POLITECHNIKA ZIELONOGÓRSKA
Wydział Elektryczny

Język Verilog w projektowaniu układów FPGA

Jacek Bieganowski
Grzegorz Wawrzyniak

Promotor:
dr inż. Marek Węgrzyn

Zielona Góra, czerwiec 2001

Spis treści

Spis rysunków

Spis tabel

Przykłady

1. Wprowadzenie

Ostatnie lata przyniosły szybki rozwój techniki cyfrowej. Pierwsze układy scalone produkowane w latach sześćdziesiątych zawierały kilkanaście tranzystorów, pod koniec lat osiemdziesiątych produkowano już układy posiadające ponad milion tranzystorów. Złożoność układów rośnie systematycznie i zgodnie z prawem Moore'a, co około 18 miesięcy następuje jej podwojenie.

Trend do ciągłej rozbudowy układów cyfrowych jest spowodowany dwoma podstawowymi czynnikami. Po pierwsze scalenie pozwala zmieścić w jednym układzie coraz więcej elementów urządzenia co w efekcie daje redukcje kosztów obudów i połączeń. Po drugie następuje zmniejszenie wpływu zjawisk pasożytniczych w poszczególnych elementach oraz między nimi. W wyniku czego możliwe jest zwiększenie szybkości działania oraz niezawodności układów.

Rozbudowa układów ma także swoje złe strony. Duże układy są trudniejsze w projektowaniu i testowaniu. Projektowanie wymaga większych nakładów pracy, rośnie też szansa pojawienia się błędów w projekcie, które są trudniejsze do wykrycia ponieważ testowanie wymaga sprawdzenia znacznie większej liczby możliwych kombinacji.

Narzędzia do komputerowego wspomagania projektowania (CAD) pozwalają w znacznym stopniu przyspieszyć proces projektowania oraz optymalizacji jakości układu. Dzięki programom tego typu możliwa jest automatyzacja niektórych etapów cyklu projektowego oraz symulacja i weryfikacja fragmentów lub całego układu ułatwiająca wykrywanie ewentualnych błędów już we wczesnych fazach projektowania.

Do modelowania układów najczęściej używa się schematów logicznych, diagramów stanów oraz języków opisu sprzętu takich jak: VHDL i Verilog. Języki opisu sprzętu (HDL) zyskują sobie coraz większą popularność, którą zawdzięczają przede wszystkim narzędziom do syntezy logicznej. Pojawienie się tych narzędzi umożliwiło automatyczną zamianę abstrakcyjnego opisu na listę połączeń w zadanej technologii.

1.1. Historia języków opisu sprzętu

Język Verilog HDL powstał na przełomie 1983/84 roku jako zastrzeżony produkt w firmie Gateway Design Automation. Sukces języka pozwolił na szybki rozwój firmy, która została przejęta w 1989 roku przez Cadence Design Systems. Od roku 1990 Verilog stał się językiem otwartym, nad którym opiekę sprawuje niezależna grupa Open Verilog International (OVI). Po powstaniu OVI szereg mniejszych firm rozpoczęło pracę nad własnymi symulatorami Veriloga, z których pierwsze pojawiły się na rynku w 1992 roku. W rezultacie rynek narzędzi związanych z Verilogiem zaczął się znacząco rozwijać i w roku 1994 szacowano go na 75 milionów dolarów.

W roku 1993 powołano grupę roboczą IEEE, która rozpoczęła pracę nad przygotowaniem standardu IEEE dla Veriloga. Prace nad standardem zakończono w 1995 roku. Powstanie standardu jeszcze bardziej umocniło pozycję Veriloga, a wartość sprzedaży samych już tylko symulatorów przekroczyła w 1998 roku 150 milionów dolarów.

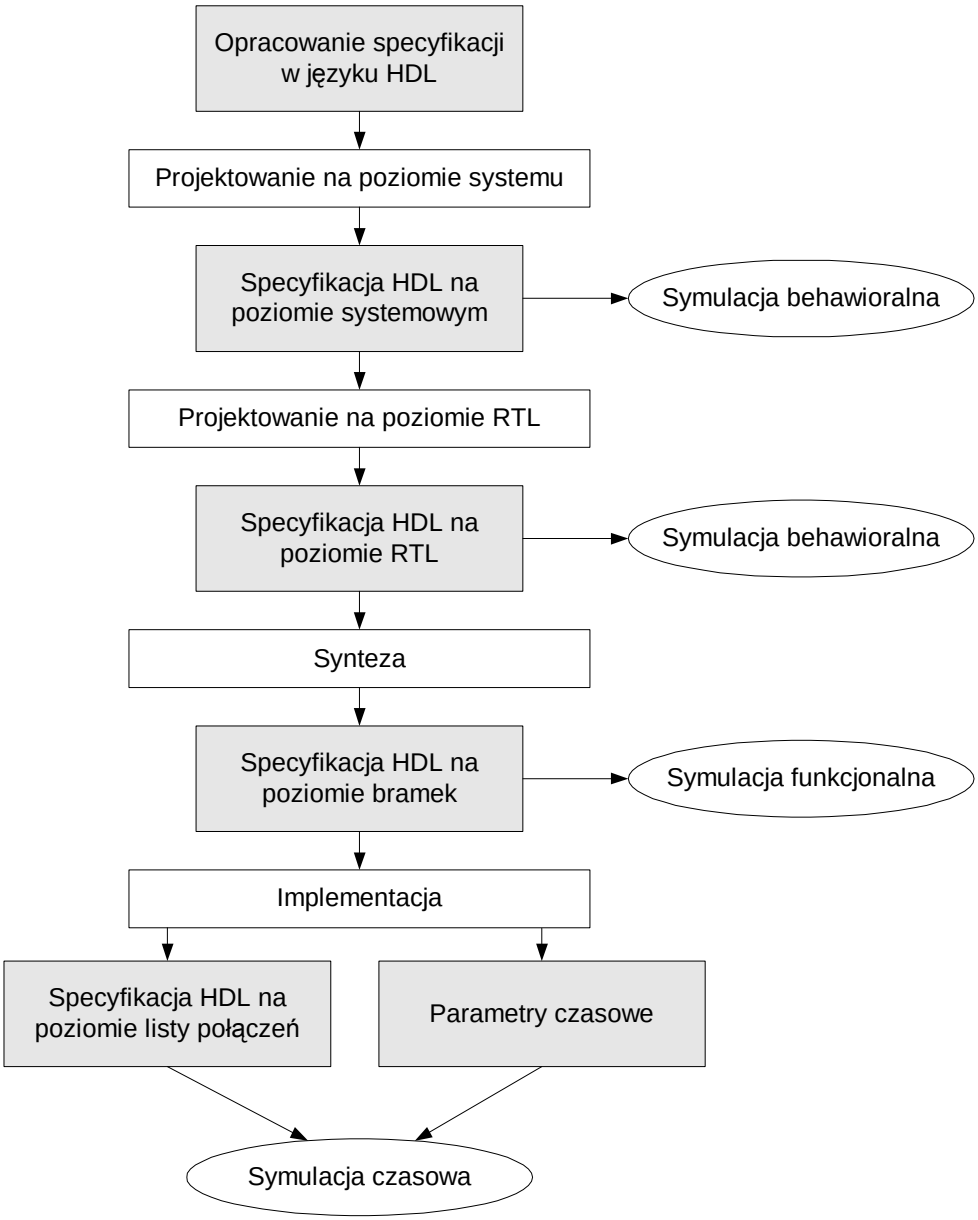
1.2. Cechy języków HDL

W porównaniu z tradycyjnym projektowaniem układów za pomocą schematów, języki opisu sprzętu oferują wiele dodatkowych ułatwień:

- w językach HDL układy można opisywać na różnych poziomach abstrakcji. Możliwe jest tworzenie opisu na poziomie RTL, który nie jest zależny od konkretnej technologii produkcji. Stosując narzędzia do syntezy można przekształcić taki opis na listę połączeń w zadanej technologii. Dzięki temu zmiana technologii nie wymaga projektowania układu od podstaw. Wystarczy ponownie wykonać proces syntezy zorientowany na nową technologię.
- modele w językach HDL można poddać weryfikacji funkcjonalnej już na początku cyklu projektowego. Dzięki temu już podczas tworzenia opisu na poziomie RTL można optymalizować układ oraz eliminować ewentualne błędy. W rezultacie prawdopodobieństwo wystąpienia błędów w dalszych etapach (np. podczas produkcji) znacznie się zmniejsza.
- projektowanie z użyciem języków opisu sprzętu jest podobne do programowania komputerów. Tekstowy zapis z komentarzami jest zwięzły i bardziej czytelny w porównaniu ze schematami na poziomie bramek.

1.3. Metodologia projektowania układów cyfrowych

Schemat blokowy przedstawiający proces projektowania układów cyfrowych z wykorzystaniem języków HDL pokazano na rysunku 1.



Rysunek 1. Proces projektowania układów cyfrowych

Pierwszym etapem projektowania jest opracowanie specyfikacji systemowej w języku HDL. Specyfikacja ta opisuje funkcjonalność układu na wysokim poziomie abstrakcji bez wnikania w szczegóły późniejszej implementacji. Poprawność modelu sprawdzana jest w symulatorach HDL. Do najbardziej popularnych programów tego typu należą: ModelSim firmy Model Technology oraz Active-HDL firmy Aldec.

Po opracowaniu i sprawdzeniu specyfikacji systemowej można przystąpić do przygotowania modelu na poziomie RTL. Przejście między modelem systemowym a modelem RTL jest wykonywane ręcznie i polega na dostosowaniu modelu systemowego do potrzeb stawianych przez narzędzia do syntezy.

Opracowanie opisu na poziomie RTL jest bardzo istotne w procesie projektowania. Na tym etapie możliwe jest przygotowanie niezależnego od technologii modelu IP przeznaczonego do wielokrotnego wykorzystania. W trakcie projektowania należy zdawać sobie sprawę w jaki sposób zostaną zinterpretowane poszczególne konstrukcje języka podczas syntezy, ponieważ w znacznym stopniu rzutuje to na jakość końcowego układu.

Kolejnym etapem procesu projektowania jest synteza modelu na poziomie RTL. Do tego celu stosuje się narzędzia do syntezy specyfikacji HDL, m.in: Simplify firmy Simplicity, Leonardo firmy Exemplar, FPGAExpres firmy Synopsys.

Wynikiem syntezy jest strukturalny opis układu składający się z listy połączeń podstawowych elementów logicznych dostępnych w zadanej technologii. Następnie wykonuje się symulację funkcjonalną aby sprawdzić zgodność otrzymanej struktury logicznej z modelem behawioralnym. Do symulacji wykorzystuje się odpowiednie biblioteki elementów dostarczane z symulatorem.

Ostatnim etapem procesu jest implementacja modelu w zadanej technologii oraz jego symulacja. Do implementacji wykorzystuje się narzędzia oferowane przez producentów układów programowalnych, m.in: firm Altera, Xilinx, Cypress. Narzędzia te pozwalają optymalizować powierzchnię oraz prędkość pracy układu. Na tym etapie dokonywane jest rozmieszczanie elementów w strukturze logicznej oraz definiowanie połączeń między nimi (ang. *Placement & Routing*). Następnie w oparciu charakterystyczne parametry układu, wykonuje się symulację czasową, w której uwzględniane są czasy propagacji poszczególnych elementów układu oraz połączeń między nimi.

2. Model układu cyfrowego w Verilogu

Bez względu na funkcjonalność układu cyfrowego musi on komunikować się z otoczeniem, pobierać z niego informacje i wyprowadzać na zewnątrz wyniki. Część układu odpowiedzialna za komunikację nazywana jest *interfejsem*. Bez interfejsu układ byłby równie bezużyteczny jak najszybszy procesor pozbawiony wyprowadzeń. Drugim ważnym elementem, obok interfejsu, jest wnętrze układu, nazywane często *ciałem* układu (ang. *body*), które odpowiada za przetwarzanie informacji pobranych z otoczenia.

Verilog pozwala specyfikować zarówno interfejs jak i ciało układu. Elementem odpowiedzialnym za reprezentację układu w Verilogu jest *moduł*. Może on opisywać układ o dowolnej złożoności począwszy od podstawowych bramek logicznych a skończywszy mikroprocesorach i innych skomplikowanych układach.

Najczęściej układy, szczególnie te złożone, dzieli się na mniejsze części, z których każdą można opisać w osobnym module. Dostarcza się w ten sposób prostych bloków funkcjonalnych, które później mogą być wykorzystane w wielu miejscach projektu.

2.1. Budowa modułu

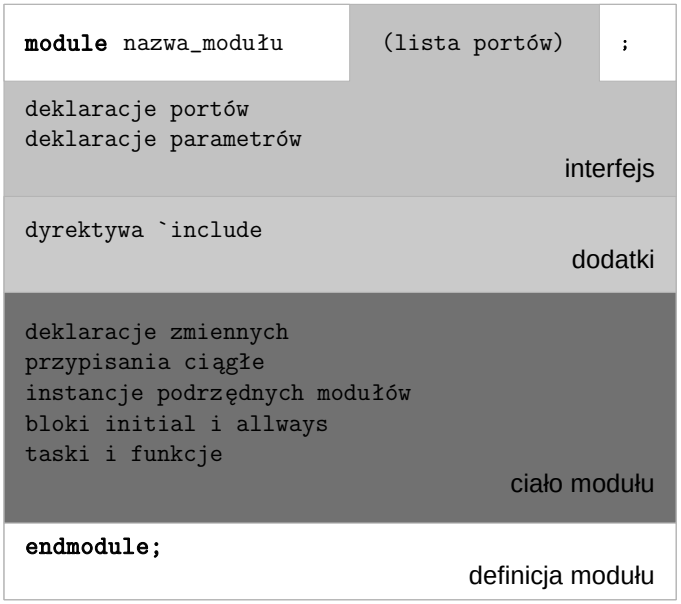
Deklarację modułu rozpoczyna się słowem kluczowym `module`, po którym należy podać nazwę modułu, a kończy się słowem `endmodule` (pisane razem). Wewnątrz moduł można podzielić na trzy części: *interfejs*, *ciało* oraz opcjonalne *dodatki*.

