty, zastąpiony wartością domyślną `B`
                       // domyślnie został zastąpiony przez $display(1,,`B`,,3);
`MACRO1( , 2, )        // argument c jest pominięty,
                       // rozwija się do $display(5,,2,,);
`MACRO1( 1 )           // niedopuszczalne: b i c są pominięte, brak wartości domyślnej dla c
```

Aby anulować poprzednio zdefiniowaną nazwę definicji makra, należy użyć dyrektywy `**undef**, która ma następujący format:

```
`undef macro_name   .
```

Dyrektywa `**undefineall** anuluje wszystkie makra tekstowe zdefiniowane wcześniej przez dyrektywy `**define** w jednostce kompilacji. Dyrektywa ta nie ma żadnych argumentów i może być umieszczona w dowolnym miejscu kodu źródłowego.

16.3.1. Zastępowanie makr w łańcuchu tekstowym

Język SystemVerilog rozszerza dyrektywę `**define** języka Verilog do deklarowania definicji makr, pozwalając, aby makro zawierało pewne znaki specjalne.

Załóżmy, że potrzebujemy stworzyć makro, które będzie wypisywało nazwę zmiennej oraz jej wartość. Takie makro w Verilogu może wyglądać jak poniżej:

```
`define print (v) $display ("variable v = %h", v)
```

Podczas wpisywania kodu makra

```
`print (data);
```

zostanie wygenerowane następujące rozszerzenie:

```
$display ("variable v = %h ", data);
```

co nie jest dokładnie tym, czego chcieliśmy (zmienna zawsze ma nazwę v).

Język SystemVerilog pozwala na podstawienie argumentu wewnątrz łańcucha tekstowego makra, gdy cudzysłów poprzedzony jest znakiem apostrofu wstecznego (`). Wymagane makro w SystemVerilog jest opisane w następujący sposób:

```
`define print (v) $display (`"variable v = %h`", v)
```

podczas pisania makra w kodzie języka SystemVerilog

```
`print (data);
```

zostanie wygenerowana następująca linia:

```
$display ("variable data = %h ", data);
```

jest dokładnie tym, czego potrzebujemy: makro wypisuje nazwę zmiennej i jej wartość.

16.3.2. Tworzenie nazw identyfikatorów z makr

Z dyrektywą `**define** języka Verilog nie jest możliwe stworzenie nazwy identyfikatora poprzez połączenie dwóch lub więcej makr tekstowych. Powodem jest to, że między każdą częścią nazwy identyfikatora złożonego zawsze będzie spacja.

Język SystemVerilog pozwala na łączenie złożonych nazw identyfikatorów bez wprowadzania spacji poprzez użycie dwóch odwrotnych apostrofów (``).

Załóżmy np., że ma być zadeklarowana następująca magistrala:

```
bit d00_bit; wand d00_net = d00_bit;
...                                    // powtórzone 62 razy
bit d63_bit; wand d63_net = d63_bit;
```

To samo w języku SystemVerilog można opisać w następujący sposób:

```
`define my_net (name) bit name ``_bit; wand name ``_net \ = name ``_bit;

`my_net (d00)
....                                    // powtórzone 62 razy
`my_net (63)
```

16.4. Dyrektywy kompilacji warunkowej

Czasami możemy chcieć dołączyć fragment kodu do swojego kodu źródłowego pod pewnym warunkiem. W SystemVerilog można to zrobić za pomocą *dyrektyw kompilacji warunkowej* (*conditional compilation directives*).

Zawierają one następujące dyrektywy: **`ifdef**, **`ifndef**, **`else**, **`elsif** i **`endif**. Pozwalają włączyć poszczególne fragmenty kodu do kodu źródłowego, w zależności od tego, czy definicje makr *macro_name* zostały wcześniej zdefiniowane za pomocą dyrektywy **`define**. Format dyrektyw kompilacji warunkowej:

```
`ifdef  macro_name
`ifnde f macro_name
        source_code
`else
        source_code
`elsif macro_name
        source_code
`endif
```

gdzie *macro_name* jest nazwą definicji makra, a *source_code* jest fragmentem kodu źródłowego.

Przykład zastosowania dyrektyw kompilacji warunkowej:

```
`ifdef RTL
        wire y = a & b;         // implementacja y jako zmiennej sieciowej
`else
        and  #1 (y, a, b);      // implementacja y za pomocą prymitywu and
`endif
```

Dyrektywy kompilacji warunkowej są przydatne w następujących przypadkach:

- przy wyborze różnych poziomów reprezentacji projektu, takich jak poziom behawioralny, strukturalny czy RTL;
- przy wyborze różnych parametrów czasowych lub informacji strukturalnych;
- przy wyborze różnych *wymuszeń* (*stimulus*) dla danego przebiegu symulacji.

16.5. Definiowanie domyślnego typu sieci

W języku SystemVerilog sieci są domyślnie typu **wire**. Jeśli jednak cały układ pracuje w *logice bipolarnej* lub *ECL* (*emitter-coupled logic*), wygodna może być zmiana domyślnego typu układu. Domyślny typ obwodu może być zmieniony za pomocą dyrektywy **`default_nettype**, która ma następujące formaty:

`default_nettype *net_data_type*

`default_nettype *none*

gdzie *net_date_type* jest nowym domyślnym typem sieci.

Podczas korzystania z drugiego formatu z wartością typu *none* niejawna (*implicit*) definicja typu obwodu jest zabroniona, tj. każdy typ obwodu musi być zdefiniowany jawnie podczas definiowania obwodu.

Domyślnymi typami obwodu mogą być: **wire**, **tri**, **tri0**, **tri1**, **wand**, **triand**, **wor**, **trior**, **trireg** i **uwire**.

Przykład użycia dyrektywy **`default_nettype**:

```
`defoult_nettype wand          // definicja wand jako domyślny typ sieci
```

16.6. Definiowanie wartości logicznych dla niepodłączonych wejść

W niektórych przypadkach dla uzyskania dokładniejszych wyników symulacji konieczne jest zdefiniowanie wartości dla niepodłączonych wejść modułu. Dyrektywy **`unconnected_drive** i **`nounconnected_drive** służą do tego celu i mają następujące formaty:

`unconnected _drive pull1

`unconnected _drive pull0

`nounconnected _drive

gdzie słowo kluczowe **pull1** oznacza, że niepodłączone wejścia modułu będą domyślnie podciągnięte (*pull up*) do jedynki, a **pull0** do zera. Zastosowanie dyrektywy **`nounconnected_drive** wskazuje, że niepodłączone wejścia modułu są w stanie wysokiej impedancji, np.:

```
`unconnected _drive pull0          // podciąganie niepodłączonych wejść do masy
```

16.7. Identyfikacja modułów jako modułów komórkowych

Dyrektywy **`celldefine** i **`endcelldefine** oznaczają moduły jako *moduły komórkowe* (*cell modules*). Komórki są wykorzystywane przez niektóre procedury *interfejsu języka programowania* (*programming language interface* – PLI) i mogą być przydatne w aplikacjach do obliczania opóźnień.

Dyrektywa **`celldefine** określa, że wszystkie moduły następujące w kodzie są modułami komórkowymi. Dyrektywa **`endcelldefine** anuluje dyrektywę **`celldefine**.

16.8. Pragmy

Pragma jest ogólnym terminem używanym do określenia konstrukcji bez predefiniowanej semantyki językowej, która wpływa na to, jak syntezator powinien tłumaczyć kod na schemat.

Dyrektywa **`pragma** jest strukturalną specyfikacją, która zmienia interpretację kodu źródłowego SystemVerilog. Specyfikacja wprowadzona przez tę dyrektywę nazywana jest *pragm*ą. Składnia dyrektywy `pragma` jest następująca:

`pragma *pragma_name* { *pragma_expression* }

gdzie *pragma_name* jest nazwą pragmy, a { *pragma_expression* } jest listą wyrażeń pragmy.

Lista wyrażeń pragmy *pragma_expression* kwalifikuje zmodyfikowaną interpretację określoną przez *pragma_name*.

Wyrażenie pragmy *pragma_expression* może mieć trzy formy:

pragma_keyword
pragma_keyword = pragma_value
pragma_value

gdzie *pragma_keyword* to słowo kluczowe pragmy, a *pragma_value* to wartość pragmy.

Słowo kluczowe *pragma_keyword* może być prostym identyfikatorem, a wartość *pragma_value* może być liczbą, łańcuchem, identyfikatorem lub listą wyrażeń pragmy w nawiasach.

Nazwy pragm nierozpoznane przez implementację nie mają wpływu na kod źródłowy języka SystemVerilog.

Pragma **resetall** resetuje stan wszystkich nazw pragm rozpoznawanych przez implementację. Pragma **protect** jest używana do definiowania *chronionych kopert* (*protected envelopes*).

Poniższy przykład pokazuje użycie pragmy **protect** do określenia szyfrowania danych projektu. Metodą szyfrowania jest prosty szyfr substytucyjny, zwany „rot13", w którym każdy znak alfabetyczny jest zastępowany przez znak na pozycji o 13 dalszej w alfabecie. Znaki niealfabetyczne nie są zastępowane. Poniższe dane konstrukcyjne zawierają kopertę szyfrującą określającą wymagane zabezpieczenie.

```
module secret (a, b);
        input a;
        output b;

 `pragma protect encoding=(enctype="raw")
 `pragma protect data_method="x-caesar", data_keyname="rot13", begin
 `pragma protect
        runtime_license=(library="lic.so",feature="runSecret",entry="chk", match=42)
        logic b;

        initial
                begin
                        b = 0;
                end
        always
                begin
                        #5 b = a;
                end
 `pragma protect end

endmodule
```

Po przetworzeniu szyfrowania powstają następujące dane projektu. Koperta deszyfrująca jest opisana z kodowaniem "raw", aby szyfrowanie substytucyjne było widoczne bezpośrednio.

16.9. Włączenie plików

Dowolne pliki tekstowe mogą być dołączone do kodu źródłowego SystemVerilog za pomocą dyrektywy `include. Formaty `**include**:

`include "*filename*"

`include <*filename*>

gdzie *filename* jest nazwą dołączonego pliku.

Nazwa pliku *filename* może być pełną lub względną nazwą ścieżki. Może być ujęta w cudzysłowy lub nawiasy kątowe, co wpływa na wyszukiwanie pliku przez kompilator w następujący sposób:

- jeżeli nazwa pliku jest ujęta w podwójny cudzysłów („*filename*"), to wyszukiwanie jest wykonywane dla względnej ścieżki bieżącego katalogu roboczego kompilatora i lokalizacji określonych przez użytkownika;
- jeśli nazwa pliku jest ujęta w nawiasy kątowe (<*filename*>), to przeszukiwana jest tylko zależna od implementacji lokalizacja zawierająca pliki określone przez standard języka; względne nazwy ścieżek są interpretowane względem tej lokalizacji.

Jeśli nazwa pliku *filename* jest ścieżką absolutną, to tylko ta nazwa pliku jest uwzględniana i można używać tylko podwójnych cudzysłowów.

Plik włączony do źródła za pomocą dyrektywy **`include** może zawierać inne dyrektywy kompilatora **`include**. Liczba poziomów zagnieżdżenia dla dołączonych plików zależy od narzędzia projektowego, ale nie może być mniejsza niż 15.

Przykłady użycia dyrektywy **`include**:

```
`include "project_macros.sv"   // ustalana jest lokalizacja plików
`include "parts/count.v"       // ścieżka względna
`include "fileB"               // katalog roboczy kompilatora
`include <List.vh>             // lokalizacja pliku List.vh jest określona przez implementację
```

16.10. Określanie numerów linii kodu źródłowego

Dla narzędzi języka SystemVerilog ważne jest śledzenie nazw plików źródłowych SystemVerilog i numerów linii w plikach. Informacje te mogą być wykorzystane do raportowania błędów lub debugowania kodu źródłowego i mogą być dostępne przez interfejs języka programowania PLI.

Kod źródłowy języka SystemVerilog może być wstępnie przetwarzany przez preprocesor. W rezultacie linie mogą być dodawane do pliku źródłowego lub wiele linii może być konkatenowanych w jedną linię.

Dyrektywa **`line** ma na celu wskazanie numeru linii kodu źródłowego i nazwy pliku źródłowego. Pozwala na zachowanie lokalizacji elementów składniowych w pliku źródłowym, jeśli inny proces zmieni kod źródłowy. W rezultacie kompilator może poprawnie odwołać się do lokalizacji w kodzie źródłowym.

Dyrektywy **`line** nie są przeznaczone do ręcznego wstawiania do kodu SystemVerilog. Narzędzia projektowe zwykle wstawiają dyrektywy **`line** do kodu automatycznie. Jednocześnie w tekście mogą wystąpić dyrektywy **`line**.

Składnia dyrektywy `**line**:

`**line** *number* "*filename*" *level*

gdzie: *numer* to nowy numer następnej linii tekstu; *filename* to nowa nazwa pliku; *level* to jedna z liczb: 0, 1 lub 2. Wartość 1 oznacza, że następna linia jest pierwszą linią po wstawieniu pliku include. Wartość 2 wskazuje, że następna linia jest pierwszą linią po opuszczeniu pliku include. Wartość 0 oznacza każdą inną linię. Na przykład:

```
`line 3 „orig.v" 2        // ta linia to linia 3
                          // po opuszczeniu pliku include orig.v
```

16.11. Dyrektywy `_FILE_ i `_LINE_

Dyrektywa `**__FILE__** interpretuje nazwę bieżącego pliku wejściowego jako literał łańcuchowy. Jest to ścieżka, z której narzędzie otworzyło plik, a nie krótka nazwa podana w parametrze dyrektywy `**include** lub jako argument do nazwy pliku wejściowego narzędzia. Format tej ścieżki może zależeć od implementacji.

Dyrektywa `**__LINE__** odwołuje się do bieżącego numeru łańcucha wejściowego jako prostej liczby dziesiętnej.

Dyrektywy `**__FILE__** i `**__LINE__** są przydatne przy tworzeniu komunikatu o błędzie, aby zgłosić problem. W komunikacie można podać linię źródłową, w której wykryto problem. Na przykład:

```
$display("Internal error: null handle at %s, line %d.",
`__FILE__, `__LINE__);
```

Dyrektywa `**include** zmienia rozszerzenia `**__FILE__** i `**__LINE__**, aby dopasować plik dołączany. Na końcu tego pliku, gdy wznawiane jest przetwarzanie pliku wejściowego zawierającego dyrektywę `**include**, rozszerzenia `**__FILE__** i `**__LINE__** powracają do wartości, jakie miały przed dyrektywą `**include** (ale `**__LINE__** zwiększa się o jeden, gdy przetwarzanie przechodzi do linii po dyrektywie `**include**).

Dyrektywa `**line** zmienia dyrektywę `**__LINE__** i może również zmienić dyrektywę `**__FILE__**.

16.12. Dyrektywy `begin_keywords i `end_keywords

Dyrektywy `**begin_keywords** i `**end_keywords** definiują listę słów kluczowych dla wersji języka SystemVerilog.

Składnia dyrektyw **`begin_keywords** i **`end_keywords**:

`begin_keywords "*version*"

`end_keywords

gdzie *version* może przyjmować następujące wartości:

1800-2017
1800-2012
1800-2009
1800-2005
1364-2005
1364-2001
1364-2001-noconfig
1364-1995

Parametr *version* określa, że tylko identyfikatory wymienione jako zarezerwowane słowa kluczowe w określonej wersji są uważane za słowa zastrzeżone. Dyrektywy **`begin_keywords** i **`end_keywords** definiują jedynie zestaw identyfikatorów, które są zarezerwowane jako słowa kluczowe.

Para dyrektyw **`begin_keywords** ...**`end_keywords** może być zagnieżdżona.

Dyrektywa **`begin_keywords** wpływa na cały kod źródłowy, który po niej następuje, nawet przez granice plików z kodem źródłowym, aż do pojawienia się odpowiedniej dyrektywy **`end_keywords** lub końca jednostki kompilacji.

Jeśli dyrektywa **`begin_keywords** nie jest określona, lista zarezerwowanych słów kluczowych powinna być domyślnym zestawem słów kluczowych tego narzędzia projektowego.

