JĘZYK VERILOG W PROJEKTOWANIU SYSTEMÓW WBUDOWANYCH NA UKŁADACH FPGA

Valery Salauyou
Adam Klimowicz
Tomasz Grześ
Irena Bułatowa

Politechnika Białostocka

JĘZYK VERILOG W PROJEKTOWANIU SYSTEMÓW WBUDOWANYCH NA UKŁADACH FPGA

Valery Salauyou, Adam Klimowicz,
Tomasz Grześ, Irena Bułatowa

Politechnika Białostocka
OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2022

Recenzenci:
prof. dr hab. inż. Oleksandr Barkalov

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

Korekta językowa:
Janina Demianowicz

Skład, grafika i okładka:
Marcin Dominów

Zdjęcie na okładce:
MasterTux
https://pixabay.com/pl/illustrations/mikroprocesor-procesor-uk%c5%82ad-edytor-3036187/
muadek
https://pixabay.com/pl/photos/szk%c5%82o-scifi-fiolet-ultrafiolet-t%c5%82o-3389935/

ISBN 978-83-67185-07-3
ISBN 978-83-67185-08-0 (eBook)
DOI: 10.24427/978-83-67185-08-0

Druk: PPH Remigraf sp. z o.o.

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

CAD – Computer Aided Design – projektowanie wspomagane komputerowo
CMOS – Complementary Metal-Oxide Semiconductor – technologia wytwarzania układów scalonych
ECL – Emitter Coupled Logic – bipolarne cyfrowe układy scalone
HDL – Hardware Description Language – język opisu sprzętu
OVI – Open Verilog International – organizacja nadzorująca rozwój języka Verilog
PLD – Programmable Logic Device – programowalny układ logiczny
PLI – Programming Language Interface – interfejs języka programowania
RTL – Register Transfer Level – poziom przesłań międzyrejestrowych
SDF – Standard Delay Format – format standardowych opóźnień
UDP – User Defined Primitives – prymitywy zdefiniowane przez użytkownika
VCD – Value Change Dump – zrzut zmiany wartości
VHDL – Very High Speed Integrated Circuits Hardware Description Language – popularny język opisu sprzętu
VLSI – Very Large Scale of Integration – wielka skala integracji układów scalonych

Przedmowa

Programowalne układy logiczne FPGA (*Field Programmable Gate Arrays*) są szeroko stosowane do projektowania złożonych urządzeń cyfrowych i wbudowanych systemów sterowania. Dynamiczny rozwój technologii półprzewodnikowej pozwolił na umieszczenie w strukturach FPGA dużych zasobów logicznych oraz bogatego wyposażenia funkcjonalnego. Ponadto, wysoka wydajność i szybkość działania układów programowalnych, a także ich duża elastyczność i uniwersalność spowodowały, że układy FPGA stały się bardzo atrakcyjną bazą do projektowania wysoko wydajnych, złożonych systemów wbudowanych, a w szczególności systemów działających w warunkach czasu rzeczywistego.

Zwiększajaca się złożoność projektowanych systemów cyfrowych przyczyniła się do rozwoju i rozpowszechnienia komputerowych narzędzi zautomatyzowanego projektowania oraz języków opisu sprzętu HDL (*Hardware Description Language*). Dzięki takim narzędziom, proces tworzenia i wdrażania systemów cyfrowych stał się znacznie prostszy i szybszy. Rola projektanta w takim procesie sprowadza się do wykonania opisu projektu w języku HDL na jednym z wybranych poziomów abstrakcji, a pozostałe żmudne, pracochłonne etapy syntezy i implementacji układu w strukturze FPGA realizowane są automatycznie przez system projektowania.

W niniejszym skrypcie przedstawiono podstawy jednego z najbardziej rozpowszechnionych języków opisu sprzętu – Verilog. Język Verilog zyskał sobie ogromną popularność nie tylko ze względu na prostą, czytelną składnię, która przypomina język C, ale także z powodu jego licznych możliwości funkcjonalnych oraz hierarchicznej struktury projektu, co pozwala z łatwością opisywać nawet najbardziej skomplikowane systemy cyfrowe. Verilog, obok języka VHDL, dostępny jest w większości systemów zautomatyzowanego projektowania jako narzędzie do opisu projektów w celu ich implementacji oraz symulacji. Skrypt powstał z myślą o tym, aby w czytelny i zrozumiały sposób przybliżyć najważniejsze właściwości i możliwości języka Verilog jak najszerszemu gronu odbiorców zainteresowanych tematyką projektowania systemów cyfrowych na podstawie układów FPGA.

Wstęp

W ostatnich latach jest zauważalny dynamiczny rozwój i popularyzacja urządzeń elektronicznych w życiu codziennym oraz wszystkich sferach działalności ludzkiej. „Ogromna armia" inżynierów pracuje nad stworzeniem nowych i coraz bardziej doskonałych urządzeń elektronicznych. Dawno minęły czasy, gdy podstawowe schematy złożonych urządzeń cyfrowych były kreślone ręcznie lub za pomocą edytorów graficznych. Proces projektowania został znacznie odmieniony w wyniku pojawienia się komputerowych narzędzi oraz języków projektowania.

Dzisiaj stworzenie jakiegokolwiek urządzenia elektronicznego bardziej przypomina napisanie programu; proces ten składa się z bardzo podobnych etapów: stworzenie kodu projektu w języku HDL (odpowiada napisaniu kodu programu), kompilacja projektu (kompilacja programu), symulacja projektu za pomocą specjalnego programu nazywanego *symulatorem* (debugowanie programu), odwzorowanie projektu w strukturę programowalną FPGA (nawiązuje do stworzenia wykonywalnego kodu programu), fizyczna symulacja projektu na płytkach prototypowych (odnosi się do testowania programu w warunkach rzeczywistych), realizacja projektu w postaci urządzenia elektronicznego (przekazanie programu zleceniodawcy).

Obecnie można zauważyć ogromny wzrost popularności języków opisu sprzętu HDL wysokiego poziomu. Jednym z takich języków jest Verilog. Język Verilog został stworzony w środowisku projektantów systemów cyfrowych, dlatego szybko zdobył popularność wśród inżynierów-praktyków. Niedługo potem język Verilog stał się głównym konkurentem języka VHDL. Język VHDL został opracowany na zlecenie ministerstwa obrony USA, które we wszelki możliwy sposób wspierało jego rozpowszechnienie. Verilog jest bliższy sprzętowi, niż język VHDL, więc pozwala tworzyć bardziej efektywne projekty z punktu widzenia szybkości działania, pobieranej mocy i zajmowanego przez projekt miejsca w strukturze układu FPGA.

Jednakże szerokiemu rozpowszechnieniu się języka Verilog pośród projektantów w głównym stopniu przeszkadza brak literatury odnośnie danego języka. Liczba polskojęzycznych pozycji książkowych dostępnych obecnie jest niewielka [1, 2]. Nauka języka ze standardów [3-5] jest jednak trudna, ponieważ standardy języka Verilog przede wszystkim są przeznaczone nie dla użytkowników, ale dla twórców kompilatorów danego języka. W tym samym czasie na świecie wydano wiele wartościowych książek poświęconych językowi Verilog, zarówno dla początkujących [6-10], jak i dla zaawansowanych specjalistów [11-13]. Warto także zauważyć, że język Verilog jest obiektem ważnych badań naukowych [14-19].

W niniejszym skrypcie dość szczegółowo opisano podstawy składni oraz konstrukcje języka Verilog z punktu widzenia ich praktycznego zastosowania. Każda konstrukcja jest wsparta przykładem. Język Verilog jest przeznaczony zarówno do opisu projektu, jak i do symulacji projektu, więc nie wszystkie konstrukcje języka Verilog mogą być realizowane za pomocą sprzętu, czyli być syntezowane. Jeżeli konstrukcja jest syntezowalna, wówczas w książce prezentowana jest także jej realizacja na poziomie przesłań międzyrejestrowych (*Register Transfer Level* – RTL). Oprócz tego, bezpośrednio w tekście skryptu zaproponowano dużą liczbę zadań, których wykonanie umożliwia dokładniejsze poznanie języka. Niektóre zadania są poświęcone badaniom cech wykorzystywanego przez czytelnika środowiska projektowego. Ich wykonanie pozwala nabyć praktyczną wiedzę o możliwościach konkretnego kompilatora. W skrypcie zawarto także wiele uwag. Kierują one uwagę czytelnika na poszczególne cechy języka Verilog, ich realizację w kompilatorach i inne ważne spostrzeżenia. Wszystkie uwagi należy dokładnie przeczytać i rozważyć.

Zawartość materiału skryptu nie jest ograniczona do jednego konkretnego środowiska projektowania czy układów FPGA jednej wybranej firmy. Zagadnienia przedstawione w tej książce mogą być wykorzystane do opracowywania projektów za pomocą dowolnego pakietu zautomatyzowanego projektowania systemów cyfrowych, ponieważ większość z nich umożliwia opis projektów w języku Verilog. Do demonstracji rezultatów realizacji przykładów wybrano pakiet Quartus Prime Lite Edition wersji 18.1 firmy Intel, jednakże czytelnik może posługiwać się dowolnym pakietem projektowania.

Książka posiada następującą strukturę. W rozdziale 1 przedstawiono krótkie wprowadzenie do języka Verilog, gdzie na prostym przykładzie zaprezentowano proces opracowywania projektu na bazie języka Verilog, od opisu projektu do jego symulacji, a także pokazano możliwości języka Verilog przy opracowywaniu skomplikowanych projektów hierarchicznych. Następnie przedstawiono podstawowe elementy języka. W ten sposób przeczytanie tylko rozdziału 1 umożliwi czytelnikowi tworzenie prostych projektów.

Głównymi konstrukcjami języka Verilog są moduły i im poświęcono rozdział 2. Wyjaśniono i zilustrowano sposoby opisu, budowę oraz tworzenie instancji modułów. Często przy realizacji projektu można wykorzystać gotowe rozwiązania, takie jak prymitywy czy moduły biblioteczne, które opisano w rozdziale 3. Każdy język posiada swoje typy danych, na których może operować. W przypadku języka Verilog, typy danych mają swoje specyficzne cechy, które przedstawiono w rozdziale 4. W rozdziale 5 omówiono podstawowe operatory języka projektowania. Szczególną rolę w języku Verilog ma operator przypisania ciągłego, któremu poświęcono rozdział 6. Funkcjonowanie projektu w języku Verilog jest opisywane w postaci zbioru współzależnych procesów, które reprezentowane są przez bloki proceduralne omówione w rozdziale 7. Podczas symulacji ważną rolę pełni czas pracy poszczególnych części projektu oraz czas pracy projektu jako całości. Operatory zarządzające czasem projektu wyjaśniono w rozdziale 8.

Do stworzenia algorytmu funkcjonowania projektu w czasie służą operatory przypisania proceduralnego przedstawione w rozdziale 9 oraz operatory programowania strukturalnego omówione w rozdziale 10. Ważną rolę w języku Verilog posiadają także atrybuty, które opisano w rozdziale 11. Za pomocą atrybutów użytkownik może przekazać wskazówki kompilatorowi, w jaki sposób wykonywać kompilację poszczególnych fragmentów kodu. Bloki generacji, omówienie których można znaleźć w rozdziale 12, pozwalają skrócić opis powtarzających się lub nieznacznie różniących się od siebie fragmentów kodu. W rozdziale 13 zaprezentowano procedury i funkcje języka Verilog; są one elementami programowania strukturalnego i pozwalają w wielu przypadkach zwiększyć czytelność i przejrzystość kodu. Procedury i funkcje systemowe przedstawiono w rozdziale 14, a dyrektywy kompilatora – w rozdziale 15. Bloki specyfikacji, opisywane w rozdziale 16, umożliwiają określenie wartości parametrów czasowych wykorzystywanych przy pracy symulatora i analizatora czasowego. W rozdziale 17 poruszono zagadnienia związane z tworzeniem bloków konfiguracji, które pozwalają wskazać biblioteki wykorzystywane do realizacji poszczególnych instancji modułów projektu. Syntezowalne konstrukcje języka Verilog wymieniono w rozdziale 18.

Skrypt ten jest przeznaczony dla studentów kierunków technicznych jako pomoc w nauce przedmiotów związanych z projektowaniem układów i systemów cyfrowych implementowanych w strukturach FPGA. Z uwagi na fakt, iż opis podstawowych elementów konstrukcji języka jest dość szczegółowy i poparty licznymi przykładami, książka może służyć jako podręcznik do nauki języka Verilog. Materiały zawarte w skrypcie mogą być używane przez wykładowców, doktorantów czy studentów do przygotowywania się do zajęć praktycznych, egzaminów, prac semestralnych i dyplomowych. Książka może być przydatna także dla inżynierów, którzy zawodowo zajmują się projektowaniem układów i systemów cyfrowych, jako źródło szczegółowych informacji o języku Verilog.

1. Wprowadzenie do języka Verilog

1.1. Historia języka Verilog

Język Verilog został opracowany przez Phila Moorby'ego oraz Prabhu Goela w latach 1983-1984 na zlecenie firmy Automated Integrated Design System (od 1985 r. znana pod nazwą Gateway Design Automation). Początkowo język ten wyłącznie był przeznaczony do symulacji urządzeń cyfrowych. W 1984 r. firma Gateway rozpoczęła sprzedaż programu do symulacji pod nazwą Verilog XL. Z biegiem czasu stworzony symulator Verilog XL stawał się coraz bardziej popularny wśród projektantów układów cyfrowych.

W 1987 r. firma Synopsys rozpoczęła wykorzystywanie języka Verilog jako języka opisu sprzętu przy syntezie projektów systemów cyfrowych. Od tego momentu zaczął być on wykorzystywany także do realizacji zadań syntezy logicznej. W 1989 r. firma Gateway została zakupiona przez Cadence – dużego wytwórcę systemów komputerowego wspomagania projektowania (CAD – *Computer Aided Design*) urządzeń elektronicznych. Zgodnie ze swoją polityką, firma Cadence podzieliła język Verilog i symulator Verilog XL na dwa niezależne produkty. W rezultacie kilka firm kupiło prawa do wykorzystywania języka Verilog.

W 1990 r. firma Cadence pozwoliła na publiczne wykorzystywanie języka Verilog. Od tego momentu dowolna osoba mogła używać tego języka bez żadnych licencji czy pozwoleń. Głównym powodem tego kroku było podniesienie konkurencyjności języka w stosunku do VHDL. W tym samym roku została założona organizacja Open Verilog International (OVI), której celem jest kontrola specyfikacji języka Verilog, ponieważ każdy twórca kompilatora tego języka mógł po swojemu go interpretować. Grupa OVI dokonała kilku ulepszeń, co w 1993 r. zaowocowało wydaniem nowej specyfikacji.

W 1995 r. został przyjęty pierwszy standard Verilog – IEEE 1364-1995 [3]. Od tego momentu Verilog, podobnie jak VHDL, stał się pełnoprawnym, otwartym językiem projektowania. W 2001 r. został wydany kolejny standard języka Verilog – IEEE 1364-2001 [4], który wprowadził szereg znaczących ulepszeń języka Verilog i został nazwany Verilog-2001.

W 2000 r. rozpoczęto prace pod kierownictwem grupy Accelera (konsorcjum firm, zajmujących się projektowaniem układów cyfrowych i rozwojem CAD) nad językiem System Verilog.

W 2005 r. zostały opublikowane od razu dwa standardy, dotyczące języka Verilog, Verilog-2005 (IEEE 1364-2005) oraz System Verilog-2005 (IEEE 1800-2005). W standardzie Verilog-2005 wprowadzono niewielkie zmiany w stosunku do standardu Verilog-2001. Pomysły na rozwój języka Verilog zostały rozwinięte w nowy język System Verilog, dla którego były opracowane standardy IEEE 1800-2009 oraz IEEE 1800-2012.

Obecnie większość twórców oprogramowania do projektowania i symulacji aparatury cyfrowej wspiera standard języka Verilog-2001 (Aldec, Altera, Axiom Design Automation, Cadence Design Systems, Dolphin Integration, Fintronic, Frontline, Huada Empyrean Software, Intel, Mentor Graphics, Simucad Design Automation, Sugawara Systems, SynaptiCAD, Synopsys, Tachyon Design Automation, WinterLogic, Xilinx i inne). Dlatego niniejsza książka skupia się przede wszystkim na wersji Verilog-2001 zgodnej ze standardem IEEE 1364-2001 [4]. Do demonstracji przykładów opisów urządzeń cyfrowych został użyty pakiet Quartus Prime Lite Edition wersji 18.1 firmy Intel.

1.2. Pierwszy projekt w języku Verilog

1.2.1. Opis projektu

Tradycyjnie, przy nauce języków programowania, pierwszy program polega na wyświetleniu na ekranie słów „Hello world!". Verilog również pozwala na wyświetlanie na ekranie komunikatów, jednakże jest on językiem projektowania, a nie programowania. Dlatego w roli pierwszego projektu napisanego w tym języku zostanie opisany jednobitowy sumator. Funkcjonowanie jednobitowego sumatora można zaprezentować w postaci *tablicy prawdy*, przedstawionej w tabeli 1.1, gdzie *a* i *b* – są pojedynczymi bitami słów *A* i *B*; *cin* – bit przeniesienia z poprzedniej pozycji; *s* – bit sumy; *cout* – przeniesienie na kolejną pozycję.

TABELA 1.1. Tablica prawdy sumatora jednobitowego

Wejścia			**Wyjścia**	
cin	*a*	*b*	*cout*	*s*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

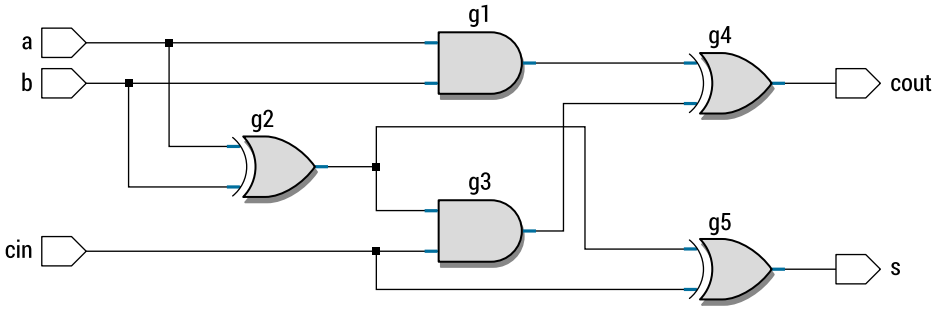
Na podstawie tabeli 1.1 można stworzyć następujące równania logiczne funkcji wyjściowych:

$$s = \overline{cin} \cdot \overline{a} \cdot b + \overline{cin} \cdot a \cdot \overline{b} + cin \cdot \overline{a} \cdot \overline{b} + cin \cdot a \cdot b \tag{1.1}$$
$$cout = \overline{cin} \cdot a \cdot b + cin \cdot \overline{a} \cdot b + cin \cdot \ a \cdot \overline{b} + cin \cdot a \cdot b$$

które w wyniku minimalizacji i przekształceń logicznych można zaprezentować w następującej postaci (sposoby przekształceń i dowód można znaleźć w dowolnym podręczniku dotyczącym podstaw techniki cyfrowej):

$$s = a \oplus b \oplus cin \tag{1.2}$$
$$cout = a \cdot b + (a \oplus b) \cdot cin$$

gdzie znak $\oplus$ oznacza operację logicznej alternatywy wykluczającej (XOR). Przewagą równań (1.2) nad równaniami (1.1) jest to, że mogą być realizowane na bramkach z dwoma wejściami (rysunek 1.1).



RYS. 1.1. Schemat jednobitowego sumatora na poziomie bramek logicznych

Opis przykładowego sumatora jednobitowego w języku Verilog przedstawiono na listingu 1.1.

LISTING 1.1. Strukturalny opis jednobitowego sumatora przedstawionego na rysunku 1.1

```
/*****************************************************************
Kod projektu jednobitowego sumatora zrealizowany w postaci opisu modelu strukturalnego na poziomie bramek logicznych
*****************************************************************/
module add_1_1 (input cin, a, b,          // opis portów wejściowych
        output s, cout);                  // opis portów wyjściowych

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

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

Podstawową konstrukcją języka Verilog jest *moduł*, który rozpoczyna się słowem kluczowym **module** i kończy słowem **endmodule**. Po słowie kluczowym **module** następuje nazwa projektu *add_1_1*, po której w nawiasie wymienione są porty modułu: wejściowe (**input**) i wyjściowe (**output**). Opis portów modułu w języku Verilog zawsze zakończony jest średnikiem.

Jeżeli do opisu projektu wymagane są dodatkowe zmienne, powinny być one zainicjalizowane przed ich pierwszym wykorzystaniem. W podanym przykładzie są to *węzły* (**wire**), odpowiadające wyjściom poszczególnych bramek: *g1_o, g2_o* i *g3_o*. Właściwie pokazany opis projektu składa się z opisu każdej bramki z rysunku 1.1. Bramki logiczne odpowiadają prymitywom języka Verilog, więc mogą być stosowane bez wcześniejszej inicjalizacji czy opisu.

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

Uwaga. Skalarne (jednobitowe) wewnętrzne połączenia w języku Verilog nie muszą być inicjalizowane, kompilator robi to automatycznie.

Zgodnie z przytoczoną uwagą, linię programu z listingu 1.1 opisującą inicjalizację zmiennych wewnętrznych *g1_o, g2_o* i *g3_o*, można pominąć.

Zadania.

1) Wykonaj w języku Verilog opis jednobitowego sumatora zgodnie z równaniami (1.1).
2) Wykonaj minimalizację równań (1.1) (np. za pomocą *tablicy Karnaugha*) i na podstawie otrzymanych równań opisz jednobitowy sumator w języku Verilog.

Listing 1.1 przedstawia strukturalny opis projektu na poziomie bramek logicznych. W podobny sposób można opisać dowolny projekt, jednak nie zawsze musi on być opisywany na poziomie bramek. Opis strukturalny dokładnie odzwierciedla

graficzną prezentację projektu, przez co nie widać zalet wykorzystywania języka projektowania w stosunku do edytora graficznego. Dlatego też powyższy styl stosuje się bardzo rzadko do opisu projektu.

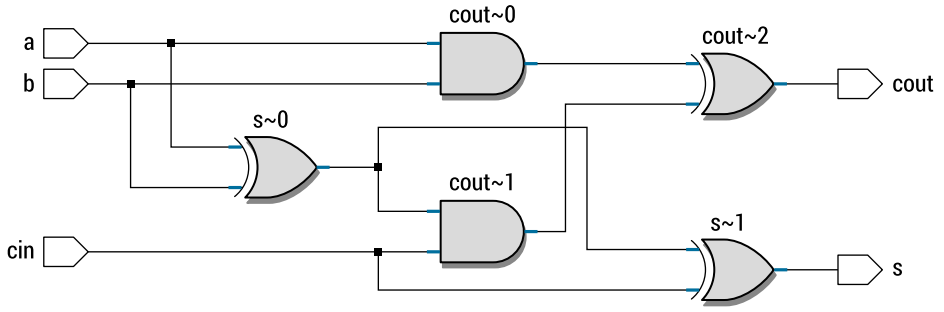
Ten sam projekt można również opisać za pomocą równań logicznych (1.2).

LISTING 1.2. Opis jednobitowego sumatora za pomocą równoległych funkcji celu

```
/*****************************************************************
Kod projektu jednobitowego sumatora zrealizowany jako opis równań logicznych
*****************************************************************/
module add_1_2 (input cin, a, b,          // opis portów wejściowych
        output s, cout);                  // opis portów wyjściowych

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

Na listingu 1.2 do opisu projektu jednobitowego sumatora wykorzystano operatory przypisania ciągłego **assign**. Owe operatory zawsze wykonywane są równolegle, niezależnie od miejsca ich umieszczenia w kodzie projektu. Z prawej strony znaku równości przedstawiono wyrażenia wykorzystujące operacje logiczne opisane znakami: | – LUB (OR), & – I (AND) oraz ^ – alternatywa wykluczająca (XOR). Warto dodać, że negację oznacza się za pomocą znaku ~. Realizację modułu *add_1_2* pokazano na rysunku 1.2. Można zauważyć, że układy z rysunku 1.1 oraz rysunku 1.2 są jednakowe, chociaż zostały przedstawione za pomocą różnych stylów opisu.

Uwaga. Styl opisu projektu zaprezentowany na listingu 1.2 jest najczęściej wykorzystywany do opisu układów kombinacyjnych na bazie *równań logicznych* (*boolowskich*).

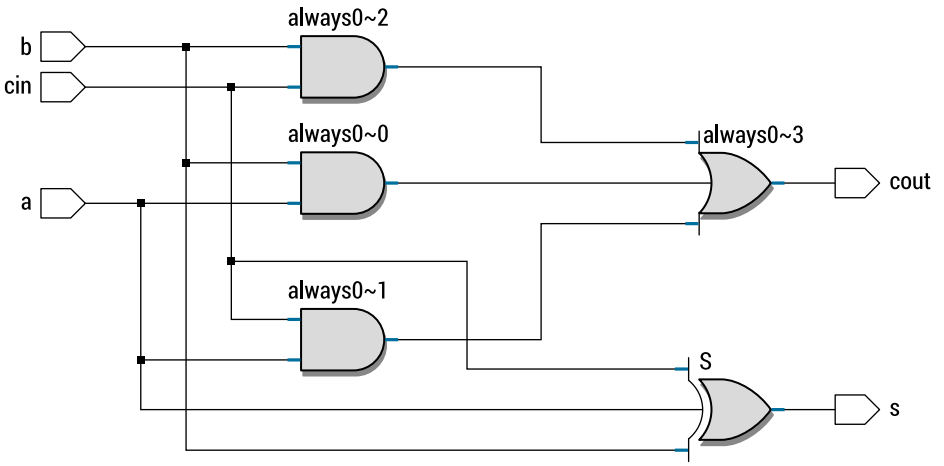


RYS. 1.2. Realizacja modułu *add_1_2* z listingu 1.2: przykład opisu sumatora jednobitowego za pomocą równań logicznych

Język Verilog pozwala także opisywać projekty na poziomie behawioralnym (zachowania), polegającym na przedstawieniu algorytmu funkcjonowania modułu za pomocą *operatorów proceduralnych*. Aby opisać przykładowy projekt jednobitowego sumatora na poziomie zachowań, należy ukazać reguły jego funkcjonowania. Na przykład funkcja przeniesienia *cout* będzie równa 1, jeżeli na wejściach sumatora jednocześnie znajdą się nie mniej niż dwie wartości 1. W tej sytuacji opis jednobitowego sumatora można zaprezentować następująco:

LISTING 1.3. Opis jednobitowego sumatora za pomocą operatorów proceduralnych

```
/*****************************************************************
Kod projektu jednobitowego sumatora zrealizowany jako opis behawioralny
*****************************************************************/
module add_1_3 (input cin, a, b,                    // opis portów wejściowych
        output reg s, cout);                        // opis portów wyjściowych

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



RYS. 1.3. Realizacja modułu *add_1_3* na podstawie listingu 1.3: przykład opisu jednobitowego sumatora za pomocą operatorów proceduralnych

Na listingu 1.3 wykorzystano operator proceduralny **if-else** oraz operator proceduralny blokowanego przypisania (=). Prócz tego, dla funkcji wyjściowych *s* i *cout* został wykorzystany typ zmiennych **reg**. Zgodnie ze standardem języka Verilog zmienne zapisane z lewej strony znaku równości w operatorach proceduralnych przypisania powinny mieć typ **reg**. Realizacja modułu *add_1_3* została przedstawiona na rysunku 1.3.

Jeżeli jakiś element projektu zostanie opisany, może być wykorzystywany wielokrotnie. Na przykład jednobitowy sumator może zostać wykorzystany do budowy sumatora 4-bitowego.

LISTING 1.4. Opis sumatora 4-bitowego

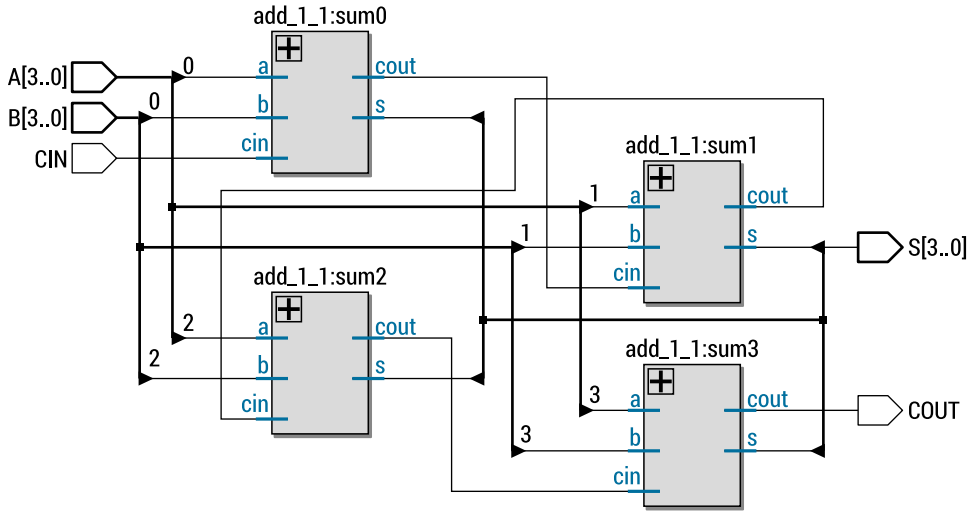
```
/**************************************************************
Kod projektu 4-bitowego sumatora zrealizowanego za pomocą czterech instancji stworzonego wcześniej sumatora jednobitowego
**************************************************************/
module add_4 (input CIN, input wire [3:0] A, B,   // opis wejścia
        output wire [3:0] S, output COUT);        // opis wyjścia

        wire [2:0] C;                             // opis zmiennych wewnętrznych

        /* opis czterech instancji sumatorów jednobitowych */
        add_1_1 sum0(CIN,A[0],B[0],S[0],C[0]);
        add_1_1 sum1(C[0],A[1],B[1],S[1],C[1]);
        add_1_1 sum2(C[1],A[2],B[2],S[2],C[2]);
        add_1_1 sum3(C[2],A[3],B[3],S[3],COUT);
endmodule
```

Na listingu 1.4 wykorzystano cztery instancje modułu sumatora jednobitowego *add_1_1* o nazwach *sum0, sum1, sum2* i *sum3*. Do przekazu wartości sygnału przeniesienia pomiędzy poszczególnymi modułami opisano zmienne *C*[*0*], *C*[*1*], *C*[*2*] typu **wire**. Realizacja modułu *add_4* została przedstawiona na rysunku 1.4.

W podobny sposób w języku Verilog opisywane są skomplikowane projekty hierarchiczne.

Podsumowując, język Verilog umożliwia opis projektu na następujących poziomach:

- tranzystorów;
- bramek logicznych;
- równań logicznych;
- przesłań międzyrejestrowych (*Register Transfer Level* – RTL);
- zachowania (*behavioral*);
- strukturalnym (systemowym).



RYS. 1.4. Realizacja modułu *add_4* na podstawie listingu 1.4: hierarchiczna struktura sumatora 4-bitowego zbudowanego z czterech instancji sumatora jednobitowego

Opis projektów na poziomie tranzystorów wykorzystuje się bardzo rzadko. Na przykład taki styl opisu może być zastosowany przy projektowaniu nowych elementów bibliotek dla dedykowanych układów scalonych. Przykład opisu projektu na poziomie bramek logicznych został przedstawiony na listingu 1.1. Do opisu projektu na poziomie równań logicznych wystarczy przedstawić projekt w postaci układu równań boolowskich. W tym przypadku można stosować dodatkowe zmienne przejściowe. Przykład opisu projektu na poziomie równań logicznych przedstawiono na listingu 1.2. Opis na poziomie przesłań międzyrejestrowych (RTL) różni się od opisu projektu na poziomie bramek jedynie tym, że zamiast bramek wykorzystuje się funkcjonalne elementy poziomu RTL takie, jak: rejestry, liczniki, sumatory, dekodery, multipleksery i inne. Do opisu projektów na poziomie zachowań wykorzystuje się operatory proceduralne języka Verilog. Przykład opisu jednobitowego sumatora na poziomie zachowań został przedstawiony na listingu 1.3. Opis projektu na poziomie systemowym jest podobny do opisu na poziomie RTL, z tą różnicą, że zamiast elementów funkcjonalnych RTL wykorzystuje się bloki funkcjonalne na poziomie systemowym (procesory, pamięć, magistrale, urządzenia sterujące, urządzenia wejścia/wyjścia i inne).

1.2.2. Symulacja projektu

Każdy opisany projekt należy zweryfikować, sprawdzić poprawność jego działania, czyli wykonać symulację projektu. Wcześniej, zanim opracowano język Verilog, do symulacji wykorzystywano edytory graficzne oraz specjalne języki, zupełnie

różne od obecnie stosowanych języków opisu sprzętu. Ponieważ Verilog od początku był przeznaczony do symulacji, wykorzystuje się go także w tym celu. Moduł do symulacji jednobitowego sumatora został przedstawiony na listingu 1.5.

LISTING 1.5. Moduł do symulacji jednobitowego sumatora

```
`timescale 1ns/1ps                  // określenie jednostki czasu symulacji
module test_add_1();                // początek modułu symulacji
        reg tcin, ta, tb;           // deklaracja zmiennych dla wektorów wejścia
        wire ts, tcout;             // deklaracja zmiennych dla funkcji wyjścia

        add_1_1 sum(tcin,ta,tb,ts,tcout); // stworzenie instancji modułu

        initial begin: test                 // operator proceduralny o nazwie test
                integer i;                  // inicjacja zmiennej lokalnej
                $display(„Rezultat symulacji sumatora jednobitowego:");
                $timeformat(–9,1,"ns",8);   // określenie 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 // początek operatora pętli
                        #10;                // operator opóźnienia o 10 jednostek czasu
                        {tcin,ta,tb} = i;   // formowanie wektora wejść
                end                         // zakończenie operatora pętli
        end                                 // zakończenie proceduralnego operatora test
endmodule                                   // zakończenie modułu
```

Warto tu wyjaśnić poszczególne cechy przytoczonego opisu. W pierwszej linii kodu za pomocą dyrektywy systemowej **`timescale** określa się: znaczenie jednostek czasu 1ns (nanosekunda) oraz dokładność pomiaru czasu 1ps (pikosekunda). Ponieważ moduł do symulacji o nazwie *test_add_1* nie otrzymuje żadnych sygnałów z zewnątrz oraz nie zwraca żadnych sygnałów na zewnątrz, moduł ten nie posiada portów. Następnie inicjowane są zmienne dla formowania wektorów wejściowych *tcin, ta, tb* oraz dla rezultatów symulacji *ts, tcout*. W kolejnym punkcie został stworzony model projektu w postaci instancji modułu *add_1_1* o nazwie *sum*.

Emulacja modelu realizowana jest za pomocą bloku proceduralnego **initial** o nazwie *test*, który zostaje wykonany tylko raz. Tutaj definiuje się zmienną lokalną *i* typu całkowitoliczbowego oraz wywoływane są funkcje systemowe: **$display** do wyświetlenia komunikatu na ekran, **$timeformat** do określenia formatu prezentowania czasu i **$monitor** do śledzenia wskazanych zmiennych i ich prezentacji w przypadku zmian chociażby jednej z monitorowanych wartości (z wyjątkiem

zmiennej czasu). Następnie występuje operator proceduralny **for**, w którego bloku wywoływane jest opóźnienie o 10 jednostek czasu. Na koniec zostaje sformowany wektor wejść dla kolejnej iteracji.

Ponieważ w języku Verilog bloki proceduralne i instancje modułów wykonywane są równolegle, to uformowane w bloku **initial** wartości wektora wejść (zmienne *tcin, ta* oraz *tb*) automatycznie będą przekazane na wejście instancji modułu *sum*. Czas pracy instancji *sum* nie jest określony, dlatego przyjmuje się wartość zerową. W taki sposób wartości zmiennych wyjściowych *ts* oraz *tcout* będą określane bez opóźnień, a funkcja systemowa **$monitor** wyświetli te wartości oraz wartości zmiennych wejściowych na ekranie.

Zadanie. Porównaj listingi opisów sumatora jednobitowego z listingiem 1.5. Jak sądzisz, co zajmuje projektantowi więcej czasu – opis projektu czy symulacja projektu?

Wynik symulacji sumatora jednobitowego przedstawiono na rysunku 1.5.

```
#     Rezultat 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. 1.5. Wynik symulacji sumatora jednobitowego

Analizując rezultaty symulacji, projektant może wysnuć wniosek o poprawności działania projektu lub w przeciwnym wypadku – o konieczności wprowadzenia odpowiednich zmian do opisu projektu.

Uwaga. Analiza wyników symulacji może także zostać wykonana automatycznie przy pomocy języka Verilog, a na ekranie mogą zostać pokazane jedynie rezultaty analizy.

1.3. Podstawowe elementy języka Verilog

1.3.1. Słowa kluczowe

Wstępne zaznajomienie się z listą słów kluczowych jest dobrą praktyką podczas nauki dowolnych języków programowania i projektowania. Znajomość słów kluczowych jest niezbędna chociażby dlatego, że nie można ich używać jako etykiet zmiennych, modułów i innych. W tabeli 1.2 oraz 1.3 zawarto słowa kluczowe ze standardów Verilog-1995

oraz Verilog-2001 wykorzystywane w pakiecie Quartus firmy Intel. Jeżeli używasz innego środowiska dla swoich projektów, warto poznać słowa kluczowe dla zaimplementowanej wersji języka Verilog. Zauważmy, że dokładna analiza słów kluczowych daje pewne informacje o nowym języku oraz rozbudza ciekawość i powoduje pytania, na które odpowiedzi można znaleźć przy dalszym poznawaniu danego języka.

TABELA 1.2. Słowa kluczowe języka Verilog (wersja z 1995 r.)

always	ifnone	rpmos
and	initial	rtran
assign	inout	rtranif0
begin	input	rtranif1
buf	integer	scalared
bufif0	join	small
bufif1	large	specify
case	macromodule	specparam
casex	medium	strong0
casez	module	strong1
cmos	nand	supply0
deassign	negedge	supply1
default	nmos	table
defparam	nor	task
disable	not	time
edge	notif0	tran
else	notif1	tranif0
end	or	tranif1
endcase	output	tri
endmodule	parameter	tri0
endfunction	pmos	tri1
endprimitive	posedge	triand
endspecify	primitive	trior
endtable	pull0	trireg
endtask	pull1	vectored
event	pullup	wait
for	pulldown	wand
force	rcmos	weak0
forever	real	weak1

fork	realtime	while
function	reg	wire
highz0	release	wor
highz1	repeat	xnor
if	rnmos	xor

TABELA 1.3. Słowa kluczowe języka Verilog w wersji z 2001 r. (niezawarte w wersji z 1995 r.)

automatic	incdir	pulsestyle_ondetect
cell	include	pulsestyle_onevent
config	instance	signed
endconfig	liblist	showcancelled
endgenerate	library	unsigned
generate	localparam	use
genvar	noshowcancelled	

Zadanie. Zapoznaj się ze słowami kluczowymi kompilatora języka Verilog wykorzystywanego w Twoim środowisku projektowym.

1.3.2. Identyfikatory

Identyfikatory języka Verilog wykorzystywane są jako nazwy (etykiety) zmiennych, modułów, funkcji, instancji modułów, instancji prymitywów itp. Identyfikatory mogą składać się z wielkich (A-Z) i małych (a-z) liter alfabetu łacińskiego, cyfr (0-9), znaku podkreślenia (_) oraz znaku dolara ($). Inne znaki ASCII mogą być używane jako składowe identyfikatorów, jeżeli są poprzedzone znakiem *backslash* (\).

Identyfikatory powinny rozpoczynać się literą bądź znakiem podkreślenia, natomiast kończyć białym znakiem. Długość identyfikatora nie powinna być jednak dłuższa niż 1024 znaki. Verilog rozróżnia wielkość liter, dlatego etykiety abc, Abc, ABC – są przykładem trzech różnych identyfikatorów.

Przykłady poprawnych identyfikatorów:

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

Przykłady nieprawidłowych identyfikatorów:

1_add, 2012_project, $RESET, xor.

Zaprezentowany w przykładzie poprawny identyfikator *XOR* został zapisany wielkimi literami, natomiast identyfikator *xor* – jest nieprawidłowy, ponieważ jest słowem kluczowym **xor**, co uniemożliwia wykorzystanie go w postaci etykiety (wszystkie słowa kluczowe pisane są małymi literami).

1.3.3. Białe znaki

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

1.3.4. Komentarze

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

Uwaga. Należy uważać podczas wyłączania części kodu za pomocą bloku komentarza. Jeżeli owa część kodu także zawiera blok komentarza, wówczas wyłączenie części kodu nastąpi jedynie do pierwszej pary znaków */.

1.4. Sygnały, sieci, sterowniki

Ogólnie w języku Verilog nie ma pojęcia sygnału (mimo że ten język wykorzystywany jest do projektowania układów cyfrowych, gdzie *sygnał* jest pojęciem kluczowym). Zamiast sygnałów w języku Verilog występuje pojęcie *sieci* (*nets*). Sieci są wykorzystywane do połączeń portów (wyprowadzeń) poszczególnych komponentów projektu. Głównymi komponentami projektu są moduły, ale w postaci komponentów projektu mogą również występować prymitywy oraz funkcje i procedury, które także mogą posiadać porty.

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

1.4.1. Wartości logiczne

W języku Verilog każdy bit (pojedynczy sygnał) może przyjmować cztery wartości:
- 0 – logiczne zero bądź *fałsz*;
- 1 – logiczna jedynka albo *prawda*;
- X – dowolny znak (to znaczy, że może być zarówno 0 jak i 1) bądź wartość nieokreślona (*don't care*);
- Z – stan wysokiej impedancji, stan trzeci bądź stan przejściowy (*floating*).

Gdy stan **Z** pojawia się na wejściu lub w wyrażeniu, wówczas rezultat będzie taki sam, jak przy stanie **X**. W taki sposób różnica stanu **X** od sygnału **Z** widoczna jest jedynie dla sygnałów wyjściowych.

W języku Verilog dopuszczalne są jeszcze dwa stany logiczne, które wykorzystywane są przez oprogramowanie tylko do wewnętrznej symulacji i nie mogą być użyte w opisie projektu:
- L – częściowo nieznany stan: albo 0, albo **Z**, jednak nie 1;
- H – częściowo nieznany stan: albo 1, albo **Z**, jednak nie 0.

Jak można zauważyć, liczba różnych stanów, które mogą przyjmować sygnały w języku Verilog nie jest duża i przy rozwiązywaniu różnych zadań symulacji może nie być wystarczająca. Dlatego wielu twórców kompilatorów języka Verilog zwiększa liczbę możliwych wartości sygnałów.

Zadanie. Dowiedz się, jakie wartości logiczne może przyjmować pojedynczy sygnał języka Verilog w środowisku, w którym pracujesz.

1.4.2. Moc logiczna sygnałów

Logiczne wartości sygnałów mogą przyjmować 8 poziomów mocy: 4 są określane źródłem (sterownikiem) sygnału, 3 określają pojemność, a 1 oznacza poziom wysokiej impedancji (czyli nie posiada mocy). Poziomy logicznej mocy zostały przedstawione w tabeli 1.4. Poziom mocy o wyższym numerze oznacza większą moc logiczną sygnału.

TABELA 1.4. Poziomy mocy źródeł sygnałów

Poziom mocy	Nazwa mocy	Słowa kluczowe	Oznaczenia mnemoniczne
7	*supply drive* (sterowanie zasilaniem)	*supply0* *supply1*	*Su0* *Su1*
6	*strong drive* (silne sterowanie)	*strong0* *strong1*	*St0* *St1*
5	*pull drive* (sterowanie wyciąganiem)	*pull0* *pull1*	*Pu0* *Pu1*
4	*large capacitive* (duża pojemność)	*large*	*La0* *La1*

Poziom mocy	Nazwa mocy	Słowa kluczowe	Oznaczenia mnemoniczne
3	*weak drive* (słabe sterowanie)	*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*

Łańcuch z kilkoma źródłami sygnałów może posiadać różne poziomy mocy, które są przedstawione w postaci liczb ósemkowych i wartości logicznej, na przykład: 65X, gdzie 6 i 5 – poziomy logicznej mocy źródeł sygnałów; X – logiczna wartość nieokreślona.

1.5. Liczby

1.5.1. Reprezentacja liczb całkowitych

Do reprezentacji liczb w języku Verilog wykorzystuje się postać:

size'base value

gdzie *size* – *rozmiar*, liczba dziesiętna określająca rozmiar pola bitowego do reprezentacji wartości; *'base* – *podstawa*, symbol określający system liczbowy, w którym reprezentowana jest wartość; *value* – *wartość liczby*. Do określenia systemu liczbowego (*'base*) korzysta się z następujących znaków:

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

Zauważmy, że do reprezentacji dowolnej liczby w języku Verilog musimy określić liczbę bitów (*size*), na których daną liczbę można zapisać. Wyjątek stanowi wykorzystanie jednobitowych wartości w niektórych konstrukcjach języka. Każdy taki przypadek zostanie specjalnie opisany w niniejszej książce wraz z odpowiadającą mu konstrukcją.

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

Zadanie. Spróbuj znaleźć domyślny rozmiar (*size*) oraz podstawę (*base*) zapisu liczb w wykorzystywanym przez Ciebie kompilatorze.

Przykłady liczb:

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

Jeżeli binarna postać liczby przekracza zadeklarowany wcześniej rozmiar (*size*), wówczas starsze bity przekraczające zadany rozmiar zostają odrzucone. W sytuacji, gdy postać binarna liczby jest mniejsza niż zadeklarowany rozmiar zmiennej, wówczas uzupełnione zostają młodsze bity, a starsze będą wyzerowane. Wyjątek stanowią przykłady, gdy najstarszym bitem jest **Z** bądź **X**, wówczas pozostałe bity zostają uzupełnione odpowiednio **Z** lub **X**. Przykłady bitowej reprezentacji liczb całkowitych zaprezentowano w tabeli 1.5.

TABELA 1.5. Przykłady reprezentacji binarnej liczb całkowitych

Liczba	Reprezentacja binarna	Uwagi
12	0…01100	32 bity
'o5	0…00101	32 bity
1'b1	1	
1'b0	0	
8'Hd5	11010101	
6'hF2	110010	pominięcie starszych bitów
6'hA	001010	uzupełnienie zerami starszych bitów
8'b0	00000000	
8'b1	00000001	
8'bx	xxxxxxxx	
8'b1x	0000001x	
8'b0x	0000000x	
8'hx	xxxxxxxx	
8'hzx	zzzzxxxx	
8'hz1	zzzz0001	
8'bx1	xxxxxxx1	
8'bx0	xxxxxxx0	
8'hz	zzzzzzzzz	
8'h0z	0000zzzz	

Uwagi.

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

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

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

1.5.2. Reprezentacja liczb rzeczywistych

Do zapisu liczb rzeczywistych w języku Verilog wykorzystuje się dwa formaty: notacja dziesiętna (z kropką pomiędzy częścią całkowitą a ułamkiem) oraz notacja wykładnicza.

Format zawierający kropkę wygląda następująco:

value_i.value_f

gdzie *value_i* – część całkowita; *value_f* – część dziesiętna.

Notacja wykładnicza liczby przyjmuje następujący format:

*base***e***exponen*t lub

*base***E***exponent*

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

Uwagi.

1) Przy reprezentacji z użyciem kropki, cyfry powinny być po obydwu jej stronach.
2) W notacji wykładniczej nie może być spacji ani przed, ani po znaku **e** bądź **E**.

Przykłady zapisu liczb rzeczywistych:

0.7

$2e4 = 2 \cdot 10^4 = 20000$

$1.8E\text{-}2 = 1.8 \cdot 10^{-2} = 0{,}018$

1.6. Równoległość języka Verilog

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

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

Uwaga. Równoległe wykonywanie poszczególnych konstrukcji języka Verilog może być zrealizowane podczas syntezy.