Interfejs znajduje się zawsze na początku modułu i musi wystąpić przed ciałem modułu. Składa się on z trzech elementów: *listy portów*, *deklaracji portów* oraz *deklaracji parametrów*. Blokową budowę modułu przedstawia rysunek 2.

Ciało modułu opisuje jego działanie. Mogą tu wystąpić: *deklaracje zmiennych*, *instrukcje równoległe*, *instancje podrzędnych modułów*, *bloki behawioralne* oraz *zadania* i *funkcje*. Wyżej wymienione elementy mogą występować w dowolnej kolejności lecz należy pamiętać, że każdy element może być użyty dopiero po deklaracji.

2.1.1. Nazwy modułów

Nazwa modułu to inaczej mówiąc jego *identyfikator*. Podobnie jak wszystkie identyfikatory w Verilogu podlega on następującym regułom:



Rysunek 2. Budowa modułu

- może składać się tylko z liter, cyfr, znaków podkreślenia(_) lub dolara ($),
- musi rozpoczynać się literą lub znakiem podkreślenia,
- nie może zawierać znaków spacji,
- rozróżniane są duże i małe litery,
- nie może być słowem kluczowym Veriloga.

Nazwa modułu jest obowiązkowa i musi być niepowtarzalna. Oznacza to, że w jednym projekcie nie może być dwóch modułów posiadających tą samą nazwę. Tworząc nazwy należy pamiętać, że pełnią one rolę dokumentacyjną, dlatego powinny być czytelne i dobrze określać działanie modułu. Czytelność nazw można poprawić stosując duże litery oraz znaki podkreślenia, na przykład: `SumatorPelny`, `Dekoder_7Seg`.

2.1.2. Lista i deklaracja portów

Lista portów jest podawana w nawiasach okrągłych zaraz po nazwie modułu. Występują na niej jedynie nazwy wyprowadzeń, które rozdziela się od siebie przecinkami. Dokładniejsze informacje o kierunku i rozmiarze portu podawane są w deklaracji portów. Ze względu na kierunek porty można podzielić na trzy typy:

- **wejściowe** – deklarowane słowem kluczowym `input`, które służą tylko do pobierania informacji z otoczenia;
- **wyjściowe** – deklarowane słowem kluczowym `output`, przeznaczone tylko do wyprowadzania danych na zewnątrz;
- **dwukierunkowe** – deklarowane słowem kluczowym `inout`, pozwalające pobierać i wysyłać informacje;

Deklaracje portów dla modułu czterobitowego sumatora pokazano w przykładzie 1. Porty `a`, `b` oraz `sum` zadeklarowano jako czterobitowe wektory, natomiast pozostałe porty są jednobitowe.

Przykład 1. Interfejs modułu sumatora czterobitowego

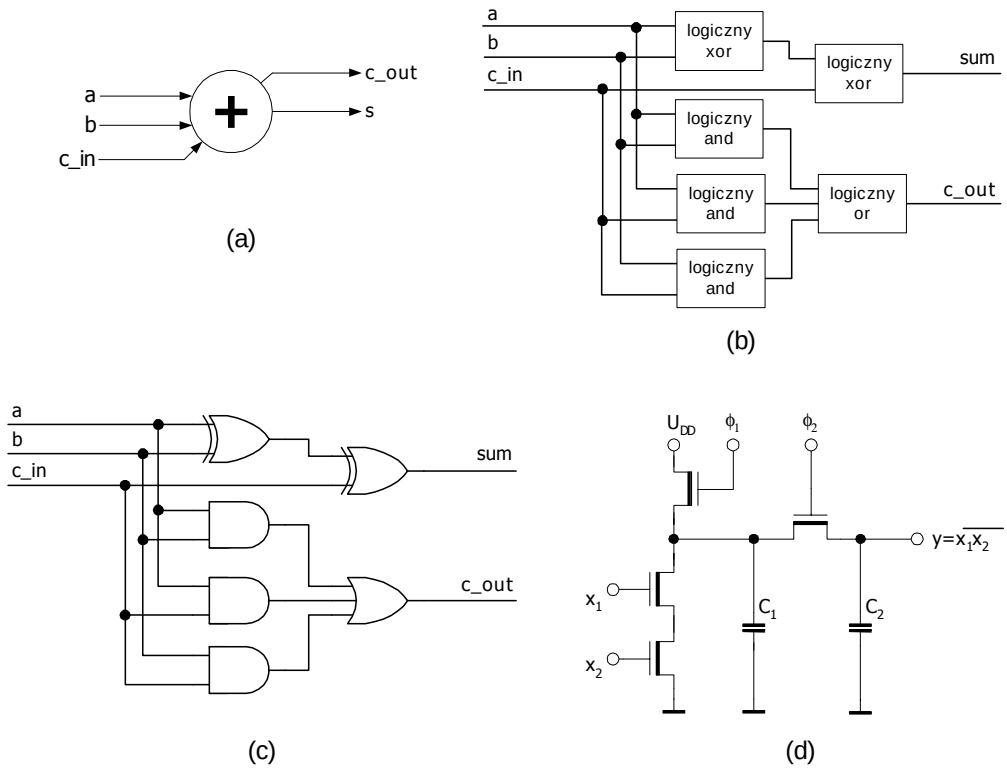
```
module adder_4 (a, b, c_in, sum, c_out);
  input [3:0] a;      // składnik sumy (4 bity)
  input [3:0] b;      // składnik sumy (4 bity)
  input c_in;         // przeniesienie wejściowe (1 bit)
  output [3:0] sum;   // suma (4 bity)
  output c_out;       // przeniesienie wyjściowe (1 bit)
  ...
endmodule
```

2.2. Poziomy abstrakcji modułu

W Verilogu interfejs specyfikuje się zawsze tak samo bez względu na rodzaj i złożoność tworzonego modelu, natomiast ciało modułu można opisać na kilka sposobów. W zależności od potrzeb projektant może wybrać jeden z czterech dostępnych poziomów abstrakcji.

- Poziom behawioralny lub algorytmiczny (ang. *behavioral level*). Jest to najwyższy poziom abstrakcji dostępny w Verilogu. Opisywanie układu na tym poziomie nie jest zależne od szczegółów technologicznych i przypomina programowanie w języku wysokiego poziomu.
- Poziom przepływu danych (ang. *data flow level*). Na tym poziomie układ przedstawia się zwracając szczególną uwagę na powiązania między poszczególnymi elementami, takimi jak sumatory i rejestry oraz na sposób przetwarzania danych w całym układzie.
- Poziom bramek (ang. *gate level*). Moduł jest przedstawiany za pomocą podstawowych bramek oraz połączeń między nimi. Projektowanie na tym poziomie przypomina tworzenie projektu w oparciu o schemat logiczny.
- Poziom przełączników (ang. *switch level*). Jest to najniższy poziom abstrakcji dostępny w Verilogu. Moduł na tym poziomie jest opisywany za pomocą tranzystorów oraz połączeń między nimi. Projektowanie za pomocą przełączników wymaga dużej znajomości szczegółów technicznych docelowej technologii.

W projektowaniu układów cyfrowych często mówi się jeszcze o poziomie RTL (ang. *Register Transfer Level, RTL*), który jest połączeniem poziomu behawioralnego i przepływu danych. Konstrukcje poziomu RTL są akceptowane przez narzędzia do syntezy.



Rysunek 3. Poziomy abstrakcji w Verilogu: a) poziom behawioralny; b) poziom przepływu danych; c) poziom bramek; d) poziom przełączników

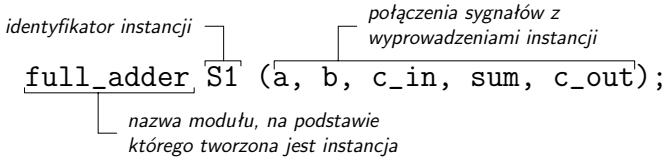
Verilog pozwala na łączenie ze sobą wszystkich dostępnych poziomów abstrakcji w jednym projekcie. Bez względu na poziom abstrakcji modułu przez otoczenie jest on zawsze widziany tak samo. Tak więc można zmieniać wnętrze modułu nie wpływając na resztę projektu.

2.3. Instancje

W Verilogu moduł można przyrównać do szablonu, na podstawie którego tworzy się niepowtarzalne obiekty nazywane instancjami. Każde odwołanie do modułu powoduje utworzenie nowej instancji mającej swoją własną nazwę, interfejs oraz zmienne. Nazwa instancji służy do identyfikacji i dlatego musi być niepowtarzalna.

Verilog nie dopuszcza zagnieżdżania definicji modułów. Oznacza to, że pomiędzy słowami kluczowymi `module` i `endmodule` nie można definiować innych modułów lecz w zamian należy tworzyć ich instancje.

Nie należy mylić definicji modułu z jego instancją. Definicja jest jedynie opisem wyprowadzeń oraz działania układu, natomiast instancja to konkretny obiekt/egzemplarz realizujący ten opis. Zatem aby w projekcie można było użyć jakiś element najpierw należy go opisać (przygotować definicję modułu), a następnie można się nim posługiwać czyli tworzyć jego instancje. Przykładową instancję przedstawia rysunek 4.



Rysunek 4. Instancja modułu sumatora pełnego

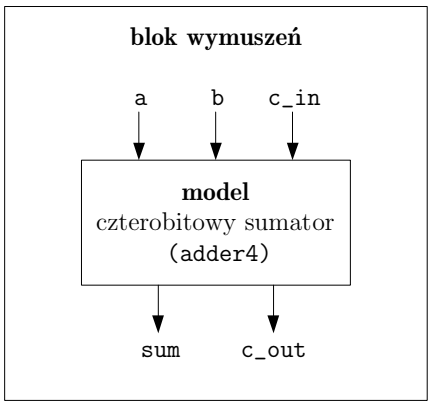
Przykład 2. Czterobitowy sumator zbudowany z sumatorów pełnych

```
module adder4 (a, b, c_in, sum, c_out);
  input [3:0] a;     // składnik sumy (4 bity)
  input [3:0] b;     // składnik sumy (4 bity)
  input c_in;        // przeniesienie wejściowe (1 bit)
  output [3:0] sum;  // suma (4 bity)
  output c_out;      // przeniesienie wyjściowe (1 bit)

  wire c1, c2, c3;   // sygnały pomocnicze dla przeniesień

  // cztery instancje sumatora jednobitowego
  full_adder S1 (a[0], b[0], c_in, sum[0], c1);
  full_adder S2 (a[1], b[1], c1, sum[1], c2);
  full_adder S3 (a[2], b[2], c2, sum[2], c3);
  full_adder S4 (a[3], b[3], c3, sum[3], c_out);

endmodule

module full_adder (a, b, c_in, sum, c_out);
  input a, b;        // składniki sumy (1 bit)
  input c_in;        // przeniesienie wejściowe (1 bit)
  output sum;        // suma (1 bit)
  output c_out;      // przeniesienie wyjściowe (1 bit)

  // wzór na sumę: sum = a xor b xor c_in
  assign sum = a ^ b ^ c_in;

  // wzór na przeniesienie:
  // c_out = (a and b) or (a and c_in) or (b and c_in)
  assign c_out = (a & b) | (a & c_in) | (b & c_in);

endmodule
```

3. Symulacja

Po przygotowaniu projektu należy go przetestować. Jest to konieczne aby można było stwierdzić czy układ działa zgodnie z oczekiwaniami projektanta. Funkcjonalność układu może być sprawdzona przez zadanie na wejścia sygnałów testowych i sprawdzenie otrzymanych wyników na wyjściu. Realizuje się to tworząc dodatkowy blok z wektorami testowymi, który nazywany jest blokiem wymuszeń (ang. *sitmulus block*) lub często z angielskiego „test bench'em".

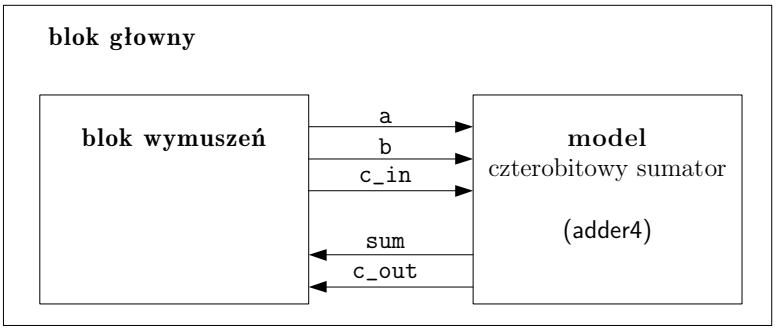
Blok wymuszeń może być opisany w Verilogu bez potrzeby stosowania innych języków. Testowanie można zrealizować na dwa sposoby. Pierwszy sposób polega na przygotowaniu bloku wymuszeń zawierającego instancje bloku testowanego. W ten sposób blok wymuszeń staje się głównym modułem w projekcie. Przykład takiego rozwiązania pokazano na rysunku 5. Blok wymuszeń generuje sygnały `a`, `b`, `c_in` i jednocześnie odczytuje i wyświetla wyniki z wyjść `sum` oraz `c_out`.



Rysunek 5. Blok wymuszeń z instancją bloku projektowego

W drugim rozwiązaniu (rysunek 6) tworzy się dodatkowy moduł zawierający instancje bloku projektowego oraz bloku wymuszeń. Obie instancje wymieniają między sobą sygnały, poprzez swoje interfejsy. Blok wymuszeń generuje sygnały `a`, `b`, `c_in` i odbiera z bloku projektowego sygnały `sum` i `c_out`.

Oba sposoby podawania wzbudzeń mogą być użyte w bardzo efektywny sposób. W metodzie z rysunku 6 możliwe jest również automatyczne zweryfikowanie otrzymanych wyników dzięki sprzężeniu między blokiem projektowym a blokiem wymuszeń.



Rysunek 6. Blok główny zawierający blok wymuszeń oraz model

Aby zilustrować pojęcia wprowadzone w tym rozdziale przygotowano kompletną symulację czterobitowego sumatora. Do symulacji został użyty model sumatora z przykładu 2, dla którego przygotowano blok generujący impulsy testowe. Na tym etapie nie jest potrzebna szczegółowa znajomość wszystkich instrukcji Veriloga, zamiast tego ważniejsze jest zrozumienie współdziałania poszczególnych elementów w projekcie.