```
`begin_keywords „1364-2001”        // używane są słowa kluczowe
                                   // Verilog IEEE Std 1364-2001

module m2 (...);
      reg [63:0] logic;            // OK: „logic” nie jest słowem kluczowym w 1364-2001
      ...
endmodule

`end_keywords
```

W tym przykładzie zmienna *logic* nie powoduje błędu kompilacji, ponieważ nie jest słowem kluczowym Verilog IEEE Std 1364-2001.

Poniższy przykład reprezentuje kod z poprzedniego przykładu, ale wyraźnie stwierdza, że należy użyć słów kluczowych SystemVerilog IEEE Std 1800-2005. Ten przykład spowoduje błąd, ponieważ **logic** jest zarezerwowana jako słowo kluczowe w tym standardzie.

```
`begin_keywords „1800-2005”    // użycie słów kluczowych
                               // IEEE Std 1800-2005 SystemVerilog

module m2 (...);
      reg [63:0] logic;        // ERROR: „logic” jest słowem kluczowym w Std 1800-2005
      ...
endmodule
`end_keywords
```

16.13. Wnioski

Dyrektywy kompilatora są poleceniami sterującymi kompilatorem. Instruują one kompilator, jak interpretować kod źródłowy projektu napisanego w języku SystemVerilog.

Niektóre dyrektywy w SystemVerilog są takie same jak w większości języków programowania; inne są specyficzne tylko dla tego języka.

Dyrektywy języka SystemVerilog pozwalają na: przywrócenie domyślnych wartości dyrektyw kompilacji; zdefiniowanie wartości jednostki czasu; pracę z makrodefinicjami; wykonanie warunkowej kompilacji projektu; zdefiniowanie domyślnego typu sieci; zdefiniowanie wartości logicznych dla niepołączonych wejść; wyznaczenie modułów jako komórkowych; pracę z pragmami; dołączenie plików z kodem źródłowym; zdefiniowanie numerów linii kodu źródłowego; zdefiniowanie nazwy pliku i numeru linii w pliku, w którym wystąpił błąd; zdefiniowanie listy słów kluczowych dla wersji języka SystemVerilog.

Dyrektywa **`resetall** zwraca wartość domyślną dla tych dyrektyw kompilatora, które mają wartość domyślną.

Wartość (czas trwania) pojedynczej jednostki czasu jest określana za pomocą dyrektywy **`timescale**. Nie ma ona wartości domyślnej, więc musi być zawsze zdefiniowana w projektach, gdzie używane są jednostki czasu.

Makrodefinicje SystemVerilog lub *makra* (*macros*) w kodzie źródłowym są używane do zastąpienia jednego łańcucha tekstowego innym i są definiowane za pomocą dyrektywy **`define**. Makra mogą mieć argumenty z wartościami domyślnymi. Można użyć definicji makr do zdefiniowania stałych, prostych funkcji itp. Do opisywania definicji makr mogą być używane komentarze.

Dyrektywa **`undef** służy do cofnięcia wcześniej zdefiniowanej nazwy definicji makra. Dyrektywa **`undefineall** nadpisuje wszystkie makra tekstowe zdefiniowane wcześniej przez dyrektywy **`define** w jednostce kompilacji.

Dyrektywy kompilacji warunkowej (**`ifdef**, **`ifndef**, **`else**, **`elsif** i **`endif**) pozwalają na włączenie poszczególnych fragmentów kodu do kodu źródłowego, w zależności od tego czy nazwa definicji makra została wcześniej zdefiniowana przy użyciu dyrektywy **`define**.

W języku SystemVerilog sieci domyślnie są typu **wire**. Jednakże jeśli cały system działa w logice bipolarnej lub ECL, może być wygodniej zmienić domyślny typ układu, używając dyrektywy **`default_nettype**.

W niektórych przypadkach konieczne jest określenie wartości wejść niepodłączonych modułów w celu uzyskania dokładniejszych wyników symulacji. Służą do tego dyrektywy **`unconnected_drive** i **`nounconnected_drive**.

Dyrektywy **`celldefine** i **`endcelldefine** oznaczają moduły jako komórki. Komórki są wykorzystywane podczas obliczania opóźnień przez niektóre podprogramy interfejsu języków programowania PLI.

Pragma jest ogólnym terminem używanym do określenia konstrukcji bez predefiniowanej semantyki językowej, która wpływa na to, jak syntezator powinien tłumaczyć kod na schemat.

Dyrektywa **`pragma** pozwala na wprowadzenie specyfikacji strukturalnej, definiuje nazwę pragmy i listę wyrażeń dla niej.

Dyrektywa **`include** może być użyta do włączenia dowolnych plików tekstowych do kodu źródłowego SystemVerilog. Nazwa pliku *filename* może być pełną albo względną nazwą ścieżki. Może być ujęta w cudzysłowy lub nawiasy kątowe, co wpływa na sposób, w jaki kompilator wyszukuje dołączany plik.

Plik dołączony z dyrektywą **`include** może zawierać inne dyrektywy **`include**. Język SystemVerilog pozwala na 15 poziomów ich zagnieżdżania.

Dyrektywa **`line** jest przeznaczona do określenia numeru linii kodu źródłowego i nazwy pliku źródłowego. Pozwala na zachowanie lokalizacji elementów składni w pliku źródłowym, jeśli inny proces zmieni kod źródłowy. W rezultacie kompilator może poprawnie odnieść się do lokalizacji błędu w kodzie źródłowym. Dyrektyw **`line** nie używa się do ręcznego wstawiania do kodu, narzędzia projektowe wstawiają je automatycznie.

Dyrektywa **`__FILE__** interpretuje nazwę bieżącego pliku wejściowego jako literał łańcuchowy. Jest to ścieżka, której narzędzie użyło do otwarcia pliku. Dyrektywa **`__LINE__** odwołuje się do bieżącego numeru łańcucha wejściowego jako prostej liczby dziesiętnej. Dyrektywy **`__FILE__** i **`__LINE__** są przydatne przy tworzeniu komunikatu o błędzie. Dyrektywa **`line** zmienia dyrektywę **`__LINE__**, może też zmienić dyrektywę **`__FILE__**.

Dyrektywy **`begin_keywords** i **`end_keywords** definiują listę słów kluczowych dla wersji języka SystemVerilog.

17. Prymitywy

Kiedy użytkownik rozpoczyna naukę języka programowania, zawsze pojawia się problem: od czego zacząć? W językach programowania zazwyczaj pierwszym programem jest wypisanie ciągu znaków „Hello World!". Aby wydrukować lub wyświetlić ciąg znaków, język programowania musi mieć pewne standardowe polecenia bądź wbudowane funkcje. Nie jest tak w przypadku *języków opisu sprzętu* (*hardware description language* – HDL), takich jak SystemVerilog, ponieważ nie są one zaprojektowane do drukowania łańcuchów znaków, ale do opisywania sprzętu.

Sprzęt nie jest jednak zbudowany z niczego; składa się z najprostszych elementów, które nazywane są *prymitywami* (*primitives*). Prymitywy są zazwyczaj zawarte w języku HDL. Język SystemVerilog nie jest pod tym względem wyjątkiem, zawiera również pewien dość ubogi zestaw prymitywów, które są zapożyczone z Veriloga.

Ograniczona liczba prymitywów języka SystemVerilog nie oznacza, że nie można znaleźć gotowego rozwiązania dla standardowych jednostek funkcjonalnych czy prostych projektów. Istnieje wiele miejsc, w których przechowywane są gotowe, bezpłatne projekty Verilog i SystemVerilog. (Niegdyś jeden z autorów był słuchaczem referatu na konferencji naukowej, w którym poruszono kwestię opracowania projektu procesora przemysłowego, którego wszystkie komponenty były pozyskiwane od innych twórców za darmo).

Może pojawić się pytanie: czy w SystemVerilog można tworzyć własne prymitywy, jeśli możliwości prymitywów pierwotnych wbudowanych w język są niewystarczające? Odpowiedź na to pytanie brzmi: tak. SystemVerilog udostępnia *mechanizm UDP* (*user defined primitive*), dzięki któremu użytkownik może tworzyć własne prymitywy. Nazywane są one *prymitywami zdefiniowanymi przez użytkownika*, a ich użycie nie różni się od prymitywów SystemVerilog.

UDP pozwala na tworzenie prymitywów kombinacyjnych i sekwencyjnych. Te ostatnie mogą być wrażliwe na poziom sygnału sterującego (zatrzaski), jak również na zbocze sygnału (przerzutniki). Ponadto w SystemVerilog możliwe jest tworzenie prymitywów sekwencyjnych, które są wrażliwe zarówno na poziom, jak i zbocze sygnałów sterujących.

17.1. Gdzie szukać gotowych rozwiązań

Przed przystąpieniem do opracowania urządzenia lub systemu cyfrowego dobrze jest zapoznać się m.in. z: prymitywami języka SystemVerilog, prymitywami stosowanego kompilatora, które są zwykle znacznie szersze niż prymitywy języka SystemVerilog, bibliotekami standardowych węzłów funkcjonalnych dostarczanych przez zintegrowane środowisko projektowe oraz bibliotekami projektów *open-source* i zakupionymi bibliotekami projektów.

Mogą się pojawić takie sytuacje, gdy po wielu wysiłkach blok, który opracowujemy, okazuje się publicznie dostępny. Jest też znacznie lepszy od opracowanego przez nas projektu pod względem kosztów implementacji, szybkości czy poboru mocy.

Wszystkie publicznie dostępne komponenty projektu SystemVerilog można podzielić na następujące grupy:

- prymitywy języka SystemVerilog;
- prymitywy kompilatora;
- biblioteki prymitywów wykorzystywanego środowiska zintegrowanego (w naszym przypadku pakietu Quartus Prime);
- biblioteki *bloków IP* (*intellectual property*) używanego środowiska zintegrowanego (w naszym przypadku IP Catalog pakietu Quartus Prime);
- publiczne biblioteki bloków IP;
- biblioteki bloków IP innych producentów.

Podstawowe prymitywy języka SystemVerilog są dość nieliczne, tzn. standard języka Verilog prezentuje bardzo ograniczony zestaw prymitywów. Dobrą rzeczą w stosowaniu prymitywów SystemVerilog jest to, że mogą być używane w dowolnym miejscu projektu bez wcześniejszej definicji lub deklaracji. Nazwy prymitywów są zawsze widoczne w całym drzewie hierarchii projektu.

Korzystanie z bibliotek prymitywów środowiska zintegrowanego jest podobne do korzystania z podstawowych prymitywów SystemVerilog: nie wymaga żadnego wysiłku ze strony użytkownika.

Zazwyczaj oprócz biblioteki prymitywów oprogramowanie do projektowania zintegrowanego, takie jak Quartus, proponuje duży zestaw gotowych rozwiązań projektowych standardowych bloków funkcjonalnych, które są zaprojektowane przez wysoko wykwalifikowanych specjalistów i dokładnie przetestowane. W dokumentacji technicznej można znaleźć zalecenia, aby nie próbować opracowywać takich bloków samodzielnie, ponieważ nie osiągniemy lepszych rezultatów. Autorzy niniejszej książki mają wątpliwości w tej kwestii i np. opracowali metodologię projektowania komparatorów [9], która pozwala budować komparatory o dużych rozmiarach (ponad tysiąc bitów na jedno słowo wejściowe), a także znacznie lepsze pod względem kosztów i szybkości działania od rozwiązań bibliotecznych.

17.2. Prymitywy języka SystemVerilog

Wbudowane prymitywy SystemVerilog można podzielić na dwie grupy: prymitywy bramek i prymitywy przełączników. Do tej pierwszej grupy oprócz samych bramek należą również bufory.

Grupa prymitywów przełączników reprezentuje tranzystory różnych typów. Może się wydawać dziwne, że prymitywami w wysokopoziomowym języku projektowania, jakim jest SystemVerilog, są tranzystory. Nie należy jednak zapominać, że Verilog (jego poprzednik) został pierwotnie stworzony jako język symulacji i powinien być w stanie modelować elementy systemów cyfrowych na wszystkich poziomach, od tranzystorowego do strukturalnego.

Standard języka SystemVerilog definiuje 14 prymitywów bramek i 12 prymitywów przełączników. Główną zaletą prymitywów jest to, że mogą być używane w kodzie projektu bez wcześniejszej deklaracji. Ogólna składnia do tworzenia instancji prymitywów jest następująca:

gate_type (*strength*) #(*delay*) *instance_name* [*range*] {*terminals*}

switch_type #(*delay*) *instance_name* [*range*] {*terminals*}

gdzie: *gate_type* to typ prymitywu bramki; *strength* siła źródła (moc logiczna); *delay* wartość opóźnienia; *instance_name* nazwa instancji; *range* zakres; *terminals* lista terminali; *switch_type* typ przełącznika.

W podanych formatach konstrukcje *strength*, *delay*, *instance_name* i *range* są opcjonalne. Siła (*strength*) sterownika określa moc logiczną sygnału, który jest generowany przez bramkę. Zakres (*range*) pozwala na deklarowanie tablic prymitywów. Terminale (*terminals*) odpowiadają portom modułu. Do terminali prymitywów podłącza się sygnały zewnętrzne (sieci). Prymitywy przełączników nie mają określonej mocy logicznej.

17.2.1. Typy prymitywów

Typy prymitywów *gate_type* i *switch_type* są zdefiniowane przez słowa kluczowe odpowiednich prymitywów. W powyższych formatach zamiast *gate_type* lub *switch_type* wpisuje się słowo kluczowe, które definiuje prymityw bramki bądź przełącznika.

Prymitywy bramek można podzielić na następujące grupy:

- bramki o n wejściach i jednym wyjściu: **and**, **nand**, **or**, **nor**, **xor** i **xnor**;
- bramki z jednym wejściem i n wyjściami: **buf** i **not**;
- bufory o trzech stanach: **bufif1**, **bufif0**, **notif1** i **notif0**;
- elementy podciągające: **pulldown** i **pullup**.

Z kolei prymitywy przełączające dzielą się na:

- przełączniki MOS (*metal oxide semiconductor* – MOS): **cmos**, **nmos**, **pmos**, **rcmos**, **rnmos** i **rpmos**;
- przełączniki dwukierunkowe: **rtran**, **rtranif0**, **rtranif1**, **tran**, **tranif0** i **tranif1**.

17.2.2. Siła sterownika

Siła sterownika określa moc logiczną wartości wyjściowych bramki. Konstrukcja *strength* określająca moc źródła może mieć następujące formaty:

```
(strength1, strength0)      // przy przełączaniu wyjścia z 1 na 0
(strength0, strength1)      // przy przełączaniu wyjścia z 0 na 1
```

gdzie *strength0* i *strength1* mogą być dowolnymi słowami kluczowymi z tabeli 3.1.

Moc logiczna źródła może być zdefiniowana tylko dla prymitywów bramek. Prymitywy przełączające przenoszą poziom mocy sygnału wejściowego na wyjście.

Prymitywy bramek mogą mieć następujące poziomy mocy wyjściowej: **supply0**, **supply1**, **strong0**, **strong1**, **pull0**, **pull1**, **weak0**, **weak1**, **highz0**, **highz1**. Specyfikacja *strength1* może być jednym z następujących słów kluczowych: **supply1**, **strong1**, **pull1**, **weak1** i **highz1**. Podobnie, specyfikacja *strength0* może być jednym z następujących słów kluczowych: **supply0**, **strong0**, **pull0**, **weak0** i **highz0**.

Jeśli **highz1** jest określone jako *strength1*, bramka generuje wartość logiczną z zamiast 1. Jeżeli *strength0* jest **highz0**, to bramka tworzy wartość z zamiast 0. W poniższym przykładzie bramka **nor** utworzy wartość z zamiast 1.

