SKRYPT DO PRZEDMIOTU

Języki modelowania i symulacji

autorzy:

dr inż. Bogdan Pankiewicz

dr inż. Marek Wójcikowski

Gdańsk, 2015

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt *„Przygotowanie i realizacja kierunku inżynieria biomedyczna – studia międzywydziałowe”*
współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Spis treści

1. Wstęp

Szybki rozwój technologii wytwarzania elementów i układów scalonych umożliwia tworzenie coraz bardziej złożonych urządzeń i systemów elektronicznych. Niestety powiększająca się złożoność stanowi istotny problem i wyzwanie w procesie projektowania urządzenia. Trudno jest pojedynczej osobie, czy grupie projektowej, nawet bardzo dobrze zaznajomionej z bieżącym stanem techniki, przewidzieć wszystkie możliwe zachowania się złożonego systemu elektronicznego. Wszelkie ewentualnie popełnione błędy powodują konieczność wykonania poprawek, co stanowi problem zarówno dla szybkiego wejścia produktu na rynek jak i powoduje zwiększenie kosztów ogólnych projektu. Z tego względu, od momentu pojawienia się pierwszych komputerów, powstały próby symulacji rzeczywistych układów elektronicznych w sztucznym wirtualnym środowisku obliczeniowym. Takie symulacje mają odpowiedzieć na pytanie jak zachowa się badany obwód elektryczny w rzeczywistości na podstawie jego opisu przy pomocy modeli matematycznych (obliczeniowych). Jednym z pierwszych symulatorów elektrycznych jest program SPICE (ang. *Simulation Program with Integrated Circuit Emphasis*), który po przeniesieniu na platformę komputerów typu PC został nazwany PSPICE. Ten program jest symulatorem operującym w dziedzinie elektrycznej i umożliwia symulację wszelkiego rodzaju obwodów elektrycznych składających się z elementów, których modele dostępne są w programie PSPICE. Potrzeba powstania takiego symulatora była szczególnie ważna przy projektowaniu układów scalonych, gdzie koszty uruchomienia produkcji są wysokie i dlatego zachodzi potrzeba aby pierwsza wersja projektu była pozbawiona wszelkich błędów i nie wymagała wprowadzania późniejszych poprawek. Symulatory elektryczne są do dzisiaj bardzo często stosowane przy projektowaniu układów zarówno analogowych jak i cyfrowych. Powstało wiele odmian takiego oprogramowania, z których najbardziej znane to: PSPICE, HSPICE, Spectre, APS, LTSpice i inne. Dodatkowo, ze względu na dużą liczbę elementów w dzisiejszych systemach cyfrowych, powstały symulatory logiczne, które kosztem zmniejszenia precyzji obliczeniowej umożliwiają symulację bardzo złożonych układów cyfrowych. W tych symulatorach nie występuje pojęcie wartości napięcia czy prądu, zamiast tego mamy sygnały w postaci stanu logicznego (np. wysoki, niski lub wysokiej impedancji), których propagacja od wejścia bloku cyfrowego do jego wyjścia zajmuje pewien czas zdefiniowany w modelu danego bloku. Takie uproszczenie umożliwia znaczące przyśpieszenie symulacji, ale tracona jest część informacji, która nie jest istotna w pierwszym przybliżeniu symulacji układu cyfrowego. Są to informacje takie jak np. pobór mocy, przesłuchy między sygnałowe, szumy, zakłócenia i wiele innych. Symulatory logiczne ewaluowały przez dłuższy okres i obecnie można przyjąć, że znaczącą większość z nich można użyć poprzez zastosowanie dwóch rodzajów języków opisu sprzętu cyfrowego typu HDL (ang. *Hardware Description Language*): Verilog i

VHDL (ang. *Very High Speed Integrated Circuit Hardware Description Language*). Języki te są bardzo rozpowszechnione w szczególności wśród projektantów układów wykorzystujących programowalne układy cyfrowe typu CPLD (ang. *Complex Programmable Logic Device*) lub FPGA (ang. *Field Programmable Gate Array*), gdzie oprócz symulacji można również wykonać projekt bloku cyfrowego. Ostatnio, w wielu systemach znajdują zastosowanie układy mieszane, które pracują zarówno z sygnałami cyfrowymi jak i analogowymi. Dla takiego rodzaju układów niestety nie możemy wykorzystać typowych języków HDL gdyż w ten sposób utracona by została informacja analogowa w postaci napięć i prądów występujących w obwodzie. Takie układy możemy symulować przy pomocy symulatorów elektrycznych, jednakże wymagany jest wówczas bardzo długi czas na obliczenia, gdyż część cyfrowa, zazwyczaj stanowiąca znaczącą większość systemu mieszanego, symulowana jest bez uproszczeń. Z tego względu pojawiły się języki opisu systemów mieszanych takie jak np. Verilog-A (ang. *Verilog - Analog*) czy VHDL-AMS (ang. *VHDL - Analog Mixed Signal*), które do opisu wykorzystują jednocześnie dziedziny elektryczną i logiczną.

W niniejszym skrypcie przedstawiono podstawy wykorzystania symulatora PSPICE, języka Verilog oraz języka VHDL i składa się on z trzech głównych części odpowiadających poszczególnym zagadnieniom. Szczególny nacisk położony został na zasady wykorzystania i nabycie umiejętności praktycznych dotyczących symulacji układów elektronicznych. Zainteresowanych metodami numerycznymi wykorzystywanymi w symulatorach zachęcamy do skorzystania z oddzielnej, specjalistycznej literatury [1] - [10].

2. Symulacje układów elektronicznych z wykorzystaniem PSPICE

Niniejszy rozdział poświęcony jest symulatorowi PSPICE. Zawarto tu podstawowe informacje niezbędne do wykonania opisu obwodu elektrycznego oraz wykonania symulacji elektrycznych. Skrypt ten nie stanowi pełnego opisu możliwości programu lecz zawiera informacje niezbędne i wystarczające w znaczącej większości przypadków dla pełnego opisu i symulacji typowych obwodów elektronicznych.

2.1. Informacje wstępne

Nazwa SPICE stanowi angielski skrót od słów *Simulation Program with Integrated Circuit Emphasis*, co oznacza program do symulacji ze szczególnym uwzględnieniem układów scalonych. Układy scalone stanowiły i stanowią w dalszym ciągu jedną z dziedzin, dla których symulacja komputerowa jest bardzo istotna z punktu widzenia jakości powstałego produktu jak i szybkości wprowadzenia go na rynek. Ze względu na duże koszty przygotowania masek produkcyjnych, poprawki w układach scalonych są zarówno czasochłonne jak i kosztowne. Drobiazgowa symulacja układu przed jego

produkcją jest zatem jednym z kluczowych etapów przygotowania projektu. Pierwsza wersja programu SPICE powstała na kalifornijskim uniwersytecie w Berkeley, USA w roku 1973. Powstanie programu było sponsorowane przez amerykański departament obrony. Program działał pod systemem operacyjnym Unix. Z czasem program stał się oprogramowaniem typu *Public Domain* a na rynku pojawiły się także symulatory komercyjne takie jak Eldo, Spectre, HSpice, Saber czy tez wiele innych. Z czasem też firma MicroSim przeniosła program SPICE na komputery klasy PC pracujące najpierw pod kontrolą systemu DOS a później Windows a nazwa programu została uzupełniona o literę P dając w efekcie PSPICE. W kolejnych latach PSPICE został wykupiony najpierw przez firmę OrCad a następnie przez firmę Cadence. W laboratorium „Układów scalonych i programowalnych” mieszczącym się w Katedrze Systemów Mikroelektronicznych na Wydziale Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej użytkowana jest wersja PSPICE firmowana przez Cadence. W sieci Internet dostępne są darmowe i demonstracyjne pakiety oprogramowania typu SPICE/PSPICE. Firma Linear Technology udostępnia oprogramowanie o nazwie LT Spice, można też ściągnąć demonstracyjną wersję PSPICE ze stron firmy CADENCE i pełną wersję oryginalnego SPICE 3fg z uniwersytetu Berkeley (pod system Linux).

W niniejszych materiach zastosowana zostanie ogólna składnia opisu, której zasady stosowania są następujące:

- zawartość umieszczona w nawiasach trójkątnych **< >** jest obowiązkowa, same nawiasy należy jednak w pliku opisowym PSPICE pominąć,
- zawartość umieszczona w nawiasach kwadratowych **[ ]** jest opcjonalna, same nawiasy należy w pliku opisowym PSPICE pominąć,
- symbol pionowej kreski **|** pomiędzy danymi wpisami oznacza, że może wystąpić tylko jeden element wpisów rozdzielonych pionową kreską.

2.2. Zasady ogólne składni PSPICE

Oryginalne oprogramowanie PSPICE bazuje na tekstowej liście połączeniowej uzupełnionej o polecenia symulatora dotyczące zleconych do wykonania zadań symulacji. Obecnie, często można spotkać nakładki umożliwiające pracę na schematach układów w postaci graficznej. W takim wypadku przed wykonaniem symulacji, na podstawie graficznego schematu tworzona jest (automatycznie przez oprogramowanie do rysowania schematów) lista połączeniowa uzupełniana o niezbędne do wykonania symulacje i przekazywania właściwemu programowi symulującemu. W niniejszym skrypcie, w celu zapewnienia uniwersalności, stosowana będzie tekstowa lista połączeniowa, która jest zgodna formatem z większością symulatorów typu PSPICE i w związku z tym może być stosowana w różnych środowiskach symulacyjnych. Taka lista połączeniowa w większości przypadków może być stosowana

również w symulatorach z innych rodzin (np. Spectre), gdyż typowe jest wyposażanie ich w parsery zdolne odczytać format SPICE. Plikiem wejściowym w symulatorze PSPICE, który najpierw podlega analizie składniowej a następnie procesowi symulacji, jest plik tekstowy spełniający wymagania składniowe. Poniżej przedstawiony jest prosty przykład takiego pliku.

```
Układ 1
R1 1 0 10k
C1 1 0 1f
* to jest komentarz
.op
.end
```

Podstawowe zasady składni PSPICE są następujące:

1) Typowe rozszerzenie nazwy pliku tekstowego zawierającego netlistę PSPICE to „*.cir”. Czasami stosowane są również inne rozszerzenia takie jak np.: „*.sp” lub „*.spice”. Oprogramowanie stosowane w ćwiczeniach laboratoryjnych do niniejszego przedmiotu wymaga aby rozszerzenie nazwy pliku było „*.cir”.
2) Należy unikać polskich znaków diakrytycznych i spacji zarówno w nazwach plików jak i w opisie wewnątrz pliku PSICE.
3) Składnia PSPICE nie jest czuła na wielkość liter.
4) Polecenia symulacyjne jak i lista połączeniowa są podawane w postaci kolejnych linii pliku tekstowego.
5) Jedna linia w pliku odpowiada jednemu poleceniu lub wstawieniu jednego elementu elektronicznego.
6) Maksymalna długość linii tekstu ograniczona jest do 80 znaków. Jeśli 80 znaków umieszczonych w pojedynczej linii nie wystarcza do opisu, wówczas można przedłużyć linię poprzez umieszczenie znaku „+” w kolejnej linii opisu, co skutkuje przedłużeniem linii poprzedniej, jak np. w poniższym fragmencie kodu:

```
...
R1 1 0
+ 10k
...
```

jest równoważne:

```
...
R1 1 0 10k
...
```

7) Pierwsza linia pliku zawiera nazwę badanego układu. Linia ta może zawierać dowolny opis i nie podlega dodatkowym ograniczeniom oprócz ograniczenia długości do maksimum 80 znaków.

8) Ostatnia linia pliku powinna zawierać polecenie:

```
.end
```

co oznacza zakończenie opisywanego układu. W przypadku gdy za powyższą linią będzie kolejny fragment tekstu, traktowany jest on przez symulator jako kolejny układ do symulacji.

9) Oprócz linii pierwszej (tytuł badanego obwodu) i ostatniej (`.end`) kolejność pozostałych linii nie ma znaczenia.
10) Puste linie są pomijane.
11) Linie, które rozpoczynają się literą są opisem wstawionego elementu lub podukładu. Np. R oznacza wstawienie rezystora, C – kondensatora i.t.d.
12) Linie, które rozpoczynają się znakiem kropki „.” stanowią polecenia symulatora.
13) Linie rozpoczynające się gwiazdką „*” są komentarzem. Dodatkowo w liniach, w których występuje średnik „;”, tekst począwszy od pozycji średnika do końca linii też jest traktowany jako komentarz.
14) Nadawane elementom i węzłom nazwy mogą składać się z cyfr, liter, znaków podkreślenia, myślników, znaków dolara i symbolu procenta. W tym miejscu należy dodać, że w pierwszych wersjach symulatora, nazwy elementów i węzłów mogły składać się wyłącznie z cyfr. Obecnie zostało to rozszerzone, ale dodatkowo wymaga się, aby w miejscach w których nazwa węzła rozpoczynająca się od litery mogłaby spowodować niejasność czy chodzi o węzeł czy element, takie nazwy były dodatkowo objęte nawiasem kwadratowym.
15) Elementami oddzielającymi poszczególne parametry są spacje, tabulatory i przecinki.

Podsumowując powyższe informacje można stwierdzić, że w tekstowym pliku PSPICE mamy dwie główne części: listę połączeniową, którą stanowią linie rozpoczynające się literą reprezentującą wstawienie danego rodzaju komponentu oraz linie poleceń rozpoczynające się od znaku kropki. Sama lista połączeniowa również obwarowana jest pewnymi wymaganiami. Zanim przedstawione zostaną szczegóły opisu listy, poniżej zamieszczone są ograniczenia, które musi ona spełnić:

1) Wszelkie napięcia w PSPICE liczone są względem napięcia węzła masy. Węzeł masy ma stałą nazwę „0” (zero) i musi występować w badanym układzie.
2) Węzłom można nadać nazwę będącą numerem lub ciągiem znaków składających się z liter, cyfr i znaków procenta, dolara, „_” oraz „-”.
3) Każdy z węzłów musi mieć przynajmniej dwa połączenia do różnych elementów, w przeciwnym razie dany węzeł uznany zostanie jako węzeł typu „*floating*” i nie będzie możliwe wykonanie symulacji (zgłoszony zostanie błąd symulacji). Niektóre symulatory (np. LT SPICE) automatycznie usuwają węzły „*floating*” i generują jedynie ostrzeżenie umożliwiając

symulację, należy jednak w takim przypadku bardzo uważnie prześledzić listę połączeniową i stwierdzić czy takie działanie symulatora jest pożądane.

4) Węzły, do których są dwa połączenia ale wykonane wyłącznie poprzez kondensatory lub bramki tranzystorów MOS także są uważane jako „*floating*”.
5) Niedozwolone jest równoległe łączenie idealnych źródeł napięciowych.
6) Niedozwolone jest szeregowe łączenie prądowych źródeł idealnych.

2.3. Jednostki i zasady zapisu wartości

Większość używanych w PSPICE jednostek to jednostki zgodne z układem SI. Temperatura jest jednym z wyjątków i jej jednostką jest stopień Celsjusza. Dopuszczalne jest stosowanie przyrostków wielkości zgodnie z tab. 1.

Tab. 1. Przyrostki wielkości stosowane w PSPICE.

Przyrostek	f	p	n	u	m	k	meg	g	t	mil
Przyrostek	F	P	N	U	M	K	MEG	G	T	MIL
Mnożnik wielkości	10^{-15}	10^{-12}	10^{-9}	10^{-6}	10^{-3}	10^{3}	10^{6}	10^{9}	10^{12}	$25{,}4 \cdot 10^{-6}$

Należy zwrócić uwagę, że ze względu brak rozróżniana wielkości liter, zarówno „m” jak i „M” oznacza ten sam mnożnik równy 10^{-3}. Mnożnik równy 10^{6} uzyskuje się poprzez ciąg liter „meg” lub „MEG”. Nietypowym mnożnikiem jest „mil”, który stanowi długość równą 1/1000 cala wyrażoną w metrach. Wartości można zapisywać w jednym z dostępnych formatów omówionych poniżej:

1) Format zwykły, np.: 123.23 co oznacza liczbę dziesiętną z częścią ułamkową. Należy zauważyć, że znakiem oddzielającym część całkowitą od ułamkowej jest w PSPICE kropka „.”
2) Format naukowy, np.: 1.2323e5 co jest równoważne wartości $1.2323 \cdot 10^{5}$.
3) Z użyciem przyrostka wartości, np. 1.2323m co jest równoważne $1.2323 \cdot 10^{3}$.
4) Z dodatkowym dodaniem dowolnego symbolu lub ciągu nie będącym przyrostkiem wartości. Taki ciąg znaków jest pomijany przez symulator ale może ułatwić interpretację dla człowieka, np. 3 V lub 1.2323 mV. W przypadku dodawania jednostek należy specjalną uwagę zwrócić na literę F gdyż może ona być interpretowana zarówno jako mnożnik wartości jak i jako symbol jednostki pojemności.
5) Wartości można także podawać w sposób pośredni poprzez wyrażenie, które przed wykorzystaniem w symulatorze zostanie najpierw wyliczone a następnie użyte. Wartości w postaci wyrażeń z wykorzystaniem operatorów jak i funkcji (wbudowanych i własnych) umieszcza się w nawiasach klamrowych. W SPICE wbudowane są podstawowe funkcje typu

sin, cos, tan, pierwiastek itp., można także definiować własne, specyficzne funkcje. Więcej informacji na temat funkcji i operatorów matematycznych umieszczone jest w dalszych rozdziałach. Poniżej przedstawiony jest przykład deklaracji rezystora, którego wartość wyliczana jest na podstawie wyrażenia umieszczonego w nawiasach klamrowych:

```
R1 1 0 {10k*2/23}
```

6) Wartości wyznaczane za pomocą wyrażeń mogą wykorzystywać także parametry definiowane przez użytkownika. Poniższy kod przedstawia przykład deklaracji parametru o nazwie `mult` a następnie wykorzystane tego parametru do wyznaczenia wartości rezystancji rezystora `R1`.

```
.param mult = 10
R1 1 0 {mult*2*sin(mult)*1k}
```

2.4. Temperatura w PSPICE

Temperatura w programie PSPICE, jako jeden z nielicznych parametrów, nie jest wyrażona w jednostkach SI lecz w stopniach Celsjusza [°C]. Zdefiniowane są dwa ogólne parametry związane z temperaturą: temperatura nominalna i temperatura bieżąca symulacji. Temperatura nominalna jest temperaturą, przy której zmierzone zostały parametry elementów elektronicznych wchodzących w skład badanego obwodu. Wszelkie zmiany wartości bieżących parametrów wynikające ze zmian temperatury liczone są uwzględniając jako podstawę różnicę pomiędzy temperaturą bieżącą a temperaturą nominalną elementów. Wartość temperatury nominalnej definiuje się poprzez polecenie `.options` jak w przykładzie poniżej:

```
.options TNOM = 37
```

W przypadku braku deklaracji wartości nominalnej temperatury przyjęta jest wartość domyślna równa 27 °C. Domyślna wartość bieżącej temperatury symulacji również jest równa 27 °C. Zmianę tej wartości można dokonać na dwa sposoby. Pierwszy z nich to wykorzystanie polecenia jak w przykładach poniżej:

```
.temp = 50
.temp   50
.temp -50 0 30 125
```

Dwa pierwsze przykłady ustawiają bieżącą temperaturę symulacji na 50 °C. Ostatni z przykładów powoduje wykonanie 4 zestawów analiz, najpierw przy temperaturze bieżącej równej -50 °C, później kolejno dla temperatur 0 °C, 30 °C i 125 °C. Inną możliwością zmiany temperatury jest modyfikacja parametru `TEMP` (co jest równoznaczne z powyższymi poleceniami), jak np. w poniższych przykładach, równoznacznych z poprzednimi przykładami:

```
.param TEMP = 50
.step param TEMP list -50 0 30 125
```

Powyżej opisane parametry **TNOM** oraz **TEMP** mają zastosowanie do wszystkich elementów badanego układu z wyjątkiem tych elementów, które mają zdefiniowany model i w tym modelu występuje przynajmniej jeden z parametrów: **T_ABS**, **T_REL_GLOBAL**, **T_REL_LOCAL**, **T_MEASURED**. W takim przypadku parametry globalne **TNOM** i **TEMP** mogą zostać zmodyfikowane zgodnie z poniższymi zasadami:

- dla elementu dla którego w modelu podano parametr **T_ABS** temperatura bieżąca jest równa temu parametrowi,
- dla elementu dla którego w modelu podano parametr **T_REL_GLOBAL** temperatura bieżąca jest równa **TEMP** + **T_REL_GLOBAL**,
- dla elementu, dla którego model jest typu AKO i podano parametr **T_REL_LOCAL**, temperatura bieżąca jest równa **TEMP** z modelu odniesienia + **T_REL_ LOCAL**,
- dla elementu, dla którego w modelu podano parametr **T_MEASURED**, zmieniona jest wartość parametru **TNOM** na wartość równą **T_MEASURED**.

Szczegóły korzystania z modeli elementów podane zostały w następnych podrozdziałach.

2.5. Modele elementów

Większość elementów elektronicznych używanych w PSPICE może mieć model. W przypadku użycia modelu wartości parametrów danego elementu są odczytywane z tego modelu. Niektóre elementy, takie jak np. rezystor czy kondensator mogą być wstawione na 2 sposoby: z użyciem modelu lub bez jego użycia. Wstawienie elementu bez użycia modelu jest prostsze i łatwiejsze do wykonania, jednak w takim przypadku niektóre zaawansowane możliwości symulatora nie mogą być wykorzystane. Jest również grupa elementów (źródła napięciowe i prądowe), które występują wyłącznie bez modelu. Składnia deklaracji modelu danego elementu jest następująca:

```
.MODEL <nazwa_modelu> [AKO: <nazwa_modelu_odniesienia>]
+ <typ_modelu>
+ ([<nazwa_parametru> = <wartość> [specyfikacja_tolerancji]]
+ [T_MEASURED=<wartość>] [[T_ABS=<wartość>] |
+ [T_REL_GLOBAL=<wartość>] | [T_REL_LOCAL=<wartość>]])
```

gdzie:

nazwa_modelu - jest nazwą modelu nadawaną przez użytkownika lub dostawcę elementu, w przypadku nazw modeli dodatkową zasadą jest wymaganie aby zaczynała się ona od litery,

AKO: nazwa_modelu_odniesienia - nazwa modelu odniesienia, parametr opcjonalny, jednak jeśli zostanie użyty oznacza to, że wartości wszystkich parametrów modelu zostaną najpierw pobrane z modelu o nazwie **nazwa_modelu_odniesienia** a następnie te parametry, które zostaną nadpisane będą miały zmienione wartości, opcja **AKO:** służy do definiowania identycznych modeli z modelami

odniesienia, w których wykonano tylko niewielkie modyfikacje (zmienione zostają tylko niektóre parametry lub ustalana jest inna temperatura nominalna lub bieżąca),

typ_modelu - typ modelu, jedna z wartości z listy dostępnych modeli, modele dostępne to m.in.: **CAP**, **CORE**, **D**, **GASFET**, **IND**, **ISWITCH**, **LPNP**, **NIGBT**, **NJF**, **NMOS**, **NPN**, **PJF**, **PMOS**, **PNP**, **RES**, **TRN**, **VSWITCH**, typ modelu automatycznie określa jakie parametry mogą być wyszczególnione w następującej dalej liście parametrów jak również automatycznie definiuje model matematyczny elementu wykorzystywany do opisu danego elementu w czasie symulacji,

T_MEASURED, **T_ABS**, **T_REL_GLOBAL**, **T_REL_LOCAL** – parametry te zostały omówione w punkcie poprzednim i służą do modyfikacji wartości temperatury nominalnej i bieżącej stosowanych tylko w odniesieniu do elementów opisanych danym modelem,

nazwa_parametru = wartość [specyfikacja_tolerancji] – lista parametrów danego modelu wraz z wartościami i opcjonalnymi specyfikacjami tolerancji tego parametru, ponieważ każdy z parametrów ma wartości domyślne można pozostawić pustą listę a symulacja przebiegnie bez błędów, należy jednak w takim przypadku pamiętać, że do obliczeń zostały użyte wartości domyślne dla danych parametrów, specyfikacje tolerancji zostaną dokładniej omówione w rozdziale dotyczącym analizy Monte Carlo.

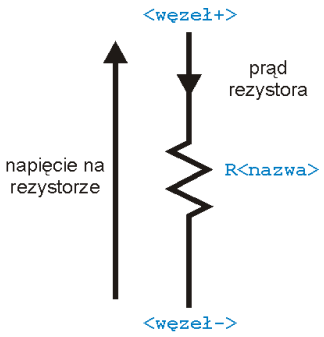
2.6. Elementy z dwoma wyprowadzeniami

Podstawowe elementy z dwoma wyprowadzeniami takie jak np. rezystor, kondensator czy cewka mogą być wstawione z modelem lub bez dodatkowego definiowania ich modelu. W przypadku użycia bez modelu, bezpośrednio w linii wstawiającej dany element podawane są parametry danego elementu. Wstawienie elementu bez podawania modelu jest proste, natomiast nie daje możliwości zaawansowanych taki jak np: symulacje statystyczne z uwzględnieniem tolerancji elementów czy wprowadzenie nieliniowości charakterystyk. Zasada wstawiania elementów R, L, C, przedstawiona jest na przykładzie składni ogólnej poniżej:

```
<R|L|C><nazwa> <węzeł+> <węzeł-> [model] <wartość>
+[wartości parametrów dodatkowych]
```

Pierwsza litera w linii oznacza typ wstawianego elementu zgodnie z listą: R - rezystor, L - cewka, C - kondensator. Bezpośrednio po literze oznaczającej typ elementu następuje nazwa tego elementu (nazwa wstawienia) mogąca składać się z cyfr, liter, znaków podkreślenia, myślników, symbolu dolara i symbolu procenta. Zasady nadawania nazw dotyczą także nazw nadawanych węzłom połączeniowym. Kolejne parametry to nazwa dodatniego węzła połączeniowego i ujemnego węzła połączeniowego. Bardzo ważną zasadą w PSPICE jest to, że napięcie na elementach mierzy się jako napięcie pomiędzy **węzeł+** a **węzeł-**. Dodatkowo jako prąd dodatni uważa się prąd wpływający z zewnątrz do danego

elementu poprzez **węzeł+**. Jeśli używamy modelu, to stanowi on kolejny parametr wstawianego elementu. Następnie podawana jest wartość parametru głównego: dla rezystora jest to rezystancja, dla cewki indukcyjność a dla kondensatora pojemność. Używane jednostki są zgodne z SI. Na końcu linii można podać dodatkowe parametry – zostaną one omówione szczegółowo dla kolejnych elementów.



Rys. 1. Elementy z dwoma wyprowadzeniami – zasada pomiaru napięcia i prądu oraz nazewnictwo węzłów na przykładzie rezystora.

2.6.1. Rezystor

Ogólna składnia wstawienia rezystora jest przedstawiona poniżej:

```
R<nazwa> <węzeł+> <węzeł-> [model] <wartość>
+[TC1=<TC1> [TC2=<TC2>]]
```

Jak widać z powyższej składni, użycie modelu i współczynników **TC1** i **TC2** jest opcjonalne. **TC1** i **TC2** są wartościami temperaturowych współczynników rezystancji i domyślne ich wartości wynoszą 0. Poniższe przykłady wstawienia rezystora wyjaśniają zasady składni w szczegółach.

```
Rload out 0 10k
```

Rezystor został wstawiony pomiędzy węzły **out** oraz **0** (dla przypomnienia 0 jest węzłem masy w PSPICE) a jego rezystancja wynosi 10 kΩ.

```
R123 wypr12 12 my_res 22k
```

Rezystor wstawiony pomiędzy węzły **wypr12** a **12** o modelu zdefiniowanym przez **my_res** i wartości inicjalnej rezystancji równej 22 kΩ. W przypadku gdy używany jest model, musi on być zdefiniowany. Poniżej przedstawiony jest przykład takiej definicji:

```
.model my_res RES(R=3 TC1=0.01 TC2=0.1)
```

Dla powyższego modelu i wstawienia rezystora wartość rzeczywistą rezystancji rezystora **R123** można określić wzorem:

$$r = 22k \cdot R \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2) \qquad (1)$$

gdzie: **TEMP** – bieżąca wartość temperatury symulacji dla danego elementu (domyślnie 27 °C), **TNOM** – temperatura nominalna dla danego elementu (temperatura przy której mierzono parametry modelu danego elementu, domyślnie 27 °C) **TC1**, **TC2** – liniowy i kwadratowy współczynnik temperaturowy rezystancji. Wzór (1) obowiązuje również w przypadku podania współczynników temperaturowych bezpośrednio w linii wstawianego elementu (bez użycia modelu).

Należy dodać, że poprawna składniowo jest również następująca definicja modelu rezystora:

```
.model my_res RES
```

co oznacza, że wszystkie wartości używane w modelu rezystora mają wartości domyślne.

W modelu rezystora można stosować parametry podane w tab. 2.

Tab. 2. Parametry modelu rezystora.

Parametr modelu	Opis	Jednostka	Wartość domyślna
R	mnożnik rezystancji		1
TC1	współczynnik temperaturowy liniowy	1/°C	0
TC2	współczynnik temperaturowy kwadratowy	$1/°C^2$	0
TCE	współczynnik temperaturowy ekspotencjalny	%/°C	0

Równania modelu rezystora określają wartość rezystancji wykorzystywanej w symulatorze PSPICE do obliczeń rozpływu prądów. Jeśli rezystor ma zdefiniowany model i w tym modelu podano wartość współczynnika **TCE**, wówczas jego rezystancja jest równa:

$$r = <\text{wartość}> \cdot R \cdot 1.01^{TCE(TEMP-TNOM)} \tag{2}$$

Jeśli rezystor ma zdefiniowany model i parametr **TCE** nie jest podany w modelu, wówczas jego rezystancja jest równa:

$$r = <\text{wartość}> \cdot R \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2) \tag{3}$$

Jeśli rezystor nie ma zdefiniowanego modelu, wówczas jego rezystancja jest równa:

$$r = <\text{wartość}> \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2) \tag{4}$$

W przypadku gdy wykonywana jest analiza szumowa, rezystor traktowany jest jako element generujący szum biały o gęstości widmowej mocy szumów równej:

$$i_{noise}^2 = 4kT / r \tag{5}$$

gdzie: `k` – stała Boltzmana, `T` – temperatura bezwzględna, wyjątkowo w tym przypadku wyrażona w Kelwinach.

2.6.2. Kondensator

Ogólna składnia wstawienia kondensatora przedstawiona jest poniżej:

```
C<nazwa> <węzeł+> <węzeł-> [model] <wartość>
+[IC=<wartość_początkowa>]
```

Jak widać, użycie modelu i wartości początkowej jest opcjonalne. Wartość początkowa, o ile jest użyta, jest równoważna poleceniu ustawienia punktu pracy `.ic` jak w kodzie poniżej:

```
.ic V(<węzeł+,węzeł-) <wartość_początkowa>
```

Polecenie `.ic` jest omówione dokładniej w jednym z kolejnych rozdziałów. W powyższym przypadku oznacza to wstawienie niezależnego źródła napięciowego o wartości napięcia równego parametrowi `wartość_początkowa` i rezystancji wewnętrznej równej 0.0002 Ω równolegle do węzłów, do których dołączony jest kondensator. To dołączenie następuje tylko na czas obliczenia punktu pracy, w innych analizach źródło napięciowe jest usuwane. W przypadku, gdy nie użyto modelu, pojemność kondensatora wynosi `wartość` faradów. Jeśli został użyty model wówczas pojemność jest wyliczana zgodnie z równaniem modelu jako równa:

$$c =< \text{wartość} > \cdot C \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2)(1 + VC1 \cdot V + VC2 \cdot V^2) \quad (6)$$

gdzie: `V` – napięcie na okładkach kondensatora, `TEMP` – temperatura bieżąca, `TNOM` – temperatura nominalna, a pozostałe parametry wyjaśnione są w tab. 3.

Tab. 3. Parametry modelu kondensatora.

Parametr modelu	Opis	Jednostka	Wartość domyślna
`C`	mnożnik pojemności		1
`TC1`	współczynnik temperaturowy liniowy	1/°C	0
`TC2`	współczynnik temperaturowy kwadratowy	$1/°C^2$	0
`VC1`	współczynnik napięciowy liniowy	1/V	0
`VC2`	współczynnik napięciowy kwadratowy	$1/V^2$	0

Kondensator jest elementem bezszumnym. Przykłady wstawienia kondensatora:

```
C12 out 0 10pF
```

Kondensator został wstawiony pomiędzy węzły `out` oraz `0` a jego pojemność wynosi 10pF.

```
Ccomp a1 a2 cap_1 22F IC=1.3V
```

Kondensator wstawiony pomiędzy węzły **a1** i **a2** o modelu zdefiniowanym przez **cap_1** i wartości inicjalnej pojemności równej 22 fF. Wartość początkowa napięcia na kondensatorze dla analizy obliczającej punkt pracy jest równa 1,3 V. W przypadku, gdy używany jest model, musi on być zdefiniowany. Poniżej przedstawiony jest przykład takiej definicji:

```
.model cap_1 CAP(C=2 TC1=0.01 TC2=0.1)
```

Dla powyższego modelu kondensator **Ccomp** z przykładu miałby pojemność równą:

$$c = 22\,fF \cdot C \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2) \quad (7)$$

2.6.3. Cewka indukcyjna

Ogólna składnia wstawienia cewki indukcyjnej przedstawiona jest poniżej:

```
L<nazwa> <węzeł+> <węzeł-> [model] <wartość>
+[IC=<wartość_początkowa>]
```

Jak widać, użycie modelu i wartości początkowej jest opcjonalne. Wartość początkowa, jeśli jest użyta, jest równoważna poleceniu ustawienia punktu pracy **.ic** jak w kodzie poniżej:

```
.ic I(L<nazwa>) <wartość_początkowa>
```

Polecenie **.ic** jest omówione dokładniej w jednym z kolejnych rozdziałów. W powyższym przypadku oznacza to, że dla obliczenia punktu pracy zakłada się, że przez cewkę płynie prąd o wartości równej parametrowi **wartość_początkowa**. Ten prąd używany jest tylko na czas obliczenia punktu pracy. W przypadku, gdy nie użyto modelu, wartość indukcyjności cewki wynosi **wartość** henrów. Jeśli został użyty model, wówczas indukcyjność jest wyliczana zgodnie z równaniem modelu jako:

$$l = <\text{wartość}> \cdot L \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2)(1 + IL1 \cdot I + IL2 \cdot I^2) \quad (8)$$

gdzie: **I** – prąd płynący przez cewkę, **TEMP** – temperatura bieżąca, **TNOM** – temperatura nominalna a pozostałe parametry wyjaśnione są w tab. 4.

Tab. 4. Parametry modelu cewki.

Parametr modelu	Opis	Jednostka	Wartość domyślna
L	mnożnik indukcyjności		1
TC1	współczynnik temperaturowy liniowy	1/°C	0
TC2	współczynnik temperaturowy kwadratowy	$1/°C^2$	0
IL1	współczynnik prądowy liniowy	1/A	0
IL2	współczynnik prądowy kwadratowy	$1/A^2$	0

Cewka jest elementem bezszumnym. Przykłady wstawienia cewki:

```
La n1 n2 {10uH/2}
```

W kodzie powyżej cewka została wstawiona pomiędzy węzły **n1** oraz **n2** a jej indukcyjność po wyliczeniu wyrażenia wynosi 5 μH.

```
Lind a1 a2 ind_1 18m IC=.3
```

Cewka włączona pomiędzy węzły **a1** i **a2** o modelu zdefiniowanym przez **ind_1** i wartości inicjalnej indukcyjności równej 18 mH. Wartość prądu cewki wykorzystywana do wyznaczenia punktu pracy jest równa 0.3 A. W przypadku, gdy używany jest model, musi on być zdefiniowany. Poniżej przedstawiony jest przykład takiej definicji:

```
.model ind_1 IND(L=2 TC1=0.01 TC2=0.1)
```

Dla powyższego modelu cewka **Lind** z przykładu powyżej miałaby indukcyjność równą:

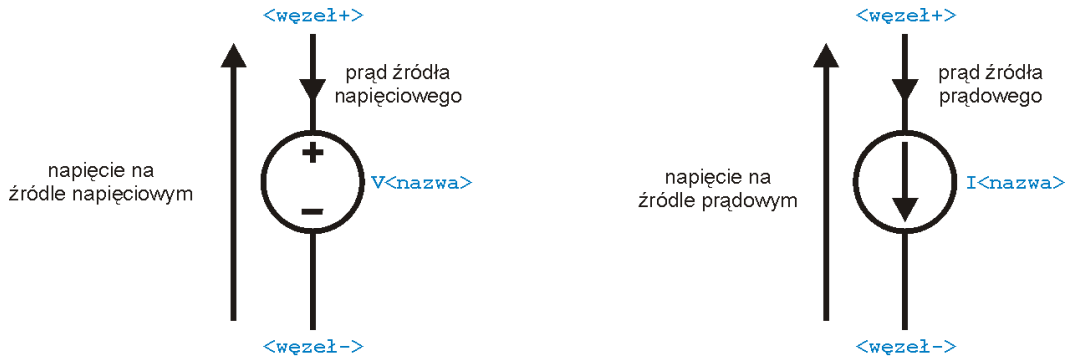
$$c = 18mH \cdot L \cdot (1 + TC1(TEMP - TNOM) + TC2(TEMP - TNOM)^2) \tag{9}$$

2.6.4. Niezależne źródło napięciowe i prądowe

Ogólna składnia wstawienia niezależnego źródła napięciowego przedstawiona jest poniżej:

```
V<nazwa> <węzeł+> <węzeł-> [[DC] wartość]
+[AC <wartość_amplitudy> [wartość_fazy]]
+[specyfikacja_transient]
```

Wstawienie źródła prądowego jest wykonywane identycznie jak napięciowego z tą różnicą, że litera **V** na początku linii zastępowana jest literą **I** symbolizującą źródło prądowe. Z powyższej definicji wynika, że jedynie podanie nazwy oraz węzłów połączeniowych jest obowiązkowe, sama wartość napięcia/prądu może nie być podana, wówczas domyślnie przyjmowana jest wartość równa zero. Zgodnie z zasadą stosowaną w PSPICE za prąd dodatni uważa się prąd wpływający do punktu **węzeł+** natomiast dodatnie napięcie jest napięciem mierzonym pomiędzy **węzeł+** a **węzeł-**. Zgodnie z tą definicją, iloczyn napięcia i prądu na źródłach napięciowym i prądowym są ujemne jeśli dane źródło generuje energię a dodatnie jeśli ją rozprasza. Definicja ta powoduje również kłopoty ze wstawianiem źródeł prądowych, gdyż dodatni prąd oznacza prąd wpływający do źródła co zazwyczaj jest interpretowane w sposób odwrotny. Dla porządku, na rys. 2 przedstawiono sposób deklaracji dodatnich i ujemnych węzłów dla źródeł niezależnych.



Rys. 2. Symbole niezależnych źródeł napięciowego i prądowego wraz z oznaczeniem węzłów i kierunków prądów.

Jak wspomniano wcześniej, jedynie nazwa źródła oraz węzły połączeniowe są obowiązkowymi elementami deklaracji źródeł niezależnych. Wartość napięcia/prądu generowana przez źródło (wartość domyślna wynosi zero) zależy od rodzaju analizy zleconej do symulacji PSPICE. PSPICE daje możliwość wykonania 3 podstawowych rodzajów analiz: stałoprądowej, częstotliwościowej małosygnałowej oraz czasowej. Wartość napięcia/prądu dla źródła niezależnego specyfikowana jest dla każdej z tych analiz niezależnie a zasady są następujące:

1) Wartość podana po opcjonalnym słowie kluczowym **DC** oznacza wartość zastosowaną w przypadku analizy stałoprądowej. Wartość ta jest również wykorzystywana w analizie częstotliwościowej małosygnałowej jednak tylko do wyznaczenia punktu pracy układu. Jeśli nie podano **specyfikacji_transient** to również wartość ta jest używana w czasie analizy czasowej i dane źródło ma stałą, niezmienną w czasie wydajność.
2) Wartość podana po słowie kluczowym **AC** oznacza amplitudę (w woltach dla źródła napięciowego lub amperach dla prądowego) i opcjonalną fazę początkową przebiegu harmonicznego wyrażoną w stopniach. Częstotliwość symulacji dla tego rodzaju analizy podaje się w poleceniu symulacji **.ac**.
3) **specyfikacja_transient** jest opcjonalna i jeśli występuje jej forma jest następująca: słowo kluczowe oznaczające rodzaj sygnału czasowego, za którym następuje lista parametrów odpowiadająca danemu rodzajowi sygnału umieszczona w nawiasach. Specyfikacja dla analizy czasowej jest funkcją napięcia (lub prądu dla źródła prądowego) w funkcji czasu. Możliwe jest wykorzystanie następujących specyfikacji czasowych: **EXP** (przebieg eksponencjalny), **PULSE** (przebieg prostokątny), **PWL** (przebieg w postaci punktów połączonych odcinkami), **SIN** (przebieg sinusoidalny) oraz **SFFM** (przebieg sinusoidalny modulowany częstotliwościowo).

Przykłady deklaracji źródeł niezależnych:

```
IBIAS 13 0 DC 2.3mA AC 1
```

- źródło prądowe o wydajności równej 2.3 mA w analizie stałoprądowej oraz w czasie obliczania punktu pracy i w czasie analiz czasowych, w analizie częstotliwościowej wydajność źródła wynosi 1 A.

```
VPULSE 1 0 PULSE(-1mA 1mA 2ns 2ns 2ns 50ns 100ns)
```

- źródło napięciowe o wydajności równej 0 V w analizie stałoprądowej oraz w czasie obliczania punktu pracy, w analizie czasowej jest to źródło przebiegu prostokątnego.

W przypadku specyfikacji czasowych, parametry globalne **TSTEP** i **TSTOP** są parametrami ustalanymi podczas wywołania polecenia symulacji czasowej **.TRAN**. Wspomniane parametry mają wpływ na ustalenie wartości domyślnych niektórych parametrów **specyfikcji_transient**. Wartości domyślne będą użyte w przypadku gdy lista parametrów wymienionych w nawiasach jest niekompletna, to znaczy kiedy jest ich mniej niż w ogólnej specyfikacji dla danego typu pobudzenia, wówczas pozostałe parametry przybierają wartości domyślne.

Specyfikacja czasowa typu EXP

Format ogólny specyfikacji czasowej typu eksponencjalnego jest następujący:

```
EXP (<v1> <v2> <td1> <tc1> <td2> <tc2>)
```

Znaczenie poszczególnych parametrów wyjaśnione jest w tab. 5.

Tab. 5. Parametry pobudzenia czasowego eksponencjalnego (EXP).

Parametr	Opis	Jednostka	Wartość domyślna
v1	wartość początkowa	V lub A	brak
v2	wartość szczytowa	V lub A	brak
td1	opóźnienie zbocza narastającego (opadającego)	s	0
tc1	stała czasowa	s	TSTEP
td2	opóźnienie zbocza opadającego (narastającego)	s	<td1>+TSTEP
tc2	stała czasowa	s	TSTEP

Kształt przebiegu czasowego dla pobudzenia **EXP** można zapisać następującymi równaniami:

Dla czasu od **0** do **td1**

$$v = v1 \tag{10}$$

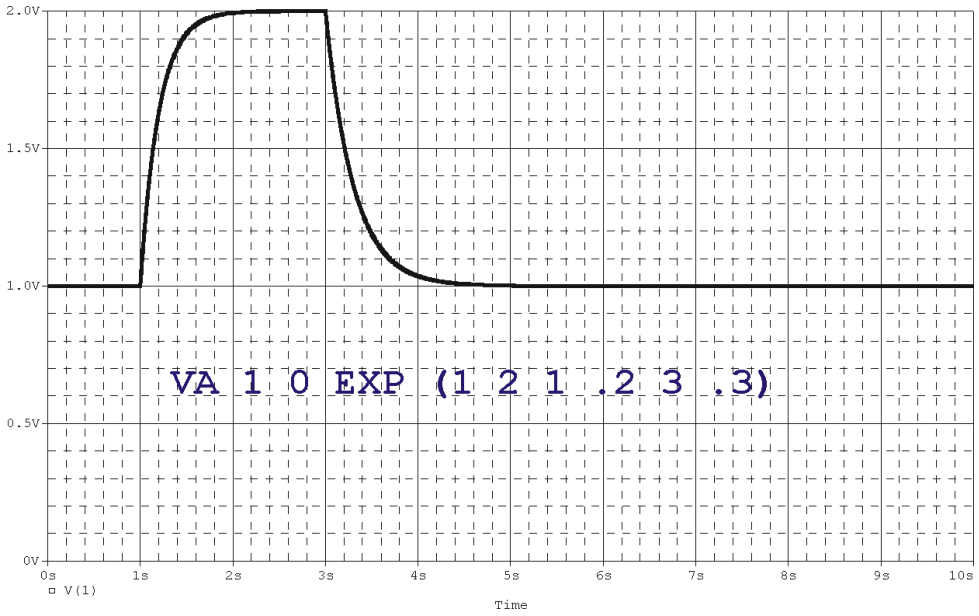
Dla czasu od **td1** do **td2**:

$$v = v1 + (v2 - v1)\left(1 - e^{-(TIME - td1)/tc1}\right) \tag{11}$$

Dla czas od **td2** do **TSTOP**:

$$v = v1 + (v2 - v1)\left[\left(1 - e^{-(TIME - td1)/tc1}\right) - \left(1 - e^{-(TIME - td2)/tc2}\right)\right] \tag{12}$$

Poniżej, na rys. 3, przedstawiony jest przykład deklaracji źródła napięciowego wraz z wynikającym z tej deklaracji przebiegiem czasowym.