Przykład 3. Blok wymuszeń dla sumatora czterobitowego

```
module blok_wymuszen;
 reg [3:0] a, b;  // deklaracja niezbędnych sygnałów
 reg c_in;
 wire[3:0] sum;
 wire c_out;

 // instancja testowanego sumatora w bloku wymuszeń
 adder4 TesowanySumator (a, b, c_in, sum, c_out);

 // wymuszenia na wejściach sumatora
 initial begin
       a = 0; b = 0; c_in = 0;
   #10 a = 0; b = 0; c_in = 1;
   #10 a = 1; b = 1; c_in = 0;
   #10 a = 5; b = 3; c_in = 0;
   #10 a = 7; b = 8; c_in = 0;
   #10 a = 8; b = 9; c_in = 0;
   #10 a = 10; b = 10; c_in = 0;
   #10 a = 15; b = 15; c_in = 1;
 end

 initial begin
     // wyświetlenie wyników symulacji
     $display ("czas | a + b + c_in = sum i c_out");
     $monitor ($time, " | %d + %d + %d = %d i c_out = %b",
               a, b, c_in, sum, c_out);
 end
endmodule
```

Wyniki symulacji przedstawia przykład 4. Pierwsza kolumna zawiera czas symulacji, następne trzy kolumny odpowiadają zmiennym wejściowym sumatora (`a`, `b` i `c_in`), a ostatnie dwie zmiennym wyjściowym `sum` i `c_out`.

Przykład 4. Wyniki symulacji sumatora czterobitowego

```
czas |  a +  b + c_in = sum i c_out
   0 |  0 +  0 + 0 =  0 i c_out = 0
  10 |  0 +  0 + 1 =  1 i c_out = 0
  20 |  1 +  1 + 0 =  2 i c_out = 0
  30 |  5 +  3 + 0 =  8 i c_out = 0
  40 |  7 +  8 + 0 = 15 i c_out = 0
  50 |  8 +  9 + 0 =  1 i c_out = 1
  60 | 10 + 10 + 0 =  4 i c_out = 1
  70 | 15 + 15 + 1 = 15 i c_out = 1
```

4. Podstawowe elementy języka

4.1. Typy danych

W Verilogu typy danych reprezentują elementy przechowujące dane oraz elementy transmisyjne. Można je podzielić na dwie grupy: typy rejestrowe (ang. *register data type*) oraz typy sieciowe (ang. *net data type*). Grupy te odpowiadają innym elementom sprzętowym i różnią się między sobą sposobem przypisywania i przechowywania wartości.

4.1.1. Wartości logiczne

Sygnały elektryczne w układach cyfrowych przyjmują tylko dwie wartości odpowiadające cyfrom 0 i 1. W językach HDL sygnały opisuje się za pomocą skończonej liczby stanów. W Verilogu jest ich tylko 4 ale są też opinie, że do opisu sygnałów powinno stosować się 7, 9, 49 lub nawet 126 stanów. Zbiór wartości dostępnych w Verilogu przedstawia tabela 1.

Tabela 1. Wartości logiczne w Verilogu

Wartość	**Znaczenie**
0	zero logiczne, warunek fałszywy
1	jedynka logiczna, warunek prawdziwy
X	wartość nieznana
Z	wysoka impedancja, stan zmienny

4.1.2. Sieci

Sieci (ang. *net*) odpowiadają fizycznym połączeniom elementów, na przykład za pomocą miedzianych linii. Sieć jest sterowana z nadajnika (źródła sygnału) i przekazuje sygnał do wszystkich odbiorników. Jeżeli żaden nadajnik nie steruje siecią, to przyjmuje ona stan wysokiej impedancji (`z`) z wyjątkiem sieci `trireg`, której domyślną wartością jest `x`.

W Verilogu można wyróżnić następujące sieci:

- `wire/tri` – są to najczęściej wykorzystywane sieci. Sposób ich deklaracji i działania jest identyczny, a dwie nazwy wprowadzono jedynie dla potrzeb dokumentacji. Sieci typu `wire` są stosowane, gdy połączenie jest sterowane tylko z jednego źródła, natomiast sieć typu `tri` jest używana gdy linia jest sterowana przez wiele zródeł sygnału (dostęp do linii jest wtedy realizowany przez bufory trójstanowe, od których pochodzi nazwa sieci).
- `wand/triand` i `wor/trior` – w przypadku jednego nadajnika sieci te zachowują się dokładnie tak samo jak `wire` i `tri`, natomiast przy wielu zródłach sygnału realizują one dodatkowo operacje logiczne. Stan linii `wand/triand` jest wyznaczany przez wykonanie operacji AND na sygnałach ze wszystkich źródeł (ang. *wired and*), natomiast dla sieci `wor/trior` jest to operacja OR (ang. *wired or*).
- `supply0`, `supply1`, `tri0`, `tri1`, `trireg` – sieci te są stosowane przy opisywaniu układu za pomocą tranzystorów oraz innych elementów związanych bezpośrednio z technologią wykonywania układów scalonych. Projektowanie układów cyfrowych z wykorzystaniem tych elementów nie jest omawiane w tej pracy. Więcej informacji można znaleźć w [1] [2].

Przykład 5. Deklaracja i użycie sieci

```
wire a;     // deklaracja sieci typu wire
wand x, y;  // deklaracja dwóch sieci typu wand

assign a = 1; // przypisanie sieci wartości 1
```

4.1.3. Rejestry

Rejestry (ang. *register*) w przeciwieństwie do sieci są zdolne do przechowywania wartości nawet po odłączeniu źródła sygnału. Raz ustalony stan rejestru jest w nim zapamiętany aż do wprowadzenia nowej wartości. Jednak nie należy mylić rejestrów w Verilogu z rejestrami sprzętowymi, na przykład przerzutnikami. Rejestr w języku Verilog jest po prostu elementem przechowującym wartość, podobnie jak zmienne w językach programowania. W Verilogu wyróżnia się rejestry typu: `reg`, `integer`, `real`, `time`, `realtime`.

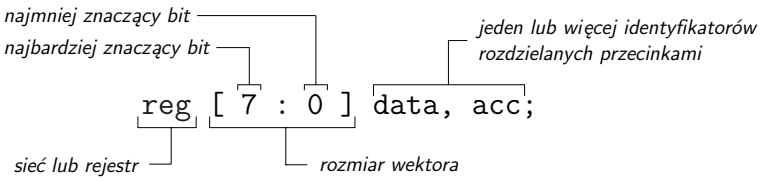
Rejestry typu `reg` są najbardziej uniwersalne i mogą być stosowane nie tylko do modelowania układów, ale również do innych celów takich jak zliczanie impulsów czy przechowywanie czasu symulacji. Jednak dla wygody oraz poprawienia czytelności przyjmuje się, że do zliczania lepiej jest użyć rejestrów typu `integer`, a do operacji na na wartościach czasowych typu `time`.

Przykład 6. Deklaracja i użycie rejestru typu `reg`

```
reg reset;  // deklaracja

initial begin
  reset = 0;      // inicjalizacja rejestru
  #50 reset = 1;  // po 50 jednostkach czasu zmień wartość na 1
end
```

4.1.4. Wektory

Zmienne wektorowe otrzymuje się podając po typie zmiennej jej rozmiar za pomocą dwóch liczb dziesiętnych rozdzielonych dwukropkiem np: [15 : 0]. Liczba po lewej stronie jest zawsze najbardziej znaczącym bitem wektora (ang. *Most Significant Bit, MSB*), natomiast liczba po prawej najmniej znaczącym bitem (ang. *Least Significant Bit, LSB*). Do zapisu rozmiaru można stosować liczby dodatnie, ujemne oraz zero.



Rysunek 7. Deklarowanie wektorów

Numeracja bitów w deklaracji wektora może być malejąca (`MSB>LSB`) lub rosnąca (`MSB<LSB`). Ma to później znaczenie przy odwołaniach do części wektora (ang. *part-select*). Na przykład, jeżeli wektor został zadeklarowany z malejącą numeracją (`reg [7:0] w`), to późniejsze odwołania również muszą być zgodne z tą numeracją (`w[6:4]`, a nie `w[4:6]`).

Przykład 7. Deklaracja i odwołania do wektorów

```
wire [7:0] bus;   // 8-bitowa sieć
reg [0:31] cx;    // 32-bitowy rejestr
reg [5:-2] ah;    // 8-bitowy rejestr
wire [-3:0] busB; // 4-bitowa sieć

bus[5];   // piąty bit wektora bus
cx[0:7];  // 8 najstarszych bitów wektora cx
cx[7:0];  // ŹLE - kierunek portu jest inny niż w deklaracji
busB[0];  // najmłodszy bit wektora busB
```

4.1.5. Rejestry integer, real, time i realtmie

Jak już wcześniej wspomniano rejestry typu `integer` są przeznaczone przede wszystkim do operacji liczbowych nie związanych bezpośrednio ze sprzętem. Ich domyślną

wielkością jest rozmiar słowa komputera, w którym nastąpiła implementacja, ale nie mniej niż 32 bity.

Między rejestrami typu `reg` i `integer` występują różnice w sposobie przechowywania liczb ujemnych. Do rejestru `reg` można zapisać liczby ujemne, jednak jeśli rejestr zostanie użyty w wyrażeniu, to jego wartość zostanie potraktowana jako liczba bez znaku. Na przykład, liczba (-1) zapisana do 4-bitowego rejestru w wyrażeniu będzie traktowana jako liczba 15. Omawiany problem nie występuje w przypadku typu `integer`, gdzie liczby są przechowywane z uwzględnieniem znaku. Więcej informacji o obliczeniach znajduje się w rozdziale 7.3.1, natomiast o zapisie liczb w rozdziale 4.2.

Przykład 8. Deklaracja i użycie rejestru typu `integer`

```
integer a; //deklaracja

initial
  a = -2;  //przypisanie do zmiennej a liczby dziesiętnej -2
```

Rejestr typu `time` jest 64 bitowy i służy do przechowywania i manipulowania wartościami czasowymi. Typ ten jest stosowany najczęściej w powiązaniu z dyrektywą kompilatora `$time`, która pozwala pobrać aktualny czas symulacji.

Przykład 9. Deklaracja i użycie rejestru typu `time`

```
time punkt_pomiarowy;       //deklaracja

initial
  punkt_pomiarowy = $time;  // pobranie aktualnego czasu symulacji
```

Typ rzeczywisty `real` jest przeznaczony do przechowywania i wykonywania obliczeń na liczbach rzeczywistych. Rejestry tego typu nie mają zakresu i domyślnie są inicjowane wartością 0. Gdy do zmiennej całkowitej jest przypisywana liczba rzeczywista, to jest ona zaokrąglana do najbliższej wartości całkowitej.

Przykład 10. Deklaracja i użycie rejestru typu `real`

```
real a, b;   // deklaracja dwóch zmiennych typu real
integer c;

initial begin
  a = 8e3;   // przypisanie wartości zapisanej w notacji naukowej
  b = 1.15;  // przypisanie wartości w notacji dziesiętnej
  c = b;     // do zmiennej c zostanie przypisana zaokrąglona wartość
             // rejestru b czyli 1
end
```

Rejestr typu `realtime` pozwala przechowywać czas symulacji zapisany w formacie liczby rzeczywistej. Najczęściej używa się go w powiązaniu z dyrektywą `$realtime`.

Rejestry `real` i `realtime` są identyczne i mogą być stosowane zamiennie. Dwie nazwy wprowadzono jedynie w celach dokumentacyjnych.

Przykład 11. Deklaracja i użycie rejestru typu `realtime`

```
realtime punkt_pomiarowy;        //deklaracja

initial
  punkt_pomiarowy = $realtime;   // pobranie aktualnego czasu symulacji
                                 // jako liczby rzeczywistej
```

4.2. Liczby

Wykonując obliczenia posługujemy się najczęściej liczbami dziesiętnymi, natomiast komputery i inne urządzenia cyfrowe wykorzystują w tym celu system dwójkowy. Verilog wspiera różne systemy liczbowe, tak by móc łatwo zapisywać liczby bez potrzeby ich ręcznej konwersji. Informacje o tym jak ma być interpretowana liczba przekazuje się za pomocą odpowiedniego przedrostka. Dla liczb dwójkowych, ósemkowych i szesnastkowych przedrostek ten jest obowiązkowy, a w przypadku liczb dziesiętnych można go pominąć.

Tabela 2. Systemy liczbowe

Przedrostek	Cyfry używane do zapisu	System liczbowy
`'d` lub `'D`	0..9	dziesiętny (domyślnie)
`'b` lub `'B`	0, 1	binarny
`'h` lub `'H`	0..9, a..f, A..F,	szesnastkowy
`'o` lub `'O`	0..7	ósemkowy

W przypadku układów cyfrowych ważny jest również rozmiar liczby czyli inaczej mówiąc liczba bitów używana do jej zapisu. Rozmiar może być podany jawnie - liczby z ustalonym rozmiarem (ang. *sized number*) lub pominięty - liczby bez rozmiaru (ang. *unsized number*). Jeśli rozmiar liczby nie jest podany, to domyślnie zostanie przyjęte 32-bity. Może też zdarzyć się tak, że liczba da się zapisać na mniejszej liczbie bitów niż to jawnie podano, wtedy zostanie ona rozszerzona z lewej strony zerami.

Przykład 12. Liczby z ustalonym rozmiarem

```
4'b1001   // 4-bitowa liczba dwójkowa
5'D3      // 5-bitowa liczba dziesiętna
7'hf      // 7-bitowa liczba szesnastkowa
```

Liczby ujemne tworzy się wstawiając znak minus (-) przed liczbą łącznie z jej rozmiarem i przedrostkiem podstawy systemu liczbowego.

Przykład 13. Liczby bez ustalonego rozmiaru

```
659       // liczba dziesiętna
'h87FF    // liczba szesnastkowa
'o7460    // liczba ósemkowa
```

Przykład 14. Liczby ujemne

```
-8'd6     // 8-bitowa ujemna liczba dziesiętna
-(8'd6)   // równoważny zapis
```

Długie liczby można podzielić na klika mniejszych fragmentów stosując znak (_). Taki zapis pozwala uniknąć pomyłek i poprawić czytelność szczególnie w przypadku liczb dwójkowych.