Równoległość języków projektowania całkowicie odpowiada fizycznej naturze układów elektrycznych: przy podłączeniu do sieci wszystkie komponenty schematu pracują równocześnie, natomiast na złączach jednocześnie i równolegle zostają przekazywane sygnały elektryczne. W języku Verilog instancje prymitywów i modułów odpowiadają komponentom systemu, operatory przypisania – połączeniom, a bloki proceduralne – blokom funkcjonalnym urządzeń cyfrowych.

Oprócz tego, równoległość języka Verilog jest bardzo wygodna podczas symulacji urządzeń i została wykorzystana w przykładzie z listingu 1.5 przy opisie modułu służącego do symulacji projektu sumatora jednobitowego.

2. Moduły

2.1. Definicje modułów

Jak już wspomniano wcześniej, moduł jest podstawową konstrukcją w języku Verilog. Wszystkie modele urządzeń cyfrowych, modele poszczególnych składowych systemów cyfrowych, a także sam system, są przedstawiane w postaci modułów.

Składnia definicji nowego modułu ma następującą postać:

module *module_name*
    #(*parameter_declaration, parameter_declaration*, …)
    (*port_declaration port_name, port_name*, …,
    port_declaration port_name, port_name, …);
    module items
endmodule

W takiej postaci deklaracja portów następuje bezpośrednio, przez umieszczenie ich na liście portów modułu (w nawiasach okrągłych). Starszy format deklaracji modułu dopuszcza inicjację portów w ciele modułu. Zaprezentowano to w przykładzie poniżej:

module *module_name* (*port_name, port_name*, …);
    port_declaration port_name, port_name, …;
    port_declaration port_name, port_name, …;
    module items
endmodule

Deklaracja modułu zawsze rozpoczyna się od słowa kluczowego **module** i kończy się słowem kluczowym **endmodule**.

Uwaga. Zamiast słowa **module** można wykorzystywać **macromodule** (do oznaczania bardziej złożonych elementów systemu).

Następnie, po słowie **module** umieszczona jest jego etykieta *module_name*, określana przez użytkownika. Dalej w nawiasach okrągłych wymienione są parametry modułu. Spis parametrów rozpoczyna znak # (dla rozróżnienia spisu parametrów od spisu portów). Po spisie parametrów, także w nawiasach okrągłych, wymieniane są porty modułu. Spis portów kończy się średnikiem, po którym następuje deklaracja elementów modułu (*module items*).

Spis parametrów nie jest wymaganym elementem modułu. Jeżeli moduł nie posiada parametrów, to od razu po etykiecie następuje spis portów, który również nie jest wymaganym elementem modułu. W procesie projektowania moduły mogą być wykorzystywane do różnych celów, na przykład mogą opisywać poszczególne części układu cyfrowego, całe układy, otoczenie w którym układ będzie pracował i inne. Warto zauważyć, że moduł najwyższego poziomu w hierarchii opisujący współpracę układu cyfrowego bądź systemu z otoczeniem nie posiada portów.

Zadanie. Podaj przykład urządzeń cyfrowych i systemów, które nie mają wejść albo wyjść.

2.2. Elementy modułów

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

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

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

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

Przykład wykorzystania stylu strukturalnego do opisu jednobitowego sumatora został przedstawiony na listingu 1.1. Behawioralny (funkcjonalny) opis tego sumatora z wykorzystaniem przypisania ciągłego **assign** na bazie równań logicznych

został przedstawiony na listingu 1.2, a za pomocą operatorów proceduralnych – na listingu 1.3. Jako alternatywa, na listingu 2.1 przedstawiony został stary styl deklaracji modułu, gdzie porty modułu są deklarowane w jego ciele.

LISTING 2.1. Opis jednobitowego sumatora z wykorzystaniem starego stylu deklaracji portów

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

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

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

W przykładzie na listingu 2.1 został zmieniony jedynie opis portów modułu.

Zadanie. Porównaj opis jednobitowego sumatora na listingach 1.2 oraz 2.1. Jaki styl opisu wydaje Ci się wygodniejszy?

2.3. Deklaracja portów

Deklaracja portów modułu ma następującą składnię:

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

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

Stara notacja deklaracji portów wygląda następująco:

port_direction **signed** *range port_name*, *port_name*, …;
data_type_declarations

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

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

Jako typ przekazywanych danych (*data_type*) można wykorzystywać dowolny typ danych z języka Verilog, za wyjątkiem typu **real**. Wartość kierunku przekazywania sygnału (*port_direction*) ogranicza typy danych przekazywanych za pośrednictwem

danego portu. Na wejściu oraz portach dwukierunkowych mogą występować tylko sieciowe typy danych (*net*). Na portach wyjściowych mogą być określone dowolne typy danych z języka Verilog, za wyjątkiem typu **real**.

Uwaga. Aby przekazać za pośrednictwem portu wartość typu **real**, należy ją wpierw przekonwertować za pomocą systemowych poleceń **$realtobits** oraz **$bitstoreal**.

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

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

[*msb*:*lsb*]

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

Jeżeli specyfikacja zakresu nie jest określona, to port posiada zakres jednego bitu, a przekazywane sygnały są traktowane jako wartości skalarne (*scalar*); w przeciwnym wypadku, przekazywane sygnały traktowane są jako wartość wektorowa (*vector*).

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

Zadanie. Sprawdź maksymalny rozmiar zakresu pola bitowego portu, który jest dopuszczalny w wykorzystywanym przez Ciebie kompilatorze języka Verilog.

Rozmiary zakresu typu danych i typu portu powinny mieć jednakowe wartości. W przypadku, gdy się od siebie różnią, niektóre kompilatory przyjmują rozmiar typu danych bez ostrzeżenia czy błędu.

Uwaga. Należy uważać przy nadawaniu rozmiarów typom danych i typom portów.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator zwraca informację o błędzie, gdy rozmiary portów i danych nie są jednakowe.

Poniżej zaprezentowano przykłady deklaracji portów:

```
input a, b, enable;                 // trzy porty skalarne
input signed [15:0] m1, m2;         /* dwa 16-bitowe porty, które przekazują dane
                                    w kodzie uzupełnienia do 2; porządek wymiany –
                                    little endian */
output signed [31:0] wyn;           /* podobnie jak poprzednio, jednak jest to 32-bitowy
                                    port */
```

```
output reg signed [32:1] sum; /* 32-bitowy port, podłączone do portu wewnętrzne
                                 sygnały mogą przyjmować wartości typu reg;
                                 porządek wymiany – little-endian */
inout [0:15] mag;             /* dwukierunkowy 16-bitowy port z porządkiem
                                 wymiany big-endian */
input [15:12] adres;          /* 4-bitowy wektor adres wykorzystuje się jako
                                 wejście (msb oraz lsb mogą mieć dowolne wartości) */
parameter WORD = 32;
input [WORD-1:0] adres;       /* deklaracja portu może zawierać stałe */
parameter SIZE=4096;
input [log2(SIZE)-1:0] adr;   /* deklaracja portu może odwoływać się do stałych
                                 funkcji */
```

Zadanie. Sprawdź standardowe stałe funkcje (które zwracają wartość stałą) w wykorzystywanym przez Ciebie kompilatorze.

2.4. Instancje modułów

Zanim będzie można wykorzystywać stworzony moduł jako część systemu cyfrowego, należy stworzyć jego instancję (*instance*). Instancja modułu odpowiada w pewnym sensie wywołaniu funkcji w językach programowania. W programowaniu wywołanie funkcji powoduje uruchomienie części kodu należącej do danej funkcji, natomiast stworzenie instancji modułu odpowiada umieszczeniu w strukturze FPGA egzemplarza układu opisanego przez moduł.

Podobnie jak przy umieszczeniu układu w strukturze FPGA należy połączyć go do poszczególnych wyprowadzeń (by móc przekazywać sygnały), również przy stworzeniu instancji modułu należy wskazać sygnały przyporządkowane odpowiednim portom. Istnieją dwa sposoby wskazania połączenia sygnałów z portami instancji: za pomocą lokalizacji oraz za pomocą nazwy portu.

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

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

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

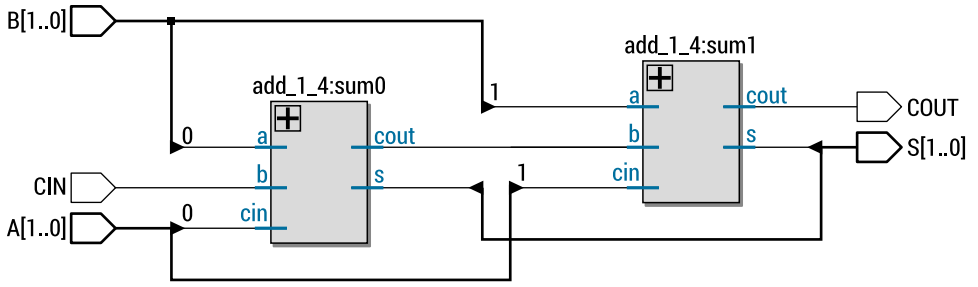
W przypadku gdy niektóre porty instancji modułu nie będą podłączone, zaznaczane jest to na liście umieszczeniem dwóch przecinków z rzędu. Technika ta nazywana jest *pominięciem sygnału*. Jeżeli wymagane jest pominięcie kilku sygnałów, wówczas w spisie sygnałów umieszcza się kilka przecinków.

Na listingu 2.2 zaprezentowano projekt sumatora dwubitowego składającego się z dwóch instancji sumatora jednobitowego.

LISTING 2.2. Opis dwubitowego sumatora składającego się z dwóch sumatorów jednobitowych

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

W danym przykładzie: *add_1_4* jest nazwą modułu sumatora jednobitowego opisanego w listingu 2.1; *sum0* oraz *sum1* są nazwami dwóch instancji modułu. Warto zauważyć, że w danym projekcie do przekazu sygnałów między dwoma instancjami sumatora jednobitowego wykorzystuje się wewnętrzny węzeł *C0*. Realizacja listingu 2.2 została przedstawiona na rysunku 2.1.



RYS. 2.1. Realizacja modułu *add_2* dwubitowego sumatora z listingu 2.2 stworzonego z dwóch sumatorów jednobitowych

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

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

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

Uwaga. W danym formacie każda nazwa portu zaczyna się od kropki.

Jeżeli do pewnego portu instancji modułu sygnał ma nie być podłączony, to nazwa tego portu zostaje pominięta w spisie portów instancji modułu. Przekazywanie sygnałów za pomocą nazwy portu wymaga większej ilości kodu, jednak nie wymaga od projektanta wiedzy o kolejności występowania portów w deklarowanym module. Prócz tego, wskazany sposób zmniejsza liczbę błędów przy pisaniu kodu projektu. Dlatego jest on wykorzystywany przy tworzeniu instancji modułów z dużą liczbą portów.

Jako przykład można rozważyć 4-bitowy rejestr na bazie prymitywu przerzutnika typu D **dff**. Nie jest znana kolejność występowania portów, jednak znane są ich nazwy: *d* – wejście danych; *q* – wyjście; *clk* – wejście sygnału synchronizacji.

LISTING 2.3. Opis 4-bitowego rejestru na podstawie prymitywu **dff** przerzutnika typu D

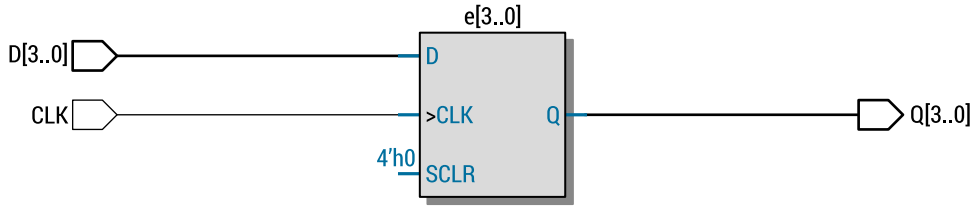
```
module reg_4_1 (input wire [3:0] D,
                input wire CLK,
                output wire [3:0] Q);
    dff e0 (.clk(CLK), .d(D[0]), .q(Q[0]));
    dff e1 (.clk(CLK), .d(D[1]), .q(Q[1]));
    dff e2 (.clk(CLK), .d(D[2]), .q(Q[2]));
    dff e3 (.clk(CLK), .d(D[3]), .q(Q[3]));
endmodule
```

W użytym przykładzie wykorzystano jedynie trzy porty prymitywu **dff**: *clk, d* oraz *q*, przy czym informacja o kolejności portów, a także nazwach innych portów nie musi być znana. Realizacja listingu 2.3 przedstawiona została na rysunku 2.2.

Uwaga. W ogólnym przypadku instancje modułów mogą być tworzone poprzez wymienienie ich nazw oddzielonych od siebie przecinkami, bez powtarzania nazwy modułu. Przykład widoczny jest na listingu 2.4.

LISTING 2.4. Alternatywny opis 4-bitowego rejestru

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



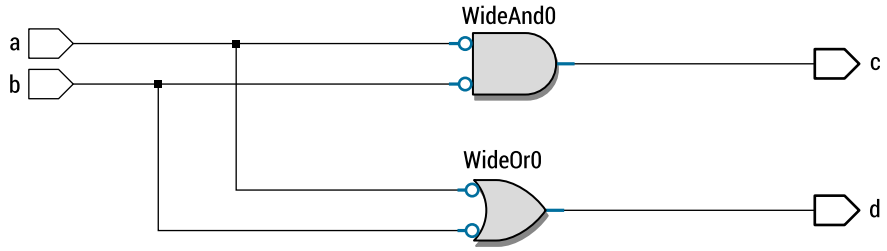
RYS. 2.2. Realizacja modułów *reg_4_1* oraz *reg_4_2* 4-bitowego rejestru z listingów 2.3 i 2.4

Uwaga. Nazwa instancji modułu nie jest wymagana, oznacza to, że można stworzyć instancję projektu bez wskazania nazwy instancji modułów, tak jak przedstawiono w przykładzie na listingu 2.5.

LISTING 2.5. Przykład stworzenia czterech inwerterów bez określania nazw ich instancji.

```
module wor_wand (input a, b,
                 output wand c,
                 output wor d);

    not (c,a), (c,b);          // sieć typu wand
    not (d,a), (d,b);          // sieć typu wor

endmodule
```

Realizacja listingu 2.5 przedstawiona została na rysunku 2.3.



RYS. 2.3. Realizacja modułu *wor_wand* z listingu 2.5: przykład połączenia wyjść inwerterów za pomocą węzłów **wor** i **wand**

Zadanie. Opisz dwa projekty 8-bitowego sumatora na bazie sumatora jednobitowego, przy czym w jednym wykorzystaj przekazywanie sygnału za pomocą lokalizacji portu, natomiast w drugim – za pomocą nazwy portu.

2.5. Parametry

Wykorzystanie parametrów w deklaracji modułu i przy tworzeniu jego instancji umożliwia w wielu przypadkach znaczące skrócenie kodu projektu, co zwiększa jego czytelność oraz zmniejsza możliwość popełnienia błędów w trakcie jego pisania.

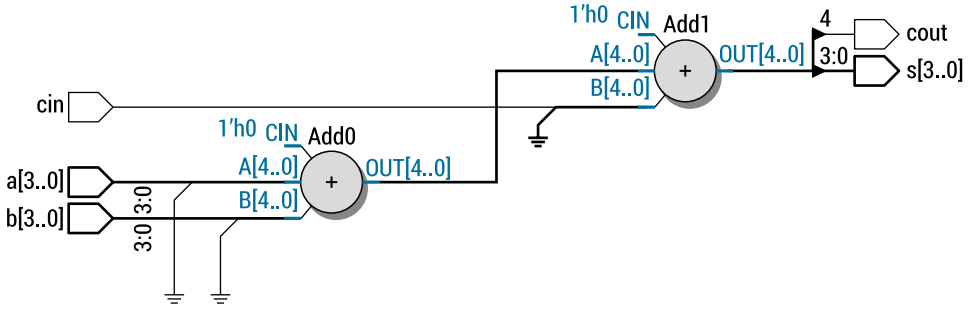
W deklaracji modułu spis parametrów rozpoczyna się znakiem # i poprzedza spis portów modułu (patrz podrozdział 2.1). Na przykład, opis parametryzowanego modułu sumatora może wyglądać następująco.

LISTING 2.6. Projekt parametryzowanego sumatora, gdzie parametr *N* określa szerokość słów wejściowych i wyjściowych

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

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

endmodule
```

W powyższym przykładzie parametr *N* wykorzystuje się do określenia zakresu wyjściowych słów *a* i *b*, oraz sumy *s*. Podczas deklaracji modułu można określać wartości domyślne poszczególnych parametrów, które są wykorzystywane, gdy przy tworzeniu instancji modułu dany parametr nie zostanie ustawiony. W wykorzystanym przykładzie domyślna wartość parametru *N* wynosi 4. Realizacja listingu 2.6 przedstawiona została na rysunku 2.4.

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



RYS. 2.4. Realizacja modułu *add_N* parametryzowanego sumatora z listingu 2.6: przykład wykorzystania funkcji arytmetycznych i operatora konkatenacji w lewej części operatora

W ogólnym przypadku moduł może mieć wiele parametrów. W takim wypadku podczas deklaracji modułu należy zamieścić spis wszystkich parametrów, jak to pokazano w podrozdziale 2.1.

Stworzenie instancji parametryzowanego modułu również może mieć dwa formaty:

module_name # (*value, value*, …) *instance_name* (*signal*, …);

module_name # (.*parameter_name* (*value*), .*parameter_name* (*value*), …)
instance_name (*signal*, …);

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

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

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

Jako dodatkowy przykład rozważony zostanie wariant opisu 16-bitowego układu odejmującego. Przedstawiono go na listingu 2.7. Do wykonania operacji odejmowania wykorzystano następującą zasadę arytmetyki binarnej: $A - B = A + \overline{B} + 1$; gdzie $\overline{B}$ – oznacza inwersję wszystkich bitów słowa *B*.

LISTING 2.7. Opis układu 16-bitowego układu odejmującego

```
module sub_16 (input [15:0] A, B,
               output [15:0] D, output COUT);
    wire [15:0] nB = ~B;                          // inwersja bitów słowa B

    add_N # (16) S_16 (A, nB, 1'b1, D, COUT);     // instancja sumatora
endmodule
```

Zadanie. Stwórz opis parametryzowanego układu realizującego odejmowanie, wykorzystując arytmetyczną operację odejmowania.

2.6. Niejawne przekazywanie parametrów

W języku Verilog możliwa jest także niejawna deklaracja wartości parametrów instancji modułu. W tym celu wykorzystuje się operator **defparam**, który posiada następującą składnię:

defparam *hierarchy_path.parameter_name* = *value*;

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

Zaletą wykorzystania operatora **defparam** jest to, że może on być używany w dowolnym miejscu kodu oraz może redefiniować wartości parametrów dowolnej instancji modułu.

Uwaga. Należy uważać, stosując operator **defparam**, ponieważ jego wykorzystanie zmniejsza czytelność kodu.

Przykład wykorzystania operatora **defparam**:

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

2.7. Tablice instancji modułów

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

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

module_name instance_name instance_array_range

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

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

Zakres tablicy instancji posiada następującą składnię:

[*left_index* : *right_index*]

gdzie *left_index* i *right_index* – lewy i prawy indeks zakresu tablicy.

Liczba tworzonych instancji modułów określa się wyrażeniem

$$\text{abs}(left\_index - right\_index) + 1$$

gdzie *abs* – funkcja zwracająca wartość bezwzględną.

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

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

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

Uwaga. Struktura tablicy instancji modułów nie jest realizowana we wszystkich kompilatorach języka Verilog.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator umożliwia tworzenie tablic instancji modułów.

Listing 2.8 przedstawia przykład tworzenia tablic instancji modułów.

LISTING 2.8. 8-bitowy bufor składający się z ośmiu jednobitowych prymitywów **bufif0**

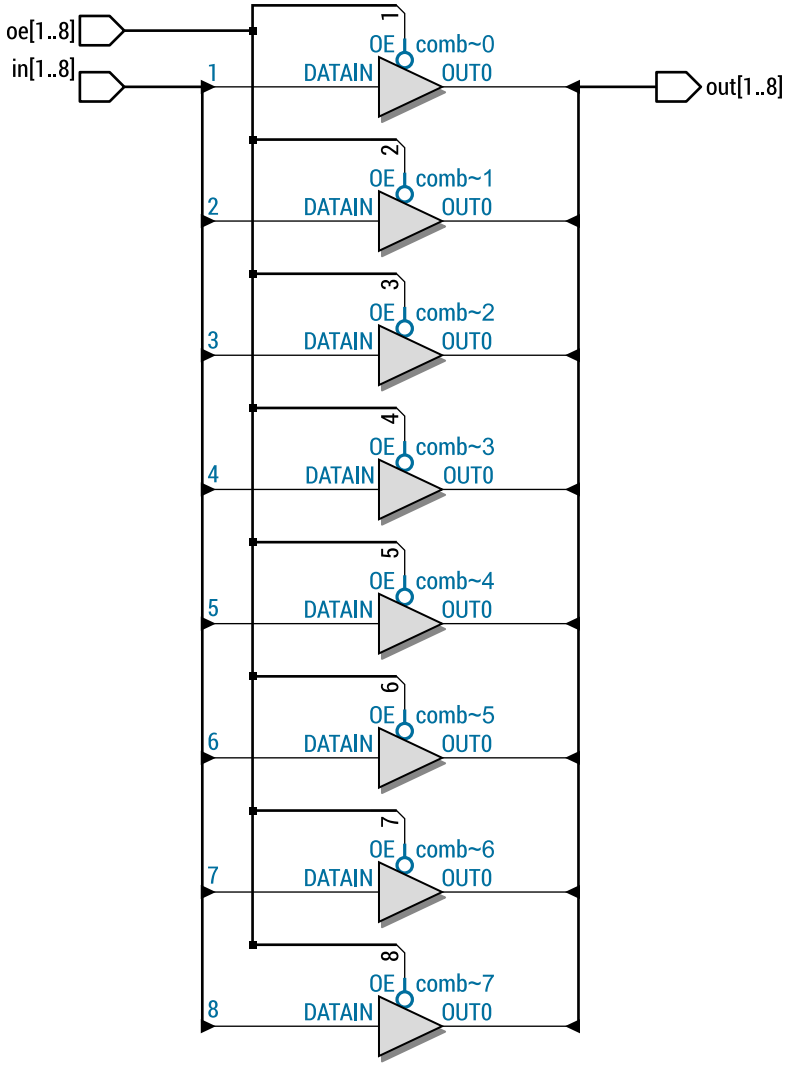
```
module buf_8_1 (input [1:8] in, oe,
                output [1:8] out);

    bufif0 b[1:8] (out, in, oe);          // stworzenie ośmiu instancji bufif0
endmodule
```

Powyższy przykład jest równoważny następującemu fragmentowi kodu:

```
module buf_8_2      (input [1:8] in, oe,
                     output [1:8] out);
bufif0  (out[1], in[1], oe[1]),
        (out[2], in[2], oe[2]),
        (out[3], in[3], oe[3]),
        (out[4], in[4], oe[4]),
        (out[5], in[5], oe[5]),
```

```
        (out[6], in[6], oe[6]),
        (out[7], in[7], oe[7]),
        (out[8], in[8], oe[8]);
endmodule
```

Realizacja listingu 2.8 została ukazana na rysunku 2.5.



RYS. 2.5. Realizacja modułu *buf_8_1* z listingu 2.8: przykład budowy 8-bitowego bufora składającego się z ośmiu jednobitowych prymitywów **bufif0**

Zadanie. Opisz 8-bitowy bufor składający się z ośmiu jednobitowych prymitywów **bufif0**, który jest sterowany pojedynczym sygnałem bitu OE.

Kilka innych przykładów tablic instancji [11]:

```
/* przykład ośmiu 8-bitowych buforów z trzema stanami; każda instancja
podłączona jest do 64-bitowego wektora; wspólna szyna sterowania en podłączona
jest do wszystkich instancji */
module tribuf64bit (output    wire [63:0] out,
                    input     wire [63:0] in,
                    input     wire en);
        tribuf8bit i [7:0] (out, in, en);
endmodule

/* przykład 8-bitowego bufora z trzema stanami zbudowanego z ośmiu
jednobitowych prymitywów bufif1
*/

module tribuf8bit (output     wire [7:0] y,
                   input      wire [7:0] a,
                   input      wire en);
        bufif1 u [7:0] (y, a, en);
endmodule
```

Jednak przy opisie rejestrów wykorzystywanie mechanizmów tworzenia tablic instancji nie jest wymagane, demonstruje to poniższy przykład.

```
/* przykład 8-bitowrego rejestru zbudowanego z jednobitowych prymitywów dff */
module reg_dff_8    (output [7:0] Q,
                     input [7:0] D,
                     input clk, rst);
        dff reg_8 (.d(D), .clk(clk), .clrn(rst), .q(Q));
endmodule
```

2.8. Hierarchia modułów i zmiennych

Zwykle projekty większości urządzeń oraz systemów cyfrowych są zbiorem wielu modułów. Wykorzystanie instancji pojedynczego modułu w ramach innego modułu tworzy hierarchię modułów. Moduł znajdujący się na szczycie drzewa hierarchii jest nazywany *głównym modułem* albo *modułem najwyższego poziomu* (*top-level module*, *highest-level module*). W każdym projekcie tylko jeden moduł może być modułem głównym. Kompilator pakietu Quartus automatycznie tworzy graficzną prezentację drzewa hierarchii modułów projektu. Przykładem hierarchicznego projektu może być 2-bitowy sumator przedstawiony na listingu 2.2.

W języku Verilog liczba poziomów hierarchii modułów i ich powiązań w drzewie hierarchii nie jest ograniczona. Nasuwają się więc pytania:

1) jak się odwołać do potrzebnego modułu z określonego miejsca w drzewie hierarchii modułów;
2) czy różne moduły mogą przyjmować takie same nazwy na różnych gałęziach drzewa hierarchii.

Odpowiedź na pierwsze pytanie jest wymagana, gdy pewien węzeł (np. licznik albo komparator) jest często wykorzystywany w różnych częściach projektu. Odpowiedź na drugie pytanie staje się aktualna, gdy należy połączyć ze sobą części projektu utworzone przez różnych projektantów.

Odpowiedzią na powyższe pytania jest *hierarchiczna nazwa modułu*. W języku Verilog położenie modułu w hierarchii modułów określa się hierarchiczną nazwą bądź hierarchiczną ścieżką (*hierarchy path*). Można rozróżnić pełną (*full*) bądź względną (*relative*) ścieżkę hierarchii.

Pełna nazwa (pełna ścieżka) modułu składa się z nazwy modułu najwyższego poziomu i kolejnych nazw instancji modułów niższych poziomów wiodących do obiektu, do którego wykonuje się odwołanie. Różne nazwy instancji modułów w hierarchicznej ścieżce oddziela się od siebie kropką. Względna nazwa (względna ścieżka) zaczyna się od nazwy instancji w bieżącym module, po której następuje dowolna liczba nazw instancji modułów prowadzących do obiektu odwołania.

W ten sposób moduły z takimi samymi nazwami, jednak znajdujące się na różnych gałęziach drzewa hierarchii, są różnymi modułami, ponieważ posiadają inne nazwy hierarchiczne. Oprócz tego, w dowolnym miejscu projektu można stworzyć instancję dowolnego modułu projektu. W tym celu wystarczy znać jego pełną hierarchiczną nazwę. Aby skrócić długie hierarchiczne nazwy, można wykorzystywać względne nazwy modułów.

2.9. Obszary hierarchii i zakresy widoczności zmiennych

W języku Verilog występują cztery główne typy zakresów działania nazw:

- obszar działania nazw globalnych;
- obszar działania określonego modułu, funkcji, zadań i nazwanych bloków;
- obszar działania bloków specyfikacji;
- obszar działania atrybutów.

Nazwy globalne są widoczne na wszystkich obszarach działania nazw. Do nazw globalnych można zaliczyć:

- nazwy modułów;
- nazwy prymitywów;
- nazwy określonych konfiguracji;
- nazwy makrotekstów tworzonych za pomocą systemowego polecenia **`define**.

Uwaga. Nazwy makrotekstów są widoczne dopiero od miejsca opisania ich w kodzie. Przed ich określeniem nie można się do nich odwoływać.

Obszary widoczności zmiennych tworzą nowe poziomy hierarchii. Do takich obszarów można zaliczyć:

- deklaracje modułów;
- deklaracje funkcji;
- deklaracje procedur;
- nazwane bloki (*named blocks*) **begin-end** i **fork-join**.

Oprócz tego, własne obszary widoczności ustalają:

- bloki specyfikacji (*specify blocks*) i
- atrybuty.

Identyfikator określony w ramach danego obszaru widoczności jest unikalny w danym obszarze i nie może w nim zostać ponownie zdefiniowany. Odwołania się do obiektów w ramach danego obszaru widoczności prowadzą do przeszukiwania w pierwszej kolejności obszaru lokalnego, a następnie obszarów znajdujących się wyżej w hierarchii modułu, aż do granic tego modułu.

3. Prymitywy i moduły biblioteczne

3.1. Gdzie można znaleźć gotowe rozwiązanie

Zanim zacznie się tworzyć układy bądź systemy cyfrowe, należałoby się dowiedzieć więcej o pierwotnych elementach (prymitywach) języka Verilog, prymitywach kompilatora, których liczba zazwyczaj jest znacznie większa od liczby prymitywów języka, a także zapoznać się z bibliotekami standardowych modułów udostępnianych przez wykorzystywane zintegrowane środowisko oraz z bezpłatnymi i płatnymi bibliotekami gotowych projektów w języku Verilog.

Oczywiście, można tego nie robić, jednak łatwo można sobie wyobrazić rozczarowanie towarzyszące wielogodzinnej pracy nad pewnym elementem, gdy okazuje się, że ten element został już wcześniej zrealizowany i jest ogólnodostępny. Oprócz tego, gotowy projekt jest znacząco lepszy, biorąc pod uwagę takie parametry jak koszt realizacji (zajmowany obszar), szybkość działania (opóźnienie sygnału na ścieżce krytycznej lub maksymalna częstotliwość pracy) oraz pobierana moc.

Wszystkie ogólnodostępne komponenty projektu w języku Verilog można podzielić na następujące grupy:

- prymitywy języka Verilog;
- prymitywy kompilatora Verilog;
- biblioteki prymitywów wykorzystywanego zintegrowanego środowiska (w danym przykładzie pakiet Quartus);
- biblioteki bloków IP (*Intellectual Property*) wykorzystywanego zintegrowanego środowiska;
- ogólnodostępne biblioteki bloków IP;
- biblioteki bloków IP innych projektantów.

Podstawowe prymitywy języka Verilog są ograniczone, tzn. standard języka Verilog zawiera tylko niewielki zbiór prymitywów. Zaletą języka Verilog jest to, że mogą one być wykorzystywane w dowolnym miejscu projektu, bez wcześniejszej deklaracji czy inicjacji. Nazwy prymitywów zawsze są widoczne na wszystkich poziomach w drzewie hierarchii.

Biblioteki prymitywów zintegrowanego środowiska projektowania nie różnią się niczym w wykorzystaniu od prymitywów języka: aby z nich skorzystać nie są wymagane żadne specjalne warunki, które musi spełnić projektant.

Zazwyczaj, oprócz bibliotek prymitywów, zintegrowane środowiska projektowe (takie jak np. pakiet Quartus) udostępniają duży zbiór gotowych rozwiązań standardowych bloków funkcjonalnych opracowanych przez specjalistów z wysokimi kwalifikacjami, które dodatkowo są bardzo dokładnie przetestowane.

3.2. Prymitywy języka Verilog

Wbudowane w język prymitywy można podzielić na dwie kategorie: bramki oraz przełączniki. Kategoria prymitywów bramkowych, oprócz samych bramek, zawiera także bufory. Opis prymitywów bramkowych znajduje się w tabeli 3.1.

TABELA 3.1. Bramkowe prymitywy języka Verilog

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

Kategoria przełączników zawiera różne rodzaje tranzystorów. Może się wydawać dziwne, że prymitywami języka projektowania wysokiego poziomu, takiego jak Verilog, są tranzystory. Jednak nie należy zapominać, że Verilog początkowo przeznaczony był do symulacji, więc powinien umożliwiać symulację elementów układów cyfrowych na każdym poziomie, począwszy od tranzystorów, a kończąc na poziomie strukturalnym czy hierarchicznym. Opis przełącznikowych prymitywów języka Verilog umieszczono w tabeli 3.2.

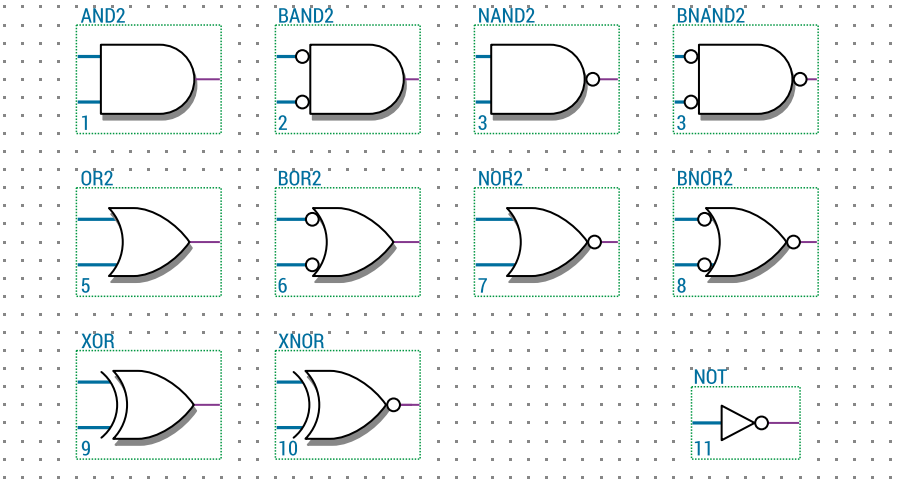
TABELA 3.2. Przełącznikowe prymitywy języka Verilog

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

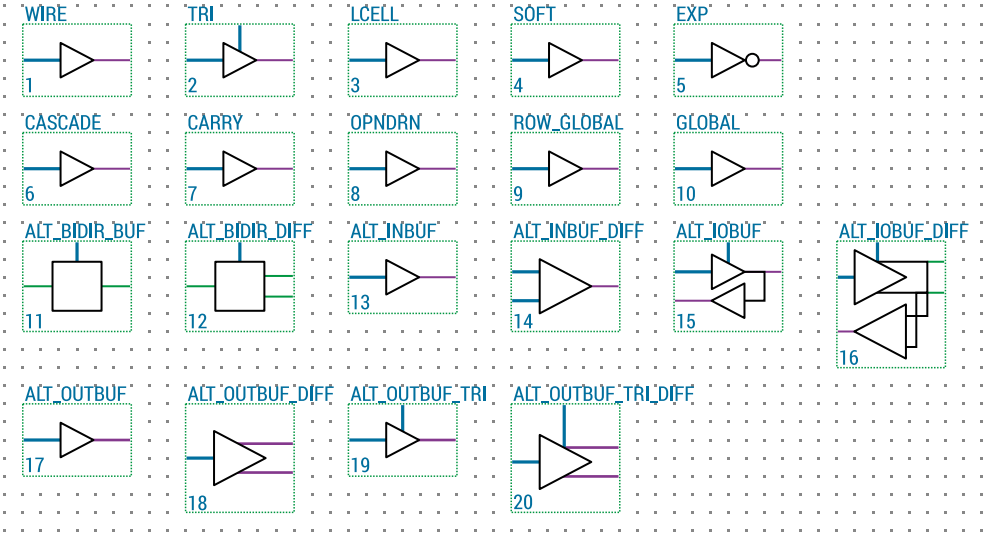
Uwaga. Nie wszystkie kompilatory języka Verilog umożliwiają wykorzystanie prymitywów przełącznikowych, ponieważ w układach programowalnych prymitywy przełącznikowe nie występują.

Zadanie. Dowiedz się, czy wykorzystywany przez Ciebie kompilator języka Verilog umożliwia wykorzystywanie prymitywów przełącznikowych.

W pakiecie Quartus prymitywy takie jak *pullup* i *pulldown*, oraz wszystkie prymitywy przełącznikowe nie są wspierane. Jednakże dodano liczny zbiór prymitywów logicznych oraz buforowych, przedstawionych odpowiednio na rysunkach 3.1 oraz 3.2.



RYS. 3.1. Prymitywy logiczne w pakiecie Quartus



RYS. 3.2. Prymitywy buforowe w pakiecie Quartus

Aby stworzyć instancję prymitywu bramkowego bądź przełącznikowego należy posłużyć się następującą składnią:

gate-type (*drive_strength*) **#** (*delay*) *instance_name*
[*instance_array_range*] (*terminal*, *terminal*, …);

switch_type **#** (*delay*) *instance_name*
[*instance_array_range*] (*terminal*, *terminal*, …);

W zaprezentowanych strukturach konstrukcje *delay*, *strength*, *instance_name* oraz *instance_array_range* są opcjonalne.

Uwaga. Porty określane w procesie tworzenia instancji prymitywu nazywane są terminalami (*terminals*).

Konstrukcja *delay* ustala opóźnienie sygnałów przechodzących przez prymityw. Domyślnie, opóźnienie to wynosi 0. Do określenia opóźnienia można wykorzystywać liczby całkowite i rzeczywiste, które wyrażają liczbę jednostek czasu opóźnienia (rozmiar pojedynczej jednostki czasu definiuje systemowa dyrektywa **`timescale**).

Wielkość opóźnienia może być określona z pomocą jednej, dwóch bądź trzech wartości:

# del	// określenie wielkości opóźnienia jedną
	// wartością
# (*del10, del01*)	// określenie wielkości opóźnienia dwoma
	// wartościami
# (*del10, del01, del2*)	// określenie wielkości opóźnienia trzema
	// wartościami

gdzie *del* – wartość opóźnienia dla wszystkich zmian stanów; *del10* – wartość opóźnienia przy zmianie stanu 1 na 0; *del01* – wartość opóźnienia przy zmianie stanu 0 na 1; *del2* – wartość opóźnienia przy zmianie na stan wysokiej impedancji.

Każda wartość *del*, *del10, del01* i *del2* może być reprezentowana albo pojedynczą liczbą, albo z wykorzystaniem niniejszego formatu:

min : typ : max

gdzie *min* – minimalna, *max* – maksymalna, *typ* – typowa wartość opóźnienia.

Przykłady deklaracji opóźnień:

- #3 – opóźnienie o 3 jednostki czasu dla wszystkich zmian stanów;
- #2:3:4 – minimalne opóźnienie 2, maksymalne 4 oraz średnie 3 jednostki czasu dla wszystkich zmian stanów;
- #(5,6) – opóźnienie zmiany stanu z 1 na 0 wynosi 5, natomiast ze stanu 0 na 1 wynosi 6 jednostek czasu;

- #(2:3:4, 3:4:5, 4:5:6) – opóźnienie zmiany stanów z 1 na 0 określone jest za pomocą trójki 2:3:4, przy zmianie stanów z 0 na 1 określone trójką 3:4:5, natomiast przy przejściu na stan wysokiej impedancji – trójką 4:5:6.

Konstrukcja *strength*, określająca siłę źródła sygnału może mieć następujący format:

(*strength1, strength0*) // przy zmianie stanów z 1 na 0
(*strength0, strength1*) // przy zmianie stanów z 0 na 1

gdzie *strength0* i *strength1* mogą być dowolnymi ze słów kluczowych zawartych w tabeli 1.4, która określa poziomy mocy poszczególnych źródeł sygnałów logicznych.

Siła źródła sygnału może być określona jedynie dla prymitywów bramkowych. Prymitywy przełącznikowe przekazują sygnały z jednakową siłą na wejściu oraz wyjściu.

Przykłady określenia siły źródła sygnału:

- (supply1, pull0) – określa siłę źródła sygnału przy przełączeniu ze stanu 1 na 0 w następujący sposób: supply1 – dla sygnału o wartości jeden oraz pull0 – dla wartości zero;
- (supply0, pull1) – określa siłę źródła sygnału przy przełączeniu ze stanu 0 na 1 w następujący sposób: supply0 – dla wartości zero oraz pull1 – dla wartości jeden;

Nazwa instancji prymitywu (*instance_name*) może być wykorzystywana w odniesieniu do określonego prymitywu w narzędziach do debugowania, przy konfiguracji diagramów czasowych, itp.

Konstrukcja zakresu tablicy instancji *instance_array_range* umożliwia jednoczesną deklarację kilku instancji prymitywów, każdy podłączony jest do określonego bitu wektora sygnałów. Zakres posiada następującą konstrukcję:

[*left_index* : *right_index*]

gdzie *left_index* i *right_index* – lewy i prawy indeks zakresu.

Instancje z tablicy prymitywów podłączane są do wektora sygnałów w kolejności od prawego indeksu do lewego. Każda instancja ze zbioru prymitywów otrzymuje własny indeks instancji. Prymitywy kolejno są przyłączane do wektora, zaczynając od skrajnie prawego indeksu tablicy instancji i podłączane są do wektora od prawego bitu do lewego. Szerokość wektora bitowego powinna być równa liczbie elementów tablicy instancji prymitywów.

Jeżeli zamiast wektora wykorzystuje się wartość skalarną (pojedynczy sygnał), wówczas jest on podłączany do wszystkich elementów tablicy.

Uwaga. Konstrukcja *instance_array_range* nie jest wspierana przez wszystkie kompilatory języka Verilog.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator wspiera konstrukcję *instance_array_range*.

Uwaga. Jeżeli Twój kompilator nie wspiera konstrukcji *instance_array_range*, możliwe jest także tworzenie podobnych struktur z wykorzystaniem bloku generacji (*generate block*).

Przykłady tworzenia instancji prymitywów:

- **and** b1 (y, in1, in2); // 2-wejściowa bramka AND z zerowym opóźnieniem
- **and** # 5 (o1, i1, i2, i3, i4); /* 4-wejściowa bramka AND z opóźnieniem 5 jednostek czasu dla wszystkich zmian stanów */
- **not** # (2,3) u7 (y, in); /* inwerter, w którym określono dwa różne opóźnienia: przy zmianie sygnału z 1 na 0 – 2 jednostki czasu, natomiast przy zmianie z 0 na 1 – 3 jednostki czasu */
- **buf** (pull0, strong1) (y, a); /* bufor z różną siłą sygnału dla zerowej (*pull0*) i jedynkowej (*strong1*) wartości */
- **wire** [31:0] y, a; // deklaracja 32-bitowego wektora
- **buf** # 3.5 i [31:0] (y, a); /* deklaracja tablicy 32 buforów prymitywów *i0*, *i1*, …,*i31*, podłączonych do wektorów *y* oraz *a*; opóźnienie dla każdego elementu wynosi 3,5 jednostek czasu */

3.3. Prymitywy definiowane przez użytkownika

Jeżeli liczba prymitywów predefiniowanych w języku Verilog jest niewystarczająca, użytkownik może tworzyć własne prymitywy. W tym celu w języku Verilog zaoferowano specjalny mechanizm o nazwie: prymitywy definiowane przez użytkownika (*user defined primitives* – UDP). Zgodnie z mechanizmem UDP, składnia służąca do określenia prymitywu wygląda następująco:

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** – słowa kluczowe; *primitive_name* – nazwa prymitywu; *logic_value* – wartość logiczna; *terminal_declaration* – deklaracja wyjściowego terminalu (portu); *terminal_declarations* – deklaracja wejściowych terminali (portów); *table_entry* – wiersz tablicy prawdy.

Stary format deklaracji prymitywu wyglądał następująco:

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

Uwaga. Nie wszystkie kompilatory języka Verilog wspierają mechanizm prymitywów definiowanych przez użytkownika.

Zadanie. Dowiedz się, czy wykorzystywany przez Ciebie kompilator wspiera mechanizm prymitywów definiowanych przez użytkownika.

Wszystkie terminale (porty) w definicji prymitywu powinny mieć szerokość równą jednemu bitowi. Prymitywy mogą mieć tylko jedno wyjście, które powinno być pierwsze na liście parametrów. Maksymalna liczba wejść do prymitywów kombinacyjnych wynosi 10, dla sekwencyjnych – 9. Działanie prymitywu określa się za pomocą *tablicy prawdy*. Do definicji określonych wartości w tablicy prawdy mogą być wykorzystane logiczne poziomy 0, 1 oraz **X**, przy czym wartość **Z** jest tu tożsama z wartością **X**.

Konstrukcja **reg** = *logic_value* jest konstrukcją opcjonalną, wykorzystuje się ją do określenia wartości początkowej (po podłączeniu zasilania) wyjścia prymitywów sekwencyjnych.

Pojedynczy wiersz tablicy prawdy (*table_entry*) określa wartość wyjścia dla jednego zbioru wartości wejściowych. Składnia linii tablicy prawdy dla prymitywów kombinacyjnych wygląda następująco:

input_logic_values : output_logic_value;

oraz dla prymitywów sekwencyjnych:

input_logic_values : previous_state : output_logic_value;

gdzie *input_logic_values* – zbiór wartości wejściowych; *output_logic_value* – wartość wyjściowa; *previous_state* – wartość poprzedniego stanu wyjścia.

Wartości wejściowe w poszczególnych wierszach tablicy prawdy oddzielane są od siebie białymi znakami, przy czym powinny występować w tej samej kolejności, w której występują na liście definiującej porty wejściowe. Jeżeli jakaś kombinacja wejściowych wartości nie zostanie uwzględniona w tablicy prawdy, wówczas automatycznie zostanie ustawiona wartość wyjściowa jako **X**.

Tylko jeden z sygnałów wejściowych (zwykle wykorzystywany jako sygnał synchronizacji w prymitywach sekwencyjnych) może przyjmować przejściową wartość sygnału – oznaczaną zboczem lub przełączeniem poziomów. Jeżeli istnieje taki terminal wejściowy, to w tablicy prawdy powinny być określone wszystkie możliwe kombinacje zboczy i przełączenia poziomów tego sygnału.

Jeżeli w wierszu tablicy prawdy wartość przejściowa zostanie określona dla jednego wejścia, prymityw staje się czuły na zbocze tego sygnału dla wszystkich wejść. W związku z tym, wszystkie pozostałe wejścia powinny mieć dodatkowe wiersze w tablicy prawdy celem pokrycia ich przejść. Jeżeli jakieś przejście nie będzie znalezione, wówczas na wyjściu prymityw będzie przyjmował wartość **X**. Poziomy czułości wejść mają przewagę nad przejściowymi wartościami w tablicy prawdy.

Do określenia wartości sygnałów w tablicy prawdy mogą być wykorzystywane następujące symbole:

- 0 – logiczne zero na wejściu bądź wyjściu;
- 1 – logiczne jeden na wejściu bądź wyjściu;
- *x* lub *X* – nieznana wartość na wejściu bądź wyjściu;
- – – nieznana wartość sygnału wejścia (dla prymitywów sekwencyjnych);
- ? – nieokreślona wartość (*don't care*); dla wejścia, może przyjmować 0, 1 lub **X**;
- *b* lub *B* – nieokreślona wartość (*don't care*); dla wejścia, może przyjmować 0 lub 1;
- (*V W*) – zmiana wartości (przejście) sygnału wejściowego z *V* na *W*, na przykład (01) przedstawia przejście z 0 na 1;
- *r* lub *R* – wzrastające przejście wejścia, to samo co (01);
- *f* lub *F* – opadające przejście wejścia, to samo co (10);
- *p* lub *P* – pozytywne przejście wejścia: (01), (0X) lub (X1);
- *n* lub *N* – negatywne przejście wejścia: (10), (1X) lub (X0);
- * – dowolne przełączenie wejścia (dowolna zmiana wartości sygnału wejściowego), to samo co (??).

Przykłady prymitywów zdefiniowanych przez użytkownika zostały przedstawione na listingach 3.1 oraz 3.2 [11].

LISTING 3.1. Przykład prymitywu kombinacyjnego zdefiniowanego przez użytkownika

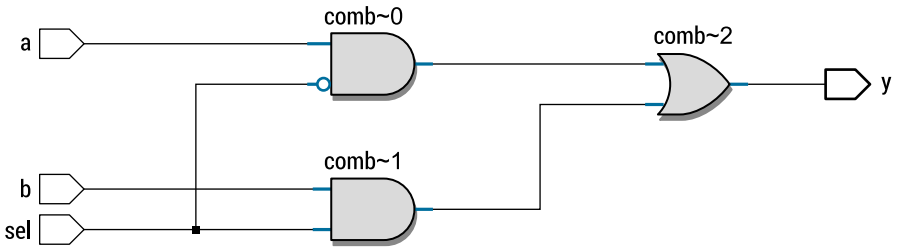
```
primitive my_mux (y, a, b, sel);          // przykład dwuwejściowego multipleksera
        output y;
        input sel, a, b;
        table
```