```
nor (highz1, strong0) e1(out, in1, in2);
```

Domyślne wartości siły to **strong0** i **strong1**. Przykłady sposobów określania siły źródła sygnału:

(**supply1**, **pull0**) – określa moc źródła sygnału przy przełączaniu z 1 na 0 w następujący sposób: **supply1** dla wartości pojedynczej i **pull0** dla wartości zerowej;

(**supply0**, **pull1**) – określa moc źródła sygnału przy przełączaniu z 0 na 1 w następujący sposób: **supply0** dla wartości zerowej i **pull1** dla wartości pojedynczej.

17.2.3. Opóźnienia

Konstrukcja *delay* reprezentuje opóźnienie czasowe sygnału przechodzącego przez prymityw. Ta sama konstrukcja definiuje opóźnienie sygnałów w sieciach (patrz podrozdział 3.2.2). Domyślnie opóźnienie wynosi 0. Do reprezentowania opóźnień

można używać liczb całkowitych i rzeczywistych, które określają liczbę jednostek czasu opóźnienia (rozmiar jednej jednostki czasu określa dyrektywa `**timescale**). Siła sygnału wejściowego nie ma wpływu na opóźnienie. Na przykład:

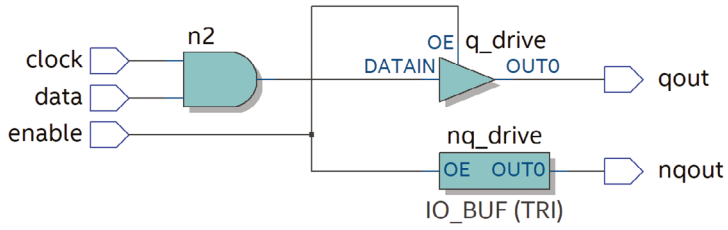
```
and #(9) a1 (out, in1, in2);          // tylko jedna wartość opóźnienia 9
and #(9,12) a2 (out, in1, in2);       // opóźnienie narastania 9 i opadania 12
bufif0 #(9,12,11) b3 (out, in, ctrl); // opóźnienie narastania 9, opóźnienie opadania 12
                                      // i wyłączenia 11
```

Poniższy przykład przedstawia prosty moduł tri_latch z wyjściami o trzech stanach, gdzie dla każdego elementu logicznego określono osobne opóźnienia. Dlatego opóźnienie propagacji od wejść do wyjść modułu zależy od ścieżki sygnału przez moduł tri_latch.

Przykład 17.1. Zatrzask z wyjściami trójstanowymi zbudowany z prymitywów

```
module tri_latch (qout, nqout, clock, data, enable);
    output qout, nqout;
    input clock, data, enable;
    tri qout, nqout;

    not    #5          n1 (ndata, data);
    nand   #(3,5)      n2 (wa, data, clock),
                       n3 (nwdba,ta, clock);
    nand   #(12,15)    n4 (q, nq, wa),
                       n5 (nq, q, wb);
    bufif1 #(3,7,13)   q_drive (qout, q, enable),
                       nq_drive (nqout, nq, enable);
endmodule
```

Implementacja projektu tri_latch została przedstawiona na rys. 17.1.



RYS. 17.1. Wyniki syntezy projektu tri_latch

ŹRÓDŁO: oprac. własne.

Poniżej przedstawiono przykłady bardziej szczegółowych definicji opóźnień, gdzie opóźnienia w formacie # (*del01*, *del10*, *del2*) są określone dla każdej wartości formatu *min : typ : max*.

```
bufif0 #(5:7:9, 8:10:12, 15:18:21)     b1 (io1, io2, dir);
bufif1 #(6:8:10, 5:7:9, 13:17:19)      b2 (io2, io1, dir);
```

17.2.4. Tablice prymitywów

Konstrukcja zakresu *range* pozwala na zdefiniowanie tablicy instancji prymitywów, z których każda jest związana z określonym bitem wektora. Zakres ma następujący format:

[*left_index* : *right_index*]

gdzie *left_index* i *right_index* są lewym i prawym indeksem zakresu.

Każdej instancji tablicy prymitywów przypisany jest indeks instancji. Jeśli tablica prymitywów jest przypisana do wektora, prymitywy łączą się sekwencyjnie z wektorem, zaczynając od prawego indeksu do lewego indeksu, oraz z bitami wektora, zaczynając od prawego bitu do lewego bitu. Szerokość wektora bitów musi być równa liczbie elementów tablicy instancji prymitywów.

Jeśli zamiast wektora użyty jest skalar (pojedynczy sygnał), jest on podłączony do wszystkich instancji tablicy.

Przykłady tworzenia instancji prymitywów:

```
and i1 (out, in1, in2);          // 2-wejściowa bramka AND z zerowym opóźnieniem

and # 5 (O, i1, i2, i3, i4);     /* 4-wejściowa bramka AND z opóźnieniem 5 jednostek
                                    czasu przy przejściach na wszystkich wyjściach */

not # (2,3)  u7 (out, in);       /* inwerter, dla którego zdefiniowane są dwa różne opóźnienia:
                                    przy przełączaniu z 0 na 1 – 2 jednostki czasu, a z 1 na 0 – 3 */

buf (pull0,strong1) (y, a);      /* bufor z różnymi mocami sygnałów dla
                                    wartości zerowej (pull0) i jedynkowej (strong1) */

wire [31:0] y, a;                // definicja wektorów 32-bitowych
buf # 2.7 i [31:0] (y, a);       /* tworzenie tablicy 32 prymitywów buforowych i0, i1,...,i31,
                                    podłączonych do szyn y i a; opóźnienie dla każdego elementu
                                    to 2.7 jednostki czasu */
```

17.2.5. Lista terminali

Lista terminali określa, jak prymityw łączy się z resztą projektu. Lista terminali jest ujęta w nawiasy, a terminale są oddzielone przecinkami (przypomnijmy, że terminale prymitywów są takie same jak porty modułów). Terminale wyjściowe i dwukierunkowe są zawsze pierwsze na liście terminali.

Poniższy przykład demonstruje użycie tablicy prymitywów **bufif0** w tworzeniu N-bitowego bufora trójstanowego.

Przykład 17.2. Bufor trójstanowy zbudowany z prymitywów **bufif0**

```
module bufif0n #(parameter N=4)
        (input [N-1:0] in,
        input          en,
        output [N-1:0] out);

        bufif0 ar[N-1:0] (out, in, en);          // tablica buforów z trzema stanami

endmodule
```

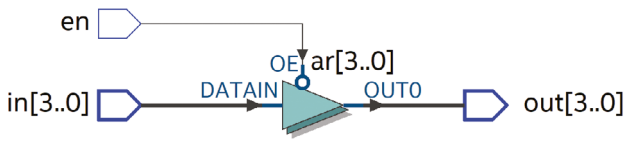
Tutaj linia

```
bufif0 ar[N-1:0] (out, in, en);
```

jest równoważna z następującym opisem:

```
bufif0 ar3 (out[3], in[3], en);
bufif0 ar2 (out[2], in[2], en);
bufif0 ar1 (out[1], in[1], en);
bufif0 ar0 (out[0], in[0], en);
```

Implementacja projektu bufif0n została przedstawiona na rys. 17.2.



RYS. 17.2. Wyniki syntezy projektu bufif0n

ŹRÓDŁO: oprac. własne.

17.2.6. Bramki logiczne and, nand, or, nor, xor i xnor

Prymitywy **and**, **nand**, **or**, **nor**, **xor** i **xnor** odnoszą się do bramek logicznych. Bramki logiczne mają jedno wyjście i mogą mieć jedno lub więcej wejść. Pierwszy terminal na liście jest wyjściem, a pozostałe terminale są wejściami. Na przykład:

```
and g1(out, in1, in2);          // bramka and z wejściami in1 i in2 oraz wyjściem out
```

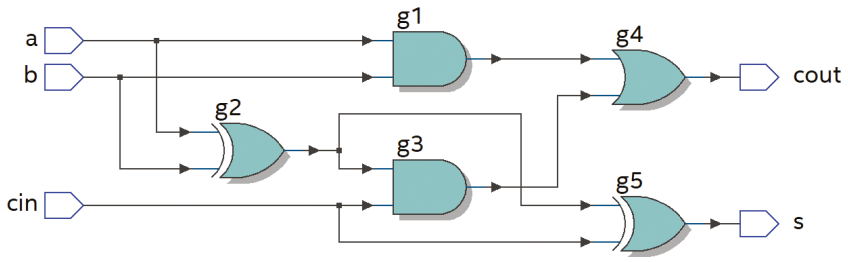
Bramki logiczne realizują odpowiednie funkcje logiczne. Jeśli jedno z wejść ma wartość x lub z, to wyjściem będzie x.

Poniższy przykład pokazuje zastosowanie prymitywów bramek logicznych w projekcie sumatora jednobitowego.

Przykład 17.3. Sumator jednobitowy zbudowany z bramek logicznych

```
module add_1 (input a, b, cin,          // opis portów wejściowych
              output s, cout);          // opis portów wyjściowych

    wire g1_o, g2_o, g3_o;              // zmienne pośrednie

    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
```

Realizacja projektu add_1 została przedstawiona na rys. 17.3.



RYS. 17.3. Wyniki syntezy projektu add_1

ŹRÓDŁO: oprac. własne.

17.2.7. Bufory buf i not

Prymitywy **buf** i **not** są buforami. Mają jedno wejście i mogą mieć jedno lub więcej wyjść (inaczej niż bramki logiczne). Ostatni terminal na liście terminali jest podłączony do wejścia, pozostałe terminale są podłączone do wyjść. Na przykład:

```
buf b1(out1, out2, in); // bufor buf z wejściem in i wyjściami out1 i out2
```

Bufor **buf** powtarza wartość wejściową, a bufor **not** odwraca wartość wejściową, z wyjątkiem wartości z. W przypadku wartości z na wejściu na wyjściu buforów **buf** i **not** będzie wartość x.

17.2.8. Bufory bufif1, bufif0, notif1 i notif0

Prymitywy **bufif1**, **bufif0**, **notif1** i **notif0** definiują bufory o trzech stanach:
- **bufif1** – zwykły bufor o trzech stanach;
- **bufif0** – bufor o trzech stanach z inwersją sygnału sterującego;
- **notif1** – bufor o trzech stanach z inwersją wyjścia;
- **notif0** – bufor o trzech stanach z inwersją wyjścia i inwersją sygnału sterującego.

Prymityw **bufif1** przy wartości 1 na wejściu sterującym, a prymityw **bufif0** przy wartości 0, przekazują sygnał z wejścia danych na wyjście bez zmian, przy czym wartości x i z na wejściu danych odpowiadają wartości nieznanej x na wyjściu. Prymitywy **notif1** i **notif0** działają podobnie, ale wartość na wyjściu jest odwrócona. Wartość 0 na wejściu sterującym dla prymitywów **bufif1** i **notif1** oraz wartość 1 dla prymitywów **bufif0** i **notif0** ustawia wyjście w stan wysokiej impedancji.

Wartość x lub z na wejściu sterującym dla prymitywów **bufif0** i **bufif1**, przy wartości logicznej 0 na wejściu danych, ustawia wyjście w stan L (sygnał niski), a wartość 1 ustawia je w stan H (sygnał wysoki). Podobnie wartość x lub z na wejściu sterującym dla prymitywów **notif0** i **notif1** przy wartości logicznej 0 na wejściu danych ustawia wyjście w stan H, a przy wartości 1 ustawia wyjście w stan L. Na przykład:

```
bufif1 bf1(out, in, control);    // bufor bufif1 z wejściem, wyjściem
                                 // i sygnałem sterującym control
```

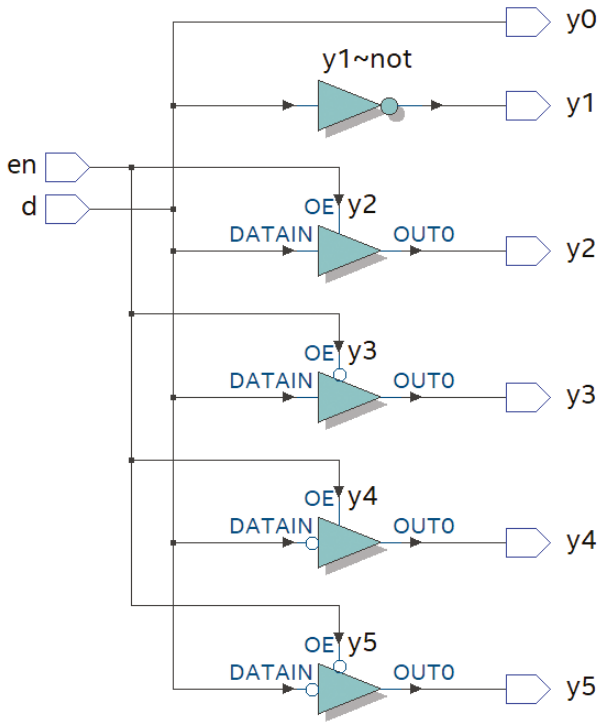
Poniższy przykład pokazuje różne prymitywy buforów.

Przykład 17.4. Prymitywy buforowe

```
module buffers
        (input d, en,
        output y0, y1, y2, y3, y4, y5);
```