Rys. 3. Przykład deklaracji przebiegu czasowego typu EXP.

Specyfikacja czasowa typu PULSE

Format ogólny specyfikacji czasowej typu PULSE jest następujący:

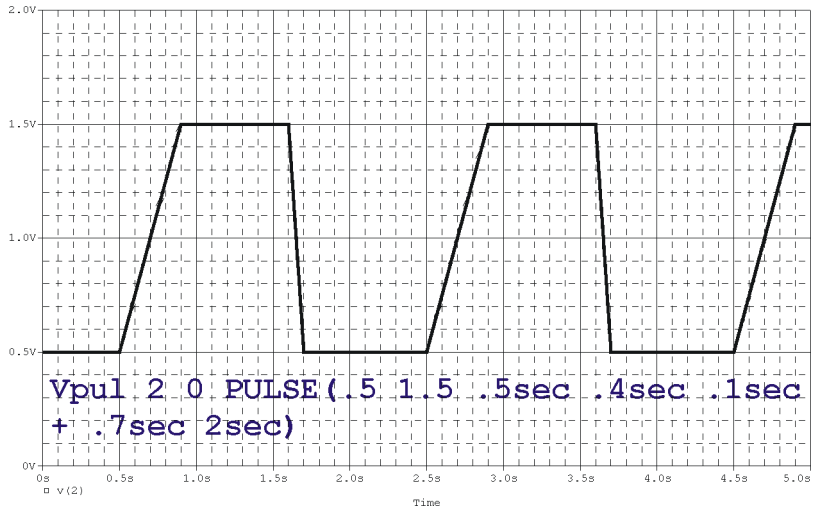
```
PULSE (<v1> <v2> <td> <tr> <tf> <pw> <per>)
```

Znaczenie poszczególnych parametrów wyjaśnione jest w tab. 6.

Tab. 6. Parametry pobudzenia czasowego PULSE.

Parametr	Opis	Jednostka	Wartość domyślna
`v1`	wartość początkowa	V lub A	brak
`v2`	wartość końcowa	V lub A	brak
`td`	opóźnienie do pierwszego zbocza	s	0
`tr`	czas narastania	s	TSTEP
`tf`	czas opadania	s	TSTEP
`pw`	czas trwania	s	TSTOP
`per`	okres przebiegu	s	TSTOP

Pobudzenie czasowe `PULSE` daje przebieg o wartości początkowej `v1`, który trwa przez `td` sekund a następnie zmienia się liniowo do wartości `v2` w czasie `tr` sekund i pozostaje stałe przez czas `pw` sekund, aby ponownie liniowo zmienić wartość do poziomu `v1` w czasie `tf` sekund, następnie pozostaje w stanie `v1` przez czas `[per-(tr+pw+tf)]` sekund, a następnie cykl jest powtarzany z wyłączeniem odcinka początkowego `td`.



Rys. 4. Przykład deklaracji przebiegu czasowego typu PULSE.

Specyfikacja czasowa typu SIN

Format ogólny specyfikacji czasowej typu **SIN** jest następujący:

SIN (<voff> <vamp> <freq> <td> <df> <phase>)

Pobudzenie czasowe **SIN** jest tłumionym przebiegiem sinusoidalnym. Znaczenie poszczególnych parametrów wyjaśnione jest w tab. 7.

Tab. 7. Parametry pobudzenia czasowego SIN.

Parametr	Opis	Jednostka	Wartość domyślna
voff	wartość przesunięcia	V lub A	brak
vamp	wartość amplitudy	V lub A	brak
freq	częstotliwość przebiegu	Hz	1/**TSTOP**
td	czas opóźnienia	s	0
df	współczynnik tłumienia	1/s	0
phase	faza początkowa	stopnie	0

Kształt przebiegu czasowego dla pobudzenia **SIN** można zapisać następującymi równaniami:

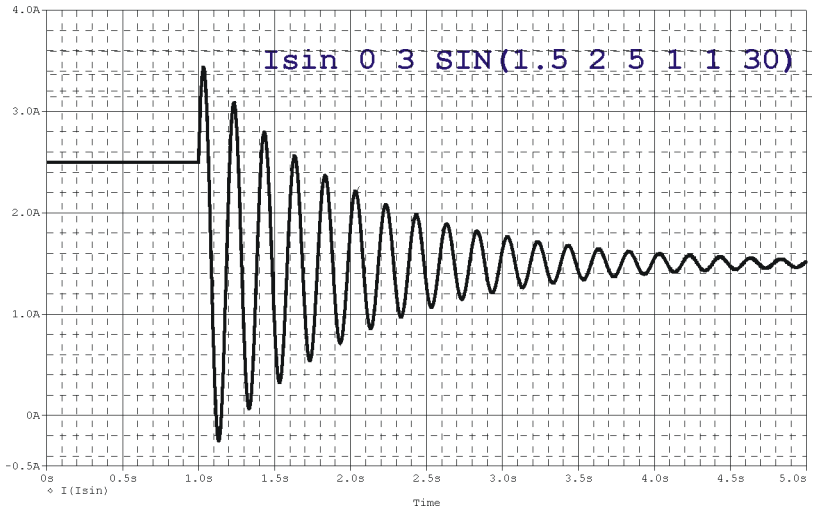
Dla czasu od **0** do **td**:

$$v = voff \cdot vamp \cdot \sin\left(2\pi \cdot phase/360\right) \tag{13}$$

Dla czasu od **td** do **TSTOP**:

$$v = voff \cdot vamp \cdot \sin\left[2\pi\left(freq\left(TIME - td\right)\right) + phase/360\right] \tag{14}$$

Na rys. 5, przedstawiony jest przykład deklaracji źródła napięciowego wraz z wynikającym z tej deklaracji przebiegiem czasowym.



Rys. 5. Przykład deklaracji przebiegu czasowego typu SIN.

Przebiegi typu **PWL** jak i **SFFM** są rzadziej stosowane i stąd zainteresowanych odsyłamy do literatury [1], [2].

2.7. Podstawowe rodzaje analiz oraz sterowanie wyjściem

Symulator elektryczny PSPICE umożliwia wykonanie 3 podstawowych analiz: stałoprądowej (polecenia **.DC** i **.OP**), małosygnałowej częstotliwościowej (polecenie **.AC**) oraz czasowej (polecenie **.TRAN**). Na podstawie analiz podstawowych możliwe jest również wykonanie analiz pochodnych (np. **.TF**, **.NOISE** czy **.FOUR**), które to bazują na wynikach analiz podstawowych i albo je dodatkowo przetwarzają albo też wykonują dodatkowe symulacje zbliżone do analiz podstawowych. Wyniki obliczeń z wykonanych analiz mogą być umieszczone w 2 następujących miejscach:

- plik wyjściowy tekstowy typu „*.out”,
- plik wyjściowy binarny typu „*.dat”.

Miejsce umieszczenia wyników obliczeń zależy od rodzaju wykonanej symulacji oraz od dodatkowych poleceń sterujących wyjściem: **.PROBE**, **.PRINT**, **.PLOT**. Po uruchomieniu symulacji zawsze tworzony jest plik „*.out”, który zawiera logi z wczytania badanego układu oraz wyniki tych symulacji, które są umieszczane w sposób tekstowy. Plik binarny „*.dat” tworzony jest tylko wtedy jeśli w pliku wejściowym pojawiło się polecenie **.PROBE**. Działanie poleceń sterujących wyjściem jest następujące:

- **.PROBE** - polecenie przekazywania wyników do postprocesora graficznego poprzez zapis wyników symulacji w binarnym pliku wyjściowym „*.dat”,
- **.PRINT** - polecenie wydruku wyników do pliku wyjściowego „*.out”,

- `.PLOT` - polecenie wykreślenia wykresu tekstowego (ze znaków ASCII) w pliku wyjściowym „*.out”.

Format ogólny polecenia `.probe` jest następujący:

```
.PROBE[/CSDF] [zmienne_wyjściowe]
```

Polecenie zapisuje napięcia i prądy występujące w układzie do binarnego pliku wyjściowego „*.dat”. Format zapisu nie jest ujawniony. W przypadku dodania parametru /CSDF zastosowany jest format wyjściowy typu `CSDF`. W przypadku braku listy `zmienne_wyjściowe` do pliku wyjściowego zapisywane są wszystkie możliwe napięcia i prądy występujące w badanym obwodzie. Przykłady:

```
.PROBE
.PROBE V(3) V(2,3) V(R1) I(VIN) I(R2) IB(Q13)
+VBE(Q13)
.PROBE/CSDF
```

W pierwszym przykładzie wszystkie napięcia i prądy układowe zostaną zapisane do pliku „*.dat”. W drugim przykładzie zapisane zostaną tylko wybrane sygnały. W trzecim przykładzie zapisane zostaną wszystkie sygnały do pliku w formacie `CSDF`. Format ten daje większe pliki wynikowe ale umożliwia przenoszenie ich do innych programów.

Format ogólny polecenia `.print` jest następujący:

```
.PRINT <rodzaj_analizy> [zmienna_wyjściowa]
```

Polecenie to powoduje umieszczenie w tekstowym pliku wyjściowym „*.out” wyników obliczeń w postaci tabeli danych. W danym poleceniu można wyszczególnić tylko jeden rodzaj analizy ale można podać kilka poleceń `.PRINT` w pliku wejściowym.

Przykłady wydanych poleceń `.PRINT`:

```
.PRINT DC  V(3)  V(2,3) V(R1) I(VIN)
.PRINT NOISE INOISE ONOISE DB(INOISE) DB(ONOISE)
```

W pierwszym przykładzie wyszczególnione sygnały dla analizy `.DC` zostaną umieszczone w pliku „*.out”. W drugim przykładzie zapisane zostaną wybrane sygnały dla analizy `.NOISE`. W ramach parametru `zmienna_wyjściowa` można umieścić: napięcie na dowolnym elemencie obwodu np. `V(R1)`, prąd dowolnego elementu obwodu np. `i(L1)`, napięcie w dowolnym węźle obwodu np. `V([out])`, różnicę napięć pomiędzy dowolnymi węzłami obwodu np. `V([in],[out])`. Oprócz powyższych wartości, możliwe są również zaawansowane opcje takie jak wartości podawane w mierze logarytmicznej lub argumenty dla analizy AC, prądy poszczególnych wyprowadzeń tranzystorów, napięcia pomiędzy wybranymi wyprowadzeniami tranzystorów, itd. - zainteresowanych odsyłamy do literatury [2].

Format ogólny polecenia `.plot` jest następujący:

```
.PLOT <typ_analizy> [zmienna_wyjściowa] ( [<dolny_limit>,<górny_limit>] )
```

Można wyszczególnić tylko jeden rodzaj analizy ale można podać kilka poleceń **.PLOT**. Można wyszczególnić maksymalnie 8 zmiennych dla jednego polecenia. Limity, jeśli są podane, oznaczają zakres osi Y dla wszystkich zmiennych stosowanych na tej osi. Dla analizy **.AC** limity nie są stosowane oraz oś Y jest zawsze logarytmiczna. Polecenie **.PLOT** było wprowadzone w czasie gdy komputery obsługiwały wyłącznie tryb tekstowy wyświetlania, obecnie lepiej jest je zastąpić poleceniem **.PROBE** a wynik oglądać w postprocesorze graficznym.

Przykład wywołania polecenia:

```
.PLOT TRAN V(3) V(2,3) (0,5V) ID(M2) I(VCC) (-50mA,50mA)
```

2.7.1. Analiza stałoprądowa

Analiza stałoprądowa występuje w dwóch wersjach: jako wyznaczenie punktu pracy (**.OP**) lub jako wyznaczenie sparametryzowanego punktu pracy (**.DC**). Polecenie **.DC** daje w rezultacie zamiast pojedynczego rozwiązania zbiór wyznaczonych punktów pracy przy zmieniającym się jednym lub dwóch parametrach. Niezależnie od zastosowanego polecenia, działanie analizy jest zawsze takie samo i polega na:

- usunięciu z badanego obwodu elementów inercyjnych, zgodnie z tą zasadą kondensatory zamienione są rozwarciami a cewki zwarciami,
- wyznaczenie stałoprądowego punktu pracy, przy tej analizie elementy nieliniowe zachowują swój nieliniowy charakter jednak wszelkie ich właściwości inercyjne są pomijane,
- w przypadku analizy **.DC**, punkt powyższy jest powtarzany wielokrotnie zgodnie ze specyfikacją zmian zawartą w poleceniu.

Przykłady wywołania analiz:

```
.OP
```

- wykonanie pojedynczej analizy punktu pracy, analiza wyznacza punkt pracy a szczegółowe wyniki umieszcza w pliku wyjściowym „.out”. Bez polecenia **.op** w pliku wyjściowym podawane są tylko wartości napięć węzłowych. Polecenie **.OP** nie ma żadnych dodatkowych parametrów.

```
.DC Vid 0 1 .1
```

- wykonanie wielokrotnej analizy punktu pracy, najpierw analiza jest wykonana dla napięcia Vid=0, następnie napięcie wzrasta o 0.1V i ponownie wykonywana jest analiza, zwiększanie Vid następuje aż do osiągnięcia wartości 1V.

Analiza **.DC** występuje w 4 formach przemiatania parametrów analizy: **LIN**, **OCT**, **DEC** oraz **LIST**. Analiza może być zagnieżdżona tzn. zmiana parametru pierwszego jest wykonywana w pętli wewnętrznej i następuje częściej niż parametru drugiego. Format ogólny deklaracji **.DC** dla przemiatania liniowego (**LIN**) jest następujący:

```
.DC [LIN] <nazwa_przemiatanej_zmiennej> <start> <koniec> <wartość_inkrementu>
```

```
+ [specyfikacja_zagnieżdżenia]
```

Dla parametrów w **specyfikacja_zagnieżdżenia** stosowane są takie same zasady jak dla podstawowego przemiatania. Przemiatanie liniowe jest domyślnie i słowo kluczowe **LIN** jest opcjonalne.

Przykłady zlecenia do wykonania analizy stałoprądowej z przemiataniem liniowym:

```
.DC VIN -.25  .25  .05
.DC LIN I2  5mA  -2mA  0.1mA
.DC VCE 0V 10V .5V IB 0mA 1mA 50uA
```

Format ogólny deklaracji **.DC** dla przemiatania logarytmicznego (**DEC** i **OCT**) jest następujący:

```
.DC [DEC lub OCT] <nazwa_przemiatanej_zmiennej> <start> <koniec>
+<liczba_punktów> [specyfikacja zagnieżdżenia]
```

Dla parametrów w **specyfikacja_zagnieżdżenia** stosowane są takie same zasady jak dla podstawowego przemiatania. W przypadku podania słowa kluczowego **DEC**, **liczba_punktów** oznacza liczbę równomiernie rozłożonych punktów na osi logarytmicznej dla dziesięciokrotnej zmiany wartości przemiatanej, jest to liczba symulowanych wartości dla dekady zmiany. W przypadku słowa kluczowego **OCT** liczba ta przypada na dwukrotną zmianę przemiatanego parametru.

Przykład:

```
.DC DEC NPN QFAST(IS) 1E-18 1E-14  5
```

- w przykładzie powyższym, parametr **IS** modelu typu **NPN** o nazwie **QFAST** zmieniany jest w zakresie od 10^{-18} do 10^{-14}, przy czym na każdą dekadę przypada 5 punktów symulacyjnych. Łączna liczba punktów symulacyjnych wynosi w powyższym przypadku $4 \cdot 5+1=21$.

Format ogólny deklaracji **.DC** dla przemiatania typu **LIST** jest następujący:

```
.DC <nazwa_przemiatanej_zmiennej> [LIST] <lista_wartości>
+ [specyfikacja_zagnieżdżenia]
```

W przypadku przemiatania typu **LIST**, podawana jest lista wartości, dla których ma być wykonana symulacja. Przykłady deklaracji symulacji:

```
.DC VDD LIST 2 3 4 5 6
.dc Vce list 1 2 3 4 5 6 Ib list 10u 100u 1m
```

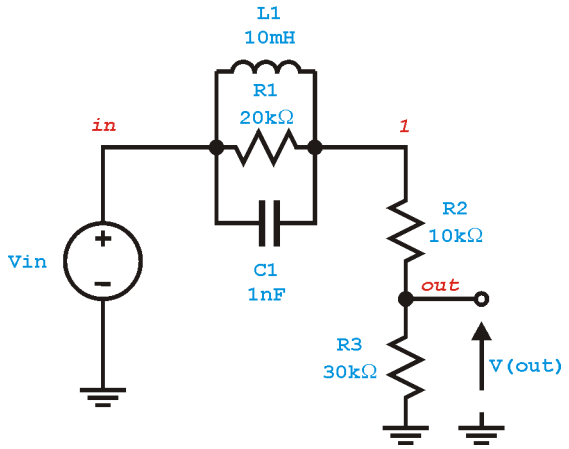
Niezależnie od rodzaju przemiatania, elementami przemiatanymi mogą być zmienne wyszczególnione w tab. 8.

Tab. 8. Zmienne możliwe do przemiatania w analizie `.DC`.

Zmienna	Sposób zapisu	Znaczenie
źródło niezależne	nazwa niezależnego źródła napięciowego lub prądowego	podczas przemiatania zmienia się wydajność źródła
parametr modelu	typ i nazwa modelu oraz nazwa parametru w nawiasie	zmieniany jest wyszczególniony parametr modelu. Nie można zmieniać niektórych parametrów modeli, są to: W i L dla tran. MOS, oraz parametry temperaturowe.
temperatura	słowo **`TEMP`**	zmieniana jest bieżąca temperatura symulacji
parametr globalny	słowo kluczowe **`PARAM`** i nazwa parametru	zmieniana jest wartość parametru, wszystkie wartości zależne od tego parametru są wyliczane na nowo

Przykład 1. Symulacja z wykorzystaniem analiz stałoprądowych

Poniższy prosty przykład przedstawia wykonanie analizy stałoprądowej dla układu z rys. 6. W układzie umieszczonych jest kilka elementów pasywnych, stanowiących łącznie dzielnik napięciowy. Napięcie wytworzone przez źródło napięciowe **`Vin`** po przejściu przez równoległy obwód **`RLC`** dochodzi do rezystorowego dzielnika napięciowego. Należy zauważyć, że dla analizy **`.DC`** cewka **`L1`** stanowi zwarcie, wobec czego obwód upraszcza się jakby układu równoległego **`RLC`** nie było.



Rys. 6. Prosty układ pasywny poddany symulacji stałoprądowej. Nazwy podane kursywą odpowiadają nazwom węzłów w pliku wejściowym PSPICE.

Zawartość pliku wejściowego z rozszerzeniem „*.cir” opisującego układ z rys. 6 i zlecającego do wykonania analizy stałoprądowe przedstawiona jest poniżej:

```
przykład nr 1, analizy stałoprądowe
```

```
*napięciowe źródło wejściowe
Vin in 0 dc 1
* obwod pasywny
L1 in 1 10mH
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analizy
.op
.dc Vin 0 5 .01
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

Pierwsza linia stanowi tytuł i nie podlega dodatkowym regułom. Linie rozpoczynające się od `*` są komentarzem. Ostatnia linia `.end` oznacza koniec pliku wejściowego. Oprócz elementów składowych obwodu mamy zlecone dwie symulacje:

```
.op
.dc Vin 0 5 .01
```

Symulacja `.OP` ustala punkt pracy na podstawie wartości napięcia podanego w linii:

```
Vin in 0 dc 1
```

i wartości elementów obwodu po usunięciu elementów inercyjnych. Obliczone wartości napięć umieszcza w pliku wyjściowym „*.out”. Poniżej przedstawiono fragment pliku wyjściowego wygenerowany przez symulator PSPICE a dotyczący analizy `.OP`:

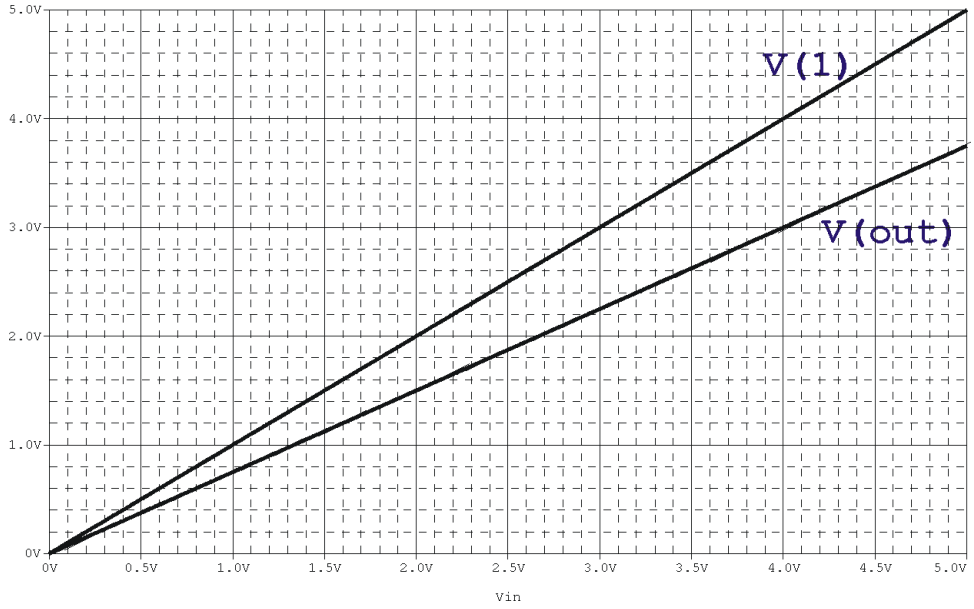
```
przykład nr 1
 ****     SMALL SIGNAL BIAS SOLUTION       TEMPERATURE =   27.000 DEG C
**********************************************************************
 NODE   VOLTAGE     NODE   VOLTAGE     NODE   VOLTAGE     NODE   VOLTAGE
(    1)    1.0000  (   in)    1.0000  (  out)     .7500

    VOLTAGE SOURCE CURRENTS
    NAME         CURRENT
    Vin         -2.500E-05

    TOTAL POWER DISSIPATION   2.50E-05  WATTS
```

Ze względu na to, że źródło napięciowe `Vin` ma zadeklarowaną wartość napięcia dla analizy `.DC` równą 1 V, napięcie w węzłach `in` oraz `1` jest równe 1 V a w węźle wyjściowym `out`, po uwzględnieniu rezystancji `R2` i `R3`, jest równe `V(out)=Vin*R3/(R2+R3)` = 3/4 V. Analiza `.dc Vin 0 5 .01` nadpisuje zdeklarowaną wydajność źródła `Vin` wartością zaczynającą się od 0 a kończącą na wartości 5 V z krokiem co 10 mV. W rzeczywistości można powiedzieć, że zamiast pojedynczej analizy `.OP`, zostanie

wykonanych 501 takich analiz dla wartości Vin od 0 do 5 V. Ze względu na umieszczenie polecenia `.probe` wyniki analizy `.dc` zostaną umieszczone w binarnym pliku wyjściowym „*.dat”. Symulator PSPICE posiada wbudowany postprocesor graficzny wyników zapisanych w plikach „*.dat” umożliwiający wykreślanie wykresów na podstawie tych wyników. Na rys. 7 przedstawiono napięcia `V(out)` oraz `v(1)` wykreślone na podstawie wykonanej analizy `.dc`.



Rys. 7. Wyniki symulacji stałoprądowej układu z rys. 6.

Do wykreślenia rodziny charakterystyk można wykorzystać zagnieżdżoną symulację stałoprądową. W prezentowanym przykładzie takie przemiatanie zostanie wykonane dla 3 wartości rezystora R3 równych odpowiednio: 10 kΩ, 20 kΩ i 30 kΩ. Zgodnie z tab. 8, w celu przemiatania rezystancji możemy wykorzystać 2 możliwości:

- dla rezystora R3 zdefiniować model a następnie przemiatać parametr modelu lub
- zdefiniować parametr globalny a następnie wartość rezystancji R3 powiązać z tym parametrem i dokonać przemiatania parametru.

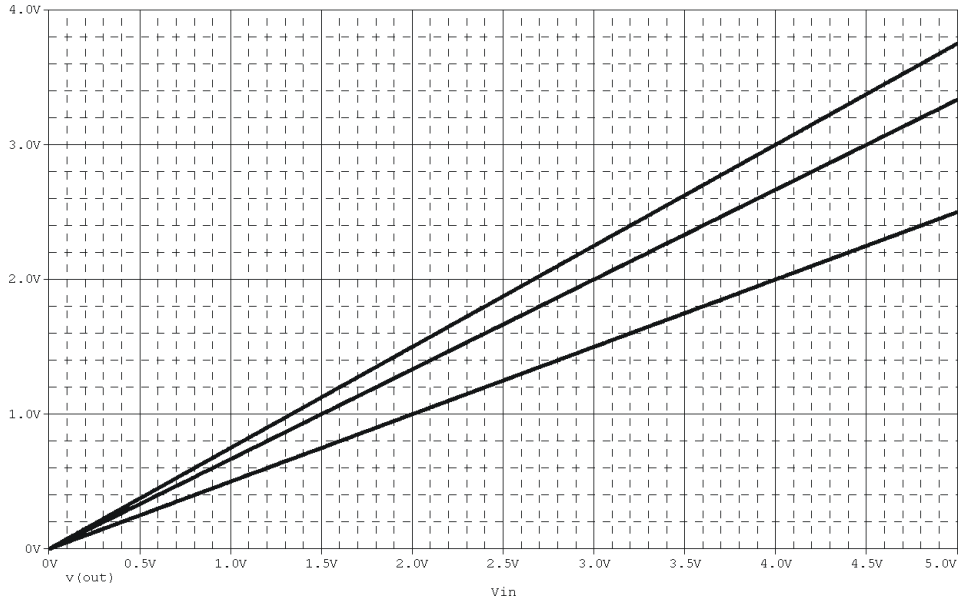
W rzeczywistości PSPICE daje jeszcze jedną możliwość przemiatania. Jej wykorzystanie nie jest ograniczone tylko do analizy stałoprądowej lecz jest stosowane do wszelkich zleconych analiz równocześnie. Takie przemiatanie globalne możliwe jest poprzez zastosowanie polecenia `.STEP` szczegółowo przedstawionego w jednym z kolejnych podrozdziałów. Wykreślenie rodziny charakterystyk przy wykorzystaniu modelu lub parametru globalnego wymaga modyfikacji elementu `R3` oraz analizy `.DC`. W przypadku zastosowania modelu taka modyfikacja jest następująca:

```
R3 out 0 r 1
.model r res(R=30k)
.dc Vin 0 5 .01 res r(R) list 10k 20k 30k
```

W przypadku zastosowania parametru globalnego taka modyfikacja jest następująca:

```
R3 out 0 {resistance}
.param resistance=30k
.dc Vin 0 5 .01 param resistance list 10k 20k 30k
```

Obie powyższe modyfikacje dają te same rezultaty a wynik wykreślenia rodziny charakterystyk dla napięcia w węźle `out` przedstawiony jest na rys. 8.



Rys. 8. Rodzina charakterystyk powstała na podstawie analizy stałoprądowej dla układu z rys. 6 dla R3 równego odpowiednio 10 kΩ, 20 kΩ i 30 kΩ .

2.7.2. Analiza częstotliwościowa małosygnałowa

Analiza częstotliwościowa małosygnałowa `.AC` zakłada, że układ jest liniowy i oblicza wartości napięć i prądów w obwodzie przy założeniu istnienia wyłącznie pobudzeń harmonicznych. Jeśli pobudzeń jest więcej niż jedno, wówczas przyjęte są identyczne ich częstotliwości, natomiast faza każdego z nich może być indywidualnie ustalona. Dla analizy `.AC` układ jest liniowy i z tego względu nie ma znaczenia wartość napięcia wejściowego – nie spowoduje ono przesterowania czy powstania zniekształceń nieliniowych. Z tego względu często do wejścia przykłada się napięcie równe 1 V a napięcie na wyjściu automatycznie jest wzmocnieniem (bo podzielone przez 1 V da ten sam wynik). Proces wykonywania symulacji `.AC` jest następujący:

- najpierw obliczany jest punkt pracy, identycznie jak w poleceniu `.OP`, jednak z tą różnicą, że nie są szczegółowo raportowane w pliku wyjściowym parametry obliczonego punktu pracy,
- następnie wartości stałych napięć i prądów obliczone w analizie punktu pracy służą do wyznaczenia wartości parametrów zastępczych modeli małosygnałowych elementów nieliniowych (np. transkonduktancja, rezystancja wyjściowa, pojemność złączowa i.t.d.),
- elementy nieliniowe zastąpione są ich zastępczymi modelami liniowymi małosygnałowymi (parametry tych modeli były wyznaczone poprzednio),

- następnie obliczane są wartości napięć i prądów w obwodzie przy czym pobudzeniami w układzie stają się wszystkie niezależne źródła napięciowe i prądowe z niezerową specyfikacją AC, oczywiście zakłada się, że te źródła generują sygnał harmoniczny,
- zakres częstotliwości dla których obliczone są napięcia i prądy obwodowe zdefiniowany jest poprzez parametry przy wywołaniu polecenia `.AC`,
- wartości obliczonych napięć i prądów są liczbami zespolonymi.

Format ogólny deklaracji analizy `.AC` jest następujący:

```
.AC <LIN | DEC | OCT> <liczba_punktów>
+ <częstotliwość_początkowa>
+ <częstotliwość_końcowa>
```

Polecenie `.AC` zleca do wykonania analizę małosygnałową częstotliwościową a w ramach parametrów polecenia definiowany jest zakres i sposób zmian wartości częstotliwości niezależnych źródeł napięciowych i prądowych o niezerowych specyfikacjach `AC`. Parametr `LIN` oznacza jednakowe odstępy pomiędzy kolejnymi wartościami częstotliwości, parametr `DEC` oraz `OCT` oznacza jednakową odległość pomiędzy kolejnymi wartościami analizowanej częstotliwości na osi częstotliwości skalowanej logarytmicznie. W przypadku przemiatania `LIN`, `liczba_punktów` oznacza całkowitą liczbę wartości częstotliwości, dla których analizowany jest dany układ. Dla przemiatania `DEC` i `OCT`, `liczba_punktów` oznacza liczbę punktów przypadającą odpowiednio na jedną dekadę (dziesięciokrotną) i oktawę (dwukrotną) zmiany częstotliwości.

Przykłady:

`.ac lin 201 1k 2k` - 201 punków częstotliwościowych, pierwsza wartość częstotliwości wynosi 1 kHz, ostatnia 2 kHz,

`.ac dec 50 100Hz 1GHz` - 50 punków na każdą dekadę, pierwsza częstotliwość 50 Hz, ostatnia 1 GHz, łączna liczba punktów częstotliwościowych 7*50+1=351,

`.ac oct 30 1 20` - 30 punków na każdą oktawę, pierwsza częstotliwość 1 Hz, ostatnia częstotliwość 20 Hz.

Przykład 2. Symulacja małosygnałowa częstotliwościowa

Jako przykład analizy częstotliwościowej wykorzystany zostanie układ z rys. 6. W stosunku do zawartości pliku wejściowego PSPICE dla przykładu poprzedniego, źródło napięcia `Vin` zostało uzupełnione o specyfikację wartości dla symulacji częstotliwościowej. Dodana została także linia zawierająca zlecenie analizy `AC`. Pełen kod przedstawiony jest poniżej.

```
przykład nr 2, analiza małosygnałowa częstotliwościowa
*napięciowe źródło wejściowe
Vin in 0 dc 1 ac 1
* obwod pasywny
```

```
L1 in 1 10mH
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analiza
.ac dec 300 1k 1000k
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

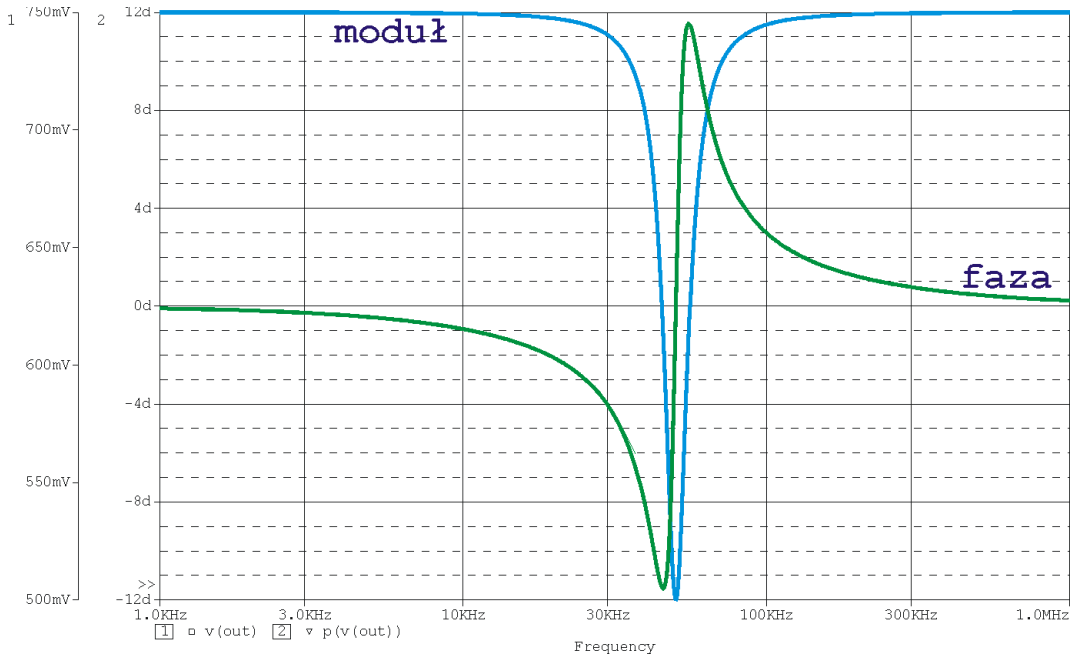
Zlecona powyżej analiza częstotliwościowa oblicza wartości napięć węzłowych i prądów obwodowych przy pobudzeniu harmonicznym w stanie ustalonym przy zmianie częstotliwości napięcia **Vin** w zakresie od 1 kHz do 1 MHz. Fazy sygnałów odniesione są do fazy sygnału **Vin**, którego faza początkowa nie została wyspecyfikowana i dlatego ma wartość domyślną równą 0. Łączna liczba punktów częstotliwościowych, dla których wykonane zostaną obliczenia wynosi: 300pkt/dek * 3dek + 1pkt. kończący = 901. Ze względu na polecenie **.probe** wartości obliczonych napięć i prądów umieszczone zostają w pliku „*.dat”. Należy zauważyć, że elementy **R1**, **L1** i **C1** stanowią równoległy obwód RLC a jego impedancja w rezonansie wynosi **R1**. Dla częstotliwości znacznie niższych i znacznie wyższych niż częstotliwość rezonansowa impedancja dąży do zera. Biorąc to pod uwagę, nasz dzielnik impedancyjny powinien dać wzmocnienie równe ¾ dla niskich i wysokich częstotliwości, natomiast w rezonansie wzmocnienie powinno wynosić:

$$\left|\frac{V_{OUT}}{V_{IN}}\right| = \frac{R3}{R1+R2+R3} = \frac{1}{2} \tag{15}$$

Sama częstotliwość rezonansowa może być wyznaczona na podstawie wzoru:

$$f_{REZ} = \frac{1}{2\pi\sqrt{L1 \cdot C1}} = \frac{1}{2\pi\sqrt{10mH \cdot 1nF}} = 50{,}329kHz \tag{16}$$

Na rys. 9 przedstawiono wykreślone wartości modułu i fazy napięcia występującego w węźle ***out*** , co jest zgodne z powyżej przytoczoną analizą przybliżoną.



Rys. 9. Wynik symulacji małosygnałowej częstotliwościowej układu z rys. 6. Wykreślony został moduł i faza napięcia w węźle *out*.

2.7.3. Analiza czasowa

Format ogólny deklaracji analizy czasowej jest następujący:

```
.TRAN [/OP] <krok_wydruku> <czas_analizy>
+[nie_drukować [krok_obliczeniowy]][SKIPBP].
```

Przykłady zlecenia analizy czasowej:

`.TRAN 1ns 100ns` - analiza do 100 ns z krokiem wydruku 1 ns,

`.TRAN 1ns 100ns 0ns .1ns SKIPBP` – analiza do czasu równego 100 ns z krokiem obliczeniowym 0,1 ns bez obliczania punktu pracy.

Opcja `/OP` powoduje umieszczenie szczegółowych danych dotyczących punktu pracy w pliku wyjściowym „*.out”. Opcja `SKIPBP` powoduje pominięcie obliczania punktu pracy, gdy jest użyta, punkt pracy jest ustalony za pomocą wartości IC w deklaracjach kondensatorów i cewek.

Wartość parametru `krok_wydruku` jest używana tylko do poleceń `.PRINT`, `.PLOT` i `.FOUR`. Parametr ten nie zmienia danych przekazywanych do binarnego pliku wyjściowego „*.dat” generowanego poleceniem `.PROBE` (dane tam umieszczane są zgodne z rzeczywistymi punktami obliczeniowymi). Ponieważ obliczenia są dokonywane w innych chwilach czasowych niż powstałe z parametru `krok_wydruku`, do określenia wartości wydruku używana jest interpolacja przy użyciu wielomianu drugiego stopnia. Parametr `nie_drukować` oznacza czas od zera do wartości tego parametru, dla którego zostaną pominięte (nie zapisane) wyniki obliczeń. Parametr `krok_obliczeniowy` zmienia domyślną wartość kroku obliczeniowego równą `czas_analizy/50`. Ogólne właściwości analizy `.TRAN` są następujące:

- analiza oblicza zachowanie się układu w czasie, argumentem tej analizy jest czas,
- analiza czasowa jest to jedyna analiza podstawowa, która nie wykonuje uproszczeń w analizowanym układzie, elementy nieliniowe zachowują swój nieliniowy charakter (nie ma linearyzacji jak dla analizy `.AC`), elementy inercyjnie nie są usuwane z obwodu (odwrotnie niż dla analiz stałoprądowych),
- analiza zawsze rozpoczyna się w czasie równym 0 i kończy w czasie równym parametrowi `czas_analizy`,
- przed rozpoczęciem analizy obliczany jest punkt pracy, ten punkt pracy może być inny niż dla analizy `.AC`, ponieważ niezależne źródła napięciowe dla analizy czasowej mogą mieć inne wydajności początkowe,
- domyślny krok analizy dobierany jest w zależności od aktywności układu (zmian w układzie),
- domyślna wartość kroku analizy wynosi `czas_analizy/50`, ale w przypadku, gdy nie ma w układzie elementów inercyjnych, krok analizy jest równy `krok_wydruku`,
- analiza `.TRAN` ustawia wartości globalnych zmiennych `TSTEP` oraz `TSTOP`, które to są następnie używane do wyznaczania niektórych wartości domyślnych `TSTEP=krok_wydruku`, `TSTOP=czas_analizy`,
- dla parametru `krok_obliczeniowy` można zastosować funkcję `SCHEDULE(time1,val1,time2,valn...timen,valn)`, funkcja ta określa jaki krok obliczeniowy ma być użyty w kolejnych przedziałach czasowych, znaczenie jest następujące: od czasu `time1` użyj wartości `val1`, następnie od czasu `time2` użyj wartości `val2` itd.