```
    // a b sel : y
      0  ?  0 :  0;      // wybrano a, wartość b nieokreślona
      1  ?  0 :  1;      // wybrano a, wartość b nieokreślona
      ?  0  1 :  0;      // wybrano b, wartość a nieokreślona
      ?  1  1 :  1;      // wybrano b, wartość a nieokreślona
    endtable
endprimitive
```

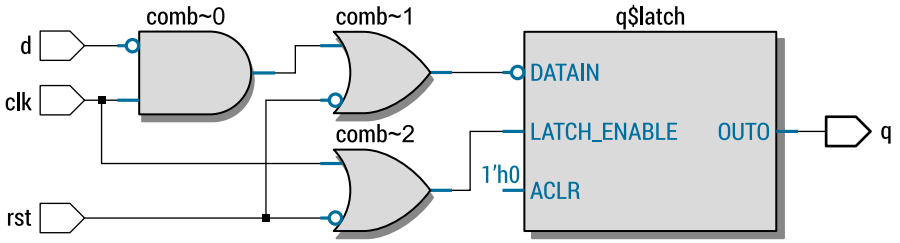
LISTING 3.2. Przykład prymitywu sekwencyjnego określonego przez użytkownika

```
primitive my_dff (output reg q = 0,
                  input d, clk, rst);
    table
    // d  clk  rst : state : q
      ?   ?    0    : ?   : 0;   // odrzucenie przy niskim poziomie rst
      0   R    1    : ?   : 0;   // przy rosnącym zboczu, wyjście przyjmuje
                                  // wartość wejścia
      1   R    1    : ?   : 1;   // przy rosnącym zboczu, wyjście przyjmuje
                                  // wartość wejścia
      ?   N    1    : ?   : –;   // zignorowanie opadającego zbocza
      *   ?    1    : ?   : –;   // zignorowanie wszystkich zboczy na wejściu d
      ?   ?    P    : ?   : –;   // zignorowanie dodatniego zbocza na rst
      0 (0X)   1    : 0   : –;   // eliminacja nieczułości
      1 (0X)   1    : 1   : –;   // eliminacja nieczułości
    endtable
endprimitive
```

Wygląd prymitywów *my_mux* oraz *my_dff* na poziomie przesłań międzyrejestrowych został zaprezentowany odpowiednio na rysunkach 3.3 oraz 3.4.



RYS. 3.3. Realizacja prymitywu użytkownika *my_mux* z listingu 3.1



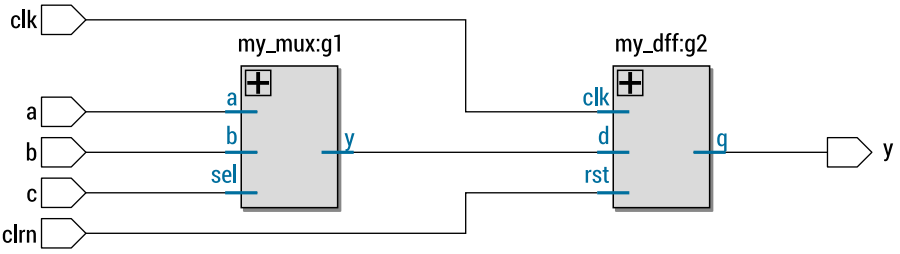
RYS. 3.4. Realizacja prymitywu użytkownika *my_dff* z listingu 3.2

Na listingu 3.3 pokazano przykład wykorzystania prymitywów użytkownika.

LISTING 3.3. Wykorzystanie prymitywów użytkownika

```
module user_primitives (input a, b, c, clk, clrn,
                        output y);
    wire r                          // zmienna wewnętrzna
    my_mux      g1(r,a,b,c);        // multiplekser użytkownika
    my_dff      g2(y,r,clk,clrn);   // przerzutnik użytkownika
endmodule
```

Wygląd projektu z listingu 3.3 na poziomie przesłań międzyrejestrowych został pokazany na rysunku 3.5.



RYS. 3.5. Realizacja modułu *user_primitives* z listingu 3.3

4. Typy danych

4.1. Dwie klasy typów danych

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

Zmienny typ danych (*reg*) wykorzystuje się do tymczasowego przechowywania danych. Deklaracje wartości tych zmiennych mogą występować w blokach proceduralnych **initial** i **always**, a także w ciele procedur i funkcji. Dlatego, jeżeli na przykład wartość wyjściowego portu jest przypisywana wewnątrz bloku proceduralnego **always**, wówczas powinna ona posiadać typ **reg**, nawet, jeżeli występuje w układzie kombinacyjnym.

Uwaga. Zastosowane tu określenie *blok proceduralny* jest rozumiane jako jednostkowa konstrukcja języka Verilog. Więcej o tej konstrukcji zostanie pokazane w dalszych rozdziałach niniejszej książki.

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

Zasady wykorzystywania typów danych:

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

4.2. Sieciowe typy danych

Sieciowe typy danych przeznaczone są do połączeń strukturalnych komponentów projektu. Typ danych *net* wykorzystuje się w następujących sytuacjach:

1) gdy sygnał sterowany jest wyjściem instancji modułu bądź prymitywu;
2) gdy sygnał podłączany jest do wejścia lub dwukierunkowego portu modułu, w którym został zadeklarowany;
3) gdy sygnał znajduje się po lewej stronie operatora przypisania ciągłego (**assign**).

Składnia deklaracji danych typu *net* (sieciowych) może występować w trzech odmianach:

- *net_type* **signed** [*range*] # (*delay*) *net_name* [*array*], …;
- *net_type* (*drive_strength*) **signed** [*range*] # (*delay*) *net_name=continuous_assignment*;
- **trireg** (*capacitance_strength*) **signed** [*range*] # (*delay,delay_time*) *net_name* [*array*], …;

Pierwsza składnia wykorzystywana jest przy deklaracji zbioru lub tablic węzłów jednego typu. Drugi typ składni jest stosowany przy deklaracji węzłów razem z niejawnym operatorem przypisania ciągłego. Trzeci typ składni wykorzystuje się do deklaracji węzła typu **trireg**. W powyższych składniach, konstrukcje *signed*, [*range*], *delay*, [*array*], (*strength*), *decay_time* nie są obowiązkowe (*optional*). Występujący w składni parametr *net_type* przyjmuje postać jednego z następujących słów kluczowych:

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

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

Struktura [*range*] określa zakres bitów sygnału w formacie [*msb*:*lsb*]. Jeżeli w deklaracji struktura zakresu nie jest obecna, wówczas szerokość węzła domyślnie jest równa 1 bitowi. Parametry *msb* i *lsb*, określające najbardziej oraz najmniej znaczące bity zakresu, mogą być liczbami, stałymi, wyrażeniami bądź odwołaniami do funkcji o stałych wartościach. Dopuszczalne jest wskazanie granic zakresu zarówno w porządku indeksów rosnących (*big-endian*), jak i malejących (*little-endian*). Maksymalna szerokość zakresu jest ograniczona wartością 2^{16} = 65 536 bitów. Jednakże wiele kompilatorów dopuszcza szerokość zakresu nawet do 1 miliona bitów.

Zadanie. Dowiedz się, jaką maksymalną szerokość zakresu zmiennych sieciowych umożliwia wykorzystywany przez Ciebie kompilator.

Struktura *delay* (opóźnienie) może być używana jedynie dla sieciowych typów danych. Składnia konstrukcji *delay* odpowiada składni prostych opóźnień (opisanych w podrozdziale 3.2).

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

[*first_address*:*last_address*] [*first_address*:*last_address*]…

gdzie każdy nawias kwadratowy określa jeden z wymiarów tablicy.

Tablica może posiadać dowolną liczbę wymiarów. Każdy wymiar tablicy określa się zakresem adresów. Elementy *first_address* i *last_address* mogą być liczbami, stałymi, wyrażeniami bądź odwołaniem do funkcji o stałej wartości. Wartości adresów mogą być specyfikowane zarówno w porządku rosnącym jak i malejącym. Maksymalny zakres pojedynczego wymiaru macierzy ograniczony jest wartością 2^{24} = 16 777 216, jednak wiele kompilatorów nie ogranicza maksymalnej wartości tego zakresu.

Zadanie. Dowiedz się, jaki maksymalny zakres może przyjmować tablica w używanym przez Ciebie kompilatorze.

Konstrukcja (*strength*) określa poziom logicznej mocy sygnału i posiada następującą składnię:

(*strength0*, *strength1*)

gdzie *strength0* i *strength1* – cyfry od 0 do 7, określające odpowiednio moc logiczną sygnału 0 i 1 (moc logiczna sygnału była opisana w podrozdziale 1.4.2).

Uwaga. Drugi format składni (*drive_strength*) określa logiczną moc źródła sygnału, a w trzecim formacie konstrukcja (*capacitance_strength*) definiuje moc pojemności węzła.

Konstrukcja *delay_time* określa przedział czasu, w ramach którego sieć **trireg** będzie pamiętać (utrzymywać) wartość sygnału po tym, jak wszystkie źródła sygnałów będą odłączone, czyli ich wyjścia będą przełączone na stan wysokiej impedancji. Po minięciu wskazanego czasu sygnał przyjmuje wartość **X**. Konstrukcja *delay_time* posiada następującą składnię:

(*rise_delay*, *fall_delay*, *delay_time*)

gdzie *rise_delay* – opóźnienie wzrastającego zbocza sygnału; *fall_delay* – opóźnienie opadającego zbocza sygnału; *delay_time* – czas wygaszania sygnału. Parametry *rise_delay*, *fall_delay* i *delay_time* przedstawiają dodatnie, całkowite liczby dziesiętne określające interwał czasu liczbą jednostek czasu (wartość jednostek czasu określa się przy pomocy systemowej dyrektywy **`timescale**).

Oprócz tego, dokładnie po wystąpieniu słowa kluczowego określającego typ sieci (*net_type*) może być umieszczone słowo kluczowe **vectored** albo **scalared**, które wskazuje na wektorowy lub skalarny charakter sygnału. Jeżeli typ danych został określony jako wektorowy, wówczas kompilator umożliwia dostęp do poszczególnych bitów danego wektora.

Przykład deklaracji sieci:

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

tri1 [7:0] mag;                  /* 8-bitowa sieć, w trzecim stanie zostaje wyrównana
                                    do poziomu wysokiego */

wire signed [1:8] wyn;           /* 8-bitowa sieć ze znakiem */

wire [7:0] M[0:15][0:63];        /* dwuwymiarowa tablica 8-bitowych węzłów */

wire # (2.8, 1.8) cout;          /* sieć z określonymi opóźnieniami wzrostu (2,8)
                                    i spadku (1,8) poziomu sygnału */

wire [0:15] (strong1, pull0) sum = a + b;  /* 16-bitowa sieć z określoną logiczną
                                              mocą źródła i niejawnym operatorem
                                              przypisania ciągłego */

trireg (small) # (0,0,40) rreg; /* węzeł z logiczną mocą małej pojemności i czasem
                                   pamiętania sygnału równym 40 jednostkom czasu */
```

4.3. Znaczenie sygnałów sieci

Znaczenie sygnału sieci określa się za pomocą następujących typów: **wire** (**tri**), **wand** (**triand**), **wor** (**trior**), **tri0** i **tri1**. Gdy łączy się dwa źródła sygnałów, wynikowa wartość sygnału w sieci zależy od wartości źródeł sygnałów, a także od typu łączącego węzła. Wyniki połączenia dwóch źródeł sygnałów za pomocą różnych typów węzłów zamieszczone są w tabeli 4.1.

TABELA 4.1. Wartość sygnału w różnych węzłach dla dwóch źródeł sygnałów

Wartość źródła		Wartość sygnału w sieci				
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

Wartość źródła		Wartość sygnału w sieci				
a	b	wire (tri)	wand (triand)	wor (trior)	tri0	tri1
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

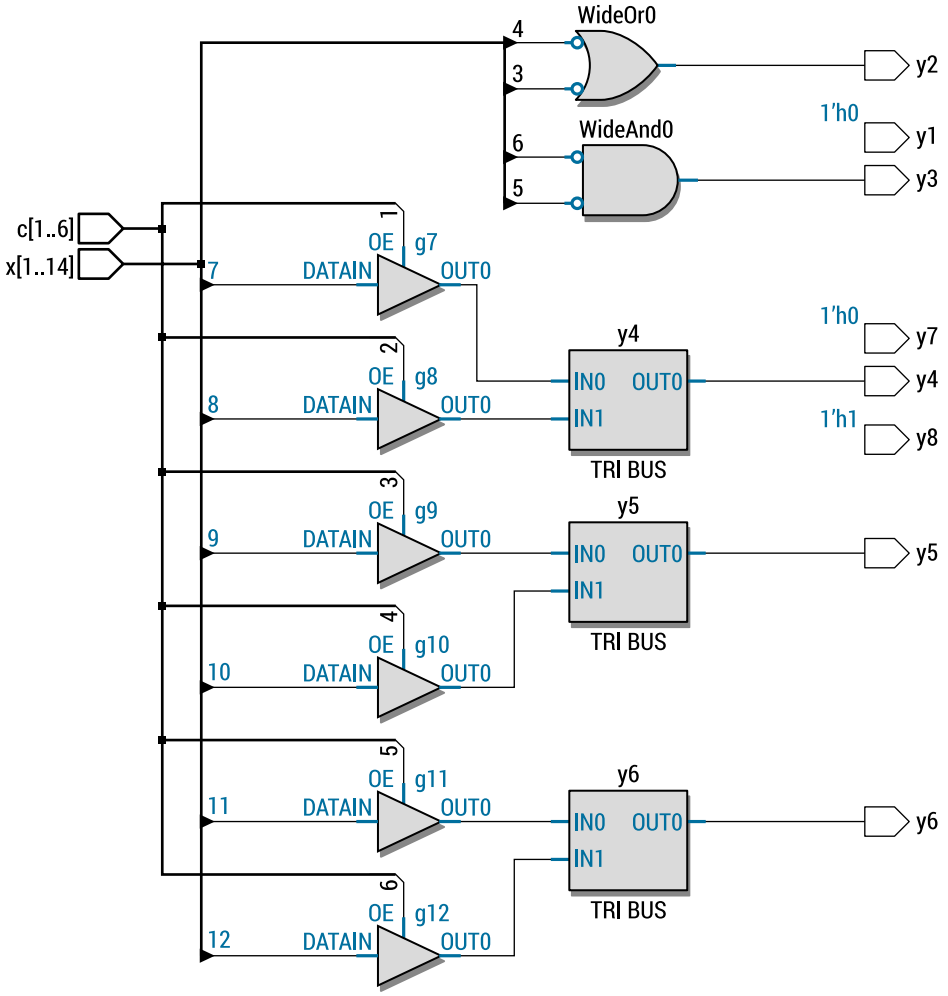
Konieczność określenia wartości sygnału w węźle pojawia się, gdy węzeł łączy kilka źródeł sygnałów. Jako przykład na listingu 4.1 przedstawione zostało połączenie wyjść dwóch buforów i inwerterów za pomocą różnych typów węzłów.

LISTING 4.1. Połączenie różnych typów węzłów

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

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

        bufif1 g7(y4,x[7],c[1]),
               g8(y4,x[8],c[2]);
        bufif0 g9(y5,x[9],c[3]),
               g10(y5,x[10],c[4]);
```

```
        bufif1 g11(y6,x[11],c[5]),
               g12(y6,x[12],c[6]);
endmodule
```

Realizacja listingu 4.1 została przedstawiona na rysunku 4.1.



RYS. 4.1. Realizacja przykładu połączenia wyjść bramek za pomocą węzłów różnego typu z listingu 4.1: *y1* – **wire**; *y2* – **wor**; *y3* – **wand**; *y4* – **tri0**; *y5* – **tri1**; *y6* – **trireg**; *y7* – **supply0**; *y8* – **supply1**

Zadanie. Przeanalizuj listing 4.1 oraz rysunek 4.1, wyjaśnij rezultat syntezy.

4.4. Zmienne typy danych

Zmienne typy danych (**reg**) wykorzystuje się do zapamiętywania wartości w blokach proceduralnych. Zmienne przechowują tylko wartości logiczne, nie mogą zapamiętać logicznej siły sygnału. Gdy sygnał znajduje się po lewej stronie operatora proceduralnego przypisania **assign**, wówczas zawsze powinien być wykorzystywany typ **reg**. W starszych wersjach języka Verilog zmienne nazywano rejestrami (*registers*).

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

variable_type **signed** [*range*] *variable_name*, *variable_name*, …;
variable_type **signed** [*range*] *variable_name* = *initial_value*, …;
variable_type **signed** [*range*] *variable_name* [*array*], …;

Pierwszy format składni wykorzystuje się do deklaracji listy zmiennych jednego typu. Drugi format pozwala na deklarację zmiennych razem z ich początkowymi wartościami. Trzeci format umożliwia deklarację tablic zmiennych. W przedstawionych formatach konstrukcje **signed**, [*range*], [*array*] i *initial_value* są opcjonalne.

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

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

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

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

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

Konstrukcja *initial_value* określa początkowe wartości zmiennych. Początkowa wartość zmiennej jest określana w zerowym punkcie czasu symulacji dokładnie tak, jak wygląda to przy nadawaniu wartości za pomocą procedury **initial**. Jeżeli wartość początkowa nie została określona, wówczas zmienne typu **reg**, **integer** i **time** przyjmują wartość **X**, natomiast zmienne typu **real** i **realtime** – 0.0.

Oprócz tego, od razu po wystąpieniu słowa kluczowego **reg**, mogą być umieszczone takie słowa kluczowe, jak **vectored** lub **scalared**, które mają dokładnie takie same znaczenie, jak dla sieciowych typów danych.

Przykłady deklaracji zmiennych:

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

4.5. Inne typy danych

Oprócz przeanalizowanych powyżej typów danych, w języku Verilog występują także inne typy danych, takie jak: **parameter**, **localparam**, **specparam**, **genvar** i **event**. Nie są to podstawowe typy danych i wykorzystuje się je do funkcji wspomagających.

4.5.1. Parametry

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

Składnia typu danych **parameter**:

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

parameter *constant_type constant_name* = *value*, …

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

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

Przykłady deklaracji parametrów:

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

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

```
s2 = 4'b0010,     // wykorzystuje się do kodowania
s3 = 4'b0100,     // stanów wewnętrznych automatów
s4 = 4'b1000;     // skończonych
```

4.5.2. Parametry lokalne

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

Składnia typu danych **localparam**:

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

localparam *constant_type constant_name* = *value*, …

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

Przykład parametru lokalnego:

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

4.5.3. Parametry bloku specyfikacji

Typ danych **specparam** umożliwia określanie wartości stałych dla bloków specyfikacji (opisywanych w rozdziale 16 niniejszej książki). Typ danych **specparam** może być deklarowany w obszarze działania modułu bądź w obszarze działania bloku specyfikacji. Wartość stałej może być redeklarowana za pomocą plików SDF albo PLI.

Składnia typu danych **specparam**:

specparam *constant_name* = *value*, …

Przykład parametru bloku specyfikacji:

```
specparam    delay_1 = 10;      // deklaracja stałej delay_1 o wartości 10
             t_2 = 3 : 4 : 6;   // przypisanie t_2 wartości opóźnienia
                                // w formacie min : typ : max
```

4.5.4. Zmienne generacji

Typ danych **genvar** określa zmienną, która wykorzystywana jest tylko wewnątrz pętli generowania kodu (*generate loop*) w blokach generacji (bloki generacji opisane są w rozdziale 12 niniejszej książki). Dana zmienna wykorzystywana jest przez kompilator do stworzenia powtarzających się części kodu i nie może być użyta do symulacji.

Składnia typu danych **genvar**:

genvar *event_name*, …

Przykład:

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

4.5.5. Typ danych zdarzenie

Typ danych **event** (zdarzenie) deklaruje flagi, które mogą być wykorzystane do synchronizacji równoległych procesów wewnątrz modułu.

Składnia typu danych **event**:

event *event_name*, …

Przykład:

```
event sync_a, sync_b;     // deklaracja zmiennych typu zdarzenie
                          // sync_a i sync_b
```

4.5.6. Łańcuchy znaków

Łańcuchy znaków w języku Verilog, w odróżnieniu od języków programowania, nie są podstawowymi typami danych. Zgodnie z zasadą łańcuchy znaków nie podlegają syntezie, a ich wykorzystanie sprowadza się do wyprowadzania tekstów podczas symulacji. Łańcuchy znaków są także wykorzystywane jako argumenty niektórych systemowych procedur i funkcji.

Łańcuchy znaków w języku Verilog są to zbiory kolejnych symboli zawartych wewnątrz cudzysłowu („…") zapisane w pojedynczej linii kodu. Łańcuchy znaków wykorzystywane jako operandy w wyrażeniach i przypisaniach powinny być traktowane jako stałe liczbowe bez znaku przedstawione kolejnymi 8-bitowymi wartościami znaków w kodzie ASCII.

W języku Verilog łańcuch znaków deklaruje się jako zmienną typu **reg**, której rozmiar jest równy liczbie symboli pomnożonej razy 8, na przykład:

```
reg [13*8:1] str;
initial begin
        str = "Hello, world!";
end
```

Wszystkie działania na łańcuchach znaków przeprowadza się za pomocą operatorów języka Verilog. Jeżeli łańcuch znaków ma mniejszy rozmiar niż zmienna, to jest ona wypełniana zerami z lewej strony. Jeżeli łańcuch znaków jest większy niż rozmiar zmiennej, wówczas symbole z lewej strony są odrzucane.

Przykład wyświetlania łańcucha znaków przedstawiony został na listingu 4.2 [4].

LISTING 4.2. Testowy przykład pracy z łańcuchami znaków

```
module str_test;
        reg [16*8:1] str;
        initial begin
          str = “Hello, world!”;
          $display(“%s is stored as %h”, str, str);
          str = {str, “!!!”};
          $display(“%s is stored as %h”, str, str);
        end
endmodule
```

Zadanie. Spróbuj zaimplementować przykład z listingu 4.2 w celu sprawdzenia czy używany przez Ciebie kompilator wspiera pracę z łańcuchami symboli.

4.6. Wybór bitów i pól bitowych

Wybór lub odwołanie do jednego bitu wektora przyjmuje następujący format:

vector_name [*bit_number*]

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

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

Uwaga. Polem bitowym nazywany jest nieprzerwany ciąg bitów pewnego wektora.

Wybór pola bitowego może przyjmować następujące trzy formaty składni:

vector_name [*bit_number* **:** *bit_number*]

vector_name [*starting_bit_number* **+:** *port_select_width*]

vector_name [*starting_bit_number* **–:** *port_select_width*]

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

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

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

Przykład:

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

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

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

Zadanie. Sprawdź prawidłowość bądź nieprawidłowość przytoczonych wcześniej przykładów za pomocą kompilatora.

4.7. Wybór elementów tablicy i pól bitowych elementów tablicy

Składnia wyboru elementów tablicy:

array_name [*index*][*index*] …

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

Przykłady wyboru elementów tablicy:

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

Składnia wyboru bitu elementu tablicy:

array_name [*index*][*index*]…[*bit_number*]

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

Przykłady wyboru bitów elementów tablicy:

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

Składnia wyboru bitowego pola elementu tablicy:

array_name [*index*] [*index*]…[*part_select*]

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

Przykłady wyboru pola bitowego:

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

Uwaga. W każdym momencie czasu tylko jeden element tablicy może być odczytany bądź zapisany.

4.8. Deklaracja pamięci

Pamięć w języku Verilog jest przedstawiana za pomocą jednowymiarowych tablic zmiennych typu **reg**. Dane tablice mogą być wykorzystywane do symulacji pamięci typu RAM, ROM lub rejestrów. W ogólnym przypadku tablice w języku Verilog mogą być wielowymiarowe, jednak nie mogą być realizowane wówczas jako zwyczajne bloki pamięci rzeczywistego urządzenia.

Przykłady deklaracji pamięci:

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

Przykłady wykorzystania pamięci:

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

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

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

Uwaga. Do odczytu i zapisu elementów tablic, które reprezentują pamięć (jednowymiarowa tablica zmiennych typu **reg**), można wykorzystywać następujące funkcje systemowe: **$readmemb**, **$readmemh**, **$sreadmemb** i **$sreadmemh**.

5. Operatory

5.1. Operatory języka Verilog

Składnia większości operatorów języka Verilog odpowiada składni, jaka występuje w językach programowania, na przykład w języku C. Operatory wykorzystuje się w wyrażeniach do określenia czynności wykonywanych na operandach. Operandami większości operatorów mogą być: liczby, sieci, zmienne, pojedyncze bity (sygnały), wektory, części wektorów (pola bitowe) czy wywołania funkcji.

Uwaga. Niektóre operatory nie wykonują operacji na liczbach rzeczywistych (zmiennoprzecinkowych).

W operatorach logicznych wartością zwracaną jest *prawda* lub *fałsz*, czyli 1-bitowa wartość 1 oznaczająca prawdę bądź 0 – oznaczająca fałsz. Oprócz tego, może występować wartość **X** odpowiadająca wartości nieokreślonej.

Operatory w języku Verilog zwyczajowo dzieli się na następujące kategorie:

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

5.2. Operatory bitowe

Operatory bitowe języka Verilog zostały przedstawione w tabeli 5.1. Operatory te są często wykorzystywane przy opisie projektów za pomocą funkcji i wyrażeń boolowskich. Dodatkowo pozwalają wykonywać operacje algebry Boole'a na poszczególnych bitach wektorów i pól bitowych.

TABELA 5.1. Operatory bitowe

Symbol	Przykład	Opis
~	~m	inwersja każdego bitu wektora *m*, operator unarny
&	m & n	operacja logiczna I (AND) każdego bitu wektora *m* z każdym bitem wektora *n*

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

Wyniki działania operatorów bitowych zostały przedstawione w tabeli 5.2. Podczas obliczeń bitowych wartość **Z** jest traktowana tak samo jak wartość **X**.

TABELA 5.2. Wyniki operacji bitowych

Operandy		Rezultaty 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
0	X	1	0	X	X	X
X	0	X	0	X	X	X
1	X	0	X	1	X	X
X	1	X	X	1	X	X
X	X	X	X	X	X	X

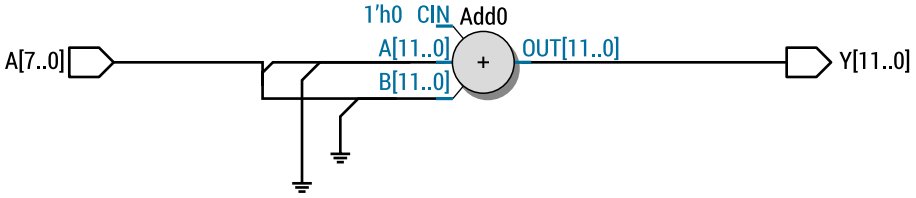
Jako przykład wykorzystania operatora bitowego przesunięcia zaprezentujemy układ mnożący liczbę wejściową przez 10. Opis układu został przedstawiony na listingu 5.1.

LISTING 5.1. Przykład wykorzystania operatora przesunięcia do mnożenia przez 10

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

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

Wynik syntezy przykładu z listingu 5.1 został przedstawiony na rysunku 5.1.



RYS. 5.1. Realizacja modułu *mult_on_10* z listingu 5.1 wykonującego mnożenie liczby *A* przez 10 za pomocą operatora przesunięcia

Zadanie. Opisz parametryzowany moduł mnożenia dowolnej liczby całkowitej przez liczbę 10, w którym jako parametr będzie występować liczba bitów wartości wejściowej.

5.3. Operatory redukcji

Operatory redukcji, nazywane także operatorami skracania, składnią przypominają operatory bitowe, jednak ich działanie różni się od nich diametralnie. Operatory bitowe po kolei są stosowane do każdej pary bitów w dwóch wektorach, w wyniku czego otrzymywany jest nowy wektor. Operatory redukcji najpierw stosowane są do pierwszych dwóch bitów wektora, a następnie operacja jest wykonywana na wyniku poprzedniej operacji oraz kolejnym bicie wektora. Dany proces jest powtarzany dla wszystkich bitów wektora. Rezultatem operacji redukcji jest pojedynczy bit. Innymi słowy, operatory redukcji skracają wektor bitowy do jednego bitu, kolejno stosując odpowiednie wyrażenie logiczne do każdego bitu wektora.

Operatory redukcji języka Verilog zostały przedstawione w tabeli 5.3, a przykład ich wykorzystania – w tabeli 5.4.

TABELA 5.3. Operatory redukcji

Symbol	Przykład	Opis
&	& m	redukuje bity wektora *m* za pomocą I (AND)
~&	~& m	redukuje bity wektora *m* za pomocą NIE-I (NAND)
|	| m	redukuje bity wektora *m* za pomocą LUB (OR)
~|	~| m	redukuje bity wektora *m* za pomocą NIE-LUB (NOR)
^	^ m	redukuje bity wektora *m* za pomocą alternatywy wykluczającej (XOR)
~^ lub ^~	~^ m	redukuje bity wektora *m* za pomocą negacji alternatywy wykluczającej (XNOR)

TABELA 5.4. Przykłady wykorzystania operatorów redukcji

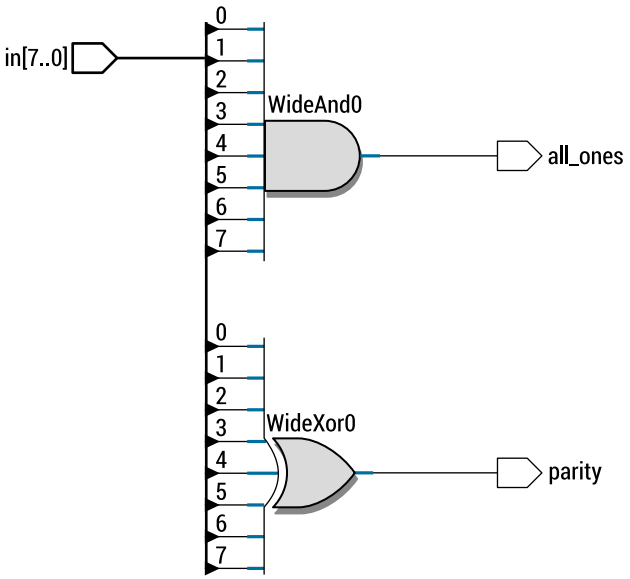
Operand	Rezultat operatorów redukcji		
a	&a	|a	^a
00000000	0	0	0
11111111	1	1	0
10101010	0	1	1
110011zz	0	1	x
1111111x	x	1	x

Przykład wykorzystania operatorów redukcji został przedstawiony na listingu 5.2, gdzie pokazano metodę sprawdzenia, czy liczba jedynek wektora bitowego jest parzysta i czy wszystkie bity mają wartość 1.

LISTING 5.2. Przykład wykorzystania operatorów redukcji

```
module inputs_test (
        input [7:0] in,         // wejściowy wektor bitów
        output parity,          // prawda, jeżeli liczba jedynek jest parzysta
        all_ones);              // prawda, jeżeli wszystkie bity mają wartość 1

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

endmodule
```

Wynik syntezy przykładu z listingu 5.2 został przedstawiony na rysunku 5.2.



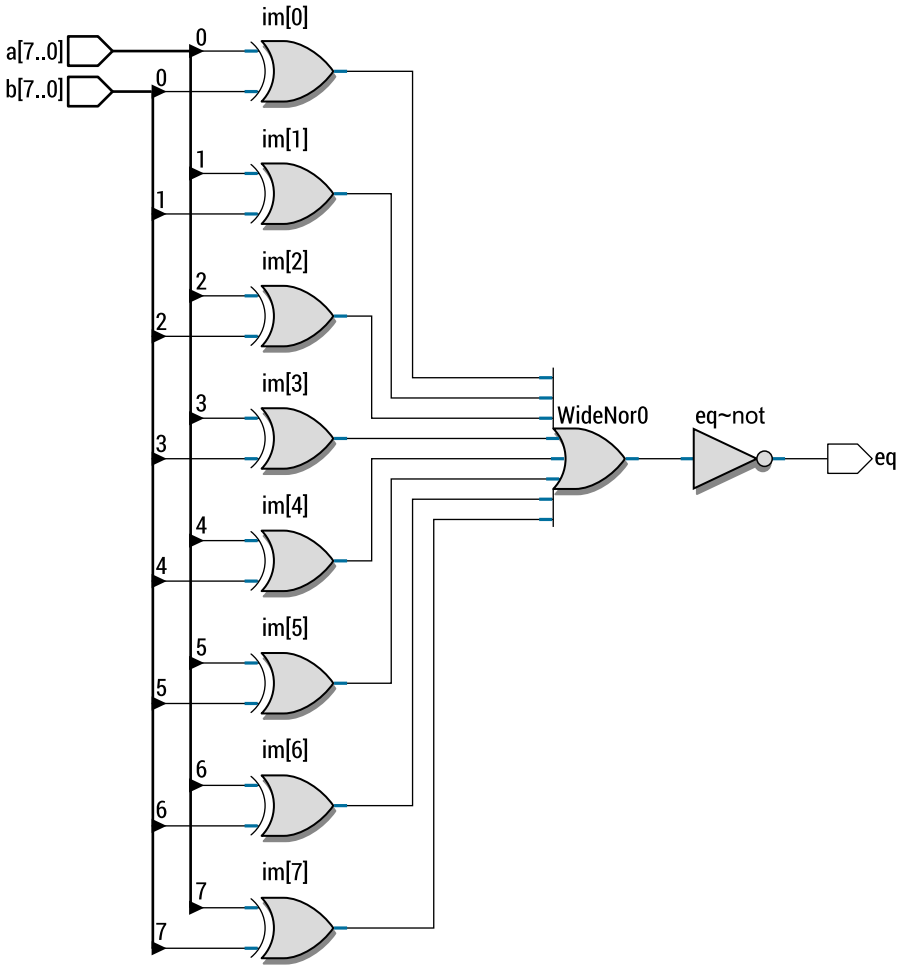
RYS. 5.2. Realizacja modułu *inputs_test* z listingu 5.2: przykład wykorzystania operatorów redukcji do sprawdzenia czy liczba jedynek w słowie jest parzysta oraz czy wszystkie bity mają wartość 1

Na listingu 5.3 zaprezentowano jeszcze jeden przykład wykorzystania operatorów bitowych i operatorów redukcji do realizacji funkcji równości w komparatorze.

LISTING 5.3. Realizacja funkcji równości

```
module comp_eq (
        input [7:0] a, b,       // wektory wejściowe
        output eq);             // funkcja równości

        wire [7:0] im;          // zmienna pomocnicza

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

endmodule
```

Wynik syntezy przykładu z listingu 5.3 został przedstawiony na rysunku 5.3.



RYS. 5.3. Realizacja modułu *comp_eq* z listingu 5.3: przykład wykorzystania operatorów redukcji NOR do realizacji funkcji równości w komparatorze

5.4. Operatory logiczne

Operatory logiczne wykorzystuje się w wyrażeniach logicznych, na przykład w instrukcjach **if**, **for**, **while**. Rezultatem operatorów logicznych jest 1-bitowa wartość przyjmująca 1, jeżeli wyrażenie jest prawdą lub 0 (bądź **X**) – jeżeli to fałsz. Operatory logiczne zostały przedstawione w tabeli 5.5.

TABELA 5.5. Operatory logiczne

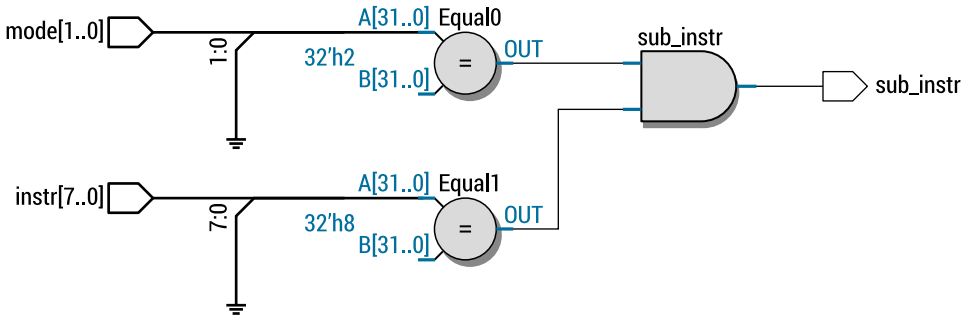
Symbol	Przykład	Opis
!	! m	prawda, jeżeli *m* to fałsz
&&	m && n	prawda, jeżeli *m* i *n* to prawda
||	m || n	prawda, jeżeli albo *m*, albo *n* to prawda

Na listingu 5.3 przedstawiono przykład urządzenia sterującego mikrokontrolera, które sprawdza, czy dane polecenie jest odejmowaniem.

LISTING 5.4. Przykład fragmentu urządzenia sterującego mikrokontrolera

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

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

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

Wynik syntezy przykładu z listingu 5.4 został przedstawiony na rysunku 5.4.



RYS. 5.4. Realizacja modułu *if_subtraction* z listingu 5.4: przykład wykorzystania operacji logicznych podczas realizacji urządzenia sterującego mikrokontrolera

5.5. Operatory relacji

Operatory relacji zazwyczaj są wykorzystywane w wyrażeniach logicznych. Pozwalają na zbadanie relacji (większe, mniejsze, równe itp.) pomiędzy wartościami porównywanych operandów. Rezultatem operacji relacji jest 1-bitowa wartość określająca czy dane wyrażenie jest prawdziwe, czy fałszywe. Operatory relacji zostały przedstawione w tabeli 5.6.

TABELA 5.6. Operatory relacji

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

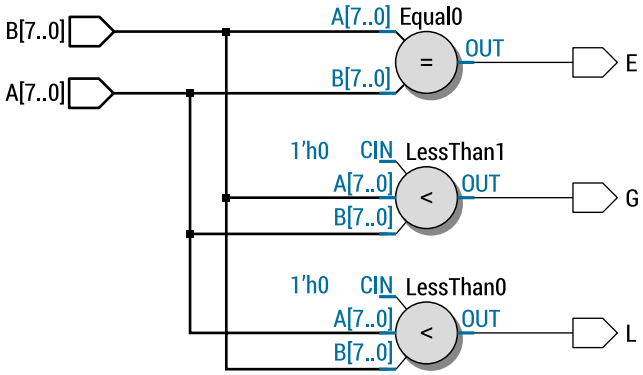
Przykład wykorzystania operatorów relacji został przedstawiony na listingu 5.5.

LISTING 5.5. Parametryzowany komparator

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

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

Rezultat syntezy przykładu z listingu 5.5 został przedstawiony na rysunku 5.5.



RYS. 5.5. Realizacja modułu *comparator* z listingu 5.5: przykład wykorzystania operatora relacji do stworzenia parametryzowanego komparatora

Uwaga. Jeżeli chociażby jeden z operandów relacji posiada wartość któregoś z bitów równą **X** lub **Z**, to rezultatem operatora relacji będzie wartość **X**.

Innymi słowy, operatory relacji pozwalają porównywać tylko wartości operandów, natomiast w przypadku wystąpienia na chociażby jednym bicie w dowolnym operandzie wartości logicznej **X** lub **Z**, wynik wyrażenia relacji będzie wynosił **X**. Powstaje pytanie: w jaki sposób porównać operandy, w których określone bity mogą przyjmować wartości **X** lub **Z**? Służą do tego operatory identyczności.

5.6. Operatory identyczności

Sprawdzanie identyczności w języku Verilog jest wyrażane za pomocą dwóch operatorów: === oraz !==. Różnica pomiędzy tymi operatorami a operatorami relacji równości (==) i nierówności (!=) polega na tym, że oprócz sprawdzenia wartości 0 i 1, pozwalają one na porównywanie wartości logicznych bitów **X** i **Z**. Innymi słowy, operatory identyczności pozwalają sprawdzić wszystkie cztery możliwe stany logiczne każdego bitu operandów: 0, 1, **X** i **Z**.

Uwaga. Nie wszystkie kompilatory wspierają operatory identyczności podczas syntezy projektu.

Zadanie. Sprawdź, czy używany przez Ciebie kompilator wspiera operatory identyczności podczas syntezy projektu.

5.7. Operatory arytmetyczne

Operatory arytmetyczne języka Verilog zostały przedstawione w tabeli 5.7.

TABELA 5.7. Operatory arytmetyczne

Symbol	Przykład	Opis
+	m + n	dodawanie *m* do *n*
–	m – n	odejmowanie *n* od *m*
–	–m	liczba przeciwna do *m* (reprezentacja w kodzie uzupełnień do 2)
*	m * n	mnożenie *m* razy *n*
/	m / n	dzielenie *m* przez *n*
%	m % n	reszta z dzielenia *m* przez *n*
**	m ** n	liczba *m* podniesiona do potęgi *n*
<<<	m <<< n	przesunięcie *m* w lewo o *n* bitów, zwolnione bity z prawej strony zostają wypełnione zerami
>>>	m >>> n	przesunięcie *m* w prawo o *n* bitów; jeżeli *m* jest liczbą ze znakiem, wówczas zwalniane bity z lewej strony wypełniane są wartością bitu znaku, w przypadku liczby bez znaku, są wypełniane zerami

Przykład wykorzystania operatora arytmetycznego:

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

Uwaga. Nie wszystkie operatory arytmetyczne są wspierane przez kompilatory podczas syntezy projektu.

Zadanie. Sprawdź, jakie operatory arytmetyczne są wspierane w wykorzystywanym przez Ciebie kompilatorze podczas syntezy projektu.

5.8. Operatory różne

Inne operacje języka Verilog, które nie zostały wymienione w żadnej z powyższych grup, zostały nazwane operatorami różnymi (*miscellaneous*). Operatory różne są przedstawione w tabeli 5.8.

TABELA 5.8. Operatory różne

Symbol	Przykład	Opis
? :	sel ? m : n	operacja warunkowa, jeżeli *sel* to prawda, zwraca *m*, w przeciwnym wypadku zwraca *n*
{ }	{m,n}	operacja konkatenacji, złączenie *m* z *n*, tworzy wektor

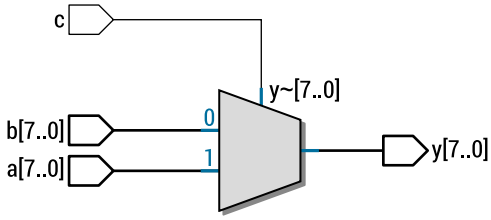
Symbol	Przykład	Opis
{{ }}	{n{m}}	operator powtórzenia, łączy ze sobą *m*, *n*-krotnie
->	-> m	operator wystąpienia zdarzenia, zmienia wartość *m* typu *event*

Za pomocą operatora warunkowego można w prosty sposób zrealizować multiplekser 2-1 (listing 5.6).

LISTING 5.6. Realizacja multipleksera za pomocą operatora warunkowego

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

        assign y = c ? a: b;

endmodule
```

Realizacja przykładu z listingu 5.6 została przedstawiona na rysunku 5.6.



RYS. 5.6. Realizacja *modułu mux_2_1* z listingu 5.6: przykład wykorzystania operatora warunkowego do stworzenia multipleksera

Jako kolejny przykład przedstawione jest wykorzystanie operatora warunkowego do realizacji prostej jednostki arytmetyczno-logicznej (ALU).

LISTING 5.7. Przykład wykorzystania operatora warunkowego

```
module mini_ALU      (input [7:0] a, b,      // argumenty
                      input [2:0] op,        // kod operacji
                      output [7:0] result);  // wynik operacji

parameter    ADD=3'h0, SUB=3'h1, AND=3'h2,  // kody wykonywanych operacji
             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
```

Zadanie. Opisz jednostkę ALU dla realizacji wszystkich binarnych operacji logicznych, wszystkich binarnych operacji arytmetycznych, wszystkich unarnych operacji logicznych oraz wszystkich unarnych operacji arytmetycznych. Stwórz ALU prostego mikrokontrolera.

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

5.9. Wykonywanie operacji

Przed wykonaniem działania wszystkie operandy zostają rozszerzone do rozmiaru największego wektora w wyrażeniu (po obu stronach znaku). Wartości bez znaku uzupełniane są zerami, natomiast ze znakiem – wartością bitu znaku. Operatory konkatenacji i powtórzenia są rozszerzane dla każdej pary operandów.

Wyrażenia arytmetyczne są wykonywane zgodnie z następującymi regułami:

- jeżeli dowolny operand jest liczbą rzeczywistą, wówczas wynikiem wyrażenia jest liczba zmiennoprzecinkowa;
- jeżeli dowolny operand jest wartością bez znaku, wówczas wynik również jest wartością bez znaku;
- jeżeli wszystkie operandy są wartościami ze znakiem, wówczas wykonywana jest arytmetyka dla liczb ze znakiem.

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

5.10. Priorytety operatorów

Operatory w języku Verilog wykonywane są zgodnie z porządkiem priorytetów operatorów. Aby zmienić porządek wykonywania działań należy użyć nawiasów okrągłych. Priorytety operatorów języka Verilog przedstawione są w tabeli 5.9, gdzie priorytet określa kolejność, w jakiej są wykonywane dane operacje.

TABELA 5.9. Priorytety operatorów

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

5.11. Rozmiar wyrażeń bitowych

Rozmiar wyrażeń bitowych w języku Verilog, otrzymywanych w zależności od wykorzystywanego operatora zaprezentowany został w tabeli 5.10.

TABELA 5.10. Rozmiary wyrażeń bitowych

Wyrażenie	Rozmiar	Operator op
i op *j*	max(*L*(*i*), *L*(*j*))	operatory binarne + − / * % & | ^ ~^
op *i*	*L*(*i*)	operatory unarne + − ~
i op *j*	1 bit	=== !== == != && || < <= > >= & ~& | ~| ^ ~^ !
i op *j*	*L*(*i*)	>> << ** >>> <<<
i ? *j* : *k*	max(*L*(*j*), *L*(*k*))	operator warunkowy
{*i*,..., *j*}	*L*(*i*)+...+*L*(*j*)	operator konkatenacji
{*i*{*j*,...,*k*}}	*i**(*L*(*j*)+...+*L*(*k*))	operator powtórzenia

6. Operator przypisania ciągłego assign

6.1. Przypisania w języku Verilog

W języku Verilog nie występują zwyczajne operatory przypisania znane z języków programowania. Jest to spowodowane tym, że nadawanie wartości poszczególnym typom danych (sieciom *net* i zmiennym **reg**) istotnie się od siebie różni. Przypisywanie wartości zmiennym (**reg)** jest podobne do przypisywania wartości w językach programowania. Jednakże sieci (*net*) są przeznaczone do połączeń elementów projektu i można je skojarzyć z przewodami łączącymi wyprowadzenia poszczególnych układów elektronicznych.

Na początku należy zauważyć, że w schematach elektronicznych ścieżki są ciągłe i nieprzerwane, czyli w czasie pracy danego urządzenia bądź schematu nie zmieniają się. Przypisanie wartości ścieżce oznacza nadanie wartości sygnałowi wewnątrz niej, określanemu źródłem sygnału (sterownikiem). Aczkolwiek dla jednej ścieżki takich źródeł sygnałów może być kilka i mogą one pracować niezależnie i jednocześnie.

Operator przypisania ciągłego **assign** określa wartość jednego ze źródeł sygnału sieci. Jako że ścieżki w układach cyfrowych nie są zmienne i na stałe są podłączone do źródeł sygnałów, operator **assign** jest nazywany ciągłym (*continuous*). Ostateczna wartość sygnału w sieci jest określona przez typ ścieżki, czyli w jaki sposób źródło sygnału łączy się z siecią, jako OR AND, z otwartym kolektorem, czy inny (połączenie ścieżek różnego typu było omówione w podrozdziale 4.3).