```
        buf(y0,d);
        not(y1,d);

        bufif1(y2,d,en);
        bufif0(y3,d,en);
        notif1(y4,d,en);
        notif0(y5,d,en);
endmodule
```

Realizacja projektu buforów została przedstawiona na rys. 17.4.



RYS. 17.4. Wyniki syntezy projektu buforów

ŹRÓDŁO: oprac. własne.

17.2.9. Przełączniki MOS

Prymitywy przełączające opisują zachowanie tranzystorów. Chociaż SystemVerilog koncentruje się na opisywaniu projektów na poziomie systemu, obsługuje wszystkie poziomy projektowania, w tym poziom tranzystorów.

Przełączniki MOS (*MOS switches*) obejmują następujące prymitywy: **nmos**, **pmos**, **rnmos**, **rpmos**, **cmos** i **rcmos**.

Prymityw **pmos** odnosi się do tranzystora typu P, a prymityw **nmos** do tranzystora typu N. Gdy tranzystory PMOS i NMOS przewodzą, mają stosunkowo niską impedancję między źródłem a drenem.

Prymityw **rpmos** odnosi się do *rezystancyjnego tranzystora PMOS* (*resistive PMOS transistor*), a prymityw **rnmos** do *rezystancyjnego tranzystora NMOS* (*resistive NMOS transistor*). Gdy oba te tranzystory są w stanie przewodzenia, mają znacznie większą impedancję między źródłem a drenem niż konwencjonalne tranzystory PMOS i NMOS. Dlatego prymitywy **rpmos** i **rnmos** są urządzeniami obciążającymi w sieciach MOS.

Jeśli chodzi o funkcjonalności, prymitywy **nmos**, **pmos**, **rnmos** i **rpmos** są podobne do buforów **bufif** i są jednokierunkowymi kanałami danych. Tablica prawdy dla prymitywów **pmos** i **rpmos** jest podana w tabeli 17.1, a dla prymitywów **nmos** i **rnmos** – w tabeli 17.2.

TABELA 17.1. Tabela prawdy dla prymitywów **pmos** i **rpmos**

pmos rpmos	Control				
	0	**1**	**x**	**z**
D	0	0	z	L	L
a	1	1	z	H	H
t	x	x	z	x	x
a	z	z	z	z	z

ŹRÓDŁO: oprac. własne.

TABELA 17.2. Tabela prawdy dla prymitywów **nmos** i **rnmos**

pmos rpmos	Control				
	0	**1**	**x**	**z**
D	0	z	0	L	L
a	1	z	1	H	H
t	x	z	x	x	x
a	z	z	z	z	z

ŹRÓDŁO: oprac. własne.

Prymitywy **pmos** i **rpmos** przy wartości 0 na wejściu sterującym oraz prymitywy **nmos** i **rnmos** przy wartości 1 przekazują dane wejściowe na wyjście w niezmienionej postaci. Prymitywy **pmos** i **rpmos** przy wartości 1 na wejściu sterującym, a prymitywy **nmos** i **rnmos** przy wartości 0 wprowadzają wyjście w stan wysokiej impedancji. Tabele 17.1 i 17.2 zawierają również wartości L i H. Symbol L reprezentuje wynik,

który ma wartość 0 lub z, a symbol H reprezentuje wynik, który ma wartość 1 bądź z. Opóźnienia przejść w stany H i L mają taką samą wartość jak dla przejść w stan x.

Dla prymitywów **nmos**, **pmos**, **rnmos** i **rpmos** pierwszy terminal jest wyjściem, drugi terminal jest wejściem, a trzeci terminal jest wejściem sterującym.

Zauważmy, że przełączniki **rnmos** i **rpmos** zmniejszają natężenie prądu przechodzących przez nie sygnałów.

Prymitywy **cmos** i **rcmos** mają cztery terminale. Pierwszy terminal odpowiada wyjściu, drugi wejściu danych, trzeci wejściu sterującemu kanałem typu n, a czwarty wejściu sterującemu kanałem typu p. Prymityw **cmos** jest połączeniem przełączników **pmos** i **nmos**, a prymityw **rcmos** jest połączeniem przełączników **rpmos** i **rnmos**. Na przykład:

```
cmos (out, in, ncontrol, pcontrol);
```

jest równoważne:

```
nmos (out, in, ncontrol);
pmos (out, in, pcontrol);
```

Należy pamiętać, że przełączniki MOS nie są obsługiwane w systemie Quartus.

17.2.10. Przełączniki dwukierunkowe

Przełączniki dwukierunkowe (*bidirectional switches*) zawierają następujące prymitywy: **tran**, **rtran**, **tranif1**, **rtranif1**, **tranif0** i **rtranif0**. Nie opóźniają one przechodzących przez nie sygnałów. Przełączniki **tranif1**, **rtranif1**, **tranif0** i **rtranif0** przepuszczają sygnały, gdy są włączone, a blokują je, gdy są wyłączone. Prymitywy **tran** i **rtran** zawsze pozwalają na przepuszczanie sygnałów, nie można ich wyłączyć.

Specyfikacja opóźnień dla przełączników **transif1**, **transif0**, **rtranif1** i **rtranif0** musi zawierać zero, jedno lub dwa opóźnienia. Jeśli specyfikacja zawiera dwa opóźnienia, to pierwsze określa opóźnienie zezwolenia, drugie – opóźnienie wyłączenia, a niższe z tych dwóch opóźnień ma zastosowanie do przejść wyjścia na wartości x i z. Jeśli określono tylko jedno opóźnienie, to określa ono zarówno opóźnienie włączenia, jak i wyłączenia. Jeśli nie jest dostępna specyfikacja opóźnienia, opóźnienie włączenia lub wyłączenia przełącznika dwukierunkowego nie ma zastosowania. Przełączniki **tran** i **rtran** nie mają możliwości specyfikacji opóźnienia.

Prymitywy **tranif1**, **rtranif1**, **tranif0** i **rtranif0** mają trzy terminale. Pierwsze dwa są wyjściami dwukierunkowymi, a trzeci terminal jest wejściem sterującym. Prymitywy **tran** i **rtran** mają dwa terminale, które są wyjściami dwukierunkowymi. Mogą się one łączyć tylko z sieciami skalarnymi lub wyborem bitowym sieci wektorowych. Na przykład:

```
tranif1 t1 (inout1,inout2,control);
```

Tutaj inout1 i inout2 są wyjściami dwukierunkowymi, control jest wejściem sterującym.

Należy pamiętać, że przełączniki dwukierunkowe nie są obsługiwane w systemie Quartus.

17.2.11. Źródła pullup i pulldown

Źródła obejmują prymitywy **pullup** i **pulldown**. Prymityw **pullup** ustawia wartość logiczną na 1, a prymityw **pulldown** ustawia wartość logiczną na 0, na obwodach podłączonych do jego terminala.

W przypadku braku specyfikacji siły sygnały podłączone do terminala mają siłę podciągania **pull**. Specyfikacja **strength1** definiuje wartość siły dla prymitywu **pullup**, a specyfikacja **strength0** definiuje wartość siły dla prymitywu **pulldown**. Specyfikacja **strength0** dla prymitywu **pullup** i specyfikacja **strength1** dla prymitywu **pulldown** są ignorowane. Prymitywy **pullup** i **pulldown** nie mają specyfikacji opóźnienia. Na przykład:

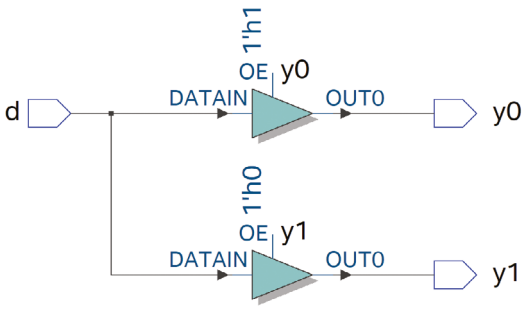
```
pullup (strong1) p1 (neta), p2 (netb);
```

Powyżej instancja p1 steruje siecią neta z siłą **strong1**, a instancja p2 steruje siecią netb. Poniższy przykład pokazuje użycie prymitywów **pullup** i **pulldown**.

Przykład 17.5. Użycie prymitywów **pullup** i **pulldown**

```
module pull_up_down
        (input d,
        output y0,y1);

        wire e0,e1;

        pullup(e1);
        pulldown(e0);

        bufif0(y0,d,e0);
        bufif1(y1,d,e1);
endmodule
```

Implementacja projektu pull_up_down została przedstawiona na rys. 17.5.



RYS. 17.5. Wyniki syntezy projektu pull_up_down

ŹRÓDŁO: oprac. własne.

17.3. Prymitywy zdefiniowane przez użytkownika

17.3.1. Deklarowanie prymitywów zdefiniowanych przez użytkownika

Jeśli zestaw predefiniowanych prymitywów w języku SystemVerilog nie jest wystarczający, użytkownik może tworzyć własne prymitywy zwane *prymitywami zdefiniowanymi przez użytkownika* (*user defined primitives* – UDP) lub po prostu *prymitywami użytkownika*.

Zgodnie z *mechanizmem UDP* format definiowania prymitywu jest następujący (styl ANSI):

```
primitive primitive_name
        (output reg = logic_value  terminal_declaration,
        input terminal_declarations);

        table
                table_entry;
                ...
                table_entry;
        endtable
endprimitive
```

gdzie: **primitive**, **output**, **reg**, **input**, **table**, **endtable**, **endprimitive** są słowami kluczowymi; *primitive_name* – nazwa prymitywu; *logic_value* – wartość logiczna; *terminal_declaration* – definicja terminalu wyjściowego (portu); *terminal_declarations* – definicje terminalu wejściowego; *table_entry* – wiersz tablicy prawdy.

Stary styl definicji prymitywów jest następujący:

```
primitive primitive_name (output, input, input,… );
        output terminal_declaration;
        input terminal_declarations;
        reg output_terminal;
        initial output_terminal = logic_value;

        table
                table_entry;
                …
                table_entry;
        endtable
endprimitive
```

gdzie *output_terminal* jest terminalem wyjściowym.

Wszystkie terminale (porty) w definicji prymitywu muszą mieć szerokość jednego bitu. Prymityw może mieć tylko jedno wyjście, które musi występować jako pierwsze na liście terminali. Maksymalna liczba wejść wynosi 10 dla prymitywów kombinacyjnych i 9 dla prymitywów sekwencyjnych. Funkcjonalność prymitywu jest określona za pomocą *tablicy prawdy* (table_entry). Do określenia w niej wartości można użyć wartości logicznych 0, 1, z i x, przy czym wartość z jest interpretowana jako x.

Konstrukcja **reg** = *logic_value* jest opcjonalna, służy do określenia stanu początkowego (po włączeniu zasilania) wyjścia prymitywów sekwencyjnych.

Zauważmy, że składnia definiująca wyjścia prymitywów sekwencyjnych używa typu **reg**, a nie typu **logic**.

Jeden wiersz tabeli prawdy określa jedną wartość wyjściową dla jednego zestawu wartości wejściowych. Wiersz tablicy prawdy składa się z pól oddzielonych dwukropkiem.

Format wiersza tablicy prawdy dla prymitywów kombinacyjnych składa się z dwóch pól i wygląda następująco:

```
input_logic_values : output_logic_value;
```

Format wiersza tablicy prawdy dla prymitywów sekwencyjnych składa się z trzech pól i wygląda tak:

```
input_logic_values : previous_state : output_logic_value;
```

gdzie: *input_logic_values* to zestaw wartości wejściowych; *output_logic_value* to wartość wyjściowa; *previous_state* to wartość poprzedniego stanu wyjściowego dla prymitywów sekwencyjnych.

Cechą charakterystyczną prymitywów sekwencyjnych jest to, że mają one *stan wewnętrzny* (*internal state*). Stan prymitywu sekwencyjnego jest określany przez wartość jego wyjścia.

Wartości wejściowe w wierszu tabeli prawdy oddzielone są białymi znakami i muszą odpowiadać kolejności terminali wejściowych. Jeśli dowolna kombinacja wartości wejściowych w tabeli prawdy nie jest zdefiniowana, to wartością wyjściową dla tej kombinacji jest x.

Symbole, które mogą wystąpić w opisie tablicy prawdy, zostały wymienione w tabeli 17.3.

TABELA 17.3. Znaki występujące w tabeli prawdy prymitywów użytkownika

Symbol	Wartość	Opis
0	logiczne 0	wartość na wejściu lub wyjściu
1	logiczna 1	wartość na wejściu lub wyjściu
x lub X	wartość nieznana	wartość na wejściu lub wyjściu oraz w polu stanu sekwencyjnego UDP
?	wartość niezdefiniowana (*don't care*); może mieć wartość 0, 1 lub x	wartość na wejściu, niedopuszczalna na wyjściu
b lub B	wartość niezdefiniowana; może mieć wartość 0 lub 1	wartość na wejściu, niedopuszczalna na wyjściu
–	nie ma żadnych zmian	dozwolone tylko na wyjściu sekwencyjnym UDP
(vw)	zmiana wartości wejściowej z v na w	dozwolone tylko na wejściu; v i w mogą być dowolnymi wartościami 0, 1, x, ? lub b, np. (01) reprezentuje przejście od 0 do 1
*	to samo, co (??)	każda zmiana wartości wejściowej
r lub R	to samo, co (01)	dodatnie (rosnąca) zbocze na wejściu
f lub F	to samo, co (10)	ujemne (opadające) zbocze na wejściu
p lub P	to samo, co (01), (0x) lub (x1)	potencjalnie dodatnie zbocze na wejściu
n lub N	to samo, co (10), (1x) lub (x0)	potencjalnie ujemne zbocze na wejściu

ŹRÓDŁO: oprac. własne.

Tylko jeden z sygnałów wejściowych (pełniący funkcję wejścia zegarowego dla prymitywów sekwencyjnych) może mieć wartości przejściowe sygnału wejściowego. Jeśli taki terminal wejściowy istnieje, musi on definiować wartości przejściowe dla wszystkich wierszy tabeli prawdy.

Jeśli wartość przejściowa jest zdefiniowana dla jednego wejścia, prymityw staje się wrażliwy na przejścia na wszystkich wejściach. Wszystkie inne wejścia muszą mieć wiersze w tabeli prawdy, aby pokryć przejścia. Jeśli jakieś przejście nie zostanie znalezione, to wartością na wyjściu prymitywu dla tego przejścia będzie x.

17.3.2. Kombinacyjne prymitywy użytkownika

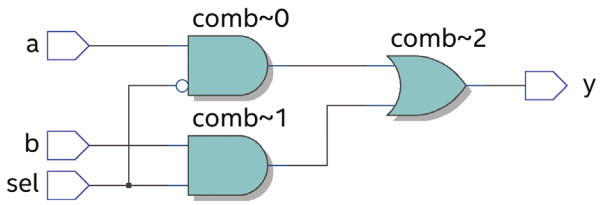
W przypadku kombinacyjnych prymitywów użytkownika wartość wyjściowa jest określona wyłącznie jako funkcja aktualnych wartości wejść. Podczas deklarowania kombinacyjnego prymitywu użytkownika pożądane jest określenie wartości wyjściowej dla wszystkich możliwych kombinacji wejść. Kombinacje wejść, które nie są jednoznacznie określone, ustawiają wyjście na wartość nieznaną x.

Poniższy przykład opisuje multiplekser dwuwejściowy jako prymityw zdefiniowany przez użytkownika.

Przykład 17.6. Prymityw użytkownika multipleksera dwuwejściowego

```
primitive my_mux (y, a, b, sel);      // opis prymitywu użytkownika, stary styl
        output y;                     // deklaracja wyjścia
        input sel, a, b;              // deklaracja wejść

        table
                // a b sel : y
                0 ?  0 :  0;          // a jest wybrane, wartość b jest nieokreślona
                1 ?  0 :  1;          // a jest wybrane, wartość b jest nieokreślona
                ? 0  1 :  0;          // b jest wybrane, wartość a jest nieokreślona
                ? 1  1 :  1;          // b jest wybrane, wartość a jest nieokreślona
                0 0  x :  0;          // x na wejściu sterującym
                1 1  x :  1;
        endtable
endprimitive
```

Implementacja projektu my_mux została przedstawiona na rys. 17.6.



RYS. 17.6. **Wyniki syntezy prymitywu użytkownika my_mux**

ŹRÓDŁO: **oprac. własne.**

17.3.3. Sekwencyjne prymitywy użytkownika wrażliwe na poziom sygnału sterującego

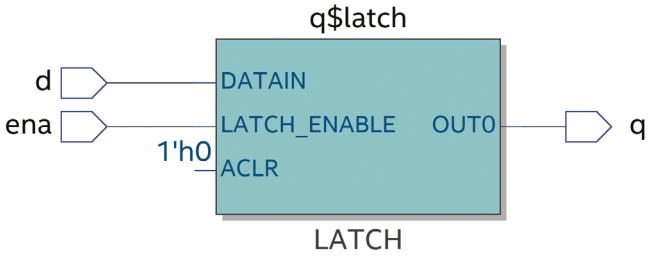
W opisie sekwencyjnych prymitywów użytkownika wrażliwych na poziom sygnału sterującego wyjście jest zadeklarowane z typem **reg**, a każdy wiersz tablicy prawdy ma dodatkowe pole z wartością stanu aktualnego (bieżącego). Na przykład zatrzask typu D można opisać w następujący sposób:

Przykład 17.7. Prymityw użytkownika zatrzasku typu D