Przykład użycia funkcji `SCHEDULE`:

```
.TRAN 1ns 90ns 0ns {SCHEDULE(0,1ns,25ns,.1ns)}
```

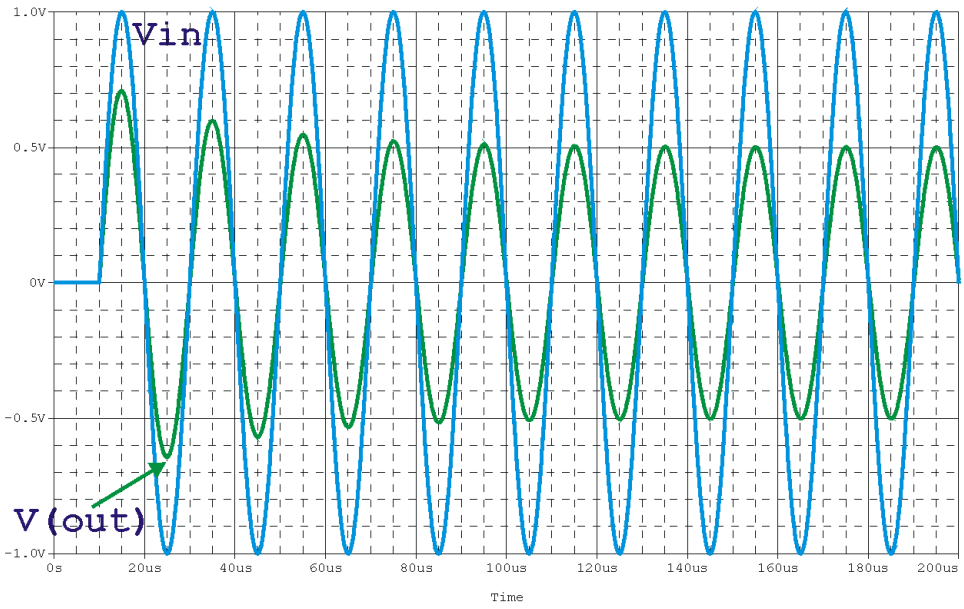
Przykład 3. Symulacja czasowa

Jako przykład wykonania analizy czasowej wykorzystany zostanie układ z rys. 6. W stosunku do zawartości pliku wejściowego PSPICE dla przykładów poprzednich, źródło napięcia `Vin` zostało uzupełnione o specyfikację wartości dla symulacji czasowej. Zastosowano przebieg sinusoidalny o amplitudzie 1 V i częstotliwości równej 50 kHz (jest to częstotliwość zbliżona do częstotliwości rezonansowej), przebieg rozpoczyna się po czasie opóźnienia równym 10 µs. Dodana została także linia zawierająca zlecenie analizy `.TRAN`. Pełen kod przedstawiony jest poniżej.

```
przykład nr 3, analiza czasowa
*napięciowe źródło wejściowe
Vin in 0 dc 1 ac 1 sin(0 1 50k 10u)
* obwod pasywny
L1 in 1 10mH
C1 in 1 1nF
```

```
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analiza
.tran 100n 200u 0 100n
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

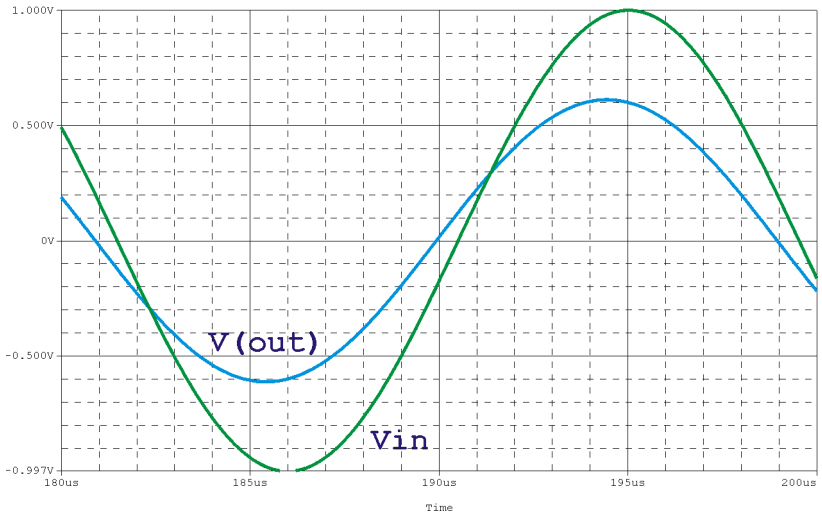
Przy deklarowaniu analizy `.TRAN` należy podać czas końca symulacji umożliwiający obserwację badanego przebiegu. Dla naszego przypadku, przebieg `Vin` o częstotliwości równej 50 kHz ma okres równy 20 μs. Czas końca analizy ustawiono na 200 μs co umożliwia obserwację ok. 10 okresów sygnału wejściowego. Domyślna wartość kroku obliczeniowego analizy byłaby równa 200 μs/50=4 μs, co dla okresu sygnału `Vin` równego 20 μs dałoby 5 punktów obliczeniowych na 1 okres przebiegu harmonicznego. Taka liczba próbek jest zbyt mała aby dokładnie oddać przebiegi napięć i prądów w naszym przypadku. Z tego względu dodano parametr `krok_obliczeniowy` i ustalono jego wartość na 100 ns, co daje 200 punktów obliczeniowych na jeden okres przebiegu harmonicznego. Na rys. 10 przedstawiono wykreślone wartości napięcia Vin oraz napięcia występującego w węźle ***out*** . Jak można zauważyć, amplituda napięcia w węźle ***out*** dąży do wartości 500 mV (do stanu ustalonego przy pobudzeniu harmonicznym) co jest zgodne z amplitudą dla symulacji `.AC` odczytaną z rys. 9 dla częstotliwości 50 kHz.



Rys. 10. Wynik symulacji czasowej układu z rys. 6. Wykreślone zostało napięcie Vin i napięcie w węźle *out*.

Na rys. 11 przedstawiono wynik dodatkowej analizy czasowej, dla której zmieniono częstotliwość pobudzenia harmonicznego na 55,4 kHz. Częstotliwość tą wybrano z tego względu, że wg rys. 9 dla

właśnie tej częstotliwości przesunięcie fazowe jest maksymalne i wynosi około 11,5 °. Na rys. 11 można wyraźnie zauważyć, że napięcie w węźle **out** pojawia się szybciej niż napięcie **Vin**. Dodatkowo amplituda na wyjściu jest znacznie wyższa niż dla częstotliwości równej 50 kHz.



Rys. 11. Wynik symulacji czasowej układu z rys. 6 dla częstotliwości napięcia Vin równego 55,4 kHz. Przybliżony został ostatni okres przebiegu. Napięcie w węźle *out* pojawia się wcześniej niż Vin.

2.8. Analizy pochodne

Analizy pochodne względem analiz podstawowych polegają na dalszym przetworzeniu wyników analiz podstawowych albo na lekkiej ich modyfikacji i w ten sposób na uzyskuje się dodatkowe informacje o badanym obwodzie. Podstawowe zasady co do stosowanych uproszczeń i zasad obliczeniowych są identyczne jak dla analiz podstawowych i dlatego bardzo ważne jest aby zwracać uwagę na której analizie podstawowej poszczególne analizy pochodne bazują.

2.8.1. Analiza Fouriera

Analiza Fouriera jest pochodną analizy czasowej. Jej składnia ogólna jest następująca:

```
.FOUR <częstotliwość> [liczba_harmonicznych]  <zmienne_wyjściowe>
```

Przykłady:

.FOUR 10kHz 20 v([inp]) - dekompozycja napięcia w węźle **inp** wokół częstotliwości podstawowej 10 kHz,

.FOUR 20 10 v([out1],[out2]) – dekompozycja różnicy napięć pomiędzy węzłami **out1** i **out2** wokół częstotliwości równej 20 Hz.

Analiza **.FOUR** powoduje rozłożenie wyniku analizy czasowej **.TRAN** na składniki harmoniczne i umieszczenie wyników obliczeń w pliku wyjściowym „*.out”. Podstawowe zasady dla analizy są następujące:

- dla analizy `.FOUR` brane są wyniki z analizy `.TRAN` z czasu od `czas_analizy-1/częstotliwość` do `czas_analizy`,
- analiza czasowa musi trwać co najmniej 1 okres częstotliwości rozkładu,
- jeśli nie podano parametru `liczba_harmonicznych` obliczana jest składowa stała, składnik podstawowy oraz harmoniczne od 2-giej do 9-tej,
- dekompozycji na szereg Fouriera podlegają sygnały podane na liście `zmienne_wyjściowe`, która to lista ma składnię identyczną jak przy poleceniu `.PRINT`, między innymi można tam umieszczać: napięcia na dowolnym elemencie obwodu np. `V(R1)`, prąd dowolnego elementu obwodu np. `i(L1)`, napięcie w dowolnym węźle obwodu np. `V([out])`, różnicę napięć pomiędzy dowolnymi węzłami obwodu np. `V([in],[out])`.

Przykład 4. Symulacja czasowa i Fouriera

Do przykładu analizy czasowej z wykorzystaniem analizy Fouriera wykorzystany zostanie układ z rys. 6. W stosunku do zawartości pliku wejściowego PSPICE dla przykładu analizy czasowej, dodano wpis dla analizy `.FOUR`. Dodana została także linia zawierająca zlecenie dodatkowej analizy `.TRAN` ze zmienionym czasem końca analizy. Pełen kod przedstawiony jest poniżej.

```
przykład nr 4, analiza czasowa + Fouriera
*napięciowe źródło wejściowe
Vin in 0 dc 1 ac 1 sin(0 1 50k 10u)
* obwod pasywny
L1 in 1 10mH
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analizy
*.tran 100n 30u 0 100n
.tran 100n 200u 0 100n
.four 50k 15 v([out])
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

Wynik symulacji Fouriera dla analizy czasowej wykonywanej do czasu 200 µs, skopiowany z tekstowego pliku wyjściowego „*.out”, pokazany jest poniżej:

```
przykład nr 3, analiza czasowa + Fouriera
 ****     FOURIER ANALYSIS                 TEMPERATURE =   27.000 DEG C
*******************************************************************************
FOURIER COMPONENTS OF TRANSIENT RESPONSE V(out)
 DC COMPONENT =  -3.536982E-05
```

HARMONIC NO	FREQUENCY (HZ)	FOURIER COMPONENT	NORMALIZED COMPONENT	PHASE (DEG)	NORMALIZED PHASE (DEG)
1	5.000E+04	5.012E-01	1.000E+00	1.784E+02	0.000E+00
2	1.000E+05	2.372E-05	4.733E-05	1.023E+02	-2.546E+02
3	1.500E+05	8.945E-06	1.785E-05	9.981E+01	-4.355E+02
4	2.000E+05	4.795E-06	9.567E-06	9.999E+01	-6.138E+02
5	2.500E+05	3.015E-06	6.016E-06	1.005E+02	-7.918E+02
6	3.000E+05	2.051E-06	4.093E-06	1.016E+02	-9.690E+02
7	3.500E+05	1.524E-06	3.041E-06	1.023E+02	-1.147E+03
8	4.000E+05	1.137E-06	2.269E-06	1.058E+02	-1.322E+03
9	4.500E+05	9.019E-07	1.799E-06	1.056E+02	-1.500E+03
10	5.000E+05	7.237E-07	1.444E-06	1.095E+02	-1.675E+03
11	5.500E+05	6.128E-07	1.223E-06	1.097E+02	-1.853E+03
12	6.000E+05	5.055E-07	1.009E-06	1.098E+02	-2.032E+03
13	6.500E+05	4.387E-07	8.753E-07	1.130E+02	-2.207E+03
14	7.000E+05	4.625E-07	9.228E-07	1.123E+02	-2.386E+03
15	7.500E+05	3.289E-07	6.562E-07	1.144E+02	-2.562E+03

TOTAL HARMONIC DISTORTION = 5.222123E-03 PERCENT

Wyniki takiej samej symulacji, ale dla czasu końcowego analizy `.TRAN` równego 30 μs, przedstawione są poniżej:

```
****     FOURIER ANALYSIS                 TEMPERATURE =   27.000 DEG C
*******************************************************************************
FOURIER COMPONENTS OF TRANSIENT RESPONSE V(out)
 DC COMPONENT =   2.094799E-02
```

HARMONIC NO	FREQUENCY (HZ)	FOURIER COMPONENT	NORMALIZED COMPONENT	PHASE (DEG)	NORMALIZED PHASE (DEG)
1	5.000E+04	6.773E-01	1.000E+00	5.278E-01	0.000E+00
2	1.000E+05	1.408E-02	2.078E-02	-7.435E+01	-7.541E+01
3	1.500E+05	5.332E-03	7.872E-03	-7.504E+01	-7.663E+01
4	2.000E+05	2.863E-03	4.227E-03	-7.328E+01	-7.539E+01
5	2.500E+05	1.803E-03	2.662E-03	-7.085E+01	-7.349E+01
6	3.000E+05	1.248E-03	1.842E-03	-6.817E+01	-7.134E+01
7	3.500E+05	9.191E-04	1.357E-03	-6.537E+01	-6.907E+01
8	4.000E+05	7.084E-04	1.046E-03	-6.252E+01	-6.675E+01
9	4.500E+05	5.650E-04	8.342E-04	-5.966E+01	-6.441E+01
10	5.000E+05	4.630E-04	6.835E-04	-5.681E+01	-6.209E+01
11	5.500E+05	3.877E-04	5.724E-04	-5.398E+01	-5.979E+01
12	6.000E+05	3.306E-04	4.881E-04	-5.119E+01	-5.752E+01
13	6.500E+05	2.862E-04	4.225E-04	-4.843E+01	-5.529E+01
14	7.000E+05	2.509E-04	3.705E-04	-4.572E+01	-5.311E+01
15	7.500E+05	2.225E-04	3.285E-04	-4.305E+01	-5.097E+01

TOTAL HARMONIC DISTORTION = 2.296479E+00 PERCENT

Widoczne są wyraźne różnice zarówno co do wartości składowej stałej, wartości amplitudy pierwszej harmonicznej jak i wartości zniekształceń THD (ang. *Total Harmonic Distortion*) wyliczonych w obydwu przypadkach. W pierwszym wydruku rozwinięty w szereg Fouriera został dziesiąty okres przebiegu harmonicznego widoczny na rys. 10 a w drugim przypadku pierwszy okres napięcia z tego samego przebiegu. Ponieważ, dla drugiego przypadku, przebieg nie jest jeszcze ustabilizowany (nie jest to jeszcze stan ustalony przebiegu harmonicznego), wartości składowej stałej, pierwszej i kolejnych harmonicznych jak i zniekształcenia nieliniowe THD są wyższe. Zniekształcenia THD są wyliczane jako stosunek mocy składowych harmonicznych od drugiej do ostatniej analizowanej w stosunku do pierwszej harmonicznej zgodnie ze wzorem:

$$THD = \frac{\sqrt{A_2^2 + A_3^2 + \ldots + A_n^2}}{A_1} \tag{17}$$

gdzie: A_1, A_2, ... A_n są amplitudami kolejnych harmonicznych a n jest liczbą obliczanych harmonicznych.

2.8.2. Analiza .TF

Analiza **.TF**, nazywana również analizą funkcji przenoszenia jest pochodną analizy **.DC** a jej format ogólny składni jest następujący:

```
.TF <zmienna_wyjściowa> <źródło_wejściowe>
```

Przykład:

```
.TF V([out]) vin
```

Analiza **.TF** jest pochodną analizy **.DC** i wyznacza stałoprądową, małosygnałową funkcję przenoszenia wyznaczoną wokół punktu pracy. Jako **zmienna_wyjściowa** mogą być użyte dowolne napięcia lub prądy obwodu – składnia jak dla polecenia **.PRINT**. Jako **źródło_wejściowe** musi być użyta nazwa występującego w obwodzie niezależnego źródła napięciowego lub prądowego. Wyznaczane jest wzmocnienie liczone od **źródło_wejściowe** do **zmienna_wyjściowa** oraz rezystancje wyjściowa i wejściowa. Wyniki analizy **.TF** są umieszczane wyłącznie w pliku wyjściowym „*.out”.

Przykład 5. Analiza .TF

Jako przykład analizy .TF ponownie zostanie wykorzystany obwód z rys. 6, dla którego plik wejściowy przedstawiony jest poniżej.

```
przykład nr 5, analiza .TF
*napięciowe źródło wejściowe
Vin in 0 dc 1
* obwod pasywny
L1 in 1 10mH
```

```
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analizy
.TF v([out]) vin
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

Polecenie `.TF v([out]) vin` wylicza wzmocnienie liczone od źródła napięciowego `Vin` do napięcia występującego w węźle *out*. Poniżej przedstawiony jest fragment tekstowego pliku wyjściowego „*.out” zawierający wynik analizy.

```
****     SMALL-SIGNAL CHARACTERISTICS
      V(out)/Vin =  7.500E-01
      INPUT RESISTANCE AT Vin =  4.000E+04
      OUTPUT RESISTANCE AT V(out) =  7.500E+03
```

Wyliczone powyżej wzmocnienie, które jest równe ¾, jest zgodne w przeprowadzonymi wcześniej symulacjami `.DC` i `.AC`. Dodatkowo wyznaczona została rezystancja wejściowa i wyjściowa.

2.8.3. Analiza wrażliwościowa

Kolejną analizą opartą na analizie `.DC` jest analiza wrażliwościowa a jej składnia ogólna przedstawiona jest poniżej:

```
.SENS <zmienne_wyjściowe>
```

Przykład deklaracji:

```
.SENS V([out]) v(2,1)
```

Analiza `.SENS` jest pochodną analizy `.DC` i wyznacza stałoprądowe czułości napięć i prądów w występujące w badanym układzie na zmiany wszystkich wartości elementów i parametrów modeli. Jako `zmienne_wyjściowe` mogą być użyte napięcia węzłowe i prądy źródeł napięciowych, przy czym napięcia mogą być podawane zgodnie ze składnią dla polecenia `.PRINT`. Wyniki analizy `.SENS` są umieszczane wyłącznie w pliku wyjściowym „*.out”.

Przykład 6. Analiza .SENS

Jako przykład analizy `.SENS` ponownie zostanie wykorzystany obwód z rys. 6, dla którego plik wejściowy przedstawiony jest poniżej.

```
przykład nr 6, analiza .SENS
*napięciowe źródło wejściowe
Vin in 0 dc 1
* obwod pasywny
L1 in 1 10mH
```

```
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analizy
.SENS v([out]) i(vin)
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

Polecenie `.SENS v([out]) i(vin)` wylicza stałoprądowe czułości napięcia w węźle `out` i prądu źródła `Vin` na zmiany wartości elementów układu. Poniżej przedstawiony jest fragment tekstowego pliku wyjściowego „*.out” zawierający wynik analizy:

```
**** 01/02/15 16:02:38 ******** PSpice 9.2.1 (Dec 2000) *******
 przykład nr 6, analiza .SENS
 ****     DC SENSITIVITY ANALYSIS          TEMPERATURE =   27.000 DEG C
*******************************************************************************
DC SENSITIVITIES OF OUTPUT V(out)
         ELEMENT          ELEMENT          ELEMENT        NORMALIZED
          NAME             VALUE       SENSITIVITY     SENSITIVITY
                                       (VOLTS/UNIT)  (VOLTS/PERCENT)
          R1             2.000E+04       0.000E+00       0.000E+00
          R2             1.000E+04      -1.875E-05      -1.875E-03
          R3             3.000E+04       6.250E-06       1.875E-03
          Vin            1.000E+00       7.500E-01       7.500E-03

DC SENSITIVITIES OF OUTPUT I(vin)
         ELEMENT          ELEMENT          ELEMENT        NORMALIZED
          NAME             VALUE       SENSITIVITY     SENSITIVITY
                                       (AMPS/UNIT)   (AMPS/PERCENT)
          R1             2.000E+04       0.000E+00       0.000E+00
          R2             1.000E+04       6.250E-10       6.250E-08
          R3             3.000E+04       6.250E-10       1.875E-07
          Vin            1.000E+00      -2.500E-05      -2.500E-07
```

Ponieważ analiza `.SENS` jest pochodną analizy stałoprądowej, wyłącznie rezystory są elementami powodującymi zmiany napięć i prądów w badanym układzie. Dodatkowo, ze względu na to, że R1 jest połączony równolegle z cewką L1, jego wpływ na rozpływ prądów jest zerowy.

2.8.4. Analiza szumowa

Analiza szumowa jest pochodną analizy małosygnałowej częstotliwościowej a format jej składni jest następujący:

```
.NOISE V(<węzeł1> [,<węzeł2>]) <nazwa_źródła_wejściowego> [wartość_interwału]
```

Przykłady deklaracji analizy szumowej:

```
.NOISE V([out]) vin 2
.NOSIE V(2,1) Iin
```

Analiza **.NOISE** jest pochodną analizy **.AC** i wyznacza wartości szumu pojawiające się na wyjściu jako na napięcie węźle **węzeł1** lub opcjonalnie jako napięcie pomiędzy węzłami **węzeł1** i **węzeł2** oraz wartość szumu odniesioną do wejścia widzianego jako **nazwa_źródła_wejściowego**. Dla wykonania analizy **.NOISE** musi w układzie być realizowana analiza **.AC**. Parametr **wartość_interwału** określa, co który krok analizy **.AC** wykonywana jest analiza **.NOISE**, domyślnie parametr ten jest równy 1. Pozostałe właściwości analizy szumowej są następujące:

- **nazwa_źródła_wejściowego** oznacza nazwę niezależnego źródła napięciowego lub prądowego traktowanego jako sygnał doprowadzany do wejścia analizowanego wzmacniacza,
- symulator wyznacza szum każdego elementu układu przeniesiony do wyjścia oraz szum całkowity na wejściu i wyjściu,
- wartość szumu na wyjściu wyznaczana jest dla każdej częstotliwości analizy i podawana jako gęstość widmowa napięcia skutecznego szumów w jednostkach [$V/Hz^{1/2}$],
- dla każdej analizowanej częstotliwości wyznaczony jest szum całkowity na wyjściu, wzmocnienie od wejścia do wyjścia i szum całkowity odniesiony do wejścia (poprzez podzielenie przez wzmocnienie),
- jeśli źródłem wejściowym jest niezależne źródło napięciowe jednostką szumu odniesionego do wejścia jest [$V/Hz^{1/2}$] (RMS),
- jeśli źródłem wejściowym jest niezależne źródło prądowe jednostką szumu odniesionego do wejścia jest [$A/Hz^{1/2}$] (RMS),
- jeśli w poleceniu **.NOISE** podano parametr **wartość_interwału**, wówczas w tekstowym pliku wyjściowym „*.out”, dla każdej analizowanej częstotliwości podane są: składowe cząstkowe szumu generowanego przez każdy generator szumu przeniesione do wyjścia, całkowity szum na wyjściu, wartość wzmocnienia i całkowity szum odniesiony do wejścia.

Wartości szumu wyliczane w symulatorze są gęstościami widmowymi napięcia lub prądu szumu. Aby wyznaczyć wartość skuteczną napięcia szumów w interesującym nas paśmie częstotliwości, należy obliczyć stosowną całkę:

$$V_{NOISE\_RMS} = \sqrt{\int_{f_{START}}^{f_{STOP}} V_{NOISE}^2 df} \qquad (18)$$

gdzie: f_{START} i f_{STOP} określają początek i koniec pasma częstotliwości, V^2_{NOISE} jest kwadratem badanej gęstości widmowej napięcia szumów (na wejściu lub wyjściu, w postprocesorze graficznym oznaczane jest to odpowiednio symbolami $V_{(INOISE)}$ lub $V_{(ONOISE)}$).

Przykład 7. Analiza . NOISE

Jako przykład analizy `.NOISE`, ponownie zostanie wykorzystany obwód z rys. 6, dla którego plik wejściowy przedstawiony jest poniżej.

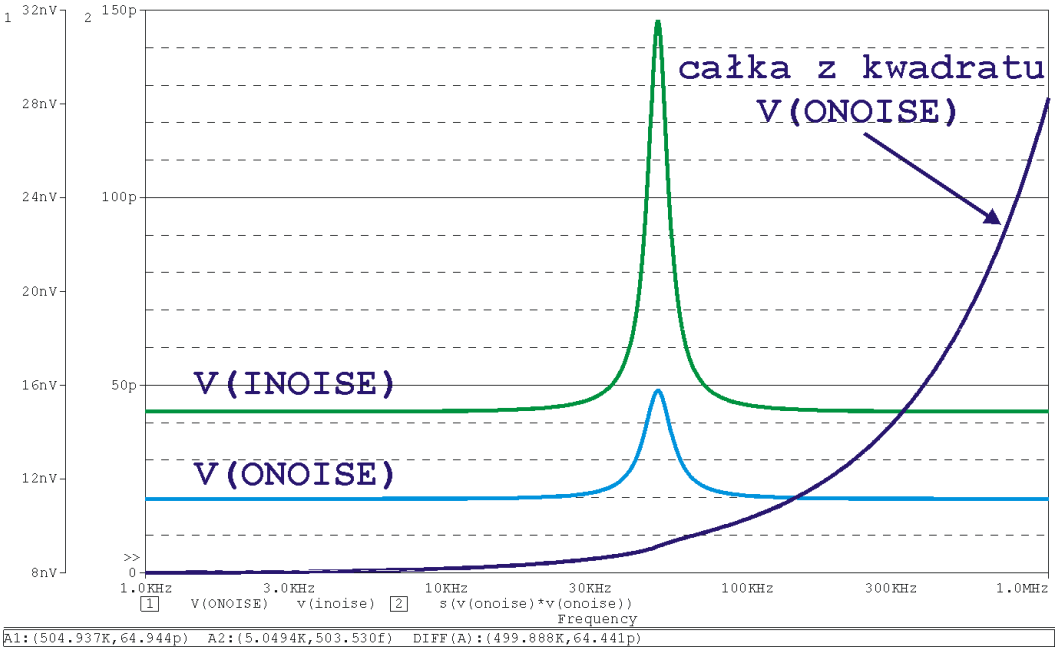
```
przykład nr 7, analiza .NOISE
*napięciowe źródło wejściowe
Vin in 0 dc 1
* obwod pasywny
L1 in 1 10mH
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 30k
*analizy
.ac dec 300 1k 1000k
.noise v([out]) vin 1
*sterowanie wyjściem i zakończenie pliku
.probe
.end
```

Polecenie `.noise v([out]) vin 1` powoduje, że dla każdej częstotliwości, dla której wykonywana jest analiza `.AC`, wykonywana jest także analiza szumowa, wyliczany jest całkowity szum widziany jako napięcie szumów w węźle `out`, wzmocnienie liczone od `Vin` do `v(out)` oraz całkowity szum odniesiony do wejścia, czyli widziany na źródle napięciowym `Vin`. Ponieważ w poleceniu `.NOISE` podano parametr `wartość_interwału` w tekstowym pliku wyjściowym, dla każdej wartości częstotliwości analizy `.AC` drukowane są wyliczone parametry szumowe. Poniżej przedstawiony jest wybrany fragment pliku dla częstotliwości rezonansowej obwodu RLC (około 50 kHz).

```
**** 01/02/15 16:16:30 ******** PSpice 9.2.1 (Dec 2000) *******
 przykład nr 7, analiza .NOISE
 ****     NOISE ANALYSIS                  TEMPERATURE =   27.000 DEG C
*******************************************************************************
     FREQUENCY =  5.050E+04 HZ
 **** RESISTOR SQUARED NOISE VOLTAGES (SQ V/HZ)
          R1          R2          R3
 TOTAL    8.281E-17   4.148E-17   1.242E-16
**** TOTAL OUTPUT NOISE VOLTAGE         =  2.485E-16 SQ V/HZ
                                        =  1.576E-08 V/RT HZ
      TRANSFER FUNCTION VALUE:
```

```
V(out)/Vin                    =  5.003E-01
EQUIVALENT INPUT NOISE AT Vin =  3.151E-08 V/RT HZ
```

Wyznaczone powyżej wzmocnienie równe ½ jest zgodne z obliczeniami z analizy `.AC` wykonanymi wcześniej i odczytanymi dla częstotliwości rezonansowej. Postprocesor graficzny programu PSPICE operujący na danych przekazanych poprzez binarny plik wyjściowy „*.dat”, również może zaprezentować dane szumowe. Poniżej na rys. 12 przedstawione są wykresy przedstawiające gęstość widmową napięcia szumów na wyjściu, gęstość widmową napięcia szumów odniesioną do wejścia oraz sposób wyznaczenia całki oznaczonej w celu obliczenia wartości skutecznej napięcia szumów. Kursor uwidoczniony u dołu rys. 12 został umieszczony na wykresie całki. Różnica z wartości kursora A1 i A2 jest wartością całki oznaczonej w paśmie od 5.05 KHz do 505 kHz. Wybrane częstotliwości to wartości od 1/10 do 10 częstotliwości rezonansowych realizowanego układu. Wartość całki oznaczonej odczytana na wykresie wynosi więc 64,41p V^2, a stąd wartość skuteczna napięcia szumów na wyjściu układu w paśmie częstotliwości wskazanym powyżej wynosi $\sqrt{64{,}441 pV^2} = 8{,}028 \mu V$.



Rys. 12. Wynik symulacji szumowej układu z rys. 6. Wykreślono gęstość widmową napięcia szumów na wyjściu V(ONOISE), gęstość widmową napięcia szumów odniesioną do wejścia V(INOISE) oraz całkę z kwadratu sygnału V(ONOISE).

2.9. Ustalenie punktu pracy i przybliżonego punktu pracy

2.9.1. Polecenie .IC

Ustalenie punktu pracy wykonuje się poprzez polecenie `.IC`, format składni jest następujący:

```
.IC < V(<węzeł> [,<węzeł>])=<wartość> >
.IC <I(<cewka>)=<wartość>>
```

Przykłady:

```
.IC V(2)=3.4 V(102)=0 V(3)=-1V I(L1)=2uAmp
.IC V(InPlus,InMinus)=1e-3 V(100,133)=5.0V
```

Polecenie ustala punkt pracy dla analizy **.AC** i analizy **.TRAN**. Do wyliczenia punktu pracy, wyspecyfikowane w poleceniu węzły są przyłączone do źródła napięciowego o rezystancji szeregowej równej 0.0002 Ω i wartości wydajności podanej w poleceniu **.IC**. Po wyliczeniu punktu pracy, wyspecyfikowane węzły są odłączane od źródeł napięciowych. Dalsze właściwości polecenia **.IC**:

- w przypadku równoczesnego wystąpienia polecenia **.IC** i **.NODESET** dla tego samego węzła, używane jest polecenie **.IC**,
- można podawać wartości napięć w węzłach, wartości napięć występujące pomiędzy węzłami lub prądy cewek,
- nie można ustalać niezerowej wartości napięcia dla cewek, gdyż dla wyliczania punktu pracy są one zwarciami,
- dla cewek można natomiast ustalić niezerowy prąd.

2.9.2. Polecenie .NODESET

Polecenie **.NODESET** służy do ustalenia punktu startowego do obliczenia właściwego punktu pracy. Polecenie to jest przydatne w 2 następujących przypadkach:

- symulowany jest układ o więcej niż jednym punkcie stabilnym i pożądane jest aby wyznaczony przez symulator punkt pracy był tym oczekiwanym punktem, przykłady takich układów dla których warto jest ustalić stan początkowy to przerzutnik i rejestr,
- symulator ma trudności z obliczeniem punktu pracy, w takim przypadku start symulacji od napięć bliskich finalnemu rozwiązaniu znacznie przyspiesza obliczenie punktu pracy a w niektórych przypadkach jest to jedyna możliwość aby symulator w ogóle go wyznaczył, takie trudności ze znalezieniem punktu pracy symulator PSPICE szczególnie często wykazuje dla układów ze sprzężeniem zwrotnym.

Format ogólny polecenia **.NODESET** jest następujący:

```
.NODESET < V(<węzeł> [,<węzeł>])=<wartość> >
.NODESET <I(<cewka>)=<wartość>>
```

Przykłady:

```
.NODESET V(2)=3.4 V(102)=0 V(3)=-1V I(L1)=2uAmp
.NODESET V(InPlus,InMinus)=1e-3 V(100,133)=5.0V
```

Polecenie pomaga w ustaleniu punktu pracy poprzez podanie początkowych wartości napięć węzłowych i prądów cewek, które służą tylko jako punkt wyjściowy do obliczenia właściwego punktu

pracy. Można podać wszystkie napięcia węzłowe lub tylko niektóre. Można również podawać napięcie pomiędzy węzłami.

2.10. Operacje na plikach

W symulatorze PSPICE dostępne są 4 następujące operacje na plikach:

- **.INC** - polecenie włączenia pliku tekstowego,
- **.LIB** - polecenie odniesienia się do biblioteki,
- **.SAVEBIAS** – zapisanie punktu polaryzacji,
- **.LOADBIAS** – wczytanie punktu polaryzacji.

2.10.1. Polecenie. INC

Polecenie powoduje wczytanie, w miejscu wystąpienia polecenia, innego pliku tekstowego. Format ogólny polecenia jest następujący:

```
.INC <nazwa_pliku>
```

Przykłady:

```
.INC "SETUP.CIR"
.INC SETUP.CIR
.INC "C:\LIB\VCO.CIR"
```

We wczytywanym pliku należy zachować składnię zgodną ze standardem SPICE. Plik wczytywany nie powinien mieć linii tytułu, chyba że będzie zakomentowana. Wczytywanie może być zagnieżdżone do 4 poziomów.

2.10.2. Polecenie .LIB

Polecenie powoduje wczytanie plików zawierających biblioteki PSPICE. Format ogólny składni jest następujący:

```
.LIB [nazwa_pliku]
```

Przykłady:

```
.LIB
.LIB linear.lib
.LIB "C:\lib\bipolar.lib"
```

W przypadku braku podania nazwy pliku wczytywana jest biblioteka ”nom.lib”. Plik ten zawiera odniesienia do elementów ze standardowej biblioteki firmy ORCAD. Z plikiem bibliotecznym związany jest plik indeksowy wykorzystywany do szybkiego odnalezienia modelu w dużych plikach. Plik indeksowy tworzony jest automatycznie przez symulator. Plik biblioteczny powinien mieć rozszerzenie „*.lib” a jego zawartość jest ograniczona do:

- linii z komentarzem,

- poleceń `.MODEL` z definicjami parametrów elementów,
- poleceń definicji podukładów (`.SUBCKT`, `.ENDS`),
- definicji parametrów `.PARAM`,
- definicji funkcji `.FUNC`,
- poleceń `.LIB`.

2.10.3. Polecenie .SAVEBIAS

Polecenie `.SAVEBIAS` zapisuje bieżące wartości napięć węzłowych analizowanego obwodu i prądów cewek z użyciem polecenia `.NODESET`. Format ogólny polecenia jest następujący:

```
.SAVEBIAS <"nazwa_pliku"> <[OP] [TRAN] [DC]> [NOSUBCKT]
+ [TIME=<wartość> [REPEAT]] [TEMP=<wartość>]
+ [STEP=<wartość>] [MCRUN=<wartość>] [DC=<wartość>]
+ [DC1=<wartość>] [DC2=<wartość>]
```

Ze względu na dość rozbudowane możliwości polecenia najważniejsze funkcje zostaną omówione za pomocą przykładów:

`.SAVEBIAS "OPPOINT" OP` – zapisanie punktu pracy (AC i OP)

`.SAVEBIAS "TRANDATA.BSP" TRAN NOSUBCKT TIME=10u` – zapisanie punktu z analizy czasowej z czasu najbliższego 10 μs (ale nie mniejszego niż 10 μs).

`.SAVEBIAS "SAVETRAN.BSP" TRAN TIME=5n REPEAT` – zapisanie punktu co każde 5 ns dla przebiegu obliczeniowego, poprzednie wyniki są nadpisywane.

`.SAVEBIAS "SAVEDC.BSP" DC MCRUN=3 DC1=3.5 DC2=100` – zapisanie punktu dla analizy `.DC` jeśli równocześnie spełnione są warunki: pierwsze przemiatanie DC równe 3.5, drugie przemiatanie DC równe 100 oraz 3 przebieg analizy Monte Carlo. Jeśli jest używane tylko jedno przemiatanie stałoprądowe można zamiast `DC1` użyć wyrażenia `DC`.

Niektóre argumenty i opcje polecenia:

- nazwa pliku musi być w podwójnym cudzysłowie,
- opcja `NOSUBCKT` powoduje brak zapisu napięć węzłowych i prądów cewek dla podukładów,
- opcje `TIME=<wartość> [REPEAT]` są używane dla punktów czasowych zapisów dla analizy czasowej, jeśli użyte jest `REPEAT` następuje nadpisywane wyników więc tylko ostatnie wartości są dostępne,
- `TEMP=<wartość>` - definiuje temperatury, dla których dokonywany jest zapis, i krok `STEP=<wartość>,`
- `MCRUN=<wartość>` - definiuje numer analizy Monte Carlo lub *Worst Case*, dla której dokonywany jest zapis,

- **DC=<wartość> DC1=<wartość> DC2=<wartość>** – używa się do specyfikacji kroku analizy **.DC**, który ma być zapisany.

Uwagi:

- jeśli nie użyto **REPEAT** zapis dokonywany jest jednokrotnie dla wartości równej lub większej niż **TIME**, jeśli użyto **REPEAT** wówczas zapisany jest tylko ostatni punkt o czasie równym całkowitej wieloktrotności **TIME**,
- dla niezagnieżdżonych analiz stałoprądowych należy używać **DC** zamiast **DC1** i **DC2**,
- w pliku zapisywane są: napięcia węzłowe i prądy cewek.

2.10.4. Polecenie .LOADBIAS

Polecenie **.LOADBIAS** służy do wczytania pliku, w którym powinno być polecenie **.NODESET**. Format ogólny polecenia jest następujący:

```
.LOADBIAS <”nazwa_pliku”>
```

Przykłady:

```
.LOADBIAS "OPPOINT.OP"
```

Nazwa pliku musi być w cudzysłowie. Wczytywane są przybliżone wartości napięć węzłowych i prądów cewek. Wczytywany plik musi zawierać polecenie **.NODESET**. Zawartość pliku można wygenerować poleceniem **.SAVEBIAS** lub wykonać ręcznie.

Uwagi co do stosowania pary poleceń **.SAVEBIAS** i **.LOADBIAS**:

- polecenia mogą być użyte do skrócenia czasu wyznaczania punktu pracy dla dużych układów,
- można też użyć poleceń do ustalenia stanu układów pamięciowych np. przerzutników,
- oba polecenia, jeśli użyte w jednym pliku wejściowym, nie mogą odnosić się do tego samego pliku,
- można użyć ww. poleceń również w przypadku problemów ze zbieżnością obliczeń.

2.11. Wybrane elementy półprzewodnikowe

W niniejszym opracowaniu, opisano wykorzystanie elementów półprzewodnikowych z punktu widzenia użytkownika symulatora, to znaczy opisano jak je umieścić w obwodzie i wykonać żądaną symulację. Nie umieszczono tu szczegółowych opisów matematycznych modeli używanych w symulatorze PSPICE. Zainteresowanych tym tematem czytelników odsyłamy do stosownej literatury [3]. Modele elementów półprzewodnikowych dostosowywane są przez producentów elementów i układów elektronicznych tak aby jak najwierniej oddawały ich rzeczywiste właściwości. Użytkownik symulatora PSPICE (jak i każdego innego) powinien zadbać aby użyty model był odpowiedni. Zazwyczaj wykonuje się to poprzez pobranie modelu danego elementu ze strony producenta elementu, który

mamy zamiar użyć w docelowym układzie. Często modele elementów są dokumentami poufnymi i dostęp do nich może być uzależniony od uzyskania odpowiedniej zgody właściciela/dystrybutora.

2.11.1. Dioda półprzewodnikowa

Wstawienie diody półprzewodnikowej do symulowanego obwodu wykonywane jest zgodnie z następującą składnią:

```
D<nazwa> <węzeł+> <węzeł-> <nazwa_modelu> [powierzchnia]
```

Przykłady wstawienia diody:

```
DCLAMP 14  0 DMOD
D13 15 17 SWITCH 1.5
```

Opis modelu diody:

```
.MODEL <nazwa_modelu> D [parametry_modelu]
```

Anoda diody to `węzeł+` natomiast katoda to `węzeł-`. Parametr `powierzchnia` skaluje niektóre parametry modelu diody, jego domyślna wartość wynosi 1.

2.11.2. Tranzystor bipolarny

Wstawienie tranzystora bipolarnego do symulowanego obwodu jest wykonywane zgodne z następującą składnią:

```
Q<nazwa> <kolektor> <baza> <emiter> [podłoże] <nazwa_modelu> [powierzchnia]
```

Przykłady wstawienia tranzystorów bipolarnych do badanego układu:

```
Q1 14 2 13 PNPNOM
Q13 15 3  0 1 NPNSTRONG 1.5
Q7 VC 5 12 [SUB] LATPNPDCLAMP 14  0 DMOD
```

Opisy modeli tranzystorów bipolarnych:

```
.MODEL <nazwa_modelu> NPN [parametry_modelu]
.MODEL <nazwa_modelu> PNP [parametry_modelu]
.MODEL <nazwa_modelu> LPNP [parametry_modelu]
```

Argumenty modelu:

- `[podłoże]` - opcjonalny węzeł podłączenia podłoża umieszczany w nawiasach `[]`, stosowany wyłącznie dla bocznych tranzystorów PNP,
- `powierzchnia` – względna powierzchnia elementu, skaluje niektóre parametry modelu, jego domyślna wartość wynosi 1.

2.11.3. Tranzystor MOS

Wstawienie tranzystora MOS do symulowanego obwodu jest wykonywane zgodne z następującą składnią:

```
M<nazwa> <dren> <bramka> <źródło> <podłoże> <model>
```

```
+ [L=<value>] [W=<value>]
+ [AD=<value>] [AS=<value>]
+ [PD=<value>] [PS=<value>]
+ [NRD=<value>] [NRS=<value>]
+ [NRG=<value>] [NRB=<value>]
+ [M=<value>] [N=<value>]
```

Przykłady wstawienia tranzystorów MOS do badanego układu:

```
M16 17 3 0 0 PSTRONG M=2
M28 0 2 100 100 NWEAK L=33u W=12u
+ AD=288p AS=288p PD=60u PS=60u NRD=14 NRS=24 NRG=10
```

Ogólne opisy modeli:

```
.MODEL <nazwa_modelu> NMOS [parametry_modelu]
.MODEL <nazwa_modelu> PMOS [parametry_modelu]
```

Argumenty i opcje stosowane w czasie wstawiania tranzystora MOS są następujące:

- **L**, **W** – długość i szerokość kanału tranzystora, parametry te są modyfikowane i w modelu obliczana jest efektywna długość kanału, można te parametry podać również w definicji modelu **.MODEL** lub poprzez parametry poleceniem **.OPTIONS**, wartość podana przy wywołaniu elementu wypiera wartość zdeklarowaną w poleceniu **.MODEL**, która to wypiera wartość z polecenia **.OPTIONS**,
- jeśli nie podano **W** lub **L** wartość domyślna wynosi 100 μm,
- **AS**, **AD** – powierzchnie obszaru źródła i drenu, wartość domyślna może być ustawiona w poleceniu **.OPTIONS**, jeśli nie została ustawiona, to wartość domyślna wynosi 0,
- **PS**, **PD** – obwody obszaru źródła i drenu, wartość domyślna wynosi 0,
- **NRS**, **NRD**, **NRG**, **NRB** – mnożniki rezystancji szeregowych wyprowadzeń tranzystora odpowiednio dla: źródła, drenu, bramki i podłoża, wartość domyślna wynosi 0, mnożony parametr to RSH,
- **M(NP)** – mnożnik ilości równolegle połączonych tranzystorów MOS, wartość domyślna wynosi 1, **NP** oznacza to samo co **M**,
- **N(NS)** – mnożnik długości kanału tranzystora, ważny tylko dla modeli MOS LEVEL=5, wartość domyślna wynosi 1, **NS** oznacza to samo co **N**.

PSPICE obsługuje 7 poziomów modeli MOS:

- LEVEL=1 – model Shichman-Hodges,
- LEVEL=2 – model geometryczny,
- LEVEL=3 – model półempiryczny,
- LEVEL=4 – model BSIM,
- LEVEL=5 – model EKV,
- LEVEL=6 – model BSIM3 v2.0,

- LEVEL=7 – model BSIM3 v3.1.

2.12. Źródła sterowane napięciem

W symulatorze PSPICE dostępne są 2 źródła sterowane napięciem: źródło napięciowe (E) i prądowe (G). Użycie takiego źródła może być wykonane w sposób podstawowy i jest to wtedy idealne, liniowe źródło sterowane lub w sposób rozszerzony i wtedy rzeczywista wydajność takiego źródła określona jest w sposób bardziej złożony. Format ogólny dla sterowania podstawowego jest następujący:

```
E<nazwa> <węzeł+> <węzeł-> <węzeł_sterujący+> <węzeł_sterujący-> <wzmocnienie>
```

Przykłady:

```
E1 1 2 3 4 12
Gq8 3 4 5 6 10uS
```

Pierwsza litera w linii **E** oznacza źródło napięciowe sterowane napięciem, **G** oznacza źródło prądowe sterowane napięciem. Dla źródła **E** wzmocnienie wyrażone jest w [V/V], dla **G** wzmocnienie wyrażone jest w [S]. W pierwszym przykładzie przedstawionym powyżej źródło o nazwie **E1** wytwarza napięcie o wartości równej różnicy napięć pomiędzy węzłami **3** i **4** pomnożonymi przez **12** a wynikowe napięcie podane jest pomiędzy węzły **1** i **2**. W drugim przykładzie mamy źródło prądowe o nazwie **Gq8** włączone pomiędzy węzły **3** i **4** a jego wydajność jest równa różnicy napięć pomiędzy węzłami **5** i **6** pomnożonemu przez współczynnik równy 10 μS.

Oprócz napięciowego źródła sterowanego w sposób podstawowy, dostępne są następujące formy sterowania rozszerzonego:

- **POLY** – deklaracja wielomianu sterującego,
- **VALUE** – deklaracja wyrażenia arytmetycznego sterującego wydajnością źródła,
- **TABLE** – deklaracja funkcji tabelarycznej,
- **LAPLACE** – deklaracja transformaty Laplace’a,
- **CHEBYSHEV** – deklaracja wielomianu Czebyszewa,
- **FREQ** – deklaracja funkcji częstotliwościowej.

Przykłady rozszerzone:

```
EAMP 13 0 POLY(1) 26  0 0 500
ENONLIN 100 101 POLY(2) 3 0  4 0  0.0 13.6 0.2 0.005
ESQROOT   5   0 VALUE = {5V*SQRT(V(3,2))}
```

Powyższy przykład daje możliwość budowy źródła o wydajności wyrażonej za pomocą równania lub funkcji. Źródło **ESQROOT** wytwarza napięcie pomiędzy węzłami **5** i **0** o wartości równej pięciu pierwiastkom z różnicy napięć pomiędzy węzłami **3** i **2**: $V(5,0) = 5\sqrt{V(3,2)}$. Inne przykłady sterowania rozszerzonego:

```
ET2 2 0 TABLE {V(ANODE,CATHODE)} = (0,0) (30,1)
```

```
ERC 5 0 LAPLACE {V(10)} = {1/(1+.001*s)}
ELOWPASS 5 0 FREQ {V(10)}=(0,0,0)(5kHz, 0,0)(6kHz -60, 0) DELAY=3.2ms
ELOWPASS 5 0 CHEBYSHEV {V(10)} = LP 800 1.2K .1dB 50dB
```

Szczegóły powyższych rodzajów sterowania można znaleźć w literaturze [2].