Ponieważ wszystkie sygnały schematu elektronicznego są przekazywane ścieżkami jednocześnie, to i wszystkie operatory przypisania ciągłego **assign** są wykonywane równolegle. Oprócz tego, w języku Verilog równolegle wykonywane są:

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

6.2. Formaty operatora przypisania ciągłego

Operator przypisania ciągłego **assign** posiada dwie składnie: jawną i niejawną. Jawny format operatora **assign** wygląda następująco:

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

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

net_type [*size*] *net_name*;

Niejawny format operatora **assign** wykorzystywany jest przy deklaracji nazwy ścieżki. W praktyce jest to złączenie dwóch wcześniejszych formatów bez słowa kluczowego **assign**:

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

W zaprezentowanych składniach konstrukcje [*size*], # (*delay*) i (*strength*) nie są obowiązkowe. Wykorzystywany tu *net_type* – jest jednym z sieciowych typów danych, za wyjątkiem typu **trireg**; [*size*] – rozmiar wektora lewej części w formacie [*msb* : *lsb*]; *delay* – czas opóźnienia wykonania operacji przypisania (domyślnie przyjmuje wartość 0), posiada taką samą składnię, jak w prymitywach (przeanalizowanych w podrozdziale 3.2); *strength* – logiczna moc sygnału (opisana w podrozdziale 1.4.2), domyślnie przyjmuje wartość (*strong1*, *strong0*).

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

Wyrażenie *expression* w prawej części operatora przypisania ciągłego **assign** może zawierać różne typy danych, różne operatory i wywołania funkcji.

Operator przypisania ciągłego **assign** działa w następujący sposób. Przy dowolnej zmianie dowolnej wartości w wyrażeniu *expression* z prawej strony, na początku jest obliczane wyrażenie *expression* i otrzymana wartość po upłynięciu czasu opóźnienia *delay* jest przypisywana wartości po lewej stronie znaku =.

Przykłady wykorzystania operatora **assign**:

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

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

// przypisanie ciągłe z wskazaniem logicznej mocy sygnału
wire (strong1, pull0) [63:0] ALU_out = ALU_function(opcode, a, b);
                                           // wywołanie funkcji
```

6.3. Wykorzystanie operatora przypisania ciągłego

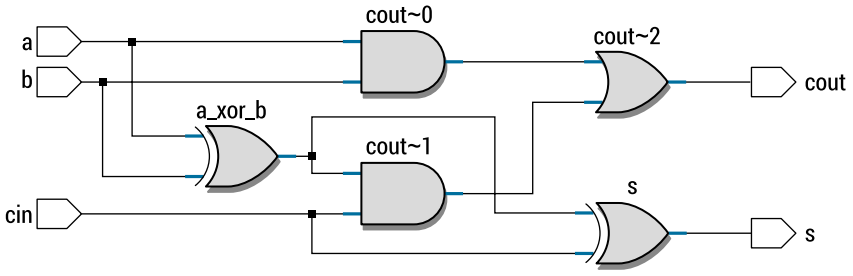
Na początku warto podkreślić dwie ważne właściwości operatora **assign**:
1) równoległość – wszystkie operatory **assign** w jednym module wykonywane są równolegle niezależnie od ich położenia;
2) wrażliwość na zdarzenia związane z sygnałami z prawej strony znaku równości, czyli za każdym razem, gdy następuje zmiana wartości ścieżki bądź zmiennych w prawej części, od początku obliczane jest dane wyrażenie i otrzymaną wartość przypisuje się ścieżce z lewej strony operatora **assign**.

Operator **assign** jest szeroko wykorzystywany do opisu układów kombinacyjnych za pomocą wyrażeń algebry Boole'a. Jako przykład może posłużyć jednobitowy sumator opisany w listingu 1.2 na podstawie równań (1.2). W tym przykładzie nie było potrzeby deklarowania dodatkowych ścieżek, ponieważ wszystkie elementy w ścieżce po lewej stronie wyrażenia były portami modułu. Alternatywą do tego rozwiązania może być opis jednobitowego sumatora z pośrednią ścieżką *a_xor_b* zaprezentowany na listingu 6.1.

LISTING 6.1. Przykład jednobitowego pełnego sumatora z niejawnym wykorzystaniem operatora przypisania ciągłego

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

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

Realizacja przykładu z listingu 6.1 została przedstawiona na rysunku 6.1.



RYS. 6.1. Realizacja modułu *sum_1_1* z listingu 6.1

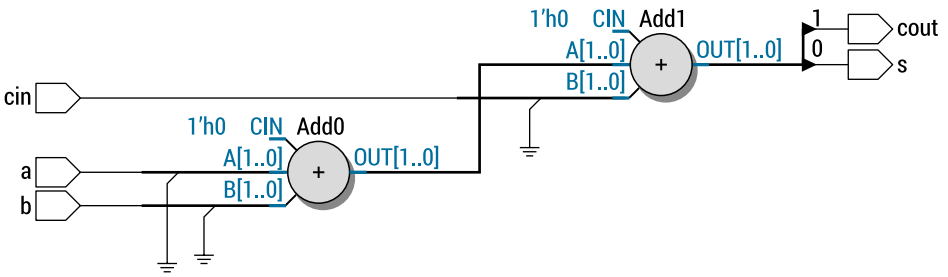
Warto zauważyć, że wykorzystanie operatora **assign** pozwala na opis układów kombinacyjnych bez znajomości struktury układu. Odpada także konieczność projektowania (przekształcania równań logicznych, minimalizacji, faktoryzacji, dekompozycji itp.). Wystarczy znać wyrażenia logiczne opisujące funkcjonowanie układu kombinacyjnego, a o wszystkie pozostałe elementy kompilator zatroszczy się automatycznie. Uwalnia to projektantów od wykonywania formalnych, dość złożonych przekształceń związanych z syntezą logiczną i pozwala skoncentrować się na algorytmie funkcjonowania układu cyfrowego. Oprócz tego, taki styl projektowania znacząco zmniejsza prawdopodobieństwo wystąpienia błędów powstałych w wyniku ręcznego projektowania i zwiększa czytelność kodu projektu.

Po lewej stronie operatora **assign** może występować operacja konkatenacji, jak zostało przedstawione na listingu 6.2.

LISTING 6.2. Przykład wykorzystania operatora konkatenacji w lewej części operatora **assign**

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

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

endmodule
```

Wynik syntezy przykładu z listingu 6.2 został zaprezentowany na rysunku 6.2.



RYS. 6.2. Realizacja modułu *sum_1_2* z listingu 6.2

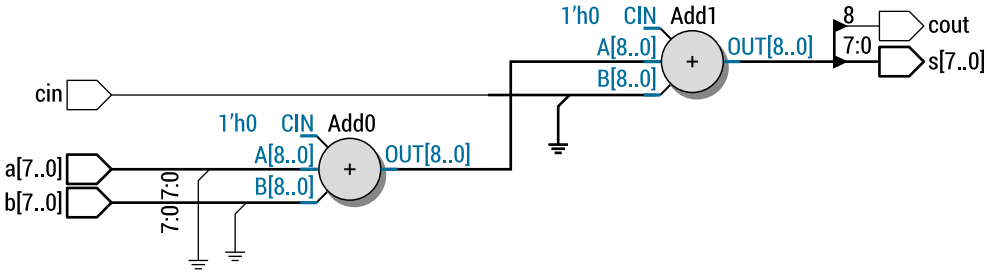
W następstwie obliczenia wartości wyrażenia i wykonania wszystkich przekształceń lewej części, w ogólnym przypadku zarówno po prawej jak i po lewej stronie znaku „=" operatora **assign** będą znajdować się wektory bitowe. W przypadku, gdy będą one posiadać jednakowy rozmiar, wartości bitów prawego wektora będą nadane bitom lewego wektora począwszy od najmniej znaczącego bitu. Jeżeli liczba bitów prawej strony jest większa niż liczba bitów lewej strony, wówczas starsze bity zostaną

odrzucone (utracone). Jeżeli liczba bitów lewej strony będzie większa niż liczba bitów prawej strony, to starsze bity lewej strony zostaną wypełnione zerami. Jako przykład na listingu 6.3 zaprezentowano opis 8-bitowego sumatora.

LISTING 6.3. Przykład 8-bitowego pełnego sumatora

```
module sum_8        (input [7:0] a, b,
                     input       cin,
                     output [7:0] s,
                     output      cout);

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

endmodule
```

Wynik syntezy przykładu z listingu 6.3 został przedstawiony na rysunku 6.3.



RYS. 6.3. Realizacja modułu *sum_8* z listingu 6.3: przykład 8-bitowego sumatora z wykorzystaniem operacji arytmetycznych

W następnym przykładzie do opisu ALU operator **assign** jest wykorzystywany wielokrotnie (listing 6.4). W danym przykładzie ALU wykonuje operacje dodawania i odejmowania na argumentach *a* i *b*. Dodatkowo ustalana jest wartość sygnału przeniesienia *c* oraz ustala się wartość dwóch flag: większa/mniejsza *gt* i wyniku zerowego *z*.

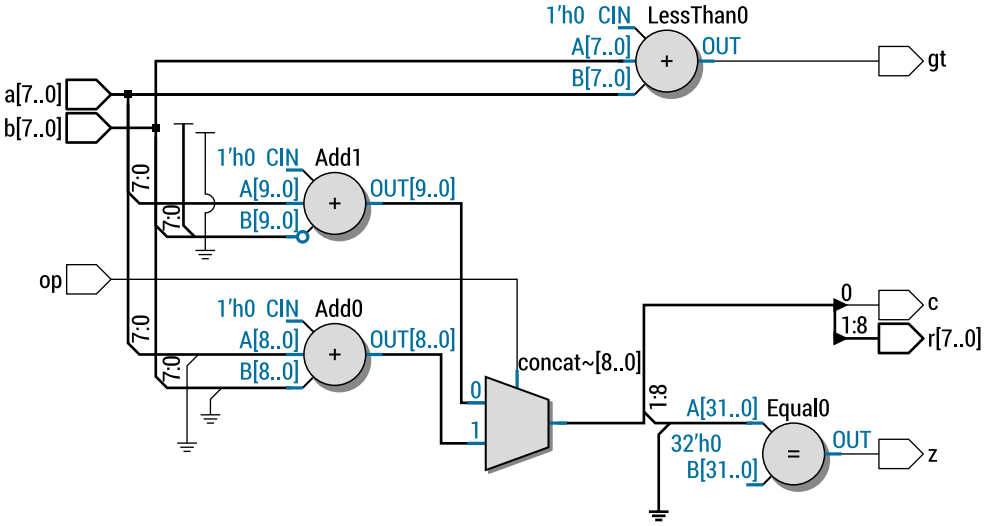
LISTING 6.4. Przykład prostej jednostki ALU

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

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

endmodule

Wynik syntezy przykładu z listingu 6.4 został przedstawiony na rysunku 6.4.



RYS. 6.4. Realizacja modułu *simple_ALU* z listingu 6.4: przykład wielokrotnego wykorzystania operatora **assign** z różnymi operacjami przy opisie układu kombinacyjnego

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

Zadanie. Przedstaw przykłady różnych sposobów wykorzystywania operatora **assign** w opisach projektów.

7. Operatory i bloki proceduralne

7.1. Operatory proceduralne initial i always, bloki proceduralne

W podobny sposób, jak języki programowania opisują algorytmy, język Verilog umożliwia opis funkcjonowania urządzeń cyfrowych, czyli tworzenie funkcjonalnych modeli projektu. Jedną z głównych cech języka Verilog jest możliwość określania rzeczywistego czasu pracy każdego elementu, co pozwala na tworzenie modeli systemów czasu rzeczywistego.

W języku Verilog istnieją tylko dwa *operatory proceduralne*, **initial** oraz **always**, za pomocą których można tworzyć *bloki proceduralne*. Blok proceduralny reprezentuje grupę operatorów zawartych w nawiasach operatorowych (**begin-end** lub **fork-join**).

Operatory wewnątrz bloku proceduralnego typu **initial** są wykonywane tylko jeden raz. Bloki typu **initial** zazwyczaj wykorzystywane są do określenia początkowych wartości zmiennych przed rozpoczęciem symulacji. Operatory wewnątrz proceduralnego bloku typu **always** wykonywane są stale w formie niekończącej się pętli: po wykonaniu ostatniego polecenia w danym bloku sterowanie jest przekazywane do pierwszej instrukcji bloku. Bloki typu **always** zazwyczaj wykorzystywane są do opisu funkcjonowania układów cyfrowych.

Uwaga. Jeżeli konstrukcje **initial** oraz **always** używane są bez nawiasów operatorowych, wówczas określane są mianem *operatorów* **initial** i **always**; natomiast jeżeli konstrukcje **initial** i **always** wykorzystuje się razem z nawiasami operatorowymi (**begin-end** czy **fork-join**), wówczas określane są jako *bloki* **initial** i **always**.

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

7.2. Operatory *begin-end* i *fork-join*

W języku Verilog istnieją dwa typy nawiasów operatorowych, określanych słowami kluczowymi **begin-end** i **fork-join**.

Przykłady bloków proceduralnych:

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

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

Linie kodu wewnątrz operatorów **begin-end** są wykonywane sekwencyjnie w takiej kolejności, w jakiej zostały one zapisane w kodzie źródłowym projektu. Czas początku wykonywania każdej instrukcji w grupie **begin-end** odpowiada czasowi zakończenia wykonywania poprzedniej instrukcji. Instrukcje wewnątrz grupy **fork-join** są wykonywane równolegle. Czas rozpoczęcia wykonania każdej operacji w grupie **fork-join** jest jednakowy i odpowiada czasowi rozpoczęcia wykonywania całej grupy. Jeżeli grupa składa się tylko z jednej instrukcji, wówczas nawiasy operatorowe nie są potrzebne i konstrukcje **initial** oraz **always** mogą być zapisywane bez operatorów **begin-end** czy **fork-join**.

Uwaga. Zazwyczaj nawiasy operatorowe **fork-join** nie są wspierane przez syntezatory.

Zadanie. Sprawdź, czy używany przez Ciebie kompilator wspiera wykorzystywanie operatorów **fork-join** podczas syntezy projektu.

7.3. Nazwane bloki proceduralne

Blokom proceduralnym można przypisać nazwy. Jeżeli blokowi zostanie przypisana nazwa, wówczas taki blok jest określany *nazwanym blokiem proceduralnym* (*named block*).

Przykłady nazwanych bloków proceduralnych:

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

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

Nazwane bloki proceduralne, w odróżnieniu od bloków bez nazwy, posiadają szereg dodatkowych cech:

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

7.4. Format bloków proceduralnych

Ogólnie bloki proceduralne mogą mieć następującą składnię:

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

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

Wartość *type_of_block* określa typ bloku proceduralnego. Może przyjmować jedno z następujących słów kluczowych: **initial** albo **always**.

Konstrukcja @ (*sensitivity_list*) nazywana jest *listą czułości*. Pozwala ona kontrolować różne zdarzenia czasowe (zmiana wartości jednego bądź kilku sygnałów, opadające bądź wzrastające zbocze sygnału i inne) służące do sterowania wykonywaniem operatorów zawartych w bloku proceduralnym. Bardziej szczegółowo format listy czułości zostanie opisany później.

Konstrukcje *statement_group* i *end_of_statement_group* – jest to jeden z nawiasów operatorowych **begin-end** lub **fork-join**. Jeżeli blok proceduralny zawiera tylko jedną instrukcję, wówczas nawiasy operatorowe nie są wymagane. W nazwanych blokach proceduralnych po słowie kluczowym **begin** albo **fork** po dwukropku występuje nazwa (*group_name*) bloku proceduralnego. Bloki proceduralne mogą także zawierać deklaracje zmiennych i parametrów lokalnych (*local_variable_declarations*), zakres działania których jest ograniczony obszarem bloku proceduralnego.

Następnie po deklaracji zmiennych lokalnych występują instrukcje bloku proceduralnego. Konstrukcja *time_control* wykorzystywana jest do sterowania czasem wykonywania następnej instrukcji. Konstrukcja *time_control* może znajdować się przed każdą instrukcją wewnątrz bloku proceduralnego.

Przykłady bloków proceduralnych:

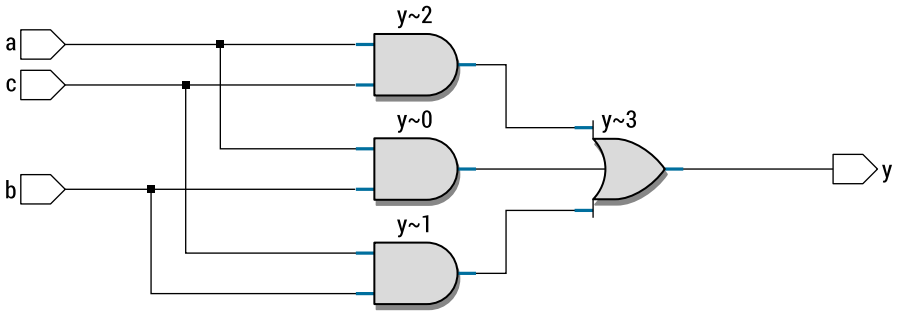
```
/* blok initial, jest wykonywany jeden raz, kolejno przypisuje początkowe wartości
zmiennej test_in z interwałem 5 jednostek czasu; nadanie blokowi nazwy test_loop
pozwala na deklarację wewnątrz bloku lokalnej zmiennej i
*/
initial
        begin: test_loop            // nadanie blokowi nazwy test_loop
            integer i;              // deklaracja lokalnej zmiennej i
            for (i=0; i<=15; i=i+1) // proceduralny operator cyklu for
            #5 test_in = i;         // przypisanie wartości co każde 5 jednostek czasu
        end

/* blok initial, również wykonywany tylko jeden raz, jednak nawiasy proceduralne
fork-join powodują jednoczesne wykonywanie instrukcji wewnątrz bloku; w rezulta-
cie czasy wykonania instrukcji są absolutne w stosunku do czasu rozpoczęcia wyko-
nywania danego bloku
*/
initial
    fork
        bus = 16'h0000;    // przypisanie wartości wykonywane z opóźnieniem 0
                           // jednostek czasu
                           // od rozpoczęcia wykonywania bloku initial
        #10 bus = 16'hC2AF;  // to samo, jednak z opóźnieniem 10 jednostek czasu
        #25 bus = 16'hD29A;  // to samo, jednak z opóźnieniem 25 jednostek czasu
    join

/* przykład bloku proceduralnego always, który sterowany jest za pomocą zmiany
dowolnego z sygnałów wejściowych
*/
always @ (a or b or cin) begin     // w danym przypadku nawiasy proceduralne
                                   // nie są wymagane
        s = a + b + cin;
end

/* przykład bloku proceduralnego always, sterowanego wzrastającym zboczem syg-
nału synchronizacji clk
*/
always @ (posedge clk)
        q = data;
```

Jako jeszcze jeden przykład można przedstawić opis układu kombinacyjnego kontroli większościowej dla trzech wejść, przedstawionego na listingu 7.1.

LISTING 7.1. Układ kontroli większościowej dla trzech wejść

```
module maj_3 (
        input a, b, c,              // opis wejść
        output reg y);              // opis wyjścia

        always @ (a, b, c)          // blok proceduralny z listą czułości
        begin                       // początek bloku proceduralnego
                y = (a & b) | (b & c) | (a & c);     // operator przypisania
        end                         // koniec bloku proceduralnego
endmodule
```

Blok **always**, opisujący zachowanie danego układu, wykorzystuje operator sterowania zdarzeniami @, który zawiera listę zmiennych (*a*, *b*, *c*) określaną jako listę czułości bloku **always**. Można więc powiedzieć, że blok **always** jest czuły na zmiany zmiennych *a*, *b* i *c*. Gdy występuje zdarzenie (zmiana wartości dowolnej z tych zmiennych), wówczas rozpoczyna się proces wykonywania bloku **always**. W wyniku tego zostanie wyliczone wyrażenie po prawej części operatora przypisania i jego wartość zostanie zapisana w zmiennej *y*. Owa zmienna przechowuje swoją wartość tak długo, dopóki nie wystąpi kolejne zdarzenie związane ze zmiennymi *a*, *b* lub *c*. W przedstawionym przykładzie blok proceduralny **always** zawiera tylko jedną instrukcję, dlatego nawiasy operatorowe **begin**-**end** można pominąć. Wynik syntezy przykładu z listingu 7.1 został przedstawiony na rysunku 7.1.



RYS. 7.1. Realizacja modułu *maj_3* z listingu 7.1: przykład wykorzystania bloku proceduralnego **always** do realizacji układu kombinacyjnego kontroli większościowej

8. Zarządzanie czasem

W języku Verilog opracowano trzy sposoby sterowania czasem wykonywania operacji, są to operatory opóźnienia #, czułości @ oraz oczekiwania **wait**.

8.1. Operator opóźnienia

Operator opóźnienia # w blokach proceduralnych posiada następującą składnię:

 # delay

gdzie *delay* – wielkość określająca liczbę jednostek czasu, o którą opóźnione jest wykonywanie kolejnej instrukcji (parametry jednostek czasu określa się za pomocą dyrektywy systemowej '**timescale**). Jako wielkość opóźnienia może występować liczba, zmienna bądź wyrażenie.

Przykłady opóźnień:

```
# 5     // opóźnienie o 5 jednostek czasu
# 3.7   // opóźnienie o 3,7 jednostek czasu
```

8.2. Operator czułości @

Operator czułości @ pozwala na zatrzymanie wykonywania kolejnej instrukcji (bądź grupy instrukcji) do momentu, gdy wystąpi chociażby jedno ze sprawdzanych zdarzeń. Takimi zdarzeniami mogą być zmiany wartości sygnału bądź rodzaj zbocza sygnału. Operator czułości przyjmuje następujące formaty składni:

@ (*edge signal* **or** *edge signal* **or** …)
@ (*edge signal*, *edge signal*, …)
@ (*)

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

Konstrukcja *edge* przed każdym z sygnałów nie jest obowiązkowa. Jeżeli przed sygnałem występuje jedno ze słów kluczowych **posedge** lub **negedge**, wówczas sprawdzane jest odpowiednie zbocze sygnału. Jeżeli przed sygnałem nie występuje konstrukcja *edge*, wówczas sprawdzana jest dowolna zmiana wartości sygnału.

Jako sygnały na liście czułości mogą występować sieci bądź zmienne dowolnego rozmiaru. Sygnały na liście czułości oddzielane są od siebie znakiem przecinka. Zamiast przecinka można stosować słowo kluczowe **or**. Jeżeli na liście czułości występuje tylko jeden sygnał, wówczas nawiasy okrągłe można pominąć. Znak gwiazdki w nawiasach okrągłych (*) oznacza, że lista czułości zawiera wszystkie rejestrowane sygnały, które są przetwarzane w danej instrukcji.

Przykłady operatora czułości:

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

8.3. Operator oczekiwania *wait*

Operator oczekiwania **wait** posiada następującą składnię:

wait (*expression*)

Dany operator zatrzymuje wykonywanie następnej instrukcji do momentu, gdy wartość wyrażenia *expression* będzie prawdą. Jako składowa *expression* może występować dowolne wyrażenie.

Przykłady wykorzystania operatora **wait**:

```
wire [7 : 0] addr;          // szyna adresowa
wait (addr != 8'hFA)        // sprawdzenie wartości szyny adresowej
data = mem [addr];          // podłączenie komórki pamięci mem spod adresu addr
                            // do szyny data
```

Zazwyczaj syntezatory nie wspierają operatorów opóźnienia # oraz operatorów oczekiwania **wait**. Jednak lista czułości w bloku proceduralnym **always** podczas syntezy projektu powinna być wspierana przez wszystkie kompilatory. Dlatego w dalszej części rozdziału zostanie omówiona bardziej dokładnie konstrukcja listy czułości oraz rola, jaką pełni podczas syntezy układów kombinacyjnych oraz sekwencyjnych.

8.4. Lista czułości

Lista czułości służy do zatrzymania (opóźnienia) wykonywania bloku proceduralnego do momentu, gdy wystąpi zdarzenie związane ze zmianą wartości sygnałów wejściowych.

Uwaga. Przez sygnały wejściowe rozumie się te sygnały, które są wejściowymi dla danego bloku proceduralnego bądź instrukcji.

Najczęściej listę czułości wykorzystuje się w operatorze proceduralnym **always**, w takim przypadku lista czułości posiada następującą składnię:

always @ (*signal, signal*, …) *statement*;

gdzie **always** – operator proceduralny, w którym występuje lista czułości; **@** – znak specjalny poprzedzający listę czułości; *signal, signal*, … – lista sygnałów (w nawiasach okrągłych), których zmiana powoduje wznowienie wykonywania operatora proceduralnego; *statement* – instrukcja (lub grupa instrukcji), której wykonywanie jest zatrzymywane daną listą czułości.

Uwaga. Jeżeli wewnątrz bloku jest kilka instrukcji, wówczas powinny być one zawarte wewnątrz konstrukcji **begin-end** lub **fork-join**.

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

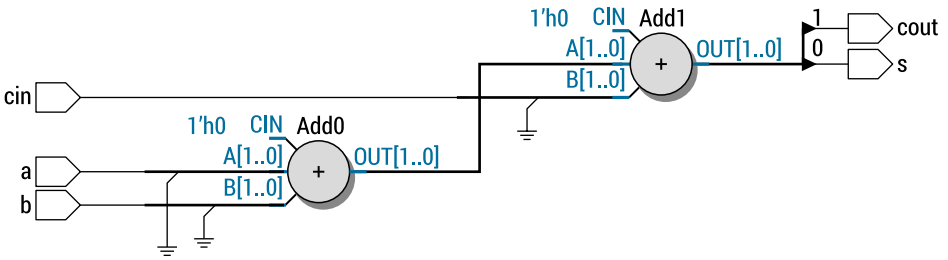
Zaprezentowany format składni zazwyczaj wykorzystywany jest do opisu funkcjonowania układów czułych na poziomy sygnałów. Typowym przykładem takiego schematu jest zatrzask (*latch*) lub transparentny przerzutnik.

Przykłady wykorzystywania listy czułości przedstawione są na listingach 8.1 oraz 8.2.

LISTING 8.1. Wariacja sumatora z listą czułości

```
module adder_sen_list_1      (input a, b, cin,
                              output reg s, cout);
        always @ (a, b, cin)
                {cout,s} = a + b + cin;
endmodule
```

Realizacja przykładu z listingu 8.1 została pokazana na rysunku 8.1.



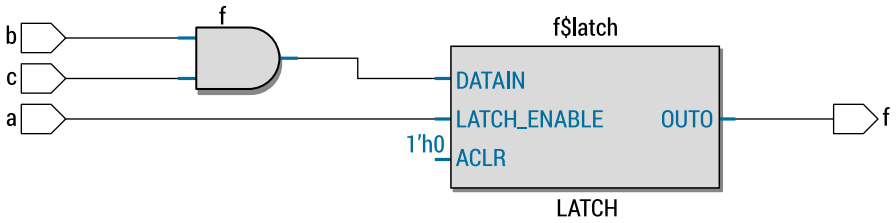
RYS. 8.1. Realizacja modułu *adder_sen_list_1* z listingu 8.1: wykorzystanie bloku proceduralnego **always** z listą czułości do opisu jednobitowego sumatora

LISTING 8.2. Schemat kombinacyjny z zatrzaskiem na wyjściu

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

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

Zadanie. Przeanalizuj kod z listingu 8.2, a następnie spróbuj odpowiedzieć na następujące pytanie: dlaczego syntezator na wyjściu schematu umieści zatrzask?

Wynik syntezy przykładu z listingu 8.2 został przedstawiony na rysunku 8.2.



RYS. 8.2. Realizacja modułu *circ_latch* z listingu 8.2: powstawanie zatrzasku na wyjściu w przypadku błędu opisu schematu kombinacyjnego

8.5. Lista czułości w układach kombinacyjnych

Jeżeli przy opisie układów kombinacyjnych na liście czułości nie będzie występować chociażby jeden z sygnałów wejściowych, wówczas syntezator automatycznie na wyjściu schematu umieści przerzutnik.

Uwaga. *Syntezator* – jest to program (część kompilatora), który wykonuje syntezę układów zgodnie z ich opisem w języku Verilog.

Aby do tego nie dopuścić przy opisie układów kombinacyjnych, na liście czułości powinny być wymienione wszystkie sygnały wejściowe. Jednakże w praktyce zasada ta jest często pomijana przez projektantów, na przykład podczas nanoszenia zmian w projekcie dodatkowe sygnały mogą zostać dodane do wyrażenia, jednak pominięte na liście czułości.

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

@ * lub @ (*).

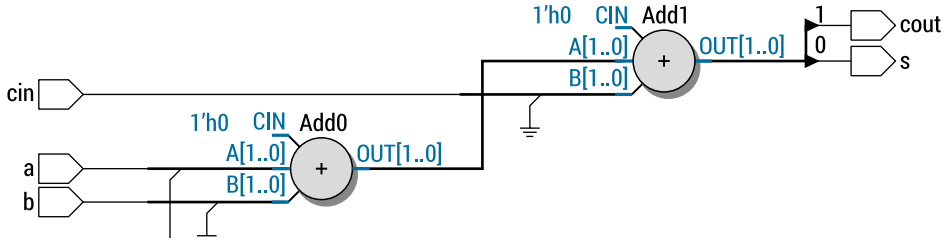
Obydwa formaty są równoznaczne. Gwiazdka występująca po znaku @ (w nawiasach lub bez) oznacza, że lista czułości zawiera wszystkie sygnały wejściowe danego operatora proceduralnego. Powyższe formaty listy czułości są wykorzystywane do opisu układów kombinacyjnych.

Przykłady wykorzystywania listy czułości ze składnią @ (*) przedstawione są na listingach 8.3 oraz 8.4.

LISTING 8.3. Kolejny przykład sumatora z listą czułości

```
module adder_sen_list_2      (input a, b, cin,
                              output reg s, cout);
        always @ (*)
                {cout,s} = a + b + cin;
endmodule
```

Realizacja przykładu z listingu 8.3 została przedstawiona na rysunku 8.3.

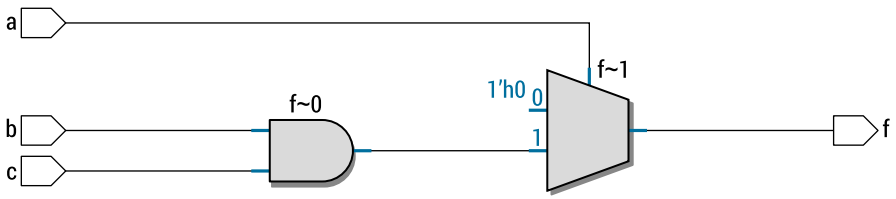


RYS. 8.3. Realizacja modułu *adder_sen_list_2* z listingu 8.3

LISTING 8.4. Odmiana układu bez zatrzasku na wyjściu, czyli układu kombinacyjnego

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

Realizacja z listingu 8.4 została przedstawiona na rysunku 8.4.



RYS. 8.4. Realizacja modułu *circ_no_latch* z listingu 8.4: rozwiązanie błędu przy opisie układu kombinacyjnego – zatrzask na wyjściu nie występuje

Zadanie. Przeanalizuj listing 8.4 i spróbuj odpowiedzieć na następujące pytanie: dlaczego syntezator na wyjściu układu nie umieścił zatrzasku?

8.6. Lista czułości w układach sekwencyjnych

Układy sekwencyjne (*sequential*) sterowane są za pomocą przełączania bądź za pomocą zboczy sygnałów. Do takich układów należą przerzutniki, rejestry, liczniki, automaty skończone, kontrolery itp. Do sterowania czasem przy modelowaniu takich urządzeń wykorzystuje się następujący format listy czułości:

always @ (**posedge** *signal*, **negedge** *signal*, …)

Dla każdego sygnału na liście czułości może być określone albo narastające (**posedge**), albo opadające (**negedge**) zbocze, jednak nie odbywa się to jednocześnie dla tego samego sygnału. Oprócz tego, typ zbocza sygnału powinien być określony dla każdego sygnału znajdującego się na tej liście. Innymi słowy, na tej samej liście sygnałów nie można mieszać sterowania poziomami i zboczami sygnałów.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator umożliwia:

- sterowanie zarówno za pomocą narastającego, jak i opadającego zbocza sygnału dla tego samego sygnału;
- wykorzystywanie na tej samej liście czułości sterowania za pomocą zboczy i poziomów dla różnych sygnałów;
- wykorzystywanie na jednej liście czułości sterowania za pomocą zboczy i poziomów dla tych samych sygnałów.

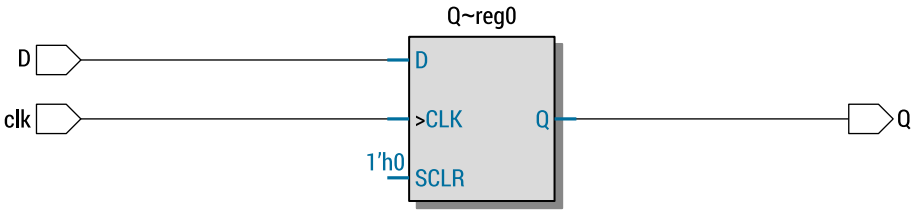
Przykład wykorzystywania listy czułości w systemach sekwencyjnych:

LISTING 8.5. Wariant realizacji przerzutnika typu D

```
module my_DFF_no_blocking (input D, clk,
                           output reg Q);
```

```
    always @ (posedge clk)
        Q <= D;                    // operator przypisania nieblokującego
endmodule
```

Realizacja przykładu z listingu 8.5 została przedstawiona na rysunku 8.5.

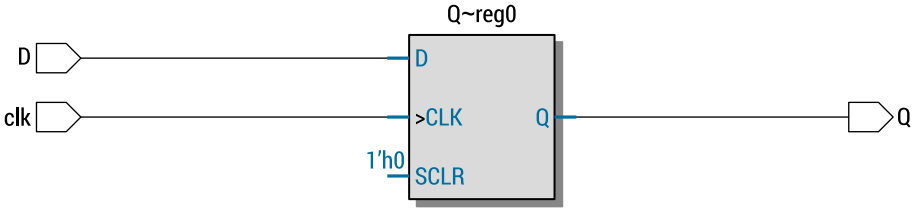


RYS. 8.5. Realizacja modułu *my_DFF_no_blocking* z listingu 8.5: przykład realizacji przerzutnika D za pomocą nieblokującego operatora przypisania

LISTING 8.6. Wariant realizacji przerzutnika typu D

```
module my_DFF_blocking (input D, clk,
                        output reg Q);

    always @ (posedge clk)
        Q = D;                     // operator przypisania blokującego
endmodule
```

Realizacja przykładu z listingu 8.6 została przedstawiona na rysunku 8.6.



RYS. 8.6. Realizacja modułu *my_DFF_blocking* z listingu 8.6: przykład realizacji przerzutnika D za pomocą operatora przypisania blokującego

Uwaga. Różnica pomiędzy operatorami przypisania blokującego i nieblokującego zostanie wyjaśniona w następnym rozdziale.

9. Operatory przypisania

9.1. Cechy wspólne

Podstawowym operatorem w blokach proceduralnych jest operator przypisania. W językach programowania zazwyczaj występuje tylko jeden operator przypisania, najczęściej oznaczany znakiem równości (=). Ogólnie operator przypisania w języku Verilog służy do przypisania konkretnych wartości zmiennym. Jednak podczas opisu urządzeń i układów cyfrowych taka funkcja bywa niewystarczająca. Dlatego też, aby umożliwić bardziej dokładny opis zachowania urządzeń cyfrowych, w języku Verilog występuje kilka operatorów przypisania, które różnią się od siebie zarówno podczas symulacji jak i podczas syntezy.

W języku Verilog operatory przypisania dzielą się przede wszystkim na: blokujące (*blocking*), oznaczane symbolem **=,** oraz nieblokujące (*non-blocking*), oznaczane symbolem <=. Typ operatorów przypisania może wpływać nie tylko na wynik symulacji, ale także na wynik syntezy projektu.

Kolejną ważną kwestią jest wykonywanie operatorów przypisania w relacji do operatorów zarządzania czasem (*timing_control*): opóźnienia **#**, czułości **@** i oczekiwania **wait**. W Verilogu operatory zarządzania czasem mogą być umieszczane zarówno przed operatorami przypisania, jak i wewnątrz nich – przed wyrażeniem z prawej części operatora przypisania. W ostatnim przypadku operator zarządzania czasem jest nazywany *wewnętrznym opóźnieniem* (*intra-assignment delay*). W przypadku wykorzystywania operatorów przypisania do symulacji projektu należy także zwrócić uwagę na nawiasy operatorowe: **begin-end** lub **fork-join**.

Oprócz tego, w języku Verilog istnieją dwie pary operatorów **assign-deassign** oraz **force-release** służące do wykonywania specjalnych przypisań. Operator **assign** w blokach proceduralnych jest nazywany *proceduralnym operatorem przypisania ciągłego*, jego działanie różni się od działania zwykłego operatora przypisania ciągłego **assign**. Operator **force** umożliwia nadawanie zmiennym lub sieciom dowolnego typu danych.

Użytkownicy języka Verilog powinni bardzo dobrze rozumieć działanie operatorów przypisania, zarówno podczas syntezy jak i symulacji projektu. Dlatego w niniejszym rozdziale zostaną dokładnie omówione cechy operatorów przypisania języka Verilog.

9.2. Operator przypisania blokującego „=”

9.2.1. Składnia

Operator przypisania blokującego posiada następującą składnię:

variable = *expression*;

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

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

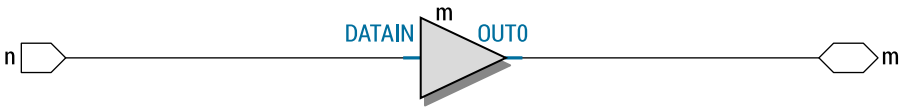
Operator przypisania blokującego działa w następujący sposób. Gdy symulator napotka w kodzie dany operator, wówczas wykonywane jest obliczenie wyrażenia *expression,* a jego obliczona wartość zostaje przypisana zmiennej *variable*. Wewnątrz bloku **begin-end** sekwencyjnie wykonywane instrukcje są zatrzymywane do momentu zakończenia operacji przypisania. Stąd też wywodzi się nazwa operatora przypisania blokującego (*blocking*).

W przykładzie z listingu 9.1 pierwsze przypisanie przekazuje wartość sygnału z wyprowadzenia *n* na wyprowadzenie *m*, przy czym blokuje się wykonanie kolejnej instrukcji przypisania. Następnie kolejne przypisanie przekazuje wartość sygnału z wyprowadzenia *m* na wyprowadzenie *n*. Dzięki nawiasom operatorowym **begin-end** cykl ten jest powtarzany w nieskończoność.

LISTING 9.1. Pierwsze przypisanie zmienia *m*, następnie drugie przypisanie zmienia *n*

```
module blocking_in_out (inout reg n, m);
        always begin
                m = n;
                n = m;
        end
endmodule
```

Wynik syntezy opisu z listingu 9.1 przedstawiono na rysunku 9.1.



RYS. 9.1. Realizacja modułu *blocking_in_out* z listingu 9.1: przykład przypisania sygnałów w bloku **always** za pomocą operatora przypisania blokującego

Uwaga. Analizując rysunek 9.1, można przypuszczać, że w danym przykładzie syntezator realizuje tylko pierwszy operator przypisania.

Poniżej zaprezentowano jeszcze kilka przykładów modułów z wykorzystaniem operatorów przypisania blokującego.

LISTNG 9.2. Inwersja wejść

```
module not_inputs(
        input a, b, c, d,
        output reg w, x, y, z);

        always begin
                w =~a;
                x =~b;
                y =~c;
                z =~d;
        end
endmodule
```

LISTING 9.3. Połączenia wejść za pomocą bramek

```
module gating_outputs(
        input a, b, c, d,
        output reg w, x, y, z);

        always begin
                w = a & b;
                x= b | c;
                y= c ^ d;
                z= d ~^ a;
        end
endmodule
```

LISTING 9.4. Sekwencyjne łączenie wejść za pomocą bramek

```
module gating_in_out_1(
        input a, b, c, d,
        output reg w, x, y, z);

        always begin
```

```
            w = a & b;
            x = w | c;
            y = x ^ d;
            z = y ~^ a;
        end
endmodule
```

LISTING 9.5. Kolejny przykład sekwencyjnego łączenia za pomocą bramek

```
module gating_in_out_2(
        input a, b, c, d,
        output reg z);

        reg w, x, y;
        always begin
            w = a & b;
            x = w | c;
            y = x ^ d;
            z = y ~^ a;
        end
endmodule
```

LISTNG 9.6. Połączenie kilku wyjść

```
module connect_outs(
        input a, b, c, d,
        output reg y, z);

        always begin
            y =!a;
            y =!b;
            z =!c;
            z =!d;
        end
endmodule
```

LISTING 9.7. Opis prostego układu kombinacyjnego

```
module comb_circuit(
        input data,
        output reg c, d);
```

```
        always @(*)
        begin
                c = data;
                d = ~c & ~data;
        end
endmodule
```

Zadania.

1) W przedstawionych wyżej przykładach postaraj się przewidzieć wynik syntezy.
2) Wykonaj kompilację przedstawionych wyżej przykładów, a następnie przeanalizuj wyniki syntezy na poziomie przesłań międzyrejestrowych (RTL). Wyjaśnij otrzymane wyniki. Czy otrzymane wyniki potwierdziły przypuszczenia z zadania 1?

Uwaga. Wyniki syntezy przykładów z listingów 9.2-9.7 zostały umieszczone na końcu niniejszego rozdziału.

9.2.2. Zarządzanie czasem

Jak wspomniano wcześniej, operatory zarządzania czasem (**#**, **@**, **wait**) mogą znajdować się zarówno przed operatorem przypisania, jak i wewnątrz niego. W pierwszym przypadku operator blokującego przypisania przyjmuje następującą postać:

timing_control variable = *expression*;

gdzie *timing_control* – jeden z operatorów zarządzania czasem **#**, **@** lub **wait**; *variable* – zmienna typu **reg**; *expression* – pewne wyrażenie.

Jeżeli symulator napotyka w kodzie podobną konstrukcję, wówczas obliczanie wartości wyrażenia *expression* jest zatrzymywane zgodnie z czasem określonym za pomocą operatora zarządzania czasem *timing_control*.

Jako przykład przeanalizujmy następujący fragment kodu:

```
# 10                x = a;
@(posedge clk)      y = b;
```

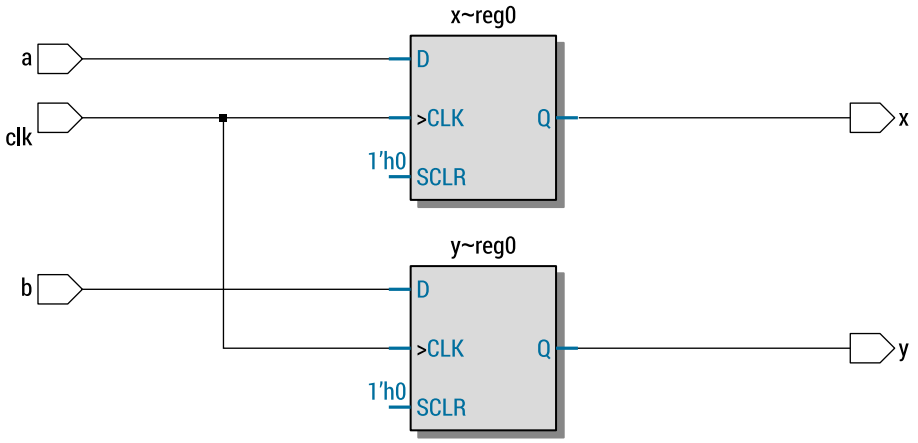
Gdy symulator napotyka pierwszą linię kodu, wówczas będzie wykonywane opóźnienie o 10 jednostek czasu, a dopiero po tym zmiennej *x* zostanie nadana wartość zmiennej *a*. Gdy symulator przejdzie do kolejnej linii, będzie wykonywane oczekiwanie na narastające zbocze sygnału *clk*, a dopiero w wyniku jego wystąpienia zmiennej *y* będzie nadana wartość zmiennej *b*. Wspólną cechą tych operatorów jest to, że przed wykonaniem przypisania wykonywane jest wstrzymanie albo na określoną ilość jednostek czasu, albo do czasu wystąpienia określonego zdarzenia. Innymi słowy, wartość wyrażenia z prawej strony znaku operatora „=" nie jest wyliczana do momentu, gdy skończy się czas oczekiwania.

Na listingu 9.8 została przedstawiona realizacja omówionego fragmentu kodu.

LISTING 9.8. Operator blokującego przypisania wykorzystywany razem z operatorami zarządzania czasem

```
module blocking_st (
        input a, b, clk,
        output reg x, y);

        always begin
                # 10                    x = a;
                @(posedge clk)          y = b;
        end
endmodule
```

Układ z listingu 9.8 zbudowany przez syntezator został przedstawiony na rysunku 9.2.



RYS. 9.2. Realizacja modułu *blocking_st* z listingu 9.8: przykład wykorzystania operatora blokującego przypisania wspólnie z operatorami zarządzania czasem

Jak wytłumaczyć wynik pracy syntezatora? Na początku syntezator ignoruje operator opóźnienia #10. Na pierwszy rzut oka może się wydawać, że wyjście *x* w układzie z rysunku 9.2 powinno być podłączone bezpośrednio z wejściem *a*. Jednak należy pamiętać, że blok **always** wykonywany jest w postaci niekończącej się pętli. Dlatego też kolejne i wszystkie następne przypisania zmiennej *x* wartości wejścia *a* nastąpią nie wcześniej, niż przyjdzie rosnące zbocze sygnału synchronizacyjnego *clk*. Stąd też w układzie występuje przerzutnik pomiędzy wejściem *a* a wyjściem *x*, sterowany rosnącym zboczem sygnału synchronizacyjnego *clk*.

9.2.3. Opóźnienie wewnętrzne

Jeżeli wewnątrz operatora przypisania znajduje się operator zarządzania czasem, wówczas takie opóźnienie nazywane jest *wewnętrznym* (*intra-assignment delay* lub *intra-assignment timing control*). Operator przypisania blokującego z wykorzystaniem wewnętrznego opóźnienia przyjmuje następujący format:

variable = *timing_control expression*;

Powyższa konstrukcja działa w następujący sposób. W danej jednostce czasu, gdy symulator spotyka w kodzie operator przypisania, zostaje obliczone wyrażenie *expression*, natomiast samo przypisanie zmiennej *variable* wartości wyliczonego wyrażenia następuje w momencie upłynięcia czasu opóźnienia zadeklarowanego za pomocą operatora *timing_control*. Wewnątrz bloku **begin-end** sekwencyjnie wykonywane instrukcje zostają wstrzymane do momentu, gdy nie zostanie zakończone przypisanie (po upłynięciu czasu opóźnienia).

Jako przykład przeanalizowano następujący fragment kodu:

```
x = # 10 a;
y = @(posedge clk) b;
```

Gdy symulator napotyka pierwszy operator przypisania, wówczas na początku zostanie wyliczona wartość zmiennej *a*. Następnie zostanie wykonana operacja opóźnienia o 10 jednostek czasowych. W tym momencie wszystkie wykonywane procesy (czyli następne instrukcje) będą zablokowane. Po zakończeniu czasu opóźnienia, zmiennej *x* będzie nadana obliczona wcześniej wartość. Działanie tej instrukcji można zaprezentować za pomocą następującego fragmentu kodu:

```
begin
        aTemp = a;          /* obliczenie wyrażenia a i przypisanie wartości
                               tymczasowej zmiennej aTemp */
        # 10 x = aTemp;     /* opóźnienie o 10 jednostek czasu i przypisanie
                               wartości aTemp do zmiennej x */
end
```

Gdy symulator przejdzie do kolejnej instrukcji, wówczas na początku zostanie wyliczona wartość wyrażenia *b*. Następnie układ będzie oczekiwał na zdarzenie wystąpienia narastającego zbocza sygnału *clk*. Po wystąpieniu tego zdarzenia zmiennej *y* zostanie nadana wartość wyliczonego wcześniej wyrażenia. Działanie drugiej instrukcji można zaprezentować za pomocą poniższego kodu:

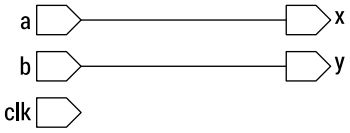
```
begin
        bTemp = b;                  /* obliczenie wyrażenia b i zapis jego wartości
                                    do zmiennej tymczasowej bTemp */
        @(posedge clk) y = bTemp;   /* oczekiwanie na rosnące zbocze sygnału clk
                                    i przypisanie wartości bTemp do zmiennej y */
end
```

Na listingu 9.9 została pokazana realizacja przeanalizowanego fragmentu kodu.