Przykład 15. Zastosowane znaku podkreślenia w zapisie liczb

```
27_195_000
16'b0011_0101_0001_1111
32'h12ab_f001
```

4.3. Łańcuchy

Ciąg znaków (ang. *string*) może być zapamiętywany w rejestrze. Każdy znak w łańcuchu jest 8-bitowy. Należy pamiętać o tym, że zmienna rejestrowa musi mieć wystarczająco duży rozmiar, aby zachować wszystkie znaki. W przypadku, gdy wielkość rejestru jest mniejsza niż rozmiar łańcucha, znaki z lewej strony ciągu zostają obcięte. Natomiast jeżeli rozmiar rejestru przekracza rozmiar łańcucha to, bity z lewej strony są wypełniane zerami. Najlepiej jest więc deklarować rejestry trochę większe niż rozmiar łańcucha aby pomieścić wszystkie znaki.

Łańcuchy umieszcza się w cudzysłowie. Należy pamiętać o tym, że muszą być one zapisane w jednej linii.

Przykład 16. Deklaracja zmiennej rejestrowej przechowującej ciąg znaków

```
reg [20*8:1] l;   // deklaracja rejestru 20 bajtowego

initial begin
  l = "Ala ma kota";  // zapamiętanie łańcucha w rejestrze
end
```

Niektóre znaki specjalne nie mogą być zapisane bezpośrednio w łańcuchach dlatego koduje się je za pomocą dodatkowych znaków nazywanych znakami ucieczki (ang. *escape character*).

Tabela 3. Znaki specjalne w łańcuchach

Znak ucieczki	Wyświetlany znak
`\n`	znak nowej linii
`\t`	tabulacja
`\\`	`\`
`\"`	`"`
`\ddd`	znak zapisany jako liczba ósemkowa

Przykład 17. Łańcuch ze znakami specjalnymi

```
l = "␣\"to jest cytat\"␣";
sciezka = "c:\\moje␣projekty\\sumator";
```

4.4. Tablice

W Verilogu można definiować tylko jednowymiarowe tablice elementów typu `reg`, `integer` i `time`. Niedozwolone jest tworzenie tablic dla sieci oraz typu `real`.

Tablice deklaruje się podobnie jak wektory z tym, że najpierw podaje się jej nazwę, a potem rozmiar w nawiasach kwadratowych.

Przykład 18. Tablice

```
integer liczby [1:10]; // 10-elementowa tablica liczb typu integer
reg bity [1:50];       // 50-elemenotwa tablica jednobitowych rejestrów

time punkty_pomiarowe [1:20];  // 20-elem. tablica zmiennych typu time
reg [15:0] rejestry [7:0];     // tablica ośmiu rejestrów 16-bitowych

liczby[3];             // trzeci element tablicy liczby
punkty_pomiarowe[1];   // pierwszy element tablicy punkty_pomiarowe
rejestry[5];           // piąty element tablicy rejestrów 16-bitowych
```

4.5. Parametry

Parametry (ang. *parameters*) służą do deklarowania stałych w modułach. Wartość parametru musi być znana podczas kompilacji i w przeciwieństwie do sieci i rejestrów nie może ulec zmianie w czasie trwania symulacji. Parametry definiuje się słowem kluczowym `parameter`. Dla poszczególnych instancji modułu można zmienić wartość parametru za pomocą słowa kluczowego `defparam`. Przykład 19 przedstawia sparametryzowany model sumatora. Parametr `size` określa rozmiar sumatora i wpływa na szerokość portów wejściowych i wyjściowych.

Przykład 19. Zastosowanie parametrów

```
module full_adder (a, b, c_in, sum, c_out);
  parameter size = 4;      // deklaracja parametru
  input [size-1:0] a, b;   // rozmiar zmiennych zależy od parametru size
  input c_in;
  output [size-1:0] sum;
  output c_out;

    assign {c_out, sum} = a + b + c_in;

endmodule

module top;
  wire [7:0] a, b, sum;
  wire c_in, c_out;

    defparam S1.size = 8;  // zmiana wartości parametru dla instancji S1
    full_adder S1 (a, b, c_in, sum, c_out);

endmodule
```

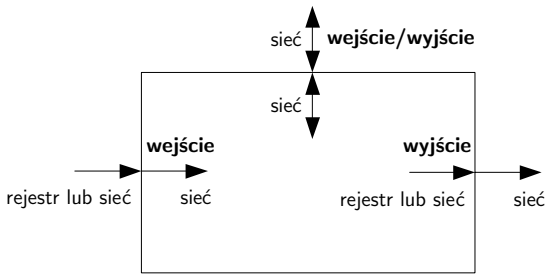
5. Porty i nazwy hierarchiczne

5.1. Reguły łączenia portów

Port można podzielić na dwie części: wewnętrzną i zewnętrzną. Podczas łączenia portów z sygnałami zewnętrznymi stosuje się następujące reguły:

- Porty wejściowe wewnątrz modułu mogą być tylko typu sieciowego. Na zewnątrz, do portu wejściowego, można podłączyć zmienne typu sieciowego i rejestrowego.
- Porty wyjściowe wewnątrz modułu mogą być typu sieciowego lub rejestrowego. Na zewnątrz do portu można podłączyć jedynie sieć.
- Porty dwukierunkowe wewnątrz i na zewnątrz modułu mogą być podłączone tylko do sieci.
- Nie jest dozwolone aby sygnały zewnętrzne i wewnętrzne miały różną szerokość.

Schematycznie sposób łączenia portów przedstawiono na rysunku 8.



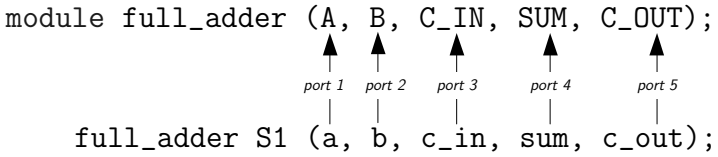
Rysunek 8. Reguły łączenia portów

5.2. Łączenie portów z sygnałami zewnętrznymi

Można wyróżnić dwie metody łączenia wyprowadzeń instancji z sygnałami zewnętrznymi:

- łączenie według kolejności,
- łączenie przez nazwę.

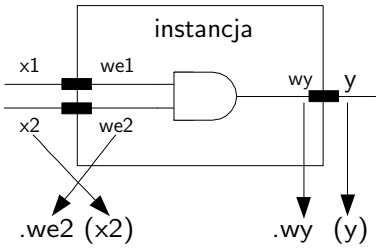
W pierwszej metodzie sygnały łączy się z wyprowadzeniami instancji według kolejności, w jakiej wyprowadzenia te występują w definicji modułu. Pierwszy sygnał jest łączony



Rysunek 9. Łączenie portów według kolejności

z pierwszym wyprowadzeniem instancji, drugi z drugim, itd. Metoda ta jest intuicyjna i została wykorzystana w większości przykładów.

W dużych projektach, gdzie moduły mogą zawierać bardzo długie listy portów, łączenie za pomocą kolejności może okazać się niepraktyczne (trudno zapamiętać kolejność np. 50 portów). W takiej sytuacji wygodniej jest zastosować metodę łączenia sygnałów przez nazwę. W metodzie tej jawnie podaje się nazwę sygnału oraz portu, do którego ma on być przyłączony. Połączenia tworzy się podając nazwę portu poprzedzoną kropką oraz nazwę sygnału w nawiasie okrągłym.



Rysunek 10. Łączenie portów przez nazwę

Może się też zdarzyć, że projektant nie potrzebuje łączyć wszystkich wyprowadzeń instancji. W metodzie łączenia według kolejności na pozycji portu, który ma być niepołączony, należy pozostawić wtedy puste miejsce. Natomiast w metodzie łączenia przez nazwę podawane są tylko te porty, które mają być połączone, a pozostałe pomija się.

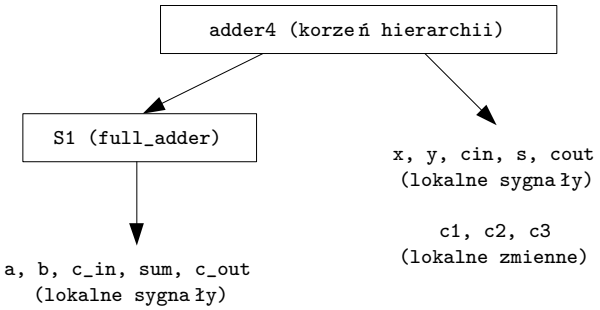
Przykład 20. Łączenie portów przez nazwę

```
full_adder S1 (.A(a), .C_IN(c_in), .B(b), .C_OUT(c_out), .SUM(sum));
```

5.3. Nazwy hierarchiczne

Wszystkie identyfikatory w Verilogu składają się na drzewiastą strukturę, w której każda instancja modułu, sygnał czy zmienna mają swoje własne, niepowtarzalne miejsce. Struktura ta przypomina drzewo plików i katalogów dyskowych. Za jej pomocą można wskazać dowolny identyfikator w projekcie.

Na szczycie hierarchii, w korzeniu drzewa (ang. *root*), znajdują się moduły, które nie mają instancji, a wewnątrz nich osobne gałęzie dla każdej instancji, zadania, funkcji czy bloku.



Rysunek 11. Hierarchia projektu z przykładu 2

Nazwy hierarchiczne tworzone są z identyfikatorów, które rozdziela się kropką. Aby odwołać się do dowolnego identyfikatora należy podać całą ścieżkę w hierarchii, rozpoczynając od modułu, który jest korzeniem, a kończąc na żądanym identyfikatorze.

Przykład 21. Nazwy hierarchiczne

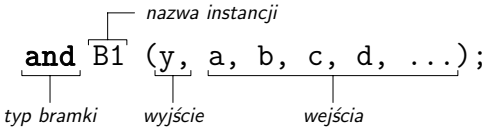
```
adder4.c1
adder4.a
adder4.S1.sum
adder4.S2.c_out
adder4.S3.a
```

6. Poziom bramek

Poziom bramek logicznych obok opisu za pomocą przełączników jest jednym z najniższych poziomów abstrakcji dostępnych w Verilogu. Na tym poziomie opisuje się układ za pomocą bramek logicznych oraz połączeń między nimi. Przy czym nie trzeba definiować podstawowych bramek, ponieważ są one wbudowane w język, a ich użycie jest dokładnie takie samo jak modułów zdefiniowanych przez użytkownika. Dostępne bramki można podzielić na trzy grupy: bramki *and/or*, *buf/not* oraz *bufif/notif*.

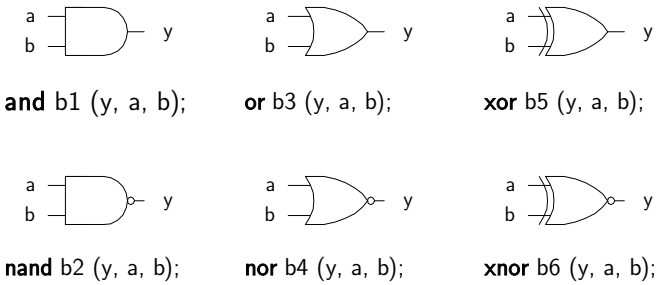
6.1. Bramki and/or

Do tej grupy zalicza się następujące bramki: *and*, *or*, *xor*, *nand*, *nor*, *xnor*. Bramki te posiadają dokładnie jedno wyjście oraz wiele wejść. Kolejność końcówek jest z góry określona i nie może być zmieniana, pierwsze wyprowadzenie jest zawsze wyjściem, a pozostałe wejściami. Ta zasada odnosi się do wszystkich bramek używanych w Verilogu.



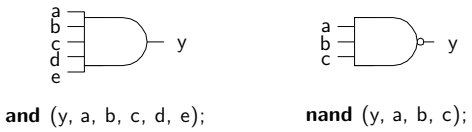
Rysunek 12. Tworzenie instancji bramek logicznych z grupy and/or

Na rysunku 13 przedstawiono graficzne symbole bramek o dwóch wejściach i jednym wyjściu. Pod każdym symbolem znajduje się jego odpowiednik zapisany w Verilogu. Bramki o większej liczbie wyprowadzeń tworzy się dodając kolejne wejścia w liście portów. Przykład takich bramek przedstawia rysunek 14.



Rysunek 13. Bramki logiczne z grupy and/or

W przypadku podstawowych bramek dopuszczalne jest opuszczenie nazw instancji, jak pokazano na rysunku 14. Takie rozwiązanie ułatwia tworzenie modeli składających się z wielu bramek ponieważ nie trzeba określać nazwy dla każdej z nich.



Rysunek 14. Wielowejściowe bramki logiczne z grupy and/or

Tabela 4 pokazuje w jaki sposób wyznaczany jest stan na wyjściu bramki w zależności od stanu jej wejść. Tabele prawdy odpowiadają bramkom dwuwejściowym. W przypadku większej ilości wejść stosuje się te same zasady. Na szczególną uwagę zasługują te przypadki, w których na wejściach pojawią się stany `x` bądź `z`.

Tabela 4. Tabele prawdy dla bramek and/or

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

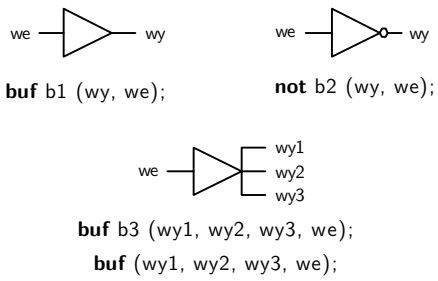
nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

6.2. Bramki buf/not

Bufory są reprezentowane w Verilogu przez dwie bramki: bramkę *buf*, która przenosi stan z wejścia na wiele wyjść oraz bramkę *not*, która dodatkowo neguje sygnał wejściowy. Bramki te stosuje się gdy zachodzi potrzeba rozdzielenia sygnału na wiele bramek, wzmocnienia go lub wprowadzenia opóźnienia.

Bramki *buf/not* posiadają jedno skalarne wejście i wiele skalarnych wyjść. Tworząc instancje w liście portów najpierw podaje się wszystkie wyjścia a na końcu wejście. Rysunek 15 przedstawia graficzne symbole buforów oraz odpowiadający im kod w Verilogu, tabele prawdy dla tych bramek znajdują się w tabeli 5.