2.13. Źródła sterowane prądem

Format ogólny wstawienia podstawowego źródła sterowanego prądem jest następujący:

```
H<nazwa> <węzeł+> <węzeł-> <nazwa_napięciowego_źródła_sterującego> <wzmocnienie>
```

Przykłady:

```
H1 1 2 vin 10
FSENSE 1 2 VSENSE 3.3
```

Litera `H` oznacza źródło napięciowe sterowane prądem, litera `F` oznacza źródło prądowe sterowane prądem. Dla źródła typu `H` wzmocnienie jest w [Ω], dla `F` wzmocnienie wyrażone jest w [A/A]. Pierwszy powyższy przykład daje w rezultacie źródło napięciowe wstawione pomiędzy węzły `1` i `2` o wydajności równej prądowi płynącemu przez źródło napięciowe `Vin` pomnożonemu przez współczynnik równy 10 Ω. Drugi przykład to źródło prądowe wstawione pomiędzy węzły `1` i `2` o wydajności równej prądowi przepływającemu przez źródło napięciowe `VSENSE` pomnożonemu prze współczynnik równy `3,3`. W przypadku źródeł sterowanych prądem jedynie prąd przepływający przez niezależne źródło napięciowe może nim sterować. W przypadku, gdy jest potrzeba, aby prąd płynący przez inny element był źródłem sygnału sterowania, należy do tej gałęzi wstawić źródło napięciowe o zerowej wydajności i prąd tego źródła wykorzystać do sterowania. Dla źródeł sterowanych prądem jedyną możliwością sterowania rozszerzonego jest wykorzystanie wielomianu – `POLY`.

2.14. Podukłady, deklaracja i wstawienie

Możliwość zdeklarowania podukładu a następnie wielokrotnego jego użycia, stanowi podstawę dzisiejszych projektów o dużej skali integracji. Używanie podukładów umożliwia wykonanie projektów hierarchicznych, podział dużych projektów na odrębne podbloki oraz równoległą pracę różnych zespołów projektowych nad odrębnymi fragmentami większego projektu. Dodatkowo możliwe jest wstępne modelowanie podukładu za pomocą ogólnych równań a następnie wymianę podbloków na modelowane bardziej szczegółowo, jak już będą one dostępne. Projektowanie hierarchiczne składa się z dwóch podstawowych etapów: deklaracji podukładów oraz ich wstawienia.

2.14.1. Deklaracja podukładu

Format ogólny deklaracji podukładu jest następujący:

```
.SUBCKT <nazwa> [węzły]
+ [OPTIONAL: < <węzeł_opcjonalny> = <węzeł_domyślny>>]
```

```
+ [PARAMS: < <nazwa_parametru> = <wartość>>]
+ [TEXT: < <nazwa> = <wartość_tekstowa>>]
...
.ENDS
```

Przykłady:

```
.SUBCKT OPAMP 1 2 101 102 17
...
.ENDS
.SUBCKT FILTER INPUT, OUTPUT PARAMS: CENTER=100kHz,
+ BANDWIDTH=10kHz
...
.ENDS
.SUBCKT PLD IN1 IN2 IN3 OUT1
+  PARAMS: MNTYMXDLY=0 IO_LEVEL=0
+  TEXT: JEDEC_FILE="PROG.JED"
...
.ENDS
.SUBCKT  74LS00  A B Y
+  OPTIONAL: DPWR=$G_DPWR DGND=$G_DGND
+  PARAMS: MNTYMXDLY=0 IO_LEVEL=0
...
.ENDS
```

Powyższe przykłady przedstawiają wyłącznie nagłówek i zakończenie deklaracji podukładu, czyli elementy istotne z punktu widzenia późniejszego wykorzystania danego podukładu. Wielokropek powinien być zastąpiony elementami wchodzącymi w skład danego podukładu. Argumenty i opcje stosowane przy deklaracjach podukładów są następujące:

- **węzły** - lista węzłów (formalnych) łączących podukład z otoczeniem (może być pusta),
- **OPTIONAL** – umożliwia specyfikację opcjonalnych węzłów połączeniowych,
- **PARAMS** – umożliwia specyfikację opcjonalnych parametrów przekazywanych do podukładu,
- **TEXT** – umożliwia specyfikację opcjonalnych parametrów tekstowych przekazywanych do podukładu (używane tylko w symulacji układów cyfrowych),
- **.ENDS** – oznacza koniec definicji podukładu.

Zasady deklaracji i użycia podukładów są następujące:

- przywołanie podukładu następuje poprzez użycie litery **X** jako typu wstawianego komponentu,
- przy powoływaniu podukładu liczba węzłów musi być taka sama jak w deklaracji,
- przywołanie podukładu powoduje zamianę nazw formalnych węzłów na nazwy aktualne (takie jak w przywołaniu),

- nie używa się węzła „0” na liście węzłów, ten węzeł jest globalną masą zawsze widzianą zarówno w układzie głównym jak i w podukładzie,
- węzły opcjonalne mogą, ale nie muszą, być specyfikowane przy wywołaniu podukładu, jeśli nie są specyfikowane wówczas użyte zostaną połączenia domyślne,
- węzły opcjonalne są przydatne do podłączania zasilania np. w bramkach cyfrowych,
- przywołanie (**X**) podukładu może być zagnieżdżone,
- deklaracja (**.SUBCKT**, **.ENDS**) podukładu nie może być zagnieżdżona.

Wewnątrz definicji podukładu (czyli pomiędzy **.SUBCKT** i **.ENDS**) można używać następującej składni PSPICE:

- wstawiania elementów,
- polecenia **.IC** (punkt pracy),
- polecenia **.NODESET** (przybliżony punkt pracy),
- polecenia **.MODEL** (definicja modelu),
- polecenia **.PARAM** (deklaracja parametru),
- polecenia **.FUNC** (deklaracje funkcji).

Podukłady i układ główny – właściwości:

- modele, parametry i funkcje zdefiniowane w podukładzie są dostępne tylko w tym podukładzie,
- modele, parametry i funkcje zdefiniowane w głównym układzie są dostępne w tym układzie i wszystkich podukładach,
- nazwy węzłów, elementów i modeli są lokalne w podukładach i takie same nazwy mogą być używane w głównym układzie,
- nazwy węzłów, elementów w podukładach są widziane jako np. **X1.Q13** (element Q13 w podukładzie X1) lub **X4.123** (węzeł 123 w podukładzie X4).

2.14.2. Wstawienie podukładu

Format ogólny:

```
X<nazwa> [węzły] <nazwa_podukładu> [PARAMS: +<<nazwa_parametu> = <wartość>>]
+ [TEXT: < <nazwa_par_text> = <text>>]
```

Proste przykłady wstawienia podukładów:

```
XFOLLOW IN OUT VCC VEE OUT OPAMP
XFELT  1  2  FILTER PARAMS: CENTER=200kHz
```

W układzie wstawianym musi być taka sama liczba węzłów jak w układzie zdeklarowanym. Wstawianie podukładów może być zagnieżdżone ale nie może być zapętlone. Jeśli użyto opcji **PARAMS:** wówczas wyszczególnione wartości parametrów zostaną zamienione z wartości w definicji podukładu do wartości bieżących. Opcja **TEXT:** jest używana w symulacji cyfrowej.

Poniżej zaprezentowane są cztery przykłady wstawienia podukładu, zakładamy następującą wcześniejszą deklarację wstawianego podukładu:

```
.SUBCKT  74LS00  A B Y
+  OPTIONAL: DPWR=$G_DPWR DGND=$G_DGND
+  PARAMS: MNTYMXDLY=0 IO_LEVEL=0
...
.ENDS
```

Przykłady wstawienia powyższego podukładu:

`X1 IN1 IN2 OUT 74LS00` - użyte są domyślne węzły połączenia do zasilania `$G_DPWR` oraz `$G_DGND`,

`X2 IN1 IN2 OUT MYPOWER MYGROUND 74LS00` - użyte są węzły połączenia do zasilania `MYPOWER` oraz `MYGROUND`,

`X3 IN1 IN2 OUT MYPOWER 74LS00` - użyty jest jeden i pierwszy węzeł opcjonalny połączenia do zasilania `MYPOWER` oraz domyślny `$G_DGND`,

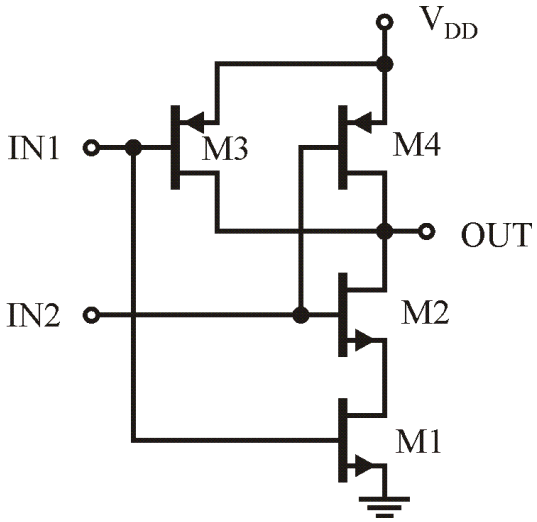
`X4 IN1 IN2 OUT $G_DPWR MYGROUND 74LS00` – jeśli trzeba użyć drugiego węzła opcjonalnego wówczas należy wstawić również ten pierwszy.

Przykład 8. Deklaracja i użycie podukładów

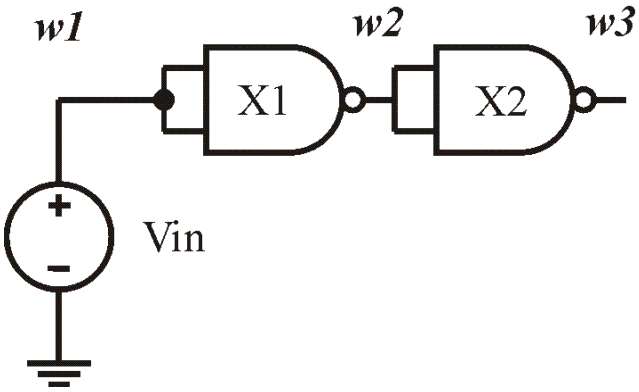
W ramach przykładu, najpierw zdeklarowany zostanie podukład zawierający bramkę typu NAND składającą się z tranzystorów CMOS, a następnie podukład ten zostanie wykorzystany do symulacji dwóch kaskadowo połączonych bramek. W przykładzie założono, że tranzystory MOS mają modele o nazwach `nfet` oraz `pfet` i modele te są umieszczone w pliku o nazwie `scn05mos59jmagic.lib`. Schemat bramki NAND przedstawiony jest na rys. 13. Poniżej przedstawiono deklarację podukładu:

```
* Zdeklarowanie podukładu - bramka NAND
*    NAZWA    węzły połączeniowe
*      |      ||||||||||||||||
.sub nand      in1 in2 out vdd
M1  10 in1   0   0 nfet W=0.9u L=0.6u
M2 out in2  10   0 nfet W=0.9u L=0.6u
M3 out in1 vdd vdd pfet W=0.9u L=0.6u
M4 out in2 vdd vdd pfet W=0.9u L=0.6u
.ends
```

Nadana nazwa podukładu to `nand`. Węzły połączeniowe zdeklarowane są w kolejności: `in1 in2 out vdd`, później w czasie wstawiania podukładu należy zachować tą samą kolejność. Dodatkowo w podukładzie występuje węzeł wewnętrzny (nie podłączany do sygnałów za zewnątrz podukładu) o nazwie `10`. Na liście połączeniowej nie występuje węzeł masy (0), który jest dostępny zawsze w podukładzie. Deklaracja podukładu może rozpoczynać się zarówno słowem kluczowym `.sub` jak i `.subckt`, koniec podukładu jest deklarowany słowem kluczowym `.ends`.



Rys. 13. Schemat bramki NAND CMOS. Węzeł pomiędzy drenem M1 a źródłem M2 nazwany został „10”. Należy przyjąć wymiary wszystkich tranzystorów równe W=0,9 µm L=0,6 µm, podłoża tranzystorów typu p należy podłączyć do węzła vdd a podłoża tranzystorów typu n do masy.



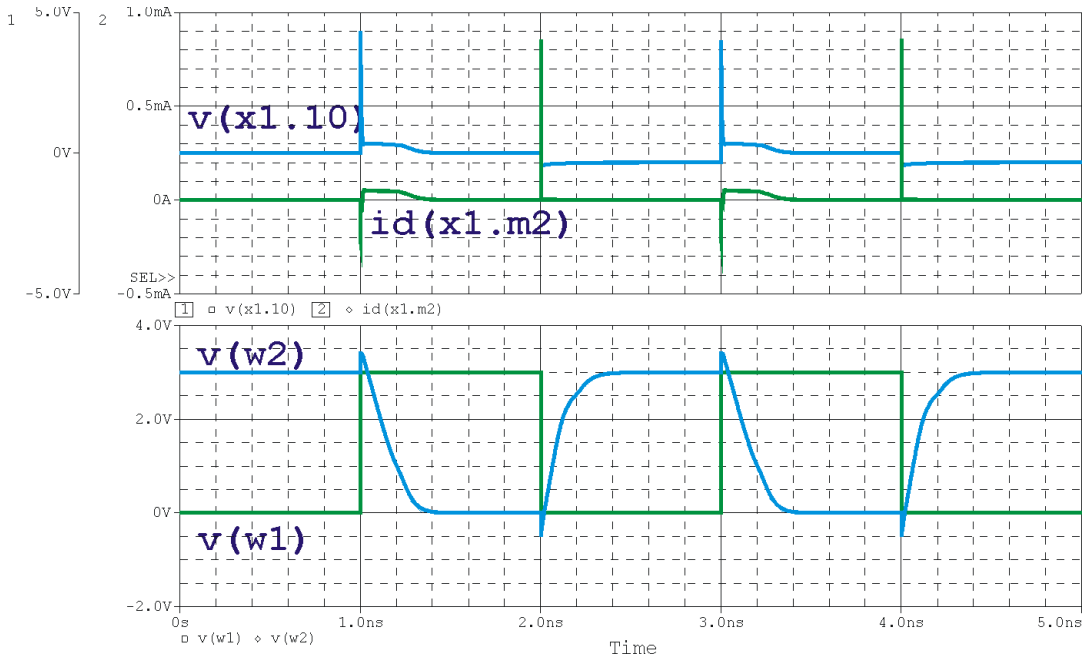
Rys. 14. Schemat połączenia dwóch bramek NAND z rys. 13 w celu zbadania czasu propagacji bramki X1.

Pełny kod PSPICE, w ramach którego dokonywane jest badanie czasów propagacji bramki X1 obciążonej bramką X2, wykonanego zgodnie z rys. 14, przedstawiony jest poniżej:

```
Przykład 8, deklaracja i wstawienie podukładu
*Zasilanie
Vdd vdd 0 3
*Sygnał wejściowy
Vin w1 0 dc 0 pulse(0 3 1n 1p 1p 1n 2n )
* Badany układ 2 bramek jako inwerterów
*  węzły połączeniowe    NAZWA
*  |||||||||||||||||      |
X1 w1   w1   w2   vdd    nand
X2 w2   w2   w3   vdd    nand
*Zdefiniowanie bramki NAND
*    NAZWA    węzły połączeniowe
*      |      ||||||||||||||||
.sub nand      in1 in2 out vdd
```

```
M1  10 in1   0   0 nfet W=0.9u L=0.6u
M2 out in2  10   0 nfet W=0.9u L=0.6u
M3 out in1 vdd vdd pfet W=0.9u L=0.6u
M4 out in2 vdd vdd pfet W=0.9u L=0.6u
.ends
*analiza
.tran 5p 5n 0n 5p
* wczytanie modeli z pliku
.inc scn05mos59jmagic.lib
.probe
.end
```

Jako sygnał wejściowy `Vin` podany jest przebieg prostokątny o częstotliwości 500 MHz. Należy zwrócić uwagę, że kolejność parametrów przy deklaracji podukładu jest następująca: nazwa podukładu - lista węzłów połączeniowych a przy wstawianiu podukładu jest odwrotna: lista węzłów połączeniowych – nazwa podukładu. Węzły formalne podukładu X1 o nazwach `in1`, `in2`, `out`, `vdd`, zgodnie z wstawieniem tego podukładu, zamieniają się odpowiednio na węzły `w1`, `w1`, `w2`, `vdd`. Dla podukładu X2, węzły o nazwach `in1`, `in2`, `out`, `vdd` zamieniają się odpowiednio na węzły `w2`, `w2`, `w3`, `vdd`. Węzły wewnętrzne o nazwie `10` przyjmują nazwy hierarchiczne, odpowiednio: `X1.10` i `X2.10`. Należy zwrócić uwagę, że nie ma znaczenia kolejność wstawienia podukładu i jego deklaracji, w analizowanym przykładzie najpierw jest wykonane wstawienie podukładów a dopiero później jest wykonana deklaracja podukładu. Jako przykład, zlecono do wykonania symulacje czasowe a ich wyniki przedstawiono na rys. 15.



Rys. 15. Wyniki symulacji układu z rys. 14. Wykres górny przedstawia dostęp do sygnałów występujących wewnątrz podukładów. Wykres dolny przedstawia przebiegi na wejściu i wyjściu bramki X1, z których można odczytać czasy propagacji bramki.

2.15. Deklaracja parametru

Parametr deklarowany jest zgodnie z następującym formatem ogólnym:

```
.PARAM < <nazwa> = <wartość> >
.PARAM < <nazwa> = { <wyrażenie> } >
```

Przykłady:

```
.PARAM VCC = 12V, VEE = -12V
.PARAM BANDWIDTH = {100kHz/3}
.PARAM PI = 3.14159, TWO_PI = {2*3.14159}
.PARAM VNUM = {2*TWO_PI}
```

Nazwa nie może być nazwą predefiniowaną (np.: **TEMP**, **VT**, **GMIN**). Wartość musi być podana za pomocą stałej lub wyrażenia. W wyrażeniu można używać stałych lub innych parametrów. Właściwości i używanie parametrów:

- kolejność deklaracji parametrów nie ma znaczenia,
- można definiować parametry w podukładzie, będą one widoczne wówczas tylko w tym podukładzie,
- parametry można używać w miejsce stałych z następującymi wyjątkami: współczynniki **TC1** i **TC2** rezystorów w linii ich deklaracji, wartości specyfikacji niezależnego źródła napięciowego i prądowego typu **PWL**, współczynniki **POLY** dla elementów **E**, **G**, **F**, **H**, nazwy węzłów układu, wartości parametrów analiz **.TRAN** i **.AC** i innych.

2.16. Analiza parametryczna

Analiza parametryczna może być zlecona zgodnie z jednym z trzech możliwych formatów:

```
.STEP [LIN] <zmienna> <początek> <koniec> <inkrement>
.STEP [DEC|OCT] <zmienna> <początek> <koniec> +<liczba_punktów_na_dekadę/oktawę>
.STEP <zmienna> LIST <lista wartości>
```

Polecenie `.STEP` wykonuje przemiatanie zmiennej dla wszystkich rodzajów analiz zleconych do wykonania w układzie. W plikach wyjściowych zapisywane są wyniki dla każdej wartości przemiatanej zmiennej. Przemiatanie może być wykonane liniowo (parametr `LIN`), logarytmicznie (parametr `OCT` lub `LOG`) lub można podać listę wartości (parametr `LIST`). Parametr `LIN` jest domyślny i można go nie podawać. Po użyciu parametru `LIST`, lista wartości musi być narastająca lub opadająca.

Przykłady zastosowania analizy parametrycznej:

`.STEP VCE 0V 10V .5V` - liniowe przemiatanie `VCE` co 0.5V od 0V do 10V,

`.STEP LIN I2 5mA -2mA 0.1mA` - liniowe przemiatanie `I2`,

`.STEP RES RMOD(R) 0.9 1.1 .001` - liniowe przemiatanie parametru `R` modelu `RMOD` typu `RES` od wartości 0.9 do 1.1 co 0.001,

`.STEP DEC NPN QFAST(IS) 1E-18 1E-14 5` - logarytmiczne przemiatanie parametru `IS` modelu o nazwie `QFAST` typu `NPN` od wartości 1e-18 do 1e-14 z 5 punktami na każdą dekadę,

`.STEP TEMP LIST 0 20 27 50 80 100` – lista temperatur pracy układu,

`.STEP PARAM CenterFreq 9.5kHz 10.5kHz 50Hz` - liniowe przemiatanie parametru `CenterFreq`.

Podsumowanie możliwych do przemiatania zmiennych przedstawione jest w tab. 9.

Tab. 9 Zmienne możliwe do przemiatania w analizie parametrycznej `.STEP`.

Zmienna	Sposób zapisu	Znaczenie
źródło	nazwa niezależnego źródła napięciowego lub prądowego	podczas przemiatania zmienia się wydajność źródła
parametr modelu	typ i nazwa modelu oraz nazwa parametru w nawiasie	zmieniany jest wyszczególniony parametr modelu, nie można zmieniać niektórych parametrów modeli, są to: `W` i `L` dla tranzystora MOS, oraz parametry temperaturowe.
temperatura	słowo `TEMP`	zmieniana jest bieżąca temperatura symulacji
parametr globalny	słowo kluczowe `PARAM` i nazwa parametru	zmieniana jest wartość parametru, wszystkie wartości zależne od tego parametru są wyliczane na nowo

Przykład 9. Rozwinięcie przykładu nr 1, wykreślenie rodziny charakterystyk przy wykorzystaniu analizy parametrycznej `.STEP`

Jako zadanie, należy wykonać analizę `.DC` układu z rys. 6, przy zmianie wartości źródła prądowego `Vin` dla trzech wartości rezystora `R3`, równych odpowiednio 10 kΩ, 20 kΩ i 30 kΩ. Poniżej przedstawiony jest kod PSPICE realizujący to zadanie z wykorzystaniem analizy parametrycznej `.STEP`. Wynik jest oczywiście identyczny z przedstawionym w przykładzie 1 na rys. 8.

```
przykład nr 9, analiza parametryczna
*napięciowe źródło wejściowe
Vin in 0 dc 1
* obwod pasywny
L1 in 1 10mH
C1 in 1 1nF
R1 in 1 20k
R2 1 out 10k
R3 out 0 {resistance}
.param resistance=30k
.step param resistance list 10k 20k 30k
*analizy
.op
.dc Vin 0 5 .01
*sterowanie wyjściem i zakończenie pliku
.probe
```

```
.end
```

2.17. Operatory i funkcje wbudowane oraz deklaracja funkcji własnych

Format ogólny deklaracji własnej funkcji jest następujący:

```
.FUNC <nazwa>([argumenty]) {<ciało_funkcji>}
```

Przykłady deklaracji funkcji:

```
.FUNC E(x) {exp(x)}
.FUNC DECAY(CNST) {E(-CNST*TIME)}
```

Polecenie `.FUNC` nie może redefiniować istniejących funkcji ani funkcji predefiniowanych w PSPICE. Argumenty funkcji nie mogą być węzłami układu. Ciało funkcji definiuje działanie deklarowanej funkcji i musi być wcześniej zdefiniowane (np. `E` w przykładach powyżej jest wykorzystywane w funkcji `DECAY`). Liczba argumentów definicji funkcji musi być równa liczbie w wywołaniu funkcji. Maksymalna liczba argumentów wynosi 10, minimalna 0. W ciele funkcji można używać parametrów, czasu `TIME`, innych funkcji (predefiniowanych i własnych), operatorów i zmiennej Laplace'a *s*. Poniżej w dwóch kolejnych tabelach przedstawione są operatory i funkcje wbudowane w symulator PSPICE.

Tab. 10. Operatory wbudowane w symulatorze PSPICE.

Operator	Działanie	Uwagi
+	dodawanie	
-	odejmowanie	
*	mnożenie	
/	dzielenie	
**	potęgowanie	
~	negacja	operator logiczny wykorzystywany w funkcji IF
|	OR	operator logiczny wykorzystywany w funkcji IF
^	XOR	operator logiczny wykorzystywany w funkcji IF
&	AND	operator logiczny wykorzystywany w funkcji IF
==	równość	operator logiczny wykorzystywany w funkcji IF
!=	nierówność	operator logiczny wykorzystywany w funkcji IF
>	większe	operator logiczny wykorzystywany w funkcji IF
>=	większe lub równe	operator logiczny wykorzystywany w funkcji IF
<	mniejsze	operator logiczny wykorzystywany w funkcji IF
<=	mniejsze lub równe	operator logiczny wykorzystywany w funkcji IF

Tab. 11. Funkcje predefiniowane w symulatorze PSPICE.

Funkcja	Działanie	Uwagi
ABS(x)	$\|x\|$	
SQRT(x)	$x^{1/2}$	
EXP(x)	e^X	
LOG(x)	ln(x)	podstawą logarytmu jest e
LOG10(x)	log(x)	podstawą logarytmu jest 10
PWR(x,y)	$\|x\|^Y$	
PWRS(x,y)	$+\|x\|^Y$ $-\|x\|^Y$	dla X>0 dla x<0
SIN(x)	sin(x)	x wyrażone w radianach
ASIN(x)	$\sin^{-1}(x)$	wynik jest wyrażony w radianach
SINH(x)	sinh(x)	x wyrażone w radianach
COS(x)	cos(x)	x wyrażone w radianach
ACOS(x)	$\cos^{-1}(x)$	wynik jest wyrażony w radianach
COSH(x)	cosh(x)	x wyrażone w radianach
TAN(x)	tan(x)	x wyrażone w radianach
ATAN(x) ARCTAN(x)	$\tan^{-1}(x)$	wynik jest wyrażony w radianach
ATAN2(x,y)	$\tan^{-1}(y/x)$	wynik jest wyrażony w radianach
TANH(x)	tanh(x)	x wyrażone w radianach
M(x)	moduł (x)	używane w wyrażeniach Laplace’a
P(x)	faza (x)	w stopniach, używane w wyrażeniach Laplace’a
R(x)	część rzeczywista (x)	używane w wyrażeniach Laplace’a
IMG(x)	część urojona (x)	używane w wyrażeniach Laplace’a
DDT(x)	różniczka (x) względem czasu	używane tylko w analizach .TRAN
SDT(x)	całka (x) względem czasu	używane tylko w analizach .TRAN
TABLE(x_1,y_1, x_2,y_2,... x_n,y_n,)	wartości podane w postaci tabeli	punkty (x_1,y_1), (x_2,y_2), ... (x_n,y_n) połączone są między sobą liniami prostymi
MIN(x,y)	wartość mniejsza z x i y	
MAX(x,y)	wartość większa z x i y	

Tab. 11.c.d. Funkcje predefiniowane w symulatorze PSPICE

Funkcja	Działanie	Uwagi
LIMIT(x,min,max)	min max x	jeśli x<min jeśli x>max x w pozostałych przypadkach
SGN(x)	+1 0 -1	jeśli x>0 jeśli x=0 jeśli x<0
STP(x)	1 0	jeśli x>0 w pozostałych przypadkach
IF(t,x,y)	x jeśli t jest prawdą y w przeciwnym przypadku	t jest operatorem logicznym lub relacyjnym z poprzedniej tabeli

2.18. Analiza Monte Carlo

Analiza Monte Carlo polega na losowaniu wartości wybranych parametrów z zakresu dopuszczalnych zmian i wykonywaniu wielokrotnych symulacji z tak zmodyfikowanymi parametrami. Analiza ta ma na celu umożliwienie oceny poprawności działania układu przy zastosowaniu elementów o wartościach zmieniających się w pewnych granicach, czyli układu złożonego z elementów o pewnych tolerancjach wykonania. Składnia ogólna analizy jest następująca:

```
.MC <liczba_analiz> <typ_analizy> <zmienna_wyjściowa>
+ <funkcja> [OPCJE] [SEED=wartość]
```

Przykłady:

```
.MC 50 DC IC(Q7) YMAX LIST
.MC 20 AC VP(13,5) YMAX LIST OUTPUT ALL
```

Przy analizie **.MC** zmianie podlega wartość parametru modelu, dla którego podano specyfikację tolerancji **DEV** lub **LOT** (lub oba). Pierwszy przebieg symulacji wykonywany jest dla wartości nominalnych wszystkich elementów, kolejne przebiegi wykonywane są dla wartości losowanych w zakresie zgodnym ze specyfikacjami **DEV** i **LOT** podanymi w modelach. Znaczenie poszczególnych parametrów analizy **.MC** jest następujące:

- **liczba_analiz** - określa liczbę losowań i wykonywanych analiz, dla wyników zapisywanych w tekstowym pliku wyjściowym „*.out” górny limit wynosi 2000, dla pliku „*.dat” wynosi 400,
- **typ_analizy** - specyfikuje typ analizy dla której będzie wykonywane Monte Carlo, analiza ta musi być dodatkowo szczegółowo zdefiniowana do wykonania poza poleceniem **.MC**,
- **zmienna_wyjściowa** - zmienna wyjściowa specyfikowana jak dla polecenia **.PRINT**,

- **funkcja** specyfikuje operacje wykonywane na parametrze **zmienna_wyjściowa** i redukuje wynik do pojedynczej wartości, można wpisać jedną z następujących specyfikacji:
    a) **YMAX** - znajduje wartość bezwzględną największej różnicy z każdym wykresie w stosunku do wykresu nominalnego,
    b) **MAX** - znajduje wartość maksymalną w każdym wykresie,
    c) **MIN** - znajduje wartość minimalną w każdym wykresie,
    d) **RISE_EDGE(val)** – znajduje pierwszy przypadek przekroczenia wykresu ponad wartość **val**, wykres musi mieć jeden lub więcej punktów poniżej **val** , wartość wyjściowa będzie pierwszym punktem wykresu przekraczającym t wartość,
    e) **FALL_EDGE(val)** – znajduje pierwszy przypadek przekroczenia wykresu poniżej wartości **val**, wykres musi mieć jeden lub więcej punktów powyżej **val** , wartość wyjściowa będzie pierwszym punktem wykresu przekraczającym tą wartość,
- **<funkcja> [opcje]** nie mają znaczenia dla danych generowanych do postprocesora graficznego PROBE umieszczanych w binarnym pliku wyjściowym „*.dat”, mają one zastosowanie do tekstowego pliku wyjściowego pliku „*.out”, wyjątkiem jest opcja **OUTPUT**, której zastosowanie jest następujące:
    a) **OUTPUT ALL** - wszystkie przebiegi są zapisywane do wyjścia,
    b) **OUTPUT FIRST N** - **N** pierwszych symulacji przekazywane jest do wyjścia,
    c) **OUTPUT EVERY N** - co **N**-ty przebieg jest kierowany do wyjścia,
- **LIST** - na początku każdego nowego przebiegu listuje bieżące parametry modeli,
- **SEED=n** - definiuje wartość inicjalizującą generatora liczb losowych, wartość **n** musi być z przedziału od 1 do 32767.

Specyfikację tolerancji stosowaną do losowań wartości parametrów używanych w czasie symulacji umieszcza się w modelu danego elementu bezpośrednio za parametrem, którego ma dotyczyć, format ogólny deklaracji tolerancji jest następujący:

```
[DEV [track&dist] <val> [%]]
[LOT [track&dist] <val> [%]]
```

gdzie:

DEV - indywidualne tolerancje parametru, każdy element ma inną wartość w zakresie dopuszczalnej tolerancji,

LOT - wspólne tolerancje parametrów identyczne co do wartości dla wszystkich elementów opisanych danym modelem,

Jeśli występuje znak procenta % wówczas wyspecyfikowana tolerancja jest w procentach wartości nominalnej, w przeciwnym wypadku tolerancja jest w jednostkach identycznych jak dany parametr. Specyfikacja tolerancji oznacza możliwe zmiany powyżej i poniżej wartości nominalnej.

track&dist – specyfikuje numer użytego generatora i dystrybucję (jeżeli nie ma być użyta domyślna **UNIFORM**), **track** podaje się jako numer w zakresie od 0 do 9, a **dist** jako jedna z wartości: **UNIFORM**, **GAUS** lub nazwa dystrybucji zdefiniowana przez użytkownika.

UNIFORM – generuje równomiernie rozłożone wartości w zakresie podanych wartości, dystrybucja domyślna, dystrybucję domyślną można zmienić na inną poleceniem **.options**,

GAUSS – generuje wartości w zakresie ±3*sigma* zgodnie z dystrybucją Gaussa, wartość specyfikowana w **LOT/DEV** jest równa 1*sigma*,

Przykłady specyfikacji tolerancji:

```
.MODEL CMOD CAP (C=1 DEV 5%)
.MODEL DLOAD D (IS=1E-9 DEV .5% LOT 10%)
.MODEL RTRACK RES (R=1 DEV/GAUSS 1% LOT/UNIFORM 5%)
```

Przykład 10. Analiza Monte Carlo prostego układu pasywnego

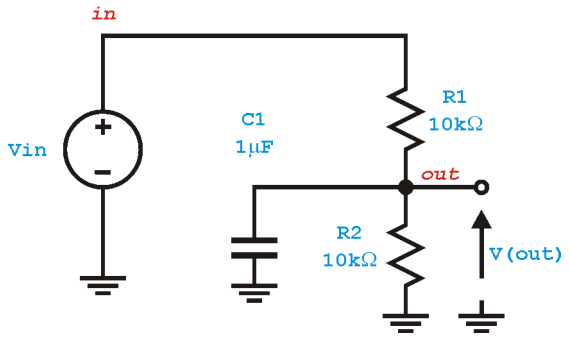
Jako badany układ wykorzystany jest dzielnik impedancyjny przedstawiony na rys. 16. Dla sygnałów o niskich częstotliwości oraz dla napięć stałych, współczynnik wzmocnienia jest równy:

$$\frac{V(out)}{Vin} = \frac{R2}{R1+R2} = \frac{10k\Omega}{10k\Omega+10k\Omega} = 0{,}5 \qquad (19)$$

Badany układ, ze względu na kondensator C1, jest układem dolnoprzepustowym o charakterystyce częstotliwościowej jednobiegunowej. Jego częstotliwość 3dB dolna wynika z bieguna powstałego przez kondensator C1 i rezystory R1 i R2 i można ją wyznaczyć na podstawie wzoru:

$$f_{3dB} = \frac{1}{2\pi}\frac{R1+R2}{R1R2C1} = \frac{1}{2\pi}\frac{10k\Omega+10k\Omega}{1uF10k\Omega10k\Omega} = 31{,}83Hz \qquad (20)$$

Dla celów analizy Monte Carlo założono, że kondensator C1 ma tolerancję 15% o dystrybucji równomiernej, natomiast rezystory R1 i R2 pochodzą z tej samej serii produkcyjnej i ich rezystancje mogą odbiegać maksymalnie o 10% od wartości nominalnej ale różnica pomiędzy nimi nie może być większa niż 4% wartości nominalnej. Dla rezystorów również należy użyć dystrybucji równomiernej.



Rys. 16. Prosty układ pasywny wykorzystany do przykładu symulacji Monte Carlo. Kursywą podane są nazwy węzłów użyte w kodzie PSPICE.

Kod PSPICE realizujący analizę Monte Carlo dla układu z rys. 16 i wcześniej podanych założeń dotyczących możliwych zmian rezystancji i pojemności, przedstawiony jest poniżej:

```
Przykład 10, analiza Monte Carlo
Vin in 0 dc 0 ac 1
R1 in out rmod 10k
R2 out 0 rmod 10k
C1 out 0 cmod 1u
.dc Vin 0 10 .01
.ac dec 200 1 1k
.MC 200 AC V([out]) MAX LIST OUTPUT ALL
.model cmod cap (C=1 dev=15% )
.model rmod res (R=1 dev=2% lot=8%)
.probe
.end
```

Należy zauważyć, że zdeklarowano modele dla kondensatora i rezystora a parametry mnożników wartości pojemności C i rezystancji R mają podaną specyfikację tolerancji. Dla kondensatora wynosi ona `DEV=15%` i ponieważ jest tylko jeden kondensator w badanym układzie to w tym przypadku nie ma znaczenia czy zastosowany zostanie parametr `DEV` czy `LOT`. Dla rezystorów mamy specyfikację `dev=2% lot=8%`, co oznacza, że maksymalna zmiana wartości danego rezystora w stosunku do wartości nominalnej może wynosić 8% + 2% = 10%, natomiast maksymalna różnica pomiędzy wartościami rezystorów R1 i R2 może wynosić 2% + 2% = 4%. Zanim zostaną zaprezentowane wyniki symulacji Monte Carlo, obliczono w jakich granicach mogą się znaleźć wartości wzmocnienia dla niskich częstotliwości oraz granice zmian częstotliwości trzydecybelowej układu. Zgodnie ze wzorem (19), wyłącznie zmiany względne rezystorów zmieniają wzmocnienie i minimalna wartość wzmocnienia wystąpi gdy R2 będzie miało minimalną możliwą rezystancję a R1 maksymalną. Maksymalna wartość wzmocnienia wystąpi dla odwrotnych warunków wartości rezystancji. Podstawiając dane do (19) otrzymujemy następujące wartości:

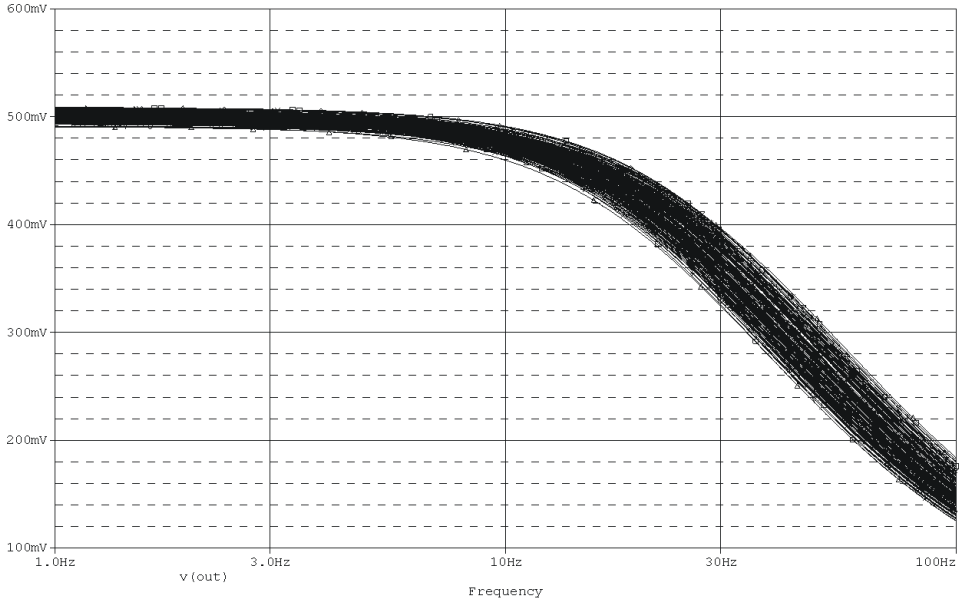
$$\left.\frac{V(out)}{Vin}\right|_{MIN} = \frac{R2_{MIN}}{R1_{MAX}+R2_{MIN}} = \frac{9{,}8k\Omega}{10{,}2k\Omega+9{,}8k\Omega} = 0{,}49 \tag{21}$$

$$\left.\frac{V(out)}{Vin}\right|_{MAX} = \frac{R2_{MAX}}{R2_{MAX}+R1_{MIN}} = \frac{10{,}2k\Omega}{10{,}2k\Omega+9{,}8k\Omega} = 0{,}51 \tag{22}$$

Podobnie można wyznaczyć granice częstotliwości trzydecybelowej jako:

$$f_{3dB,MIN} = \frac{1}{2\pi}\frac{R1_{MAX}+R2_{MAX}}{R1_{MAX}R2_{MAX}C1_{MAX}} = \frac{1}{2\pi}\frac{11k\Omega+11k\Omega}{1{,}15uF11k\Omega11k\Omega} = 25{,}16Hz \tag{23}$$

$$f_{3dB,MAX} = \frac{1}{2\pi}\frac{R1_{MIN}+R2_{MIN}}{R1_{MIN}R2_{MIN}C1_{MIN}} = \frac{1}{2\pi}\frac{9k\Omega+9k\Omega}{0{,}85uF9k\Omega9k\Omega} = 41{,}61Hz \tag{24}$$



Rys. 17. Wyniki symulacji Monte Carlo układu z rys. 16.

Na rys. 17 wykreślone zostało napięcie `v(out)` dla 200 przebiegów analizy `.AC` zgodnie ze zleceniem analizy Monte Carlo `.MC`. Niestety dość trudno jest odczytać wartości z wykresów przy tak dużej ilości danych. Z tego względu z pomocą przychodzą funkcje drukujące wyniki w tekstowym pliku wyjściowym „*.out”. W przypadku zastosowania funkcji `V([out]) MAX` jak w kodzie powyżej, fragment pliku wyjściowego zawiera dane przedstawione poniżej. Wynika z nich, że maksymalne uzyskane wzmocnienie wynosi 0,5087 a minimalne 0,4905. Wartości te mieszczą się w granicach wyliczonych wzorami (22) i (21). W przypadku wykonania analizy z większą liczbą losowań nastąpiłoby lepsze zbliżenie wartości otrzymanych na podstawie symulacji z obliczonymi teoretycznie.

```
Przykład 10, analiza Monte Carlo
 ****     SORTED DEVIATIONS OF V(out)      TEMPERATURE =   27.000 DEG C
                      MONTE CARLO SUMMARY
```

```
**************************************************************************
 RUN                     MAXIMUM VALUE
Pass  141                 .5087 at F =    1
                         ( 101.78% of Nominal)
.
.
.
Pass   74                 .4905 at F =    1
                         (  98.149% of Nominal)
```

Podobnie, jeśli zmienimy funkcję celu w poleceniu analizy Monte Carlo, wówczas można poszukiwać częstotliwości 3dB. Poniżej przedstawiono fragmenty tekstowego pliku wyjściowego „*.out” dla następującego polecenia `.MC`:

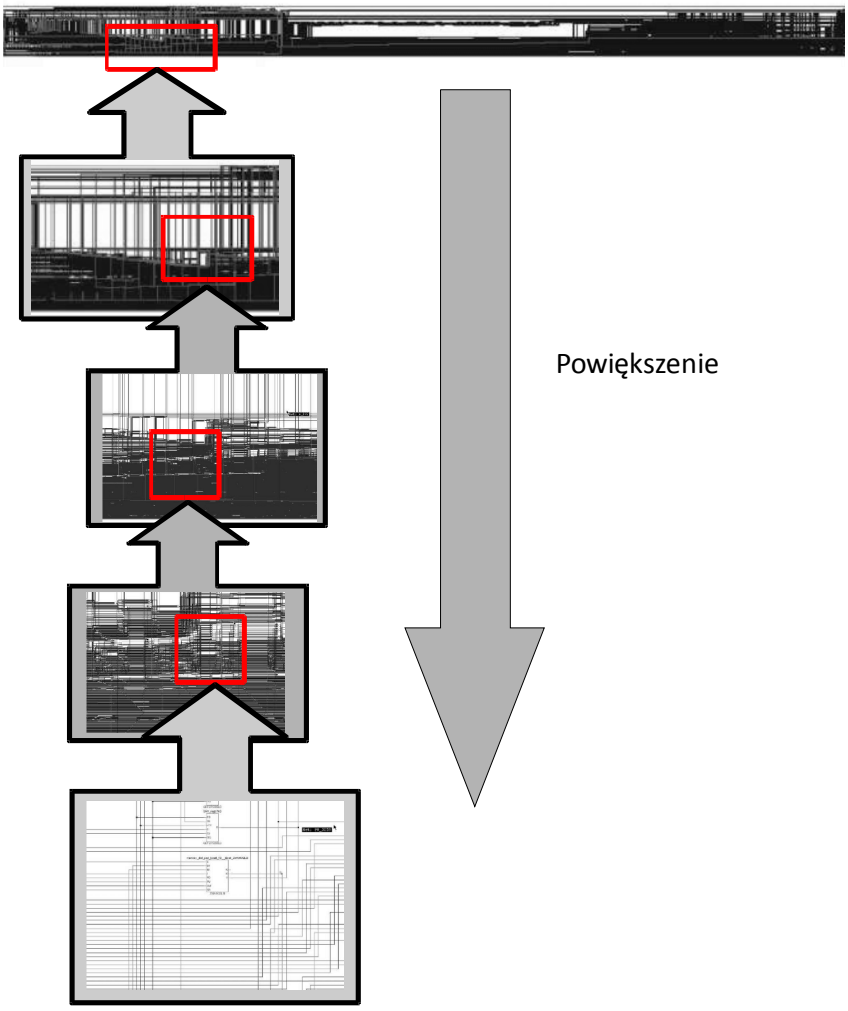
```
.MC 200 AC V([out]) FALL_EDGE(0.3535533) LIST OUTPUT ALL
```

Wartość 0,3535533 jest wyliczoną wartością napięcia wyjściowego dla 3dB spadku wzmocnienia w stosunku do nominalnej równej wartości 0,5.

```
Przykład 10, analiza Monte Carlo
 ****     SORTED DEVIATIONS OF V(out)      TEMPERATURE =   27.000 DEG C
                      MONTE CARLO SUMMARY
**************************************************************************
 RUN                     FIRST FALLING EDGE VALUE THRU     .3536
Pass  174                 39.202
Pass  181                 38.641
.
.
.
Pass  184                 25.429
```

3. Wstęp do języków HDL

Pierwsze cyfrowe układy scalone zawierały niewielką liczbę elementów logicznych. Do ich zaprojektowania wystarczyła kartka papieru, na której mieścił się schemat całego układu. Wraz z coraz większym stopniem złożoności układów, ich projektowanie za pomocą rysowania schematów stało się kłopotliwe - schematy składały się z wielu arkuszy i były bardzo trudne do zrozumienia i modyfikacji. Na rys. 18 zamieszczono schemat złożonego układu cyfrowego w różnych stopniach powiększenia. Jak widać, schematy złożonych układów są skomplikowane i mało czytelne, zatem bardzo trudno jest wywnioskować w jaki sposób działa układ.