LISTING 9.9. Operator blokującego przypisania z wewnętrznym opóźnieniem

```
module blocking_st_intra (
        input a, b, clk,
        output reg x, y);

        always begin
                x = # 10 a;
                y = @(posedge clk) b;
        end
endmodule
```

Układ z listingu 9.9 zrealizowany przez syntezator został przedstawiony na rysunku 9.3.



RYS. 9.3. Realizacja modułu *blocking_st_intra* z listingu 9.9: przykład wykorzystania operatora blokującego przypisania z wewnętrznym opóźnieniem

Uwaga. Na podstawie rysunku 9.3 można dojść do wniosku, że syntezator ignoruje wewnętrzne opóźnienia dla operatora blokującego przypisania.

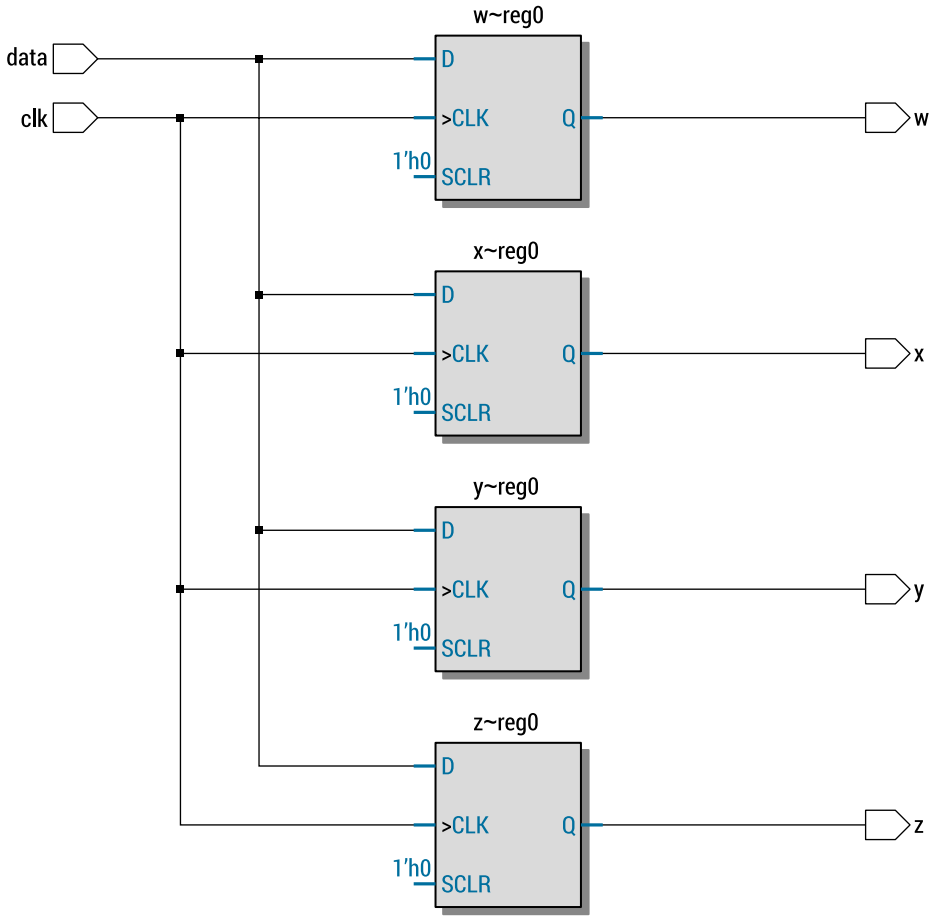
9.2.4. Cechy syntezy

Poniżej zaprezentowano i przeanalizowano przykład wykorzystania operatora przypisania blokującego.

LISTING 9.10. Równoległa realizacja przerzutników za pomocą operatorów blokującego przypisania

```
module blocking_assignments (
        input data, clk,
        output reg x, y, w, z);

        always @(posedge clk)
        begin
                x = data;
                y = x;
                w = y;
                z = w;
        end
endmodule
```

Zgodnie z opisem kodu z listingu 9.10 spróbujmy określić wynik syntezy. Na początku można założyć, że wartości zmiennych *x, y, w* i *z* będą formowane na wyjściach przerzutników, które są sterowane rosnącym zboczem sygnału synchronizacyjnego *clk*, jako że cały blok jest sterowany za pomocą operatora czułości **@** (**posedge** clk). Pierwsza instrukcja przypisania nada zmiennej *x* wartość wejścia *data*, przy czym zablokuje wykonywanie wszystkich następnych instrukcji przypisania. Druga instrukcja przypisania powinna określić wartość zmiennej *x* i przypisać ją do zmiennej *y*, jednak wartość zmiennej *x* jest już ustalona i jest równa wartości zmiennej *data*. Idąc podobnym tokiem rozumowania można dojść do wniosku, że wszystkim zmiennym *x, y, w* i *z* powinna być przypisana wartość tej samej zmiennej wejściowej *data*. W ten sposób schemat opisany na listingu 9.10 realizowany jest za pomocą czterech przerzutników, na wyjściach których formowane są wartości zmiennych *x, y, w* i *z*, a na wejścia wszystkich przerzutników podawana jest wartość wejściowego sygnału *data* (rysunek 9.4).



RYS. 9.4. Realizacja modułu *blocking_assignments* z listingu 9.10: przykład przypisania sygnałów w bloku **always**, czułego na zbocze sygnału za pomocą operatorów blokującego przypisania

9.3. Operator przypisania nieblokującego „<=”

9.3.1. Składnia

Operator przypisania nieblokującego posiada następującą składnię:

variable <= *expression*;

gdzie *expression* – pewne wyrażenie; *variable* – zmienna (typu **reg**), której będzie nadana wartość wyliczona z wyrażenia.

Działanie operatora nieblokującego przypisania jest następujące. W momencie, gdy symulator napotka dany operator, wówczas wykonywane jest obliczenie wyrażenia *expression*, jednak nadanie otrzymanej wartości zmiennej z lewej strony równania zostaje odłożone do czasu zakończenia czasu symulacji. W bloku **begin-end** sekwencyjne wykonywanie operatorów nie zostaje zakłócone (nie są blokowane), dlatego też operator ten nosi nazwę przypisania nieblokującego (*non-blocking*). Innymi słowy, następna instrukcja będzie wykonywana bez oczekiwania na zakończenie przypisania.

Przykład wykorzystania operatorów nieblokującego przypisania przedstawia listing 9.11.

LISTING 9.11. Obydwa przypisania będą wykorzystane do czasu zmiany wartości zmiennych *n* i *m*

```
module non_blocking_in_out
        (inout reg n, m);

        always begin
                m <= n;
                n <= m;
        end
endmodule
```

W pokazanym przykładzie można oczekiwać, że dane z dwukierunkowego wyprowadzenia *n* będą przekazywane na dwukierunkowe wyprowadzenie *m* i jednocześnie z wyprowadzenia *m* – na wyprowadzenie *n*. Jednak syntezator odmawia realizacji opisanego rozwiązania, ponieważ jednoczesny przekaz dwóch sygnałów w różnych kierunkach jedną linią fizycznie nie jest możliwy.

9.3.2. Zarządzanie czasem

Operator przypisania nieblokującego, w przypadku wykorzystania jednego z operatorów zarządzania czasem (**#**, **@**, **wait**), posiada następującą składnię:

timing_control variable <= *expression*;

gdzie *timing_control* – jeden z operatorów zarządzania czasem **#**, **@** albo **wait**; *variable* – zmienna typu **reg**; *expression* – pewne wyrażenie.

Gdy symulator napotyka w kodzie daną konstrukcję, to najpierw wprowadza opóźnienie określone za pomocą operatora *timing_control*, a dopiero po upływie opóźnienia oblicza wartość wyrażenia *expression* i przypisuje ją do zmiennej *variable*.

Jako przykład przeanalizowany zostanie następujący fragment kodu:

```
# 10                    x <= a;
@(posedge clk)          y <= a & b;
```

Gdy symulator napotka pierwszą linię tego kodu, na początku wykona opóźnienie o 10 jednostek czasu, a dopiero potem zmiennej *x* zostanie nadana wartość zmiennej *a*. Przy czym wykonywanie kolejnych instrukcji nie jest blokowane. W drugiej linii kodu oczekuje się na wystąpienie zdarzenia rosnącego zbocza sygnału *clk*, po czym zmiennej *y* zostanie nadana wartość obliczonego wyrażenia *a* & *b*.

Na listingu 9.12 została zaprezentowana realizacja analizowanego fragmentu kodu.

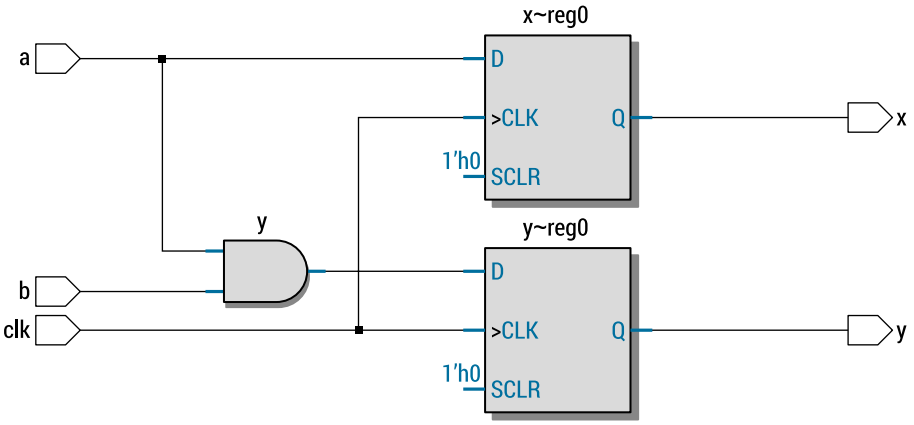
LISTING 9.12. Operator przypisania nieblokującego razem z operatorami zarządzania czasem

```
module non_blocking_st (
        input a, b, clk,
        output reg x, y);

always begin
                # 10 x <= a;
                @(posedge clk) y <= a & b;
        end
endmodule
```

Analogicznie do analizy przykładu z listingu 9.8, można dojść do następujących wniosków odnośnie postaci syntezowanego schematu:

- wartości zmiennych *x* i *y* będą obecne na wyjściach przerzutników, które są sterowane rosnącym zboczem sygnału *clk*;
- wejście przerzutnika, na którym generowana jest wartość zmiennej *y*, będzie poprzedzać układ kombinacyjny odpowiadający przedstawionemu wcześniej opisowi.

Układ opisany w listingu 9.12 zrealizowany przez syntezator został przedstawiony na rysunku 9.5.



RYS. 9.5. Realizacja modułu *non_blocking_st* z listingu 9.12: przykład wykorzystania operatora przypisania nieblokującego razem z operatorami zarządzania czasem

9.3.3. Opóźnienie wewnętrzne

Operator przypisania nieblokującego w przypadku wykorzystania wewnętrznego opóźnienia przyjmuje następującą składnię:

variable <= *timing_control expression*;

Przedstawiona konstrukcja działa w następujący sposób. W momencie gdy symulator napotyka instrukcję przypisania, obliczane jest wyrażenie *expression*, natomiast nadanie wyliczonej wartości zmiennej *variable* następuje po upłynięciu czasu opóźnienia wskazanego za pomocą operatora zarządzania czasem *timing_control*. W bloku **begin-end** sekwencyjne wykonywanie instrukcji nie zostaje zaburzone.

Uwaga. Za pomocą tej składni symulowane są *opóźnienia przekazywania danych*.

Jako przykład przeanalizowano następujący fragment kodu:

```
x <= # 10 a;
y <= @(posedge clk) a & b;
```

Gdy symulator natrafi na pierwszą instrukcję, na początku zostanie obliczone wyrażenie *a,* a jego wartość zostanie zapamiętana. Dany proces nie jest blokowany – kontynuowane jest wykonywanie następnej instrukcji. Następnie po upływie 10 jednostek czasu zmiennej *x* zostanie nadana wartość obliczonego wyrażenia.

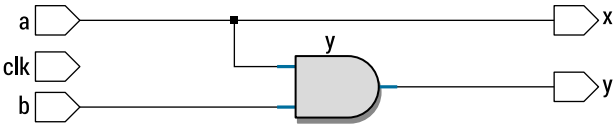
Na listingu 9.13 zademonstrowano realizację rozpatrzonego fragmentu kodu.

LISTING 9.13. Operator przypisania nieblokującego z wewnętrznymi opóźnieniami

```
module non_blocking_st_intra(
        input a, b, clk,
        output reg x, y);
```

```
        always begin
                x <= # 10 a;
                y <= @(posedge clk) a & b;
        end
endmodule
```

Układ opisany na listingu 9.13 zrealizowany przez syntezator został przedstawiony na rysunku 9.6.



RYS. 9.6. Realizacja modułu *non_blocking_st_intra* z listingu 9.13: przykład wykorzystania operatora przypisania nieblokującego z wewnętrznym opóźnieniem

Uwaga. Na podstawie rysunku 9.6 można wywnioskować, że syntezator ignoruje występowanie wewnętrznych opóźnień dla operatora przypisania nieblokującego.

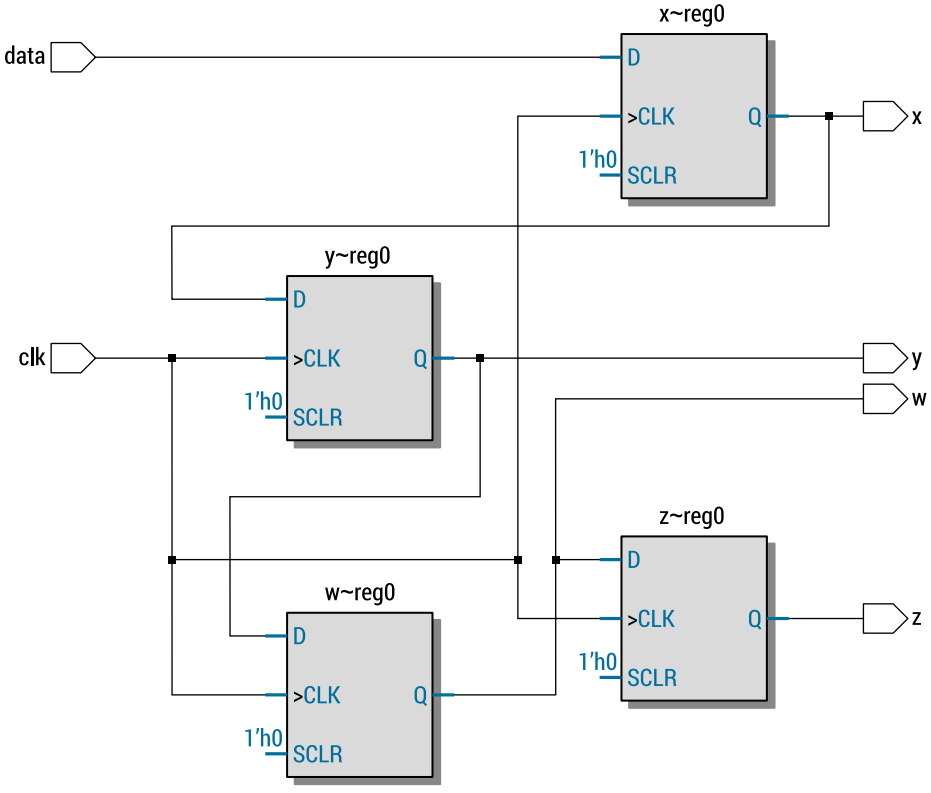
9.3.4. Cechy syntezy

Przeanalizujemy przykład wykorzystania operatora przypisania nieblokującego z listingu 9.14 (odpowiednik listingu 9.10).

LISTING 9.14. Sekwencyjna realizacja przerzutników za pomocą operatorów przypisania nieblokującego

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

Spróbujmy przewidzieć rezultat syntezy opisu z listingu 9.14. Na początku można się spodziewać, że wartości zmiennych *x*, *y*, *w* i *z* będą określone na wyjściach przerzutników, które są sterowane rosnącym zboczem sygnału synchronizacyjnego *clk*, ponieważ cały blok sterowany jest za pomocą operatora czułości @ (**posedge** clk). Pierwsza instrukcja przypisania nadaje wartość wejścia *data* zmiennej *x*, przy czym pozostałe instrukcje nie są blokowane. Druga instrukcja przypisania powinna nadać zmiennej *y* wartość zmiennej *x*, jednak wartość zmiennej *x* nie jest jeszcze znana, gdyż dopiero w kolejnym takcie po przyjściu rosnącego zbocza sygnału *clk* zostanie ona zaktualizowana. Dlatego wejście przerzutnika, na którym formowany jest sygnał zmiennej *y*, powinno być podłączone do wyjścia przerzutnika, który zwraca wartość zmiennej *x*. W podobny sposób można dojść do wniosku, że przerzutniki, na wyjściach których ustalane są wartości zmiennych *x*, *y*, *w* i *z*, powinny być połączone sekwencyjnie, jak to zostało zaprezentowane na rysunku 9.7.



RYS. 9.7. Realizacja modułu *non_blocking_assignments* z listingu 9.14: przykład przypisania sygnałów w bloku **always**, czułym na zbocze sygnału synchronizacyjnego, za pomocą operatorów przypisania nieblokującego (porównaj dany przykład z rysunkiem 9.4)

Podsumowując wykorzystanie operatorów przypisania, należy zauważyć, że syntezator zazwyczaj ignoruje operatory zarządzania czasem w danych przypisaniach. Wyjątek stanowi operator listy czułości @, który sprawdza zbocze sygnału synchronizacji. Gdy ten operator steruje blokiem instrukcji zawierającym operatory przypisania blokującego, utworzony zostanie układ równoległych elementów pamięci, a w przypadku operatorów nieblokujących – schemat sekwencyjnie połączonych elementów pamięci.

Uwaga. Aby uniknąć potencjalnych wyścigów w syntezowanym układzie oraz w celu uzyskania wiarygodnych danych podczas symulacji z zerowym czasem opóźnienia, należy operatora przypisania blokującego (=) używać do opisu układów kombinacyjnych, a operatora przypisania nieblokującego (<=) wykorzystywać do opisu układów sekwencyjnych.

9.4. Zarządzanie czasem w operatorach proceduralnych podczas symulacji

Operatory zarządzania czasem w operacjach przypisania (zarówno w blokujących, jak i w nieblokujących) najczęściej są wykorzystywane do opisu modułów generowania wartości testowych oraz do wskazania momentu przypisania niektórych wartości zmiennym. Przy czym wykonywanie procesu może być zatrzymywane (dla blokujących operatorów przypisania) lub też niezatrzymywane (dla nieblokujących operatorach przypisania). Wskazane możliwości są bardzo przydatne przy wykonaniu symulacji projektu.

Jako przykład przeanalizowany zostanie opis jednobitowego sumatora z listingu 9.15.

LISTING 9.15. Jednobitowy sumator z blokującym operatorem przypisania

```
`timescale 1ns/100ps
module add_1_blocking (
        input cin, a, b,
        output reg s, cout);

        always @ (cin, a, b) begin
                s = #5 a ^ b ^ cin;
                cout = #3 (a & b) | (b & cin) | (a | cin);
        end
endmodule
```

Do określenia wartości sumy i przeniesienia wykorzystany został operator przypisania blokującego z wewnętrznym opóźnieniem. Dlatego, gdy symulator napotka w kodzie pierwszą instrukcję przypisania, natychmiast zostanie określona wartość wyrażenia *a* ^ *b* ^ *cin*, jednak nadanie tej wartości zmiennej *s* nastąpi dopiero po upłynięciu 5 nanosekund. Jako że dana instrukcja jest operatorem przypisania blokującego, wykonanie kolejnej instrukcji rozpocznie się nie wcześniej, niż po 5 nanosekundach. Podsumowując, w ciągu 5 nanosekund będzie obliczone wyrażenie (*a* & *b*) | (*b* & *cin*) | (*a* | *cin*), natomiast przypisanie wartości tego wyrażenia zmiennej *cout* zostanie wykonane po kolejnych 3 nanosekundach, czyli po upływie 8 nanosekund od momentu wystąpienia zmiany sygnałów na wejściu.

Nieblokujące operatory przypisania pozwalają wykonywać kolejne instrukcje bez oczekiwania na zakończenie przekazywania wartości lewej stronie pierwszej instrukcji. Na listingu 9.16 pokazano opis jednobitowego sumatora z operatorem przypisania nieblokującego.

LISTING 9.16. Jednobitowy sumator z operatorem przypisania nieblokującego

```
`timescale 1ns/100ps
module add_1_non_blocking (
        input cin, a, b,
        output reg s, cout);

        always @ (cin, a, b) begin
                s <= #5 a ^ b ^ cin;
                cout <= #8 (a & b) | (b & cin) | (a | cin);
        end
endmodule
```

Gdy symulator napotka w kodzie pierwszą instrukcję przypisania, nastąpi natychmiastowe obliczenie wyrażenia prawej strony, jednak nadanie tej wartości zmiennej *s* zostanie opóźnione o 5 nanosekund. Jednakże w tym przypadku nie występuje blokada wykonywania kolejnej instrukcji. Innymi słowy, wyrażenia dla zmiennych *s* i *cout* będą obliczane w tym samym czasie, jednak wartość zmiennej *s* będzie nadana po upływie 5 nanosekund, a zmiennej *cout* – 8 nanosekund. W ten sposób działanie modułów z listingów 9.15 i 9.16 będzie zajmowało tyle samo czasu.

Zadanie. Sprawdź, jak zmieni się działanie modułu z listingu 9.16, jeżeli dla drugiej instrukcji będzie podany czas opóźnienia 3 nanosekundy?

Rozpatrzmy jeszcze kilka przykładów wykorzystania operatorów zarządzania czasem w operatorach przypisania w czasie symulacji.

```
always begin
    #5 w = ~a;   // zmiennej w zostanie nadana wartość !a po 5 jednostkach czasu
    #5 x = ~b;   // zmiennej x zostanie nadana wartość !b po 10 jednostkach czasu
```

```
    #5 y = ~ c;   // zmiennej y zostanie nadana wartość !c po 15 jednostkach czasu
    #5 z = ~ d;   // zmiennej z zostanie nadana wartość !d po 20 jednostkach czasu
end

always begin
    #5 w <= ~a; // zmiennej w zostanie nadana wartość !a po ? jednostkach czasu
    #5 x <= ~b;  // zmiennej x zostanie nadana wartość !b po ? jednostkach czasu
    #5 y <= ~c;  // zmiennej y zostanie nadana wartość !c po ? jednostkach czasu
    #5 z <= ~d;  // zmiennej z zostanie nadana wartość !d po ? jednostkach czasu
end

always fork
    #5 w = ~a;   // zmiennej w zostanie nadana wartość !a po ? jednostkach czasu
    #5 x = ~b;   // zmiennej x zostanie nadana wartość !b po ? jednostkach czasu
    #5 y = ~c;   // zmiennej y zostanie nadana wartość !c po ? jednostkach czasu
    #5 z = ~d;   // zmiennej z zostanie nadana wartość !d po ? jednostkach czasu
join

always fork
    #5 w <= ~a; // zmiennej w zostanie nadana wartość !a po ? jednostkach czasu
    #5 x <= ~b;  // zmiennej x zostanie nadana wartość !b po ? jednostkach czasu
    #5 y <= ~c;  // zmiennej y zostanie nadana wartość !c po ? jednostkach czasu
    #5 z <= ~d;  // zmiennej z zostanie nadana wartość !d po ? jednostkach czasu
join
```

Zadanie. Wykonaj symulację przytoczonych powyżej przykładów za pomocą dowolnego środowiska projektowania, a następnie zamień znaki zapytana na konkretne wartości. Czy rezultat eksperymentu odpowiadał Twoim przypuszczeniom?

Przykłady wykorzystania wewnętrznych opóźnień w operatorach przypisania.

```
always begin
    w = #5 ~a;      /* zmiennej w wartość !a z chwili 0 będzie nadana
                       po 5 jednostkach czasu */
    x = #5 ~b;      /* zmiennej x wartość !b z chwili 5 będzie nadana
                       po 10 jednostkach czasu */
    y = #5 ~c;      /* zmiennej y wartość !c z chwili 10 będzie nadana
                       po 15 jednostkach czasu */
    z = #5 ~d;      /* zmiennej z wartość !d z chwili 15 będzie nadana
                       po 20 jednostkach czasu */
end
```

```
always begin
        w <= #5 ~a;     /* zmiennej w wartość !a z chwili ? będzie nadana
                           po ? jednostkach czasu */
        x <= #5 ~b;     /* zmiennej x wartość !b z chwili ? będzie nadana
                           po ? jednostkach czasu */
        y <= #5 ~c;     /* zmiennej y wartość !c z chwili ? będzie nadana
                           po ? jednostkach czasu */
        z <= #5 ~d;     /* zmiennej z wartość !d z chwili ? będzie nadana
                           po ? jednostkach czasu */
end

always fork
        w = #5 ~a;      /* zmiennej w wartość !a z chwili ? będzie nadana
                           po ? jednostkach czasu */
        x = #5 ~b;      /* zmiennej x wartość !b z chwili ? będzie nadana
                           po ? jednostkach czasu */
        y = #5 ~c;      /* zmiennej y wartość !c z chwili ? będzie nadana
                           po ? jednostkach czasu */
        z = #5 ~d;      /* zmiennej z wartość !d z chwili ? będzie nadana
                           po ? jednostkach czasu */
join

always fork
        w <= #5 ~a;     /* zmiennej w wartość !a z chwili ? będzie nadana
                           po ? jednostkach czasu */
        x <= #5 ~b;     /* zmiennej x wartość !b z chwili ? będzie nadana
                           po ? jednostkach czasu */
        y <= #5 ~c;     /* zmiennej y wartość !c z chwili ? będzie nadana
                           po ? jednostkach czasu */
        z <= #5 ~d;     /* zmiennej z wartość !d z chwili ? będzie nadana
                           po ? jednostkach czasu */
join
```

Zadanie. Wykonaj symulację przedstawionych wyżej przykładów za pomocą dowolnego środowiska projektowania oraz zamień znaki zapytania na konkretne wartości. Czy rezultat eksperymentu odpowiadał Twoim przypuszczeniom?

9.5. Operatory proceduralne *assign* i *deassign*

Operator przypisania ciągłego posiada następującą składnię:

assign *variable* = *expression*;

Gdy symulator napotka w kodzie danego modułu ten operator, odrzuca wszystkie inne operatory przypisania zmiennej *variable* i wykonuje nadanie wartości wyrażenia *expression* zmiennej *variable*.

W odróżnieniu od zwykłego operatora przypisania ciągłego, działanie proceduralnego operatora przypisania ciągłego może zostać odwołane za pomocą operatora **deassign**. Składnia operatora **deassign**:

deassign *variable*;

Operator **deassign** odwołuje działanie występującego wcześniej w kodzie operatora przypisania ciągłego **assign** w stosunku do zmiennej *variable*.

Jako przykład wykorzystania operatorów **assign** i **deassign** przeanalizujemy układ opisany na listingu 9.17, który w zależności od kodu operacji *op* wykonuje albo dodawanie (*op*=0), albo odejmowanie (*op*=1) 8-bitowych operandów *a* i *b*. Przy czym, w trakcie dodawania zostaje uwzględnione przeniesienie bitu *cin* i wygenerowany zostaje sygnał przeniesienia *cout*, natomiast operacja odejmowania wykorzystuje operację dodawania zgodnie ze wzorem $a-b = a+(\sim b)+1$, gdzie znak ~ oznacza operację inwersji wszystkich bitów operandu *b*.

LISTING 9.17. Schemat dodawania i odejmowania z operatorami **assign** i **deassign**

```
module add_sub_1 (
    input [7:0] a, b,              // operandy
    input cin, op,                 // bit przeniesienia i kod operacji
    output reg [7:0] s,            // wynik
    output reg cout);              // wyjściowy bit przeniesienia

    always @(*) begin
        if (!op) begin
            deassign s;            // odwołanie poprzedniego przypisania
            deassign cout;         // odwołanie poprzedniego przypisania
            assign {cout,s} = a+b+cin;    // nadanie nowej wartości
        end
        else begin
```

```
                deassign s;         // odwołanie poprzedniego przypisania
                deassign cout;      // odwołanie poprzedniego przypisania
                assign {cout,s} = a+(~b)+op; // nadanie nowej wartości
            end
        end
endmodule
```

Uwaga. Nie wszystkie kompilatory wspierają operatory przypisania ciągłego **assign** i **deassign**.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator wspiera operatory przypisania ciągłego **assign** i **deassign**.

9.6. Operatory proceduralne *force* i *release*

Operator proceduralny **force** posiada następującą składnię:

force *net_or_variable* = *expression*;

Operator ten umożliwia wewnątrz bloku nadanie wartości wyrażenia *expression* zmiennej bądź sieci dowolnego typu po lewej stronie znaku równości (=). Przy czym odrzucone zostają dowolne inne przypisania do tej zmiennej.

Operator **release** odwołuje działanie operatora **force**. Operator **release** posiada następującą składnię:

release *net_or_variable*;

Jako przykład wykorzystania operatorów **force** i **release** został zmodyfikowany układ z listingu 9.17, gdzie zamiast operatorów **assign** i **deassign** wykorzystano operatory **force** i **release**.

LISTING 9.18. Schemat dodawania i odejmowania z wykorzystaniem operatorów **force** i **release**

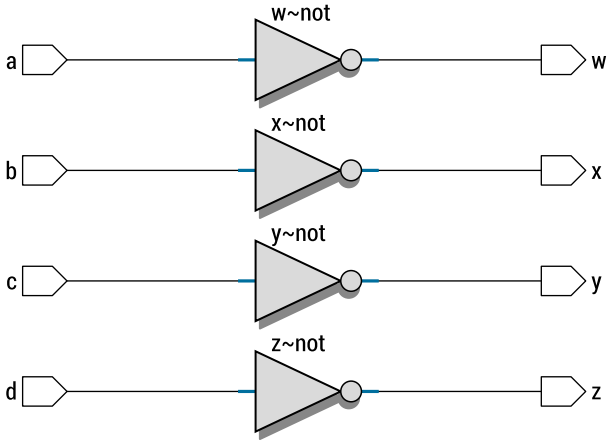
```
module add_sub_2 (
        input [7:0] a, b,       // operandy
        input cin, op,          // bit przeniesienia i kod operacji
        output reg [7:0] s,     // wynik
        output reg cout);       // wyjściowy bit przeniesienia
```

```
    always @(*) begin
        if (!op) begin
            release s;          // odwołanie poprzedniego przypisania
            release cout;       // odwołanie poprzedniego przypisania
            force {cout,s} = a+b+cin;          // nadanie nowej wartości
        end
        else begin
            release s;          // odwołanie poprzedniego przypisania
            release cout;       // odwołanie poprzedniego przypisania
            force {cout,s} = a+(~b)+op;        // nadanie nowej wartości
        end
    end
endmodule
```

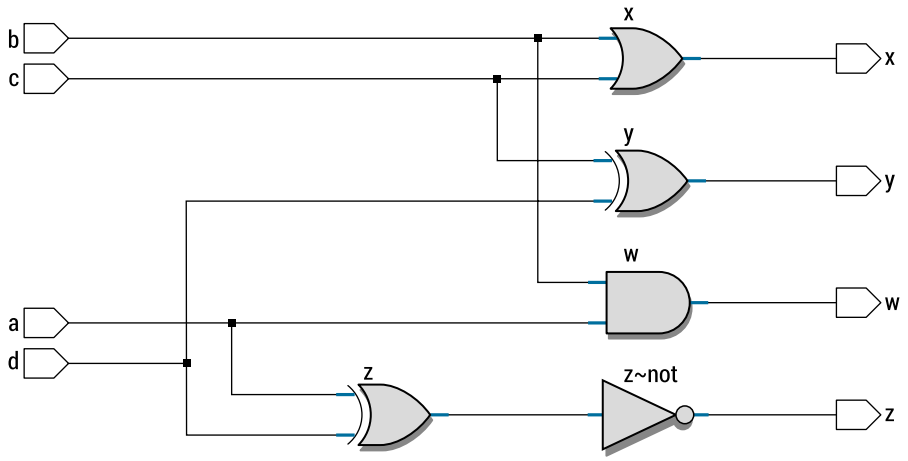
Uwaga. Nie wszystkie kompilatory wspierają proceduralne operatory **force** i **release**.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator wspiera operatory **force** i **release**.

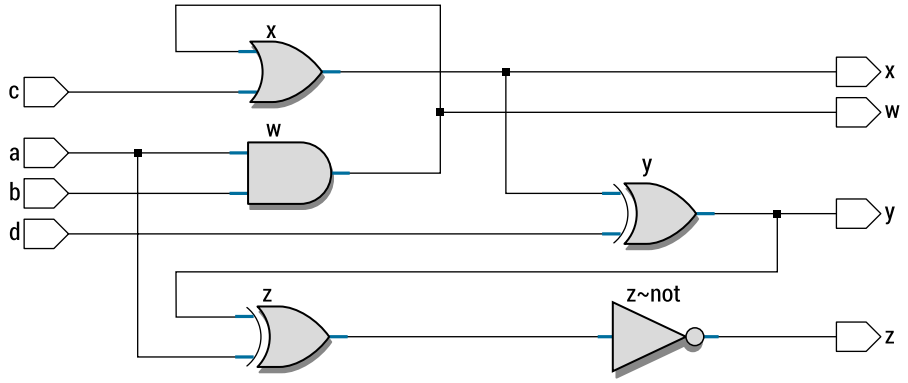
Na zakończenie niniejszego rozdziału zostały zaprezentowane wyniki syntezy przykładów z listingów 9.2-9.7.



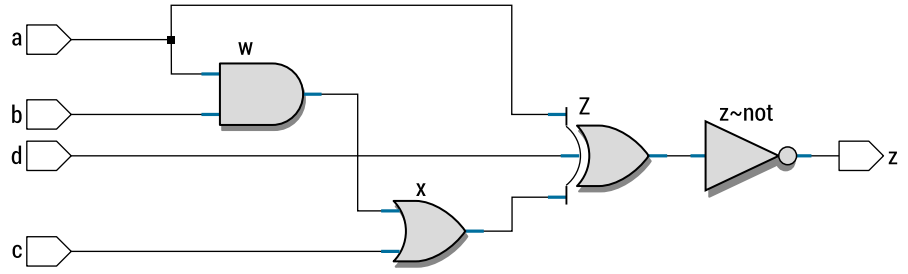
RYS. 9.8. Realizacja modułu *not_inputs* z listingu 9.2: przykład inwersji wejść



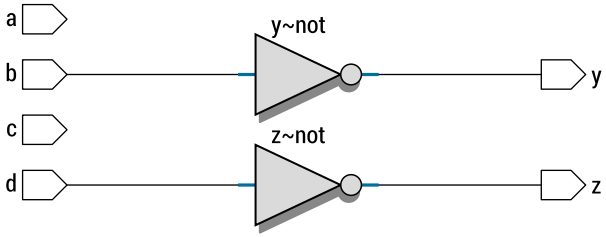
RYS. 9.9. Realizacja modułu *gating_outputs* z listingu 9.3: połączenie wejść za pomocą bramek różnego typu



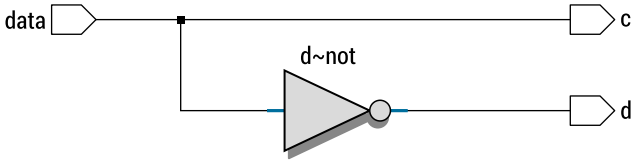
RYS. 9.10. Realizacja modułu *gating_in_out_1* z listingu 9.4: sekwencyjne połączenie wejść za pomocą bramek logicznych



RYS. 9.11. Realizacja modułu gating_in_out_2 z listingu 9.5: sekwencyjne połączenie wejść za pomocą bramek z wykorzystaniem wewnętrznej zmiennej tymczasowej



RYS. 9.12. Realizacja modułu *connect_outs* z listingu 9.6: połączenie wyjść inwerterów



RYS. 9.13. Realizacja modułu *comb_circuit* z listingu 9.7: przykład opisu prostego układu kombinacyjnego

10. Operatory programowania strukturalnego

10.1. Cechy wspólne

Operatory programowania strukturalnego języka Verilog służą przede wszystkim do zmiany porządku wykonywania instrukcji. Za pomocą tych operatorów realizowane są rozgałęzienia przy sekwencyjnym wykonywaniu instrukcji, tworzą się pętle, powtórzenia, umożliwia się wykonywanie wybranych fragmentów kodu w przypadku wystąpienia określonych warunków. Większość operatorów programowania strukturalnego języka Verilog została zaczerpnięta z języków programowania wysokiego poziomu, jednakże występują też pewne wyjątki.

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

Uwaga. Nie wszystkie operatory programowania strukturalnego są syntezowalne.

Zadanie. Dowiedz się, które operatory programowania strukturalnego w wykorzystywanym przez Ciebie kompilatorze są syntezowalne, a które nie.

Uwaga. W zaprezentowanych poniżej składniach operatorów programowania strukturalnego, zamiast instrukcji *statement* może występować blok instrukcji umieszczony wewnątrz nawiasów operatorowych **begin-end** albo **fork-join**.

10.2. Operator *if-else*

Operator **if-else** umożliwia wykonywanie określonych fragmentów kodu, gdy zostanie spełniony podany warunek. Składnia operatora **if-else**:

if (*expression*) *statement*

else *statement*

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

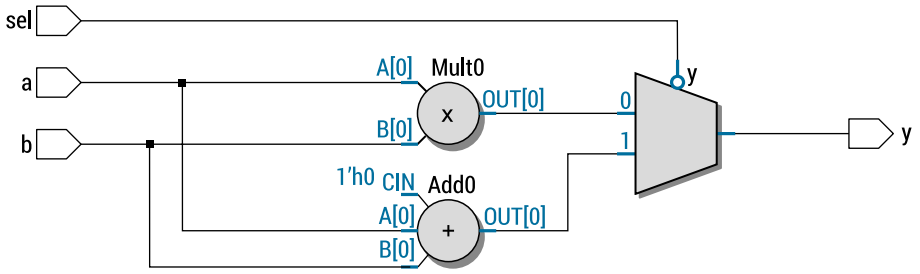
Działanie operatora **if-else**: na początku zostaje obliczone wyrażenie *expression*. Jeżeli wartość wyrażenia *expression* jest prawdą (wynosi 1'b1), wówczas zostaje wykonana instrukcja *statement* występująca po wyrażeniu *expression*. Jeżeli natomiast wartość wyrażenia *expression* jest fałszem (wynosi 1'b0) lub jest nieznane (wynosi **X**), wówczas wykonywana jest instrukcja *statement* występująca po słowie kluczowym **else**; w przypadku braku konstrukcji **else** wykonywana jest kolejna instrukcja kodu.

Na listingach 10.1-10.8 przedstawione zostały przykłady wykorzystania operatora **if-else**.

LISTING 10.1. Urządzenie arytmetyczne wykonujące albo dodawanie, albo mnożenie.

```
module adder_mult (input a, b, sel,
                   output reg y);
        always @(*) begin
                   if (sel==0)        y = a + b;
                   else               y = a * b;
        end
endmodule
```

Realizacja przykładu z listingu 10.1 została przedstawiona na rysunku 10.1.



RYS. 10.1. Realizacja modułu *adder_mult* z listingu 10.1: przykład wykorzystania operatora **if** do opisu ALU realizującego arytmetyczne operacje dodawania lub mnożenia

LISTING 10.2. Parametryzowany moduł dwukierunkowego multipleksera

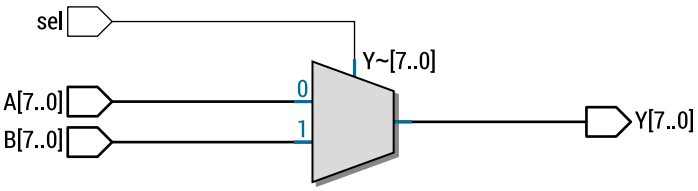
```
module mux_N_if #(parameter N=8)    // N – szerokość słów
        (input [N–1:0] A, B,        // słowa wejściowe
        output reg [N–1:0] Y,       // wyjście
        input sel);                 // wejście sterujące

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

LISTING 10.3. Wykorzystanie zamiast operatora **if** operacji warunkowej

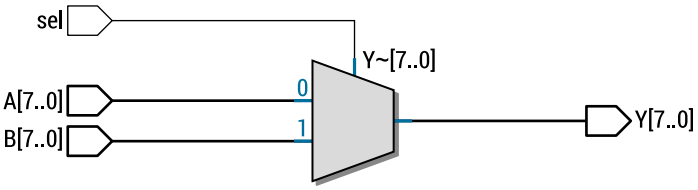
```
module mux_N_conditional #(parameter N=8)   // N – szerokość słów
        (input [N–1:0] A, B,                // słowa wejściowe
        output [N–1:0] Y,                   // wyjście
```

```
    input sel);                          // wejście sterujące

    assign Y = (sel) ? B : A;

endmodule
```

Realizacja przykładów z listingów 10.2 i 10.3 została przedstawiona odpowiednio na rysunkach 10.2 i 10.3.



RYS. 10.2. Realizacja modułu *mux_N_if* z listingu 10.2: przykład realizacji szynowego multipleksera za pomocą operatora **if**



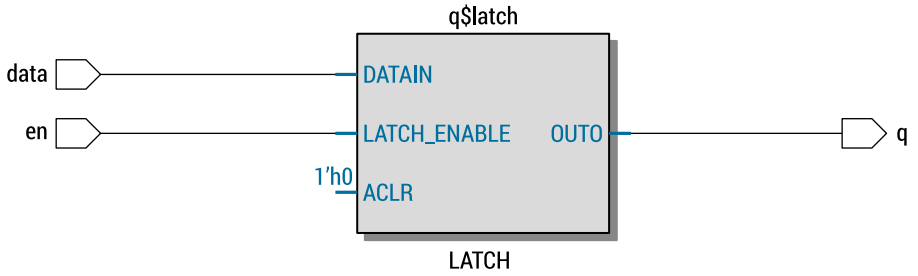
RYS. 10.3. Realizacja modułu *mux_N_conditional* z listingu 10.3: przykład realizacji szynowego multipleksera za pomocą wyrażenia warunkowego

Analiza schematów z rysunków 10.2 i 10.3 pokazuje, że operator **if** jest odpowiednikiem wyrażenia warunkowego.

LISTING 10.4. Opis zatrzasku

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

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

Realizacja przykładu z listingu 10.4 została przedstawiona na rysunku 10.4.



RYS. 10.4. Realizacja modułu *latch_1* z listingu 10.4: realizacja zatrzasku za pomocą operatora **if**

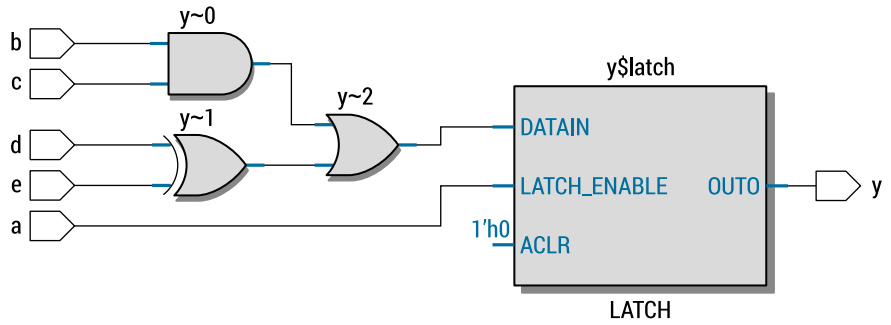
Uwaga. Brak konstrukcji **else** przy operatorze **if** w opisie układu kombinacyjnego powoduje, że syntezator na wyjściu schematu ustawia zatrzask.

Należy uważać przy wykorzystaniu operatora **if** do opisu układów kombinacyjnych. Jeżeli dla niektórych danych wejściowych nie są jawnie określone wartości wyjść układu kombinacyjnego, przyjmuje się, że dla danych wejść wartości wyjściowe nie zostają zmienione, czyli wyjścia zachowują poprzednie wartości. Jednak zachowanie wartości wyjść wymaga wykorzystania elementów pamięci. W rezultacie syntezator tworzy na wyjściu schematu zatrzask, a układ przestaje być kombinacyjny i przechodzi do klasy układów z pamięcią (czyli układów sekwencyjnych). Podobna sytuacja będzie występować, gdy w operatorze **if** nie uwzględni się konstrukcji **else**.

LISTING 10.5. Błędny opis schematu kombinacyjnego

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

Wynik syntezy układu z listingu 10.5 został przedstawiony na rysunku 10.5.



RYS. 10.5. Realizacja modułu *latch_2* z listingu 10.5: wynik błędnego opisu układu kombinacyjnego – na wyjściu schematu zostaje ustawiony zatrzask

Poniżej zamieszczono kilka możliwych sposobów wykorzystania operatora **if-else** do realizacji układów kombinacyjnych.

LISTING 10.6. Wykorzystanie konstrukcji **else** przy realizacji układu kombinacyjnego

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

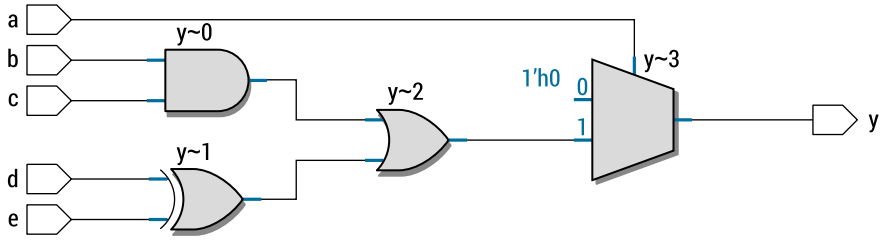
LISTING 10.7. Opis układu kombinacyjnego z nadaniem początkowych wartości

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

LISTING 10.8. Opis układu kombinacyjnego z nadaniem wartości sygnału sterującego

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

Wynik syntezy przykładów z listingów 10.6-10.8 jest identyczny i został przedstawiony na rysunku 10.6.



RYS. 10.6. Realizacja modułów *comb_circuit_1-3* z listingów 10.6-10.8

Zadanie. Przeanalizuj przykłady z listingów 10.6-10.8. Spróbuj zauważyć mocne i słabe strony każdego ze sposobów opisu układu kombinacyjnego. Jakie jeszcze sposoby opisu danego układu kombinacyjnego możesz wymyślić? Który ze sposobów jest najlepszy?

10.3. Operator *case*

Operator **case** umożliwia wykonywanie różnych fragmentów kodu w zależności od wartości wyliczonego wyrażenia. Składnia operatora **case**:

case (*expression*)
    *case_item***:** *statement;*
    …
    *case_item, …, case_item***:** *statement;*
    default: *statement*;
endcase

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

Działanie operatora **case** jest następujące: na początku obliczane zostaje wyrażenie *expression*. Następnie otrzymana wartość kolejno jest porównywana z występującymi niżej elementami stałymi *case_item*. W przypadku natrafienia na element stały, odpowiadający wartości wyrażenia *expression,* wykonywana jest instrukcja *statement*, występująca od razu za danym elementem stałym. Jeżeli żaden ze stałych elementów nie odpowiada wartości wyliczonego wyrażenia, wówczas wykonywana jest instrukcja występująca po słowie kluczowym **default**.

Uwagi.

1) Jednej instrukcji *statement* może być przypisanych kilka wartości elementów stałych. Instrukcja będzie wykonywana, gdy wartość wyrażenia będzie odpowiadać dowolnemu z elementów stałych.
2) Po wykonaniu instrukcji występującej za odpowiednim elementem stałym, pozostałe elementy nie są już sprawdzane i wykonywanie instrukcji **case** zostaje zakończone.

Na poniższych listingach zaprezentujemy przykłady wykorzystania instrukcji **case** do zaprojektowania układu kombinacyjnego.

Aby porównać możliwości operatorów **case** oraz **if-else** przy projektowaniu schematów kombinacyjnych, na początku rozpatrzymy realizację poniższej funkcji boolowskiej:

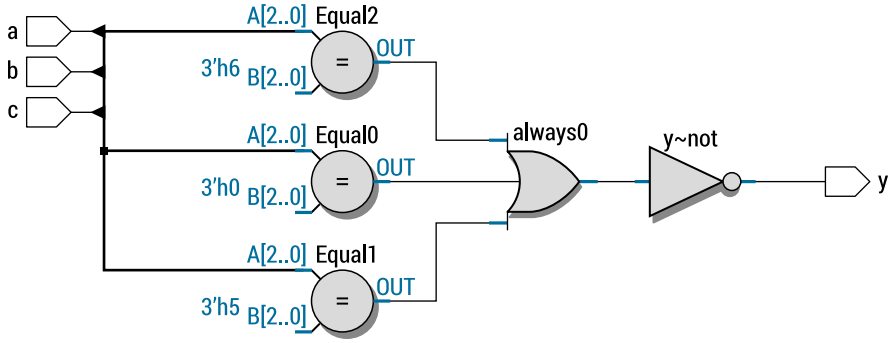
$$y = f(a,b,c) = \Sigma\, m(a,b,c) = \Sigma(1,2,3,4,7) = \Pi(0,5,6)$$

za pomocą operatora **if-else**.

LISTING 10.9. Opis funkcji boolowskiej za pomocą operatora **if-else**

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

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

Wynik syntezy schematu z listingu 10.9 został przedstawiony na rysunku 10.7.



RYS. 10.7. Realizacja modułu *sm_IF* z listingu 10.9: przykład opisu funkcji boolowskiej za pomocą operatora **if**

Na listingach 10.10 – 10.12 przedstawiony został ten sam przykład, ale z wykorzystaniem operatora **case**.

LISTING 10.10. Opis funkcji *y* za pomocą operatora **case** odpowiadającego tablicy prawdy

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

    always @(*)
```