```
primitive my_latch
        (output reg q,
        input ena, d);

        table
        // ena  d  : q  : q+
           1    1  : ?  : 1 ;
           1    0  : ?  : 0 ;
           0    ?  : ?  : - ;     // bez zmian
        endtable
endprimitive
```

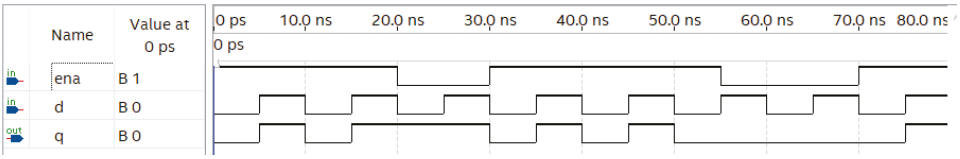
W powyższym przykładzie wejście *d* jest wejściem danych, a *ena* jest sygnałem sterującym. Gdy *ena* = 1, wyjście (*q*+) powtarza wartość wejścia *d* niezależnie od wartości stanu bieżącego (*q*). Gdy *ena* = 0, wyjście (*q*+) nie zmienia swojej wartości i jest również niezależne od wartości na wejściu (*d*) i aktualnego stanu (*q*). Takie zachowanie odpowiada dokładnie działaniu zatrzasku (transparentnego przerzutnika typu D).

Wyniki syntezy i symulacji projektu my_latch są przedstawione odpowiednio na rys. 17.7 i 17.8.



RYS. 17.7. Wyniki syntezy projektu my_latch

ŹRÓDŁO: oprac. własne.



RYS. 17.8. **Wyniki symulacji projektu my_latch**

ŹRÓDŁO: **oprac. własne.**

Na rys. 17.7 widać, że zatrzask D jest zaimplementowany z wykorzystaniem zatrzasku kompilatora Quartus, a wyniki symulacji (rys. 17.8) również opisują sposób funkcjonowania takiego zatrzasku.

17.3.4. Sekwencyjne prymitywy użytkownika wrażliwe na zbocze sygnału sterującego

Przy opisie prymitywów sekwencyjnych czułych na zbocze jedno z wejść jest zdefiniowane jako *sygnał zegarowy* (nazywany również *sygnałem synchronizującym*), dla którego możliwe są następujące wartości w tabeli prawdy: (vw), r, f, p i n. W jednym wierszu tabeli prawdy tylko jeden sygnał wejściowy może wskazywać na zmianę wartości sygnału. Na przykład poniższy opis jest nieprawidłowy:

```
(01)  (01) 0 : 0 : 1;      // błąd: zmiana sygnału jest określona dla dwóch wejść
```

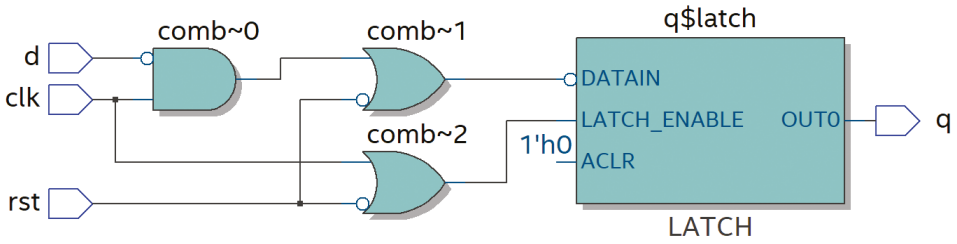
Poniższy przykład pokazuje zdefiniowany przez użytkownika prymityw przerzutnika typu D z sygnałem resetującym rst o aktywnym poziomie niskim.

Przykład 17.8. Sekwencyjny prymityw użytkownika przerzutnika typu D z sygnałem reset

```
primitive my_DFF_reset
        (output reg q = 0,
        input d, clk, rst);

        table
            // d  clk  rst  : state : q
               ?   ?    0   :   ?   : 0;   // reset na niskim poziomie rst
               0   R    1   :   ?   : 0;   // przy rosnącym zboczu stan wyjścia przyjmuje
                                            // wartość wejścia
               1   R    1   :   ?   : 1;   // przy rosnącym zboczu stan wyjścia przyjmuje
                                            // wartość wejścia
               ?   N    1   :   ?   : -;   // ignorowanie opadającego zbocza
               *   ?    1   :   ?   : -;   // ignorowanie wszystkich zboczy na wejściu d
```

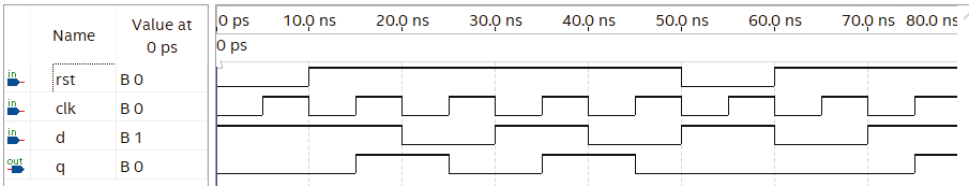
```
        ? ? P : ? : -;  // ignorowanie dodatniego zbocza na rst
        0 (0X) 1 : 0 : -;  // usuwanie drgań (bouncing)
        1 (0X) 1 : 1 : -;  // usuwanie drgań
    endtable
endprimitive
```

Wyniki syntezy i symulacji projektu my_DFF_reset przedstawiono odpowiednio na rys. 17.9 i 17.10.



RYS. 17.9. **Wyniki syntezy przerzutnika typu D zdefiniowanego przez użytkownika**

ŹRÓDŁO: **oprac. własne.**



RYS. 17.10. **Wyniki przerzutnika typu D zdefiniowanego przez użytkownika**

ŹRÓDŁO: **oprac. własne.**

Na rys. 17.9 widać, że przerzutnik typu D jest zaimplementowany z wykorzystaniem zatrzasku kompilatora Quartus, ale wyniki symulacji (rys. 17.10) odpowiadają funkcjonowaniu przerzutnika typu D.

17.3.5. Inicjalizacja sekwencyjnych prymitywów użytkownika

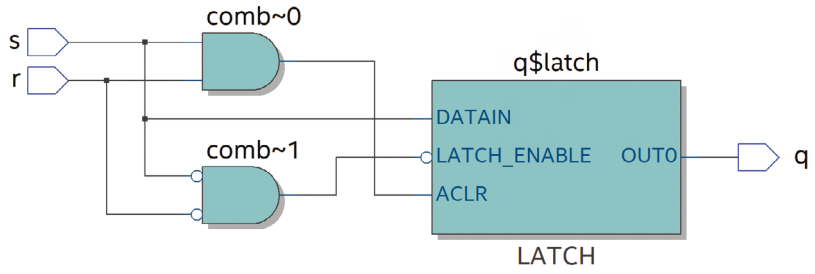
Wyjściom sekwencyjnych prymitywów użytkownika można nadać wartości początkowe. Do tego celu służy operator **initial**. W opisie prymitywu sekwencyjnego może on wystąpić tylko raz i może zawierać tylko jedno proceduralne przypisanie wyjścia prymitywu. Przypisywanymi wartościami mogą być: 1›bx, 1'b0, 1'b1, 1 lub 0.

Poniższy przykład opisuje przerzutnik typu SR z początkową wartością wyjściową 1.

Przykład 17.9. Sekwencyjny prymityw użytkownika przerzutnika SR z inicjalizacją wyjścia

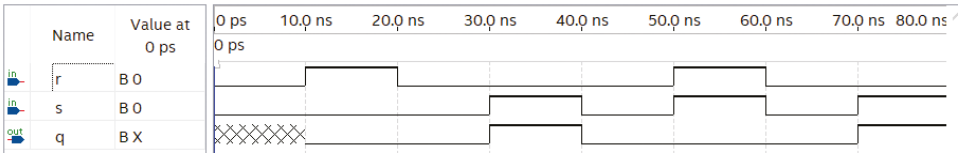
```
primitive my_SRFF
        (output reg q,
        input s, r);

        initial q = 1'b1;          // inicjalizacja wyjścia

        table
        //      s r   q   q+
                0 r : ? : 0 ;
                r 0 : ? : 1 ;
                f 0 : 1 : - ;
                0 f : 0 : - ;
                1 1 : ? : 0 ;
        endtable
endprimitive
```

Wyniki syntezy i symulacji projektu my_SRFF_reset przedstawiono odpowiednio na rys. 17.11 i 17.12.



RYS. 17.11. Wyniki syntezy przerzutnika SR zdefiniowanego przez użytkownika z inicjalizacją wyjścia

ŹRÓDŁO: oprac. własne.



RYS. 17.12. Wyniki symulacji przerzutnika SR zdefiniowanego przez użytkownika z inicjalizacją wyjścia

ŹRÓDŁO: oprac. własne.

Na rys. 17.11 widać, że przerzutnik SR użytkownika jest zaimplementowany z wykorzystaniem zatrzasku typu D kompilatora Quartus. Wyniki symulacji (rys. 17.12) pokazują, że inicjalizacja wartości początkowej prymitywów użytkownika nie jest zaimplementowana w ramach syntezy.

17.3.6. Sekwencyjne prymitywy użytkownika wrażliwe zarówno na poziom, jak i na zbocze sygnału sterującego

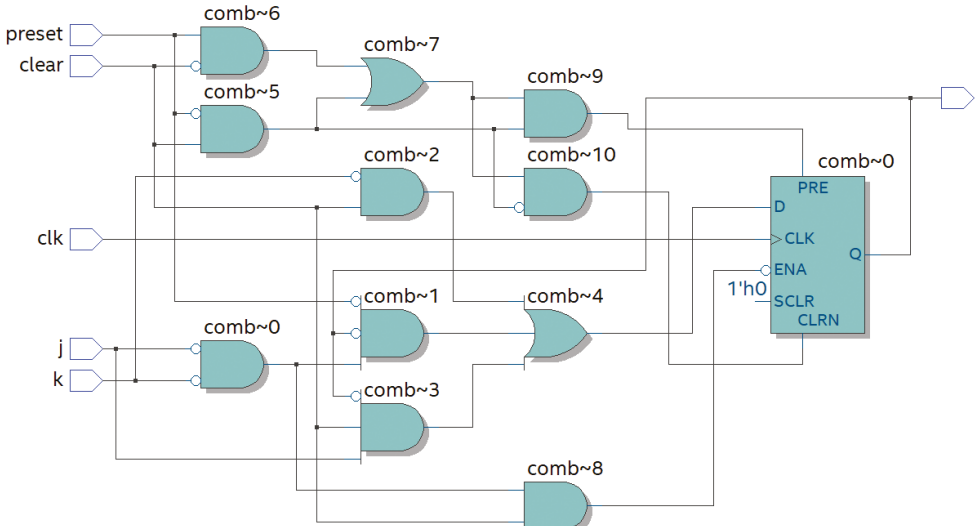
Mechanizm UDP języka SystemVerilog pozwala na opisywanie prymitywów użytkownika, które są wrażliwe zarówno na poziom, jak i na zbocze sygnału sterującego. W przypadku różnych wartości wyjściowych dla poziomu i zbocza sygnału sterującego o wyniku decyduje wrażliwość na poziom sygnału sterującego.

Poniżej przedstawiono przykład przerzutnika zatrzaskowego typu JK z asynchronicznymi sygnałami ustawiania *preset* i zerowania *clear*.

Przykład 17.10. Sekwencyjny prymityw użytkownika przerzutnika JK

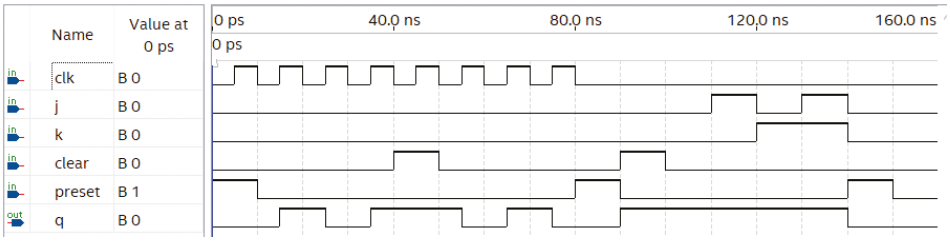
```
primitive jk_latch_ff
        (output reg q,
        input clk, j, k, preset, clear);

        table
                //      clk     jk      pc      q       q+
                        ?       ??      01      : ? :   1 ;  // logika ustawiania
                        ?       ??      *1      : 1 :   1 ;
                        ?       ??      10      : ? :   0 ;  // logika resetowania
                        ?       ??      1*      : 0 :   0 ;
                        // przypadki normalnej synchronizacji
                        r       00      00      : 0 :   1 ;
                        r       00      11      : ? :   - ;
                        r       01      11      : ? :   0 ;
                        r       10      11      : ? :   1 ;
                        r       11      11      : 0 :   1 ;
                        r       11      11      : 1 :   0 ;
                        f       ??      ??      : ? :   - ;
                        // przypadki przejść j i k
                        b       *?      ??      : ? :   - ;
                        b       ?*      ??      : ? :   - ;
        endtable
endprimitive
```

Wyniki syntezy i symulacji projektu jk_latch_ff przedstawiono odpowiednio na rys. 17.13 i 17.14.



RYS. 17.13. **Wyniki syntezy przerzutnika zatrzaskowego użytkownika typu JK**

ŹRÓDŁO: **oprac. własne.**



RYS. 17.14. **Wyniki symulacji przerzutnika zatrzaskowego użytkownika typu JK**

ŹRÓDŁO: **oprac. własne.**

Na rys. 17.13 widać, że przerzutnik zatrzaskowy JK jest zaimplementowany z wykorzystaniem przerzutnika D kompilatora Quartus. Wyniki symulacji (rys. 17.14) pokazują, że prymityw reaguje zarówno na zbocza, jak i na poziomy sygnałów wejściowych.

17.3.7. Tworzenie instancji prymitywów użytkownika

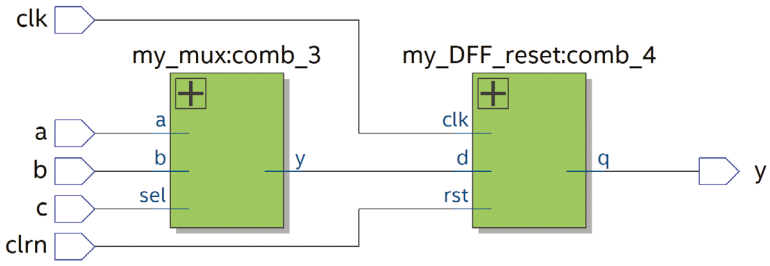
Instancje prymitywów użytkownika są tworzone w taki sam sposób jak instancje bramek i podobnie jak tam, nazwa instancji dla użytkownika prymitywu jest opcjonalna. Można również tworzyć tablice prymitywów użytkownika.

Poniższy kod jest przykładem użycia prymitywów użytkownika.

Przykład 17.11. Użycie prymitywów użytkownika *my_mux* i *my_DFF_reset*