Rys. 18. Schemat złożonego układu cyfrowego w różnych skalach.

Z tego powodu opracowano tzw. języki opisu sprzętu HDL (ang. *Hardware Description Language*), za pomocą których opisuje się działanie układów cyfrowych z wykorzystaniem otwartego tekstu. Taki opis przypomina wyglądem klasyczny program komputerowy, gdyż może zawierać np. instrukcje typu *if/then/else* czy pętle *for*. Kod opisany za pomocą języka opisu sprzętu nie zawiera jednak opisu algorytmu realizowanego krok po kroku za pomocą kolejnych rozkazów wykonywanych przez

procesor, jak to ma miejsce w typowym programie komputerowym. Zamiast tego jest to opis działania sprzętu – bramek i przerzutników w układzie cyfrowym, a jedna linia w kodzie HDL może definiować działanie bardzo wielu bramek lub przerzutników.

Opis projektu z wykorzystaniem języka opisu sprzętu można wprowadzać do komputera za pomocą zwykłego edytora tekstowego. Wraz z nastaniem ery układów scalonych bardzo dużej skali integracji VLSI (ang. *Very Large Scale of Integration*), projektowanie układów z wykorzystaniem języków opisu sprzętu stało się standardem. Projekt układu opisany za pomocą języków opisu sprzętu jest dużo bardziej przyjazny człowiekowi - opis tekstowy wraz z komentarzem jest łatwiej zrozumieć i analizować. Języki projektowania sprzętu umożliwiają opis projektu na wysokim poziomie abstrakcji - projektant nie musi się skupiać na szczegółach technicznych poszczególnych operacji, które ma realizować projektowany układ. Dzięki temu możliwe jest sprawdzenie działania układu na bardzo wczesnym etapie projektowania za pomocą symulacji kodu HDL. W ten sposób można także podzielić pracę pomiędzy różne zespoły projektantów: na przykład jednemu zespołowi zlecamy opracowanie płyty głównej komputera, a drugi zespół projektowy otrzymuje za zadanie zaprojektowanie karty graficznej do tego komputera. Obydwa zespoły na samym początku szybko opracowują modele swoich docelowych urządzeń na wysokim poziomie abstrakcji, następnie wymieniają się nimi pomiędzy sobą. Od tego momentu zespół projektujący płytę główną dysponuje modelem karty graficznej i dzięki temu może sprawdzać działanie płyty głównej za pomocą symulatora. Równocześnie zespół projektujący kartę graficzną otrzymuje do testowania model płyty głównej. Wraz z czasem, modele obydwu projektowanych urządzeń stają się coraz dokładniejsze, umożliwiając coraz bardziej szczegółową analizę ewentualnych problemów. Wymiana modeli spowodowała, że projekty mogły powstawać w sposób równoległy od samego początku.

Podsumowując, poniżej wymieniono najważniejsze zastosowania języków opisu sprzętu:

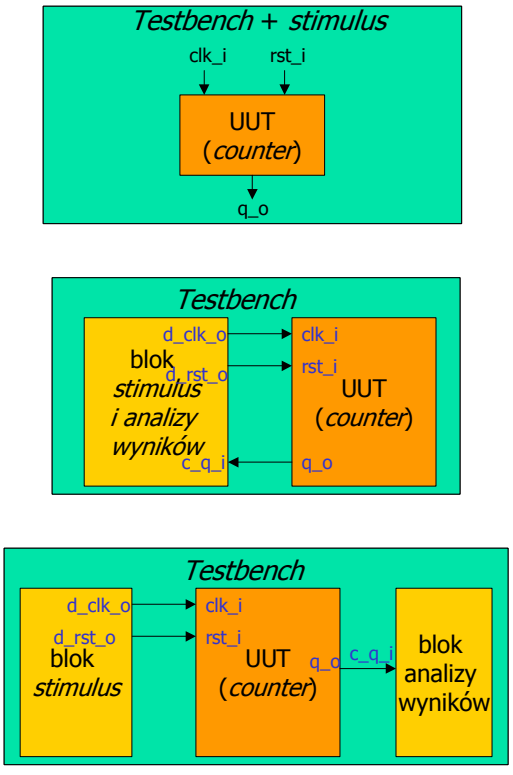
- Wprowadzanie projektu do komputerowych systemów automatycznej syntezy i projektowania – jest to alternatywa do wprowadzania projektu np. poprzez rysowanie schematu.
- Opisu układu (*netlist*), który może być wymieniany pomiędzy różnymi narzędziami do projektowania.
- Modelowanie i symulacja układów cyfrowych.
- Weryfikacja projektu. Za pomocą języków opisu sprzętu nie tylko opisuje się działanie układu, ale także przygotowuje środowisko do testowania projektowanych modułów, zwane *test bench*'em.
- Sporządzanie dokumentacji projektu. Opis projektu w języku opisu sprzętu, o ile jest wyposażony w odpowiednie komentarze, stanowi jednocześnie dokumentację projektu.

3.1. Symulacje i testowanie

W ciągu procesu projektowego moduły zaprojektowanie w języku HDL są testowane z wykorzystaniem symulatora. Do symulacji potrzebny jest zarówno opis badanego układu jak i wejściowych sygnałów pobudzających łącznie z procedurą badania wyników symulacji. Do celów testowania przygotowuje się specjalny blok tzw. *test bench*, napisany również w języku HDL, w którym:

- generowane są sygnały wejściowe do testowanych bloków;
- osadzany jest testowany moduł/projekt (tzw. *Unit Under Test*, w skrócie: UUT);
- odbierane są sygnały wyjściowe z testowanego bloku, ewentualnie możliwa jest także ich analiza.

Test bench jest poddawany symulacji, podczas której można sprawdzić poprawność projektu, gdyż symulowany jest projektowany moduł wraz z odpowiednimi pobudzeniami. Oddzielne bloki *test bench* mogą być stworzone dla poszczególnych podbloków sytemu cyfrowego, jak i dla kompletnego układu. Na rys. 19 przedstawiono różne możliwości testowania bloku UUT za pomocą *test bench*'a.



Rys. 19. Różne warianty testowania bloku UUT z wykorzystaniem *test bench*'a.

Należy zauważyć, że kompletny *test bench* jest blokiem nie zawierającym portów zewnętrznych.

4. Język Verilog

4.1. Pojęcia podstawowe

Język Verilog [4][5][6] zyskał dużą popularność, gdyż jest łatwy do nauczenia i używania, m.in. ze względu na podobieństwo do języka C, który zazwyczaj jest dobrze znany inżynierom projektującym sprzęt. Większość popularnych systemów CAD (ang. *Computer Aided Design*) posiada implementację języka Verilog, a nawet zdarzają się narzędzia, które nie akceptują innych niż Verilog języków opisu sprzętu. Producenci układów ASIC (ang. *Application Specific Integrated Circuit*) przeważnie dostarczają biblioteki do symulacji napisane właśnie w tym języku. Dla zaawansowanych użytkowników, język Verilog umożliwia dopisywanie własnych procedur w języku C współdziałających z wewnętrznymi strukturami Verilog za pomocą tzw. *Verilog Procedural Interface*. Dzięki temu można wywoływać własne funkcje C z poziomu Veriloga i odwrotnie: z poziomu programu w języku C można łączyć się z kodem napisanym w Verilogu.

Język Verilog umożliwia opis układu na różnych poziomach abstrakcji jednocześnie, przez co można mieszać opis na wysokim poziomie abstrakcji z opisem niskopoziomowym, nawet na poziomie poszczególnych tranzystorów. Podział opisu na różne poziomy abstrakcji jest sprawą umowną, ale w literaturze najczęściej występuje podział na następujące poziomy [4]:

- poziom kluczy,
- poziom bramek logicznych,
- poziom rejestrów (*Dataflow*),
- poziom behawioralny.

Przyjęło się, że do każdego z tych poziomów opisu przypisuje się konkretne elementy języka Verilog, przez co łatwiej jest zapamiętać składnię języka. W dalszej części tego skryptu opisano poszczególne poziomy abstrakcji, z wyłączeniem najniższego poziomu kluczy, który jest najrzadziej używany.

4.1.1. Podstawowe zasady składni języka Verilog

W Verilogu małe i DUŻE litery są rozróżniane, więc symbole:

```
licznik
```

oraz

```
Licznik
```

będą reprezentowały dwie zupełnie różne zmienne.

Wszystkie słowa kluczowe piszemy małymi literami. Poszczególne słowa języka Verilog rozdzielone są tzw. białą spacją (spacja, tabulator, koniec linii), poza tym białe spacje są ignorowane (wyjątek to łańcuchy).

W języku Verilog mamy do dyspozycji dwa sposoby komentowania tekstu opisu HDL. Pierwszy sposób umożliwia komentowanie pojedynczych linii – po dwóch ukośnikach `//` cały tekst do końca linii jest traktowany jako komentarz:

```
// komentarz do końca linii
```

Drugi sposób pozwala na komentowanie większej ilości tekstu:

```
/* komentarz
przez kilka linii */
```

Taki sposób zaznaczania komentarzy został zaczerpnięty z języka C, tak jak i wiele innych zasad składni języka Verilog. Ze względu na podobieństwo do języka C, składnia języka Verilog jest dużo łatwiejsza do przyswojenia niż składnia języka VHDL, która to jest oparta o mało znane w dzisiejszych czasach języki Ada i Algol.

4.1.2. Operatory

Ze względu na liczbę operandów (argumentów), w języku Verilog wyróżniamy trzy rodzaje operatorów:

- operatory jednoargumentowe, np.:

```
A = ~B;
```

- operatory dwuargumentowe, np.:

```
A = B && C;
```

- operatory trzyargumentowe, np.:

```
A = B ? C : D
```

4.1.3. Liczby

Dla osoby znającej język C, sposób zapisu liczb w języku Verilog jest najbardziej egzotycznym elementem składni. Stosuje się następującą konwencję zapisu liczb:

```
<rozmiar>'<podstawa><liczba>
```

gdzie `<rozmiar>` to liczba dziesiętna określająca liczbę bitów w binarnej reprezentacji liczby, a `<podstawa>` to litera oznaczająca podstawę zapisu liczby `<liczba>`. Litera `d` lub `D` oznacza, że `<liczba>` jest zapisana jako dziesiętna, `h` lub `H` oznacza liczbę szesnastkową, a `o` lub `O` – liczbę zapisaną w formacie ósemkowym.

Najlepiej wyjaśnić to za pomocą przykładów:

`8'b00001111` - to jest 8-bitowa liczba dwójkowa o wartości binarnej 00001111, dziesiętnie 15;

`16'habab` - to jest 16-bitowa liczba szesnastkowa *abab*, dziesiętnie 43947;

`32'd255` - to jest 32-bitowa liczba dziesiętna o wartości dziesiętnej 255.

Jeśli nie określimy podstawy `<podstawa>`, to wtedy liczba `<liczba>` traktowana będzie jako dziesiętna. Jeśli nie określimy rozmiaru liczby `<rozmiar>`, to najczęściej będzie przyjęte, że jest to liczba 32-bitowa, np.:

`12345` – po prostu liczba dziesiętna;

`'ha1` – 32-bitowa liczba szesnastkowa;

`'o22` – 32-bitowa liczba ósemkowa.

Jeśli w wartości liczby występuje symbol `x`, to oznacza stan nieokreślony; symbol `z` oznacza stan wysokiej impedancji, np.:

`64'bz` – 64-bitowa liczba o wysokiej impedancji;

`8'hx` – 8-bitowa liczba szesnastkowa;

`12'haxc` – 12-bitowa liczba szesnastkowa, 4 środkowe bity są nieokreślone.

W zależności od podstawy, `x` lub `z` oznacza: cztery bity w reprezentacji szesnastkowej, trzy bity w reprezentacji ósemkowej i jeden bit w dwójkowej. Jeśli najbardziej znaczący bit w liczbie to `0`, `z` lub `x`, to najbardziej znaczące bity są automatycznie wypełniane tą wartością. Jeśli najbardziej znaczącym bitem jest `1`, to pozostałe najbardziej znaczące bity są uzupełniane zerami.

Liczby ujemne określa się poprzez dodanie znaku minus przed określeniem rozmiaru liczby, np.:

`-8'd14` - 8-bitowa liczba ujemna przechowywana binarnie w formacie uzupełnienia do dwóch.

Aby zwiększyć czytelność liczb w Verilogu, można stosować znak podkreślenia (byle nie na początku liczby), np.:

```
16'b1001_0000_1111_0101
```

Znak zapytania w liczbie jest odpowiednikiem stanu wysokiej impedancji `z`.

4.1.4. Identyfikatory

W identyfikatorach (tj. nazwach zmiennych, nazwach funkcji) można stosować wyłącznie znaki alfanumeryczne, znak podkreślenia oraz znak `$`, przy czym na początku identyfikatora nie może występować znak `$` ani cyfra.

Jest możliwość definiowania identyfikatorów zawierających dowolne znaki, ale taki specjalny identyfikator należy poprzedzić znakiem `\`:

```
\*zmiennaXXX**
\XYX++ZZZ
```

Identyfikatory specjalne można czasem spotkać np. w kodzie Verilog, który został automatycznie wygenerowany przez narzędzia CAD.

4.1.5. Zestaw wartości

Język Verilog służy do opisu i modelowania układów cyfrowych, dlatego też sygnałom można przypisywać wartości, których znaczenie ma sens fizyczny. Mamy do dyspozycji cztery podstawowe wartości sygnałów, zgodnie z tab. 12.

Tab. 12. Zestaw wartości w języku Verilog.

Wartość	Znaczenie w układach cyfrowych
`0`	logiczne zero, fałsz
`1`	logiczna jedynka, prawda
`x`	wartość nieokreślona
`z`	stan wysokiej impedancji

Oprócz zdefiniowanego zestawu wartości, sygnałom przypisuje się różne poziomy siły. Jeśli dwa sygnały sterujące tym samym przewodem mają równe siły a różne wartości, to na przewodzie wystąpi sygnał nieokreślony `x`. Jeśli jednak jeden z sygnałów jest silniejszy, to jego wartość będzie ustalona na wspólnym przewodzie. Poziomy siły sygnału używane w języku Verilog przedstawiono w tab. 13.

Tab. 13. Poziomy siły sygnału.

Siła sygnału	Typ	Opis
`supply`	*Driving*	najsilniejszy
`strong`	*Driving*	
`pull`	*Driving*	
`large`	*Storage*	
`weak`	*Driving*	
`medium`	*Storage*	
`small`	*Storage*	
`highz`	*High Impedance*	najsłabszy

4.1.6. Sieci

Sieci reprezentują połączenia pomiędzy elementami układu cyfrowego. Standardowo sieci przyjmują szerokość 1-bitową, a wartością domyślną jest wartość `z`. Sieci nie maja zdolności do pamiętania wartości, wymagają zatem zawsze jakiegoś elementu, który będzie sterował wartością sieci.

Przykłady deklaracji sieci:

```
wire signalA;
wire data1, data2;
wire data_out=1'b0; // sieć data_out będzie miała zawsze wartość 0
```

4.1.7. Rejestry

Rejestry służą do modelowania elementów pamiętających. Można sobie wyobrazić, że są to zmienne, które pamiętają ostatnio zapisaną wartość. Nie należy jednak sądzić, że w docelowym układzie, który powstanie z kodu, w którym została zadeklarowana zmienna typu **reg**, w miejscu tej zmiennej będzie zawsze występował przerzutnik – nie ma na to żadnej gwarancji, gdyż taka zmienna może zostać np. zredukowana w wyniku optymalizacji kodu Verilog podczas syntezy lub implementacji.

Przykłady deklaracji zmiennych typu **reg**:

```
reg switch;     // <- deklaracja zmiennej switch
initial         // <- będzie wyjaśnione później
begin
    switch =1'b0;           // ustaw switch = 0
    #220 switch =1'b1;      // po 220 jednostkach czasu ustaw switch = 1
end
```

4.1.8. Wektory

Sieci typu **wire** oraz zmienne typu **reg** mogą być wektorami. Przykłady deklaracji wektorów:

```
wire A;                     // skalar
wire [7:0] data;            // 8-bitowa magistrala
wire [31:0] dataA, dataB;   // dwie 32-bitowe magistrale
reg [0:31] addr;            // 32-bitowa zmienna typu reg
```

Sposób indeksowania wektorów może być malejący [*max*:*min*] lub rosnący [*min*:*max*], ale zawsze lewa liczba oznacza najbardziej znaczący bit.

Użycie części wektorów odbywa się za pomocą nawiasów klamrowych:

```
data[7]         // 7-my bit wektora data
data[2:0]       // trzy najmniej znaczące bity wektora data
                // (nie wolno odwracać bitów: bus[0:2]!)
```

4.1.9. Liczby całkowite i rzeczywiste

Verilog umożliwia deklarowanie zmiennych typu **integer**. Takie zmienne mają zdolność zapamiętywania zapisanych do nich wartości, podobnie jak zmienne typu **reg**.

Przykład deklaracji zmiennych całkowitych:

```
integer licznik;
initial         // <-- będzie objaśnione później
    licznik = -1;
```

Podobnie można korzystać z liczb rzeczywistych **real**:

```
real wartosc;
initial
begin
```

```
    wartosc=14.7e10;
  end
```

W przypadku stosowania języka Verilog do modelowania i symulacji, nie ma specjalnych ograniczeń dotyczących wykorzystania zmiennych typu **integer** oraz **real**. Jeśli jednak dany kod Verilog jest przeznaczony do realizacji rzeczywistego układu cyfrowego, to należy pamiętać, że program syntezujący nie będzie potrafił zamienić zmiennych typu **real** na sprzęt[1], natomiast zmienne typu **integer** będą realizowane fizycznie w układzie jako sygnały 32-bitowe.

4.1.10. Tablice i pamięci

Tablice są to zbiory indeksowanych elementów tego samego typu. W języku Verilog można definiować tablice elementów typu **reg**, **integer** lub **time**. Nie wolno deklarować tablic dla zmiennych typu **real**. Dostęp do elementu tablicy odbywa się za pomocą nawiasów kwadratowych:

```
<nazwa_tablicy>[<indeks>]
```

W podstawowym standardzie języka Verilog (tzw. Verilog 1995) nie wolno definiować tablic wielowymiarowych (można to robić dopiero w standardzie Verilog 2001 – patrz opis w rozdziale 4.8). Poniżej przedstawiono różne przykłady korzystania z tablic:

```
integer licznik[0:7];       // tablica 8 liczników
reg bits[31:0];             // tabl. 32, jednobitowych rejestrów
time chkPoint[1:1000];
reg [4:0] port_id[0:7];     // tablica 8 zmiennych,
                            // każda ma 5 bitów

licznik[5]                  // 5-ty element tablicy licznik
port_id[3]                  // 3-ci element zmiennej (5-bitowej)
```

Nie można odwoływać się bezpośrednio do pojedynczych bitów tablicy – trzeba to zrobić dwuetapowo: najpierw należy odwołać się to pojedynczego elementu tablicy, a w drugim kroku wyodrębnić poszczególny bit.

Nie należy mylić tablic z wektorami: wektor to pojedynczy element o szerokości *n*-bitów, a tablica to zestaw wielu elementów o szerokości 1 lub *n* bitów. Tablice w Verilogu nazywane są także pamięciami, przy założeniu, że pamięci to tablice rejestrów, gdzie każdy element tablicy to słowo, a każde słowo może mieć długość 1 lub więcej bitów.

[1] Typowe programy syntezujące nie obsługują syntezy zmiennych typu **real**. Synteza zmiennych typu **real** jest możliwa po dołączeniu specjalistycznych bibliotek.

4.1.11. Łańcuchy

Stałe typu łańcuchowego muszą być zdefiniowane w jednej linii - nie dopuszcza się znaków CR (tj. ENTER) w łańcuchu. Przykład zdefiniowania łańcucha:

```
string_variable="Hello Verilog Word";
```

Łańcuchy są przechowywane w zmiennych typu `reg`. Szerokość rejestru musi być wystarczająca do przechowania łańcucha, przy założeniu, że każdy znak w łańcuchu zajmuje 8 bitów:

```
reg [8*11:1] txt; // zmienna o szer. 11 bajtów
initial
    txt="Hello World";
```

Jeśli łańcuch jest krótszy, Verilog uzupełnia bity z lewej strony zerami. Jeśli łańcuch jest za długi, to Verilog obcina lewą część łańcucha.

W łańcuchu można umieszczać znaki specjalne poprzedzone znakiem `\`:

```
\n    =    nowa linia
\t    =    tabulator
%%    =    %
\\    =    \
\"    =    "
\ooo  =    znak zapisany 1-3 cyfr ósemkowych
```

4.1.12. Zadania systemowe

Zadania systemowe to specjalne polecenia dla symulatora rozpoczynające się od znaku `$`. Najprostszym zadaniem systemowym jest polecenie `$display` drukujące na ekranie komunikat, np.:

```
$display("Hello World");
Hello Word
```

Polecenie `$time` zwraca aktualny czas symulacji:

```
$display($time);
1231
```

Polecenie `$display` umożliwia wydrukowanie wartości zmiennych, podobnie jak to ma miejsce w poleceniu `printf` języka C:

```
$display("At time %d value is %h", $time, licznik);
At time 200 value is 0000000d

$display("At time %d value is %b", $time, licznik);
At time 200 value is 00000000000000000000000000001101
```

Znacznik `%m` powoduje wydrukowanie nazwy hierarchicznej aktualnej instancji:

```
$display ("Printout from module %m");
Printout from module odbiornik.b1
```

W celu wydrukowania znaku `%`, należy w łańcuchu formatującym umieścić podwójny znak `%`:

```
$display("Multi-line \n message, finished 100%%");
Multi-line
 message, finished 100%
```

Polecenie systemowe `$display` na koniec zawsze wstawia znak nowej linii. Poniżej zamieszczono objaśnienia znaków formatujących, których znaczenie jest podobne do stosowanych w funkcji `printf` w C:

```
%d lub %D   wyświetl zmienną w postaci dziesiętnej
%b lub %B   wyświetl zmienną w postaci binarnej
%s lub %S   wyświetl zmienną jako łańcuch
%h lub %H   wyświetl zmienną w postaci szesnastkowej
%c lub %C   wyświetl zmienną w postaci znaku ASCII
%m lub %M   wyświetl nazwę hierarchiczną (nie potrzebny argument)
%v lub %V   wyświetl siłę sygnału
%o lub %O   wyświetl zmienną w postaci ósemkowej
%t lub %T   wyświetl zmienną w postaci czasu
%e lub %E   wyświetl zmienną w postaci naukowej (np. 3.45e6)
%f lub %F   wyświetl zmienną w postaci zmiennoprzecinkowej
%g lub %G   wyświetl zmienną w postaci naukowej lub zmiennoprzecinkowej, w
zależności od tego, która jest prostsza.
```

Za pomocą zadań systemowych można także monitorować zmienne: po każdej zmianie sygnału zawartego w liście argumentów polecenia `$monitor`, będzie drukowana wartość tego sygnału:

```
$monitor(var1, var2, var3, ..., varN);
```

Parametry polecenia `$monitor` określamy tak samo, jak w poleceniu `$display`. Różnicą w stosunku do `$display` jest to, że polecenie `$monitor` wystarczy podać tylko raz i jest ono cały czas aktywne. Jeśli w module Verilog występuje więcej niż jedno polecenie `$monitor`, to aktywne jest tylko ostatnie wystąpienie tego polecenia.

Kontrolę nad drukowaniem komunikatów przez zadanie systemowe `$monitor` umożliwiają następujące zadania:

`$monitoron;` – załącza monitorowanie,

`$monitoroff;` – wyłącza monitorowanie.

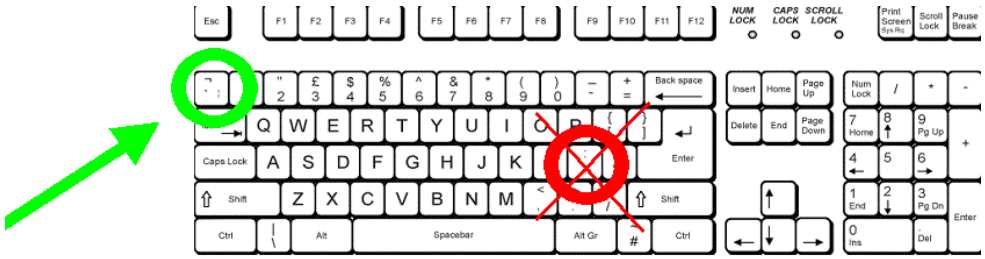
Do przerywania i kończenia symulacji służą następujące zadania systemowe:

`$stop;` - zatrzymuje symulację i przełącza symulator w tryb interaktywny;

`$finish;` - kończy symulację.

4.1.13. Dyrektywy kompilatora

W języku Verilog, podobnie jak w języku C, mamy możliwość definiowania makr tekstowych. Makra tekstowe oznaczamy za pomocą znaku *back tick* (`), którego położenie na klawiaturze komputera przedstawiono na Rys. 20, aby nie pomylić ze znakiem *tick* (').



Rys. 20 Umiejscowienie znaku *back tick* na klawiaturze.

Polecenie `` `define <nazwa_makra> <tekst> `` definiuje makro tekstowe. Verilog zastępuje każde wystąpienie `` `<nazwa_makra> `` określonym tekstem `<tekst>`. Przykłady definicji makr:

```
`define TRUE 1'b1
`define DATA_WIDTH 32
`define S $stop;
```

Wykorzystanie makra pokazuje poniższy przykład:

```
assign out = `TRUE;
```

gdzie ciąg znaków "`` `TRUE ``" zostanie zastąpiony tekstem "`1'b1`", zgodnie ze zdefiniowanym powyżej makrem `` `define TRUE 1'b1 ``.

Podobny mechanizm zastosowano w języku C, z tym, że zamiast *backtick* (`) jest stosowany w C znak *hash* (#), a wykorzystanie makra w C nie wymaga zastosowania dodatkowego znaku przed nazwą makra (w Verilogu przed nazwą makra należy także umieścić *backtick* (np. `assign out = `TRUE;`).

Polecenie `` `include `` umożliwia wstawienie w danym miejscu w pliku całej zawartości innego pliku:

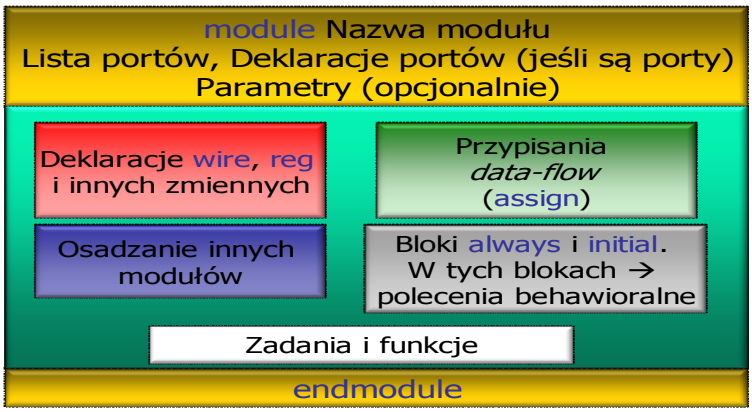
```
`include header.v
```

Warunkową kompilację kodu Verilog uzyskamy za pomocą słów kluczowych `` `ifdef ``, `` `else `` oraz `` `endif ``. Warunki sprawdzane przez `` `ifdef `` definiuje się za pomocą polecenia `` `define ``.

```
`ifdef asic
    `include "memory_asic.v";
`else
    `include "memory_fpga.v";
`endif
```

4.2. Moduły i Porty

Podstawową jednostką projektu w języku Verilog jest moduł. Definicję modułu przedstawiono za pomocą schematu blokowego na rys. 21.



Rys. 21. Definicja modułu.

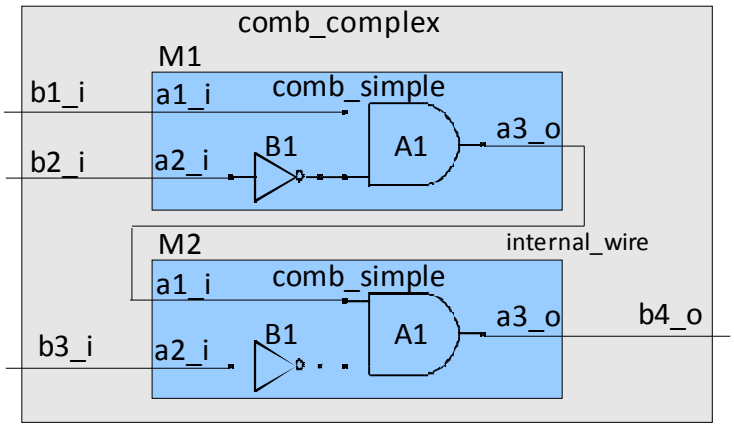
Po słowie kluczowym **`module`** powinna wystąpić nazwa modułu, a następnie w nawiasie lista portów modułu, zakończona średnikiem. Jeśli moduł nie wymienia sygnałów z otoczeniem, to lista portów jest niepotrzebna. Oczywiści moduł, który ma być fizycznie zrealizowany musi mieć porty, bo jego istnienie nie miałoby sensu. Dla modułu przeznaczonego wyłącznie do symulacji brak portów nie jest niczym niezwykłym – symulator i tak ma dostęp do wszystkich sygnałów wewnątrz modułu.

Definicję modułu kończymy słowem kluczowym **`endmodule`**, po którym nie występuje średnik.

Wewnątrz modułu, tj. pomiędzy nazwą modułu z listą portów, a słowem **`endmodule`**, możemy umieścić następujące elementy języka Verilog:

- deklaracje zmiennych typu **`wire`**, **`reg`** oraz innych zmiennych;
- przypisania ciągłe typu *data-flow*;
- osadzenie innych modułów (wewnątrz modułu nie definiować innego modułu, ale można osadzać inne moduły, zdefiniowane poza modułem);
- bloki behawioralne typu **`always`** i **`initial`**;
- zadania i funkcje.

Poniżej pokazano przykład osadzenia dwóch modułów **`comb_single`** w module **`comb_complex`** (Rys. 22):



Rys. 22. Przykład budowy hierarchicznej projektu.

```
module comb_simple (a1_i, a2_i, a3_o);
    input a1_i, a2_i;
    output a3_o;
    wire tmp;
    not N1(tmp, a2_i);
    and A1(a3_o, a1_i, tmp);
    assign a3_o = a1_i & ~a2_i;
endmodule

module comb_complex (b1_i, b2_i, b3_i, b4_o);
    input b1_i, b2_i, b3_i;
    output b4_o;
    wire internal_wire;
    comb_simple M1(b1_i, b2_i, internal_wire);
    comb_simple M2(internal_wire, b3_i, b4_o);
endmodule
```

4.2.1. Porty

Porty służą modułowi do wymiany informacji z otoczeniem. W definicji modułu wszystkie zdefiniowane porty powinny zostać zadeklarowane jako **input**, **output** lub **inout**. Jeśli nie określimy typu sygnału portu, to domyślnie przyjmowany jest typ **wire** (jest to zazwyczaj wystarczające dla portów **input** i **output**). Porty **output** można deklarować jako **reg**, jeśli jest konieczne, aby pamiętały zapisaną wartość.

Istnieją specjalne reguły, do jakich sygnałów możemy podłączać porty modułu:

- porty **input** możemy podłączyć wewnątrz modułu do sieci typu **wire**, na zewnątrz mogą być podłączone do sygnałów typu **reg** lub **wire**;
- porty **output** możemy podłączyć wewnątrz modułu do sieci typu **wire** lub **reg**, na zewnątrz mogą być podłączone wyłącznie do sygnałów typu **wire**;

- porty `inout` możemy podłączyć wewnątrz modułu tylko do sieci typu `wire`, na zewnątrz mogą być również podłączone wyłącznie do sygnałów typu `wire`.

Dozwolone jest łączenie ze sobą portów będących magistralami (tj. portami wieloprzewodowymi), a w przypadku łączenia magistral o różnej liczbie bitów, kompilator wygeneruje jedynie ostrzeżenie. Można także zostawiać porty nie podłączone do żadnego sygnału.

Podłączenie modułu do istniejących przewodów może się odbywać na dwa sposoby (na przykładzie połączeń hierarchicznych pomiędzy modułami z przykładu z rys. 22) :

- lista uporządkowana (wymagana jest kolejność zgodna z kolejnością portów w definicji modułu):

```
comb_simple M1(b1_i, b2_i, internal_wire);
```

- łączenie poprzez podanie nazwy (kolejność portów jest dowolna):

```
comb_simple M1(.a3_o(internal_wire), .a2_i(b2_i), .a1_i(b1_i));
```

4.2.2. Definiowanie parametrów modułu

Parametry umożliwiają ustawienie pewnych wartości w modułach podczas ich powoływania do życia. Są dwie możliwości podawania parametrów:

- poprzez polecenie `defparam`;
- podczas osadzania modułu.

Polecenie `defparam` umożliwia ustawienie parametrów w dowolnym module, a w jednym module może być wiele poleceń `defparam`. Poniższy przykład pokazuje wykorzystanie polecenia `defparam`:

```
module defparam_example;
    parameter bin = 0;
    initial
          $display("Ten moduł ma parametr bin = %d", bin);
endmodule

module main;
    defparam EX1.bin = 1, EX2.bin = 2;
    defparam_example EX1();
    defparam_example EX2();
endmodule
```

Zamiast wywoływania polecenia `defparam` przed osadzeniem modułu, można wykorzystać drugi sposób i osadzić moduł wraz z przekazaniem wartości parametru, tak, jak to pokazano w kolejnym poniżej przedstawionym przykładzie:

```
module main2;
    defparam_example #(1) EX1();
    defparam_example #(2) EX2();
endmodule
```

W przypadku potrzeby przekazania kilku parametrów, należy je wymienić po przecinku, w takiej kolejności, w jakiej zostały one zdefiniowane w definicji modułu lub poprzez podanie nazwy parametru:

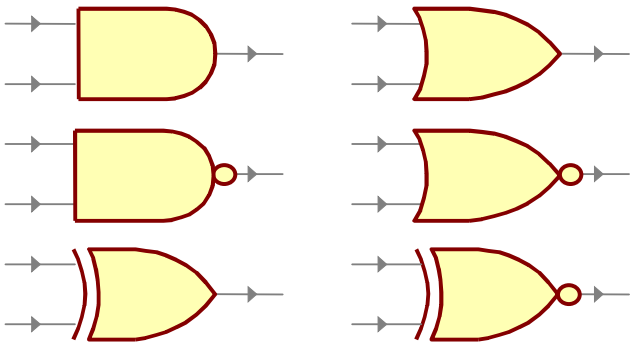
```
module defparam_example3;
    parameter bin1 = 0;
    parameter bin2 = 0;
    parameter bin3 = 0;
    initial
          $display("Ten moduł ma parametr bin1 = %d, bin2 = %d, bin3 = %d, ",
                    bin1, bin2, bin3);
endmodule

module main3;
    defparam_example3 #(1,22,12) EX31();
    defparam_example3 #(.bin3(33), .bin1(11), .bin2(22)) EX32();
endmodule
```

4.3. Projektowanie i modelowanie na poziomie bramek logicznych

4.3.1. Bramki

W języku Verilog dostępne są następujące podstawowe bramki logiczne: `and`, `nand`, `or`, `nor`, `xor`, `xnor` (Rys. 23).



Rys. 23. Bramki dostępne w języku Verilog.

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

Rys. 24 Tablice wartości dla bramek `and`, `or` i `xor`.

Poniżej pokazano podstawowy przykład wykorzystania bramki:

```
and a1(OUT, IN1, IN2);
```

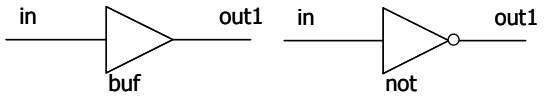
Na rys. 23 przedstawiono bramki dwuwejściowe, lecz aby otrzymać bramki o większej liczbie wejść, wystarczy podanie większej liczby wejść na liście portów:

```
nand na3(OUT, IN1, IN2, IN3);
```

Dla bramek wielowejściowych, wyjście występuje jako pierwszy port. Gdy w projekcie występuje duża liczba bramek i nie jesteśmy zainteresowani dostępem do danej bramki z poziomu symulatora, to możemy nie nadawać bramce żadnej nazwy (etykiety).

```
and (OUT, IN1, IN2); //można nie podawać nazwy bramki
```

Oprócz typowych bramek, mamy do dyspozycji także bramki buforowe typu **buf** (bufor) i **not** (inwerter) (rys. 25). Mogą mieć one jedno wejście oraz jedno lub więcej wyjść. Wyjścia są zawsze na początku listy portów i podajemy je najpierw, ostatnim portem jest wejście. Dla bramek tego typu również nie musimy nadawać nazwy poszczególnym instancjom.



buf:

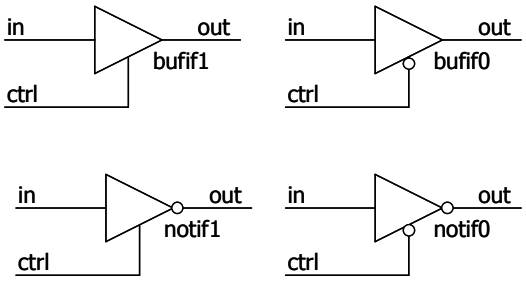
in	out
0	0
1	1
x	x
z	x

not

in	out
0	1
1	0
x	x
z	x

Rys. 25. Bramki buforowe `buf` i `not` oraz ich tabele wartości.

Bramki buforowe mogą być wyposażone w dodatkowe wejście sterujące, tak jak to występuje w bramkach typu **bufif** i **notif**:



Rys. 26. Bramki z wejściem sterującym.

Bufif1		ctrl			
		0	1	x	z
In	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

Notif1		ctrl			
		0	1	x	z
in	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

Bufif0		ctrl			
		0	1	x	z
In	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

notif0		ctrl			
		0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

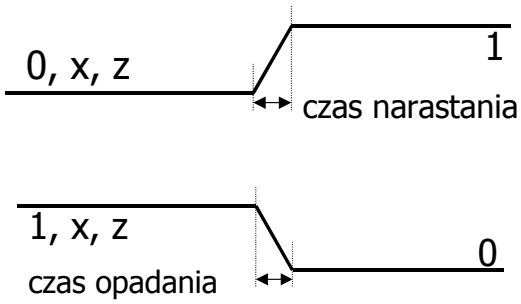
Rys. 27. Tabele wartości dla bramek z wejściem sterującym. `L` oznacza `0` lub `z`. `H` oznacza `1` lub `z` (w zależności od implementacji symulatora).

4.3.2. Opóźnienia w bramkach

W bramkach, do celów modelowania, rozróżniamy następujące czasy:

- czas narastania – czas, w jakim wyjście osiągnie stan `1` z dowolnego innego stanu;
- czas opadania – czas, w jakim wyjście osiągnie stan `0` z dowolnego innego stanu;
- czas wyłączenia – czas, w jakim wyjście osiągnie stan wysokiej impedancji `z` z dowolnego innego stanu.

Jeśli wyjście zmienia się do wartości `x`, to dzieje się to po czasie najkrótszym z trzech czasów opisanych powyżej.



Rys. Czasy opóźnienia w bramkach

Mamy możliwość określenia czasów opóźnień za pomocą trzech form zapisów z wykorzystaniem:

- jednej wartości, która oznacza czas wszystkich trzech przejść;
- dwóch wartości, są to odpowiednio czasy narastania i opadania, natomiast czas wyłączenia jest równy mniejszej z tych liczb;
- trzech wartości, są to odpowiednio: czas narastania, opadania i wyłączenia.

Jeśli nie podano żadnej wartości, wszystkie trzy czasy są równe zeru.

Taki sam czas dla trzech rodzajów przejść:

```
and #(4) g1(out1, in1, in2);
```

Określenie czasu narastania i opadania:

```
and #(3, 2) g2(out1, in1, in2);
```

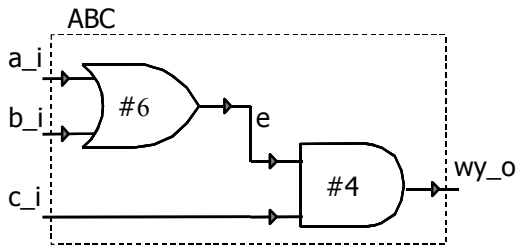
Określenie trzech czasów:

```
bufif0 #(3, 2, 5) g3(out1, in1, ctrl);
```

Dodatkowo, dla każdego z powyższych czasów (tj. dla czasu narastania, opadania i wyłączenia), można określić trzy wartości: minimalne, typowe i maksymalne. Przed rozpoczęciem symulacji użytkownik musi zdecydować, wg jakich czasów odbyć się ma symulacja (min/typ/max), a symulator wybiera wtedy odpowiednie wartości.

```
// jedno opóźnienie:
and #(4:5:6) a1(out, i1,i2);
      Min. Typ. Max.
// dwa opóźnienia:
and #(3:4:5, 5:6:7) a2(out, i1, i2);
      Min. Typ. Max.  Min. Typ. Max.
// trzy opóźnienia:
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1, i2);
      Min. Typ. Max. Min. Typ. Max.  Min. Typ. Max.
```

Rys. 28. Wartości minimalne, typowe i maksymalne dla opóźnień.



Rys. 29. Przykład opóźnień w bramkach.

Dla przykładu: moduł `ABC` ma realizować funkcję:

```
wy_o = (a_i + b_i) * c_i
```

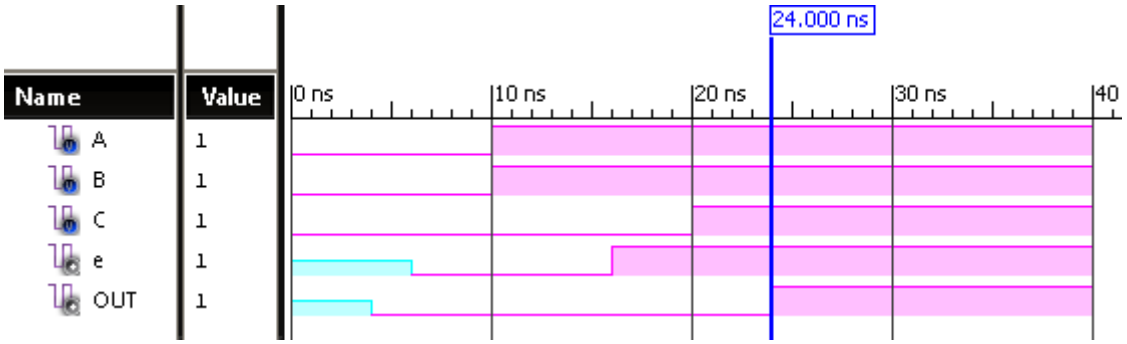
Jego schemat przedstawiono na rys. 29, a implementacja w języku Verilog wygląda następująco:

```
// definicja modułu ABC:
module ABC(wy_o, a_i, b_i, c_i);
    // deklaracje portów wej/wyj
    output wy_o;
    input a_i, b_i, c_i;
    // sieć wewnętrzna:
    wire e;
    // bramki:
    or #(6) a1(e, a_i, b_i);
    and #(4) o1(wy_o, e, c_i);
endmodule
```

Dla przetestowania modułu, przygotowano moduł testujący **stimulus**, w którym wygenerowano pobudzenia oraz osadzono moduł **ABC**:

```
module stimulus;
    // deklaracja zmiennych:
    reg A, B, C;
    wire OUT;
    ABC abc1(OUT, A, B, C); // osadzenie modułu ABC:
    // pobudzanie wejść:
    initial
    begin
      A= 1'b0; B= 1'b0; C= 1'b0;
      #10 A= 1'b1; B= 1'b1; C= 1'b0;
      #10 A= 1'b1; B= 1'b1; C= 1'b1;
      #20 $finish;
    end
endmodule
```

Wyniki symulacji pokazano na rys. 30.