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

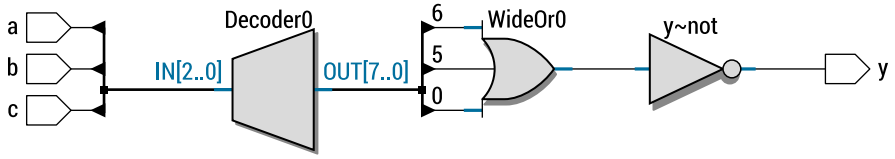
LISTING 10.11. Opis funkcji *y* z wykorzystaniem konstrukcji **default**

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

    always @(*)
      case ({a, b, c})
        3'b000: y = 1'b0;  // sprawdzane są tylko zerowe wartości wyjścia
        3'b101: y = 1'b0;
        3'b110: y = 1'b0;
        default: y = 1'b1; // wartość wyjścia równa 1 zostaje przyjęta domyślnie
      endcase
endmodule
```

LISTING 10.12. Wariant realizacji poprzedniego przykładu, gdy dla jednej instrukcji przypisanych jest kilka elementów stałych operatora **case**

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

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

W wyniku syntezy przykładów z listingów 10.10-10.12 otrzymany został taki sam schemat, zaprezentowany na rysunku 10.8.



RYS. 10.8. Realizacja modułów *sm_Case1-3* z listingów 10.10-10.12: przykład opisu funkcji boolowskiej za pomocą operatorów **case**

Zadanie. Przeanalizuj przykłady z listingów 10.10-10.12. Spróbuj zauważyć mocne i słabe strony każdego ze sposobów opisu schematu kombinacyjnego. Jakie jeszcze sposoby opisu tego układu z wykorzystaniem operatora **case** jesteś w stanie wymyśleć? Który ze sposobów jest najlepszy?

Dużą wadą operatora **case**, ograniczającą jego wykorzystanie, jest to, że poszczególne bity w wartości wyrażenia, a także poszczególne bity elementów stałych mogą przyjmować jedynie wartości 0 albo 1. Aby rozszerzyć możliwości operatora **case**, do języka Verilog zostały dodane operatory **casez** i **casex**.

10.4. Operatory *casez* i *casex*

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

Uwaga. Operatory **casez** i **casex** kończą się jednym i tym samym słowem kluczowym **endcase**.

Operator **casez** pozwala w rozpatrywanym wyrażeniu oraz w elementach stałych wykorzystywać logiczne wartości **Z** do oznaczania nieokreślonych (*don't care*) wartości bitów. Operator **casex**, podobnie jak operator **casez**, umożliwia wykorzystanie dwóch wartości logicznych **Z** oraz **X** do oznaczenia nieokreślonych (*don't care*) wartości bitów. Oprócz tego, w elementach stałych operatorów **casez** i **casex** do oznaczenia nieokreślonych i nieznanych wartości bitów można wykorzystywać znak zapytania (?).

Poniżej przytoczono kilka przykładów wykorzystania operatorów **casez** i **casex**.

LISTING 10.13. Przykład wykorzystania nieokreślonej wartości wejść w zwyczajnym operatorze **case**

```
module sm_Case4          (input a, b, c,
                          output reg f);
    always @(*)
      case ({a, b, c})
          3'b001: f = 1'b1;
          3'b010: f = 1'b1;
```

```
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b110: f = 1'b0;
            3'b111: f = 1'b1;
            default: f = 1'bx;   // wyjście może przyjmować nieokreśloną wartość
        endcase
endmodule
```

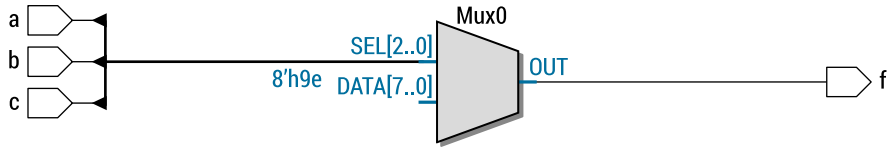
LISTING 10.14. Przykład wykorzystania nieokreślonych wartości wejść, wykorzystanie operatora **casex**

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

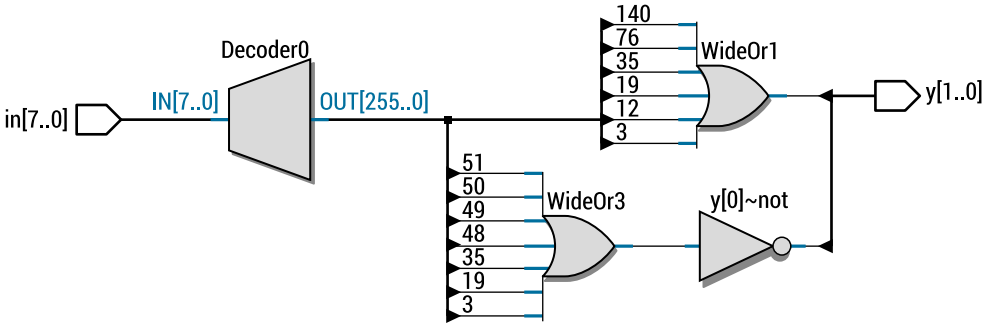
LISTING 10.15. Wykorzystanie znaku zapytania (?) w operatorze **casex** przy opisie schematów kombinacyjnych

```
module sm_Case5         (output reg 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
```

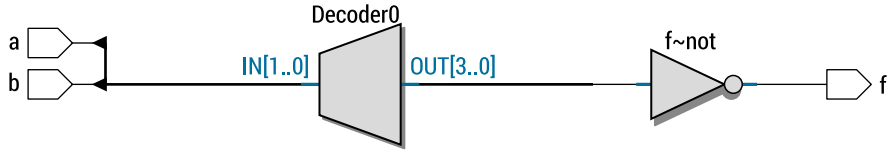
Wyniki syntezy przykładów z listingów 10.13-10.15 zostały przedstawione odpowiednio na rysunkach 10.9-10.11.



RYS. 10.9. Realizacja modułu *sm_Case4* z listingu 10.13: przykład wykorzystania nieokreśloności do nadania wartości wyjścia w operatorze **case**



RYS. 10.10. Realizacja modułu *decoder* z listingu 10.14: przykład wykorzystania nieokreślonej wartości wejść w operatorze **casex**



RYS. 10.11. Realizacja modułu *sm_Case5* z listingu 10.15: przykład wykorzystania znaku „?" przy opisie wartości wejść w operatorze **casex**

Zadania.

1) Spróbuj wyjaśnić, dlaczego podczas realizacji przykładu z listingu 10.13 był wykorzystany multiplekser, natomiast przy realizacji przykładów z listingów 10.14 i 10.15 wykorzystano dekoder?
2) Opisz za pomocą operatora **casex** priorytetowy koder na 8 wejść.

10.5. Operator *for*

Operator **for** przeznaczony jest do tworzenia pętli. Jego składnia w praktyce odpowiada składni analogicznego operatora języka programowania C. Składnia operatora **for** przedstawia się następująco:

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

gdzie *initial_assignment* – instrukcja przypisania, która wykonywana jest tylko jeden raz na początku operatora **for**; *expression* – wyrażenie, wartość którego sprawdzana jest przed wykonywaniem każdej iteracji pętli; *step_assignment* – instrukcja przypisania, która wykonywana jest na końcu każdej iteracji.

Działanie operatora **for** jest następujące: na początku wykonywania operatora **for** jako pierwsza zostaje wykonana instrukcja *initial_assignment*, która przypisuje początkowe wartości zmiennej iteracyjnej (zazwyczaj jako zmienną iteracyjną wykorzystuje się zmienną typu **integer**). Następnie zostaje obliczana wartość wyrażenia *expression*. Jeżeli wyliczona wartość jest prawdą, wówczas wykonywana jest instrukcja *statement*; w przeciwnym wypadku wykonywana jest kolejna instrukcja występująca po operatorze **for**. Po zakończeniu każdej iteracji wykonywana jest instrukcja *step_assignment*, która zazwyczaj przypisuje nową wartość zmiennej iteracyjnej. Następnie ponownie sprawdza się wyrażenie *expression* i proces wykonywania operatora **for** jest powtarzany.

Uwaga. Do inkrementacji (dekrementacji) zmiennej w operatorze **for** nie wolno wykorzystywać instrukcji *++i* lub *--i* znanych z języka C, gdyż w Verilogu instrukcje te nie występują.

Na zmienną iteracyjną oraz wykorzystywane w operatorze **for** instrukcje przypisania konkretne kompilatory nakładają także inne ograniczenia.

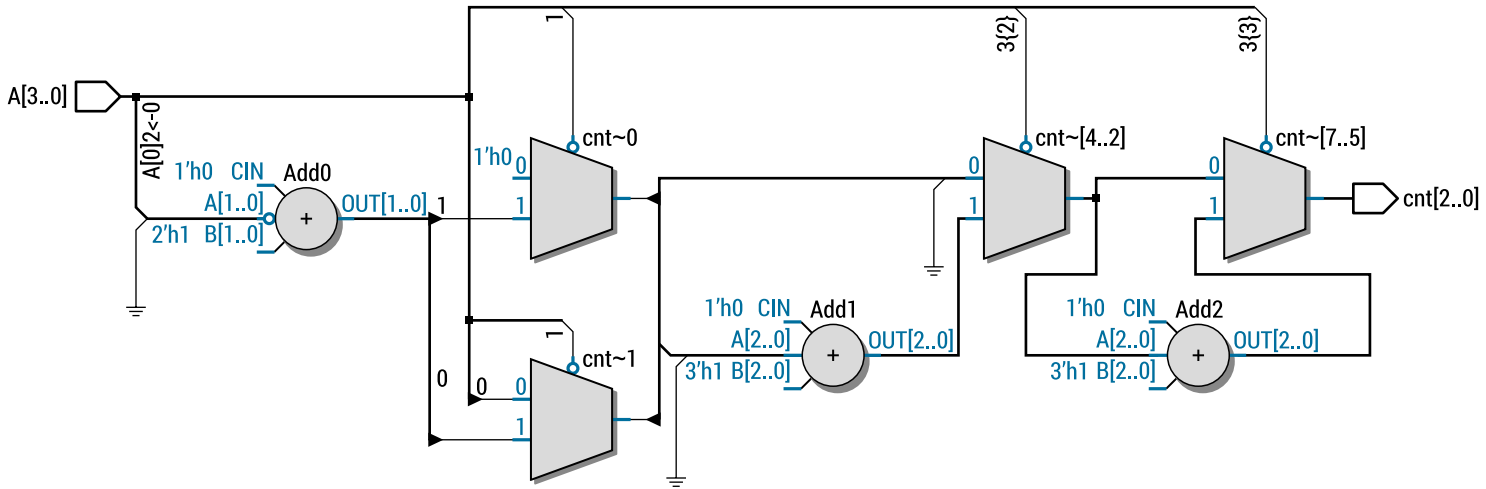
Zadanie. Dowiedz się, jakie ograniczenia na operatora **for** nakłada wykorzystywany przez Ciebie kompilator.

Listing 10.16 prezentuje przykład zastosowania operatora **for**.

LISTING 10.16. Określenie liczby zer w 4-bitowym słowie za pomocą instrukcji **for**

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

integer i;                                  // zmienna iteracyjna

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

Wynik syntezy przykładu z listingu 10.16 został pokazany na rysunku 10.12.



RYS. 10.12. Realizacja modułu *count_zero1* z listingu 10.16: przykład wykorzystania operatora **for** do określenia liczby zer w 4-bitowym słowie

Uwaga. Analiza rysunku 10.12 pokazuje, że wykorzystanie operatora **for** umożliwia skrócenie opisu, jednak powoduje dość złożoną realizację z punktu widzenia wykorzystywanych zasobów sprzętowych oraz prowadzi do stworzenia wielopoziomowych układów spowalniających działanie projektu.

10.6. Operator *while*

Operator **while** jest przeznaczony do tworzenia pętli, które są przerywane w przypadku niespełnienia podanych warunków. Składnia operatora **while**:

while (*expression*) *statement*

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

Działanie operatora **while** jest następujące: przed rozpoczęciem każdej iteracji pętli zostaje obliczona wartość wyrażenia *expression*. Jeżeli jest ono prawdą, wówczas wykonywana zostaje instrukcja *statement*; w przeciwnym wypadku, wykonywana jest instrukcja znajdująca się za operatorem **while**. Innymi słowy, instrukcja *statement* wykonywana jest tak długo, póki wartość wyrażenia *expression* jest prawdą.

Uwaga. Aby zapewnić wyjście z pętli **while**, w bloku instrukcji *statement* należy przewidzieć działania, które zmieniają wartość wyrażenia *expression*, w przeciwnym wypadku pętla operatora **while** będzie wykonywana bez końca.

Przykład wykorzystania operatora **while** podany jest na listingu 10.17.

LISTING 10.17. Określenie liczby zer w 4-bitowym słowie z wykorzystaniem operatora **while**

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

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

W wyniku będzie syntezowany dokładnie taki sam układ jak na rysunku 10.12.

10.7. Operator *repeat*

Operator **repeat** służy do wykonania określonej liczby powtórzeń pętli. W odróżnieniu od operatora **while**, wykonanie operatora **repeat** zawsze się kończy. Oprócz tego, w pętli nie ma obowiązku zamieszczania instrukcji do zmiany wyrażenia *expression*. Jedynym minusem operatora **repeat** jest to, że przed jego wykorzystaniem należy wiedzieć ile razy pętla ma być powtórzona. Składnia operatora **repeat**:

repeat (*number*) *statement*

gdzie *number* – liczba powtórzeń pętli. Konstrukcja *number* w operatorze **repeat** jest wyrażeniem, którego wartość obliczana jest tylko jeden raz, podczas startu wykonywania operatora **repeat** i nie zmienia się w procesie wykonywania instrukcji.

Działanie operatora **repeat** jest następujące – instrukcja *statement* wykonywana jest tyle razy, ile wynosi liczba powtórzeń pętli wskazana za pomocą wyrażenia *number*.

Jako przykład wykorzystania operatora **repeat** zaprezentowano opis urządzenia mnożącego 8-bitowe słowa.

LISTING 10.18. Opis urządzenia mnożącego 8-bitowe słowa

```
module mult (output [15:0] y,          // wyjścia
             input [7:0] a, b);        // wejścia

    // wykorzystanie funkcji do realizacji algorytmu mnożenia
    function [15:0] multiply (input [7:0] a, b);
    begin: serialMult
        reg [7:0] mcnd, mpy;
        mpy = b;                       // mnożnik
        mcnd = a;                      // mnożna
        multiply = 0;                  // wynik mnożenia

        repeat (8) begin
            if (mpy[0])                // jeżeli młodszy bit wynosi 1, wówczas
            multiply = multiply + {mcnd, 8'b00000000};
            multiply = multiply >> 1;
            mpy = mpy >> 1;
        end
    end
    endfunction

    assign y = multiply(a,b);
endmodule
```

Uwaga. Wykorzystanie operatorów iteracyjnych (**for**, **while** i **repeat**) powoduje stworzenie wielopoziomowego, kaskadowego układu z sumatorem na każdym poziomie, dlatego niektóre kompilatory ograniczają maksymalną liczbę iteracji na etapie syntezy projektów.

Zadanie. Znajdź maksymalną liczbę iteracji podczas syntezy dla każdego z operatorów **for**, **while** i **repeat**, które wspiera wykorzystywany przez Ciebie kompilator.

10.8. Operator *forever*

Operator **forever** przeznaczony jest do tworzenia nieskończonych pętli. Składnia operatora **forever**:

forever *statement*

gdzie *statement* – instrukcja wykonywana w niekończącej się pętli.

Działanie operatora **forever** jest następujące: operator **forever** rozpoczyna cyklicznie powtarzające się wykonanie instrukcji *statement*, które trwa aż do zakończenia czasu symulacji.

Uwaga. Operator **forever** często jest wykorzystywany do tworzenia nieskończonych sekwencji sygnałów, na przykład sygnału synchronizacji.

Przykład wykorzystania operatora **forever** prezentuje listing 10.19.

LISTING 10.19. Generator sygnału synchronizacji z okresem 50 jednostek czasu, który rozpoczyna swoją pracę po upłynięciu 1000 jednostek czasu od początku symulacji.

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

Przytoczony przykład nie może zostać zrealizowany przez syntezator, ponieważ syntezator zabrania realizacji niekończących się cykli. Jednakże taki opis może być wykorzystany przy modelowaniu projektu.

10.9. Operator *disable*

Operator **disable** służy do przerwania wykonywania sekwencji instrukcji w nazwanym bloku (*named block*). Składnia operatora **disable**:

disable *block_name*

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

Działanie operatora **disable**: gdy symulator napotka w kodzie operator **disable**, wówczas realizowane jest wyjście z nazwanego bloku instrukcji *block_name*, czyli sterowanie przekazywane jest instrukcji następującej po nazwanym bloku *block_name*.

Uwaga. Operator **disable** często jest wykorzystywany do wymuszania wyjścia z pętli, czyli zakończenia pracy operatorów **for**, **while** i innych.

Poniżej zaprezentowano przykłady wykorzystania operatora **disable**.

Następujący fragment kodu realizuje funkcje operatorów *continue* i *break*, znanych z języka C, za pomocą operatora **disable**:

```
begin: break

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

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

            // inne instrukcje

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

LISTING 10.20. Wyszukanie w wektorze bitowym indeksu pierwszej jedynki z lewej strony

```
module index_of_left_one # (parameter N=8)
        (input [N–1:0] a,                   // wektor wejściowy
        output integer y);                  // zwracana wartość
        always begin: block_1               // nazwany blok block_1
```

```
        for (y=0; y<N; y=y+1)
            if (a[y]==1'b1)
                disable block_1;      // wyjście z bloku block_1
    end                               // koniec bloku block_1
endmodule
```

10.10. Różnice pomiędzy operatorami *wait* i *while*

Różnice pomiędzy operatorami **wait** i **while** znajdują odzwierciedlenie w istocie tych operatorów. Operator **wait** steruje wykonywaniem procesu, może więc zatrzymywać proces do momentu spełnienia wskazanego warunku. Jako warunki mogą występować sygnały będące zewnętrznymi w stosunku do danego procesu.

Uwaga. Procesem w języku Verilog są bloki proceduralne **initial** oraz **always**.

Procesy w języku Verilog są wykonywane równolegle i zatrzymanie jednego z nich nie zatrzymuje symulacji całego projektu.

Operator **while** steruje procesem wykonywania bloku instrukcji. Dodatkowo operator **while** nie zatrzymuje swojego procesu. Żeby cykl **while** się zakończył, w ciele bloku instrukcji powinna pojawić się zmiana pewnej wartości, która sprawdzana jest w wyrażeniu operatora **while** przed wykonaniem każdej iteracji pętli. Dodatkowo, sterowanie operatorem **while** za pomocą sygnału pochodzącego spoza danego operatora, a tym bardziej spoza procesu, jest niemożliwe, ponieważ przeczy to koncepcji operatora **while**.

Zadanie. Napisz fragment kodu z operatorem **while**, w którym sprawdzane są wartości sygnałów: *a* – zewnętrznego w stosunku do tego operatora; *b* – zewnętrznego w stosunku do danego procesu. Jakie komunikaty przy tej czynności zwraca kompilator?

LISTING 10.21. Przykład wykorzystania operatora **while** do sterowania procesem

```
module endlessloop (output reg [15:0] cnt);

    reg a=0;
    always #50 a=~a;          // pierwszy proces, cyklicznie zmienia wartość
                              // sygnału a

    always begin              // drugi proces
        cnt=0;
        while (a) begin       // niekończąca się pętla
            cnt=cnt+1;        // zmiana wartości zmiennej cnt
                              // nie zmienia sprawdzanego wyrażenia
```

```
            $display(„cnt=%0d”, cnt);
        end
        $display(„Ten komunikat nigdy nie zostanie wyświetlony.
Dlaczego?”);
    end
endmodule
```

Zadanie. Sprawdź, co się stanie z poprzednim przykładem, jeżeli zamiast operatora **while** zastosujesz operator **wait**?

11. Atrybuty

11.1. Atrybuty w języku Verilog

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

(* *attribute, attribute,…* ****)***

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

Uwaga. Nawiasy stosowane przy atrybutach (pary znaków (* i *)) są wymagane nawet, gdy określany jest pojedynczy atrybut.

Atrybuty w języku Verilog mogą być przypisywane w postaci prefiksów podczas deklaracji zmiennych, instancji modułów, instrukcji, portów itd. Atrybuty mogą także być przypisywane w postaci sufiksów do instrukcji bądź wywołań funkcji. Atrybuty mogą posiadać wartości. Jeżeli wartość atrybutu nie jest określona, wówczas domyślnie przyjmuje się wartość 1. Umożliwiono także jednoczesne określenie kilku atrybutów. W tym przypadku listę atrybutów oddziela się przecinkami.

Przykłady wykorzystania atrybutów:

```
(* INIT="1" *) reg Q;                  // wykorzystanie atrybutu przy inicjacji zmiennej Q
sum = a + (* ripple_adder *) b;        // wykorzystanie atrybutu w wyrażeniu w postaci
                                       // sufiksu do instrukcji dodawania
```

Atrybuty, jako możliwa jednostka konstrukcyjna, została po raz pierwszy wprowadzona w języku Verilog-2001. Jednak standard języka Verilog-2001 nie określał konkretnych atrybutów. Konkretne atrybuty w języku Verilog wnoszą twórcy oprogramowania (kompilatorów, syntezatorów, symulatorów), które wspiera język Verilog.

Zadanie. Dowiedz się jakie atrybuty są dostępne w wykorzystywanym przez Ciebie środowisku do projektowania.

Pomimo tego, że konkretne atrybuty w języku Verilog są określane przez projektantów konkretnych pakietów oprogramowania, wszystkie współczesne kompilatory języka Verilog wspierają dwa atrybuty: *full_case* oraz *parallel_case*.

11.2. Atrybut *full_case*

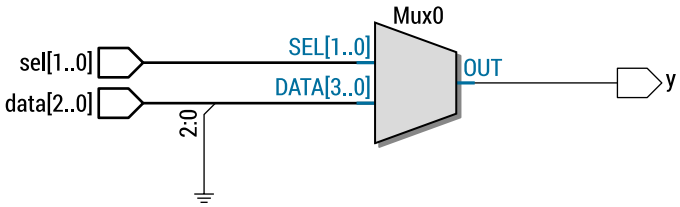
Wykorzystanie operatora **case** podczas opisu złożonego schematu kombinacyjnego może prowadzić do sytuacji, gdy wartość wyjść schematu nie jest określona dla wszystkich możliwych wartości wejść. Taka sytuacja często jest spotykana w przypadku, gdy w operatorze **case** nie występuje konstrukcja **default**. W przypadku jeśli nie zostanie określona wartość wyjścia układu dla wszystkich możliwych wejść, syntezator na wyjściu schematu automatycznie wstawi zatrzask (podobnie jak w sytuacji z operatorem **if** opisanym w podrozdziale 10.2). Aby zapobiec automatycznemu ustawieniu zatrzasku na wyjściu schematu kombinacyjnego stosuje się atrybut *full_case*.

Atrybut *full_case* wskazuje syntezatorowi, by ustawiał wyjścia schematu jako nieokreślone (*don't care*) dla danych wejściowych, które nie zostały podane w operatorze **case**. Poniżej zaprezentowano wykorzystanie atrybutu *full_case* do opisu schematu kombinacyjnego.

LISTING 11.1. Przykład wykorzystania atrybutu *full_case*

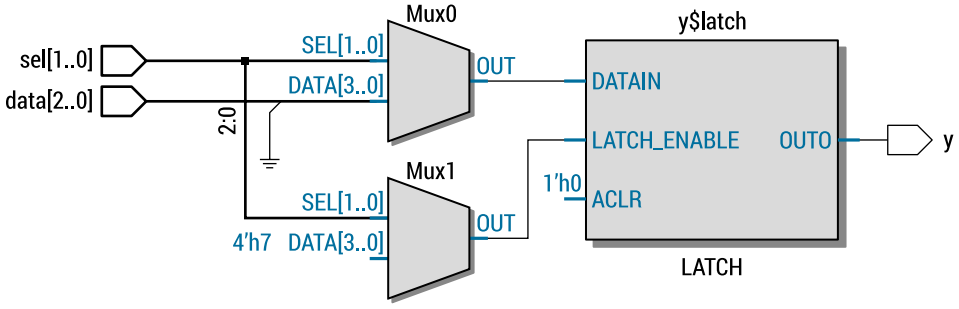
```
module use_full_case (input [2:0] data,
                      input [1:0] sel,
                      output reg y);
    always @ (*)
        (* full_case *)              // wykorzystanie atrybutu full_case
        case (sel)                   // przy operatorze case
            2'b00 : y = data[0];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

Wynik syntezy przykładu z listingu 11.1 został przytoczony na rysunku 11.1.



RYS. 11.1. Realizacja modułu *use_full_case* z listingu 11.1: przykład wykorzystania atrybutu *full_case* przy operatorze **case**

Realizacja tego przykładu bez atrybutu *full_case* wymaga większej liczby elementów logicznych. Została ona przedstawiona na rysunku 11.2.



RYS. 11.2. Realizacja przykładu z listingu 11.1 bez atrybutu *full_case*

Zadanie. Na listingu 11.1 do operatora **case** dodaj konstrukcję **default** do ustalenia nieokreślonej wartości wyjścia w przypadku kombinacji wejściowej 2'b11. Wynik syntezy porównaj z rysunkami 11.1 oraz 11.2.

11.3. Atrybut *parallel_case*

Dla przypomnienia opiszemy jeszcze raz zasadę działania operatora **case**: zostaje obliczone wyrażenie; otrzymana wartość sekwencyjnie jest porównywana z elementami stałymi w porządku, w jakim zostały one zapisane w kodzie; w przypadku, gdy wartość elementu stałego odpowiada wartości wyrażenia, wykonywana jest odpowiednia instrukcja; po zakończeniu wykonywania instrukcji operator **case** kończy swoje działanie, przy czym pozostałe elementy stałe nie są sprawdzane. Zaprezentowany algorytm wykonywania operatora **case** wymaga realizacji w schemacie kodera priorytetowego, który będzie kończył wykonywanie operatora **case** w przypadku znalezienia odpowiedniej wartości elementu stałego.

Istnieją dwa powody, aby zmienić strategię realizacji operatora **case**.

Po pierwsze, jeżeli wiadomo, że w przypadku znalezienia wartości elementu stałego odpowiadającej wartości wyrażenia, inne elementy stałe nie mogą być mu równe. Jako przykład może być wykorzystany opis za pomocą operatora **case** pełnej tablicy prawdy funkcji boole'owskiej. W danym przypadku do zwiększenia szybkości działania układu, a czasami do zmniejszenia kosztów realizacji, rozsądne jest usunięcie kodera priorytetowego przy schemacie realizacji operatora **case**.

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

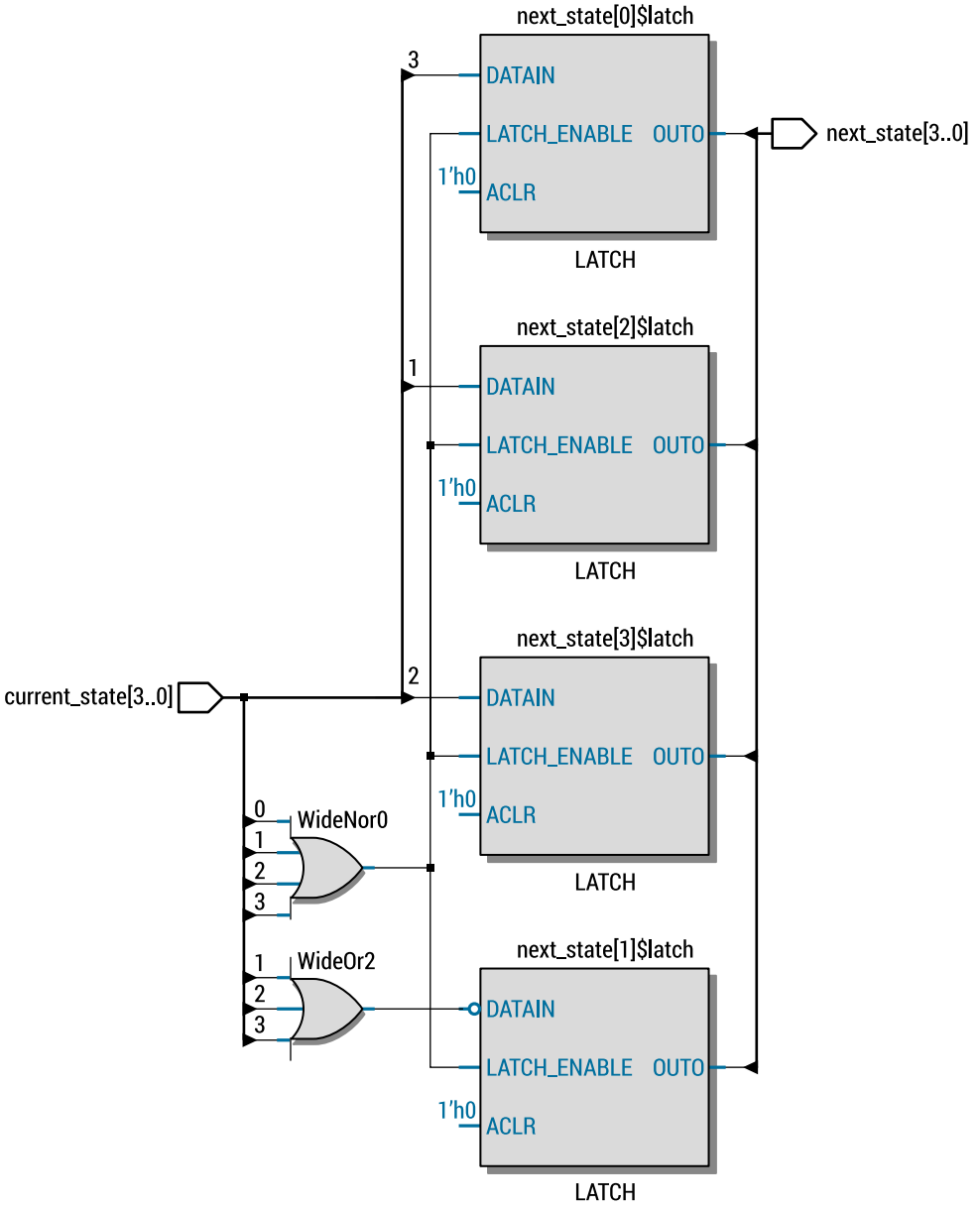
Do rozwiązania przytoczonych problemów służy atrybut *parallel_case*. Atrybut *parallel_case* wskazuje syntezatorowi, aby zamiast realizacji z wykorzystaniem kodera priorytetowego wykonać równoległą realizację instrukcji dla wszystkich elementów stałych operatora **case**.

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

LISTING 11.2. Przykład schematu realizującego funkcję przejść automatu skończonego

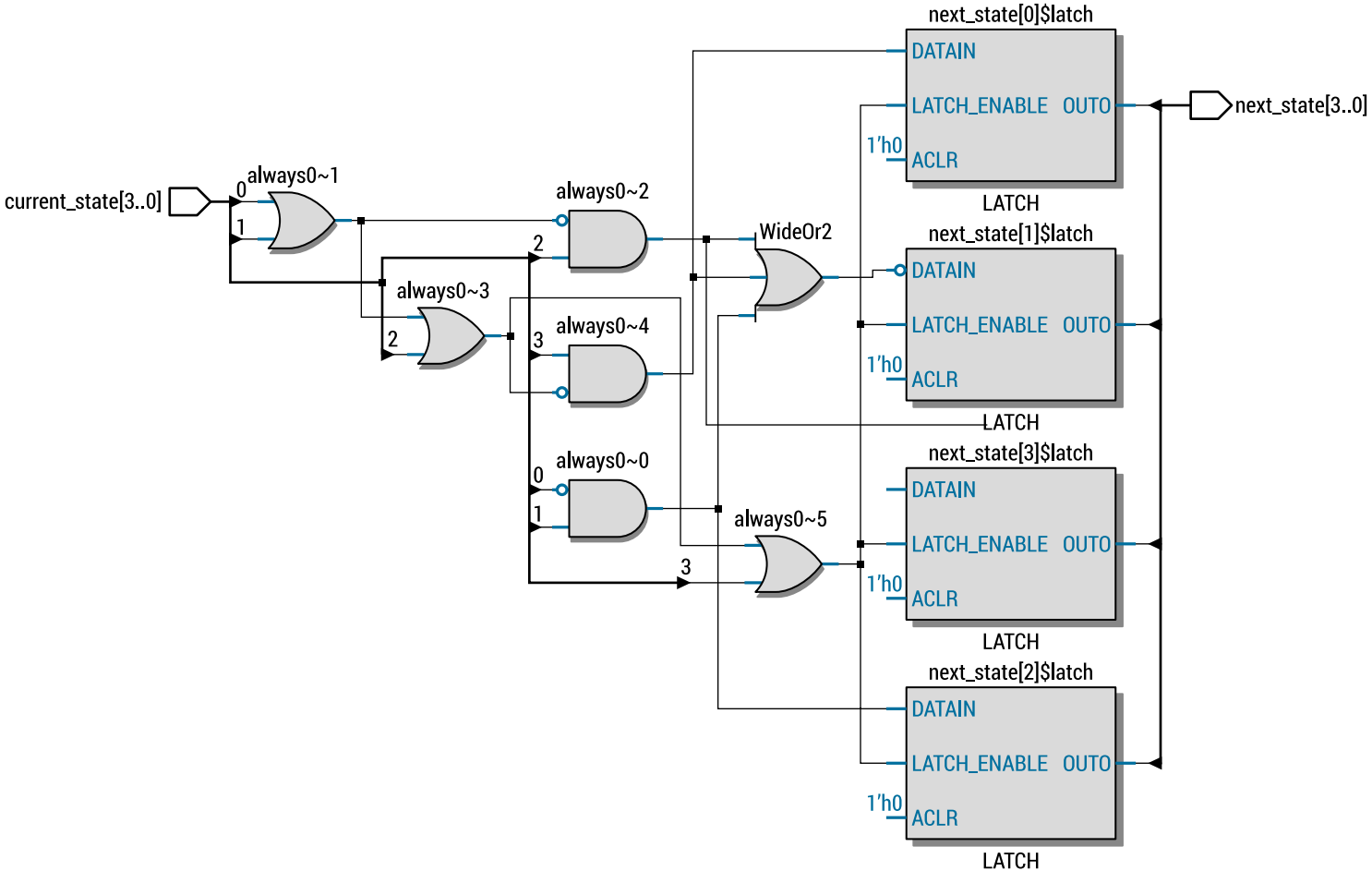
```
module one_hot_assign1 (
        input [3:0] current_state,              // bieżący stan automatu
        output reg [3:0] next_state);           // następny stan automatu

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

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

Realizacja przykładu z listingu 11.2 wymaga 6 elementów logicznych i została przedstawiona na rysunku 11.3.



RYS. 11.3. Realizacja modułu *one_hot_assign1* z listingu 11.2: przykład wykorzystania atrybutu *parallel_case* z operatorem **case**

Realizacja tego przykładu bez wykorzystania atrybutu *parallel_case* wymaga 9 elementów logicznych i została przedstawiona na rysunku 11.4.



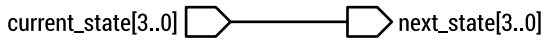
RYS. 11.4. Realizacja przykładu z listingu 11.2 bez atrybutu *parallel_case*

Atrybuty *full_case* oraz *parallel_case* mogą być wykorzystywane wspólnie.

LISTING 11.3. Przykład wspólnego wykorzystania atrybutów *full_case* oraz *parallel_case*

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

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

    always @(*)
    (* parallel_case, full_case *)
    casex (current_state)
        4'b???1 : next_state = s2;
        4'b??1? : next_state = s3;
        4'b?1?? : next_state = s4;
        4'b1??? : next_state = s1;
    endcase
endmodule
```

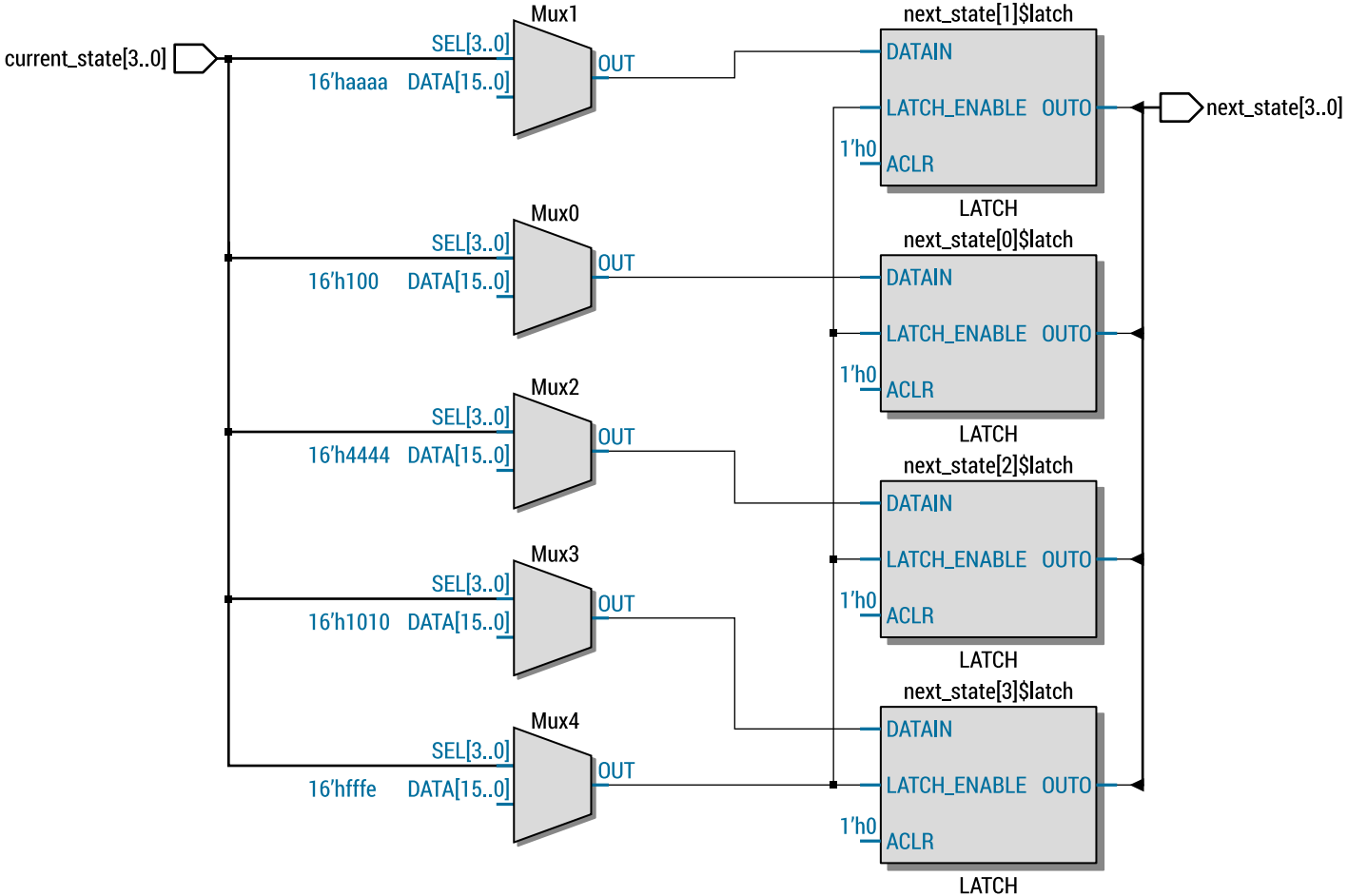
Realizacja przykładu z listingu 11.3 nie wymaga wykorzystywania żadnych elementów logicznych i została przedstawiona na rysunku 11.5.



RYS. 11.5. Realizacja modułu *one_hot_assign2* z listingu 11.3: przykład jednoczesnego wykorzystania atrybutów *full_case* oraz *parallel_case*

Realizacja tego samego przykładu z pominięciem atrybutów *full_case* oraz *parallel_case* wymaga zastosowania 9 elementów logicznych i została pokazana na rysunku 11.6.

Zadanie. Wyjaśnij działanie modułu z listingu 11.3. Sprawdź swoją analizę za pomocą symulatora.



RYS. 11.6. Realizacja przykładu z listingu 11.3 z pominięciem atrybutów *full_case* i *parallel_case*

12. Bloki generacji

12.1. Bloki generacji języka Verilog

Bloki generacji (*generate blocks*) służą do generowania fragmentów kodu języka Verilog. Podobnie do dyrektyw kompilatora w językach programowania, bloki generacji wykonują funkcje preprocesora przekształceń kodu źródłowego przed jego ostateczną kompilacją. Bloki generacji języka Verilog wykonują wstępną obróbkę kodu projektu przed jego przetwarzaniem za pomocą symulatora bądź syntezatora. Bloki generacji języka Verilog umożliwiają tworzenie sekwencji powtarzających się fragmentów kodu, nieznacznie odróżniających się między sobą, wybieranie wykonywanych przez symulator fragmentów kodu w zależności od pewnych warunków i inne. Ponieważ bloki generacji dokonują przekształceń kodu projektu przed jego wykonaniem, to wszystkie wykorzystywane w blokach generacji wyrażenia powinny być stałe.

W językach programowania często instrukcje preprocesora i instrukcje języka znacznie się od siebie różnią (na przykład w językach Assembler, C). W języku Verilog składnia instrukcji wykorzystywanych w blokach generacji odpowiada składni instrukcji samego języka Verilog.

Do wyjaśnienia działania instrukcji generacji wprowadzone zostały następujące pojęcia: *kod źródłowy* oraz *kod kompilowany*. *Kod źródłowy* – jest to kod projektu w języku Verilog stworzony przez projektanta. *Kod kompilowany* – jest kodem projektu po jego obróbce za pomocą bloków generacji. Jeżeli kod źródłowy zawiera bloki generacji, wówczas kod kompilowany różni się od kodu źródłowego (może być zarówno większy, jak i mniejszy, jednak najczęściej jest on większy).

12.2. Składnia bloku generacji

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

genvar *genvar_name*, …;
generate
    genvar *genvar_name*, …;
    generate_items
endgenerate

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

Zmienna typu **genvar** powinna być liczbą całkowitą dodatnią. Służy do tworzenia fragmentów kodu kompilacji na podstawie pewnego fragmentu kodu źródłowego (na zasadzie zmiennej iteracyjnej wewnątrz operatora **for**). Zmienne typu **genvar** wykorzystywane są tylko wewnątrz bloku generacji, mogą mieć przypisaną wartość jedynie w trakcie generacji kodu i nie istnieją w czasie trwania symulacji. Zmienne typu **genvar** mogą być deklarowane zarówno wewnątrz, jak i na zewnątrz bloku generacji. Jeżeli zmienna typu **genvar** jest deklarowana poza blokiem generacji, wówczas może być wykorzystywana też w innych blokach generacji.

Elementami bloku generacji (*generate_items*) mogą być:

- instrukcje przypisania wartości zmiennym typu **genvar** w postaci:
  genvar_name = *constant_expression*;
  gdzie *constant_expression* – wartość stała;
- deklaracje sieci (*net*) i zmiennych (**reg**);
- instancje modułów i prymitywów;
- bloki proceduralne;
- deklaracje procedur i funkcji;
- operatory generacji **if-else**, **case** i **for**.

Uwaga. Należy rozróżnić operatory bloku generacji **if-else**, **case** i **for** od analogicznych operatorów proceduralnych. Operatory bloku generacji **if-else**, **case** i **for** wykonywane są na początku kompilacji w celu przekształcenia kodu wyjściowego, natomiast operatory proceduralne **if-else**, **case** i **for** wykonywane są w trakcie trwania symulacji.

12.3. Operatory generacji

12.3.1. Grupa elementów generacji

W przedstawionych poniżej przykładach składni operatorów generacji zamiast elementu generacji *generate_item* może występować grupa elementów generacji, która posiada następującą składnię:

begin : *generate_block_name*
generate_item
generate_item
…
end

gdzie *generate_block_name* – nazwa bloku generacji.

12.3.2. Operator *if-else*

Operator **if-else** służy do warunkowego włączenia do kodu kompilowanego pewnych fragmentów kodu źródłowego. Składnia operatora generacji **if-else**:

if (*constant_expression*)
    generate_item
else
    generate_item

gdzie **if** i **else** – słowa kluczowe; *constant_expression* – wyrażenie stałe; *generate_item* – element generacji. W przytoczonej składni konstrukcja **else** nie jest obowiązkowa.

Działanie operatora generacji **if-else** jest następujące: w przypadku prawdziwości stałego wyrażenia (*constant_expression*) w kodzie kompilowanym zostanie umieszczony fragment kodu źródłowego odpowiadający elementowi generacji występującemu bezpośrednio za konstrukcją (*constant_expression*). W przeciwnym wypadku, do kodu kompilacji będzie włączony fragment kodu źródłowego umieszczony po słowie kluczowym **else**. W przypadku gdy wyrażenie *constant_expression* nie jest prawdziwe oraz nie występuje konstrukcja **else**, do kodu kompilowanego nie dołącza się żadnych elementów kodu źródłowego.

LISTING 12.1. Przykład wykorzystania operatora generacji **if-else** do wyboru funkcji realizującej różne warianty algorytmu mnożenia w zależności od rozmiaru argumentów [4]

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

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

Uwaga. W listingu 12.1 wykorzystany został stary styl deklaracji portów modułu.

12.3.3. Operator *case*

Operator **case** służy do włączenia do kodu kompilowanego określonych fragmentów kodu źródłowego w przypadku znalezienia odpowiadającej wartości wyliczonego wyrażenia stałego. Składnia operatora generacji kodu **case**:

case (*constant_expression*)
*genvar_value***:** *generate_item*
…
default: *generate_item*
endcase

gdzie **case**, **default** i **endcase** – słowa kluczowe; *constant_expression* – obliczane wyrażenie stałe; *genvar_value* – porównywane wartości (stałe); *generate_item* – elementy generacji. W przytoczonej składni konstrukcja **default** jest opcjonalna.

Działanie operatora **case** jest następujące: na początku zostaje obliczone wyrażenie stałe *constant_expression*. Następnie otrzymana wartość kolejno jest porównywana ze stałymi *genvar_value*. W przypadku znalezienia odpowiedniej wartości jednej ze stałych *genvar_value*, do kodu kompilowanego zostaje włączony element generacji *generate_item* występujący bezpośrednio za daną stałą. Jeżeli wartość wyrażenia *constant_expression* nie posiada odpowiadającej mu stałej, to do kodu kompilowanego zostaje włączony element generacji występujący po słowie kluczowym **default**; w przypadku gdy konstrukcja **default** nie występuje, do kodu kompilowanego nie zostaje włączony żaden element bloku generacji.

Przykład wykorzystania operatora generacji **case** został przytoczony na listingu 12.2.

LISTING 12.2. Przykład wykorzystania operatora generacji **case** do stworzenia instancji sumatora *x1* za pomocą różnych sposobów realizacji w zależności od szerokości słowa

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

12.3.4. Operator *for*

Operator generacji **for** służy do tworzenia w kodzie kompilowanym sekwencyjnych fragmentów, różniących się między sobą wartością zmiennej generacji lub wartością wyrażenia obliczanego na podstawie zmiennej generacji. Składnia operatora generacji **for**:

for (*genvar_name* = *constant_expression*; *constant_expression*;
genvar_name = *constant_expression*)
generate_item

gdzie **for** – słowo kluczowe; *genvar_name* – zmienna generacji; *constant_expression* – wyrażenie stałe; *generate_item* – element generacji.

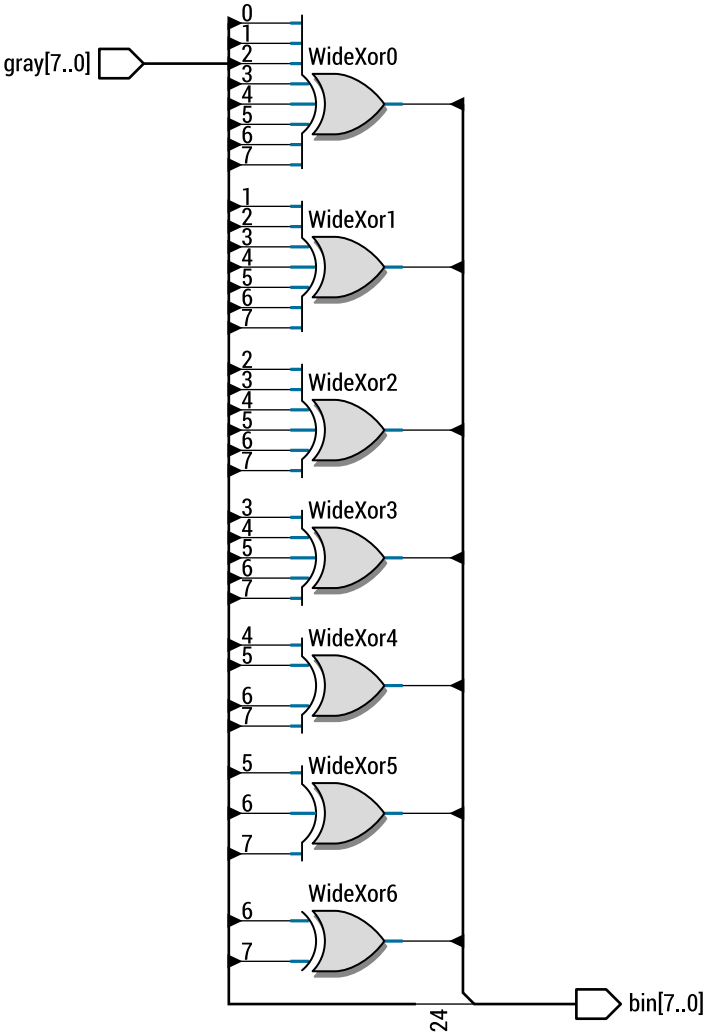
Działanie operatora generacji **for** w zasadzie odpowiada działaniu analogicznego operatora proceduralnego. Pierwsze wyrażenie stałe występujące w nawiasach okrągłych określa początkową wartość zmiennej generacji, drugie wyrażenie określa warunek zakończenia wykonywania operatora **for**, trzecie wyrażenie przeznaczone jest do zmiany wartości zmiennej generacji po wykonaniu każdej iteracji. W wyniku wykonywania operatora **for** w kodzie kompilowanym na podstawie elementu generacji *generate_item* kodu źródłowego będzie stworzona sekwencja fragmentów kodu, które różnią się między sobą jedynie wartością zmiennej generacji *genvar_name* lub wartością wyrażenia obliczanego na podstawie tej zmiennej.

Przykład wykorzystania operatora generacji **for**.

LISTING 12.3. Przykład przekształcenia kodu Graya do kodu binarnego za pomocą operatora generacji **for**

```
module gray_bin # (parameter N = 8)
        (input [N–1:0] gray,          // wartość wejściowa – kod Graya
        output [N–1:0] bin);          // wartość wyjściowa – kod binarny
    genvar i;                         // i – zmienna generacji

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

Wynik syntezy przykładu z listingu 12.3 został pokazany na rysunku 12.1.