```
module user_primitives
        (input a, b, c, clk, clrn,
        output y);
        wire r;                          // zmienna pośrednia

        my_mux (r,a,b,c);                // multiplekser użytkownika
        my_DFF_reset (y,r,clk,clrn);     // D-przyrzutnik użytkownika
endmodule
```

Wynik syntezy projektu user_primitives przedstawiono na rys. 17.15.



RYS. 17.15. Wyniki syntezy projektu user_primitives

ŹRÓDŁO: oprac. własne.

17.4. Wnioski

Dostępne elementy projektowe języka SystemVerilog można podzielić na: prymitywy języka SystemVerilog; prymitywy kompilatora; biblioteki prymitywów używanego środowiska zintegrowanego; biblioteki bloków IP środowiska zintegrowanego; publiczne biblioteki bloków IP; biblioteki bloków IP innych producentów.

Prymitywy SystemVerilog mogą być używane w dowolnym miejscu projektu bez wcześniejszej definicji lub deklaracji. Nazwy prymitywów są zawsze widoczne w całym drzewie hierarchii projektu.

Biblioteki prymitywów środowiska zintegrowanego są podobne w użyciu do podstawowych prymitywów języka SystemVerilog.

Wbudowane prymitywy SystemVerilog dzielą się na dwie grupy: prymitywy bramek i prymitywy przełączników. Te pierwsze obejmują również bufory. Prymitywy przełączające reprezentują tranzystory różnych typów. Standard języka SystemVerilog definiuje 14 prymitywów bramek i 12 prymitywów przełączników.

Użycie konstrukcji *range* (zakres) pozwala na deklarowanie tablic prymitywów.

Terminale prymitywów odpowiadają portom modułów. Terminale wyjściowe i dwukierunkowe są zawsze pierwsze na liście terminali. Do terminali prymitywów podłączane są sygnały zewnętrzne (sieci).

Prymitywy bramek dzielą się na następujące grupy: bramki o n wejściach i jednym wyjściu (**and**, **nand**, **or**, **nor**, **xor**, **xnor**); bramki o jednym wejściu i n wyjściach (**buf**, **not**); bufory o trzech stanach (**bufif1**, **bufif0**, **notif1**, **notif0**); bramki podciągające (**pulldown**, **pullup**).

Prymitywy przełączające dzielą się na przełączniki MOS (**cmos**, **nmos**, **pmos**, **rcmos**, **rnmos**, **rpmos**) oraz przełączniki dwukierunkowe (**rtran**, **rtranif0**, **rtranif1**, **tran**, **tranif0**, **tranif1**).

Siła sterownika (*drive strength*) określa moc logiczną wartości wyjściowych bramki i jest określona tylko dla prymitywów bramek. Prymitywy przełączające przenoszą siłę sterownika wejściowego na wyjście.

Konstrukcja *delay* reprezentuje opóźnienie czasowe dla sygnału, który ma przejść przez prymityw. Domyślnie opóźnienie wynosi 0 jednostek czasu. Siła sterownika nie ma wpływu na opóźnienie.

Konstrukcja zakresu *range* pozwala na zdefiniowanie tablicy prymitywów instancji, z których każda jest związana z określonym bitem wektora. Szerokość wektora bitowego musi być równa liczbie elementów w tablicy. Jeśli zamiast wektora użyty jest skalar, jest on połączony ze wszystkimi instancjami tablicy.

Bramki logiczne obejmują prymitywy **and**, **nand**, **or**, **nor**, **xor** i **xnor**. Mają jedno wyjście i mogą mieć jedno lub więcej wejść.

Bramki logiczne realizują odpowiednie funkcje logiczne. Jeśli jedno z wyjść ma wartość x lub z, to wyjściem będzie x.

Prymitywy **buf** i **not** są buforami i mają jedno wejście oraz mogą mieć jedno lub więcej wyjść.

Bufor **buf** powtarza wartość wejścia, a bufor **not** ją odwraca. W przypadku wartości z na wejściu buforów wyjściem będzie x.

Prymitywy **bufif1**, **bufif0**, **notif1** i **notif0** definiują bufory o trzech stanach: **bufif1** jest zwykłym buforem o trzech stanach; **bufif0** jest buforem o trzech stanach z inwersją sygnału sterującego; **notif1** jest buforem o trzech stanach z inwersją wyjścia; **notif0** jest buforem o trzech stanach z inwersją wyjścia i inwersją sygnału sterującego.

Prymitywy przełączające opisują zachowanie tranzystorów. Przełączniki MOS obejmują następujące prymitywy: **nmos**, **pmos**, **rnmos**, **rpmos**, **cmos** i **rcmos**.

Prymitywy **nmos**, **pmos**, **rnmos** i **rpmos** są podobne w funkcji do buforów **bufif** i reprezentują jednokierunkowe kanały danych. Prymityw **cmos** jest połączeniem przełączników **pmos** i **nmos**, a prymityw **rcmos** jest połączeniem przełączników **rpmos** i **rnmos**.

Przełączniki dwukierunkowe zawierają następujące prymitywy: **tran**, **rtran**, **tranif1**, **rtranif1**, **tranif0** i **rtranif0**. Nie opóźniają one przechodzących przez nie sygnałów. Przełączniki **tranif1**, **rtranif1**, **tranif0** i **rtranif0**, gdy są włączone, przepuszczają sygnały, a gdy są wyłączone, blokują je. Prymitywy **tran** i **rtran** zawsze przepuszczają sygnały, nie można ich wyłączyć.

Źródła sygnału mogą być określane przez prymitywy **pullup** i **pulldown**. Prymityw **pullup** ustawia wartość logiczną 1, a prymityw **pulldown** ustawia wartość logiczną 0 na elementach sieci podłączonych do jego terminala.

Do tworzenia prymitywów zdefiniowanych przez użytkownika język SystemVerilog udostępnia specjalny mechanizm zwany *user defined primitives* (UDP).

Wszystkie terminale (porty) w prymitywach zdefiniowanych przez użytkownika muszą mieć szerokość jednego bitu. Dla prymitywu dozwolone jest tylko jedno wyjście, które musi być pierwszym na liście terminali. Maksymalna liczba wejść dla prymitywów kombinacyjnych wynosi 10, dla prymitywów sekwencyjnych 9. Funkcjonalność prymitywu jest określona za pomocą *tablicy prawdy* (table_entry). Do określenia wartości w tablicy prawdy można użyć wartości logicznych 0, 1, z i x, przy czym wartość z jest interpretowana jako x. Jeden wiersz tabeli prawdy określa jedną wartość wyjściową dla jednego zestawu wartości wejściowych.

Stan prymitywu sekwencyjnego jest określany przez wartość jego wyjścia.

Jeśli dowolna kombinacja wartości wejściowych w tablicy prawdy jest nieokreślona, to dla tej kombinacji wartością wyjściową jest x.

Tylko jeden z sygnałów wejściowych (sygnał zegarowy) dla prymitywów sekwencyjnych może mieć wartości przejściowe sygnału wejściowego.

Dla kombinacyjnych prymitywów użytkownika wartość wyjściowa jest określona wyłącznie jako funkcja rzeczywistych wartości wejść. Kombinacje wejść, które nie są jednoznacznie określone, ustawiają wyjście na wartość nieznaną x.

W opisie sekwencyjnych prymitywów użytkownika, które są wrażliwe na poziom sygnału sterującego, wyjście jest deklarowane za pomocą typu **reg**, a w każdym wierszu tabeli prawdy znajduje się dodatkowe pole z wartością aktualnego (bieżącego) stanu.

Przy opisie sekwencyjnych prymitywów użytkownika, które są wrażliwe na zbocze sygnału sterującego, jedno z wejść definiuje się jako *sygnał zegarowy* (zwany też *sygnałem synchronizującym*). Sygnał ten określa się też jako sygnał przełączający i możliwe są dla niego następujące wartości w tabeli prawdy: (vw), r, f, p i n.

Wyjścia sekwencyjnych prymitywów użytkownika mogą być ustawione na wartości początkowe za pomocą operatora **initial**.

Mechanizm UDP języka SystemVerilog pozwala na opisywanie prymitywów użytkownika, które są wrażliwe zarówno na poziom, jak i na zbocze sygnału sterującego.

Instancje prymitywów użytkownika są tworzone w taki sam sposób jak instancje bramek. Nazwa instancji dla prymitywów użytkownika jest opcjonalna. Można również tworzyć tablice prymitywów użytkownika.

18. Konfiguracja projektu

Pojęcie *konfiguracji projektu* (*project configuration*) jest bezpośrednio związane z tworzeniem dużych projektów przez grupę projektantów. Gdy pewne oddzielne części projektu są opracowywane przez różnych inżynierów, jest bardziej prawdopodobne, że moduły i prymitywy użytkownika będą miały tę samą nazwę, np. *adder*, *register*, *counter*, *memory* itp. Przy wspólnym budowaniu dużego projektu pojawia się pytanie, gdzie jest kod źródłowy każdego konkretnego modułu, a także każdej jego instancji. W języku SystemVerilog problem ten jest rozwiązany za pomocą konfiguracji projektu.

W SystemVerilog konfiguracja projektu jest zdefiniowana w postaci bloków konfiguracyjnych i zorganizowana w następujący sposób. Nazwy komponentów projektu (modułów, prymitywów, pakietów, interfejsów itp.) są identyfikowane za pomocą *komórek* (*cells*), które są umieszczane w *bibliotekach* (*libraries*). Pliki map bibliotek (*library map files*) mapują nazwy bibliotek na fizyczną lokalizację plików kodu źródłowego. Bloki konfiguracyjne wykorzystują nazwy bibliotek i komórek do lokalizowania konkretnych składników projektu.

Bloki konfiguracyjne mogą być użyte do określenia: modułu najwyższego poziomu, zestawu bibliotek do wyszukiwania nazwy modułu lub prymitywu, zestawu bibliotek do wyszukiwania nazwy instancji modułu bądź prymitywu, kolejności przeszukiwania listy bibliotek oraz domyślnej biblioteki wyszukiwania. Bloki konfiguracyjne są częścią kodu źródłowego projektu i razem z tym kodem są opisywane.

Jednak lokalizacja plików kodu źródłowego w trakcie realizacji projektu może się zmienić z dowolnego powodu. W tym przypadku należy zmodyfikować jedynie pliki map bibliotek, a kod źródłowy modułu i bloki konfiguracyjne pozostają niezmienione.

18.1. Konfiguracje

Język SystemVerilog wykorzystuje pojęcie *konfiguracji* (*configuration*), aby ułatwić współdzielenie projektów pomiędzy zespołami projektantów i dokładnie odwzorować sesję symulacyjną. Konfiguracja to zbiór wytycznych (reguł) określających dokładną lokalizację kodu źródłowego opisującego każdą instancję modułu lub prymitywu, który jest używany w projekcie. Konfiguracje są określane za pomocą *bloków*

konfiguracyjnych (*configuration blocks*). Operacja wyboru kodu źródłowego dla instancji modułu lub prymitywu nazywana jest *wiązaniem* (*binding*).

Opis projektu rozpoczyna się od modułu najwyższego poziomu (bądź modułów, ponieważ w projekcie SystemVerilog może być kilka modułów najwyższego poziomu). Na podstawie jego opisu źródłowego zostaną najpierw odnalezione instancje modułu (czyli potomkowie), następnie opisy źródłowe dla definicji modułów podrzędnych i tak dalej, aż każda instancja w projekcie zostanie odwzorowana na opis źródłowy.

Blok konfiguracyjny jest częścią kodu źródłowego projektu w języku SystemVerilog i jest razem z tym kodem opisany, choć poza modułami projektu. Bloki konfiguracyjne mogą znajdować się albo w tych samych plikach kodu źródłowego projektu, albo w oddzielnych.

Podstawową jednostką konfiguracji projektu jest *komórka* (*cell*). Komórka to nazwa modułu, prymitywu, interfejsu, programu, pakietu lub innej konfiguracji.

W jednostkach konfiguracyjnych pojawiają się *biblioteki* (*libraries*), tj. logiczne zbiory komórek (czyli nazw). Nazwa każdej komórki powinna pokrywać się z odpowiadającą jej nazwą modułu, prymitywu, interfejsu, programu, pakietu lub konfiguracji.

Pliki map bibliotek (*library map files*) służą do mapowania nazw bibliotek na fizyczną lokalizację plików kodu źródłowego. Informacje łączące instancje bibliotek i modułów mogą być odwzorowane podczas symulacji za pomocą znaku formatu %l, który wypisze nazwę biblioteki i nazwę modułu w postaci:

library_name.cell_name

gdzie *library_name* jest nazwą biblioteki, a *cell_name* jest nazwą komórki odpowiadającej modułowi, w którym wykonywane jest polecenie wypisania (informacji wyjściowej).

18.2. Bloki konfiguracyjne

Blok konfiguracyjny (*configuration block*) zawiera zestaw instrukcji dotyczących wyszukiwania opisu źródła w SystemVerilog w celu związania konkretnej instancji projektu.

Blok konfiguracyjny ma następujący format:

```
config config_name;
      design lib_name.cell_name;
      default liblist list_of_library_names;
      cell lib_name.cell_name liblist list_of_library_names;
      cell lib_name.cell_name use lib_name.cell_name:config_name;
      instance hierarchy_name liblist list_of_library_names;
      instance hierarchy_name use lib_name.cell_name:config_name;
endconfig
```

W powyższej definicji **config** i **endconfig** są słowami kluczowymi określającymi jednostkę konfiguracyjną.

Operator **design** określa bibliotekę i komórkę modułu (modułów) najwyższego poziomu w hierarchii projektu. W bloku konfiguracyjnym może być tylko jeden taki operator, ale można wymienić wiele modułów najwyższego poziomu. Operator **design** musi być pierwszym operatorem w bloku konfiguracyjnym. Opcjonalna nazwa biblioteki *lib_name* określa, która biblioteka zawiera komórkę. Jeśli nazwa biblioteki jest pominięta, do znalezienia komórki używana jest biblioteka zawierająca blok konfiguracyjny. Obowiązkowa pozycja *cell_name* jest nazwą modułu najwyższego poziomu w hierarchii projektu, który reprezentuje blok konfiguracyjny.

Operator **default liblist** określa, w jakich bibliotekach szukać początkowego opisu instancji, dla których nie dopasowano określonego punktu wyboru. Biblioteki są przeszukiwane w kolejności, w jakiej są wymienione. W przypadku wielu projektów lista bibliotek może zawierać wszystkie biblioteki, które są potrzebne do zdefiniowania konfiguracji. *List_of_library_names* jest oddzieloną przecinkami listą symbolicznych nazw bibliotek. Operator **default** jest często używany razem z konstrukcją **liblist**, która określa uporządkowany zestaw bibliotek do przeszukania dla bieżącej instancji. Listy bibliotek **liblists** są dziedziczone hierarchicznie w dół, gdy instancje są związane.

Operator **cell** definiuje określony zestaw bibliotek, w których wyszukiwany jest kod źródłowy dla nazwy modułu lub prymitywu. Opcjonalny element *lib_name* określa bibliotekę symboliczną, która zawiera komórkę; obowiązkowy element *cell_name* jest nazwą modułu bądź prymitywu.

Operator **instance** definiuje określony zestaw bibliotek, w których wyszukiwany jest kod źródłowy dla konkretnej instancji modułu albo prymitywu. Element *hierarchy_name* jest pełną ścieżką hierarchiczną nazwy instancji modułu lub prymitywu.

Klauzula **use** określa konkretne wiązanie dla wybranej komórki. Może być używana tylko w połączeniu z operatorem wyboru instancji **instance** lub komórki **cell**. Określa dokładną bibliotekę i komórkę, do której mapowana jest wybrana komórka bądź instancja.

Jeśli lib.cell, do którego odwołuje się instrukcja **use**, jest konfiguracją, której nazwa jest taka sama jak nazwa modułu lub prymitywu w tej samej bibliotece, to do lib.cell można dodać dodatkowy przyrostek **:config**, aby wyraźnie określić konfigurację. Jeśli nazwa biblioteki jest pominięta, biblioteka jest dziedziczona z komórki nadrzędnej.

Przykładowo niech istnieje następujący moduł najwyższego poziomu:

```
module top();
        adder a1(...);
        adder a2(...);
endmodule
```

Niech moduł *top* i instancja sumatora *a1* będą w bibliotece *rtlLib*, a instancja sumatora *a2* w bibliotece *gateLib*. Aby określić ten konkretny zestaw wiązań, można użyć następującej konfiguracji:

```
config cfg1;                          // początek bloku konfiguracyjnego
      design rtlLib.top;              // definicja modułu najwyższego poziomu
      default liblist rtlLib;         // definicja biblioteki domyślnej
      instance top.a2 liblist gateLib; // definicja biblioteki dla instancji top.a2
endconfig
```

Powyższa konfiguracja określa, że moduł *top* znajduje się w bibliotece *rtlLib*. Wszystkie moduły projektu są również domyślnie w niej wyszukiwane, ale instancja *top.a2* będzie wyszukiwana w bibliotece *gateLib*.

18.3. Plik mapy biblioteki

Do mapowania bibliotek na lokalizację plików fizycznych służy osobny plik o nazwie *plik mapy biblioteki* (*library map file*). Jeśli w trakcie projektowania z jakiegokolwiek powodu zmieni się lokalizacja plików kodu źródłowego, należy zmodyfikować jedynie plik mapy biblioteki. Kod źródłowy SystemVerilog i bloki konfiguracyjne nie są modyfikowane. Plik mapy biblioteki nie jest kodem źródłowym języka SystemVerilog.

Plik mapy biblioteki może zawierać operatory **library** i **include**, a także komentarze SystemVerilog. Operatory te mają następujący format:

library *lib_name list_of_file_paths*, **-incdir** *list_of_file_paths;*
include *library_map_file_path*;

Element *lib_name* określa nazwę biblioteki, która ma być wymieniona w blokach konfiguracyjnych. Element *list_of_file_paths* jest listą ścieżek systemu operacyjnego do określonych plików. Ścieżka zakończona ukośnikiem (/) obejmuje wszystkie pliki w danym katalogu (identycznie jak ścieżka zakończona znakiem /*). Ścieżka, która nie kończy się znakiem ukośnika (/), odnosi się do katalogu, w którym znajduje się plik mapy biblioteki. W oznaczeniu ścieżki można stosować następujące znaki specjalne:

- ? – znak odpowiadający dowolnemu pojedynczemu znakowi;
- * – znak odpowiadający dowolnej liczbie dowolnych znaków;
- ... – hierarchiczny symbol wieloznaczny, odpowiada dowolnej liczbie hierarchicznych katalogów;
- .. – definiuje katalog nadrzędny (najbliższy wyżej w hierarchii katalogów);
- . – definiuje katalog zawierający plik mapy biblioteki.

Na przykład:

```
library rtlLib *.v;          // wyświetla wszystkie pliki w bieżącym katalogu
                             // z przyrostkiem .v

library gateLib ./*.vg;      // wyświetla wszystkie pliki w bieżącym katalogu
                             // z przyrostkiem .vg
```

Konstrukcja **-incdir** określa, gdzie szukać plików include, do których odwołuje się dyrektywa **`include** w kodzie źródłowym SystemVerilog. Konstrukcja **include** pozwala jednemu plikowi mapy biblioteki na włączenie innego pliku tego samego typu.

18.4. Konfiguracje hierarchiczne

W sytuacjach, gdy pożądane jest określenie konkretnego zestawu reguł konfiguracyjnych dla podsekcji projektu, możliwe jest powiązanie konkretnej instancji modułu bezpośrednio z konfiguracją za pomocą *klauzuli wiążącej* (*binding clause*). Na przykład:

```
instance top.a1.c use lib1.c:config;
```

To określa, że instancja *top.a1.c* ma być zastąpiona przez hierarchię projektową określoną w konfiguracji lib1.c**:config**. Operator **design** w lib1.c**:config** określa rzeczywiste wiązanie dla instancji *top.a1.c*, a reguły określone w konfiguracji definiują konfigurację wszystkich innych *podinstancji* (*subinstances*) w *top.a1.c*.

18.5. Określanie wartości parametrów modułu w konfiguracji

Konfiguracje mogą być również używane do nadpisywania wartości parametrów zadeklarowanych w projekcie lub do ponownego definiowania parametrów dla poszczególnych instancji modułu. W tym celu w bloku konfiguracyjnym można zadeklarować parametry lokalne za pomocą słowa kluczowego **localparam**.

Na przykład załóżmy, że istnieje następujący moduł *adder*:

```
module adder #(parameter ID = "id",      // parametry modułu
                         W = 8,
                         D = 512)
      (...);                             // porty modułu
    ...
```

```
        $display(„ID = %s, W = %d, D = %d", ID, W, D);
        ...
endmodule: adder
```

i moduł najwyższego poziomu *top*:

```
module top (...);
        parameter WIDTH = 16;              // parametr dla całego projektu
        adder a1 (...);                    // instancja modułu adder
endmodule
```

Bez redefinicji parametrów instancja *a1* modułu *adder* wypisze, co następuje:

```
ID = id, W=8, D = 512
```

Poniższa konfiguracja nadpisuje domyślną wartość parametru WIDTH dla całego projektu, jak również parametr W dla instancji *a1* utworzonej w module *top*.

```
config cfgl;
        design rtlLib.top;                      // definicja modułu najwyższego poziomu
        instance top use #(.WIDTH(32));         // redefinicja parametru WIDTH
        instance top.a1 use #(.W(top.WIDTH));   // redefinicja parametru W dla
                                                // instancji top.a1
endconfig
```

Spowoduje to, że instancja *a1* modułu *adder* wypisze następujące dane:

```
ID = id, W=32, D = 512
```

Definiując parametry lokalne za pomocą **localparam** w konfiguracji, można nadpisać parametry tylko dla poszczególnych instancji modułu.

Na przykład załóżmy, że istnieje następujący moduł najwyższego poziomu:

```
module top4 ();
        parameter S = 16;
        adder a1 #(.ID("a1"))(...);
        adder a2 #(.ID("a2"))(...);
        adder a3 #(.ID("a3"))(...);
        adder a4 #(.ID(„a4"))(...);
endmodule
```

Poniższa konfiguracja zmienia wartość parametru W dla instancji *a1* i *a2* modułu *adder*:

```
config cfg2;
        localparam S = 24;
        design rtlLib.top4;
        instance top4.a1 use #(.W(top4.S));
        instance top4.a2 use #(.W(S));
endconfig
```

W efekcie zostaną wypisane następujące elementy:

```
ID = a1, W=16, D = 512
ID = a2, W=24, D = 512
ID = a3, W=8, D = 512
ID = a4, W=8, D = 512
```

18.6. Przykłady konfiguracji projektu

18.6.1. Wstępny opis projektu

Niech projekt będzie opisany czterema plikami top.v, adder.v, adder.vg i lib.map, które mają następującą zawartość:

plik top.v:

```
        module top(...);
                ...
                adder a1(...);
                adder a2(...);
        endmodule
        module foo(...);
                ... // rtl
        endmodule
```

plik adder.v:

```
        module adder(...);
                ... // rtl
                foo f1(...);
                foo f2(...);
        endmodule
        module foo(...);
                ... // rtl
        endmodule
```

plik adder.vg:

```
module adder(...);
    ... // gate-level
    foo f1(...);
    foo f2(...);
endmodule
module foo(...);
    ... // gate-level
endmodule
```

plik lib.map:

```
library rtlLib top.v;
library aLib adder.*;
library gateLib
    adder.vg;
```

W naszym przykładzie pliki top.v, adder.v i adder.vg są kompilowane wraz z określonym plikiem mapy biblioteki lib.map, który ma następującą postać:

```
library rtlLib top.v;          // biblioteka na poziomie RTL
library aLib adder.*;          // biblioteka aLib
library gateLib adder.vg;      // biblioteka na poziomie bramek
```

Daje to następującą strukturę biblioteki:

```
rtlLib.top        // z top.v
rtlLib.foo        // z top.v
aLib.adder        // z adder.v
aLib.foo          // rtl z adder.v
gateLib.adder     // z adder.vg
gateLib.foo       // z adder.vg
```

18.6.2. Użycie konfiguracji zdefiniowanej w pliku mapy biblioteki

W przypadku, gdy w projekcie nie ma bloków konfiguracyjnych, biblioteki są wyszukiwane zgodnie z kolejnością, w jakiej zostały zadeklarowane w pliku mapy bibliotek. Oznacza to, że wszystkie instancje modułu *adder* muszą używać pliku *aLib.adder*, ponieważ *aLib* jest pierwszą biblioteką, której specyfikacja zawiera komórkę o nazwie *adder*. Dodatkowo wszystkie instancje modułu *foo* muszą używać pliku *rtlLib.foo*, gdyż *rtlLib* jest pierwszą biblioteką, której specyfikacja zawiera komórkę o nazwie *foo*.

18.6.3. Użycie operatora default

Aby zawsze korzystać z modułu *foo* opisanego w *adder.v*, można użyć następującej konfiguracji:

```
config cfg1;
        design rtlLib.top
        default liblist aLib rtlLib;
endconfig
```

W tym przypadku operator **default liblist** zmienia kolejność wyszukiwania biblioteki w pliku *lib.map*. Dlatego biblioteka *aLib* jest zawsze wyszukiwana przed biblioteką *rtlLib*. Ponieważ biblioteka *gateLib* nie jest zawarta w **liblist**, nie można korzystać z opisów modułów *adder* i *foo* na poziomie bramek. Aby było to możliwe, należy dodać następującą konfigurację:

```
config cfg2;
        design rtlLib.top
        default liblist gateLib aLib rtlLib;
endconfig
```

Ta konfiguracja nakazuje najpierw przejrzeć bibliotekę *gateLib* i wybrać z pliku *adder.vg* opisy modułów *adder* i *foo* na poziomie bramek.

18.6.4. Użycie operatora cell

Poniższa konfiguracja pozwala na wykorzystanie opisu RTL z modułu *adder* oraz bramkowej reprezentacji modułu *foo* z biblioteki *gateLib*:

```
config cfg3;
        design rtlLib.top
        default liblist aLib rtlLib;
        cell foo use gateLib.foo;
endconfig
```

Tutaj operator **cell** wybiera wszystkie komórki o nazwie *foo* i jawnie wiąże je z reprezentacją bramek w bibliotece *gateLib*.

18.6.5. Użycie operatora instance

Poniższa konfiguracja pozwala instancji *top.a1* modułu *adder* (i jego potomkom) korzystać z opisu bramek, a instancji *top.a2* modułu *adder* (i jego potomkom) z reprezentacji RTL z biblioteki *aLib*:

```
config cfg4
        design rtlLib.top
        default liblist gateLib rtlLib;
        instance top.a2 liblist aLib;
endconfig
```

Ponieważ **liblist** jest dziedziczona, wszyscy potomkowie instancji *top.a2* dziedziczą jej **liblist** z operatora wyboru **instance**.

18.6.6. Użycie konfiguracji hierarchicznej

Niech nasz projekt zostanie opisany w taki sposób, aby w poniższej konfiguracji użyć komórki *rtlLib.foo* dla instancji *f1* oraz komórki *getLib.foo* dla instancji *f2*:

```
config cfg5;
        design aLib.adder;
        default liblist gateLib aLib;
        instance adder.f1 liblist rtlLib;
endconfig
```

Aby użyć tej konfiguracji *cfg5* dla instancji modułu *adder top.a2* i pobrać w pełni domyślny moduł *adder* z biblioteki *aLib* dla instancji *top.a1*, należy zastosować następującą konfigurację:

```
config cfg6;
        design rtlLib.top;
        default liblist aLib rtlLib;
        instance top.a2 use work.cfg5:config
endconfig
```

W tym przypadku operator **instance** wiąże konfigurację *work.cfg5:config*, która jest używana do wiązania z instancją *top.a2* i jej potomkami. Operator **design** w konfiguracji *cfg5* określa dokładne wiązanie dla instancji *top.a2*. Reszta konfiguracji *cfg5* definiuje zasady konstruowania potomków instancji *top.a2*. Zauważmy, że operator **instance** w konfiguracji *cfg5* jest „krewnym" własnego modułu *adder* najwyższego poziomu.

18.7. Wnioski

Konfiguracja to zestaw instrukcji określających dokładną lokalizację kodu źródłowego, opisujący każdą instancję modułu lub prymitywu, który jest używany w projekcie. Konfiguracje są definiowane za pomocą *bloków konfiguracyjnych* (*configuration blocks*).

Operacja wyboru kodu źródłowego dla instancji modułu lub prymitywu nazywana jest *wiązaniem* (*binding*).

W projekcie SystemVerilog może istnieć kilka modułów najwyższego poziomu.

Bloki konfiguracyjne są częścią kodu źródłowego projektu SystemVerilog, są opisane poza jego modułami. Bloki te mogą znajdować się w tych samych plikach co kod źródłowy projektu lub w oddzielnych.

Podstawową jednostką w konfiguracji projektu jest *komórka* (*cell*). Komórka to nazwa modułu, prymitywu, interfejsu, programu, pakietu lub innej konfiguracji.

W blokach konfiguracyjnych pojawiają się *biblioteki* (*libraries*), tj. logiczne zbiory komórek (czyli nazw). Nazwa każdej komórki powinna być zgodna z odpowiadającą jej nazwą modułu, prymitywu, interfejsu, programu, pakietu lub konfiguracji.

Pliki map bibliotek służą do mapowania nazw bibliotek na fizyczną lokalizację *plików mapy biblioteki* (*library map files*).

Blok konfiguracyjny zawiera zestaw instrukcji dotyczących poszukiwania opisu źródła w SystemVerilog w celu związania konkretnej instancji projektu.

Operator **design** określa bibliotekę i komórkę modułu (lub modułów) najwyższego poziomu w hierarchii projektu. W projekcie SystemVerilog może być kilka modułów najwyższego poziomu. W bloku konfiguracji może być tylko jeden operator **design**, ale można zdefiniować wiele modułów najwyższego poziomu. Operator ten musi być pierwszym operatorem w bloku konfiguracyjnym.

Operator **default liblist** określa, w jakich bibliotekach szukać początkowego opisu instancji, dla których nie dopasowano określonego punktu wyboru. Biblioteki są przeszukiwane w kolejności występowania na liście.

Operator **liblist** definiuje uporządkowany zestaw bibliotek do przeszukania dla bieżącej instancji.

Operator **cell** definiuje określony zestaw bibliotek, w których wyszukiwany jest kod źródłowy dla nazwy modułu lub prymitywu.

Operator **instance** definiuje określony zestaw bibliotek, w których wyszukiwany jest kod źródłowy dla konkretnej instancji modułu albo prymitywu.

Klauzula **use** określa konkretne wiązanie dla wybranej komórki. Może być używana tylko w połączeniu z operatorem wyboru **instance** lub komórki **cell**. Określa dokładną bibliotekę i komórkę, do której mapowana jest wybrana komórka bądź instancja. Jeśli lib.cell, do którego odwołuje się operator **use**, jest konfiguracją, dodawany jest przyrostek **:config**.

Jeśli w trakcie pracy nad projektem z jakiegokolwiek powodu zmieni się lokalizacja plików kodu źródłowego, należy zmodyfikować jedynie *plik mapy biblioteki* (*library map file*). Kod źródłowy SystemVerilog i bloki konfiguracyjne nie muszą być modyfikowane.

Jeśli w projekcie nie ma bloków konfiguracyjnych, biblioteki są wyszukiwane zgodnie z kolejnością, w jakiej zostały zadeklarowane w pliku mapy bibliotek.

Plik mapy biblioteki może zawierać operatory **library** i **include**, a także komentarze języka SystemVerilog. Do określenia ścieżki w pliku mapy biblioteki mogą być użyte znaki specjalne (?, *, ..., ...).

Konstrukcja **-incdir** określa, gdzie szukać plików include, do których odwołuje się dyrektywa `include w kodzie źródłowym SystemVerilog.

Konkretną instancję modułu można powiązać bezpośrednio z konfiguracją za pomocą *klauzuli wiążącej* (*binding clause*) zawierającej słowa kluczowe **instance**, **use** i **:config**.

Konfiguracje mogą być użyte do nadpisania wartości parametrów zadeklarowanych w projekcie lub nadpisania parametrów dla poszczególnych instancji modułu za pomocą słowa kluczowego **localparam**.

Podsumowanie

Należy przede wszystkim zwrócić uwagę, jakie elementy występujące w standardzie SystemVerilog Std 1800-2017 (Revision 2012) nie zostały uwzględnione w tej książce:

- planowanie semantyki symulacji zdarzeń (rozdział 4);
- klasy (rozdział 8);
- bloki synchronizacji (rozdział 14);
- synchronizacja i komunikacja między procesami (rozdział 15);
- asercje (rozdział 16);
- kontrolery (rozdział 17);
- tworzenie ograniczonych wartości losowych (rozdział 18);
- pokrycie funkcjonalne (rozdział 19);
- zadania i funkcje systemowe dotyczące symulacji (rozdział 20);
- programy (rozdział 24);
- bloki specyfikacji (rozdział 30);
- kontrola synchronizacji (rozdział 31);
- adnotacje wsteczne przy użyciu standardowego formatu opóźnienia (rozdział 32);
- konfiguracja zawartości projektu (rozdział 33);
- koperty zabezpieczone (rozdział 34);
- interfejsy aplikacji programowych (rozdziały 35–41).

Autorzy planują poruszyć większość z tych tematów w kolejnej książce poświęconej symulacji projektów z wykorzystaniem SystemVerilog.

Zwróćmy uwagę na pewne cechy języka SystemVerilog, dzięki którym staje się on coraz bardziej popularny wśród twórców sprzętu cyfrowego. Z jednej strony jest to prostota jego składni, znacznie przypominająca język programowania C. Ponieważ większość inżynierów-projektantów i programistów zna język C, nauka SystemVerilog dla tej kategorii użytkowników jest dość łatwa. Z drugiej strony język SystemVerilog daje duże możliwości w zakresie opisu urządzeń i systemów cyfrowych. Dzięki temu opis może być wykonywany zarówno do celu syntezy (realizacji sprzętowej), jak i do symulacji (dla sprawdzenia różnych właściwości projektowanego systemu). Należy zauważyć, że nie wszystkie języki opisu sprzętu mają tę właściwość.

Poza tym SystemVerilog pozwala opisywać złożone systemy na wszystkich poziomach projektowania – behawioralnym (funkcjonalnym), systemowym, abstrakcyjnym i transakcyjnym – tranzystorów, bramek, przerzutników, rejestrów, równań logicznych.

Język SystemVerilog daje również możliwości rozbudowy. Do tego celu służą mechanizmy UDP (umożliwiający definiowanie prymitywów użytkownika) i interfejsów aplikacji programowych. Dzięki mechanizmowi UDP użytkownik może zdefiniować własne prymitywy, które nie są dostępne w języku Verilog. Z kolei poprzez interfejsy aplikacji można połączyć język SystemVerilog z językami programowania i w ten sposób rozszerzyć go zgodnie z wymaganiami użytkownika.

Należy zaznaczyć, że SystemVerilog nie jest martwym językiem, jest ciągle udoskonalany, pojawiają się nowe konstrukcje, które znajdują odzwierciedlenie w jego standardach. Włączenie elementów sztucznej inteligencji do SystemVerilog, zwłaszcza służących do weryfikacji i symulacji projektów, jest postrzegane przez społeczność projektantów jako perspektywiczne.

Udana, przemyślana koncepcja języka Verilog jest potwierdzona jego popularnością od prawie 40 lat (nie każdy język programowania może się tym pochwalić). Od czasu wprowadzenia w 1983 roku był on stale ulepszany. W rezultacie powstał SystemVerilog, który zawiera wszystkie najlepsze koncepcje zaproponowane w ciągu ostatnich dwóch dekad z dziedziny lingwistycznego opisu modeli systemów na potrzeby syntezy i symulacji. W ostatnim czasie pojawiło się wiele nowych języków opisu sprzętu, które pozwalają na uwzględnienie specyfiki różnych urządzeń i systemów. Jednak Verilog i SystemVerilog nadal pozostają jednymi z najpopularniejszych języków projektowych o szerokim zastosowaniu. Dlatego można się spodziewać, że w ciągu najbliższych kilkudziesięciu lat kolejne wersje rozwojowe języka SystemVerilog staną się jednymi z głównych języków do projektowania sprzętu cyfrowego.

Wszelkie pytania i sugestie dotyczące książki lub jej zawartości można kierować do autorów na adres v.salauyou@pb.edu.pl i a.klimowicz@pb.edu.pl.

Lista przykładów

Dodatek. Słowa kluczowe języka SystemVerilog

accept_on
alias
always
always_comb
always_ff
always_latch
and
assert
assign
assume
automatic
before
begin
bind
bins
binsof
bit
break
buf
bufif0
bufif1
byte
case
casex
casez
cell
chandle
checker
class
clocking
cmos
config
const
constraint
context
continue
cover
covergroup
coverpoint
cross
deassign
default
defparam
design
disable
dist
do
edge else
end
endcase
endchecker
endclass
endclocking
endconfig
endfunction
endgenerate
endgroup
endinterface
endmodule
endpackage
endprimitive
endprogram
endproperty
endspecify
endsequence
endtable
endtask
enum
event
eventually
expect
export
extends
extern
final
first_match
for
force
foreach
forever
fork
forkjoin
function
generate
genvar
global
highz0
highz1
if
iff
ifnone
ignore_bins
illegal_bins
implements
implies import
incdir
include
initial
inout
input
inside
instance
int
integer
interconnect
interface
intersect
join
join_any
join_none
large
let
liblist
library
local
localparam
logic
longint
macromodule
matches
medium
modport
module
nand
negedge
nettype
new
nexttime
nmos
nor
noshowcancelled
not
notif0
notif1
null
or

output
package
packed
parameter
pmos
posedge
primitive
priority
program
property protected
pull0
pull1
pulldown
pullup
pulsestyle_ondetect
pulsestyle_onevent
pure
rand
randc
randcase
randsequence
rcmos
real
realtime
ref
reg
reject_on
release
repeat
restrict
return
rnmos
rpmos
rtran
rtranif0
rtranif1
s_always
s_eventually
s_nexttime
s_until
s_until_with
scalared
sequence
shortint
shortreal
showcancelled
signed
small
soft
solve
specify
specparam
static
string
strong
strong0
strong1
struct
super
supply0
supply1
sync_accept_on
sync_reject_on
table
tagged
task
this
throughout
time
timeprecision
timeunit
tran
tranif0
tranif1
tri
tri0
tri1
triand
trior
trireg type
typedef
union
unique
unique0
unsigned until
until_with
untyped
use
uwire
var
vectored
virtual
void wait
wait_order
wand
weak
weak0
weak1
while
wildcard
wire
with
within
wor
xnor
xor

Bibliografia

1. *IEEE Standard for SystemVerilog – unified hardware design, specification, and verification language. Standard 1800–2017 (Revision of IEEE Standard 1800–2012)*, IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, New York 2018, ss. 1315.
2. Sutherland S., Davidmann S., Flake P., *SystemVerilog for design second edition: a guide to using SystemVerilog for hardware design and modeling*, Springer Science & Business Media, New York 2006, ss. 436.
3. Соловьев В.В., *Основы языка проектирования цифровой аппаратуры Verilog*, [Podstawy języka projektowania sprzętu cyfrowego Verilog], 2-е издание, исправленное и дополненное, Телеком, Москва 2021, ss. 284.
4. Соловьев В.В., *Язык Verilog в проектировании встраиваемых систем на FPGA* [Język Verilog w projektowaniu systemów wbudowanych na FPGA], Телеком, Москва 2020, ss. 440.
5. Соловьев В.В., *Проектирование функциональных блоков встраиваемых систем на FPGA* [Projektowanie jednostek funkcjonalnych systemów wbudowanych na układach FPGA], Телеком, Москва 2020, ss. 348.
6. Thomas D.E., *Logic Design and Verification Using SystemVerilog*, CreateSpace, Lexington 2016, ss. 384.
7. Соловьев В.В., *Временной анализ программируемых логических интегральных схем* [Analiza czasowa programowalnych logicznych układów scalonych], Телеком, Москва 2018, ss. 360.
8. Соловьев В.В., *Объектно-ориентированное программирование на языке C++* [Programowanie obiektowe w języku C++], Белорусский государственный университет информатики и радиоэлектроники, Минск 1994, ss. 162.
9. Соловьев В.В., Посредникова А.А., *Иерархический метод синтеза на программируемых логических интегральных схемах компараторов большого размера* [Hierarchiczna metoda syntezy na programowalnych logicznych układach scalonych dużych komparatorów], „Радиотехника и электроника" 2009, t. 54, nr 3, s. 354–362.

Skorowidz

W

Z

Spis tabel

Spis rysunków

Streszczenie

W książce omówiono zastosowanie popularnego języka projektowego SystemVerilog w syntezie systemów cyfrowych. Główne elementy syntaktyczne i konstrukcje języka są opisane w sposób wystarczający do ich praktycznego zastosowania. Każda konstrukcja językowa jest opatrzona przykładem. Prezentacja materiału nie jest związana z konkretną bazą sprzętową ani z konkretnym oprogramowaniem projektowym, dlatego materiał zawarty w książce może być wykorzystywany zarówno przy tworzeniu projektów na układach ASIC, jak i na FPGA.

Język SystemVerilog w pełni dziedziczy właściwości języka Verilog i dostarcza nowych możliwości opracowywania dużych i złożonych projektów na najwyższych poziomach projektowania: systemowym, abstrakcyjnym i transakcyjnym. Ponadto SystemVerilog zawiera nowe konstrukcje językowe do weryfikacji projektów, które zapewniają nowe możliwości opisu ich środowiska testowego. W rezultacie powstał język o bardzo szerokich i różnorodnych możliwościach, które trudno szczegółowo opisać w jednej książce. W proponowanej pozycji skupiono się głównie na konstrukcjach języka SystemVerilog, które są przeznaczone do syntezy, jak również na niektórych konstrukcjach językowych używanych wyłącznie do symulacji.

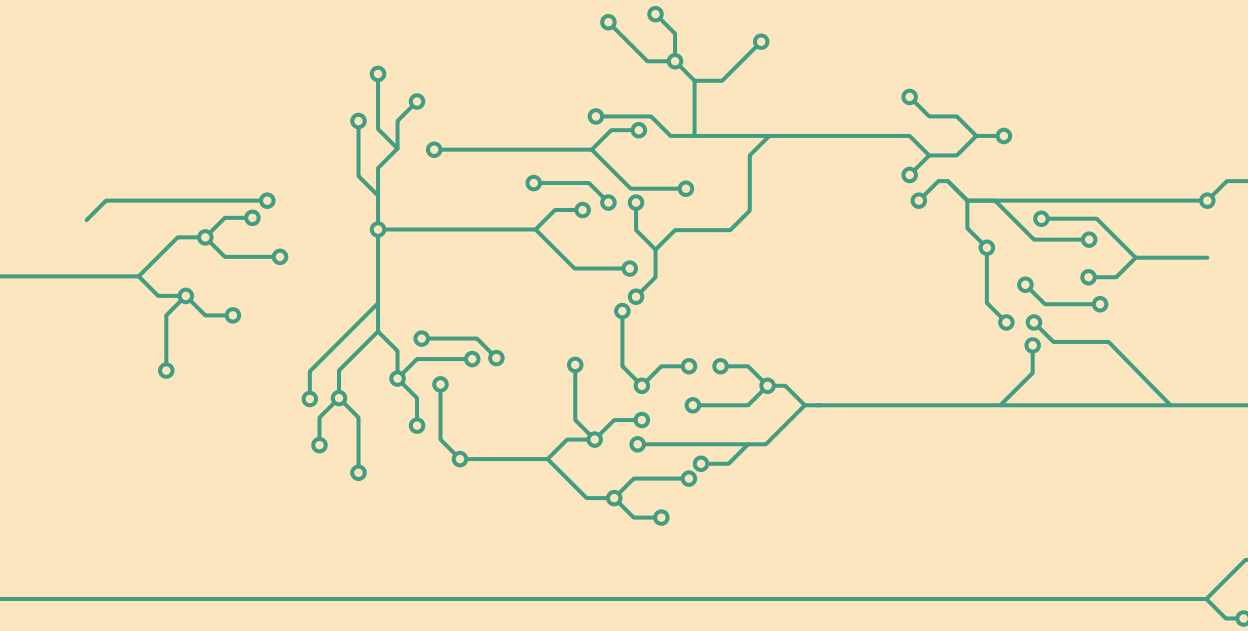
Książka przeznaczona jest dla projektantów systemów cyfrowych, nauczycieli i studentów odpowiednich specjalności, pracowników naukowych, absolwentów, może być używana jako informator, jak też jako samodzielne studium języka.

Summary

The book discusses the application of the popular hardware description language, called SystemVerilog, in the synthesis of digital systems. The main syntactic elements and language constructs are described in a way that is sufficient for practical application. Each language construct is accompanied by an example. The material presentation is not tied to any specific hardware platform or design software, making the book content suitable for projects on both ASIC and FPGA platforms.

SystemVerilog fully inherits the features of the Verilog language and introduces new capabilities for developing large and complex designs at the highest levels of design process: system-level, abstract, and transactional. Additionally, SystemVerilog includes new language constructs for design verification, offering enhanced ways to describe their testing environments. As a result, the language has very wide and diverse capabilities, which are difficult to describe in detail within a single book. This work focuses primarily on SystemVerilog constructs intended for synthesis, as well as some language constructs used exclusively for simulation.

The book is intended for digital system designers, teachers and students of relevant fields, researchers, graduates, and can be used both as a reference guide and as a standalone study of the language.



Politechnika
Białostocka