Rys. 30. Wyniki symulacji (z programu Xilinx ISim).

4.4. Projektowanie i modelowanie na poziomie rejestrów

4.4.1. Przypisanie ciągłe

Modelowanie na poziomie rejestrów odbywa się praktycznie za pomocą jednego polecenia: **assign**. Polecenie to w sposób ciągły steruje siecią, wpisując do niej określoną wartość logiczną, działając podobnie jak bramki logiczne. Składnia polecenia wygląda następująco:

```
assign [<siła sygnału>][<opóźnienie>] <lista przypisań>;
```

Definiowanie siły sygnału jest opcjonalne. Definiowanie opóźnienia jest również opcjonalnie i podaje się je tak samo, jak w bramkach. Przypisanie do sieci **C** wartości **A or B** odbywa się za pomocą znaku równości:

```
assign C = A | B;
```

Z lewej strony znaku równości może występować sieć `wire` (skalar lub wektor), albo konkatenacja sieci skalarnych lub wektorowych. Takie przypisanie ciągłe jest zawsze aktywne, a wyrażenie obliczane jest zawsze, gdy dowolna ze zmiennych z prawej strony zmieni swą wartość. Wynik jest natychmiast przesyłany do sieci z lewej strony. Wyrażenia z prawej strony mogą być rejestrami, sieciami lub wywołaniami funkcji (skalary lub wektory).

```
assign C = A & B;
assign SUM[15:0] = SUM1[15:0] ^ SUM2[15:0];
```

Nawiasy klamrowe oznaczają łączenie (konkatenację) bitów:

```
assign {CARRY, SUM[15:0]} = IN1[15:0] + IN2[15:0] + CARRY_IN;
```

Zamiast najpierw deklarować sieć, a potem wykonywać do niej operację przypisania:

```
wire C;
assign C = A & B;
```

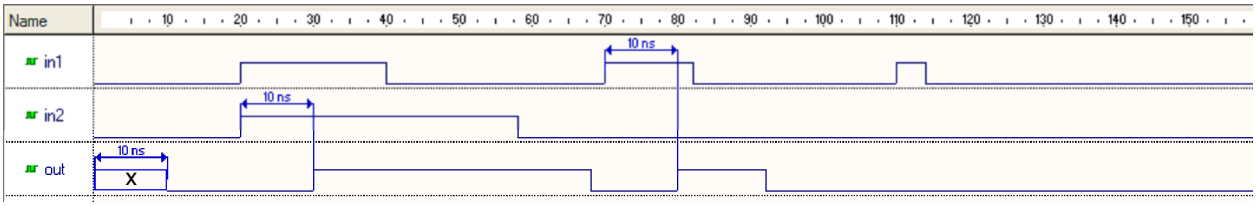
można to zrobić krócej, dzięki niejawnemu przypisaniu ciągłemu:

```
wire C = A & B;
```

4.4.2. Opóźnienia

Do polecenia `assign` można przypisać opóźnienie, które będzie w sposób stały związane z przypisaniem, co pokazuje poniższy przykład i rys. 31.

```
assign #10 out = in1 | in2;
```



Rys. 31. Wynik działania opóźnienia w poleceniu `assign`.

Uwaga: impulsy krótsze niż czas opóźnienia (w tym przypadku 10 jednostek czasowych) nie przechodzą na wyjście.

Możliwe jest również niejawne określenie przypisania z opóźnieniem:

```
wire out;
assign #10 out = in1 | in2;
```

co jest równoważne określeniu:

```
wire #10 out = in1 | in2;
```

Można zdefiniować opóźnienie dla całej sieci - każda zmiana sygnału na sieci pojawi się z tym opóźnieniem:

```
wire # 10 out;
assign out = in1 | in2;
```

Powyższy zapis jest równoważny następującemu:

```
wire out;
assign #10 out = in1 | in2;
```

W przykładach tych założono, że sygnały `in1` i `in2` zostały wcześniej zadeklarowane, np.:

```
wire in1, in2;
```

4.4.3. Wyrażenia i operatory

Polecenie `assign` wykorzystuje operatory, które zostały przedstawione w tab. 14 i 15:

Tab. 14. Operatory.

Typ	symbol	operacja	ilość operandów
arytm.	*	mnożenie	2
	/	dzielenie	2
	+	dodawanie	2
	-	odejmowanie	2
	%	modulo	2
log.	!	negacja logiczna	1
	&&	logiczne and	2
	||	logiczne or	2
relacje	>	większe niż	2
	<	mniejsze niż	2
	>=	większe lub równe	2
	<=	mniejsze lub równe	2
równość	==	równość	2
	!=	różność	2
	===	równość warunkowa	2
	!==	różność warunkowa	2

Tab. 15. Operatory (cd.).

Typ	symbol	operacja	ilość operandów
bitowe	~	negacja bitowa	1
	&	and bitowe	2
	|	or bitowe	2
	^	xor bitowe	2
	^~ lub ~^	xnor bitowe	2
redukujące	&	redukujące and	1
	~&	redukujące nand	1
	|	redukujące or	1
	~|	redukujące nor	1
	^	redukujące xor	1
	^~ lub ~^	redukujące xnor	1
przesunięcie	>>	przesunięcie w prawo	2
	<<	przesunięcie w lewo	2
łączenie	{}	łączenie	dowolna
replikacja	{{}}	replikacja	dowolna
warunek	?:	warunek	3

Przykłady wykorzystania operatorów dwuargumentowych pokazano poniżej:

```
A=8'b00110101; B=8'b00000010;
A*B // mnożenie, wynik: 8'b01101010
A/B // dzielenie, wynik: 8'b00011010 (część ułamkowa jest ucięta)
A+B // dodawanie, wynik: 8'b00110111
A-B // odejmowanie, wynik: 8'b00110011
```

Jeśli jakiś argument przyjmuje wartość `x`, to wynik też będzie miał również wartość `x`.

Operator modulo zwraca resztę z dzielenia dwóch liczb (działa podobnie jak w C):

```
15%4      // wynik: 3
15%5      // wynik: 0
```

Zaleca się nie używać ujemnych liczb w specyfikacji `<sss>'<podstawa> <nnn>`, gdyż może to prowadzić do złych wyników (liczby w tej postaci są reprezentowane jako dopełnienie do dwóch).

Operatory logiczne oznaczane są następująco:

`&&` - logiczne *and*

`||` - logiczne *or*

`!` - logiczne *not*

Operatory logiczne zwracają zawsze 1-bitową wartość: `0` (fałsz), `1` (prawda) lub `x` (niejednoznaczność). Jeśli argument nie jest równy zero, jest traktowany jako `1` logiczna (prawda). Jeśli argument jest równy zero, to jest traktowany jako `0` logiczne (fałsz).

Jeśli jakiś bit argumentu jest równy `z` lub `x`, to cały argument traktowany jest jako `x` (niejednoznaczność).

Przykłady działania operatorów logicznych dla zmiennych:

```
A=2; B=0;
```

przedstawiono poniżej:

```
A&&B      // wynik: 0
A||B      // wynik: 1
!A        // wynik: 0
!B        // wynik: 1
```

Operatory porównania również wykorzystują niejednoznaczności. Jeżeli w wyrażeniu użyto operatorów porównania, wyrażenie zwraca wartość `1`, `0` lub `x`. Operatory równości/różności są opisane w Tabeli 16:

Tabela 16. Operatory równości.

wyrażenie	opis	możliwe wartości
a == b	a jest równe b. Wynik nieznany, gdy a lub b mają wartość **x** lub **z**.	**0,1,x**
a != b	a jest różne od b. Wynik nieznany, gdy a lub b mają wartość **x** lub **z**.	**0,1,x**
a === b	a jest równe b, włączając **x** i **z**	**0,1**
a !== b	a jest różne od b, włączając **x** i **z**	**0,1**

Operatory `===` i `!==` porównują bit po bicie i biorą pod uwagę również zgodność wartości `x` oraz `z`.

Operatory bitowe (rys. 32) działają podobnie jak w języku C.

~	**negacja** bit po bicie
&	**and** bit po bicie
|	**or** bit po bicie
^	**xor** bit po bicie
^~ lub ~^	**xnor** bit po bicie

and bitowe

	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

or bitowe

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

xor bitowe

	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

xnor bitowe

	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

negacja

0	1
1	0
x	x

Rys. 32. Operatory bitowe.

W języku Verilog mamy do dyspozycji bardzo wygodne operatory redukujące: `&` (and), `~&` (nand), `|` (`or`), `~|` (`nor`), `^` (`xor`), `^~` , `~^` (`xnor`), które wymagają tylko jednego argumentu – operują na poszczególnych bitach wykonując odpowiednie operacje według tabel operatorów bitowych, działając od prawej do lewej. Np. dla `X=4'b1110`:

```
&X // 1&1&1&0 = 1'b0
|X // 1|1|1|0 = 1'b1
^X // 1^1^1^0 = 1'b1
```

Redukcja `xor` lub `xnor` może być wykorzystana przy obliczaniu liczby bitów parzystych lub nieparzystych w wektorze.

Operatory przesuwania bitów wyglądają następująco:

`>>` - przesunięcie w prawo

`<<` - przesunięcie w lewo

Przykładowo:

```
Y=X<<3 //przesuń X 3 bity w lewo
```

Nowe pozycje są wypełniane zerami (ostatni bit nie przechodzi na początek).

Kolejny z operatorów to operator łączenia. Umożliwia on łączenie poszczególnych argumentów o znanych rozmiarach. Argumentami mogą być skalary (sieci, rejestry), wektory (sieci, rejestry), części wektorów (zakres bitów) i stałe o znanym rozmiarze. Przykładowo, dla:

```
A=1'b1, B=3'b01, C=3'b11, D=3'b011
```

otrzymamy następujące wyniki:

```
Y={B, C} // wynik: Y=6'b0111
Y={A, B, C, D, 4'b1111} // wynik: Y=12'b101110111111
```

Operator replikacji umożliwia wielokrotne łączenie tego samego argumentu, co może być wyrażone za pomocą stałej oraz elementu powtarzanego w nawiasach klamrowych:

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A= 1'b1; B=2'b01; C=2'b10;
Y={4{A}}; // wynik: Y = 4'b1111
Y={ {4{A}}, {2{B}} }; // wynik: Y=8'b11110101
```

Jedyny z operatorów trójargumentowych to operator warunkowy. Wymaga on podania trzech argumentów:

```
<warunek> ? <wyrażenie_true> : <wyrażenie_false>;
```

Najpierw obliczany jest `warunek` - jeśli wynik obliczeń jest `1`, to wykonywane jest `<wyrażenie_true>`, a w przeciwnym wypadku: `<wyrażenie_false>`. Jeśli wynik wyrażenia jest `x`, to obliczane są `<wyrażenie_true>` oraz `<wyrażenie_false>`, a następnie są one porównywane bit po bicie. Jeśli bity się nie zgadzają, to w wyniku na tym miejscu będzie wartość bitu `x`.

Operator ten przypomina w pracy multiplekser lub polecenie *if-then-else*. Operator warunkowy wykorzystuje się często do przypisań warunkowych. Przykład modelu bufora trójstanowego:

```
assign data_bus = tristate ? 16'bz : data;
```

Przykład modelu multipleksera 2-do-1:

```
assign mux = C1 ? A0 : A1;
```

W tab. 17 zestawiono priorytety operatorów:

Tab. 17. Priorytety operatorów.

Operator	Symbole	Priorytet
jednoargumentowe: mnożenie, dzielenie, modulo	+ - ! ~ * / %	najwyższy
dodawanie, odejmowanie przesunięcie	+ - << >>	
porównanie równość	< <= > >= == != === !==	
redukcja logiczne	& ~& ^ ^~ | ~| && ||	
Warunkowe	?:	najniższy

4.5. Projektowanie i modelowanie na poziomie behawioralnym

4.5.1. Procedury strukturalne

Zwykłe polecenia umieszczone wewnątrz modułu wykonywane są jednocześnie, więc sposób działania nie zależy od ich kolejności w kodzie. Jest to zgodne z działaniem rzeczywistego układu cyfrowego, gdzie bramki i przerzutniki również działają w sposób równoległy w stosunku do siebie w

przeciwieństwie do programu komputerowego realizowanego przez jeden procesor, gdzie poszczególne operacje wykonywane są sekwencyjnie. Podstawową jednostką na poziomie behawioralnym jest blok. Takie bloki (traktowane jako całość), również działają w sposób równoległy, tj. kolejność umieszczenia bloków w kodzie nie jest istotna. Istotna jest natomiast kolejność poleceń wewnątrz bloków, gdyż każdy z bloków zawiera swój osobny ciąg poleceń. Polecenia wewnątrz bloku wykonywane są po kolei i muszą być zgrupowane za pomocą słów kluczowych **begin** i **end**. Jeśli w bloku występuje tylko jedno polecenie, to grupowanie jest zbędne. Mechanizm grupowania jest podobny do występującego w Pascalu grupowania za pomocą słów kluczowych **begin** i **end**, a w języku C z wykorzystaniem nawiasów klamrowych **{ }**.

Rozróżniamy dwa typy bloków: bloki **always** i **initial**. Wszystkie bloki rozpoczynają swoje działanie w czasie *t*=0. Bloki nie mogą być zagnieżdżane jedne w drugich.

Zestaw poleceń wewnątrz bloku **initial** wykonuje się tylko jeden raz podczas symulacji, począwszy od czasu *t*=0. Działanie każdego bloku **initial** kończy się niezależnie od pozostałych, a po wykonaniu ostatniej instrukcji w bloku, dany blok **initial** przestaje działać. Ponieważ polecenia wewnątrz bloku wykonywane są po kolei, to opóźnienie opisane jako **#<czas>**, oznacza opóźnienie od poprzedniej instrukcji w bloku. Bloki **initial** wykorzystuje się m.in. do inicjalizacji, monitorowania.

```
module stimulus;
    // deklaracja zmiennych:
    reg A, B, C;
    initial
    begin
          A= 1'b0; B= 1'b0; C= 1'b0;
          #10 A= 1'b1; B= 1'b1; C= 1'b0;
          #10 A= 1'b1; B= 1'b1; C= 1'b1;
    end
endmodule
```

Wszystkie bloki **always** rozpoczynają działanie równocześnie w czasie *t*=0. Są one od siebie niezależne i w przeciwieństwie do bloków **initial**, wykonują się w sposób ciągły, tj. po zakończeniu ostatniej instrukcji wykonuje się pierwsza.

```
module clk_generator;
    reg clk;
    initial
          clk = 1'b0;
    always
          #10 clk = ~clk;
endmodule
```

4.5.2. Przypisanie proceduralne

Przypisanie proceduralne uaktualnia wartości zmiennych typu `reg`, `integer`, `real` lub `time`. Wartości zmiennych nie ulegną zmianie, aż do następnego przypisania proceduralnego. Przypisanie odbywa się jednorazowo – w odróżnieniu od przypisania `assign`, które działa w sposób ciągły. Przypisania proceduralne, tak jak wszystkie wyrażenia behawioralne, muszą być umieszczone wewnątrz bloku `initial` lub `always`.

Wyróżnia się dwa typy przypisań: *blocking* i *nonblocking*. Składnia przypisania proceduralnego wygląda następująco:

`<lvalue> = <wyrażenie>` - przypisanie typu *blocking*,

`<lvalue> <= <wyrażenie>` - przypisanie typu *nonblocking*.

Przypisania typu *blocking* (oznaczane znakiem `=`) wykonywane są według kolejności ich umieszczenia w programie. Nie wstrzymują one działania innych, równolegle działających, bloków. Przykład przypisania typu *blocking* pokazano poniżej oraz na rys. 33.

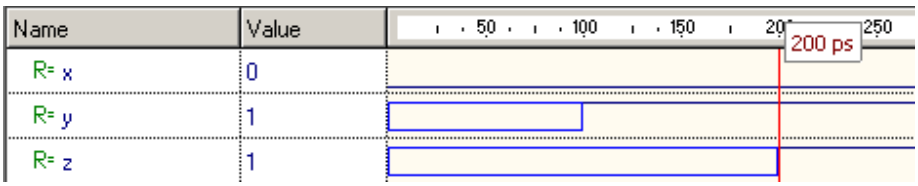
```
module test;
    reg x, y, z;
    initial
    begin
          x = 1'b0;
          y = #100 1'b1;
          z = #100 1'b1;
    end
endmodule
```

W przykładzie zastosowano tzw. opóźnienie wewnętrzne w instrukcji:

```
y = #100 1'b1;
```

które działa wg następującego algorytmu:

1. oblicz i zapamiętaj wartość z prawej strony (tj. w naszym przypadku jest to stała `1'b1`);
2. poczekaj 100 jednostek czasu;
3. po odczekaniu 100 jednostek czasu przypisz wartość zapamiętaną w kroku 1. do zmiennej `y`.

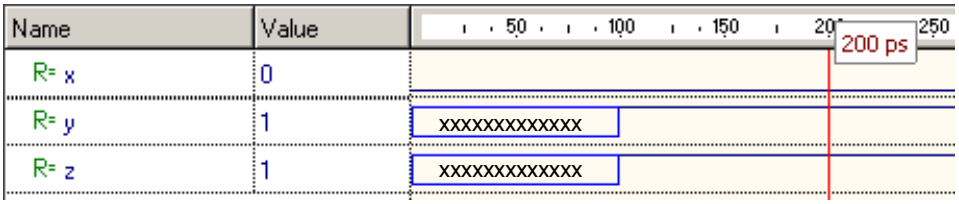


Rys. 33. Wynik działania przykładu z przypisaniem typu *blocking*.

Przypisanie typu *nonblocking* (`<=`) pozwala na wykonanie przypisań bez wstrzymywania działania pozostałych instrukcji. Symbol `<=` jest taki sam, jak operator warunkowy "mniejsze lub równe", więc

jego znaczenie rozstrzygane jest w zależności od kontekstu. Wszystkie przypisania typu *nonblocking* występujące w tym samym kroku czasu symulatora będą wykonane równocześnie. Zazwyczaj symulator wykonuje wszystkie przypisania *nonblocking* na końcu danego kroku czasowego. Przykład przypisania typu *nonblocking* pokazano poniżej oraz na rys. 34.

```
module test;
    reg x, y, z;
    initial
    begin
      x <= 1'b0;
      y <= #100 1'b1;
      z <= #100 1'b1;
    end
endmodule
```



Rys. 34. Wynik działania przykładu z przypisaniem typu *nonblocking*.

4.5.3. Sterowanie wykonaniem instrukcji

Verilog zawiera mechanizmy sterowania za pomocą zdarzeń. Najprostsze, zwykłe sterowanie przy użyciu zdarzeń polega na wstrzymaniu działania części układu do momentu wystąpienia określonego zdarzenia. Do sterowania zdarzeniami służy symbol `@`, a instrukcje mogą reagować na:

- zmianę wartości sygnału,
- rosnące zbocze (`posedge`),
- opadające zbocze (`negedge`).

Poniżej przedstawiono przykłady sterowania przy użyciu zwykłych zdarzeń wraz z komentarzem:

```
@clk q = d; // poczekaj na zmianę sygnału clk i przypisz q=d
@(posedge clk) q = d;       // q=d gdy clk zmienia się:
                            // 0->1,0->x,0->z,x->1,z->1
@(negedge clk) q = d;       // q=d gdy clk zmienia się:
                            // 1->0,1->x,1->z,x->0,z->0
q=@(posedge clk) d;         // d jest obliczane natychmiast,
                            // a wartość jest przypisywana do q
                            // podczas rosnącego zbocza clk.
```

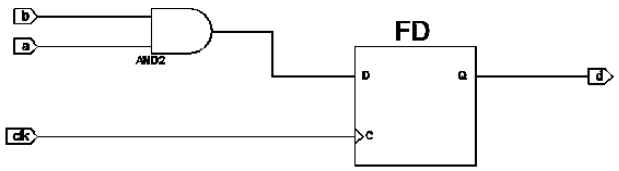
Można również sterować przy użyciu wielu zdarzeń, łącząc je operatorem `or`, wyzwolenie może wtedy nastąpić poprzez jeden z kilku sygnałów. Poniżej podano przykład opisu zatrzasku wyzwalanego poziomem z asynchronicznym resetem:

```
always @(rst or clk or d)
    // czekaj na zmianę sygnału rst, clk lub d
begin
    if (rst) // jeśli rst ma stan wysoki, ustaw q=0
          q = 1'b0;
    else if (clk) // jeśli clk ma stan wysoki (tj. wystąpiło narastające zbocze)
          q = d;
end
```

4.5.4. Różnica pomiędzy przypisaniem *blocking* i *nonblocking*

Należy zwrócić uwagę na różnicę pomiędzy przypisaniami typu *blocking* i *nonblocking*. Poniżej podano dwa przykłady wraz z wynikiem syntezy logicznej różniące się wyłącznie rodzajem przypisania, gdzie można zaobserwować skutki różnic w działaniu pomiędzy tymi przypisaniami:

```
module test(clk, a, b, d);
    input a;
    input b;
    input clk;
    output d;
    reg c;
    reg d;
    always @(posedge clk)
          begin
                c = a & b;
                d = c;
          end
endmodule
```

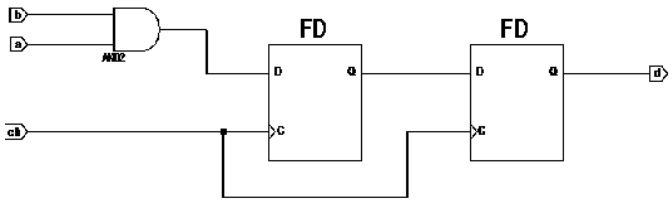


Rys. 35. Wynik syntezy przykładu z przypisaniem typu *blocking*.

Po zamianie przypisania typu *blocking* na przypisanie *nonblocking*, wynik syntezy będzie inny:

```
module test(clk, a, b, d);
    input a;
    input b;
    input clk;
    output d;
    reg c;
    reg d;
    always @(posedge clk)
```

```
        begin
              c <= a & b;
              d <= c;
        end
endmodule
```



Rys. 36. Wynik syntezy przykładu z przypisaniem typu *nonblocking*.

4.5.5. Wyrażenie warunkowe *if*

Wyróżniamy trzy wersje wyrażenia warunkowego typu `if`:

Najprostsza wersja nie posiada części `else`:

```
if (<wyrażenie>) polecenie_true;
```

Wersja druga zawiera pojedynczą część `else`:

```
if (<wyrażenie>) polecenie_true; else polecenie_false;
```

Forma najbardziej rozbudowana wygląda następująco:

```
if (<wyrażenie1>) <polecenie_true1>;
else if (<wyrażenie2>) <polecenie_true2>;
else if (<wyrażenie3>) <polecenie_true3>;
else <polecenie_domyślne>;
```

4.5.6. Wyrażenie typu *case*

Wyrażenie typu `case` ma następującą składnię:

```
case (wyrażenie)
    alternatywa_1: polecenie1;
    alternatywa_2: polecenie2;
    alternatywa_3: polecenie3;
    ...
    ...
    default: polecenie_domyślne;
endcase
```

Pierwsze dopasowanie wyrażenia do alternatywy spowoduje uruchomienie odpowiedniego polecenia, a wtedy pozostałe alternatywy i polecenia nie będę rozpatrywane.

Wartości porównywane są bit po bicie, także wartości `x` i `z` są brane pod uwagę. Gdy nie zgadza się liczba bitów, to krótsze wyrażenie uzupełniane jest zerami, aby długość była identyczna. Dopuszcza się

oddzielenie przecinkiem kilku alternatyw, natomiast wielokrotne użycie części `default` nie jest dozwolone. Wyrażenia `case` mogą być zagnieżdżone.

Są możliwe dwie modyfikacje wyrażenia `case`:

- `casez` traktuje wszystkie wartości `z` jako nieznaczące. Wszystkie bity o wartości `z` można też zapisać za pomocą `?`.
- `casex` traktuje wszystkie wartości `x` i `z` jako nieznaczące.

4.5.7. Pętle

Pętla *while*

Pętla `while` uruchamia się dopóki wyrażenie jest prawdziwe. Jeśli testowane wyrażenie jest fałszywe od początku, to pętla `while` nie uruchomi się w ogóle. Przykład wykorzystania pętli `while`:

```
initial
begin
      licznik=0;
      while (licznik < 12)
      begin
            $display("Licznik=%d", licznik);
            licznik = licznik + 1;
      end
end
```

Pętla *for*

Pętla `for` zawiera trzy części:

- warunek początkowy;
- warunek trwania w pętli;
- przypisanie zmieniające zmienną pętli.

```
initial
      for (licznik=0; licznik<12; licznik=licznik+1)
      $display("Licznik=%d", licznik);
```

Pętla *repeat*

Pętla `repeat` wykonuje polecenie określoną liczbę razy i musi zawierać liczbę powtórzeń (liczba może być określona poprzez stałą, zmienną lub sygnał). Liczba powtórzeń jest obliczana tylko raz na początku uruchamiania polecenia `repeat`.

```
initial
begin
      licznik=0;
      repeat(12)
      begin
```

```
            $display("Licznik=%d", licznik);
            licznik=licznik+1;
        end
    end
```

Pętla *forever*

Pętla `forever` wykonuje się aż do napotkania polecenia `$finish`. Jest ona równoważna pętli `while(1)`. Pętlę `forever` najczęściej używa się wraz z poleceniami kontrolującymi czas symulacji. Gdyby ich nie było, pętla wykonywałaby się w nieskończoność, wstrzymując pozostałe polecenia. W poniższym przykładzie zależy nam na wykonywaniu przypisania `#10 clk = ~clk` w nieskończoność, więc wykorzystanie pętli `forever` jest jak najbardziej uzasadnione:

```
initial
begin
    clk = 1'b0;
    forever #10 clk = ~clk;
end
```

4.5.8. Bloki sekwencyjne i równoległe

W języku Verilog wyróżniamy dwa typy bloków: sekwencyjne i równoległe.

Bloki sekwencyjne są zgrupowane za pomocą słów kluczowych `begin` i `end`. Polecenia umieszczone wewnątrz takiego bloku wykonywane są w kolejności, w jakiej są tam umieszczone. Następne polecenie jest wykonywane, gdy zakończy się poprzednie (z wyjątkiem przypisań *nonblocking* z wewnętrznym opóźnieniem). Jeśli określone jest opóźnienie, to jest ono liczone względem czasu zakończenia poprzedniego polecenia w bloku.

Bloki równoległe są zgrupowanie z wykorzystaniem słów kluczowych: `fork` i `join`. Wyrażenia wewnątrz takich bloków wykonywane są jednocześnie, a kolejność wykonywania tych poleceń jest określona przez opóźnienia lub wydarzenia związane z konkretnymi poleceniami. Opóźnienia przypisane poszczególnym poleceniom są liczone względem czasu wejścia do bloku.

```
reg a;
initial
fork
    a=1'b0;          // wykonuje się w czasie=0
    #5 a=1'b1;       // wykonuje się w czasie=5
    #10 a=1'b0;      // wykonuje się w czasie=10
    #15 a=1'b1;      // wykonuje się w czasie=15
join
```

4.6. Zadania i funkcje

W celu zdefiniowania fragmentu kodu, który będzie wielokrotnie wykorzystany, można się posłużyć funkcją (**function**) lub zadaniem (**task**). Funkcja i zadanie różnią się między sobą istotnymi cechami i możliwościami.

Funkcja posiada następujące cechy:

- funkcja może uruchamiać inną funkcję, nie może uruchamiać zadania,
- czas wykonywania funkcji jest zawsze zerowy,
- funkcja nie może zawierać opóźnień, wydarzeń ani innych struktur kontroli czasowej,
- funkcja musi posiadać przynajmniej jeden argument typu **input**,
- funkcja zawsze zwraca pojedynczą wartość.

Natomiast zadanie posiada cechy:

- zadanie może uruchamiać inne zadania i funkcje,
- zadanie może wykonywać się przez czas niezerowy,
- zadanie może zawierać opóźnienia, wydarzenia i inne struktury kontroli czasowej,
- zadanie może mieć zero lub więcej argumentów typu **input**, **output** lub **inout**,
- zadanie nie zwraca wartości, ale może zwracać wartości poprzez argumenty typu **output** lub **inout**.

Zarówno funkcja, jak i zadanie muszą być zdefiniowane w module i są lokalne dla danego modułu. Funkcja i zadanie mogą zawierać lokalne zmienne, rejestry, zdarzenia itp., ale nie mogą zawierać zmiennych typu **wire**. Wewnątrz funkcji i zadania dozwolone jest wykorzystanie wyłącznie wyrażeń behawioralnych, ale bez bloków **initial** czy **always**.

Funkcje definiuje się za pomocą słów kluczowych **function** i **endfunction**. Funkcje wykorzystuje się, gdy:

- nie ma opóźnień lub zdarzeń w procedurze;
- procedura zwraca dokładnie jedną wartość;
- procedura posiada przynajmniej jeden argument wejściowy.

Gdy zdeklarowano funkcję, automatycznie tworzona jest zmienna typu **reg** o takiej samej nazwie jak nazwa funkcji. Wartość funkcji jest zwracana w ten sposób, że wewnątrz funkcji odpowiednio ustawiana jest ta zmienna typu **reg**. Funkcja nie może wywoływać innych zadań (ponieważ mogą zawierać opóźnienia), ale funkcja może wywoływać inne funkcje. Przykład definicji funkcji:

```
function[11:0] sqr;
   input[5:0] value;
   begin
        sqr = value * value;
```

```
    end
endfunction
```

Wykorzystanie funkcji w bloku:

```
always @(a)
    b = sqr(a);
```

Zadania definiuje się za pomocą słów kluczowych **task** i **endtask**. Zadania wykorzystuje się, gdy:

- potrzebne jest opóźnienie lub zdarzenie w procedurze,
- procedura posiada zero lub więcej argumentów wyjściowych,
- procedura nie posiada argumentów wejściowych.

Definicję zadania realizuje się w oparciu o następującą składnię:

```
task <nazwa_zadania>;
    <deklaracja>*
    <polecenia>
endtask
```

Argumenty zadania mogą mieć kierunek **input**, **output** lub **inout**. Argumenty **input** i **inout** są przekazywane do zadania. Argumenty **output** i **inout** są zwracane z powrotem po zakończeniu zadania. Zadanie może wywoływać inne zadania lub funkcje i może operować na zmiennej typu **reg** zdefiniowanej w module. Przykład definicji zadania i późnieszego jego wykorzystania do wyznaczania kwadratu wartości przedstawiono poniżej:

```
module tb;
reg [11:0] b;
reg [5:0] a;
// definicja zadania sqr
task sqr;
   input [5:0] value;
    output [11:0] sqr;
   begin
      sqr = value * value;
   end
endtask
// wywołanie zadania sqr
always @(a)
    sqr(a,b);
endmodule
```

4.7. Techniki modelowania

4.7.1. Proceduralne przypisanie ciągłe

Techniki modelowania służą do uruchamiania modułów z wykorzystaniem symulatorów i *test bench*'y poprzez sterowanie wartościami sygnałów. Jedną z takich metod jest proceduralne przypisanie ciągłe, którego działanie polega na ciągłym podawaniu do rejestru wartości stałej lub wyrażenia przez określony okres czasu. Proceduralne przypisanie ciągłe różni się od zwykłego przypisania proceduralnego, które jednorazowo przypisuje wartość do rejestru, a rejestr zachowuje daną wartość, aż do następnego wpisu.

Są dwa typy przypisań proceduralnych ciągłych:

- typ 1: **assign** i **deassign**
- typ 2: **force** i **release**

W przypisaniu typu 1 (**assign** i **deassign**), lewa cześć przypisania może być tylko rejestrem lub połączeniem rejestrów, nie może być częścią bitów sieci ani też macierzą rejestrów. To przypisanie ciągłe ma większy priorytet niż zwykłe przypisania proceduralne. Przykład zastosowania przypisania proceduralnego ciągłego typu 1 przedstawiono poniżej:

```
always @(preset)
    if (preset) assign q = 1'b1;
    else deassign q;
```

Przypisania typu 2 (**force** i **release**) pozwalają zapisywać wartości zarówno do rejestrów jak i do sieci. W przypadku sterowania rejestrem, **force** ma większy priorytet niż przypisania proceduralne czy przypisania proceduralne ciągłe. Po poleceniu **release** wartość rejestru będzie pamiętana, ale będzie już można ją zmienić. Dla sieci, przypisanie **force** nadpisuje jakiekolwiek inne wartości, a wystąpienie **release** powoduje powrót do normalnej wartości wymuszanej w sieci. Przykład wykorzystania przypisania proceduralnego ciągłego typu 2 przedstawiono poniżej:

```
always @(preset)
    if (preset)
    begin
          force FF.q = preset;
    end
    else
    begin
          release FF.q;
    end
```

4.7.2. Skala czasu

Do tej pory we wszystkich przykładach zawierających opóźnienia czasowe nie określano jednostek czasu tych opóźnień. Wygodnie jest określać te czasy w jednym module w [ns] a w innym w [ms]. Verilog umożliwia definicję jednostki czasu dla modułu za pomocą polecenia:

```
`timescale <jednostka_czasu>/<dokładność>
```

gdzie:

<jednostka_czasu> - jednostka czasu dla opóźnień;

<dokładność> - dokładność, z jaką są zaokrąglane opóźnienia podczas symulacji.

Do określenia obydwu tych parametrów można wykorzystać tylko trzy wartości liczbowe: 1, 10 i 100.

Przykłady:

```
`timescale 10 ns / 1 ns
`timescale 100 us / 10ns
```

4.7.3. Praca z plikami

Polecenie systemowe **$fopen** umożliwia otwieranie plików:

```
<file_handle> = $fopen("<nazwa_pliku>");
```

Zadanie systemowe **$fopen** zwraca 32-bitowy deskryptor typu **integer**. W deskryptorze tym tylko jeden bit jest ustawiony. Każde wywołanie funkcji **$fopen** otwiera nowy kanał z kolejnym bitem ustawionym na 1, aż do 31. Zaletą takiego rozwiązania jest możliwość selektywnego zapisu do kilku plików jednocześnie. Standardowe wyjście (*stdout*) reprezentowane jest przez deskryptor z ustawionym pierwszym, najmniej znaczącym bitem (tzw. kanał 0), który jest zawsze otwarty.

Do pisania do pliku służą następujące zadania:

```
$fdisplay(<deskryptor>, format, p1, p2, ... ,pn);
$fmonitor(<deskryptor>, format, p1, p2, ... ,pn);
```

gdzie **format** to łańcuch formatujący, a **p1**, **p2**,..., **pn** to zmienne, sygnały lub łańcuchy. Działają one podobnie jak **$display** i **$monitor**.

Deskryptor może być kombinacją kilku deskryptorów jednokanałowych - Verilog będzie pisał do wszystkich kanałów, które mają ustawioną 1 na odpowiednim miejscu.

Zamykanie pliku odbywa się za pomocą polecenia:

```
$fclose(<handle>);
```

4.8. Verilog 2001

W roku 2001 zmodyfikowano istniejący standard języka Verilog z roku 1995 poprzez dodanie szeregu nowych możliwości. Poniżej wymieniono najważniejsze zmiany. Aby polecenia wchodzące w skład tego standardu były obsługiwane, w niektórych narzędziach należy włączyć opcję obsługi standardu Verilog 2001.

4.8.1. Blok konfiguracji

Dodanie możliwości wstawienia specjalnego bloku konfiguracyjnego umożliwia określenie dodatkowych parametrów dla projektowanego systemu, np.:

```
config cfg4
    /* określenie modułu toplevel: */
    design rtl_Main.top
    /* domyślna kolejność przeszukiwania bibliotek: */
    default lib20um lib_rtl lib_gate;
    /* wskazanie biblioteki dla konkretnych modułów:*/
    instance counter.dut.g2 lib_rtl lib_gate;
endconfig
```

4.8.2. Polecenie *generate*

Polecenie `generate` umożliwia automatyczne osadzenie wielu modułów, które są osadzane według powtarzalnego wzorca:

```
`define MAX_I 5
wire [1:`MAX_I] a, b, c, sum;
genvar i;
generate
    for(i=0; i<`MAX_I; i=i+1)
    begin:add_bit
      wire net1,net2,net3;
      xor g1 ( net1, a[i], b[i]);
      xor g2 (sum[i],net1, c[i]);
      and g3 ( net2, a[i], b[i]);
      and g4 ( net3, net1, c[i]);
      or g5 (c[i+1], net2, net3);
    end
endgenerate
```

Dodatkowo można stosować warunkowe osadzenie modułów:

```
generate
    if((a_width < 8) || (b_width < 8))
          SMALL_multiplier #(a_width, b_width) u1 (a, b, result);
    else
          WIDE_multiplier #(a_width, b_width) u1 (a, b, result);
endgenerate
```

4.8.3. Nowy sposób indeksowania wektorów

Wprowadzono nowy sposób indeksowania wektorów za pomocą notacji podstawa/szerokość:

```
[base_expr +: width_expr]
[base_expr -: width_expr]
```

`base_exp` może być zmienne, lecz `width_exp` musi być stałe. Przykład:

```
wire [7:0] dataN = word[byte_num*8 -: 8];
```

4.8.4. Tablice wielowymiarowe

Stary standard języka Verilog z roku 1995 umożliwiał definicję 1-wymiarowych tablic *n*-bitowych o zmiennych typu `reg`, np:

```
reg [7:0] array1 [0:255];
wire [7:0] out1 = array1[address];
```

W standardzie Verilog 2001 dodano możliwość korzystania z tablic wielowymiarowych, np. definicja trójwymiarowej tablicy 8-bitowych zmiennych typu `wire` wygląda następująco:

```
wire [7:0] array3 [0:255][0:255][0:15];
wire [7:0] out3 = array3[addr1][addr2][addr3];
```

Dodano także możliwość indeksowania zakresów w tablicach, np.:

```
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[10][17][31:24];
```

4.8.5. Operacje na liczbach *signed*

W standardzie Verilog 2001 uporządkowano użycie liczb ze znakiem, wprowadzając możliwość zadeklarowania zmiennych typu `reg`, `wire` i `integer` jako `signed`. Możliwa jest konwersja `signed` – `unsigned`, dodano także operatory przesunięcia arytmetycznego:

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
16'hC501        // 16-bitowa liczba hex unsigned
16'shC501       // 16-bitowa liczba hex signed
reg [63:0] a;   // wartość unsigned
always @(a) begin
    result1 = a / 2;             // operacja na wartościach unsigned
    result2 = $signed(a) / 2;    // operacja na wartościach signed
end
// D=8'b10100011
D >> 3    // logical shift --> 8'b00010100
D >>> 3   // arithmetic shift --> 8'b11110100
```

4.8.6. Inne zmiany

W Verilog 2001 dodano operator potęgowania `**`, ułatwiono także zapis reakcji na wszystkie zdarzenia w bloku kombinacyjnym:

```
always @* // dla układów kombinacyjnych
    if (sel)
```

```
            y = a;
      else
            y = b;
```

Uproszczono zdarzenia typu `or`, gdzie umożliwiono proste wymienianie sygnałów po przecinku. Dotychczasowy zapis:

```
always @(a or b or c or d or sel)
```

jest równoważny:

```
always @(a, b, c, d, sel)
```

Połączono deklarację portu z deklaracją typu danych, dzięki czemu nie trzeba osobno deklarować portu jako `input` a potem `reg`:

```
module mux8 (y, a, b, en);
    output reg [7:0] y;
    input wire [7:0] a, b;
    input wire en;
```

Umożliwiono również deklarację portów podobną do występującej w ANSI-C:

```
module mux8 (   output reg [7:0] y,
                input wire [7:0] a,
                input wire [7:0] b,
                input wire en );
```

5. Język VHDL

5.1. Pojęcia podstawowe

Skrót VHDL pochodzi od nazwy ***V****ery High Speed Integrated Circuits* ***H****ardware* ***D****escription* ***L****anguage*. Prace badawcze nad językiem VHDL, sponsorowane przez Departament Obrony USA, były prowadzone w latach 1983-1985 przez firmy IBM, Intermetics oraz Texas Instruments. W efekcie tych prac powstał język o składni bazującej na językach Ada i Algol, którego standard IEEE (ang. *Institute of Electrical and Electronics Engineers*) o numerze 1076-1987 został zatwierdzony w 1987r. Standard ten opisuje dokładnie składnię języka, nie określa natomiast typów danych, w związku z tym producenci narzędzi CAD definiowali własne typy danych wykorzystywane podczas symulacji. Z tego powodu język był trudny do wykorzystania i aby umożliwić swobodną wymianę modeli symulacyjnych, w roku 1993 opracowano standard 1164, w którym zdefiniowano standardowe typy danych oparte o logikę złożoną z 9 wartości MVL9. Typy danych zostały zgrupowane w pakiecie zwanym *std_logic_1164*. W tym samym roku znowelizowano standard 1076-1987, który otrzymał oznaczenie 1076-1993 i w tej formie jest on najbardziej rozpowszechniony. W latach 2000, 2002 oraz 2008 wprowadzono dodatkowe modyfikacje standardu języka VHDL.

Oprócz tego istnieją jeszcze inne rozszerzenia języka VHDL zawarte w standardach IEEE:

- 1076.1 – VHDL-AMS – do języka VHDL dodano możliwość symulacji i modelowania układów analogowych;
- 1076.2 – zawiera typy danych potrzebne do modelowania i symulacji na wysokim poziomie (*math_real, math_complex*);
- 1076.3 – zawiera typy danych wykorzystywane podczas syntezy (*numeric_bit, numeric_std*).
- 1076.4 - precyzuje sposób opisywania opóźnień występujących w układach cyfrowych (*vital*).

Język VHDL [7][8][9][10] umożliwia dopisywanie własnych procedur w języku C współdziałających z wewnętrznymi strukturami VHDL za pomocą tzw. VHPI (ang. *VHDL Procedural Interface*). Dzięki temu można wywoływać własne funkcje C z poziomu VHDL i odwrotnie: z poziomu programu w języku C można łączyć się z kodem napisanym w VHDL.

Język VHDL umożliwia opis układu na różnych poziomach abstrakcji jednocześnie, przez co można mieszać opis na wysokim poziomie abstrakcji z opisem niskopoziomowym. Podział opisu na różne poziomy abstrakcji jest sprawą umowną, ale w literaturze najczęściej występuje podział na następujące poziomy:

- abstrakcyjny poziom behawioralny,
- poziom przesłań międzyrejestrowych RTL (ang. *Register Transfer Level*),

- poziom strukturalny.

Abstrakcyjny poziom behawioralny

Abstrakcyjny poziom behawioralny (ang. *behavior* – zachowanie) jest to najwyższy poziom opisu systemu w ramach VHDL, gdzie całą uwagę skupia się na to, w jaki sposób wyjście układu reaguje na sygnały wejściowe, jednakże nie jest brana pod uwagę realizacja fizyczna systemu (budowa wewnętrzna czy podział na bloki funkcjonalne). W sytuacjach typowych, w tej fazie projektowania nie korzysta się z wielu szczegółów, takich jak częstotliwości zegarów, długości słów oraz przyporządkowanie poszczególnych bitów. Niektóre sygnały np. zegarowe, nie muszą być wykorzystywane w sposób standardowy, a szyny danych mogą być traktowane jako liczby całkowite lub rzeczywiste bez precyzowania liczby bitów.

Dla przykładu: pewna operacja może zostać nazwana „*ADD*” bez potrzeby definiowania kodu binarnego, który ją reprezentuje. Można sobie także wyobrazić system cyfrowy oczekujący na pewną wiadomość, a gdy się ona pojawi na szynie danych ustawiane są pewne sygnały wyjściowe na okres 80 ms. To, jak wykonane zostanie opóźnienie o wartości 80 ms i ustawienie poszczególnych wyjść systemu jest przedmiotem szczegółowej implementacji sprzętowej, ale wyrażenie tego na poziomie behawioralnym opisu VHDL nie jest wymagane.

Używanie opisu VHDL na tym poziomie jest podobne do programowania w typowych językach takich jak PASCAL czy C++ z wyjątkiem tego, że VHDL ma właściwość obsługi operacji współbieżnych oraz sekwencyjnych. Programowanie behawioralne jest używane także podczas symulacji i testowania dla zapewnienia możliwie dużej uniwersalności.