RYS. 12.1. Realizacja modułu *gray_bin* z listingu 12.3: przykład wykorzystania bloku generacji i operatora **for** do opisu przekształcenia kodu Graya do kodu binarnego

Uwagi.

1) Każda procedura lub funkcja w operatorach generacji może być deklarowana tylko jeden raz.
2) Wewnątrz pętli operatora **for** nie wolno umieszczać deklaracji procedur ani funkcji.

Zadanie. Opisz za pomocą bloku generacji sekwencyjny sumator na 8 bitów z wykorzystaniem sumatorów jednobitowych.

13. Procedury i funkcje

13.1. Procedury i funkcje w języku Verilog

Procedury i funkcje należą do konstrukcji języka Verilog, które umożliwiają strukturalny opis kodu, co powoduje zwiększenie czytelności i kompaktowości kodu. Ponieważ procedury i funkcje mogą być wielokrotnie wywoływane, za ich pomocą można skrócić kod źródłowy projektu. Oprócz tego, procedury i funkcje pozwalają na wykorzystanie zmiennych lokalnych. Analogicznie do języków programowania, procedury i funkcje są elementami programowania strukturalnego.

Procedury w języku Verilog są zbliżone do podprogramów w językach programowania, podczas gdy funkcje w języku Verilog są podobne funkcjom w językach programowania. Oprócz tego, w języku Verilog wprowadzono pojęcie *funkcji stałych* (*constant function*).

Nie należy zapominać, że głównym elementem konstrukcji projektu w języku Verilog jest moduł, a procedury i funkcje wykorzystywane są jako konstrukcje pomocnicze przy opisie modułów. W przytaczanych niżej przykładach składni zamiast instrukcji (*statement*) można użyć bloku instrukcji wewnątrz nawiasów operatorowych.

13.2. Dynamiczne i statyczne procedury i funkcje

Pojęcie dynamicznych i statycznych procedur i funkcji wywodzi się z języków programowania i jest związane z pracą symulatora. Wszystkie procedury i funkcje domyślnie są statyczne. Oznacza to, że dla wszystkich zmiennych (wejściowych, wyjściowych i lokalnych) podczas symulacji w pamięci komputera zostaje wydzielony statyczny obszar pamięci. Przy każdym wywołaniu procedury lub funkcji do tego obszaru zapisywane są wartości zmiennych.

Problemy statycznej pamięci pojawiają się przy symulacji procesów równoległych, gdy możliwe jest wykonywanie kilku instancji jednej i tej samej procedury lub funkcji. W tym przypadku wartości zmiennych dwóch lub większej liczby równolegle pracujących instancji procedur lub funkcji będą zapisywane w tym samym obszarze pamięci.

W przypadku dynamicznych procedur i funkcji, pamięć dla zmiennych przydzielana jest przy każdym wywołaniu funkcji lub procedury i zwalniana jest po zakończeniu jej pracy. Wykorzystanie dynamicznej pamięci umożliwia jednoczesne wykonywanie kilku instancji procedury lub funkcji. W kontekście pracy symulatora oznacza to, że jest możliwe modelowanie procesów równoległych.

W ten sposób, jeżeli w algorytmie funkcjonowania projektu mogą być równolegle wywoływane różne instancje procedur lub funkcji, wówczas te procedury lub funkcje powinny zostać zadeklarowane jako dynamiczne. W pozostałych przypadkach procedury i funkcje mogą być statyczne.

13.3. Procedury

Jak wcześniej zauważono, procedury w języku Verilog przypominają podprogramy lub procedury w językach programowania. Różnica polega jedynie na tym, że w językach programowania wykorzystuje się listę parametrów do przekazania wartości argumentów do procedury oraz zwracanych wartości, natomiast w procedurach w języku Verilog w tym celu wykorzystuje się spis portów, przez które wchodzą i wychodzą określone wartości.

Procedury deklaruje się wewnątrz modułu i są one elementami lokalnymi w stosunku do danego modułu. Procedury mogą być wywoływane z bloków proceduralnych **initial** oraz **always**, a także z innych procedur.

Procedury mogą posiadać dowolną liczbę wejściowych, wyjściowych i dwukierunkowych portów, jak również w ogóle ich nie posiadać (podobnie jak moduły). Procedury mogą zawierać operatory zarządzania czasem: **#**, **@** lub **wait**.

Deklaracja procedur posiada następującą składnię:

task automatic *task_name* (
port_declaration port_name, port_name, …,
port_declaration port_name, port_name, …);
local_variable_declarations
procedural_statement
endtask

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

Stary styl deklaracji procedury wygląda następująco:

task automatic *task_name*;
port_declaration port_name, port_name, …;
port_declaration port_name, port_name, …;
local_variable_declarations
procedural_statement or statement_group
endtask

Konstrukcja **automatic** nie jest obowiązkowa. Wskazuje ona, że procedura będzie korzystała z dynamicznie przydzielanej pamięci, czyli możliwa jest równoległa praca kilku instancji tej samej procedury.

Konstrukcja *port_declaration* może przyjmować następujące składnie:

port_direction **reg signed** *range* oraz *port_ direction port_type*

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

Wywołanie procedury wygląda następująco:

task_name (*signal, signal*, …);

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

Przykład deklaracji procedury został pokazany na listingu 13.1.

LISTING 13.1. Przykład deklaracji i wywołania procedury, która czyta dane z pamięci

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

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

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

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

13.4. Funkcje

Funkcje w języku Verilog są bardzo podobne do funkcji języka C, jedynie zamiast listy zmiennych wykorzystuje się listę portów. Zarówno funkcja, jak i procedura jest deklarowana wewnątrz modułu oraz jest lokalną w stosunku do danego modułu. W odróżnieniu od procedur, funkcje zawsze zwracają jakąś wartość, która jest utożsamiana z nazwą funkcji. Funkcja może być wywoływana w dowolnym miejscu, gdzie można wykorzystywać wartość wyrażenia. Każda funkcja powinna posiadać co najmniej jeden port wejściowy. Funkcje nie mogą posiadać portów wyjściowych lub dwukierunkowych. Funkcje nie powinny też zawierać operatorów zarządzania czasem (**#**, **@** i **wait**) oraz operatorów nieblokowanego przypisania (<=). Zakłada się, że funkcje zawsze są wykonywane w zerowym czasie symulacji (0.0).

Deklaracja funkcji posiada następującą składnię:

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

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

Stary styl deklaracji funkcji posiada następującą składnię:

function automatic [*range_or_type*] *function_name*;
input *range_or_type port_name, port_name*, …;
input *range_or_type port_name, port_name*, …;
local_variable_declarations
procedural_statement or *statement_group*
endfunction

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

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

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

Wywołanie funkcji posiada następującą składnię:

function_name (*signal*, *signal*, …)

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

Przykład deklaracji funkcji został pokazany na listingu 13.2.

LISTING 13.2. Deklaracja i wywołanie funkcji do obliczenia silni

```
module fact_1 #(parameter LIMIT=32)
               (input [31:0] data,
               output reg [63:0] result);
```

```
// opis funkcji do wyliczania silni
function automatic [63:0] factorial (input reg [31:0] n);
        if (n <= 1)   factorial = 1;
        else          factorial = n * factorial (n-1);      // wywołanie rekurencyjne
endfunction

always
        if (data <= LIMIT)  result = factorial(data);  // wywołanie funkcji factorial
        else                result =0;
endmodule
```

Uwaga. Nie wszystkie kompilatory umożliwiają rekurencyjne wywoływanie funkcji.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator wspiera rekurencyjne wywoływanie funkcji.

13.5. Funkcje stałe

Funkcje stałe w języku Verilog są analogiczne funkcjom preprocesora w językach programowania. Innymi słowy funkcja stała powinna umożliwiać swoje wywołanie (i zwrócić pewną wartość) do rozpoczęcia działania symulatora. Stąd też wywodzi się wiele ograniczeń na deklarację i wykorzystanie funkcji stałych w porównaniu ze zwykłymi funkcjami języka Verilog:

- w funkcjach stałych można wykorzystywać tylko zmienne lokalne;
- nie wolno wykorzystywać sieciowych (*net*) typów danych;
- wartości parametrów wykorzystywanych przez funkcję powinny być określone przed jej wywołaniem;
- wartości parametrów nie mogą być zmieniane za pomocą operatora **defparam**;
- nie wolno wykorzystywać wywołań funkcji stałych przy deklaracji rozmiarów portu;
- w funkcjach stałych zabronione jest wywoływanie funkcji systemowych i procedur;
- w funkcjach stałych nie wolno także korzystać z odsyłaczy i nazw.

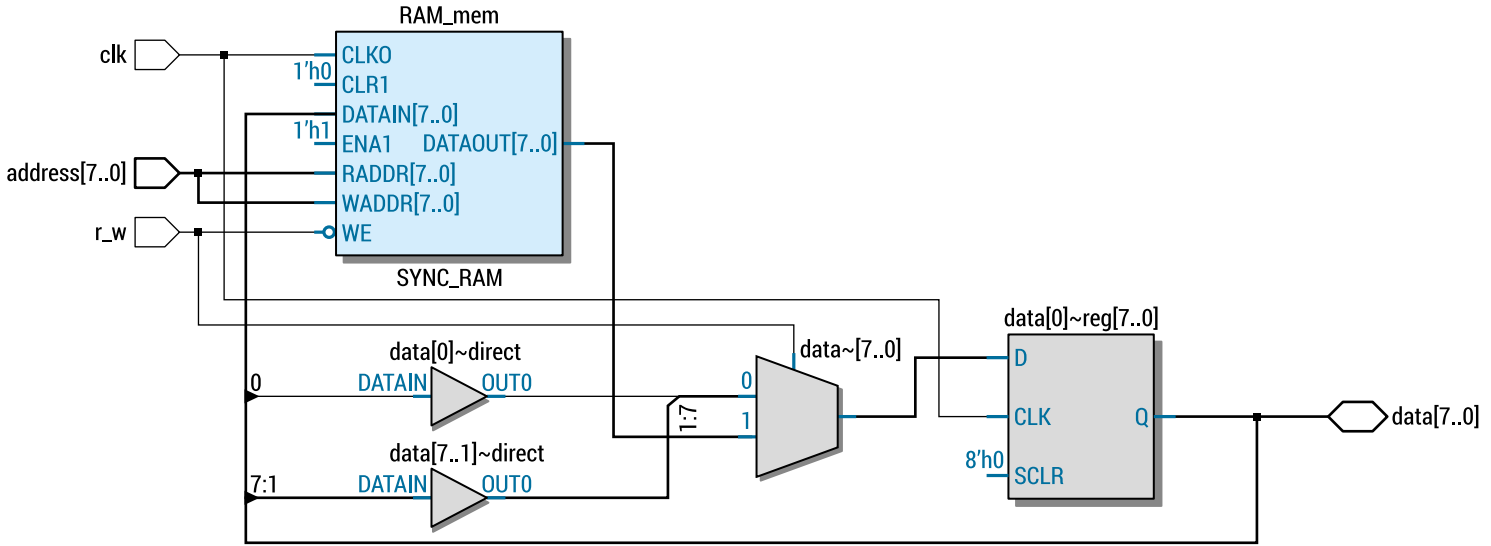
Jednakże funkcje stałe mogą wywoływać inne funkcje stałe.

Przykład wykorzystania funkcji stałej został zaprezentowany na listingu 13.3.

LISTING 13.3. Wykorzystanie funkcji stałej do określenia rozmiaru adresu pamięci

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

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

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

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

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

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

Wynik syntezy przykładu z listingu 13.3 został pokazany na rysunku 13.1.



RYS. 13.1. Realizacja modułu *ram_model* z listingu 13.3: przykład jednoportowej synchronicznej pamięci, przy opisie której wykorzystano funkcję stałą *clogb2*

13.6. Porównanie funkcji i procedur

Główną różnicą pomiędzy funkcjami a procedurami w języku Verilog (podobnie jak w językach programowania) jest to, że wynik wykonania funkcji może być tylko jeden i jest on przypisywany nazwie funkcji. Dzięki temu wywołania funkcji mogą być wykorzystywane w wyrażeniach jako zwykłe operandy. Procedury mogą mieć kilka wyników, które przekazywane są przez wyjściowe i dwukierunkowe porty. Jednakże wywołania procedur nie można wykonywać wewnątrz wyrażeń. W języku Verilog występują pewne dodatkowe cechy przy wykorzystywaniu procedur i funkcji, które zamieszczono w tabeli 13.1.

TABELA 13.1. Porównanie funkcji i procedur

Kategoria	Funkcja	Procedura
Wywołanie (aktywacja)	• Umożliwia wywołanie funkcji wewnątrz wyrażenia, a zwracana wartość może być wykorzystana w danym wyrażeniu. • Możliwe jest wywoływanie funkcji w instrukcjach ciągłego przypisania **assign** i innych.	• Wykonywane jest w postaci osobnej instrukcji. • Wywołanie procedury nie może być wykonane w instrukcjach ciągłego przypisania.
Wywołanie innych procedur i funkcji	• W ciele funkcji można odwoływać się do innych funkcji, między innymi do siebie samej (rekurencja). • Jednakże nie można wywoływać procedur.	• Procedury mogą wywoływać zarówno funkcje, jak i inne procedury.
Wejścia i wyjścia	• Funkcja powinna mieć co najmniej jeden port wejściowy, nie może posiadać portów wyjściowych oraz dwukierunkowych. • Funkcja zwraca jedną wartość, która jest związana z nazwą funkcji.	• Procedura może mieć dowolną liczbę portów dowolnego typu albo nie mieć ich wcale.
Sterowanie czasem wykonywania	• Funkcje nie mogą sterować czasem wykonywania za pomocą operatorów **#**, **@** oraz **wait**	• Procedury mogą sterować czasem wykonywania procedur za pomocą operatorów **#**, **@** oraz **wait**
Operator nieblokowanego przypisania <=	• Funkcje nie mogą zawierać operatora nieblokowanego przypisania **<=**	• Procedury mogą zawierać operator nieblokowanego przypisania **<=**

Podsumowując porównanie procedur i funkcji, można wysnuć następujący wniosek: funkcje są bardziej wygodne do opisu układów kombinacyjnych, a procedury – układów sekwencyjnych.

14. Procedury i funkcje systemowe

14.1. Systemowe procedury i funkcje w języku Verilog

W języku Verilog, podobnie jak w języku programowania C, istnieje wiele wbudowanych procedur i funkcji, które nazywane są *systemowymi*. Systemowe procedury i funkcje w języku Verilog rozpoczynają się od znaku dolara $. Standard języka Verilog IEEE 1364 określa niewielką liczbę systemowych procedur i funkcji. Twórcy komputerowych systemów projektowania układów cyfrowych często definiują dodatkowe procedury i funkcje systemowe, na przykład do wyświetlenia na ekranie diagramów czasowych. Oprócz tego, użytkownicy mogą sami tworzyć systemowe procedury i funkcje za pomocą *języka programowania interfejsu* Verilog (*Programming Language Interface – PLI*).

14.2. Systemowe procedury do obsługi tekstu

Do procedur systemowych języka Verilog służących do wyświetlania tekstu należą: **$display**, **$write**, **$strobe** i **$monitor**. Pierwsze trzy procedury są bardzo do siebie podobne. Wykonywane są, gdy tylko symulator napotka je w kodzie projektu. Procedurę **$monitor** można zapisać w kodzie projektu tylko jeden raz i będzie ona wykonywana przy zmianie wartości dowolnego z argumentów, z wyjątkiem czasu. Domyślnie wartości liczbowe wszystkich wymienionych wyżej procedur są wyświetlane w systemie dziesiętnym. Dodanie do nazwy procedury sufiksów **b**, **o** lub **h** umożliwia domyślne wyświetlanie wartości liczbowych w systemach dwójkowym, ósemkowym lub szesnastkowym.

Procedura systemowa **$display** wypisuje na ekranie formatowaną wiadomość. Procedura **$display** pod wieloma względami przypomina funkcję **printf()** w języku programowania **C**. Składnia procedury **$display**:

$display (“*text_format*”, *list_of_arguments*);

gdzie *text_format* – linia tekstu z symbolami formatującymi; *list_of_arguments* – lista argumentów.

Symbole formatujące zostały przedstawione w tabeli 14.1.

TABELA 14.1. Lista symboli formatujących dla procedury **$display**

Symbol	Opis
%b	wartość binarna
%o	wartość ósemkowa
%d	wartość dziesiętna
%h	wartość szesnastkowa
%e	rzeczywista wartość w notacji naukowej
%f	rzeczywista wartość w postaci dziesiętnej
%t	sformatowana wartość czasu
%s	linia symboli
%m	hierarchiczna nazwa obszaru działania
%l	związana biblioteka konfiguracji
\t	symbol tabulacji
\n	symbol nowej linii
\"	symbol cudzysłowu
****	symbol backslash
%%	symbol procentu

Dodatkowo symbole **%0b**, **%0o**, **%0d** oraz **%0h** usuwają początkowe zera wyświetlanej wartości. Symbole formatowania **%e** i **%f** pozwalają określić szerokość pola (liczbę wypisywanych znaków) wyświetlanego wyrażenia (na przykład, %5.2f określa liczbę znaków dla zmiennej z przecinkiem – 5 znaków dla całej liczby i 2 znaki po przecinku). Symbole **%m** i **%l** nie mają argumentów, gdyż są one określane niejawnie. Wszystkie symbole formatujące nie są rozróżniane wielkością znaków, czyli symbole **%b** oraz **%B** oznaczają to samo.

Po wypisaniu ciągu znaków na ekranie, procedura **$display** wykonuje przejście do nowej linii. Procedura systemowa **$write** działa tak samo, jak procedura **$display**, jednak na końcu wypisywanego ciągu nie dodaje znaku nowej linii. Procedura systemowa **$strobe** także jest podobna do procedury **$display**, za wyjątkiem tęgo, że wyświetlenie tekstu zostaje wstrzymane do momentu, gdy wszystkie modelowane zdarzenia w danym przedziale czasu zostaną wykonane.

Procedura systemowa **$monitor** posiada dokładnie taką samą składnię jak procedura **$display**, jednak działa w zupełnie inny sposób. Jeżeli procedura **$display** jest wykonywana gdy symulator napotka daną instrukcję w kodzie projektu, to procedura **$monitor** wywołuje dodatkowy proces, który stale sprawdza wartości argumentów na liście, i jak tylko co najmniej jeden z nich, za wyjątkiem zmiennej czasu, zmieni swoją wartość, zostaje wyświetlony na ekran sformatowany ciąg.

Przykład wykorzystania procedury systemowej **$display**:

$display (“Czas symulacji %t “, **$time**);

Przykład wykorzystania procedury systemowej **$monitor**:

$monitor ($time, “: A=%b B=%b Cin=%b S=%b Cout=%b”, A, B, Cin, S, Cout);

14.3. Systemowe procedury i funkcje do pracy z plikami

W przypadku dużej liczby informacji wyświetlanej przez symulator na ekranie, często ciężko jest przeanalizować wszystkie otrzymane wyniki. W tym przypadku informację można zapisywać do plików tekstowych, wykorzystując odpowiednie systemowe procedury lub funkcje, a następnie analizować zawartość plików za pomocą dowolnego edytora tekstowego. Analogicznie w przypadku przygotowania dużej liczby informacji dla pracy symulatora, dane można wstępnie umieścić w pliku tekstowym, a następnie, wykorzystując odpowiednie systemowe procedury i funkcje, wczytać z pliku.

14.3.1. Otwieranie i zamykanie plików

Otwarcie pliku w języku Verilog dokonuje się za pomocą funkcji **$fopen**. Funkcja **$fopen** otwiera (lub tworzy) plik na dysku i zwraca wartość deskryptora pliku – 32-bitową wartość typu **integer**. Funkcja **$fopen** posiada następujące formaty składni:

mcd = **$fopen** („*file_name*”);

fd = **$fopen** (“*file_name*”, “*type*”);

gdzie *mcd* – wielokanałowy deskryptor z jedynym ustalonym bitem; *fd* – jednokanałowy deskryptor z kilkoma określeniami bitów; *file_name* – ścieżka i nazwa pliku; *type* – tryb otwierania pliku.

W przypadku jednoczesnego zapisu informacji do kilku plików, deskryptory typu *mcd* mogą być łączone za pomocą operatora **or**. W deskryptorze typu *mcd* zerowy bit jest zarezerwowany dla standardowego strumienia wyjściowego, czyli wypisywania na ekranie. W deskryptorze typu *fd* zawsze jest ustawiony bit 31 i określa się co najmniej jeszcze jeden bit.

Pliki z deskryptorami typu *mcd* zawsze są otwierane tylko do zapisu. Pliki z deskryptorami *fd* mogą być otwierane zarówno do zapisu jak i do odczytu, mogą być także otwarte w trybie dopisywania. Wykorzystując deskryptor typu *fd*, w dowolnym momencie można odczytywać lub zapisywać tylko jeden plik.

Parametr *type* przedstawia ciąg znaków, który określa tryb otwierania pliku. Wartości parametru *type* zostały przedstawione w tabeli 14.2.

TABELA 14.2. Wartości parametru *type* przy otwieraniu plików

Parametry	Opis
r lub rb	otwarcie do odczytu
w lub wb	stworzenie lub otwarcie do zapisu
a lub ab	otwarcie do dopisywania
r+, r+b lub rb+	otwarcie do aktualizacji (odczytu i zapisu)
w+, w+b lub wb+	otwarcie lub stworzenie do aktualizacji
a+, a+b lub ab+	otwarcie lub stworzenie do dopisywania

Otwarty wcześniej plik można zamknąć za pomocą funkcji **$fclose**, która przyjmuje następującą postać:

$fclose (*mcd_or_fd*);

gdzie *mcd_or_fd* – deskryptor typu *mcd* lub *fd* otrzymany przy otwieraniu pliku za pomocą funkcji **$fopen**.

Przykłady otwierania i zamykania plików:

```
fp1 = $fopen(„result.txt”, „w”);     // otwarcie na początku pliku (lub stworzenie)
                                     // do zapisu
fp2 = $fopen(„result.txt”, „a”);     // otwarcie na końcu pliku do dopisywania
fp3 = $fopen(„result.txt”, „r+”);    // otwarcie pliku do odczytu i zapisu
fp4 = $fopen(„result.txt”, „r”);     // otwarcie pliku tylko do odczytu
$fclose(fp1);                        // zamknięcie pliku fp1
```

14.3.2. Zapis do pliku

Zapisywać dane do pliku można za pomocą systemowych procedur **$fmonitor**, **$fdisplay**, **$fwrite** lub **$fstrobe**, które mają następujące składnie:

$fmonitor (mcd_or_fd, “text with format specifiers”, list_of_arguments);
$fdisplay (mcd_or_fd, “text with format specifiers”, list_of_arguments);
$fwrite (mcd_or_fd, “text with format specifiers”, list_of_arguments);
$fstrobe (mcd_or_fd, “text with format specifiers”, list_of_arguments);

gdzie *mcd_or_fd* – deskryptor pliku typu *mcd* lub *fd*; *text_with_format_specifiers* – linia tekstowa z symbolami formatującymi; *list_of_arguments* – lista argumentów.

Wykorzystanie funkcji **$fmonitor**, **$fdisplay**, **$fwrite** oraz **$fstrobe** jest analogiczne do odpowiednich funkcji **$monitor**, **$display**, **$write** oraz **$strobe**, jedynie z tą różnicą, że ma być określony dodatkowy argument w postaci deskryptora pliku, do którego dane mają zostać zapisane.

Przykład wykorzystania funkcji zapisu do pliku:

```
fp = $fopen („result_adding.txt”);   // otwarcie pliku typu mcd
$timeformat (–9, 1, „ns”, 8);        // określenie formatu czasu
$fmonitor (fp, “%t: A=%b B=%b Cin=%b S=%b Cout=%b”, $time, A, B, Cin, S, Cout);
```

14.3.3. Inne funkcje do pracy na plikach

Standard języka Verilog-2001 definiuje cały zestaw funkcji systemowych działających na plikach, podobnych do analogicznych funkcji z języka programowania C. Formaty wywołania tych funkcji podano poniżej:

```
c = $fgetc (fd);
code = $ungetc (c, fd);
code = $fgets (str, fd);
code = $fscanf (fd, format, arguments);
code = $fread (reg_variable, fd);
code = $fread (memory_array, fd, start, count);
position = $ftell (fd);
code = $fseek (fd, offset, operation);
code = $rewind (fd);
errno = $ferror (fd, str);
code = $fflush (mod_or_fd);
```

gdzie c – symbol odczytany z pliku; *code* – kod zakończenia działania funkcji (równy kodowi zakończenia analogicznych funkcji z języka C); *position* – wskaźnik pozycji pliku; *errno* – kod błędu przy pracy z plikiem.

Działanie wymienionych funkcji zostało opisane w tabeli 14.3.

TABELA 14.3. Opis dodatkowych funkcji pracy z plikami

Składnia funkcji	Działanie
c = **$fgetc** (*fd*)	odczyt znaku *c* z pliku *fd*
code = **$ungetc** (*c*, *fd*)	zwrot znaku *c* do pliku *fd*
code = **$fgets** (*str*, *fd*)	odczyt ciągu znaków *str* z pliku *fd*
code = **$fscanf** (*fd*, *format*, *arguments*)	odczyt z pliku *fd* danych sformatowanych zgodnie ze składnią *format* i przypisanie wartości argumentom *arguments*
code = **$fread** (*reg_variable*, *fd*)	odczyt wartości zmiennej typu *reg* z pliku *fd*
code = **$fread** (*memory_array*, *fd*, *start*, *count*)	odczyt obszaru pamięci *memory_array* z pliku *fd* począwszy od pozycji *start*; *count* – liczba odczytywanych znaków
position = **$ftell** (*fd*)	zwraca bieżącą pozycję w pliku *fd*

Składnia funkcji	Działanie
code = **$fseek** (*fd*, *offset*, *operation*)	ustawienie bieżącej pozycji w pliku *fd* od pozycji *operation* przesuniętej o *offset*
code = **$rewind** (*fd*)	przesunięcie bieżącej pozycji pliku *fd* na początek pliku
errno = **$ferror** (*fd*, *str*)	sprawdza czy wystąpił błąd przy pracy z plikiem *fd*; zwraca kod błędu *errno* i komunikat *str*
code = **$fflush** (*mcd_or_fd*)	do pliku *mcd_or_fd* dopisywana jest zawartość wyjściowego bufora

Uwaga. Nie wszystkie kompilatory języka Verilog wspierają dodatkowe funkcje pracy z plikami.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator języka Verilog wspiera dodatkowe funkcje pracy z plikami.

14.4. Inne systemowe procedury i funkcje

14.4.1. Zarządzanie procesem symulacji

Do sterowania procesem symulacji służą dwie funkcje: **$finish** oraz **$stop**, które przyjmują następującą postać:

$finish (*n*)**;**

$stop (*n*)**;**

Procedura **$finish** kończy wykonywanie procesu symulacji. Argument *n* może przyjmować wartości 0, 1 lub 2, które sterują zakresem widocznej na ekranie informacji o zakończonym procesie. Procedura **$stop**(*n*) zatrzymuje proces symulacji i wprowadza interaktywny tryb debugowania.

14.4.2. Zarządzanie czasem symulacji

Aktualny czas symulacji można otrzymać za pomocą funkcji **$time**, **$stime** oraz **$realtime**. Funkcje te nie posiadają argumentów. Funkcja **$time** zwraca aktualny czas symulacji w postaci 64-bitowego wektora; **$stime** – w postaci 32-bitowej wartości całkowitej; **$realtime** – w postaci liczby rzeczywistej.

Postać, w której zwracany jest czas symulacji, określa się za pomocą procedury **$timeformat** posiadającej następującą składnię:

$timeformat (*unit*, *precision*, "*suffix*", *min_field_width*);

Parametr *unit* określa wartość potęgi liczby 10 do określania jednostek czasu, na przykład 0 odpowiada 10^0 = 1 sekunda; –3 odpowiada 10^{-3} = 1 milisekunda;

–6 odpowiada 10^{-6} = 1 mikrosekunda; –9 odpowiada 10^{-9} = 1 nanosekunda; –12 odpowiada 10^{-12} = 1 pikosekunda.

Parametr *precision* (dokładność) określa liczbę cyfr wyświetlanych po przecinku. Parametr *suffix* definiuje ciąg znaków, który jest wyświetlany po wartości czasu, na przykład „ns". Parametr *min_field_width* określa minimalną liczbę wyświetlanych symboli.

Przykład wykorzystania procedury **$timeformat**:

$timeformat (–9, 2, „ns", 10);

Przykład ten określa jednostkę pomiaru czasu jako nanosekundę, będzie wyświetlona z dokładnością dwóch cyfr po przecinku, na końcu wartości zostanie dodany symbol „ns", natomiast do przedstawienia wartości czasu wykorzystuje się pole 10 znaków.

Funkcja **$printtimescale** wyświetla na ekranie skalę czasową (*time scale*) określonego modułu, a jeżeli moduł nie został określony, wówczas wyświetla obszar działania projektu, z którego dany moduł jest wywoływany.

14.4.3. Zmiana wielkości ze znakiem i bez znaku

Dwie funkcje **$signed** oraz **$unsigned** służą do przekształcania liczb ze znakiem w liczby bez znaku i odwrotnie. Format funkcji:

signed_value = **$signed** (*unsigned_value*);
unsigned_value = **$unsigned** (*signed_value*);

gdzie *unsigned_value* – liczba bez znaku; *signed_value* – liczba ze znakiem.

Przykłady wykorzystania funkcji **$signed** i **$unsigned**:

```
reg [7:0] a;                    // deklaracja zmiennej bez znaku a
reg signed [7:0] b;             // deklaracja zmiennej ze znakiem b
a = $unsigned (–4);             // zmienna a otrzymuje wartość 4'b1100
b = $signed (4'b1100);          // zmienna b otrzymuje wartość –4
```

14.4.4. Zapis i odczyt zmiennych z rejestrów

Funkcja **$swrite** umożliwia zapisanie do zmiennej typu **reg**. Działanie funkcji **$swrite** jest podobne do funkcji **$fwrite**, jednakże zapis nie odbywa się do pliku, a do zmiennej typu **reg**. Składnia funkcji **$swrite**:

$swrite (*reg_variable*, *format, arguments*, *format*, *arguments*, …);
$sformat (*reg_variable*, *format*, *arguments*);

gdzie *reg_variable* – zmienna typu **reg**; *format* – łańcuch tekstowy z symbolami formatowania; *arguments* – argumenty.

Funkcja **$sformat** jest podobna do funkcji **$swrite**, jednak zawiera tylko jedną linię z symbolami formatującymi w postaci drugiego argumentu. Do nazwy funkcji **$swrite** jako sufiks mogą być dodawane symbole **b**, **o** oraz **h**, które określają domyślny system prezentacji liczb.

Funkcja **$sscanf** umożliwia odczyt wartości z listy symboli. Składnia funkcji **$sscanf**:

code = **$sscanf**(*str*, *format*, *arguments*);

gdzie *code* – zwracany kod; *str* – lista symboli; *format* – lista tekstowa z symbolami formatującymi; *arguments* – argumenty.

14.4.5. Ładowanie zawartości pamięci

Do ładowania zawartości do pamięci projektu służą dwie funkcje: **$readmemb** oraz **$readmemh**, które posiadają następującą składnię:

$readmemb ("*file_name*", *variable_array*, *start_address*, *end_address*);
$readmemh ("*file_name*", *variable_array*, *start_address*, *end_address*);

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

Adresy *start_address* oraz *end_address* są parametrami opcjonalnymi. Plik, z którego dokonywany jest odczyt danych, reprezentuje jest plikiem tekstowym, w którym dane powinny być dwójkowe (dla funkcji **$readmemb**) lub szesnastkowe (dla funkcji **$readmemh**).

Przykład wykorzystania funkcji **$readmemh**:

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

14.4.6. Konwersja zmiennych typu *real* na wektor 64-bitowy

Jak było wcześniej zauważone, wartość zmiennych typu **real** nie może być bezpośrednio przekazana przez port. Aby to wykonać, zmienną typu **real** wstępnie należy przekształcić w 64-bitowy wektor. Aby wykonać takie przekształcenie (i odwrotne) należy użyć funkcji systemowych **$realtobits** oraz **$bitstoreal**, które przyjmują następującą postać:

64-bit_reg_variable = **$realtobits**(*real_variable*);
real_variable = **$bitstoreal**(*64-bit_reg_variable*);

gdzie *64-bit_reg_variable* – zmienna typu **reg** w postaci wektora 64-bitowego; *real_variable* – zmienna typu **real**.

Przykład wykorzystania funkcji **$realtobits** i **$bitstoreal** został przedstawiony na listingu 14.1.

LISTING 14.1. Przekazanie zmiennych typu **real** przez port

```
module driver (net_r);
        output net_r;
        real r;
        wire [64:1] net_r = $realtobits(r);
endmodule
module receiver (net_r);
        input net_r;
        wire [64:1] net_r;
        real r;
        initial assign r = $bitstoreal(net_r);
endmodule
```

Uwaga. Nie wszystkie kompilatory wspierają typ danych ***real***.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator posiada typ danych **real**.

14.4.7. Funkcje do pracy z wierszem poleceń

Funkcja **$test$plusargs** umożliwia testowanie wiersza poleceń w poszukiwaniu określonych opcji. Opcja w linii poleceń powinna rozpoczynać się znakiem plus (+), jednak znak + nie jest włączany do listy symboli. Jeżeli wskazana opcja wiersza poleceń została znaleziona, wówczas funkcja zwraca wartość 0. Składnia funkcji **$test$plusargs**:

integer = **$test$plusargs** („*invocation_option*”)

gdzie *integer* – zwracana wartość typu **integer**; *invocation_option* – linia symboli zawierająca szukaną opcję.

Przykład wykorzystania funkcji **$test$plusargs**. Niech wywołanie symulatora z linii poleceń zawiera opcję +HELLO. Wówczas rezultat wykonania poniższego fragmentu kodu:

```
initial begin
        if ($test$plusargs(“HELLO”)) $display(“Odebrano argument HELLO.”);
        if ($test$plusargs(“HE”)) $display(“Zawiera napis HE.”);
```

```
        if ($test$plusargs("H")) $display("Pierwsza litera to H.");
        if ($test$plusargs("HELLO_HERE"))$display("Inny argument.");
        if ($test$plusargs("HI")) $display("Zawiera napis HI.");
        if ($test$plusargs("LO")) $display("Nie pasuje.");
end
```

będzie następujący:

```
Odebrano argument HELLO.
Zawiera napis HE.
Pierwsza litera to H.
```

Funkcja **$value$plusargs** przekształca wiersz tekstowy za pomocą określonego formatu i przypisuje wartości do drugiego argumentu. W postaci linii symboli może być wykorzystywana opcja wywołania linii poleceń z symbolem formatowania. Jako symbole formatowania mogą być wykorzystywane następujące symbole: **%b**, **%o**, **%d**, **%h**, **%e**, **%f**, **%g** oraz **%s**.

Zazwyczaj funkcja **$value$plusargs** służy do przedstawienia określonej opcji wywołania z linii poleceń w postaci wartości pewnych zmiennych. Składnia funkcji **$value$plusargs**:

integer = **$value$plusargs** ("*invocation_option=format*", *variable*)

Uwaga. Nie wszystkie opisane w rozdziale 14 funkcje i procedury języka Verilog są wspierane przez wszystkie kompilatory.

Zadania.

1) Sprawdź, jakie z przytoczonych w rozdziale 14 funkcje i procedury wspiera wykorzystywany przez Ciebie kompilator.
2) Sprawdź, jakie dodatkowe procedury i funkcje zawiera wykorzystywany przez Ciebie kompilator.

15. Dyrektywy kompilatora

15.1. Dyrektywy kompilatora w języku Verilog

Dyrektywy kompilatora są poleceniami wskazującymi kompilatorowi, jak powinien interpretować kod źródłowy projektu napisany w Verilogu. Wszystkie dyrektywy kompilatora rozpoczynają się od znaku grawis (`). Ponieważ dyrektywy języka Verilog nie są instrukcjami, to na końcu linii z daną dyrektywą znak średnika (;) nie jest wymagany.

Obszar działania każdej dyrektywy kompilatora nie jest ograniczony do danego modułu czy pliku. Jeżeli kompilator spotyka w kodzie źródłowym dyrektywę, wówczas jej działanie obowiązuje tak długo, póki inna dyrektywa nie zmodyfikuje bądź anuluje działania poprzedniej.

Dyrektywa `**resetall** przywraca wartości domyślne dla tych dyrektyw kompilatora, które je posiadają.

Składnia dyrektywy `**resetall**:

`resetall

15.2. Określenie wartości jednostki czasu

Jednostki czasu są powszechnie wykorzystywane w różnych konstrukcjach języka Verilog do opisu modeli projektu. Wartość (czas trwania) jednej jednostki czasu określa się za pomocą dyrektywy `**timescale**, która posiada następującą składnię:

`timescale *time_unit base* / *precision base*

gdzie *time_unit* – wielkość opóźnienia dla jednej jednostki czasu, może przyjmować wartości 1, 10 lub 100; *base* – podstawa jednostek czasu, może przyjmować wartości *s* (sekundy), *ms* (milisekundy), *us* (mikrosekundy), *ns* (nanosekundy), *ps* (pikosekundy) oraz *fs* (femtosekundy); *precision* – dokładność prezentacji jednostek czasu, określa jednostki czasu do prezentacji czasu po znaku przecinka; *base* występujący po słowie *precision* – podstawa jednostek czasu dla dokładności prezentacji czasu. Przykład wykorzystania dyrektywy `**timescale**:

`timescale 1ns / 10ps

określa jednostki czasowe o długości 1 nanosekundy z dokładnością prezentacji 10 pikosekund (0.01 nanosekundy), czyli dwoma miejscami po przecinku.

Uwaga. Dyrektywa **`timescale** nie posiada wartości domyślnej, dlatego powinna zawsze być określana w projektach, gdzie są wykorzystywane jednostki czasu.

15.3. Makra

Makra przypisują nazwę (etykietę) pojedynczemu łańcuchowi znaków i pozwalają na wygodne, wielokrotne umieszczanie danego łańcucha w kodzie źródłowym, wykorzystując nazwę makra. Przy zamianie kodu źródłowego na kod kompilowany, w każdym miejscu napotkania etykiety makra podstawiony zostanie odpowiedni łańcuch znaków. W języku Verilog *makra* deklaruje się za pomocą dyrektywy **`define**, która przyjmuje następujące dwie postacie:

`define *macro_name text_string*

`define *macro_name* (*arguments*) *text_string* (*arguments*)

gdzie *macro_name* – nazwa makra; przy wywołaniach makra w kodzie projektu należy wykorzystać znak grawis (`) jako przedrostek; *text_string* – łańcuch znaków, na który zostanie zamieniona nazwa makra w kodzie projektu; *arguments* – lista argumentów przekazywanych do makra.

Uwaga. Łańcuch znaków *text_string* wewnątrz makra kończy się symbolem nowej linii (*carriage return*), czyli nie może być dłuższy niż do końca danej linii.

Za pomocą makra można deklarować stałe, proste funkcje i inne podstawienia. Do opisu makr można wykorzystywać komentarze.

Przykłady makr:

```
`define cycle 20                          // okres sygnału synchronizacyjnego cycle
always # (`cycle/2) clk = ~clk;           // generowanie sygnału synchronizacji clk
                                          // z okresem cycle

`define NAND (dval) nand # (dval) // opis prymitywu nand wielkimi literami
`NAND (3)      i1 (y, a, b);      // określenie instancji i1 prymitywu nand
`NAND (3:4:5)  i2 (o, c, d);      // określenie instancji i2 prymitywu nand
```

Do anulowania określonego wcześniej makra służy dyrektywa **`undef** posiadająca następującą składnię:

`undef *macro_name*

15.4. Dyrektywy kompilacji warunkowej

Do dyrektyw warunkowej kompilacji należą następujące dyrektywy: **`ifdef**, **`ifndef**, **`else**, **`elsif** oraz **`endif**. Pozwalają one włączać do kodu projektu określone fragmenty kodu w zależności od tego, czy było wcześniej zdefiniowano makro *macro_name* za pomocą dyrektywy **`define** lub **+define+**. Składnia dyrektyw warunkowych kompilatora:

`ifdef *macro_name*

`ifndef *macro_name*
    verilog_source_code
`else
    verilog_source_code
`elsif *macro_name*
    verilog_source_code
`endif

gdzie *macro_name* – nazwa makra; *verilog_source_code* – fragment kodu wyjściowego.

Przykład wykorzystania dyrektywy kompilacji warunkowej:

```
`ifdef RTL
    wire y = a & b;      // realizacja y w postaci zmiennej sieciowej
`else
    and #1 (y, a, b);    // realizacja y za pomocą prymitywu and
`endif
```

15.5. Załączenie plików

W kodzie źródłowym języka Verilog mogą znajdować się odwołania do dowolnych plików tekstowych za pomocą dyrektywy **`include**. Składnia dyrektywy **`include**:

`include „*file_name*”

gdzie *file_name* – nazwa dołączanego pliku.

Przykład wykorzystania dyrektywy **`include**:

```
`include „project_macros.v”
```

15.6. Określenie domyślnego typu sieciowego

W języku Verilog połączenia sieciowe (*net*) domyślnie posiadają typ **wire**. Domyślny typ sieciowy można zmienić za pomocą dyrektywy **`default_nettype**, która posiada następującą składnię:

`default_nettype *net_data_type*
`default_nettype none

gdzie *net_date_type* – nowy domyślny typ sieciowy.

Dyrektywa **`default_nettype** z parametrem **none** odwołuje niejawne (domyślne) określenie typu sieciowego, czyli typ każdego połączenia sieciowego powinien być obowiązkowo określony w sposób jawny.

Przykład wykorzystania dyrektywy **`default_nettype**:

```
`default_nettype wand      // określenie wand jako domyślnego typu połączeń
                           // sieciowych (net)
```

15.7. Określenie wartości logicznych dla niepodłączonych wejść

W niektórych wypadkach dla bardziej dokładnego wyniku symulacji niezbędne jest określenie wartości niepodłączonych wejść modułów. Do tego celu służą dyrektywy **`unconnected_drive** oraz **`nounconnected_drive**, które posiadają następującą składnię:

`unconnected_drive pull1
`unconnected_drive pull0
`nounconnected_drive

gdzie słowo kluczowe **pull1** oznacza, że niepodłączone wejścia modułów domyślnie będą miały wartość jeden, natomiast **pull0** – zero. Dyrektywy te mogą być umieszczone wyłącznie poza modułami projektu. Dyrektywa **`nounconnected_drive** określa koniec obszaru działania poprzedniej dyrektywy **`unconnected_drive**.

Przykład określenia wartości logicznych na niepodłączonych wejściach:

```
`unconnected_drive pull0   // ustalenie wartości niepodłączonych wejść jako 0.
```

15.8. Określenie wykorzystywanych bibliotek

Za pomocą dyrektywy **`uselib** można wskazać bibliotekę, gdzie kompilator będzie wyszukiwać określonych modułów i prymitywów użytkownika (UDP) w pierwszej kolejności.

Uwaga. Dyrektywa **`uselib** nie jest określona standardem języka IEEE 1364, jednak jest obecna w większości kompilatorów.

Zadanie. Sprawdź, czy wykorzystywany przez Ciebie kompilator wspiera dyrektywę **`uselib**.