Rysunek 15. Bramki logiczne z grupy buf/not

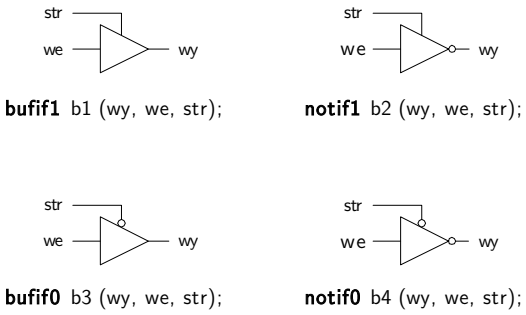
Tabela 5. Tabele prawdy dla bramek buf/not

buf	we	wy
	0	1
	1	0
	x	x
	z	x

not	we	wy
	0	0
	1	1
	x	x
	z	x

6.3. Bramki bufif/notif

Grupa *bufif/notif* reprezentuje bufory trójstanowe. Do tej grupy zalicza się bramki: *bufif1*, *bufif0*, *notif1*, *notif0*. Bramki *bufif/notif* przenoszą sygnał na wyjście tylko wtedy, gdy sygnał sterujący jest aktywny, w przeciwnym wypadku na wyjściu bramki pojawia się stan wysokiej impedancji (`z`). Bufory trójstanowe stosuje się przede wszystkim tam, gdzie zachodzi potrzeba podłączenia wielu sygnałów do jednej linii. Odpowiednie przełączanie sygnałów sterujących pozwala wtedy tak sterować dostępem, aby w jednym momencie linia była sterowana tylko z jednego źródła.



Rysunek 16. Bramki logiczne z grupy bufif/notif

Bramki *bufif/notif* mają jedno skalarne wejście danych, jedno wejście sterujące oraz jedno skalarne wyjście. Pierwsza końcówka jest wejściem, druga wyjściem, natomiast trzecia przeznaczona jest dla sygnału sterującego. W przypadku tej grupy nie można tworzyć bramek o większej liczbie wejść, tak więc rysunek 16 przedstawia wszystkie dostępne kombinacje. Tabele prawdy dla buforów trójstanowych zawierają dwa dodatkowe stany: `L` oznacza stan `0` lub `z`, natomiast `H` oznacza `1` lub `z`.

Tabela 6. Tabele prawdy dla bramek bufif/notif

	sygnał sterujący			
notif1	0	1	x	z
0	z	1	h	h
1	z	0	l	l
x	z	x	x	x
z	z	x	x	x

	sygnał sterujący			
notif0	0	1	x	z
0	1	z	h	h
1	0	z	l	l
x	x	z	x	x
z	x	z	x	x

	sygnał sterujący			
bufif1	0	1	x	z
0	z	0	l	l
1	z	1	h	h
x	z	x	x	x
z	z	x	x	x

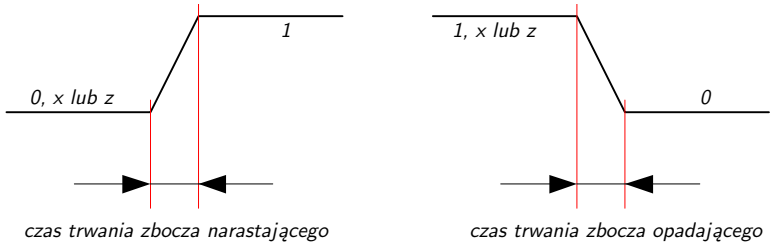
	sygnał sterujący			
bufif0	0	1	x	z
0	0	z	l	l
1	1	z	h	h
x	x	z	x	x
z	x	z	x	x

6.4. Opóźnienia bramek

Wszystkie bramki opisane w pierwszej części rozdziału przedstawiono bez modelowania opóźnień. Oznacza to, że działają one z nieskończoną szybkością i stan wyjść jest określany natychmiast po zmianie sygnałów wejściowych. Do symulowania rzeczywistych układów, mających skończoną prędkość pracy, często zachodzi potrzeba uzupełnienia definicji bramek o dodatkowe informacje o wprowadzanych przez nie opóźnieniach.

Język Verilog pozwala na definiowanie opóźnień dla dowolnych bramek logicznych. Opóźnienia te można podzielić na trzy typy:

- opóźnienie przy narastającym zboczu (ang. *rise delay*) – czas przejścia sygnału ze stanu `0`, `x` lub `z` do stanu `1`,
- opóźnienie przy opadającym zboczu (ang. *fall delay*) – czas przejścia sygnału ze stanu `1`, `x` lub `z` do stanu `0`,
- opóźnienie przy wyłączeniu (ang. *turn-off*) – przejście do stanu wysokiej impedancji `z` z dowolnego innego stanu.



Rysunek 17. Opóźnienia przy narastającym i opadającym zboczu

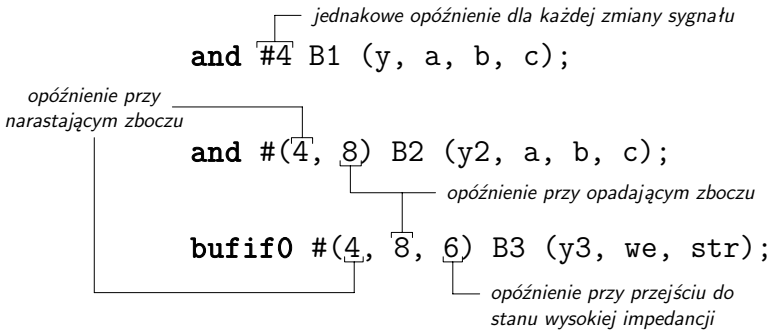
Dla bramek z grupy *and/or* oraz *buf/not* można ustalić opóźnienia przy narastającym i opadającym zboczu. Na wyjściu tych bramek nie może pojawić się stan wysokiej

impedancji (z), dlatego nie możliwe jest podanie opóźnienia przy wyłączeniu. Próba ustalenia tego typu opóźnienia spowoduje wystąpienie błędu podczas kompilacji.

Opóźnienia można zapisywać na kilka sposobów. Jeśli podana jest jedna wartość będzie ona użyta przy każdej zmianie stanu. Gdy podane zostaną dwie wartości, pierwsza jest traktowana jako opóźnienie przy narastającym zboczu, natomiast druga jako opóźnienie przy opadającym zboczu. W przypadku przejścia do stanu x, wybierana jest najmniejsza z tych dwóch wartości.

Dla bramek z grupy *bufif/notif* opóźnienia można podawać podobnie jak w przypadku pozostałych bramek lub dodatkowo rozszerzyć o trzecią wartość, czyli opóźnienie przy przejściu do stanu wysokiej impedancji. Gdy podane są dwie wartości, to najmniejsza z nich używana jest przy przełączaniu do stanu x i z. Jeśli zostaną podane trzy wartości, pierwsza jest traktowana jako czas narastającego zbocza, druga jako czas opadającego zbocza, natomiast trzecia jako czas przejścia do stanu wysokiej impedancji. W przypadku przełączenia do stanu x zostanie użyta najmniejsza z tych trzech wartości.

Na rysunku 18 pokazano schematycznie jak specyfikować opóźnienia. Jeżeli podana jest tylko jedna wartość to, można pominąć nawiasy.



Rysunek 18. Specyfikacja opóźnień przy narastającym i opadającym zboczu oraz przy przejściu do stanu wysokiej impedancji

Tabela 7 w zwięzły sposób przedstawia jak wyliczane są opóźnienia. Uwzględniono w niej wszystkie możliwe zmiany stanów oraz odpowiadające im wartości opóźnień. W drugiej kolumnie zawarto wyniki dla wariantu z dwoma opóźnieniami, natomiast w trzeciej z trzema.

6.5. Opóźnienia minimalne, typowe i maksymalne

Definiowanie opóźnień można jeszcze bardziej uszczegółowić podając minimalne, typowe oraz maksymalne wartości każdego z nich. Projektant może przed i w czasie symulacji decydować, która wartość opóźnienia ma być używana przez symulator do testowania układu. W ten sposób możliwe jest, uwzględnienie w symulacji zmian czasu

Tabela 7. Wartości opóźnień przy różnych zmianach stanu

zmiana stanu	opóźnienie	
	dwie wartości	trzy wartości
0 ⇒ 1	w1	w1
0 ⇒ x	min(w1,w2)	min(w1,w2,w3)
0 ⇒ z	min(w1,w2)	w3
1 ⇒ 0	w2	w2
1 ⇒ x	min(w1,w2)	min(w1,w2,w3)
1 ⇒ z	min(w1,w2)	w3
x ⇒ 0	w2	w2
x ⇒ 1	w1	w1
x ⇒ z	min(w1,w2)	w3
z ⇒ 0	w2	w2
z ⇒ 1	w1	w1
z ⇒ x	min(w1,w2)	min(w1,w2,w3)

propagacji bramek na skutek wahań temperatury lub innych czynników bez potrzeby tworzenia różnych wersji projektu.

Sposób wyboru wartości do symulacji oraz ich ewentualnych zmian podczas jej trwania zależy od używanego symulatora oraz systemu operacyjnego.

Przykład 22. Opóźnienia minimalne, typowe i maksymalne

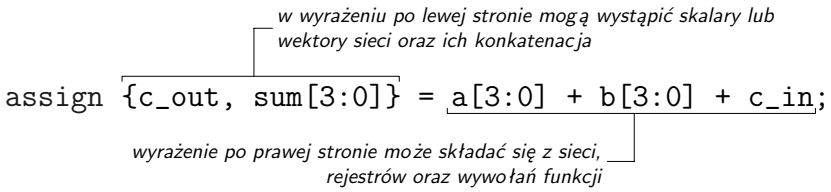
```
//jedno opóźnienie
and #(4:5:6) A1 (y, a, b);

//dwa opóźnienia
and #(4:5:6, 2:3:4) A2 (y, a, b);

//trzy opóźnienia
bufif1 #(4:5:6, 2:3:4, 3:4:5) A2 (y, a, ctrl);
```

7. Poziom przepływu danych

7.1. Przypisania ciągłe

Przypisanie ciągłe jest podstawową instrukcją stosowaną w modelowaniu przepływu danych. Za jego pomocą można modelować logikę kombinacyjną bez użycia bramek logicznych. W zamian model opisuje się korzystając wyrażeń logicznych, które sterują sieciami.

Przypisanie ciągłe rozpoczyna się słowem kluczowym `assign`, po którym występuje wyrażenie logiczne. Lewa strona przypisania musi być skalarem lub wektorem sieci, nie jest dozwolone stosowanie rejestrów. Operandami po prawej stronie wyrażenia mogą być rejestry, sieci lub wywołania funkcji.



Rysunek 19. Przypisanie ciągłe

Przypisanie ciągłe jest aktywne przez cały czas trwania symulacji. Każda zmiana stanu operandów występujących po prawej stronie wyrażenia powoduje jego ponowne przeliczenie. Obliczona wartość może być przypisana do sieci natychmiast lub z pewnym opóźnieniem.

Istnieje również możliwość połączenia deklaracji sieci z przypisaniem ciągłym. Przypisanie tego typu nazywa się niejawnym (ang. *implicit continuous assignment*)

Przykład 23. Przypisanie niejawne

```
wire y = a & b;  // przypisanie niejawne

wire y;          // równoważny zapis z użyciem słowa kluczowego assign
assign y = a & b;
```

7.2. Opóźnienia

Za pomocą opóźnień można podać czas, który ma upłynąć od momentu wystąpienia zmiany operandu po prawej stronie przypisania do uaktualnienia sieci po lewej stronie.

Można wyróżnić trzy metody specyfikacji opóźnień. Pierwsza z nich polega na podaniu opóźnienia w przypisaniu ciągłym zaraz po słowie `assign` (ang. *regular assignment delay*).

Przykład 24. Opóźninenie w przypisaniu ciągłym

```
wire y;
assign #10 y = a & b;
```

Druga metoda polega na podaniu opóźnienia w przypisaniu niejawnym (ang. *implicit continuous delay*). Opóźnienie takie podaje się po typie sieci.

Przykład 25. Opóźninenie w niejawnym przypisaniu ciągłym

```
wire #10 y = a & b;
```

Ostatnią metodą specyfikacji opóźnień jest zadeklarowanie sieci z opóźnieniem (ang. *net declaration delay*). Czas opóźnienia jest podawany podczas deklaracji sieci przez co każde przypisanie nowej wartości jest wykonywane z tym opóźnieniem.

Przykład 26. Deklaracja sieci z opóźnieniem

```
wire #10 y;
assign y = a & b;
```

7.3. Operatory

W tabeli 8 przedstawiono wszystkie dostępne operatory Veriloga pogrupowane według typu. Następne rozdziały zawierają dokładny opis każdej grupy.

7.3.1. Operatory arytmetyczne

Do grupy operatorów arytmetycznych zalicza się:

- dodawanie (`+`),
- odejmowanie (`-`),
- mnożenie (`*`),
- dzielenie (`/`),
- modulo (`%`).

Operatory (`+`) i (`-`) występują jako jedno i dwuargumentowe. Jednoargumentowe wersje są stosowane do określania znaku operandu (czy jest on dodatni czy ujemny) i mają wyższy priorytet niż operatory dwuargumentowe realizujące dodawanie i odejmowanie.

Tabela 8. Typy oraz symbole operatorów w Verilogu

Typ operatora	Symbol operatora	Wykonywana operacja	Liczba operandów
Arytmetyczny	*	Mnożenie	2
	/	Dzielenie	2
	+	Dodawanie	2
	-	Odejmowanie	2
	%	Modulo	2
Logiczny	!	Negacja logiczna	1
	&&	Logiczny AND	2
	||	Logiczny OR	2
Relacyjny	>	Mniejszy	2
	<	Mniejszy	2
	>=	Większy lub równy	2
	<=	Mniejszy lub równy	2
Równości	== oraz ===	Równy	2
	!= oraz !==	Nierówny	2
Bitowy	~	Negacja bitowa	1
	&	Bitowy AND	2
	|	Bitowy OR	2
	^	Bitowy XOR	2
	^~ lub ~^	Bitowy XNOR	2
Redukcji	&	Redukcja AND	1
	~&	Redukcja NAND	1
	|	Redukcja OR	1
	~|	Redukcja NOR	1
	^	Redukcja XOR	1
	^~ lub ~^	Redukcja XNOR	1
Przesunięć	>>	Przesunięcie w lewo	2
	<<	Przesunięcie w prawo	2
Konkatenacji	{}	Konkatenacja	Dowolna liczba
Powielania	{ { } }	Powielanie	Dowolna liczba
Warunkowy	?:	Warunek	3

Operator modulo (%) daje resztę z dzielenia pierwszego operandu przez drugi. Znak wyniku jest zgodny ze znakiem pierwszego operandu. Jeżeli operandy dzielą się dokładnie, to wynikiem jest 0.