Poziom przesłań miedzyrejestrowych RTL

Poziom przesłań miedzyrejestrowych RTL jest niższym poziomem abstrakcji niż poziom behawioralny i odpowiada szczegółowemu opisowi blokowemu w tradycyjnej metodzie projektowania, w którym operuje się blokami funkcjonalnymi oraz ich funkcjami, sygnałami wejściowymi i wyjściowymi oraz szynami danych. W projekcie muszą być zdefiniowane sygnały zegarowe oraz zerujące (ang. *reset*), a szyny danych oraz elementy pamiętające (przerzutniki, liczniki, pamięci) muszą mieć przypisane konkretne liczby bitów. Użycie nazwy RTL (*Register Transfer Level*) zmieniło się od momentu pierwszego jej wprowadzenia i obecnie pojęcie to może, ale nie musi, mieć związku z rejestrami. Na poziomie przesłań miedzyrejestrowych zakłada się, że każdy sygnał logiczny lub szyna danych ma pewien ustalony stan lub wartość, które są funkcją wejść oraz to, że wartości logiczne w systemie są przenoszone z wejścia na wyjście. Ogólna reprezentacja RTL takiego procesu ma postać:

```
Output<=Input
```

gdzie `Input` może być złożoną kombinacją różnych typów funkcji a znak `<=` reprezentuje przesłanie sygnałów. Ponieważ poziom RTL pozwala na całkiem wysoki poziom abstrakcji projektowania układów ASIC, metoda ta może zapewnić bardzo dobre rezultaty z punktu widzenia efektywności projektowania. Projekty z opisem na poziomie RTL mogą być niezależne od technologii i wytwórcy układów scalonych, co zapewnia uniwersalność oraz możliwość wielokrotnego używania tych samych projektów. Główne komercyjne systemy syntezy VHDL akceptują opis na poziomie RTL.

Poziom opisu strukturalnego

Składnia VHDL na poziomie opisu strukturalnego jest podobna do listy połączeń, która przedstawia projekt jako strukturę „elementów” (ang. *components*) połączonych sieciami sygnałowymi (ang. *interconnections*). Elementy mogą być prostymi bramkami, przerzutnikami lub większymi blokami opisanymi kodem na poziomie behawioralnym lub RTL. VHDL dopuszcza jednoczesne występowanie opisów różnego poziomu. Na przykład, w projektowaniu typu ‘*top-down*’ można rozpocząć od opisu abstrakcyjnego a następnie przeprowadzać oceny i symulacje różnych wyborów architektur i algorytmów. W czasie tego procesu, kiedy poszczególne bloki przechodzą kolejne kroki projektowania, można je zamieniać na opisane na poziomie RTL lub strukturalnym ciągle posiadając możliwość symulacji i weryfikacji poprawności działania całego systemu. W końcu, przy użyciu narzędzi VHDL, cały układ ASIC, łącznie z elementami wejścia/wyjścia (I/O), może być symulowany na poziomie bramek. Możliwa jest także symulacja łącznie z innymi układami ASIC oraz dodatkowymi układami dyskretnymi.

5.1.1. Podstawowe zasady składni języka VHDL

Poszczególne słowa języka VHDL rozdzielone są tzw. białą spacją (spacja, tabulator, koniec linii), poza tym białe spacje są ignorowane (wyjątek to łańcuchy). W przeciwieństwie do języka Verilog, nie są rozróżniane duże i małe litery w nazwach zmiennych i słowach kluczowych, np.: słowo kluczowe `ENTITY` jest tożsame z `EntitY`, a wartość `TRUE` jest równa wartości `true`. Powyższa zasada nie dotyczy wartości typów wyliczeniowych umieszczonych w pojedynczym apostrofie: np. w definicji typu określono jedną z wartości jako `'Z'`, to do zmiennej `A` tego typu nie można przypisać:

```
A <= 'z';
```

ale należy przypisać:

```
A <= 'Z';
```

W języku VHDL możemy komentować wyłącznie pojedyncze linie; po znakach "`--`" cały tekst do końca linii jest traktowany jako komentarz:

```
rst_i : in std_ulogic; --Komentarz
-- Dalszy ciąg komentarza
```

5.1.2. Identyfikatory

VHDL ma listę słów zarezerwowanych (tab. 18), które są częścią języka i nie mogą być używane dla nazw bloków, sygnałów lub innych identyfikatorów. Dodatkowo zbiór znaków ASCII, który może być używany w nazwach i identyfikatorach jest ograniczony do liter i cyfr plus znak podkreślenia z pewnymi wyjątkami:

- nazwa musi zaczynać się od litery,
- znak podkreślenia (_) może być używany dla zwiększenia czytelności zapisu, ale nie może być ostatnim znakiem nazwy ani nie można używać dwóch znaków podkreślenia obok siebie. `reg_1` and `data_07` są poprawne, ale `reg__1` and `data_` nie są dozwolone,
- przerwy (spacje) nie są dozwolone w nazwach.

Większość innych znaków z klawiatury jest używanych dla operacji (+,-,*,/,&) i różnego rodzaju ograniczników (;,()<=#) i dlatego nie mogą być użyte w nazwach.

Długość nazw używanych dla sygnałów, procesów, jednostek projektowych itd. jest ważna ze względu na zachowanie zgodności z innymi narzędziami projektowymi, zewnętrznymi względem VHDL. Normalnym zbiorem wyjściowym (po syntezie, czyli zamianie kodu HDL na schemat elektryczny) jest lista połączeń, która może być wykorzystana w celu weryfikacji układu ASIC w symulacji na poziomie bramek logicznych, a nazwy nadane sygnałom mają swoje odbicie w liście połączeń. Wiele komercyjnych systemów ogranicza dopuszczalną długość nazw sygnałów, co może doprowadzić do obcięcia nazwy stosowanej wcześniej w trakcie kodowania VHDL. Ponieważ jest to zjawisko niepożądane, jeśli nawet nie dyskwalifikujące, należy ograniczać długość nazw i identyfikatorów używanych w kodzie VHDL.

Tab 18. Słowa zarezerwowane VHDL.

abs	generate	procedure
access	generic	process
after	guarded	
alias		range
all	if	record
and	in	register
architecture	inout	rem
array	is	report
assert		return
attribute	label	
	library	select
begin	linkage	severity
block	loop	signal
body		subtype
buffer	map	
bus	mod	then
		to
case	nand	transport
component	new	type
configuration	next	
constant	nor	unit
	not	until
disconnect	null	use
downto		
	of	variable
else	on	
elsif	open	wait
end	or	when
entity	other	while
Exit	out	with
File		
For	package	xor
function	port	

W specyficznych przypadkach można korzystać z rozszerzonych nazw identyfikatorów, w których można używać dowolnych nazw. Warunkiem jest rozpoczęcie i zakończenie identyfikatora znakiem ukośnika \:

```
\reg+val\
\when\
```

5.1.3. Literały

Literały są to symbole reprezentujące pewne wartości. Literały **całkowite** zapisujemy w następujący sposób, w zależności od podstawy:

- podstawa dziesiętna: `170, 1_7_0, 10#170#`
- podstawa dwójkowa: `2#1010_1010#`
- podstawa ósemkowa: `8#252#`
- podstawa szesnastkowa: `16#AA#`

Literał **rzeczywisty** to liczba rzeczywista zapisana w postaci liczby o podstawie dziesiętnej, dwójkowej, ósemkowej lub szesnastkowej, na przykład:

- podstawa dziesiętna: `123.1`
- podstawa dwójkowa: `2#101001.01#e1`
- podstawa ósemkowa: `8#1234567.54#`
- podstawa szesnastkowa: `16#abC.d#`

Literały **znakowe** reprezentują pojedynczy znak alfanumeryczny zamknięty w pojedynczym cudzysłowie:

```
'a'
'A'
```

Symbole, czyli wartości zmiennych występujące w typie wyliczeniowym, reprezentowane są jako ciągi znaków:

```
ZIELONY
STAN0
CZEKAJ_1
```

Stałe **łańcuchowe** otoczone są znakami cudzysłowia:

```
"abc"
"ABC"
```

W języku VHDL ciągi bitów możemy zapisać jako literały **łańcuchowo-bitowe**:

```
"1001_1001"
B"1001_1001"
O"167"
X"abC"
```

gdzie litera `B`, `O` lub `X` oznacza podstawę. Brak litery interpretowany jest jak ciąg binarny.

Literały **fizyczne** zapisujemy jako liczba wraz z jednostką oddzielona od liczby spacją:

```
1.5 ns
2 kOhm
60 Hz
```

Dla czytelności zapisu, poszczególne znaki literału mogą być rozdzielone znakiem podkreślenia.

5.1.4. Typ wyliczeniowy

Typ wyliczeniowy jest podstawowym typem danych w VHDL. Definiuje się go poprzez określenie wszystkich wartości, jakie może przyjmować zmienna danego typu:

```
type <Nazwa_typu> is ( <wartość_1>, <wartość_2>, ..., <wartość_n>);
```

Kolejność wartości jest istotna, gdyż wartości zmiennych typu wyliczeniowego można porównywać ze sobą za pomocą operatorów <, >, itp. Wartość z lewej strony jest uznawana za najmniejszą, a symulator inicjalizuje sygnały wartością pierwszą z lewej strony. Poniżej podano przykłady definicji typów wyliczeniowych:

```
type t_KOLOR is (NIEBIESKI, ZIELONY, CZERWONY);
type t_MOJTYP is ('0', '1', 'U', 'Z');
```

a tak wyglądają przykłady przypisań wartości do zmiennych typu wyliczeniowego:

```
variable x_v : t_KOLOR;
signal a: t_MOJTYP;
x_v := NIEBIESKI;
a <= 'Z';
```

Możliwe jest wykorzystanie tych samych wartości w kilku różnych typach jednocześnie, jednak kompilator lub program do syntezy może mieć problemy z rozróżnieniem, o jaki typ chodzi. Konieczne wtedy jest dokładne określenie typu, w jakim jest określona dana wartość za pomocą rzutowania:

```
type KOLOR_PODSTAWOWY is (NIEBIESKI, ZIELONY, CZERWONY);
type KOLOR is (NIEBIESKI, ZIELONY, CZERWONY, FIOLETOWY, RÓŻOWY);
signal A : KOLOR;
A <= KOLOR'(CZERWONY);
```

W rzeczywistym układzie cyfrowym, każdemu symbolowi typu wyliczeniowego musi być przypisana jakaś wartość słowa bitowego, np. kolejnym symbolom przypisuje się kolejne liczby naturalne 0, 1, 2 itd. zakodowane na minimalnej możliwej liczbie bitów. Większość programów do syntezy logicznej pozwala na zmianę sposobu kodowania na np. kod Graya'a, kodowanie typu *one-hot* lub dowolnie zdefiniowane przez użytkownika.

5.1.5. Typ całkowity

Język VHDL posiada wbudowany, nienazwany, typ całkowity. Maksymalny zakres wartości całkowitych rozciąga się od $-(2^{31}-1)$ do $2^{31}-1$ (tj. -2 147 483 647 ... 2 147 483 647). Typ ten służy jako podstawa do definiowania innych typów całkowitych. Można definiować typy całkowite jako podzbiory maksymalnego zakresu.

Definicja typu całkowitego wygląda następująco:

```
type <Nazwa_typu> is range <Zakres_integer>
```

Przykłady:

```
type integer is range -2147483647 to 2147483647;
type PROCENT is range –100 to 100;
```

Zazwyczaj syntezer zamienia wartości tego typu na wektory bitowe o długości niezbędnej do opisania wszystkich wartości danego typu, np. po syntezie wartości typu `PROCENT` reprezentowane będą przez 8-bitowe wektory (1 bit znaku, 7 bitów wartość).

Nie jest możliwy dostęp do poszczególnych bitów wartości typu całkowitego, nie można także z góry określić liczby bitów reprezentujących wartości danego typu całkowitego.

5.1.6. Typy tablicowe

Język VHDL umożliwia deklarowanie zmiennych tablicowych, przy czym wyróżnia się dwa typy tablicowe:

- ograniczony typ tablicowy;
- nieograniczony typ tablicowy.

Ograniczony typ tablicowy

Definiując ograniczony typ tablicowy, należy przy definicji tego typu od razu określić zakres wartości indeksu:

```
type <Nazwa_typu_tablicowego> is array (<Zakres_całkowity>) of
<Typ_elementu_tablicy>
```

gdzie `<Zakres_całkowity>` może być określony jako rosnący: `(<a> to <b>)` lub malejący: `(<b> downto <a>)`, gdzie `<a>` i `<b>` to liczby całkowite oraz `<a>` jest mniejsze lub równe `<b>`.

Przykłady definicji ograniczonych typów tablicowych:

```
type t_A is array (7 downto 0) of std_logic;
type t_B is array (0 to 7)     of std_logic;
```

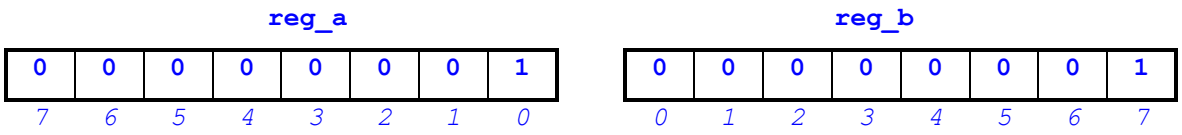
Od momentu zdefiniowania typu tablicowego, można deklarować zmienne tego typu. Poniżej przedstawiono deklarację sygnałów typów `t_A` i `t_B`:

```
signal reg_a : t_A;
signal reg_b : t_B;
```

Implikacją definiowania rejestru 8 bitowego jako `reg_a (7 downto 0)` lub `reg_b(0 to 7)` jest to, że jeśli zawartością tych zmiennych jest ciąg binarny `00000001`, to wówczas poszczególne bity rejestrów są równe:

`reg_a(0)` jest równe `1` a pozostałe bity są równe `0`,

`reg_b(7)` jest równe `1` a pozostałe bity są równe `0`.



Rys. 37 Przykład zmiennych typu tablicowego

Nieograniczony typ tablicowy

Wszystkie zmienne danego typu zdefiniowanego jako ograniczony typ tablicowy muszą mieć taką samą liczbę bitów, co może być wadą, gdyż w projekcie często używamy zmiennych o różnej szerokości bitowej. Dlatego wygodnie jest stosować tzw. nieograniczony typ tablicowy, w którym zamiast liczby elementów podaje się typ całkowity `integer` lub jego pochodne. Rozmiar tablicy jest określany dopiero podczas deklaracji zmiennej danego typu, co pozwala na wykorzystanie w programie różnych zmiennych tablicowych tego samego typu, ale o różnych długościach i różnych zakresach indeksów.

```
type <Nazwa_typu_tablicowego> is array (<Nazwa_typu_całkowitego> range <>) of
<Typ_elementu_tablicy>
```

Przykłady definicji nieograniczonych typów tablicowych:

```
type bit_vector is array (integer range <>) of bit;
type std_logic_vector is array (integer range <>) of std_logic;
```

W momencie deklaracji ustala się liczbę bitów danej zmiennej i zakres indeksów:

```
signal moj_wektor1 : std_logic_vector (5 downto -5);
signal mój_wektor2 : std_logic_vector (1 to 11);
```

Dostęp do elementów tablicy

Dostęp do elementów tablicy odbywa się za pomocą nawiasów klamrowych:

```
<identyfikator_tablicy>(<wyrażenie>)
```

gdzie `<wyrażenie>` może być:

- określeniem liczbowym (np. `3`),
- zmienną (np. `x`)
- zakresem (np. `3 to 6`).

Użycie w zakresie słowa `to` lub `downto` musi być zgodne z deklaracją tablicy. Aby zapis był syntezowalny, program do syntezy może wymagać określenia granic zakresu za pomocą stałych liczb, a nie zmiennych.

Przykłady dostępu do elementów tablicy:

```
tab(6);
tab(x);
tab(3 to 6);
```

Stałe tablicowe

Często istnieje potrzeba przypisania do całej tablicy pewnej stałej wartości. Można to zrobić poprzez przypisanie kolejnym elementom tablicy poszczególnych wartości:

```
type WEKTOR4 is array (1 to 4) of bit;
signal X: WEKTOR4;
X(1) <= '1'; -- Podstawienie poszczególnych elementów
X(2) <= '1';
X(3) <= '0';
X(4) <= '0';
```

Jest także możliwe przypisanie od razu wszystkich elementów tablicy. Można to zrobić na dwa sposoby:

- poprzez wymienienie wszystkich elementów tablicy w kolejności, w jakiej zostały umieszczone w deklaracji typu:

```
X <= ('1', '1', '0', '0');
```

- można także korzystać z zapisu pozwalającego podanie wartości w innej kolejności z określeniem indeksu elementu i użyciem znaku `=>` :

```
X <= (2 => '1', 3 => '0', 1 => '1', 4 => '0');
```

Nie wolno stosować jednocześnie tych dwóch sposobów przypisania. Polom tablicy, którym nie przypisano wartości, może być przypisana taka sama wartość za pomocą składni `others => <Wartość>` umieszczonej jako ostatnie przypisanie, tak jak pokazano w poniższych przykładach:

```
X <= ('1', '1', others => '0');
X <= (3 => '0', 4 => '0', others => '1');
X <= (3 to 4 => '0', 2 downto 1 => '1');
X <= (3 to 4 => '0', others => '1');
X <= (others => '1');
```

W pewnych wypadkach, aby uniknąć dwuznaczności, może być konieczne zaznaczenie, że chcemy wyrazić stałą jako wartość konkretnego typu, dlatego można wyraźnie określić typ za pomocą rzutowania:

```
X <= WEKTOR4'(3 to 4 => '0', others => '1');
```

Atrybuty tablic

Wykorzystując nieograniczony typ tablicowy oraz tzw. atrybuty tablic, można definiować struktury, które działają poprawnie na zmiennych o dowolnej liczbie bitów. Poniżej wymieniono najczęściej stosowane atrybuty, ułatwiające pisanie kodu:

- `'left` – lewy indeks tablicy;
- `'right` – prawy indeks tablicy;
- `'high` – największy indeks tablicy;

- `'low` – najmniejszy indeks tablicy;
- `'lenght` – liczba elementów tablicy;
- `'range` – zakres indeksów tablicy;
- `'reverse_range` – odwrócony zakres indeksów tablicy;

Wykorzystanie atrybutów przedstawiono na przykładach w tab. 19, przy założeniu, że zdefiniowano następującą zmienną `moj_wektor`:

```
signal moj_wektor : std_logic_vector (5 downto -5);
```

Tab. 19. Atrybuty dla zmiennej typu tablicowego.

Wyrażenie zawierające atrybut	Wartość
`moj_wektor'left`	`5`
`moj_wektor'right`	`-5`
`moj_wektor'high`	`5`
`moj_wektor'low`	`-5`
`moj_wektor'length`	`11`
`moj_wektor'range`	`(5 downto -5)`
`moj_wektor'reverse_range`	`(-5 to 5)`

Język VHDL umożliwia tworzenie tablic *n*-wymiarowych (gdzie *n*=1,2,3,...), przy czym większość programów syntezy akceptuje tylko tablice jednowymiarowe.

Przykład definicji typów syntezowalnych tablic dwuwymiarowych, które są zdefiniowane jako jednowymiarowe tablice wektorów:

```
type t_BAJT   is array (7 downto 0) of std_logic;
type t_WEKTOR1 is array (3 downto 0) of t_BAJT;
type t_WEKTOR2 is array (3 downto 0) of std_logic_vector(7 downto 0);
```

Tablice wielowymiarowe są niesyntezowalne, ale mogą być wykorzystane do symulacji, np.:

```
type t_MULTI is array ( 7 downto 0, 255 downto 0) of std_logic;
```

5.1.7. Typ łańcuchowy *string*

Typ łańcuchowy jest zdefiniowany jako nieograniczona tablica znaków typu `character`. Zmienne tego typu deklaruje się następująco:

```
signal NAPIS : string (1 to 8);
...
NAPIS <= "Kowalski";
```

5.1.8. Typ *bit_vector*

Typ `bit_vector` zdefiniowany jest jako tablica elementów typu `bit`. Aby można było łatwo przypisywać wartości stałe zmiennym typu `bit_vector`, określa się je jako ciąg bitów w postaci liczby o podstawie dwójkowej, ósemkowej lub szesnastkowej:

```
signal DATA : bit_vector(7 downto 0);
...
DATA <=  "1011_1100";
DATA <= B"1011_1100";
DATA <= O"5274";
DATA <= X"ABC";
```

5.1.9. Rekordy

Wartości różnych typów mogą być zgrupowane w rekordzie. Poszczególne pola w rekordzie mogą być dowolnego typu (mogą być także rekordem).

```
subtype BYTE_VEC is bit_vector(7 downto 0);
type BYTE_AND_IX is
  record
    BYTE: BYTE_VEC;
    IX: integer range 0 to 8;
  end record;
signal X, Z: BYTE_AND_IX;
signal DATA: BYTE_VEC;
signal NUM: integer;
. . .
X.BYTE <= "11110000";
X.IX <= 2;
DATA <= X.BYTE;
NUM <= X.IX;
Z <= X;
```

W celu przypisania wartości do zmiennej będącej rekordem, można wymienić wszystkie elementy ściśle przestrzegając kolejności pól rekordu (tak jak w definicji rekordu):

```
X <= ("11110000", 2);
```

lub zastosować następującą składnię nazywając poszczególne składowe (kolejność jest wtedy nieistotna):

```
X <= (BYTE => "11110000", IX => 2);
```

Istnieje możliwość wykorzystania przypisania wszystkim pozostałym polom rekordu takiej samej wartości za pomocą polecenia `others=>`, pod warunkiem, że pola te są tego samego typu.

5.1.10. Typ rzeczywisty

Typ rzeczywisty `real` reprezentuje liczbę rzeczywistą:

```
variable X : real;
...
X := 1.234;
```

Typ rzeczywisty jest niesyntezowalny.

5.1.11. Typ fizyczny

Zmienne typu fizycznego to wartość rzeczywista, po której następuje określenie jednostki. Przykładem typu fizycznego jest typ `time` określający czas:

```
wait for 10 ns;
```

5.1.12. Typy predefiniowane

VHDL jest wyposażony w dwa standardowe pakiety: `standard` i `textio` zawierające definicje podstawowych typów. Przykładowa zawartość pakietu standardowego `standard`:

```
package standard is
  type boolean is (FALSE, TRUE);
  type bit is ('0', '1');
  type integer is range -2147483647 to 2147483647;
  subtype natural is integer range 0 to 2147483647;
  subtype positive is integer range 1 to 2147483647;
  type character is ( <tu wymieniono wszystkie 128
                       znaków ASCII> );
  type string is array (positive range <>) of character;
  type bit_vector is array (natural range <>) of bit;
end standard;
```

Pakiet `textio` zawiera typy i polecenia potrzebne do współpracy symulatora z monitorem i klawiaturą – zazwyczaj elementy te nie są potrzebne do syntezy, ale przydają się do zaawansowanej symulacji.

5.1.13. Podtypy

Podtyp jest zdefiniowany jako podzbiór wcześniej zdefiniowanego typu. Podtyp dziedziczy wszystkie właściwości typu podstawowego i może występować tam, gdzie dopuszczalne jest występowanie typu podstawowego. Deklaracja podtypu może odbywać się na dwa sposoby:

```
type bit_vector is array (integer range <>) of bit;
subtype WEKTOR4 is bit_vector(0 to 3);
```

oraz:

```
type MÓJ_TYP is ('0', '1', 'U', 'Z');
subtype MÓJ_TYP2 is MÓJ_TYP range '0' to '1';
```

W rzeczywistości deklaracja podtypu nie definiuje nowego typu – podtyp funkcjonuje jako ograniczony typ podstawowy. O ile różne typy mogą powodować błędy kompilacji spowodowane niedopasowaniem typów (*type mismatch*), to wykorzystanie podtypów pozwala na uniknięcie tych błędów.

5.1.14. Aliasy

Alias nie jest typem danych, ale oznacza nowy identyfikator dla istniejącego obiektu – np. sygnału (ale nie powoduje powstania nowego obiektu!). Najczęstszym wykorzystaniem *aliasu* jest nazwanie określonego podzakresu wektora, np.:

```
signal addr : std_logic_vector(31 downto 0);
alias top: std_logic_vector (3 downto 0) is addr (31 downto 28);
```

5.1.15. Konwersja typów

Konwersja typów możliwa jest tylko pomiędzy podobnymi typami, np. pomiędzy dwoma typami całkowitymi lub dwoma tablicami o takiej samej liczbie elementów i identycznych lub możliwych do przekształcenia typach elementów. Jeśli operator wymaga danego typu, podanie podobnego typu (ale nie poddanego konwersji do typu właściwego) wygeneruje błąd:

```
type t_T1 is range 0 to 255;
signal a1 : t_T1;
signal a2 : integer range 0 to 255;
if a1 = a2 then -- =błąd składniowy (type mismatch,
...             -- operator not defined for such
                -- operands).
```

Rozwiązaniem problemu jest następujące przypisanie:

```
type t_T1 is range 0 to 255;
signal a1 : t_T1;
signal a2 : integer range 0 to 255;

if a1 = t_T1(a2) then -- nie ma błędu!
...
```

5.1.16. Podsumowanie najważniejszych typów danych

Poniżej zamieszczono, jako podsumowanie, przykłady definiowania większości typów:

- typ wyliczeniowy:

```
type KOLOR is (NIEBIESKI, ZIELONY, CZERWONY);
```

- typ całkowity:

```
type PROCENT is range -100 to 100
```

- ograniczony typ tablicowy:

```
type TYPE_A is array (7 downto 0) of bit;
```

- nieograniczony typ tablicowy:

```
type bit_vector is array (integer range <>) of bit;
```

- tablica wielowymiarowa:

```
type MULTI is array ( 7 downto 0, 255 downto 0) of bit;
```

- rekord:

```
type BYTE_AND_IX is
  record
    BYTE: BYTE_VEC;
    IX: integer range 0 to 8;
  end record;
```

- deklaracja podtypu (dwie możliwości):

```
subtype WEKTOR4 is bit_vector(0 to 3);
subtype MOJ_TYP2 is MOJ_TYP range '0' to '1';
```

5.1.17. Biblioteki

Każdy fragment kodu VHDL zawarty w blokach **entity**, **configuration**, **package** jest analizowany (kompilowany) i umieszczany w bibliotece bieżącego projektu o nazwie **work**. W ogólności biblioteki są implementowane jako pliki dyskowe i dostępne przez ich logiczną nazwę. Przeważnie nazwa logiczna biblioteki skojarzona jest z fizyczną ścieżką do odpowiedniego katalogu. Przyporządkowanie to jest realizowane w różnorodny sposób, w zależności od systemu projektowego. Jednakże, podobnie jak w przypadku zmiennych czy sygnałów, przed użyciem musimy bibliotekę zdeklarować poprzez wyrażenie **library**, tak jak to przedstawiono w poniższym przykładzie:

```
library Nazwa_1,Nazwa_1,...;
```

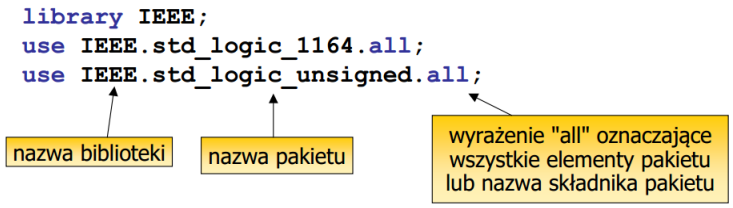
W języku VHDL biblioteki **std** oraz **work** widoczne są nawet bez ich jawnej deklaracji. W ramach pierwszej z nich widoczny jest pakiet **standard** wraz ze wszystkimi elementami składowymi. Biblioteka **work** zawiera natomiast elementy bieżącego projektu. Biblioteki inne niż **std** oraz **work** muszą być deklarowane jawnie.

Wstawianie pakietu z biblioteki

Biblioteki składają się z pakietów, pakiety natomiast składają się z udostępnianych elementów (np. deklaracji typów, funkcji procedur, operatorów itp.). Z tego powodu istnieje potrzeba wskazania, które pakiety czy elementy danego pakietu mają być widoczne w bieżącym projekcie. Wykonuje się to za pomocą wyrażenia **use**:

```
library <Nazwa_biblioteki>;
use <Nazwa_biblioteki>.<Nazwa_pakietu>.all;
```

tak, jak w przykładzie z rys. 38.



Rys. 38. Przykład załączania biblioteki w VHDL.

Widzialność bibliotek i pakietów jest przenoszona wyłącznie do najbliższego bloku `entity`, `package` lub `configuration` występującego bezpośrednio poniżej deklaracji `library`/`use`, tj. polecenie to musi być wydane bezpośrednio przed blokiem (np. `entity`) i dotyczy tylko tego bloku. W przypadku występowania w jednym pliku kilku bloków, każdy z nich należy poprzedzić poleceniem użycia pakietu.

Istnieje możliwość włączenia tylko jednej deklaracji z pakietu (np. wybranej funkcji) – w tym celu zamiast `.all` należy podać nazwę konkretnej deklaracji:

```
library IEEE;
use IEEE.std_logic_1164.my_func
```

Uwaga: gdy przed blokiem `entity` umieszczone jest polecenie `use`, to nie ma potrzeby umieszczać tego polecenia jeszcze raz przed blokiem `architecture`.

Polecenie `library` nie może być umieszczone w części deklaracyjnej architektury – powinno być umieszczone przed słowem kluczowym `architecture` (lub `entity`). W części deklaracyjnej architektury może występować polecenie `use`.

Biblioteki `std` oraz `work` są zazwyczaj dołączane automatycznie, tj. nie trzeba dołączać ich za pomocą polecenia `library` (np. `library work;`), ale przeważnie trzeba deklarować dostęp do pakietów z tych bibliotek, tj. należy użyć polecenia `use` (np. `use work.my_package.all`).

5.1.18. Pakiet *std_logic_1164*

Oprócz omówionej już biblioteki `std` zawierającej pakiet standard, język VHDL wyposażono w bibliotekę `ieee` zawierającą m.in. pakiet: `std_logic_1164`. Poniżej podano fragment implementacji pakietu `std_logic_1164`:

```
package std_logic_1164 is
  type std_ulogic is ('U', 'X', '0', '1', 'Z',
                      'W', 'L', 'H', '-');
  type std_ulogic_vector is array (natural range <>)
                        of std_ulogic;
  function resolved (s: std_ulogic vector)
                    return std_ulogic;
  subtype std_logic is resolved std_ulogic;
  type std_logic vector is array (natural range <>)
```

```
                        of std_logic;
    . . .
    function "and" (I, r std_logic vector)
                   return std_logic_vector;
    function "and" (I, r: std_ulogic vector)
                   return std_ulogic_vector;
  end package std_logic_1164;
```

Używając tego pakietu, można deklarować zmienne typów `std_ulogic`, które mogą przyjmować 9 wartości:

`0` = stan logiczny `'0'`;

`1` = stan logiczny `'1'`;

`U` = stan niezainicjowany (*uninitialized*);

`X` = stan nieznany (*unknown*);

`Z` = stan wysokiej impedancji dla sygnału trzystanowego (*tri-state*);

`W` = stan nieznany dla sygnału niskoobciążalnego (*weak unknown*);

`H` = stan wysokiej rezystancji (dla wyjść typu otwarty kolektor) (*weak* `'1'`);

`L` = stan niskiej rezystancji (dla wyjść typu otwarty emiter) (*weak* `'0'`) ;

`-` = stan nieistotny (*don't care*).

Te 9 stanów wystarcza dla symulacji i opisu najczęściej występujących warunków logicznych.

Typ `std_logic` jest typem pochodnym od `std_ulogic`, ale uzupełnionym o tzw. funkcję arbitrażową. Funkcja arbitrażowa rozstrzyga konflikty powstające podczas sterowania jednego sygnału przez dwie różne wartości (np. jednocześnie `'1'` i `'Z'`). O ile taki konflikt dla typu `std_ulogic` generuje błąd symulacji, to funkcja arbitrażowa dodana do typu `std_logic` może rozstrzygać, jaki sygnał powinien występować w przypadku konfliktów. Umożliwia to symulację np. układów posiadających magistralę danych z dołączonymi układami o wyjściach trójstanowych.

Korzystanie z biblioteki jest możliwe po dodaniu następujących linii:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

Wartości `'0'`, `'1'`, `'L'` i `'H'` są syntezowalne. Dodatkowo wartość `'Z'` jest wykorzystywana do syntezy buforów trójstanowych. Wartość `'-'` jest traktowana jako "*don't care*" i może spowodować uproszczenie układu po syntezie. Pozostałe wartości typu `std_logic` są zwykle niesyntezowalne.

Zalety i wady funkcji arbitrażowej

Kod VHDL wykorzystujący typy wyposażone w funkcję arbitrażową symuluje się dużo wolniej, gdyż przy każdym przypisaniu sygnału uruchamiana jest funkcja arbitrażowa. W przypadku skomplikowanych modeli zaleca się stosowanie typu bez funkcji arbitrażowej. Brak funkcji

arbitrażowej pozwala od razu wychwycić konflikty, nawet krótkotrwałe, spowodowane błędami w projekcie. Jednak w przypadku symulacji układu, w którym następuje celowe przypisanie kilku wartości do jednego sygnału (np. układ z buforami trójstanowymi), stosowanie typu z funkcją arbitrażową jest konieczne.

5.1.19. Pakiet *std_logic_arith*

Zmienne typu `std_logic_vector` nie są interpretowane jako liczby bitowe, lecz jako zwykłe tablice bitów. W celu umożliwienia realizacji operacji matematycznych na zmiennych binarnych, opracowany został przez firmę Synopsys pakiet `std_logic_arith`. W pakiecie tym zdefiniowano dwa podstawowe typy danych: `signed` i `unsigned`, zgodnie z następującymi definicjami:

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

oraz dodano zestaw operatorów umożliwiających operowanie na zmiennych tych typów, jak na liczbach binarnych. Aby korzystać z pakietu `std_logic_arith`, należy wstawić następujące polecenia:

```
library IEEE;
use IEEE.std_logic_arith.all;
```

Dane typu `unsigned` reprezentowane są jako wektory, gdzie bit z lewej strony jest najbardziej znaczący, np.:

```
variable X: unsigned(1 to 8);
```

gdzie `X` to 8–bitowa liczba, a `X(X'left) = X(1)` to najbardziej znaczący bit.

W przypadku odwrotnej kolejności indeksowania:

```
variable Y: unsigned(4 downto 0);
```

najbardziej znaczący bit to `Y(Y'left) = Y(4)`.

Dane typu `signed` są reprezentowane jako dwójkowa reprezentacja z uzupełnieniem do dwóch:

```
signed'("0101") --  +5
signed'("1011") --  -5
```

Dla typów `signed`, `unsigned` i `integer` dostępne są podstawowe operacje matematyczne `+`, `-`. Dodatkowo dla `signed` i `unsigned` dostępny jest również operator mnożenia `*` oraz poniższe operacje przesuwania bitowego, najczęściej realizowane jako rejestry przesuwne:

```
function shl(ARG: unsigned; COUNT: unsigned) return unsigned;
function shl(ARG: signed; COUNT: unsigned) return signed;
function shr(ARG: unsigned; COUNT: unsigned) return unsigned;
function shr(ARG: signed; COUNT: unsigned) return signed;
```

Jeśli argument funkcji przesuwania bitowego zawiera przynajmniej jedną wartość `'X'`, to wynikiem operacji przesuwania będzie tablica złożona z samych wartości `'X'`: `"XX...X"` a symulator wygeneruje ostrzeżenie.

Funkcje `shl` przesuwają bity w lewo, uzupełniając wolne miejsca z prawej strony zerami `'0'`.

Funkcje `shr` przesuwają bity w prawo i uzupełniają wolne miejsca z lewej strony:

- zerami `'0'` dla argumentu typu `unsigned`;
- bitami znaku dla argumentu typu `signed`.

Operacje przesuwania bitowego mogą być wykorzystane np. do prostego mnożenia lub dzielenia liczby typu `unsigned` przez potęgę liczby 2.

W pakiecie `std_logic_arith` zdefiniowano podstawowe operatory porównania, wynik ich działania może być różny dla różnych typów, ze względu na to, że są one różnie interpretowane, co pokazano w Tabeli 20.

Tabela 20 Działanie operatorów porównania dla różnych typów zmiennych.

ARG1	operator	ARG2	unsigned	signed	std_logic_vector
"000"	=	"000"	true	true	true
"00"	=	"000"	true	true	false
"100"	=	"0100"	true	false	false
"000"	<	"000"	false	false	false
"00"	<	"000"	false	false	true
"100"	<	"0100"	false	true	false

Pakiet `std_logic_arith` definiuje następujące funkcje konwersji:

```
subtype SMALL_INT is integer range 0 to 1;
function conv_integer(ARG: integer) return integer;
function conv_integer(ARG: unsigned) return integer;
function conv_integer(ARG: signed) return integer;
function conv_integer(ARG: std_ulogic) return SMALL_INT;
function conv_unsigned(ARG: integer; SIZE: integer) return unsigned;
function conv_unsigned(ARG: unsigned; SIZE: integer) return unsigned;
function conv_unsigned(ARG: signed; SIZE: integer) return unsigned;
function conv_unsigned(ARG: std_ulogic; SIZE: integer) return unsigned;
function conv_signed(ARG: integer; SIZE: integer) return signed;
function conv_signed(ARG: unsigned; SIZE: integer) return signed;
function conv_signed(ARG: signed; SIZE: integer) return signed;
function conv_signed(ARG: std_ulogic; SIZE: integer) return signed;
function conv_std_logic_vector(ARG: integer; SIZE: integer) return
std_logic_vector;
function conv_std_logic_vector(ARG: unsigned; SIZE: integer) return
std_logic_vector;
function conv_std_logic_vector(ARG: signed; SIZE: integer) return
std_logic_vector;
function conv_std_logic_vector(ARG: std_ulogic;
SIZE: integer) return std_logic_vector;
```

Funkcje `conv_unsigned` i `conv_signed` wymagają dwóch argumentów – drugi argument określa liczbę bitów dla wynikowej wartości. Maksymalna liczba bitów dopuszczalna podczas konwersji to 32 bity.

Funkcję konwersji można wykorzystać do zwiększenia lub zmniejszenia liczby bitów w liczbie, np.:

```
conv_signed(signed'("110"),8)
```

dzięki czemu otrzymamy `"11111110"`.

5.1.20. Pakiet *std_logic_unsigned*

Pakiet ten, opracowany również przez firmę Synopsys, umożliwia wykonywanie operacji arytmetycznych, konwersji i porównywania danych typu `std_logic` i `std_logic_vector` traktowanych jako liczby całkowite bez znaku (*unsigned*).

Standardowo, wszystkie zmienne typu `std_logic_vector` są traktowane jako tablice bitowe i nie mają znaczenia liczbowego. W momencie, gdy włączymy pakiet `std_logic_unsigned` do architektury lub bloku `entity`, wszystkie zmienne `std_logic_vector` będą traktowane jako liczby binarne bez znaku i będzie można przeprowadzać na nich działania matematyczne bądź porównania z odpowiednią interpretacją liczbową a nie leksykograficzną.

5.1.21. Pakiet *std_logic_signed*

Pakiet ten działa podobnie jak pakiet `std_logic_unsigned`, z tym, że umożliwia wykonywanie operacji arytmetycznych, konwersji i porównywania danych typu `std_logic/std_logic_vector` traktowanych jako liczby całkowite ze znakiem (*signed*) zakodowanych jako uzupełnienie do 2.

5.1.22. Tworzenie własnego pakietu

Pakiet składa się z części deklaracyjnej zgrupowanej pomiędzy słowami kluczowymi `package` i `end package`:

```
package mój_pakiet is
...
-- deklaracje typów, stałych,
-- deklaracje funkcji i procedur
...
end package mój_pakiet;
```

Jeśli w części deklaracyjnej występują deklaracje funkcji lub procedur, to oprócz części deklaracyjnej musi występować także część definiująca `package body`:

```
package body mój_pakiet is
...
-- deklaracje i definicje (implementacje)
-- funkcji i procedur
```

```
end package mój_pakiet;
```

5.2. Poziom strukturalny

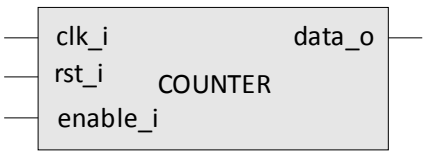
5.2.1. Blok *entity*

Podstawową jednostką projektową w VHDL jest blok `entity`. Nazwa jednostki pozwala w prosty sposób zidentyfikować tę jednostkę w całym systemie w trakcie jego syntezy. Jednostka projektowa komunikuje się z otoczeniem poprzez porty (wejściowe, wyjściowe oraz dwukierunkowe), będące specjalnym rodzajem sygnałów. Opis bloku `entity` obejmuje:

- nazwę,
- deklarację portów wejściowych i wyjściowych.

Przykład definicji bloku `entity`:

```
entity COUNTER is
  port(clk_i    : in bit;
       rst_i    : in bit;
       enable_i : in bit;
       data_o   : out bit_vector(31 downto 0)
      );
end entity;
-- lub: end COUNTER;
-- lub: end entity COUNTER;
```



Rys. 39 Blok `entity` w języku VHDL

Wyrażenie `port` definiuje wejścia i wyjścia bloku `entity` i musi być zgodne w składni z poniższym przykładem:

```
port (<Nazwa_sygnału> : <Kierunek> <Typ_logiczny> ;
      <Nazwa_sygnału> : <Kierunek> <Typ_logiczny> ;
      <Nazwa_sygnału> : <Kierunek> <Typ_logiczny> ;
       itd...
      <Nazwa_sygnału> : <Kierunek> <Typ_logiczny> );
```

gdzie `<Kierunek>` oznacza kierunek przepływu sygnału:

- `in` – port wejściowy, można tylko z niego czytać;
- `out` – port wyjściowy, można tylko do niego pisać, ale nie można czytać;
- `inout` – port dwukierunkowy, można z niego czytać oraz do niego pisać;

- `buffer` – port wyjściowy z możliwością czytania.

Należy zauważyć, że każda linia definiująca port kończy się średnikiem, z wyjątkiem ostatniej, gdzie na końcu jest nawias i średnik.

Kolejność sygnałów w wyrażeniu `port` nie ma znaczenia. Dobrym zwyczajem jest np. grupowanie alfabetyczne sygnałów danych typów oraz dodawanie zakończenia nazwy portu "`_i`" dla portów wejściowych `in`, "`_o`" dla portów wyjściowych `out` oraz "`_io`" dla portów `inout`.

5.2.2. Blok architektury

Blok architektury `architecture` definiuje, jak powinny zachowywać się, lub z czego powinny się składać, bloki `entity`. Możliwe jest występowanie wielu bloków `architecture` skojarzonych z jednym blokiem `entity`, lecz podczas symulacji lub syntezy dla danego bloku `entity` wybierany jest tylko jeden blok `architecture`.

```
architecture <Nazwa_architektury> of <Nazwa_entity> is
    <Deklaracje>
        ......
begin
    <Polecenia_współbieżne>
            ......
end architecture;
-- lub end <Nazwa_architektury>
-- lub end architecture <Nazwa_architektury>
```

Architektura ma swoją nazwę `<Nazwa_architektury>`. Zaleca się, aby nazwa architektury opisywała działanie komórki.

`<Nazwa_entity>` to nazwa bloku `entity`, którego działanie opisuje dana architektura.

`<Deklaracje>` to miejsce gdzie występować mogą deklaracje typów, podtypów stałych, sygnałów, zmiennych oraz komponentów.

Układ cyfrowy składa się zazwyczaj z wielu bloków `entity` (każdy z nich reprezentuje funkcjonalny blok logiczny) oraz z sieci sygnałowych łączących te bloki ze sobą poprzez ich porty. Każdy opis `architecture` ma z definicji dostęp do portów sygnałowych. Dodatkowo dla każdego bloku `architecture` można zdefiniować sygnały dostępne tylko wewnątrz tego bloku.

5.2.3. Stałe

Stałe można definiować w architekturach, pakietach, procesach i podprogramach. Stałe zadeklarowane w architekturze są widziane tylko wewnątrz architektury. Przykłady definicji stałych:

```
constant width: integer := 8;
constant memory: MEMORY_t := ("0000", "0011", "1100");
```

5.2.4. Sygnały

Sygnały definiowane przez wyrażenie **port** tworzą połączenia pomiędzy blokami **entity**, analogicznie jak końcówki (tzw. konektory, ang. *connectors*) łączące poszczególne płyty lub inne moduły systemu elektronicznego. Wewnątrz bloku **entity** używane są także dodatkowe sygnały zapewniające realizację odpowiedniej funkcji tego bloku, podobnie jak to się dzieje z płytami systemów cyfrowych. Te wewnętrzne sygnały nie są dostępne na portach wejścia/wyjścia.