Składnia dyrektywy **`uselib**:

`uselib file=*file* dir=*directory* libext=*extension*

gdzie *file* – nazwa pliku (może zawierać ścieżkę pliku) biblioteki użytkownika; *directory* – ścieżka do biblioteki użytkownika; *extension* – format pliku biblioteki użytkownika.

W podanej składni każda z konstrukcji *file*, *dir* oraz *libext* jest opcjonalna. Wykorzystanie dyrektywy **`uselib** bez parametrów anuluje wyszukiwanie bibliotek użytkowników.

Przykłady wykorzystywania dyrektywy **`uselib**:

```
`uselib file=/models/rtl_lib
    ALU i1 (yl, a, b, op);          // wykorzystanie modeli RTL
`uselib dir=/models/gate_lib libext=.v
    ALU i2 (y2, a, b, op);          // wykorzystanie modeli bramek logicznych
`uselib                             // odłączenie bibliotek użytkownika z projektu
```

Uwaga. Nie wszystkie opisane w rozdziale 15 dyrektywy języka Verilog są wspierane przez różne kompilatory.

Zadania.

1) Sprawdź, które z opisanych w rozdziale 15 dyrektyw są wspierane przez wykorzystywany przez Ciebie kompilator.
2) Sprawdź, jakie dodatkowe dyrektywy są wspierane w wykorzystywanym przez Ciebie kompilatorze.

16. Bloki specyfikacji

16.1. Bloki specyfikacji w języku Verilog

Bloki specyfikacji służą do określania parametrów czasowych wykorzystywanych przy pracy symulatora oraz analizatora czasowego.

Składnia bloków specyfikacji wygląda następująco:

specify
specparam_declarations
simple_path_delay
edge-sensitive_path_delay
state-dependent_path_delay
timing_constraint_checks
endspecify

gdzie **specify** oraz **endspecify** – słowa kluczowe ograniczające blok specyfikacji.

Konstrukcja *specparam_declarations* określa parametry bloku specyfikacji i posiada następującą składnię:

specparam *constant_name* = *value*, …;

gdzie **specparam** – słowo kluczowe; *constant_name* – nazwa stałej (parametru); *value* – wartość stałej. Jako wartości parametrów mogą występować liczby całkowite i dziesiętne, opóźnienia oraz łańcuchy w kodzie ASCII.

Przykłady określenia parametrów specyfikacji:

```
specparam   delay_1 = 10,   // inicjacja stałej delay_1 wartością 10
            t_2 = 3:4:6;    // inicjacja t_2 wartością w postaci opóźnienia
                            // min : typ : max
```

Konstrukcja *simple_path_delay* (opóźnienie prostej ścieżki) posiada następujący format:

(*input_port polarity* **:** *path_token output_port*) = (*delay*);

gdzie *input_port* – port wejściowy, *output_port* – port wyjściowy, *delay* – wielkość opóźnienia; konstrukcja *polarity* jest opcjonalna.

Konstrukcja *polarity* może przyjmować jako wartość znak plus (+) albo minus (–). Znak minus (–) wskazuje, że wartość wejścia będzie inwertowana. Konstrukcja *polarity* jest ignorowana w większości symulatorów, jednak może być wykorzystana w analizatorach czasowych.

Konstrukcja *path_token* (token ścieżki) jest określana dwoma symbolami i może mieć następujące wartości: *> – dla pełnego podłączenia, => – dla podłączenia równoległego. Pełne podłączenie *> wskazuje, że sygnał (bit sygnału wejściowego) może być przekazywany do dowolnego wyjścia. Połączenie równoległe => wskazuje, że każdy bit wektora wchodzącego jest połączony z odpowiadającym mu bitem wektora wyjściowego (zerowy – z zerowym, pierwszy – z pierwszym itd.).

Konstrukcja *delay* (wielkość opóźnienia) może być wyrażona pojedynczą wartością lub w postaci następującej składni: *min* : *typ* : *max*, gdzie *min* – wartość minimalna; *typ* – wartość typowa; *max* – wartość maksymalna opóźnienia.

Poszczególne zbiory opóźnień mogą być deklarowane dla 1, 2, 3, 6 lub 12 przełączeń (przejść) sygnałów, co zaprezentowano w tabeli 16.1.

TABELA 16.1. Zbiory wartości opóźnień dla różnej liczby przełączeń sygnału

Opóźnienie	Przedstawiane przełączenia sygnału
1	wszystkie przełączenia
2	rosnące i opadające zbocza sygnału
3	rosnące i opadające zbocza sygnału oraz przejście do stanu wysokiej impedancji
6	rosnące i opadające zbocza sygnału oraz następujące przełączenia: 0→Z, Z→1, 1→Z, Z→0
12	rosnące i opadające zbocza sygnału oraz następujące przełączenia: 0→Z, Z→1, 1→Z, Z→0, 0→X, X→1, 1→X, X→0, X→Z, Z→X

Konstrukcja *edge_sensitive_path_delay* definiuje opóźnienie ścieżki względem aktywnego zbocza sygnału (narastającego lub opadającego) i posiada następującą składnię:

(*edge input_port path_token* (*output_port polarity* **:** *source*)) = (*delay*);

gdzie konstrukcje *input_port*, *path_token*, *output_port* oraz *polarity* zostały opisane powyżej; *edge* (element opcjonalny) może przyjmować wartości **posedge** albo **negedge**, jeżeli nie został on określony, wówczas brane pod uwagę są wszystkie przełączenia wejść; *source* (element opcjonalny) – port wejściowy lub wartość, którą przyjmują wejścia.

Konstrukcja *state_dependent_path_delay* określa opóźnienie czasowe ścieżki zależne od stanu sygnału. Może przyjmować następujące składnie:

if (*first_condition*) *simple_or_edge-sensitive_path_delay*

if (*next_condition*) *simple_or_edge-sensitive_path_delay*
ifnone *simple_path_delay*

gdzie **if** oraz **ifnone** – słowa kluczowe; *first_condition* oraz *next_condition* – wyrażenia, których wartości są rozpatrywane jako warunki; *simple_or_edge-sensitive_path_delay* – opóźnienie prostej ścieżki lub ścieżki czułej na zmianę zbocza sygnału; *simple_path_delay* – opóźnienie prostej ścieżki. Konstrukcji ze słowami kluczowymi **if** może być kilka, konstrukcja ze słowem kluczowym **ifnone** jest opcjonalna.

Dla jednej ścieżki mogą być określone różne opóźnienia. Warunki *first_condition* oraz *next_condition* mogą przyjmować tylko wartości portów wejściowych. W warunkach tych mogą występować dowolne instrukcje, jednak otrzymana wartość musi być prawdą lub fałszem (**X** albo **Z** są przyjmowane jako prawda; jeżeli wynikiem rozpatrzenia warunku jest wektor, wówczas wykorzystuje się jego najmłodszy bit). Każde opóźnienie dla tej samej ścieżki powinno posiadać inne warunki lub być czułe na zmianę innego zbocza. Po słowie kluczowym **ifnone** może być określone opóźnienie tylko dla ścieżki prostej. Opóźnienie to jest stosowane wówczas, gdy żaden z warunków **if** nie posiadał wartości prawdy.

Przykłady deklaracji opóźnień ścieżek [11]:

```
(a => b) = 1.8;          /* ścieżka dla połączeń równoległych, wielkość opóźnienia
                            dla wszystkich przełączanych wyjść */

(a –*> b) = 2:3:4;       /* opóźnienie ścieżki dla pełnego połączenia, dla wszystkich
                            przełączanych wyjść zakresu określone za pomocą formatu
                            min : typ : max, b jest inwersją wartości a */

specparam   t1 = 3:4:6,  // opóźnienia różnych ścieżek dla rosnących i opadających
            t2 = 2:3:4;  // zbocz sygnału
(a => y) = (t1, t2);

(posedge clk => (qb –: d)) = (2.6, 1.8);  /* opóźnienie ścieżki czułej na rosnące
                                             zbocze sygnału synchronizacji clk; qb
                                             jest inwersją wartości d */

if (rst && pst)                          // opóźnienie ścieżki zależne od jej stanu oraz
      (posedge clk => (q +: d)) = 2;     // czułe na rosnące zbocze clk
if (opcode = 3'b000)          // opóźnienie ścieżek, zależne od wartości kodu
        (a,b *> o) = 15;      // z różnymi opóźnieniami dla określonych operacji
if (opcode = 3'b001)
        (a,b *> o) = 25;
ifnone  (a,b *> o) = 10;      // domyślna wartość opóźnienia
```

16.2. Filtrowanie impulsów

W przypadku, gdy impuls generowany na wejściu modułu jest krótszy niż opóźnienie ścieżki, specjalna stała PATHPULSE parametru **specparam** może być wykorzystana do sterowania przepuszczaniem impulsu: czy impuls będzie przesyłany dalej do wyjścia (opóźnienie przesyłu), nie będzie przesyłany do wyjścia (opóźnienie inercyjne) czy prowadzi do nieoznaczoności (**X**) wyniku na wyjściu.

Format specjalnej stałej parametru **specparam**:

specparam PATHPULSE$input$output = (*reject_limit*, *error_limit*);

specparam PATHPULSE$ = (*reject_limit*, *error_limit*);

gdzie: *reject_limit* (zakres odrzucenia) – wielkość opóźnienia (lub zbiór opóźnień w postaci *min* : *typ* : *max*), która jest mniejsza lub równa opóźnieniu ścieżki modułu; dowolny impuls na wejściu, który jest mniejszy lub równy od zakresu odrzucenia, będzie wchłaniany, czyli nie będzie przenoszony na wyjście;
error_limit (ograniczenie błędu) – wielkość opóźnienia (lub zbiór opóźnień w postaci *min* : *typ* : *max*), która jest większa lub równa zakresowi odrzucenia *reject_limit* i mniejsza lub równa opóźnieniu ścieżki; dowolny impuls na wejściu, większy niż *error_limit*, będzie propagowany na wyjście; dowolny impuls mniejszy lub równy *error_limit* i większy niż *reject_limit*, będzie propagowany na wyjście jako wartość nieokreślona **X**.

Drugi format ze słowem kluczowym PATHPULSE$ stosowany jest do wszystkich modułów, które nie mają określonej specjalnej stałej **specparam**. W przypadku, gdy ograniczenie *error_limit* i zakres odrzucenia *reject_limit* są równe, może być wskazywana tylko jedna wartość opóźnienia, a nawiasy okrągłe nie są potrzebne.

Dla większej dokładności propagacji impulsów w języku Verilog-2001 wprowadzono dodatkowe słowa kluczowe **pulsestyle_onevent**, **pulsestyle_ondetect**, **showcancelled** oraz **noshowcancelled**, które mogą być wykorzystywane w poniższy sposób:

pulsestyle_onevent *list_of_path_outputs*;

– wskazuje, że impuls jest propagowany po ścieżce na wyjście jako wartość nieokreślona **X**, z tym samym opóźnieniem, jak zwykła zmiana wejścia propagowana jest na wyjście; takie działanie impulsów jest określone domyślnie;

pulsestyle_ondetect *list_of_path_outputs*;

– wskazuje, że jak tylko impuls będzie wykryty, nieoznaczona wartość **X** będzie propagowana na wyjście bez opóźnień;

showcancelled *list_of_path_outputs*;

– wskazuje, że impuls ujemny (gdy opadające zbocze impulsu pojawia się przed zboczem narastającym) nie jest propagowany na wyjście; takie działanie impulsu jest określone domyślnie;

noshowcancelled *list_of_path_outputs*;

– wskazuje, że impulsy ujemne są rozpatrywane na wyjściach jako wartość nieoznaczona **X**.

16.3. Test ograniczeń czasowych

Test ograniczeń czasowych wykonywany jest za pomocą zadań systemowych, które modelują ograniczenia zmiany sygnałów wejściowych, takie jak czas formowania i czas opóźnienia. Dane zadania systemowe posiadają następujące składnie:

$setup (*data_event*, *reference_event*, *limit*, *notifier*);

$hold (*reference_event*, *data_event*, *limit*, *notifier*);

$setuphold (*reference_event*, *data_event*, *setup_limit*, *hold_limit*, *notifier*, *stamptime_condition*, *checktime_condition*, *delayed_ref*, *delayed_data*);

$recovery (*reference_event*, *data_event*, *limit*, *notifier*);

$removal (*reference_event*, *data_event*, *limit*, *notifier*);

$recrem (*reference_event*, *data_event*, *recovery_limit*, *removal_limit*, *notifier*, *stamptime_cond*, *checktime_cond*, *delayed_ref*, *delayed_data*);

$skew (*reference_event*, *data_event*, *limit*, *notifier*);

$timeskew (*reference_event*, *data_event*, *limit*, *notifier*, *event_based_flag*, *remain_active_flag*);

$fullskew (*reference_event*, *data_event*, *data_skew_limit*, *ref_skew_limit*, *notifier*, *event_based_flag*, *remain_active_flag*);

$period (*reference_event*, *limit*, *notifier*);

$width (*reference_event*, *limit*, *width_threshold*, *notifier*);

$nochange (*reference_event*, *data_event*, *start_edge_offset*, *end_edge_offset*, *notifier*);

Testy czasowe mierzą różnicę pomiędzy referencyjnym zdarzeniem *reference_event* i bieżącym zdarzeniem *data_event*, gdzie:

- *reference_event* oraz *data_event* – porty wejściowe modułu;
- *limit* – wyrażenie stałe, które określa liczbę jednostek czasu, po których rozpoczyna się przekroczenie ograniczenia; wyrażenie może być zbiorem opóźnień w formacie *min* : *typ* : *max*;
- *notifier* – jednobitowa zmienna typu **reg**, która automatycznie się przełącza, gdy tylko testy czasowe wykryją błędy (parametr opcjonalny);
- *stamptime_condition* oraz *checktime_condition* – warunki do umożliwienia lub zabronienia ujemnych testów czasowych (parametry opcjonalne);
- *delayed_ref* oraz *delayed_data* – sygnały opóźniające do ujemnych testów (parametry opcjonalne);

- *event_based_flag* – flaga oparta na zdarzeniu; gdy jest ustawiona, wówczas uruchamia test czasowy, który jest oparty na zdarzeniu (parametr opcjonalny);
- *remain_active_flag* – flaga aktywacji; gdy jest ustawiona, wówczas uruchamia test czasowy, który aktywuje się w momencie przyjścia komunikatu o pierwszym naruszeniu (parametr opcjonalny);
- *start_edge_offset* oraz *end_edge_offset* – wartości przesunięcia opóźnień (mogą być dodatnie lub ujemne), zwiększają bądź zmniejszają czas, w którym zmiana nie może być dokonana.

17. Konfiguracja projektu

17.1. Konfiguracje

Konfiguracje (*configurations*) są zbiorem wskazówek do określenia dokładnej lokalizacji kodu źródłowego z opisem każdej instancji modułu czy prymitywu, który był wykorzystywany w projekcie. Konfiguracje określa się za pomocą *bloków konfiguracyjnych* (*configuration blocks*). Blok konfiguracji jest częścią kodu źródłowego projektu w języku Verilog i jest opisywany razem z kodem projektu. Jednakże bloki konfiguracji opisuje się poza modułami projektu. Mogą występować zarówno w tych samych plikach co kod projektu, jak i w plikach dodatkowych.

Podstawową jednostką przy konfiguracji projektu jest *komórka* (*cell*). Komórka – jest to nazwa modułu, prymitywu lub innej konfiguracji. W blokach konfiguracji wykorzystuje się nazwy bibliotek symbolicznych (*symbolic library*). Biblioteka symboliczna reprezentuje logiczny zbiór komórek (czyli nazw). Nazwa każdej komórki powinna być zgodna z nazwą odpowiedniego modułu, prymitywu lub konfiguracji.

Do odwzorowania nazw symbolicznych bibliotek na fizyczne położenie plików z kodami źródłowymi wykorzystuje się *pliki mapowania bibliotek* (*library map files*). Informacja wiążąca bibliotekę symboliczną z egzemplarzem modułu może być wykorzystywana w czasie symulacji za pomocą symbolu formatującego **%l**, który będzie wyświetlał nazwę biblioteki i nazwę modułu w postaci:

library_name.cell_name

gdzie *library_name* – nazwa biblioteki symbolicznej; *cell_name* – nazwa komórki odpowiadającej modułowi, w którym wykonywana jest instrukcja wyświetlenia na ekran.

17.2. Bloki konfiguracyjne

Bloki konfiguracyjne zawierają zbiór wskazówek do wyszukiwania kodów źródłowych poszczególnych instancji komponentów projektu.

Blok konfiguracji posiada następującą składnię:

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

Występujące tutaj **config** oraz **endconfig –** są słowami kluczowymi ograniczającymi blok konfiguracji. Operator **design** wskazuje bibliotekę oraz moduł najwyższego poziomu w hierarchii projektu. W bloku konfiguracji może być tylko jeden operator **design**, jednakże może być wymienionych w nim kilka modułów wyższego poziomu. Operator **design** powinien być pierwszą instrukcją w bloku konfiguracji. Opcjonalna nazwa biblioteki *lib_name* określa, jaka biblioteka symboliczna zawiera daną komórkę. Jeżeli nazwa biblioteki nie występuje, wówczas do poszukiwania modułu najwyższego poziomu wykorzystuje się lokację, która zawiera blok konfiguracji. Obowiązkowa nazwa komórki *cell_name* jest nazwą modułu najwyższego poziomu w hierarchii projektu, który przedstawia blok konfiguracji.

Operator **default liblist** określa domyślną kolejność przeszukiwania bibliotek w celu odnalezienia kodów źródłowych instancji modułów, prymitywów czy innych konfiguracji, dla których nie zostaną określone konkretne lokalizacje. Biblioteki są przeszukiwane w porządku ich wymienienia. Dla niektórych projektów lista bibliotek może zawierać wszystkie biblioteki, które są niezbędne do określenia konfiguracji. Parametr *list_of_library_names* przedstawia listę nazw bibliotek symbolicznych, oddzielonych od siebie przecinkami.

Operator **cell** określa konkretny zbiór bibliotek, w których wyszukiwane są kody źródłowe dla podanej nazwy modułu lub prymitywu (zamiast bibliotek i porządku określonego w operatorze **default**). Opcjonalny element *lib_name* określa bibliotekę symboliczną, która zawiera komórkę; obowiązkowy element *cell_name* jest nazwą modułu lub prymitywu.

Operator **instance** określa konkretną grupę bibliotek, w których wykonywane jest wyszukiwanie zadanego egzemplarza modułu lub prymitywu (zamiast bibliotek i/lub porządku określonego w operatorze **default**). Element *hierarchy_name* jest pełną ścieżką hierarchiczną nazwy egzemplarza modułu lub prymitywu.

Opcjonalna konstrukcja **use** wskazuje położenie konkretnej komórki lub egzemplarza komórki (zamiast poszukiwania komórki w bibliotekach operatora **default**). Element *lib_name* określa nazwę odpowiedniej biblioteki zawierającej komórkę; element *config_name* wskazuje na wykorzystanie innego bloku konfiguracji do wyszukiwania egzemplarza komórki. Operator **design** w takim bloku konfiguracji określa aktualną wiążącą informację.

17.3. Pliki biblioteczne

Do odzwierciedlenia bibliotek symbolicznych w postaci rzeczywistych plików wykorzystuje się osobny plik, który nazywa się *mapą bibliotek* (*library map file*). Jeżeli w czasie pracy nad projektem z pewnego powodu zmieni się położenie plików z kodami źródłowymi, wówczas należy zmodyfikować jedynie plik mapy bibliotek. Jednocześnie kody źródłowe w języku Verilog i bloki konfiguracyjne nie muszą być zmieniane. Plik mapy bibliotek może zawierać operatory **library** i **include**, oraz komentarze. Plik mapy bibliotek nie jest kodem źródłowym języka Verilog.

Operatory **library** oraz **include** posiadają następującą składnię:

library *lib_name list_of_file_paths*, **-incdir** *list_of_file_paths*;
include *library_map_file_path*;

Element *lib_name* określa nazwę biblioteki symbolicznej, która będzie używana w blokach konfiguracji. Element *list_of_file_paths* jest listą ścieżek do konkretnych plików. Ścieżka, która kończy się symbolem ukośnika (/), zawiera wszystkie pliki w określonym katalogu (tożsame do ścieżki, która kończy się symbolem /*). Ścieżka, która nie jest zakończona symbolem ukośnika (/), odpowiada katalogowi, w którym powinien znajdować się plik mapy bibliotek. Do określania ścieżek można wykorzystywać następujące znaki specjalne:

? – znak wieloznaczności, odpowiada dowolnemu pojedynczemu symbolowi;
* – znak kilku symboli wieloznaczności, odpowiada dowolnej liczbie dowolnych symboli;
… – hierarchiczny znak wieloznaczności, odpowiada dowolnej liczbie hierarchicznych katalogów;
.. – określa katalog rodzicielski;
. – określa dany katalog, zawierający plik *lib.map*.

Konstrukcja **-incdir** określa, gdzie należy szukać załączanych plików, na które wskazują dyrektywy `**include** w kodzie źródłowym projektu.

Mapa bibliotek może odnosić się także do innych plików mapowania bibliotek, które mogą być załączone za pomocą konstrukcji **include**.

17.4. Przykłady konfiguracji projektów

17.4.1. Kod źródłowy projektu

Niech projekt będzie opisany za pomocą czterech plików top.v, adder.v, adder.vg oraz lib.map, które mają następującą zawartość:
file top.v:

module top(...);

```
    ...
     adder a1(...);
     adder a2(...);
     endmodule
     module foo(...);
    ... // rtl
     endmodule

file adder.v:
     module adder(...);
    ... // rtl
     foo f1(...);
     foo f2(...);
     endmodule
     module foo(...);
    ... // rtl
     endmodule

file adder.vg:
     module adder(...);
    ... // gate-level
     foo f1(...);
     foo f2(...);
     endmodule
     module foo(...);
    ... // gate-level
     endmodule

file lib.map:
     library rtlLib top.v;
     library aLib adder.*;
     library gateLib
     adder.vg;
```

W przytoczonym przykładzie pliki top.v, adder.v oraz adder.vg kompilowane są razem z załączonym plikiem bibliotecznym lib.map, który posiada następującą strukturę:

```
rtlLib.top // from top.v
```

rtlLib.foo // from top.v
aLib.adder // from adder.v
aLib.foo // rtl from adder.v
gateLib.adder // from adder.vg
gateLib.foo // from adder.vg

17.4.2. Wykorzystanie konfiguracji zawartej w plikach bibliotecznych

W przypadku braku jakichkolwiek bloków konfiguracji w projekcie, biblioteki są przeszukiwane zgodnie z porządkiem ustalonym w pliku map bibliotek. Oznacza to, że wszystkie egzemplarze modułu *adder* powinny wykorzystywać plik *aLib.adder*, jako że *aLib* jest pierwszą biblioteką, której specyfikacja zawiera komórkę z nazwą *adder*. Oprócz tego, wszystkie egzemplarze modułu *foo* powinny wykorzystywać plik *rtlLib.foo*, ponieważ *rtlLib* jest pierwszą biblioteką, która zawiera komórkę o nazwie *foo*.

17.4.3. Wykorzystanie operatora *default*

Aby zawsze wykorzystywać moduł *foo*, opisany w pliku *adder.v*, można zdefiniować następującą konfigurację:

```
config cfg1;
    design rtlLib.top;
    default liblist aLib rtlLib;
endconfig
```

Występujący tu operator **default liblist** zmienia porządek wyszukiwania bibliotek ustalony w pliku *lib.map*. Dlatego biblioteka *aLib* zawsze jest wybierana przed biblioteką *rtlLib*. Jako że biblioteka *gateLib* nie jest wymieniona w **liblist**, opis modułów *adder* oraz *foo* na poziomie bramkowym jest niemożliwy.

W celu umożliwienia wykorzystania opisów modułów *adder* oraz *foo* na poziomie bramek, należy dodać następującą konstrukcję:

```
config cfg2;
    design rtlLib.top;
    default liblist gateLib aLib rtlLib;
endconfig
```

Dana konstrukcja wskazuje, że w pierwszej kolejności należy przeszukiwać bibliotekę *gateLib* i wybrać opis modułów *adder* oraz *foo* na poziomie bramkowym z pliku *adder.vg*.

17.4.4. Wykorzystanie operatora *cell*

Następująca konfiguracja umożliwia wykorzystanie opisu *rtl* dla modułu *adder* i bramkową reprezentację modułu *foo* z biblioteki *gateLib*:

```
config cfg3;
    design rtlLib.top;
    default liblist aLib rtlLib;
    cell foo use gateLib.foo;
endconfig
```

Zastosowany tu operator **cell** wybiera wszystkie komórki o nazwie *foo* i jawnie przypisuje ich do bramkowej reprezentacji w bibliotece *gateLib*.

17.4.5. Wykorzystanie operatora *instance*

Kolejna konstrukcja umożliwia wykorzystanie dla instancji *top.a1* modułu *adder* (i jej potomków) opisu na poziomie bramek, natomiast dla instancji *top.a2* modułu *adder* (i jej potomków) – reprezentacji *rtl* z biblioteki *aLib*:

```
config cfg4
    design rtlLib.top;
    default liblist gateLib rtlLib;
    instance top.a2 liblist aLib;
endconfig
```

Ponieważ **liblist** jest dziedziczne, wszystkie potomki instancji *top.a2* dziedziczą jej ustawienia **liblist** z operatora **instance**.

17.4.6. Wykorzystanie konfiguracji hierarchicznej

Niech projekt będzie opisany w taki sposób, że kolejna konfiguracja wykorzystuje komórkę *rtlLib.foo* dla instancji *f1* i komórkę *getLib.foo* dla instancji *f2*:

```
config cfg5;
    design aLib.adder;
    default liblist gateLib aLib;
    instance adder.f1 liblist rtlLib;
endconfig
```

Poniższy przykład umożliwia wykorzystanie konfiguracji *cfg5* dla instancji *top.a2* modułu *adder*, natomiast dla instancji *top.a1* moduł *adder* wybierany jest domyślnie z biblioteki *aLib*:

```
config cfg6;
        design rtlLib.top;
        default liblist aLib rtlLib;
        instance top.a2 use work.cfg5:config
endconfig
```

Zdefiniowany tu operator **instance** przywołuje konfigurację *work.cfg5:config* dla instancji *top.a2* i jej potomków. Operator **design** w konfiguracji *cfg5* określa dokładne przywiązanie do egzemplarza *top.a2*. Pozostała część konfiguracji *cfg5* określa zasady dla tworzenia potomków instancji *top.a2*. Warto zauważyć, że operator **instance** w konstrukcji *cfg5* jest krewnym swojego modułu *adder* wyższego poziomu.

18. Syntezowalne konstrukcje języka Verilog

18.1. Cechy wspólne

Z uwagi na to, że język Verilog przeznaczony jest zarówno do syntezy, jak i do modelowania układów cyfrowych, nie wszystkie konstrukcje języka mogą być syntezowane, czyli fizycznie realizowalne na sprzęcie. Standard IEEE 1364.1 Verilog Register Transfer Level Synthesis określa te konstrukcje języka Verilog, które powinny być realizowane przez środowiska programowe syntezy (syntezatory).

Uwaga. Nie wszyscy twórcy kompilatorów języka Verilog wspierają rekomendacje standardu IEEE 1364.1.

Zadanie. Sprawdź, jakie konstrukcje języka Verilog wspiera, a jakich nie wspiera syntezator wykorzystywanego przez Ciebie środowiska do projektowania.

Znajomość konstrukcji języka Verilog wspieranych przez konkretny syntezator jest niezbędna, aby mieć pewność, że opisywany projekt może być realizowany sprzętowo. Syntezowalne konstrukcje języka Verilog, które określa standard IEEE 1364.1, przedstawiono w tabeli 18.1.

TABELA 18.1. Konstrukcje języka Verilog wspierane przez syntezator

Konstrukcja	Oznaczenie	Opis
Deklaracja modułów		• w pełni wspierana
Deklaracja portów	**input**, **output**, **inout**	• w pełni wspierana dla dowolnych rozmiarów wektorów
Sieciowe typy danych	**wire**, **wand**, **wor**, **supply0**, **supply1**	• w pełni wspierane, zarówno wartości skalarne jak i wektory
Typy danych zmiennych	**reg**, **integer**	• mogą być wartości skalarne, wektory lub macierze zmiennych; • mogą być ograniczane wykorzystaniem zmiennej tylko w jednym operatorze proceduralnym; • domyślnie zmienna typu **integer** zajmuje 32 bity
Parametry	**parameter**	• stałe wartości mogą być ograniczone liczbami całkowitymi; • ponowne nadanie wartości parametrom może nie być wspierane

Konstrukcja	Oznaczenie	Opis
Liczby całkowite		• w pełni wspierane; • wspierane wszystkie rozmiary i systemy liczbowe
Instancje modułów		• w pełni wspierane; • wspierane przekazywanie wartości sygnałów zarówno za pośrednictwem nazw portów, jak i ich porządku
Instancje prymitywów	**and**, **nand**, **or**, **nor**, **xor**, **buf**, **not**, **bufif1**, **bufif0**, **notif1**, **notif0**	• w pełni wspierane;
Operator nieprzerwanego przypisania	**assign**	• w pełni wspierany, zarówno jawna jak i niejawna forma
Proceduralny operator nieprzerwanego przypisania	**assign**	• w pełni wspierany; • może nie wspierać konstrukcji **deassign**
Deklaracja funkcji	**function**	• mogą być wykorzystywane tylko wspierane konstrukcje
Deklaracja zadań	**task**	• mogą być wykorzystywane tylko wspierane konstrukcje
Blok proceduralny **always**	**always**	• powinna być wspierana lista czułości
Operator bloku	**begin-end**	• w pełni wspierany, zarówno bloki nazwane jak i bez nazwy
Operator bloku	**fork-join**	• nie jest wspierany
Operator blokowanego proceduralnego przypisania	**=**	• w pełni wspierany; może być ograniczone wykorzystaniem tylko jednego typu przypisania dla jednej zmiennej
Operator nieblokowanego proceduralnego przypisania	**<=**	• w pełni wspierany; może być ograniczony wykorzystaniem tylko jednego typu przypisania dla wszystkich przypisań tej samej zmiennej
Operator wyboru	**if_else**, **case**, **casez**, **casex**	• logiczne wartości bitów **X** oraz **Z** są wspierane jako wartość nieokreślona (*don't care*)
Operator pętli	**for**	• zmienna iteracyjna może być zwiększana lub zmniejszana tylko o jeden
Operatory pętli	**while**, **forever**	• pętla powinna posiadać jeden sygnał synchronizujący dla każdej iteracji pętli, tzn. wewnątrz petli powinna być wykorzystywana konstrukcja @ (**posedge** clk) albo @ (**negedge** clk)

Konstrukcja	Oznaczenie	Opis
Operator **disable**	**disable**	• powinien być wykorzystany wewnątrz jednego i tego samego nienazwanego bloku, który przerywa
Operatory	&, ~&, |, ~|, ^, ~^, ^~, ==, !=, <, >, <=, =>, !, &&, ||, <<, >>, {}, {{}}, ?:, +, –, *, /	• operandami instrukcji mogą być: skalar, wektor, stała lub zmienna
Operatory	= = =, ! = =	• nie są wpierane
Wybór bitu lub części wektora	[*i*], [*msb*:*lsb*]	• w pełni wspierany z prawej strony operatora przypisania; • wartości *i*, *msb* oraz *lsb* powinny być stałymi w lewej części operatora przypisania

Oprócz tego syntezator powinien wspierać następujące konstrukcje języka Verilog-2001:

- listy czułości;
- @* – dla syntezy schematów kombinacyjnych;
- zamiana miejsc przy deklaracji portów i danych;
- deklaracja portów w stylu ANSI C;
- niejawne sieci z operatorami nieprzerwanego przypisania **assign**;
- macierze wielowymiarowe;
- wybór bitów i pól bitowych macierzy;
- typy danych ze znakiem;
- literały znakowe i liczbowe;
- operacje przesunięć arytmetycznych <<< oraz >>>;
- operacje potęgowania **, jednak może posiadać ograniczenia;
- funkcje rekurencyjne, może być ograniczona liczba wywołań rekurencyjnych funkcji;
- parametry wielkościowe;
- jawny przekaz parametrów w linii poleceń;
- dyrektywy kompilatora `**ifndef** oraz `**elsif**.

18.2. Konstrukcje języka Verilog wspierane przez środowisko Quartus

W tabelach 18.2-18.12 zostały przytoczone konstrukcje języka Verilog w standardzie IEEE 1364-2001, które są wspierane przez środowisko pakietu Quartus firmy Intel. W pierwszej kolumnie każdej z tabel 18.2-18.12 zaznaczono numer odpowiedniego rozdziału standardu IEEE 1364-2001, w którym opisywana jest dana konstrukcja.

TABELA 18.2. Typy danych

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
3.2	Sieci (*nets*) i zmienne (**reg**)	wspierane, z wyjątkiem typów **real** oraz **realtime**
3.3	Wektory	wspierane
3.4	Moce (*strengths*)	ignorowane przy syntezie
3.5	Niejawne deklaracje	wspierane
3.6	Deklaracja sieci	wspierana
3.7.1	**wire** oraz **tri**	wspierane
3.7.2	**wor**, **wand**, **trior** oraz **triand**	wspierane
3.7.3	**trireg**	brak wsparcia
3.7.4	**tri0** oraz **tri1**	wspierane
3.7.5	**supply0** oraz **supply1**	wspierane
3.9	**integer** oraz **time**	wspierane
3.9	**real** oraz **realtime**	brak wsparcia
3.11	Parametry (*parameters*)	wspierane

TABELA 18.3. Instrukcje

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
4.1.5	Operacje arytmetyczne	wspierane
4.1.7	Operacje porównania	wspierane
4.1.8	Operacje równości	wspierane
4.1.9	Operacje logiczne	wspierane
4.1.10	Operacje bitowe	wspierane
4.1.11	Operacje redukcji	wspierane
4.1.12	Operacje przesunięcia	wspierane
4.1.13	Instrukcja warunkowa	wspierana
4.1.14	Operacja konkatenacji	wspierana
4.1.15	Instrukcja **or** na liście czułości	wspierana
4.2.1	Wybór bitów i pól bitowych	wspierany
4.2.2	Adresacja elementów pamięci	wspierana
4.2.3	Operacje na łańcuchach: przypisanie, konkatenacja i porównanie	wspierane
4.3	Wyrażenia dla minimalnej, zwyczajowej i maksymalnej wartości opóźnienia	wspierane; ignorowane przy syntezie

TABELA 18.4. Operatory przypisania

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
6.1	Przypisanie ciągłe	wspierane
6.2	Przypisanie proceduralne	wspierane

TABELA 18.5. Prymitywy

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus II
7.2	**and**, **nand**, **nor**, **or**, **xor** oraz **xnor**	wspierane
7.3	**buf** oraz **not**	wspierane
7.4	**bufif1**, **bufif0**, **notif1** oraz **notif0**	wspierane
7.5	Prymitywy MOS	brak wsparcia
7.6	Dwukierunkowe elementy przełączające	brak wsparcia
7.7	Prymitywy CMOS	brak wsparcia
7.8	Prymitywy **pullup** oraz **pulldown**	brak wsparcia

TABELA 18.6. Prymitywy definiowane przez użytkownika (UDP)

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
8.2	Kombinacyjne	wspierane
8.3, 8.4	Sekwencyjne	wspierane

TABELA 18.7. Proceduralne operatory i bloki

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
9.2.1	Blokowane proceduralne przypisanie	wspierane
9.2.2	Nieblokowane proceduralne przypisanie	wspierane
9.3	Proceduralne operatory przypisania ciągłego	brak wsparcia
9.3.1	Operatory **assign** i **deassign**	brak wsparcia
9.3.2	Operatory **force** i **release**	brak wsparcia
9.4	Operator **null**	wspierany dla wszystkich operatorów, za wyjątkiem operatora **for**
9.4	Operator warunkowy **if-else**	wspierany
9.5	Operator **case**	wspierany
9.6	Operatory pętli **forever**, **repeat**, **while** oraz **for**	wspierane

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
9.7.1	Operator opóźnienia **#**	wspierany; ignorowany podczas syntezy
9.7.2	Operator czułości **@**	wspierany tylko przy operatorze **always**
9.7.3	Nazwane zdarzenia	brak wsparcia
9.7.4	Instrukcja **or** na liście czułości	wspierana
9.7.5	Operator **wait**	brak wsparcia
9.7.6	Wewnętrzne opóźnienia	Wspierane; ignorowane podczas syntezy
9.8.1	Bloki sekwencyjne (nawiasy operatorowe **begin-end**)	wspierane
9.8.2	Bloki równoległe (nawiasy operatorowe **fork-join**)	brak wsparcia
9.9.1	Operator **initial**	wspierany do określenia warunków po podłączeniu zasilania
9.9.2	Operator **always**	wspierany
11	Operator **disable**	wspierany

TABELA 18.8. Procedury i funkcje

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
10.2	Procedury	wspierane
10.3	Funkcje	wspierane, tylko gdy zmienne wewnątrz funkcji są lokalne dla danej funkcji

TABELA 18.9. Struktury hierarchiczne

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
12.1	Moduły	wspierane
12.2	Parametry	wspierane
12.2.1	Operator **defparam**	wspierany
12.2.2	Operator przypisania wartości parametrów instancji modułu	wspierany
12.3	Porty	wspierane
12.4	Nazwy hierarchiczne	wspierane; są dopuszczalne odnośniki tylko do obiektów wewnątrz aktualnego modułu

TABELA 18.10. Bloki specyfikacji

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
13.1	Bloki specyfikacji	ignorowane podczas syntezy
13.2	Parametry bloku specyfikacji (*specparams*)	ignorowane podczas syntezy
13.3	Opóźnienia ścieżek modułów w bloku specyfikacji	ignorowane podczas syntezy

TABELA 18.11. Systemowe procedury i funkcje

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
14.1	Systemowe procedury wyświetlania	wspierane, ignorowane podczas syntezy
14.2	Procedury systemowe wejścia-wyjścia do pliku	wspierane tylko procedury **$readmemb** oraz **$readmemh**
14.3	Systemowa procedura $**timescale**	brak wsparcia
14.4	Systemowa procedura sterowania symulacją	brak wsparcia
14.5	Systemowa procedura testów czasowych	brak wsparcia
14.6	Systemowa procedura modelowania macierzy programowania logicznego	brak wsparcia
14.7	Systemowe procedury analizy stochastycznej	brak wsparcia
14.8	Systemowe funkcje czasu symulacji	brak wsparcia
14.9	Systemowe funkcje do przekształceń zmiennych typu **real**	brak wsparcia
14.10	Funkcje podziałów prawdopodobieństwa	brak wsparcia

TABELA 18.12. Dyrektywy kompilatora

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
16.1	**`celldefine** oraz **`endcelldefine**	brak wsparcia
16.2	**`default_nettype**	wspierane
16.3	**`define** oraz **`undef**	wspierane
16.4	**`ifdef**, **`else** oraz **`endif**	wspierane
16.5	**`include**	wspierane

Rozdział standardu	Konstrukcja	Wsparcie w pakiecie Quartus
16.6	**`resetall**	wspierane
16.7	**`timescale**	ignorowane podczas syntezy
16.8	**`unconnected_drive** oraz **`nounconnected_drive**	wspierane

Podsumowanie

Podsumujmy niektóre właściwości języka Verilog, które sprawiają, że jest on coraz bardziej popularny wśród projektantów układów cyfrowych. Po pierwsze, jest to prosta składnia języka, w wielu przypadkach odpowiadająca językowi programowania C. Ponieważ większość inżynierów, projektantów i programistów bardzo dobrze zna język C, „oswojenie" z językiem Verilog nie stanowi dla nich najmniejszego problemu. Z drugiej strony, język Verilog udostępnia ogromne możliwości do opisu układów i systemów cyfrowych. Przy tym, opis ten może być wykorzystany zarówno do syntezy (automatycznej realizacji sprzętowej), jak i do modelowania (sprawdzenia różnych cech projektowanego układu). Zauważmy, że owa cecha jest rzadkością wśród innych języków opisu sprzętu.

Oprócz tego, język Verilog umożliwia opis projektu praktycznie na wszystkich wykorzystywanych obecnie w praktyce inżynierskiej poziomach: tranzystorów, bramek logicznych, transferu rejestrów (RTL), układów logicznych, behawioralnym (funkcjonalnym), w postaci struktur hierarchicznych.

Struktura języka Verilog zapewnia również jego łatwe rozszerzanie. Do tego celu wykorzystuje się mechanizmy definiowania prymitywów użytkownika UDP oraz język programowania interfejsu PLI. Za pomocą mechanizmu UDP użytkownik może określać własne prymitywy, niewystępujące w języku Verilog. Interfejs PLI służy do komunikacji Veriloga z językiem programowania C. Możliwe jest stworzenie w języku C własnych zadań systemowych i funkcji, a następnie połączenie ich z językiem Verilog za pomocą interfejsu PLI. W taki sposób umożliwia się rozwinięcie języka Verilog do poziomu odpowiadającego wymaganiom użytkownika bądź projektu.

Wygodna, dobrze opracowana koncepcja języka Verilog, potwierdza się jego stabilnością w przeciągu ostatnich dwudziestu lat (standard z roku 2005 wprowadził mało znaczące zmiany). Dlatego też można oczekiwać, że przez kolejne dziesięciolecia język Verilog pozostanie podstawowym językiem opisu sprzętu.

Bibliografia

[1] Z. Hajduk, *Wprowadzenie do języka Verilog*, BTC, Warszawa, 2009, 320 s.

[2] W. Wrona, *Język Verilog w projektowaniu układów cyfrowych*, Wyd. Pracownia Komputerowa Jacka Skalmierskiego, 2009, 276 s.

[3] IEEE Std 1364-1995, *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, The Institute of Electrical and Electronics Engineers, Inc. New York, NY, USA, 653 s.

[4] IEEE Std 1364-2001 (Revision of IEEE Std 1364-1995), *IEEE Standard Verilog Hardware Description Language, IEEE Computer Society*, The Institute of Electrical and Electronics Engineers, Inc. New York, NY, USA, 828 s.

[5] IEEE Std 1364.1, *IEEE Standard for Verilog Register Transfer Level Synthesis*, The Institute of Electrical and Electronics Engineers, Inc. New York, NY, USA, 100 s.

[6] D.E. Thomas, P.R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, New York, USA, 2002, 381 s.

[7] S. Palnitkar, *Verilog HDL: A guide to digital design and synthesis*, Mountain View, California, USA, Prentice Hall PTR, 2003, 496 s.

[8] J. LaMeres Brock, *Introduction to Logic Circuit and Logic Design with Verilog*, Springer Int. Publishing A, 2017, 459 s.

[9] B. Readler, *Verilog by Example. A concise Introduction for FPGA Design*, Full ARC Press, 2011, 124 s.

[10] J. Bhasker, *A Verilog HDL Primer*, Star Galaxy Pub., 2018.

[11] S. Sutherland, Verilog-2001: A Guide to the New Features of the Verilog Hardware, Sutherland HDL Pub., 2001.

[12] M.D. Ciletti, *Advanced digital design with the Verilog HDL*, Prentice Hall, New Jersey, USA, 2003, 985 s.

[13] S. Ramachandran, *Digital VLSI system design. A design manual for implementation of projects on FPGAs and ASICs using Verilog*, Springer, Dordrecht, The Netherlands, 2007, 709 s.

[14] S. Palnitkar, P. Saggurti, S.-H Kuang, *Finite state machine trace analysis program*, [w] Proc. of the IEEE Int. Conf. Verilog HDL Conference, 1994, s. 52-57.

[15] M. Arnold, A. Wallace, J. Cupal, J. Cowles, F. Engineer, *Towards a formal model of hardware synthesized from Verilog*, [w] Proc. of the IEEE Int. Conf. Verilog HDL Conference, 1996, s. 60-66.

[16] S.-T. Cheng, R. K. Brayton, *Synthesizing multi-phase HDL programs*, [w] Proc. of the IEEE Int. Conf. Verilog HDL Conference, 1996, s. 67-76.

[17] T.-H. Wang, T. Edsall T, *Practical FSM analysis for Verilog*, [w] Proc. of the IEEE Int. Conf. Verilog HDL Conference and VHDL International Users Forum, 1998, s. 52-58.

[18] F. Balarin, R. Passerone, *Specification, synthesis, and simulation of transactor processes*, "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems", 2007, t. 26, nr. 10, s. 1749-1762.

[19] Z. Zhen, Z. Hui, *The hardware interface design in SoC with Verilog language*, [w] Proc. of the IITA Int. Conf. Service Science, Management and Engineering (SSME'09), 2009, s. 30-33.

Spis tabel

Spis rysunków

Streszczenie

Valery Salauyou, Adam Klimowicz, Tomasz Grześ, Irena Bułatowa, *Język Verilog w projektowaniu systemów wbudowanych na układach FPGA*

Książka stanowi wprowadzenie do jednego z najbardziej rozpowszechnionych języków opisu sprzętu, jakim jest Verilog. Szczegółowo opisuje podstawy składni oraz konstrukcje języka Verilog z punktu widzenia ich praktycznego zastosowania. Każda przedstawiona konstrukcja wsparta została przykładem kodu oraz schematem jej realizacji na poziomie przesłań międzyrejestrowych RTL. Prezentowany materiał nie jest ograniczony do jednego konkretnego środowiska projektowania czy układów FPGA jednej wybranej firmy. Przedstawione zagadnienia mogą być wykorzystane do opracowywania projektów za pomocą dowolnego pakietu zautomatyzowanego projektowania systemów cyfrowych, ponieważ większość z nich umożliwia opis projektów w języku Verilog. Popularność języka Verilog spowodowana jest jego prostą składnią, zbliżoną do języka C, oraz bogatymi możliwościami opisu urządzeń i systemów cyfrowych, zarówno przy ich syntezie, jak i symulacji, począwszy od poziomu tranzystorów, a skończywszy na skomplikowanych strukturach hierarchicznych. Struktura języka Verilog umożliwia także jego łatwe rozszerzenie, w tym celu można wykorzystać mechanizmy definiowania prymitywów użytkownika UDP oraz interfejs języka programowania PLI.

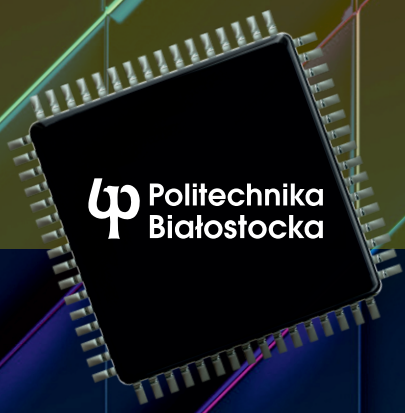
Książka jest przeznaczona dla studentów kierunków technicznych uczelni wyższych, może być pomocna przy realizacji przedmiotów, których efekty kształcenia przewidują nabycie umiejętności projektowania systemów cyfrowych z wykorzystaniem układów FPGA, a także przy wykonaniu zadań projektowych oraz prac dyplomowych. Może być przydatna także dla inżynierów, którzy zawodowo zajmują się projektowaniem układów i systemów cyfrowych, jako źródło szczegółowych informacji o języku Verilog.

Abstract

Valery Salauyou, Adam Klimowicz, Tomasz Grześ, Irena Bułatowa, *Verolog HDL for Embedded System Design on FPGA*

This book presents one of the most widely used hardware description languages, Verilog. It describes in detail the syntax and constructs of Verilog, from the point of view of their practical application. Each of the presented constructs is illustrated by a code example and its register-transfer level implementation. The presented material is not limited to a specific CAD software package or only one selected FPGA family, it can be used for digital systems design with any commercial CAD software package as most of them support Verilog. The popularity of the Verilog language is due to its simple syntax, similar to the C language, and the rich possibilities of specifying digital devices and systems, both in their synthesis and simulation, from the transistors level to complex hierarchical structures. Furthermore, Verilog language enables its easy extension by means of the user defined primitives UDP and the programming language interface PLI.

The book is addressed to students of technical faculties of universities, it can be helpful in learning subjects related to the design of digital systems with the use of FPGA circuits, as well as in the implementation of design tasks and diploma theses. It can also be useful for engineers who professionally design digital circuits and systems as a source of detailed information about the Verilog language.


Politechnika
Białostocka