W przypadku wszystkich operatorów arytmetycznych, jeżeli którykolwiek operand zawiera wartości `x` lub `z`, to w rezultacie otrzymuje się wynik składający się z samych wartości `x`.

Operacje arytmetyczne dla typu `reg` należy traktować inaczej niż dla typu `integer`. Dla typu `reg` operandy rozpatruje się jako wartości bez znaku, a dla typu `integer` jako wartości ze znakiem. Ujemna liczba zostanie zapisana w rejestrze typu `reg` jako liczba w kodzie uzupełnienie do dwóch. Jeśli rejestr ten zostanie użyty w wyrażeniu arytmetycznym, to liczba z rejestru zostanie potraktowana jako liczba bez znaku. Może

Przykład 27. Operatory arytmetyczne

```
reg [3:0] a, b;
integer c, d;

a = 4'b0010; b = 4'b0100;
c = 8; d = 5;
...
a * b;   // mnożenie, wynik 4'b1000
c / d;   // dzielenie, wynik 1; odcięto część ułamkową
a + b;   // dodawanie, wynik 4'b0110
b - a;   // odejmowanie, wynik 4'b0010
c % d;   // modulo, wynik 3
```

to prowadzić do błędów, które nie są sygnalizowane podczas kompilacji. Błędów tych można uniknąć stosując rejestr typu `integer`, gdzie ujemna liczba zostanie zapisana w kodzie uzupełnienie do dwóch i w wyrażeniach arytmetycznych będzie traktowana jako liczba ze znakiem.

7.3.2. Operatory relacyjne

Do operatorów relacyjnych (ang. *relational operators*) zalicza się następujące operatory:

- większy (`>`),
- mniejszy (`<`),
- większy lub równy (`>=`),
- mniejszy lub równy (`<=`).

Wyrażenie, w którym występują operatory relacyjne daje w wyniku 0 jeżeli relacja jest niespełniona lub 1 jeśli relacja jest prawdziwa. Jeśli operandy zawierają wartość nieznaną lub wysoką impedancję wynikiem jest zawsze wartość `x`. W przypadku gdy operandy są różnej szerokości to mniejszy operand jest rozszerzany do większego przez powielenie najbardziej znaczącego bitu. Wszystkie operatory relacyjne mają ten sam priorytet.

Przykład 28. Operatory relacyjne

```
a = 2; b = 3;
k = 4'b0111; m = 4'b1100; n = 4'b0xxx;

a < b    // wynik 1
a > b    // wynik 0
k <= m   // wynik 1
k >= w   // wynik x, ponieważ w operandach są wartości x
```

7.3.3. Operatory równości

Występują cztery operatory równości (ang. *equality operators*):

- równości (==),
- równości z uwzględnieniem x i z (===),
- nierówności (!=),
- nierówności z uwzględnieniem x i z (!==),

Operatory równości porównują operandy bit po bicie. Przy operandach o różnej długości krótszy z nich jest rozszerzany zerami. Podobnie jak w przypadku operatorów relacyjnych 0 oznacza, że porównanie jest nieprawdziwe, natomiast 1, że jest prawdziwe. Operatory dwuznakowe (==, !=) i trzyznakowe (===, !==) różnie traktują operandy zawierające wartości x i z. W przypadku operatorów (==, !=) porównywane są ze sobą tylko wartości 0 i 1. Jeżeli w operandach wystąpią wartości x lub z, to wynikiem porównania będzie zawsze wartość nieznana (x). Operatory (===, !==) działają inaczej. W porównaniach uwzględniają one również wartości x i z, przez co wynik jest zawsze określony jako 1 lub 0.

Przykład 29. Operatory równości

```
a = 4; b = 3;
k = 4'b0111; m = 4'b1100;
u = 4'b0xxz; t = 4'b0xxz; w = 4'b0xxx;

a == b   // wynik 0
a != b   // wynik 1
u == t   // wynik x, ponieważ w operandach są wartości x i z

u === t  // wynik 1, ponieważ wszystkie bity są identyczne
u === w  // wynik 0, ponieważ najmłodsze bity różnią się
k !== w  // wynik 1
```

7.3.4. Operatory logiczne

W Verilogu wystepują trzy operatory logiczne (ang. *logical operators*):

- logiczny iloczyn(&&)
- logiczna suma (||)
- logiczna negacja (!)

Jeżeli operand jest równy 0 to jego wartość logiczna wynosi również 0, gdy operand jest różny od zera, jego wartość logiczna jest równa jeden. Natomiast jeśli zawiera wartości x lub z, to wynik przyjmuje wartość x.

Przykład 30. Operatory logiczne

```
a = 425; b = 0;

a && b;  // wynik 0
a || b;  // wynik 1

(a == 10) && (b == 0)  // wynik 0
(a == 0) || (b == 0)   // wynik 1
```

W wyniku działań logicznych można otrzymać wartość 1 traktowaną jako prawda, wartość 0 traktowaną jako fałsz oraz wartość `x` jeśli wynik jest nieznany. Priorytet operatora (`&&`) jest wyższy niż operatora (||), jednak oba operatory mają niższy priorytet niż operatory porównania.

7.3.5. Operatory bitowe

Do grupy operatorów bitowych (ang. *bit-wise operators*) należą: and (`&`), or (|), xor (`^`), xnor (`^~` lub `~^`) oraz (`~`). Operatory te wykonują działania na poszczególnych bitach operandów - pierwszy bit z pierwszym, drugi z drugim itd. Jeśli operandy mają różną długość, krótszy z nich uzupełniany jest z lewej strony zerami. W przypadku operatorów bitowych wartość `z` jest traktowana jak `x`.

Tabela 9. Tabele prawdy dla operatorów bitowych

bitowy and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitowy or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitowy xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitowy xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

negacja	wynik
0	1
1	0
x	x

Należy pamiętać o różnicy między operatorami bitowymi i logicznymi. Operatory logiczne traktują operandy jako całość natomiast operatory bitowe wykonują działania bit po bicie. Rezultatem wyrażeń z operatorami logicznymi mogą być tylko trzy wartości 0, 1, `x`, a wynikiem wyrażeń z operatorami bitowymi są wartości bitowe.

Przykład 31. Operatory bitowe

```
a = 4'b1100; b = 4'b0110; c = 4'bx0x1

~a       // negacja, wynik 4'b0011
a & b    // and, wynik 4'b0100
a | b    // or, wynik 4'b1110
a ^ b    // xor, wynik 4'b1010
a ^~ b   // xnor, wynik 4'b0101
~c       // negacja, wynik 4'bx1x0
```

7.3.6. Operatory redukcji

Operatory redukcji (ang. *reduction operators*) to: and (`&`), nand (`~&`), or (`|`), xor (`^`), xnor (`^~` lub `~^`). Wykonują one operacje na poszczególnych bitach jednego operatora - pierwszy bit z drugim, drugi z trzecim itd. Po wykonaniu operacji na wszystkich bitach otrzymuje się jednobitowy wynik. Redukcja **nand**, **nor** oraz **xnor** jest obliczana przez zanegowanie odpowiednio redukcji **and**, **or** oraz **xor**.

Przykład 32. Operatory redukcji

```
a = 4'b1011;

&a    // (1 & 0 & 1 & 1) = 1b'0
|a    // (1 | 0 | 1 | 1) = 1b'1
^a    // (1 ^ 0 ^ 1 ^ 1) = 1b'1
```

7.3.7. Operatory przesunięć

Do grupy operatorów przesunięć (ang. *shift operators*) zalicza się:

- operator przesunięcia w lewo (`<<`),
- operator przesunięcia w prawo (`>>`).

Operatory te są dwuargumentowe. Lewy operand jest wektorem, którego bity przesuwane są o liczbę pozycji zapisanych w prawym operandzie. Puste miejsca, powstałe na skutek przesunięcia, są wypełniane zerami. Jeżeli prawy operand zawiera wartość nieznaną lub wysoką impedancję cały wynik przesunięcia jest nieznany.

Przykład 33. Operatory przesunięć

```
a = 12'b1100_1001_1011; b = 4;

a << b    // wynik po przesunięciu 12'b1001_1011_0000;
a >> b    // wynik po przesunięciu 12'b0000_1100_1001;
```

Operatory przesunięcia są często wykorzystywane przy operacjach mnożenia i dzielenia. Przesunięcie w lewo o jedną pozycję jest równoznaczne z pomnożeniem liczby przez 2, natomiast przesunięcie w prawo z podzieleniem przez 2.

7.3.8. Operatory konkatenacji

Operator konkatenacji (ang. *concatenation operator*) służy do łączenia dowolnej liczby operandów w jeden wektor. Operandy należy zapisać w nawiasie klamrowym rozdzielając je przecinkami. Wszystkie operandy konkatenacji muszą mieć określony rozmiar.

Przykład 34. Operatory konkatenacji

```
a = 4'b1010; b = 4'b0000; c = 4'b1111;

{a, b, c}                  // 12'b1010_0000_1111;
{4'b1011, b, 4'b0110}      // 12'b1011_0000_0110;
{a[0], b[1], c[2], 1'b1}   // 4'b0011
```

7.3.9. Operator powielania

Operator powielania (ang. *replication operator*) to złożona konkatenacja. Za pomocą tego operatora można określić ile razy dany operand ma być powielony .

Przykład 35. Operator powielania

```
a = 4'b1010; b = 4'b0000;

{3{a}}                     // 12'b1010_1010_1010;
{2{4'b1001}, 4{1'b1}}      // 12'b1001_1001_1111;
{2{3{a[0]}, 3{b[1]}}}      // 12'b111000_111000;
```

7.3.10. Operator warunkowy

Operator warunkowy (ang. *conditional operator*) jest trzyargumentowy. Pierwszy operand jest warunkiem, drugi jest wartością dla całego wyrażenia warunkowego, jeżeli warunek jest prawdziwy, a trzeci jest wartością dla całego wyrażenia warunkowego, jeżeli warunek jest fałszywy. Jeżeli warunek przyjmuje wartość `x` lub `z` to wykonywane są wyrażenia dla prawdy i fałszu, a następnie na ich podstawie bit po bicie obliczane jest wyrażenie końcowe zgodnie z tabelą 10.

Operatory warunkowe mogą być zagnieżdżane. W przykładzie 37 przedstawiono multiplekser 4 na 1 opisany za pomocą dwóch operatorów warunkowych. Sygnał `sel` kontroluje wejścia adresowe multipleksera, sygnały wejściowe to: `inA`, `inB`, `inC`, `inD`, a wyjściowy to `out`.

Tabela 10. Wartość wyrażenia warunkowego przy nieznanym warunku

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Przykład 36. Operator warunkowy

```
// funkcjonalny model bufora trójstanowego
assign addr_bus enable ? addr_out : 32'hzzzz_zzzz;

// funkcjonalny model multipleksera 2 do 1
assign out = ctrl ? we1 : we2;
```

Przykład 37. Zagnieżdżenie operatora warunkowego

```
assign out = sel[1] ? (sel[0] ? inA : inB) : (sel[0] ? inC : inD);
```

7.3.11. Priorytet operatorów

Jeżeli wyrażenie zawiera wiele operatorów są one rozpatrywane zgodnie z określoną kolejnością. Każdy operator ma swój priorytet. Operatory o najwyższym priorytecie są rozpatrywane w pierwszej kolejności, następnie interpretowane są operatory o niższym priorytecie. Jeżeli w wyrażeniu występuje szereg operatorów o tym samym priorytecie wtedy są one rozpatrywane od lewej do prawej, nie licząc operatora warunkowego, który jest rozpatrywany od prawej do lewej.

Tabela 11. Priorytet operatorów

`+ - ! ~`	najwyższy priorytet
`* / %`	

`+ -`
`<< >>`
`< <= > >=`
`== != === !==`
`& ~&`
`^ ^~ ~^`
`| ~|`
`&&`	

`||`	
`?:`	najniższy priotytet

8. Poziom behawioralny

Omawiane do tej pory konstrukcje były bezpośrednio związane ze sprzętem. Modelowanie układów za pomocą bramek logicznych i przypisań ciągłych dość dokładnie odpowiada strukturze logicznej układu cyfrowego. Jednak konstrukcje te nie pozwalają projektować na wysokim poziomie abstrakcji, który jest wymagany w przypadku złożonych systemów. Dopiero konstrukcje proceduralne opisane w tym rozdziale pozwolą na tworzenie złożonych projektów takich jak mikroprocesory czy rozbudowane sterowniki.

W Verilogu modele behawioralne składają się z konstrukcji proceduralnych (ang. *procedural statement*), które pozwalają kontrolować symulację i manipulować wcześniej zadeklarowanymi zmiennymi. Konstrukcje te zamyka się w procedurach lub blokach strukturalnych (ang. *structured statements*).

Można wyróżnić dwa bloki strukturalne: `initial` i `always`. Każdy z wymienionych bloków wykonywany jest jako osobny proces. Poszczególne procesy są wykonywane współbieżnie co umożliwia modelowanie równoległości występującej w układach cyfrowych. Przykład 38 przedstawia prosty model behawioralny.

Przykład 38. Model behawioralny

```
module zachowanie;
reg a, b;

  initial begin
    a = 'b1;
    b = 'b0;
  end

  always begin
    #50 a = ~a;
  end

  always begin
    #100 b = ~b;
  end

  initial begin
    #5000 $finish;
  end

endmodule
```

Przy starcie symulacji, gdy czas jest równy 0, dla każdego bloku `initial` i `always` tworzony jest osobny proces. Następnie równolegle wykonywane są instrukcje znajdujące się wewnątrz tych bloków. Instrukcje w bloku `initial` wykonywane są tylko jeden raz, natomiast instrukcje w bloku `always` powtarzane są wielokrotnie w nieskończonej pętli.

W modelu z przykładu 38 do zmiennych `a` i `b` przypisywane są odpowiednio wartości `1` i `0` w zerowym czasie symulacji. Po wykonaniu przypisań działanie pierwszego bloku `initial` kończy się i nie będzie on już wywoływany podczas symulacji. Konstrukcje `always` również rozpoczynają swoje działanie w czasie 0. Jednak negacja rejestrów zostanie wykonana dopiero gdy upłynie czas opóźnień podany po znaku `#`. Rejestr `a` zmieni swoją wartość po 50 jednostkach czasu, a rejestr `b` po 100 jednostkach. Instrukcje w bloku `always` są powtarzane wielokrotnie, zatem ich wykonanie spowoduje cykliczne zmiany sygnałów `a` i `b` odpowiednio co 100 i 200 jednostek czasu.

Blok `initial` znajdujący na końcu modułu jest przeznaczony do zakończenia symulacji po 5000 jednostek czasu. Jeżeli w modelu nie byłoby instrukcji zatrzymującej symulację (`$stop` lub `$finish`) to, wykonywałaby się ona w nieskończoność.

8.1. Przypisania proceduralne

Za pomocą przypisań proceduralnych (ang. *procedural assignments*) można aktualizować wartości zmiennych typu: `reg`, `integer`, `real` i `time`. Po wykonaniu przypisania wartość zmiennej jest przechowywana aż nie zostanie zmieniona przez kolejne przypisanie.

Prawa strona przypisania proceduralnego może być dowolnym wyrażeniem, którego wykonanie daje wartość. Do budowy wyrażeń można wykorzystać wszystkie operatory opisane w rozdziale 7. Po obliczeniu wyrażenia jego wartość jest przypisywana do zmiennej po lewej stronie, gdzie może wystąpić:

- nazwa zmiennej typu `reg`, `integer`, `real`, `realtime` lub `time`,
- wybrany bit rejestrów typu `reg`, `integer` lub `time` np. `adres[0]`,
- wybrany ciągły fragment rejestru np. `adres[7:0]`,
- pojedyncze słowo z pamięci np. `pamiec[adres] = 8'h77`,
- konkatenacja wszystkich wyżej wymienionych elementów
  np. `{c_out, sum} = a + b`

W Verilogu występują dwa rodzaje przypisań:

- blokujące (ang. *blocking assignment*),
- nieblokujące (ang. *nonblocking assignment*).

Przypisania te różnią się tym, że inaczej wpływają na kolejność przetwarzania instrukcji występujących w blokach sekwencyjnych. Więcej informacji o blokach sekwencyjnych i równoległych znajduje się w rozdziałach 8.7.

8.1.1. Przypisania blokujące

Przypisania blokowane wykonywane są po kolei jedno za drugim, dokładnie w takiej kolejności w jakiej występują w projekcie. Słowo „blokowane" oznacza, że symulacja jest zatrzymywana w procesie na danym przypisaniu i kontynuowana dopiero po jego wykonaniu. Przypisania blokowane zapisuje się stosując operator (=).

Przykład 39. Przypisania blokowane

```
reg a, b;

initial begin
  a = 1;      // etap 1 a = 1; b = x;
  b = a;      // etap 2 a = 1; b = 1;
  a = ~b;     // etap 3 a = 0; b = 1;
end
```

Symulacja przykładu 39 odbywa się w trzech etapach. Najpierw rozpatrywane jest przypisanie `a = 1`, następnie możliwe jest wykonanie przypisania `b = a`, a po nim `a = ~b`. Poszczególne przypisania są zależne od siebie i muszą być rozpatrywane z uwzględniem zachodzących między nimi zależności. Stosując przypisania blokowane można mieć pewność, że podana przez projektanta kolejność przypisań zostanie uwzględniona przez symulator.

8.1.2. Przypisania nieblokujące

Przypisania nieblokujące pozwalają przypisywać wartości nie blokując wykonania kolejnych instrukcji. Mogą być stosowane wszędzie tam, gdzie istnieje potrzeba przypisań rejestrowych, które nie muszą być wykonywane w określonej kolejności i nie zależą od siebie.

Do tworzenia przypisań nieblokujących stosuje się operator (<=), który jest używany również jako operator relacyjny. Stąd interpretacja tego operatora zależy od kontekstu jego użycia. Jeśli operator jest użyty w wyrażeniu to, zostanie zinterpretowany jako operator relacyjny, natomiast w przypadku przypisania nieblokującego jako operator przypisania.

Wykonanie przypisań nieblokujących odbywa się w dwóch etapach. W pierwszym etapie obliczane są wartości wszystkich przypisań rozpatrywanych w danej jednostce czasowej, które następnie są zapamiętywane przez symulator. W drugim etapie, tuż przed zakończeniem symulacji aktualnej jednostki czasowej, symulator uaktualnia rejestry wcześniej zapamiętanymi wartościami.

Przykład 40. Przypisania nieblokujące

```
reg a, b;

initial begin
  a = 0;
  b = 1;
end

always begin
  #10;        // czekaj 10 jednostek
  a <= b;     // przypisania wykonywane sa w tej samej
  b <= a;     // jednostce czasu w dwoch etapach
end
```

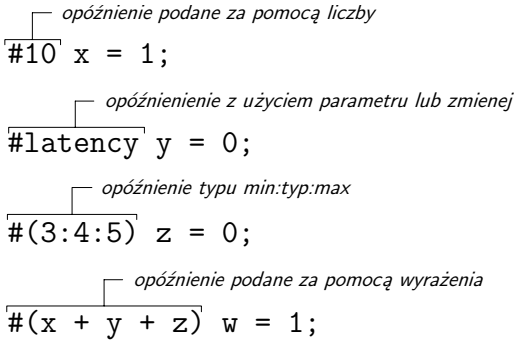
8.2. Sterowanie czasem wykonania przypisań

W Verilogu na poziomie behawioralnym dostępne są trzy metody kontrolowania czasu wykonania przypisań:

- za pomocą opóźnień,
- za pomocą zdarzeń,
- za pomocą opóźnień z uwzględnieniem poziomu (wait).

Sterowanie czasem wykonania za pomocą opóźnień polega na podaniu czasu, który ma upłynąć od momentu napotkania przypisania do jego wykonania. Kontrola czasowa tego typu była stosowana w większości wcześniejszych przykładów.

Do zapisu opóźnień stosuje się symbol #. Opóźnienie może być podane jako liczba, parametr lub wyrażenie. Dozwolone sposoby specyfikowania opóźnień przedstawia rysunek 20.



Rysunek 20. Różne sposoby specyfikacji opóźnień

W przypisaniach proceduralnych opóźnienia można wprowadzać przed przypisaniem - opóźnienie regularne (ang. *regular delay*) lub wewnątrz przypisania - opóźnienie wewnętrzne (ang. *intra-assignment delay*). W przypadku opóźnienia regularnego wartość wyrażenia jest obliczana i przypisywana do rejestru dopiero po upływie opóźnienia. Przykłady kontroli za pomocą opóźnień regularnych pokazano na rysunku 20. W przypadku opóźnień wewnętrznych jest nieco inaczej. Opóźnienie podaje się po prawej stronie przypisania proceduralnego. Wartość wyrażenia jest obliczana natychmiast po napotkaniu przypisania, natomiast zaps do rejestru odbywa się dopiero po upłynięciu czasu opóźnienia.

8.3. Instrukcja warunkowa

Konstrukcja warunkowa (ang. *conditonal satement*) jest używana do podejmowania decyzji czy instrukcje mają być wykonane czy nie. Do tworzenia warunków służą słowa kluczowe `if` oraz `else`.

Przykład 41. Instrukcja if

```
if (warunek)
  instrukcja lub blok instrukcji
else
  instrukcja lub blok instrukcji
```

Jeżeli wyrażenie w nawiasie jest prawdziwe (przyjmuje wartość różną od 0), to wykonywana jest instrukcja występująca zaraz po warunku. Natomiast jeżeli wyrażenie w nawiasie jest nieprawdziwe (przyjmuje wartość 0, `x`, lub `z`), to wykonywana jest instrukcja po słowie kluczowym `else`. W przypadku gdy w rozgałęzieniach konstrukcji warunkowej występuje więcej niż jedna instrukcja, należy je zgrupować stosując słowa kluczowe `begin` i `end`.

W instrukcji warunkowej dozwolone jest pominięcie części ze słowem kluczowym `else`. W przypadku wielokrotnego zagnieżdżania instrukcji `if`, może to powodować

niejasności. Dlatego przyjmuje się zasadę, że słowo kluczowe `else` odnosi się zawsze do najbliższego słowa `if`. W przykładzie 42 słowo kluczowe `else` odnosi się do `if (b > c)`.

Przykład 42. Domyślne łączenie `if` z `else`

```
if (a > b)
  if (b > c)
    wynik = b;
  else
    wynik = c;
```

W razie potrzeby można wymusić inne wiązanie słów kluczowych `if-else` stosując blok `begin-end` jak pokazano w przykładzie 43

Przykład 43. Wymuszone łączenie `if` z `else` za pomocą bloku `begin-end`

```
if (a > b) begin
  if (b > c)
    wynik = b;
end
else wynik = c;
```

8.4. Konstrukcja *case*

Słowo kluczowe `case` pozwala tworzyć wyrażenia warunkowe zawierające serie decyzji, w których zmienne lub wyrażenia są sprawdzane oddzielnie pod kątem tego, czy wystąpiła każda z integralnych wartości, które mogą przyjąć. Wyrażenia w konstrukcji `case` mogą być obliczane podczas symulacji i nie jest wymagane, aby którekolwiek z nich było stałe.

Przykład 44. Konstrukcja `case`

```
reg [3:0] a;
reg [9:0] wynik;

case (a)
  4'd0 : wynik = 16'b0111111111;
  4'd1 : wynik = 16'b1011111111;
  4'd2 : wynik = 16'b1101111111;
  4'd3 : wynik = 16'b1110111111;
  4'd4 : wynik = 16'b1111011111;
  4'd5 : wynik = 16'b1111101111;
  4'd6 : wynik = 16'b1111110111;
  4'd7 : wynik = 16'b1111111011;
  4'd8 : wynik = 16'b1111111101;
  4'd9 : wynik = 16'b1111111110;
  default: wynik = 'bx;
endcase
```

Poszczególne rozgałęzienia instrukcji `case` są rozpatrywane według kolejności ich występowania. Podczas wykonywania instrukcji porównywane jest wyrażenie w nawiasie z każdym rozgałęzieniem. Jeżeli porównanie zakończy się pomyślnie wykonywane są instrukcje skojarzone z tym rozgałęzieniem. W przypadku gdy wszystkie porównania zawiodą wykonywane jest domyślne wyrażenie po słowie kluczowym `default`. Jeżeli nie zostanie ono podane i wszystkie porównania zawiodą instrukcja `case` jest opuszczana.

Porównanie w instrukcji `case` kończy się sukcesem tylko wtedy, gdy każdy bit obu wyrażeń jest zgodny (uwzględniane są wartości `0`, `1`, `x` i `z`). Ponieważ porównania wykonywane są na poszczególnych bitach wyrażenia muszą mieć tą samą długość. Dotyczy to zarówno wyrażenia w nawiasie, jaki i wyrażeń w rozgałęzieniach. Powodem rozróżniania w porównaniach wartości `x` i `z` jest umożliwienie wykrycia tych wartości.

Przykład 45. Instrukcja `case` rozróżnia stany `x` i `z`

```
case (sig)
  1'bz: $display("wysoka impedancja");
  1'bx: $display("nieznana wartosc sygnalu");
  default: $display ("sygnal ma wartosc %b", sig);
endcase
```

8.5. Instrukcje *casex* i *casez*

Verilog dostarcza dwie dodatkowe odmiany konstrukcji `case` różniące się sposobem interpretacji wartości `x` i `z`. Pierwsza z nich to `casex`, w której wysoką impedancję (`z`) traktuje się jako wartość dowolną. Druga to `casez`, w której zarówno wartość `x` jak i `z` jest traktowana jako dowolna. Bity wyrażeń zawierające wartość dowolną nie są uwzględniane w porównaniach.

Przykład 46. W instrukcji `casez` wartość `z` jest pomijania w porównaniach

```
reg [7:0] a;

casez (a)
  8'b10??_???? : stan1;
  8'b01??_???? : stan2;
  8'b11??_???? : stan3;
  8'b00??_???? : stan3;
endcase
```

Przykład 47. Dekoder 1 z 10 zrealizowany za pomocą instrukcji `casex`

```
reg [9:0] in;
reg [3:0] out;

casex (in)
  10'b1????????? : out = 4'd0;
  10'b?1???????? : out = 4'd1;
  10'b??1??????? : out = 4'd2;
  10'b???1?????? : out = 4'd3;
  10'b????1????? : out = 4'd4;
  10'b?????1???? : out = 4'd5;
  10'b??????1??? : out = 4'd6;
  10'b???????1?? : out = 4'd7;
  10'b????????1? : out = 4'd8;
  10'b?????????1 : out = 4'd9;
  default: out = 4'bxxxx;
endcase
```

8.6. Pętle

W języku Verilog występują cztery rodzaje pętli: `while`, `for`, `repeat` i `forever`. Pozwalają one powtarzać instrukcje zero, raz i wiele razy.

8.6.1. Pętla *while*

Pętla while (ang. *while loop*) wykonuje się dopóki spełniony jest warunek. Warunek jest sprawdzany zawsze przy rozpoczęciu nowego powtórzenia. W szczególnym przypadku jeżeli nie będzie on spełniony już przy rozpoczęciu pętli nie będzie ona wykonana ani razu. Jeżeli w pętli występuje więcej niż jedna instrukcja, to instrukcje te należy zgrupować w bloku `begin-end`.

Przykład 48. Zliczanie jedynek w wektorze za pomocą pętli `while`

```
reg [7:0] temp;  // pomocniczy rejestr
i = 0;           // licznik jedynek
temp = a;

while (temp) begin
  if (temp[0])   // jeśli na najmłodszej pozycji 1 to zwiększ licznik
    i = i + 1;
  temp = temp >> 1;  // przesunięcie bitowe w prawo
end
```

8.6.2. Pętla *for*

Pętla for (ang. *for loop*) składa się z trzech części: warunku początkowego, warunku końcowego i przypisania zmieniającego wartość zmiennej.

W pętli nie jest konieczne definiowanie licznika pętli (warunek początkowy). Mimo, iż struktura pętli `for` jest bardziej złożona niż `while` jest ona mniej uniwersalna i nie zawsze może być stosowana zamiennie z pętlą `while`. Pętle `for` są używane przede wszystkim tam, gdzie dobrze jest określony początek i koniec pętli.

Przykład 49. Inicjalizacja wektora za pomocą pętli `for`

```
parameter MSB = 8;
reg [MSB-1:0] w;
integer i;

initial begin
  for (i=0; i<MSB; i=i+1) begin
    w[i] = 1'b0;
  end
end
```

8.6.3. Pętla *repeat*

Pętla repeat (ang. *repeat loop*) pozwala powtórzyć instrukcje określoną liczbę razy. Liczbę powtórzeń podawana jest w nawiasie po słowie kluczowym `repeat`. Może ona być stałą lub zmienną. Jednak jeżeli użyta będzie zmienna, to jej wartość zostanie pobrana przy starcie pętli i jakakolwiek zmiana wartości podczas wykonywania pętli nie ma wpływu na liczbę powtórzeń.

Przykład 50. Inicjalizacja wektora za pomocą pętli `repeat`

```
parameter size = 8;
reg [size-1:0] w;
integer i;

initial begin
  i = 0;
  repeat (size-1) begin
    w[i] = 1'b0;
    i = i + 1;
  end
end
```

8.6.4. Pętla *forever*

Pętla forever (ang. *forever loop*) wykonuje się przez cały czas trwania symulacji. Jest ona bardzo podobna do pętli `while`, z tym, że nie posiada warunku. Wyjście z pętli może nastąpić tylko przez użycie zadania `$finish` lub instrukcji `disable`.

Przykład 51. Generowanie sygnału zegarowego z użyciem pętli `forever`

```
reg clock;

initial begin
  clock = 1'b0;

  forever
    #10 clock = ~clock; // zmiana stanu zegara co 10 jednostek
end;
```

Pętla `forever` powinna być używana z instrukcjami kontroli czasowej. Jeśli w pętli nie będzie tych instrukcji symulator po rozpoczęciu wykonania pętli nigdy jej nie opuści zawieszając wykonywanie wszystkich pozostałych procesów.

8.7. Bloki sekwencyjne i równoległe

Za pomocą bloków sekwencyjnych (ang. *sequential block*) i równoległych (ang. *parallel block*) można grupować instrukcje. Taki blok jest później traktowany dokładnie tak samo jak jedna instrukcja. We wcześniejszych przykładach do grupowania instrukcji używany był blok sekwencyjny `begin-end`. Obok tego bloku, w Verilogu występuje jeszcze blok równoległy `fork-join`.

8.7.1. Blok sekwencyjny

Blok sekwencyjny rozpoczyna się od słowa kluczowego `begin`, a kończy się słowem kluczowym `end`. Instrukcje wewnątrz bloku są wykonywane sekwencyjne, jedna po drugiej, zgodnie z kolejnością w jakiej występują. Instrukcja w bloku sekwencyjnym może być wykonana dopiero po zakończeniu instrukcji poprzedzającej ją. Wyjątek od tej reguły stanowią przypisania nieblokujące z wewnętrznym przypisaniem opóźnienia.

8.7.2. Blok równoległy

Blok równoległy rozpoczyna się od słowa kluczowego `fork`, a kończy się słowem kluczowym `join`. Instrukcje wewnątrz bloku wykonywane są równolegle, a ich kolejność zależy od opóźnień w przypisaniach. Opóźnienia w bloku równoległym są obliczane względem czasu jego rozpoczęcia.

Przykład 52. Generowanie przebiegu czasowego za pomocą przypisań w bloku `begin-end`

```
reg [7:0] data;

begin
  #10 data = 8'h00;
  #20 data = 8'h1a;
  #40 data = 8'hb0;
  #80 data = 8'hff;
end
```

Przykład 53. Generowanie przebiegu czasowego za pomocą przypisań w bloku `fork-join`

```
reg [7:0] data;

fork
  #10 data = 8'h00;
  #30 data = 8'h1a;
  #70 data = 8'hb0;
  #150 data = 8'hff;
join
```

W blokach równoległych należy uważać na zjawisko wyścigu (ang. *race*), które może się pojawić wtedy, gdy jedna zmienna wystąpi w dwóch przypisaniach w tym samym czasie. Problem ten został pokazany w przykładzie 54. W bloku `fork-join` wszystkie wyrażenia występują bez opóźnień, przez co nie wiadomo w jakiej kolejności zostaną wykonane. Jeżeli najpierw wykonane zostaną przypisania `a = 1'b0` i `b = 1'b0`, wtedy zmienna `x` będzie miała wartość 1, a `y` 0. Natomiast jeżeli przypisania `x = a` i `y = b` wykonają się jako pierwsze, wtedy zmienne `x` i `y` przyjmą wartość nieznaną.

Przykład 54. Blok równoległy, w którym może dojść do wyścigu

```
reg a, b, x, y;

initial fork
  a = 1'b1;
  b = 1'b0;

  x = a;
  y = b;
join
```

8.7.3. Nazwy bloków

Zarówno bloki sekwencyjne jak i równoległe mogą mieć swoje nazwy. Nazwę bloku podaje się po słowie kluczowym `begin` lub `fork` poprzedzając ją dwukropkiem.

Za pomocą bloków z nazwą można:

Przykład 55. Blok z nazwą

```
begin : lokalny_blok
  integer a;  // deklaracja lokalnej zmiennej w bloku
  for (a=0; a<10; a=a+1)
    $display ("a = %d");
end
```

- deklarować lokalne rejestry wewnątrz bloku,
- tworzyć odniesienia do tych bloków w instrukcji `disable`,
- nazwane bloki są częścią hierarchii projektu. Zmienne w nazwanych blokach są dostępne za pomocą nazw hierarchicznych.

8.8. Instrukcja *disable*

Instrukcja `disable` umożliwia zakończenie działania aktualnie wykonywanej procedury lub bloku z uwzględnieniem struktury projektu. Instrukcja ta dostarcza mechanizmów pozwalających na przerwanie zadań przed wykonaniem wszystkich instrukcji, przerwania pętli oraz pominięcia fragmentu pętli i przejścia do następnego powtórzenia.

Każda forma instrukcji `disable` kończy działanie zadania lub bloku z nazwą. Symulacja jest dalej wykonywana od kolejnej instrukcji po bloku lub zadaniu. Wszystkie zadania uaktywnione wewnątrz bloku z nazwą lub wewnątrz zadania są również kończone.Jeżeli zadanie uaktywni wiele zadań wtedy wykonanie instrukcji `disable` kończy wszystkie zadania wywołane przez to zadanie nadrzędne. Jeżeli jedno zadanie zostanie wywołane wielokrotnie wtedy instrukcja `disable` kończy wszystkie te wywołania.

Przykład 56. Zakończenie pętli za pomocą instrukcji `disable`

```
reg [15:0] rejestr;
integer i;

initial begin
  rejestr = 16'b 0010_0000_0000_0000;
  i = 0;
  begin: blok
    while (i < 16) begin
        if (rejestr[i]) begin
          $display("Bit z wartością 1 na pozycji %d", i);
          disable blok;
        end
        i = i + 1;
    end
  end
end
```

9. Zadania i funkcje

Zadania (ang. *task*) i funkcje (ang. *function*) pozwalają na grupowanie często powtarzających się fragmentów kodu, a następnie ich użycie w wielu miejscach projektu. Za ich pomocą można podzielić złożone algorytmy na mniejsze części, które są bardziej czytelne i prostsze do weryfikacji. W tym rozdziale przedstawione zostaną różnice między zadaniami i funkcjami oraz sposób ich wywoływania.

Tabela 12. Różnice między zadaniami i funkcjami

Funkcje	**Zadania**
Mogą wywoływać inne funkcje ale nie mogą wywołać zadań.	Mogą uaktywnić zarówno funkcje jak i zadanie.
Wykonywane są w jednej jednostce czasu.	Ich wykonanie może trwać dowolny czas.
Nie mogą zawierać opóźnień, zdarzeń lub innych wyrażeń służących do kontroli czasowej.	Mogą zawierać opóźnienia, zdarzenia lub inne wyrażenia kontroli czasowej.
Muszą mieć przynajmniej jeden argument. Dozwolone są tylko argumenty typu `input`.	Nie muszą w ogóle posiadać argumentów. Dozwolone są argumenty typu `input`, `output` i `inout`
Zawsze zwracają pojedynczą wartość.	Nie zwracają wartości.

Zadania i funkcje muszą być deklarowane wewnątrz modułu i są lokalne w jego obrębie. Można w nich stosować wyłącznie wyrażenia behawioralne, i nie wolno deklarować sieci. Zadania stosuje się w przypadku kodu zawierającego opóźnienia, wyrażenia czasowe lub wiele argumentów wyjściowych. Natomiast funkcje znajdują zastosowanie przy opisie układów czysto kombinacyjnych. Funkcje w przeciwieństwie do zadań są one wykonywane natychmiast i zwracają tylko jedną wartość.

9.1. Zadania

Deklarację zadania rozpoczyna się od słowa kluczowego `task`, po którym występuje nazwa zadania, a kończy słowem kluczowym `endtask`. Argumenty wejściowe i wyjściowe deklaruje się za pomocą słów kluczowych `input`, `output`, `inout` podobnie jak to ma miejsce w deklaracji modułu.

Przykład 57. Składnia deklaracji zadania

```
task <nazwa_zadania> ;
     <deklaracje_argumentów>
     <instrukcje>
endtask
```

Po deklaracji zadanie wywołuje się podając jego nazwę oraz listę argumentów w nawiasach okrągłych. Poszczególne argumenty oddziela się od siebie przecinkami.

Jeżeli argument w deklaracji zadania jest zadeklarowany jako wejściowy to, na tej pozycji w wywołaniu zadania może pojawić się dowolne wyrażenie. Gdy argument zadeklarowany jest jako wyjściowy lub wejściowo-wyjściowy, wtedy wyrażenie jest ograniczone do takiego, które może się pojawić po lewej stronie przypisania.

Przykład 58 przedstawia prosty model procesora. W bloku `always` wywoływane jest zadanie, które zmienia stan procesora przy narastającym zboczu sygnału zegarowego.

Przykład 58. Prosty model procesora

```
module procesor;
  reg clock;
  reg [1:0] stan;

  parameter fetch = 0, decode = 1,
            execute = 2, write = 3;

always begin
  zmien_stan (fetch, stan);
  zmien_stan (decode, stan);
  zmien_stan (execute, stan);
  zmien_stan (write, stan);
end

task zmien_stan;  // zmien stan przy narastajacym zboczu zegara
input [1:0] nazwa_stanu;
output [1:0] nowy_stan;
begin
   @(posedge clock) nowy_stan = nazwa_stanu;

   case (nowy_stan)
     fetch: $display ("fetch");
     decode: $display ("decode");
     execute: $display ("execute");
     write: $display ("wrtie");
   endcase
end
endtask
endmodule
```

9.2. Funkcje

Funkcje deklaruje się za pomocą słowa kluczowego `function`, po którym podaje się typ zwracanego wyniku i nazwę funkcji, a kończy słowem kluczowym `endfunction`.

Przykład 59. Składnia deklaracji funkcji

```
function <zakres_lub_typ> <nazwa_funkcji>;
   <deklaracje argumentów typu input>;
   <deklaracje lokalnych zmiennych>;
   <instrukcje>;
endfunction
```

Funkcja może zwracać tylko jedną wartość typu: `real`, `integer`, `time`, `realtime` lub wektor o rozmiarze [m:n]. Jeśli nie zostanie podany typ zwracanego wyniku to domyślnie przyjmowany jest jednobitowy rejestr. Po nagłówku podawane są argumenty funkcji. W odróżnieniu od zadań funkcja może mieć tylko argumenty wejściowe i musi być podany co najmniej jeden taki argument.

Wcześniej zadeklarowaną funkcję można wywołać podając jej nazwę oraz listę argumentów w nawiasach okrągłych.

Przykład 60. Składnia wywołania funkcji

```
zmienna = nazwa_funkcji(argumenty);
```

W przykładzie 61 przedstawiono funkcję zamieniającą miejscami cztery najstarsze bity z czterema najmłodszymi. Funkcja pobiera jeden argument wejściowy, którym jest 8-bitowy rejestr i zwraca ten rejestr po zamianie jako argument wyjściowy.

Przykład 61. Funkcja zamieniająca bity w rejestrze

```
module test_funkcji;
reg [7:0] x, y;

initial begin
  x = 1'b0000_1111;
  y = zamiana (x);
end

function [7:0] zamiana;  // zamiana bitów w rejestrze
input [7:0] rejestr;     // z aaaa_bbbb na bbbb_aaaa
begin
  zamiana = {rejestr[3:0], rejestr[7:4]};
endfunction

endmodule;
```

Spis literatury

[1] Samir Planitkar, *Verilog HDL – A Guide to Digital and Synthesis*, SunSoft Press – A Prentice Hall, 1996

[2] *IEEE Standard HDL Based on the Verilog HDL*, IEEE, New York, 1996

[3] J. Mirkowski, Z. Skowroński, A. Biniszkiewicz: *Evita: Interactive Verilog Tutorial*; przyjęte do druku, ALDEC Inc., Henderson, Nevada, USA, 1998

[4] Giovanni De Micheli, *Synteza i optymalizacja układów cyfrowych*, Wydawnictwa Naukowo-Techniczne, Warszawa 1998

[5] D. Thomas, P. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publisher, Boston, MA, 1993

[6] Daniel C. Hyde, „Handbook on Verilog HDL", *http://www.eg.bucknell.edu/cs320*

[7] Stuart Sutherland, „On-line Verilog HDL Quick Reference Guide", *http://www.sutherland-hdl.com*

[8] „XC4000 Field Programmable Gate Arrays", *http://www.xilinx.com*

[9] *http://www.java.sun.com*

[10] J. Kalisz: *Podstawy elektroniki cyfrowej*, Wydawnictwa Komunikacji i Łączności, Warszawa, 1993

[11] Elizabeth Castro, *Po prostu HTML 4*, Wydawinictwo Helion, 2000

[12] Paweł Wimmer, „Kurs języka HTML - poradnik webmastera", *http://www-mag.com.pl/kurs/*

[13] „Polska Strona Ogonkowa", *http://www.agh.edu.pl/ogonki/*