Najprostszy sygnał posiada typ **bit** i jest zdefiniowany tylko dla dwóch stanów logicznych **0** oraz **1**. Następny typ logiczny został nazwany MVL4 (ang. *Multi-Valued Logic*). Zezwala on na posługiwanie się logiką trójstanową oraz ma stan 'nieznany' (ang. *unknown*) używany podczas symulacji (stany **0**, **1**, **Z**, **X**). Typ MVL4 wystarczał do opisu podstawowych operacji, ale nie nadawał się do modelowania bardziej złożonych przypadków, jak np. logiki typu otwarty kolektor i stanu typu „nieistotny" (ang. *don't care*).

Dla zapewnienia większej uniwersalności, we wczesnych latach 90-tych organizacja IEEE zaadoptowała powstały w przemyśle system stanów logicznych włączając w to obciążalność (ang. *driving strength*) oraz inne właściwości przydatne w syntezie i symulacji. Zdefiniowano wówczas standardową bibliotekę logiczną IEEE **std_logic_1164** i zawierającą definicję logiki 9-cio stanowej o nazwie **std_ulogic** oraz system o zbliżonej nazwie **std_logic**. System ten jest znany także jako MVL9. Więcej szczegółów dotyczących **std_logic_1164** podano w rozdziale 5.1.18.

Sygnały służą do wymiany informacji pomiędzy poszczególnymi procesami w danej architekturze. Sygnały zadeklarowane słowem kluczowym **signal** są dostępne tylko wewnątrz architektury, w której zostały zadeklarowane. Porty są specjalną wersją sygnałów, które mogą wychodzić poza architekturę.

Sygnały można deklarować tylko w części deklaracyjnej:

- pakietu (**package**);
- architektury (**architecture**);
- bloku współbieżnego (**block**).

Sygnałów nie można deklarować w procesach i podprogramach. Sygnały są deklarowane poprzez podanie ich nazwy oraz typu logicznego, jak to przedstawiono poniżej. Podczas deklaracji, sygnałom można nadawać wartości początkowe, ale przypisanie wartości początkowej może być ignorowane przez proces syntezy.

```
signal a1, a2: std_logic;
signal cnt: integer := 1;
```

Uwaga: Sygnałami są także porty bloku **entity** i są one widoczne we wszystkich jego architekturach. Nie wolno nazywać sygnałów lub stałych wewnątrz architektury takimi samymi nazwami jak porty w bloku **entity**.

5.2.5. Osadzanie komponentu

Aby osadzić w danym bloku `entity` inny blok, należy go najpierw zadeklarować jako komponent w części deklaracyjnej architektury. Tak zadeklarowany komponent można osadzać tylko wewnątrz tej architektury. W przypadku konieczności osadzania komponentu w wielu architekturach, należy zadeklarować dany komponent w pakiecie `package` i dołączyć `package` do architektury. Informacje jak zdefiniować własny pakiet znajdują się w rozdziale 5.1.22.

Deklaracja komponentu wygląda następująco:

```
component <Nazwa_komponentu>
  port (<Deklaracje_portów>);
end component;
```

Poniżej podano przykład deklaracji komponentu – bramki `nand2`:

```
component nand2 is
  port (a : in bit;
        b : in bit;
        y : out bit);
end component;
```

przy założeniu, że komponent `nand2` został wcześniej zdefiniowany jako zestaw `entity` i `architecture`:

```
entity nand2
  port (A, B : in bit,
        Y    : out bit);
end entity;

architecture beh of NAND2 is
...
begin
...
end architecture;
```

W celu wstawienia zadeklarowanego komponentu, należy podać informację o podłączeniach jego końcówek za pomocą polecenia `port map`. Można to zrobić na dwa sposoby:

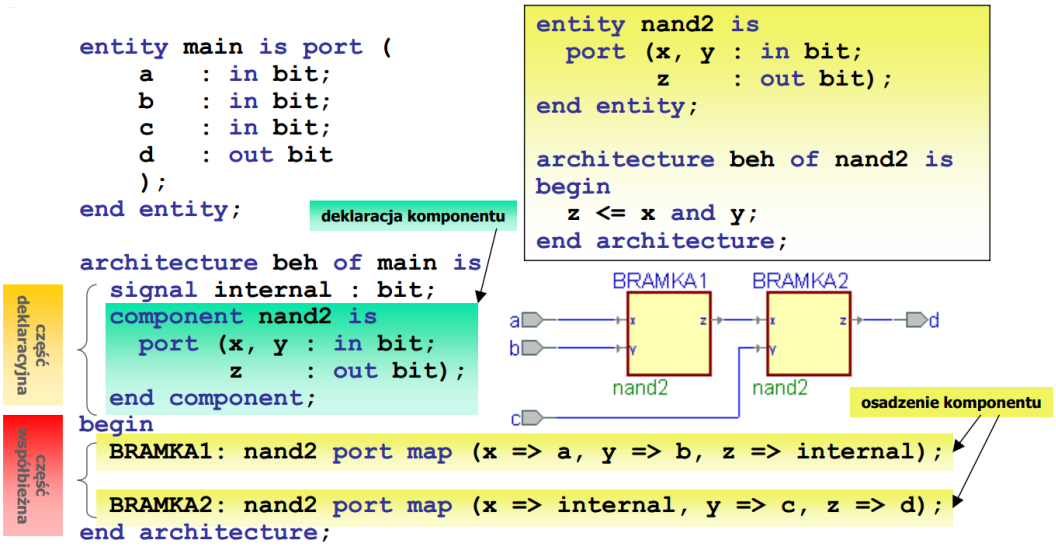
- sposób uproszczony przy użyciu kolejności końcówek jak w deklaracji komponentu:

```
B1 : nand2 port map (in1, in2, out1);
```

- poprzez nazwy portów wówczas kolejność jest nieistotna:

```
B1 : nand2 port map (y => out1, a => in1, b => in2);
```

Na rys. 40 pokazano kompletny przykład osadzenia komponentu `nand2` w bloku `entity` o nazwie `main`.



Rys. 40. Przykład osadzenia komponentu.

5.2.6. Polecenie *generate*

Polecenie `generate` służy do osadzenia wielu takich samych poleceń współbieżnych, najczęściej komponentów lub np. poleceń przypisania współbieżnego `<=` (przypisanie współbieżne opisano w rozdziale 5.3.1).

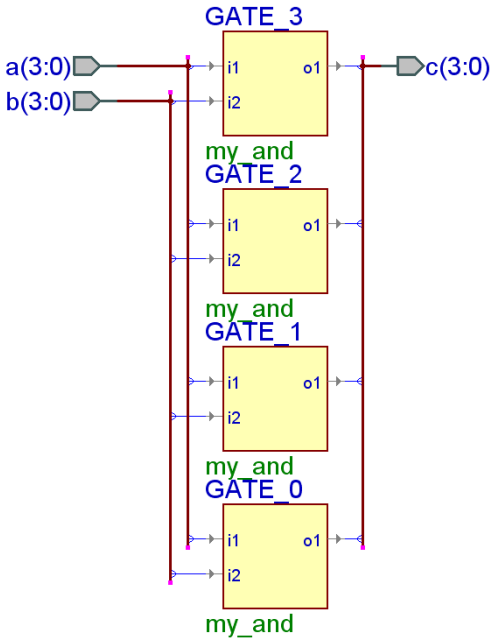
Występują dwie formy polecenia `generate`:

- Polecenie `for-generate`
- Polecenie `if-generate`

Za pomocą polecenia `for-generate` osadzona zostaje seria komponentów (w poniższym przykładzie są to komponenty `my_and`, gdzie poszczególne komponenty będą miały nazwę składająca się z etykiety `GATE` oraz wartości indeksu `i`). Wymaga się, aby przed poleceniem `for-generate` była umieszczona etykieta (w poniższym przykładzie etykieta to `AND_BLOCK`).

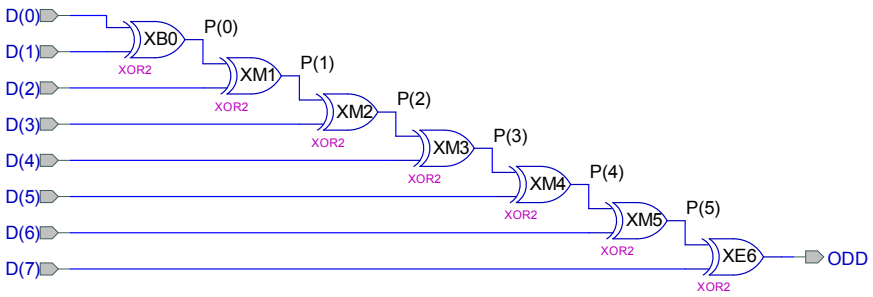
```
entity test is
  port (a, b : in  bit_vector(3 downto 0);
    c : out bit_vector(3 downto 0) );
end entity;
architecture rtl of test is
  component my_and
    port (
      i1 : in  bit;
      i2 : in  bit;
      o1 : out bit);
  end component;
begin
  AND_BLOCK: for i in 0 to 3 generate
    GATE : my_and port map (a(i), b(i), c(i) );
  end generate;
```

```
end architecture;
```



Rys. 41. Wynik działania polecenia `for-generate` z przykładu.

Polecenie `if-generate` stosuje się zazwyczaj w połączeniu z `for-generate` do osadzenia komponentów będących szczególnymi przypadkami, np. gdy pierwszy i ostatni element muszą być inne od pozostałych. Poniżej pokazano przykład układu generującego bit parzystości:



Rys. 42. Układ generujący bit parzystości

```
entity parity8 is
  port (d : in  bit_vector(0 to 7);
    odd : out bit );
end entity;

architecture rtl of parity8 is
  component xor2
    port (a, b : in  bit;
      y : out bit);
  end component;
  signal p: bit_vector(d'low to d'high - 2);
```

```
begin
  G: for i in d'low to d'high-1  generate
    G_BEG: if i=d'low generate
      XB: xor2 port map (a => d(d'low),
        b => d(d'low+1),
        y => p(d'low) );
    end generate;
    G_MIG: if i > d'low and i < d'high-1 generate
      XM: xor2 port map (a => p(i-1),
        b => d(i+1),
        y => p(i) );
    end generate;
    G_END: if i = d'high-1 generate
      XE: xor2 port map (a=>p(i-1),
        b=>d(i+1),
        y => odd );
    end generate;
  end generate;
end architecture;
```

5.2.7. Parametry bloku *entity* (*generic*)

Składnia `generic` umożliwia sparametryzowanie bloku `entity`, przez co poszczególne egzemplarze danego bloku mogą działać w zróżnicowany sposób. Najprostszym zastosowaniem `generic` jest opisanie bloku, który może np. działać na szynach danych o różnej długości lub może mieć różne czasy opóźnień. Poniżej zamieszczono przykład modułu `OR_N` działającego jako bramka `or`, która może operować na danych o różnej szerokości:

```
entity OR_N is
  generic (W : positive);
  port( A : in  bit_vector(W-1 downto 0);
        B : in  bit_vector(W-1 downto 0);
        Y : out bit_vector(W-1 downto 0) );
end entity;

architecture RTL of OR_N is

begin
  G: for I in W-1 downto 0 generate
    Y(I) <= A(I) or B(I);
  end generate;
end architecture;
```

Osadzenie powyższego modułu z parametrem *W*=32:

```
entity OR32 is
  port (A32 : in  bit_vector(31 downto 0);
        B32 : in  bit_vector(31 downto 0);
        Y32 : out bit_vector(31 downto 0) );
end entity;

architecture RTL of OR32 is
  component OR_N
    generic (W : positive);
    port( A : in  bit_vector(W-1 downto 0);
          B : in  bit_vector(W-1 downto 0);
          Y : out bit_vector(W-1 downto 0) );
  end component;
begin
  A1: OR_N
        generic map (W => 32)
        port map (A => A32, B => B32, Y => Y32);
end architecture;
```

5.3. Poziom przesłań międzyrejestrowych RTL

Opis układu lub systemu na poziomie przesłań międzyrejestrowych RTL składa się z dwóch najważniejszych elementów: z przypisań sygnałów oraz z operatorów logicznych. Najczęściej stosowany operator przypisania, to przypisanie współbieżne. Oprócz tego istnieją jeszcze dwa rodzaje przypisań współbieżnych warunkowych. Z operatorów dostępne są standardowe operatory logiczne, arytmetyczne i warunkowe. Wszystkie te elementy języka VHDL zostały opisane w niniejszym rozdziale.

5.3.1. Przypisanie współbieżne

Operacja przypisania `<=` jest podstawową operacją dla poziomu opisu RTL. Operacje przypisania umieszczone w bloku `architecture` pomiędzy słowami `begin` i `end` wykonują się współbieżnie. Ich kolejność nie ma znaczenia dla działania układu (wyjątkiem są instrukcje umieszczone wewnątrz procesu, funkcji lub procedury – patrz rozdziały 5.3.11 oraz 5.5).

Składnia operacji przypisania wygląda następująco:

```
<Odbiorca> <= <Wyrażenie>;
```

gdzie `<Odbiorca>` to sygnał, który jest sterowany w sposób ciągły wartością wyrażenia `<Wyrażenie>`.

Przykład:

```
Y <= A and B;
```

Odbiorcą przypisania może być:

- sygnał, np.: `Y`

- jeden element tablicy, np.: `Y(3)`
- zakres elementów tablicy, np.: `Y(3 to 5)`
- pole w rekordzie, np.: `mój_rec.pole1`
- zbiór sygnałów zgrupowany w nawiasach, tzw. agregat.

5.3.2. Współbieżne przypisanie warunkowe *when ... else*

Współbieżne przypisanie warunkowe `when ... else` umożliwia realizację przypisania ciągłego z wykorzystaniem warunków. Składnia przypisania jest następująca:

```
<Odbiorca> <= <Wyrażenie1> when <Warunek1> else
              <Wyrażenie2> when <Warunek2> else
              ...
              <WyrażenieN>;
```

Sygnałowi `<Odbiorca>` przypisywane jest pierwsze wyrażenie, dla którego warunek jest spełniony, czyli osiąga wartość `TRUE`. Jeśli żaden z warunków nie jest spełniony, to sygnałowi przypisane zostanie ostatnie wyrażenie `<WyrażenieN>`. Przykład:

```
Y <= A when SEL_A = '1' else
     B when SEL_B = '1' else
     C;
```

Dobrym przykładem zastosowania przypisania warunkowego jest synteza trójstanowego bufora magistrali:

```
Y <= A when ENABLE = '1' else 'Z';
```

5.3.3. Przypisanie współbieżne *select ... when*

Składnia przypisania `select...when`:

```
with <Wyrażenie_wyboru> select
     <Odbiorca> <= <Wyrażenie1> when <Wybór1>,
                   <Wyrażenie2> when <Wybór2>,
                   ...
                   <WyrażenieN> when <WybórN>;
```

Sygnałowi `<Odbiorca>` przypisywane jest wyrażenie, dla którego wartość `<Wyrażenie_wyboru>` odpowiada danemu wyrażeniu `<WybórX>`. Wyrażenia `<WybórX>` mogą być zarówno konkretnymi wartościami, np. `3`, jak i zakresami wartości: `2 to 4` lub `0x"0000" to 0x"7FFF"`.

Przykład:

```
signal A, B, C, D, Y : bit;
signal SEL: bit_vector(1 downto 0);
...
with SEL select
  Y <= A when "00",
```

```
        B when "01",
        C when "10",
        D when others;
```

Jako ostatnie wyrażenie **<WybórN>** może być wykorzystane słowo kluczowe **others**, wtedy w przypadku, gdy żadne z wyrażeń **<WybórX>** nie pasuje do **<Wyrażenia_wyboru>**, **<Odbiorcy>** przypisywane jest ostatnie wyrażenie. Należy przestrzegać następujących zasad:

- żadne z wyrażeń **<WybórX>** nie mogą zachodzić na siebie,
- jeśli jako ostatnie nie występuje wyrażenie **others**, to wyrażenia **<WybórX>** muszą pokrywać wszystkie możliwe wartości wyrażenia **<Wyrażenie_wyboru>**.

5.3.4. Różnice pomiędzy przypisaniem *when...else* i *select...when*

Przypisanie warunkowe **when...else** charakteryzuje się tym, że sygnały sprawdzane są według kolejności umieszczenia ich w poleceniu – preferowane są sygnały występujące na początku. Umożliwia to łatwą realizację kodowania z priorytetem (*priority encoding*).

Przypisanie warunkowe **select** różni się od przypisania warunkowego **when...else** tym, że nie preferuje żadnego sygnału.

Przypisanie **when...else** umożliwia określenie różnych, nawet złożonych warunków dla każdej z opcji. Przypisanie **select...when** może sprawdzać tylko różne wartości dla jednej zmiennej.

5.3.5. Operatory logiczne

Dla większości typów reprezentujących sygnały cyfrowe (np. **bit**, **std_logic**, **bit_vector**, **std_logic_vector**) dostępne są następujące operatory logiczne:

```
and
or
nand
nor
xor
xnor
not
```

Argumenty operatorów muszą być tego samego typu, mogą być również tablicami jednowymiarowymi, ale o takiej samej liczbie elementów. W przypadku operacji na dwóch tablicach, odbywa się ona pomiędzy wszystkimi odpowiednimi elementami obydwu tablic jednocześnie. W przypadku użycia większej niż dwie operacje logiczne w jednym wyrażeniu, należy je zgrupować w nawiasach () po dwie, z wyjątkiem przypadku, gdy wszystkie operacje są tego samego typu:

```
A and B and C and D
```

ale:

```
(A and B) or C
```

```
A and (B or C)
```

5.3.6. Operatory porównania

Dla wszystkich typów standardowych zdefiniowane są operatory równości (=) i różności (/=). Dwa argumenty operatora są sobie równe, gdy posiadają takie same wartości. W przypadku tablic i rekordów porównywane są między sobą wszystkie odpowiednie pola.

Operatory <, <=, > i >= zdefiniowane są dla typów wyliczeniowych, typów całkowitych oraz jednowymiarowych tablic tych typów. Typy wyliczeniowe porównywane są ze sobą według kolejności deklaracji poszczególnych elementów. Tablice porównywane są ze sobą w sposób leksykograficzny, zaczynając od bitu z lewej strony (bez względu na sposób deklaracji `to` czy `downto`). Przykładowo wartość `"101011"` jest mniejsza od `"1011"`, ponieważ różni się czwartym bitem od lewej strony. Gdy tablice są różnej długości i mają identyczny początek, to mniejszą rangę ma krótsza tablica, np.: `"101"` jest mniejsze niż `"101000"`.

5.3.7. Operatory dodawania i konkatenacji

Operatory arytmetyczne + i – są zdefiniowane dla wszystkich typów `integer`.

```
signal A, B, C: integer range 0 to 3;
C <= A + B;
```

Operator konkatenacji & działa dla jednowymiarowych tablic dowolnego typu. Argumentem konkatenacji może być tablica lub element tablicy. W ten sposób można łączyć tablice ze sobą, a także dodawać jeden element na początku lub końcu tablicy:

```
signal A, D: bit_vector(3 downto 0);
signal B, C, G: bit_vector(1 downto 0);
signal E: bit_vector(2 downto 0);
signal F, H, I: bit;
A <= not B & not C; -- tablica & tablica
D <= not E & not F; -- tablica & element
G <= not H & not I; -- element & element
```

Operator jednoargumentowy – (`minus`) pozwala zmienić znak liczby typu `integer`:

```
signal A, B: integer range -8 to 7;
A <= -B;
```

5.3.8. Inne operatory

Dla typów `integer` zdefiniowane są operatory:

- mnożenia *;
- dzielenia /;
- dzielenia modulo `mod`;

- reszty z dzielenia `rem`;

Często do syntezy operatorów `/`, `mod` i `rem` wymagane jest, aby prawy argument był stałą liczbą dodatnią będącą potęgą liczby 2 (ich realizacja polega na przesunięciu bitowym).

Dla typu `integer` dostępne są operatory `abs` i potęgowania `**`. W przypadku syntezy operatora `**` wymaga się przeważnie, aby lewy argument był równy 2.

5.3.9. Opóźnienia

Język VHDL umożliwia modelowanie opóźnień powstających w bramkach i na połączeniach. Rozróżnia się opóźnienia:

- inercyjne;
- typu `transport`.

Opóźnienie inercyjne jest standardowym typem opóźnienia i służy do modelowania opóźnień powodowanych przez bramki logiczne, np.:

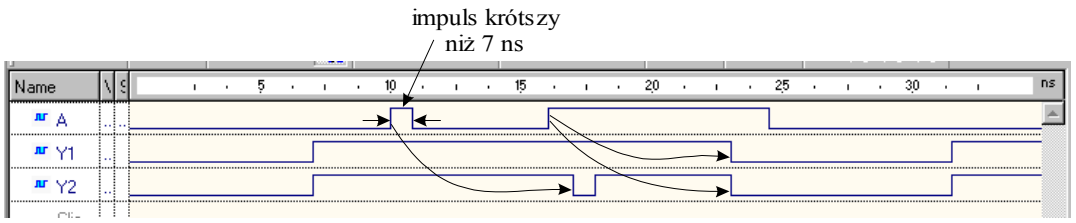
```
Y1 <= inertial not A after 7 ns;
```

Słowo kluczowe `inertial` można pominąć. Sygnały są opóźnione o określony czas, lecz sygnały krótsze niż ten czas nie są wcale przesyłane na wyjście.

Opóźnienia typu `transport` służą przeważnie do opisu opóźnień powstających na przewodach połączeniowych, np.:

```
Y2 <= transport not A after 7 ns;
```

Sygnały są opóźnione o określony czas, sygnały krótsze niż ten czas są również przesyłane na wyjście.



Rys. 43. Przykład modelowania opóźnień.

Za pomocą polecenia `reject` istnieje możliwość określenia, jakie impulsy będą odrzucane:

```
Y1<= reject 3 ns inertial not A after 7 ns;
```

W powyższym przykładzie impulsy krótsze niż 3ns nie będą przesyłane, pozostałe sygnały będą opóźnione o 7ns. Opóźnienia z opcją `reject` dotyczą tylko opóźnień inercyjnych.

5.3.10. Instrukcje współbieżne i sekwencyjne

Instrukcje opisujące działanie projektowanego układu zawarte są wewnątrz bloku `architecture` pomiędzy wyrażeniami `begin` oraz `end`. Wszystkie instrukcje tam umieszczone wykonywane są współbieżnie (tj. równocześnie), a ich pozycja w kodzie VHDL nie ma żadnego znaczenia (poza

czytelnością i przejrzystością zapisu). Wśród tych instrukcji znajdują się trzy następujące instrukcje: proces, funkcja i procedura, które grupują wewnątrz swego ciała szereg instrukcji wykonywanych sekwencyjnie (patrz rozdziały 5.3.11 oraz 5.5). Same instrukcje procesu, procedury i funkcji (traktowane jako całość) wywoływane są równocześnie z innymi instrukcjami umieszczonymi w architekturze, ich wnętrza natomiast przetwarzane są sekwencyjnie.

Poniżej podano podział instrukcji na współbieżne i sekwencyjne:

Instrukcje współbieżne:

- instrukcja przypisania wartości sygnału (**<=**),
- instrukcja kontroli (**assert**),
- proces (**process**),
- procedura,
- funkcja,
- instrukcja blokowa (**block**),
- instrukcja wstawiania składnika (**component**),
- instrukcja powielania (**generate**).

Instrukcje sekwencyjne.

- instrukcja oczekiwania,
- instrukcja przypisania wartości sygnału (**<=**),
- instrukcja przypisania wartości zmiennej (**:=**),
- instrukcja kontroli (**assert**),
- instrukcja raportu,
- wywołanie procedury,
- instrukcje warunkowe (**if**, **case**),
- instrukcje pętli,
- instrukcja wyjścia,
- instrukcja powrotu,
- instrukcja pusta.

Czasami o przynależności do jednej z tych grup decyduje usytuowanie danej instrukcji. Przykładem może być przypisanie wartości sygnału symbolizowane znakiem ”**<=**”, które umieszczone w procesie lub podprogramach jest instrukcją sekwencyjną natomiast poza nimi - współbieżną.

5.3.11. Procesy

Proces jest najważniejszym elementem składni VHDL, gdzie wykonywanie instrukcji jest sekwencyjne. Ogólna postać definicji procesu wygląda następująco:

```
<nazwa_procesu> : process (<lista_czułości>)
  <część_deklaracyjna>
begin
  <część_sekwencyjna>
end process;
```

Proces jako całość jest traktowany jako jedno polecenie współbieżne, gdyż jest on umieszczany w części współbieżnej architektury. Jeśli w architekturze zdefiniowano kilka procesów, wszystkie one wykonują się współbieżnie względem siebie, a także niezależnie od siebie.

Polecenia wewnątrz procesu wykonują się sekwencyjnie. Jeżeli wśród tych poleceń nie ma poleceń zawierających opóźnienia, to proces wykona się w zerowym czasie.

Proces uruchamia się tylko wtedy, gdy nastąpi zmiana któregokolwiek sygnału umieszczonego na liście czułości (ang. *sensitivity list*) procesu – wykonane zostaną wszystkie instrukcje z części sekwencyjnej procesu, po czym proces zatrzyma się i będzie czekał na kolejną zmianę sygnału z listy czułości.

Proces może nie posiadać listy czułości, wtedy jego wykonywanie jest cykliczne – jak tylko skończy się wykonywać ostatnia instrukcja sekwencyjna, cały proces uruchamiany jest od początku. Proces bez listy czułości musi posiadać w części sekwencyjnej co najmniej jedno polecenie wykorzystujące opóźnienie czasowe – w przeciwnym przypadku proces wykonywałby się nieskończoną liczbę razy w jednej chwili czasu.

Oznaczenie procesu etykietą `<nazwa procesu>` jest opcjonalne.

5.3.12. Zmienne

Zmienne deklaruje się najczęściej w procesach lub podprogramach, ponieważ nie można tam deklarować sygnałów.

```
variable  a_v, b_v : std_logic;
variable count_v : integer := 1;
```

Zmienne są widoczne tylko wewnątrz procesu lub podprogramu, w którym zostały zadeklarowane.

W przypisaniu wartości do zmiennej nie można stosować opóźnień.

Zmiennej nie musi odpowiadać konkretny przerzutnik bądź przewód. Jest to element wykorzystywany w opisie na wysokim poziomie abstrakcji. Synteza zmiennych nie jest dokładnie określona, więc o ile to możliwe, należy korzystać z sygnałów. Dobrym zwyczajem jest nadawanie zmiennym nazw kończących się na "`_v`", co pozwala na łatwe odróżnienie ich od sygnałów w kodzie VHDL.

5.3.13. Przypisanie sekwencyjne

Wewnątrz procesu można przypisać wartość sygnałowi lub zmiennej. Składnia przypisania dla sygnału:

```
<sygnał> <= <wyrażenie>;
```

Dla zmiennej (zadeklarowanej jako `variable`) składnia przypisania wygląda następująco:

```
<zmienna> := <wyrażenie>;
```

Odbiorcami przypisania (tj. `<sygnał>` lub `<zmienna>`) mogą być, tak jak w przypisaniu współbieżnym:

- sygnał lub zmienna, np.: `Y`,
- jeden element tablicy, np.: `Y(3)`,
- zakres elementów tablicy, np.: `Y(3 to 5)`,
- pole w rekordzie, np.: `mój_rec.pole1`,
- zbiór sygnałów lub zmiennych zgrupowany w nawiasach, tzw. agregat.

```
signal S : bit_vector(1 to 4);
signal A, B, C, D : bit;
...
S <= ('1', '0', '1', '0');
(A, B, C, D) <= S;
-- A=1 B=0 C=1 D=0
-- lub
(A, B, C, D) <= bit_vector'('1', '0', '1', '0');
-- A=1 B=0 C=1 D=0
-- lub
(3=>A, 2=>B, 4=>C, 1=>D) <= S;
-- tj.: (D,B,A,C) <= (1,0,1,0) czyli A=1 B=0 C=0 D=1
```

5.3.14. Różnice pomiędzy sygnałem i zmienną

Pomiędzy sygnałem i zmienną występują zasadnicze różnice:

- zmienne `variable` występują tylko wewnątrz procesów i podprogramów, przypisania odbywają się tylko w częściach sekwencyjnych,
- przypisania do sygnału `signal` mogą występować zarówno w częściach współbieżnych, jak i sekwencyjnych – wyróżnia się więc dwa rodzaje przypisania do obiektów typu `signal`: współbieżne (także warunkowe `when` i `select`) oraz sekwencyjne,
- przypisanie wartości do zmiennej `variable` następuje natychmiastowo – od momentu przypisania zmienna posiada nową wartość,
- przypisanie wewnątrz procesu wartości do sygnału `signal` zawsze ma opóźnienie, jeśli nie jest ono podane słowem `after <czas>` wynosi wartość *delta T* dążącą do zera, dodatkowo takie przypisanie niekoniecznie może nastąpić od razu, gdyż sygnał może mieć kilka sterowników i wówczas wygrywa ostatnie przypisanie.

Poniższy przykład ma na celu wyjaśnienie mechanizmu uaktualniania stanów logicznych. Rozważmy trzy przerzutniki w połączeniu prostego rejestru przesuwnego. Rejestr może być opisany wewnątrz procesu jako:

```
REJESTR_PRZESUWNY : process(clk_i, rst_i)
```

```
begin
  if rst_i='1' then
    a <= '0';
    b <= '0';
    c <= '0';
  elsif rising_edge(clk_i) then
    a <= shift_i;
    b <= a;
    c <= b;
  end if;
end process;
```

W momencie pojawienia się zbocza zegarowego `clk_i`, stan układu jest określany na podstawie stanów przerzutników `a`, `b` i `c` oraz stanu wejścia `shift_i`. Załóżmy, że system miał ustalone wartości logiczne równe `'0'` wszystkich czterech sygnałów. Jeśli na wejściu `shift_i` pojawi się `'1'`, wówczas przy następnym zboczu narastającym zegara, który pojawi się w momencie *T*, symulator VHDL wyliczy nowe stany logiczne wyjść `a`, `b` oraz `c` na podstawie stanów w momencie *T*, tj. dla `shift_i='1'`, `a='0'` oraz `b='0'`. Nowe stany logiczne będą równe `a='1'`, `b = c = '0'`, w momencie czasu *T* + *delta T*. W momencie, gdy symulator osiągnie koniec procesu, zwiększa czas *T* + *delta T* i oczekuje na następne zmiany stanu sygnału zegara lub RESET. Obecny stan systemu jest więc następujący: `a='1'`, `b='0'` i `c='0'`. Należy zauważyć, że kolejność umieszczania funkcji przypisywania sygnału nie zmienia wyniku, następujący kod dałby identyczne rezultaty:

```
b <= a;
a <= shift_i;
c <= b;
```

ponieważ tylko warunki w momencie czasu *T* (`shift_i='1'`, `a=b=c='0'`) są używane do obliczania nowego stanu.

5.4. Abstrakcyjny poziom behawioralny

Poziom behawioralny umożliwia opis układu na wysokim poziomie abstrakcji, w sposób podobny do klasycznych języków programowania, takich jak C czy Pascal.

5.4.1. Wyrażenie warunkowe *if*

Polecenie warunkowe `if` może występować w różnych wersjach:

- wersja I:

```
if <warunek> then
  <akcja>
end if;
```

- wersja II:

```
if <warunek> then
  <akcja1>
else
  <akcja2>
end if;
```

- wersja III:

```
if <warunek1> then
  <akcja1>
elsif <warunek2> then
  <akcja2>
end if;
```

- wersja IV:

```
if <warunek1> then
  <akcja1>
elsif <warunek2> then
  <akcja2>
else
  <akcja3>
end if;
```

Należy być ostrożnym podczas wykorzystywania operatorów porównania, takich jak: równe, mniejsze niż, większe niż etc. i zapewnić równość rozmiarów obu stron relacji. W przypadku porównania wektora 8-bitowego z wektorem o innym rozmiarze, nie otrzymamy zamierzonych rezultatów. Na przykład, jeśli `a` jest rejestrem 6-cio bitowym o wartości `"000101"` wówczas porównanie:

```
if a="000101";
```

będzie prawdą, ale wyrażenie:

```
if a="0101";
```

nigdy nie będzie prawdziwe. Dla uniknięcia takich problemów należy się upewnić, czy długości porównywanych ciągów są takie same.

Składnia `if` może być używana tylko wewnątrz procesów, funkcji i procedur.

5.4.2. Wyrażenie warunkowe *case*

Wyrażenie warunkowe `case` służy do selektywnego przypisania wartości wewnątrz procesów. Dla każdego z warunków może wystąpić wielokrotna akcja. Przykład:

```
case sel is
  when "00" => led_o <= Data1;
               state  <= S1;
  when "01" => led_o <= Data2;
               state  <= S2;
  when "10" => led_o <= Data3;
               state  <= S3;
```

```
  when "11" => led_o <= Data4;
               state  <= S1;
end case;
```

Wyrażenie `case` musi sprawdzić wszystkie możliwe wartości sygnału testowanego. Jeżeli jednak nie wszystkie wartości są istotne, to można zastosować następującą składnię:

```
  when others => Akcja;
```

lub:

```
  when others => null;
```

gdzie `null` oznacza, że nie zostanie podjęta żadna akcja. Dodatkowo, można używać pionowej kreski ‘`|`’ dla sprawdzenia więcej niż jednego warunku, który prowadzi do tej samej akcji:

```
  when "00" | "01" => Output <= Dataval;
```

co oznacza, że zarówno dla "`00`" jak i "`01`" będzie wykonana taka sama akcja.

Poszczególne opcje wymienione po słowie kluczowym `when` nie mogą na siebie zachodzić.

5.4.3. Polecenia pętli *loop*

Polecenie `loop` pozwala na wielokrotne wykonanie poleceń. Istnieją trzy różne wersje polecenia pętli `loop`, które zostały opisane poniżej.

Polecenie *loop*

Składnia polecenia w najprostszej postaci wygląda następująco:

```
  <etykieta>: loop
    <polecenia sekwencyjne>
  end loop;
```

Użycie etykiety `<etykieta>` jest opcjonalne.

Oprócz zwykłych poleceń sekwencyjnych, w części sekwencyjnej można także użyć poleceń `next` oraz `exit` opisanych w rozdziale 5.4.4 i 5.4.5.

Pętla `loop` powinna zawierać w części sekwencyjnej przynajmniej jedno polecenie opóźnienia (np. `wait` lub `after`), gdyż w przeciwnym przypadku może nastąpić zapętlenie wykonywania się instrukcji.

Polecenie *while...loop*

Składnia polecenia `while`...`loop` jest następująca:

```
  <etykieta>: while <warunek> loop
    <polecenia sekwencyjne>
  end loop;
```

Polecenie `while`...`loop` zawiera warunek zwracający wartość boolowską. Jeśli warunek `<warunek>` jest spełniony (`true`), część sekwencyjna pętli jest wykonywana jeden raz, po czym następuje ponowne obliczenie i sprawdzenie warunku `<warunek>`. Polecenia z części sekwencyjnej wykonywane

są tak długo, jak długo warunek polecenia `loop` jest spełniony. Użycie etykiety `<etykieta>` jest opcjonalne.

Oprócz zwykłych poleceń sekwencyjnych, w części sekwencyjnej można także użyć poleceń `next` oraz `exit` opisanych w rozdziale 5.4.4 i 5.4.5.

Pętla `while`...`loop`, podobnie jak prosta pętla `loop`, powinna zawierać w części sekwencyjnej przynajmniej jedno polecenie opóźnienia, gdyż w przeciwnym przypadku może nastąpić zapętlenie wykonywania się instrukcji.

Polecenie *for...loop*

Składnia polecenia `for`...`loop` wygląda następująco:

```
<etykieta>: for <identyfikator> in <zakres> loop
  <polecenia sekwencyjne>
end loop;
```

Pętla `for`...`loop` posiada identyfikator `<identyfikator>`, który jest lokalną zmienną tylko dla pętli. Jeśli poza pętlą istnieje taki sam identyfikator, to na czas wykonywania pętli staje się on niewidoczny dla poleceń pętli, ale nie jest on modyfikowany. Identyfikator `<identyfikator>` może być tylko czytany wewnątrz pętli, nie można do niego podstawiać wartości. Wartością identyfikatora mogą być tylko liczby całkowite. Zakres `<zakres>` może być określony za pomocą `to` lub `downto`. Użycie etykiety `<etykieta>` jest opcjonalne.

Oprócz zwykłych poleceń sekwencyjnych, w części sekwencyjnej można także użyć poleceń `next` oraz `exit` opisanych w rozdziale 5.4.4 i 5.4.5.

Pętla `for`...`loop` wykonuje się w następujący sposób:

1. Najpierw następuje zadeklarowanie nowej zmiennej całkowitej `<identyfikator>` widocznej tylko wewnątrz pętli.
2. Zmiennej `<identyfikator>` przypisana jest pierwsza wartość z zadanego zakresu `<zakres>`, a następnie wykonane zostają jeden raz wszystkie polecenia sekwencyjne.
3. Następuje przypisanie zmiennej `<identyfikator>` kolejnej wartości z zakresu `<zakres>` i wykonują się polecenia sekwencyjne.
4. Krok 3. jest powtarzany, dopóki zmienna nie osiągnie ostatniej wartości z zakresu. Po przypisaniu ostatniej wartości, następuje ostatnie wykonanie poleceń sekwencyjnych, a następnie wyjście z pętli.

Pętle mogą ułatwić operacje na tablicach, bez wcześniejszego określenia rozmiaru tablicy. Zakładając, że zmienne `a` i `b` są tablicami o takiej samej liczbie elementów, można napisać przypisanie dla wszystkich elementów tablic jako:

```
for i in a'range loop
```

```
    a(i) <= not b(i);
  end loop;
```

Przykład typowego błędu w wykorzystaniu sygnału w pętli

Projektant miał zamiar napisać kod VHDL wykonujący operację `and` na wszystkich bitach 8-bitowej magistrali:

```
entity test_sig is port (
          a_i : in std_logic_vector(7 downto 0);
          x_o : out std_logic);
end entity;

architecture beh of test_sig is
    signal x: std_logic;
begin
    process (a_i)
    begin
          x <= '1';
          for i in 7 downto 0 loop
                x <= a_i(i) and x;
          end loop;
    end process;
x_o <= x;
end architecture;
```

Początkowo wektor `a_i` ma wartość `"00000000"`. Jednak dla wartości `"11111111"`, wyjście `x_o`, według oczekiwań projektanta, powinno przyjąć wartość `'1'`.

Niestety tak się nie dzieje. Problem wynika z zasady działania przypisania wartości do sygnału – pętla `for` wykonuje się w zerowym czasie, a wartość `x` jest uaktualniana dopiero po jej zakończeniu. Poniższa analiza odbywa się przy założeniu, że `a_i="11111111"`. Domyślnie wartość `x` jest równa `'U'`. Po wejściu do procesu następuje rozpisanie przypisania `x <= '1';` ale wartość ta będzie mogła być uaktualniona dopiero pod koniec aktualnego kroku czasowego. Następna jest instrukcja `x <= a_i(i) and x` wewnątrz pętli `for`, która działa na aktualnej wartości `x` (a `x` jest jeszcze równe `'U'`!), w wyniku jej działania do `x` jest rozpisanie przypisanie wartości `'U'`. Ponieważ na koniec danego kroku czasu brane jest pod uwagę tylko ostatnie przypisanie, to wartość `x` będzie dalej równa `'U'`.

Rozwiązaniem tego problemu sprowadza się do wykorzystania tymczasowej zmiennej `variable`, która, w przeciwieństwie do sygnału jest uaktualniana natychmiastowo:

```
entity test_var is port (
          a_i : in std_logic_vector(7 downto 0);
          x_o : out std_logic);
end entity;
```

```
architecture beh of test_var is
begin
    process (a_i)
    variable tmp: std_logic;
    begin
          tmp := '1';
          for i in 7 downto 0 loop
                tmp := a_i(i) and tmp;
          end loop;
          x_o <= tmp;
    end process;
end architecture;
```

W wyniku działania poprawionego kodu, otrzymuje się poprawną wartość sygnału **x**, zgodną z zamierzeniami projektanta.

5.4.4. Polecenie *next*

Składnia polecenia **next** jest następująca:

```
next <etykieta> when <warunek>;
```

przy czym zarówno **<etykieta>** jak i część **when <warunek>** jest opcjonalna.

W najprostszej postaci polecenie **next** wygląda następująco:

```
next;
```

i powoduje zakończenie wykonywania aktualnej, najbardziej zagnieżdżonej iteracji pętli, a następnie skok do początku pętli i rozpoczęcie następnej iteracji.

Gdy w poleceniu **next** występuje część **when**, wtedy polecenie to wykona się tylko wtedy, gdy **<warunek>** zwraca wartość **TRUE**.

Użycie etykiety umożliwia przerwanie dowolnej pętli, a nie tylko aktualnej, najbardziej zagnieżdżonej. W poniższym przykładzie następuje przerwanie aż dwóch pętli jednocześnie:

```
petla_1: for I in 1 to 8 loop
  k := 0;
  petla_2: loop
    B(k) := '0';
    next petla_1 when warunek_1;
    k := k + 1;
  end loop petla_2;
end loop petla_1;
```

Rys. 44. Przykład polecenia next.

5.4.5. Polecenie *exit*

Składnia ogólna polecenia **exit** jest podobna do polecenia **next**:

```
exit <etykieta> when <warunek>;
```

przy czym zarówno **<etykieta>** jak i część **when <warunek>** jest opcjonalna.

Polecenie `exit` powoduje przerwanie pętli oznaczonej etykietą `<etykieta>`. Gdy nie podano etykiety, następuje przerwanie najbardziej wewnętrznej aktualnej pętli, podobnie jak w poleceniu `next`. Różnica pomiędzy poleceniem `next` i `exit` polega na tym, że `next` przerywa polecenia sekwencyjne i rozpoczyna następną iterację a polecenie `exit` przerywa polecenia sekwencyjne i kończy działanie pętli.

5.4.6. Podstawowe rodzaje procesów

W tym rozdziale przedstawiono podstawowe rodzaje procesów. Przestrzeganie poniższych zaleceń dotyczących opisu układu za pomocą procesów spowoduje, że program wykonujący syntezę poprawnie zrealizuje układ zgodnie z zamysłem projektanta.

Procesy sekwencyjne

Procesy sekwencyjne są używane do syntezy układów z przerzutnikami, rejestrami lub dowolnymi innymi elementami sterowanymi zboczem sygnału zegara. W procesach sekwencyjnych lista czułości procesu zawiera sygnał zegara i opcjonalnie asynchroniczny sygnał *reset* (jest to reguła projektowania układów ASIC/FPGA, a nie VHDL). W procesie takim może wystąpić tylko jeden sygnał zegarowy oraz jeden sygnał zerowania, jeśli jest więcej niż jeden *reset*, to powinny być one zsumowane logicznie funkcją `or` poza procesem.

Rozważmy najprostszy przypadek, kiedy nie ma sygnału zerowania (*reset*) oraz załóżmy, że nasza biblioteka zawiera przerzutniki wyzwalane zboczem narastającym.

Proces będzie oczekiwał na narastające zbocze sygnału i wtedy ładował zawartość szyny danych `data8` do rejestru `reg8`, o których załóżmy, że były wcześniej zdefiniowane jako wektory np. ośmiobitowe. Proces będzie wyglądał następująco:

```
process (clk_i)
begin
  if rising_edge(clk_i) then
    reg8 <= data8;
  end if;
end process;
```

Proces jest aktywowany, kiedy tylko następuje zmiana w jakimś sygnale umieszczonym na liście czułości. W przypadku naszego zegara (`clk_i`), jedyne zmiany zachodzą w czasie narastającego (*rising*, *leading edge*) lub opadającego (*falling*, *trailing edge*) zbocza. W przypadku kodu przedstawionego powyżej, wykrywane jest tylko zbocze narastające za pomocą funkcji `rising_edge`. Podczas zbocza opadającego warunek `if rising_edge(clk_i)` nie jest spełniony, a proces zostaje zakończony lub raczej zawieszony do następnej zmiany sygnałów z listy czułości. Użycie wyrażenia `if-then`, pokazanego powyżej, jest intuicyjne i oznacza: gdy pojawi się narastające zbocze sygnału `clk_i`,

wówczas przepisz zawartość szyny `data8` do rejestru `reg8`, w przeciwnym wypadku nic nie rób. Taki kod zostanie zsyntezowany jako prosty rejestr złożony z przerzutników typu D bez wejść zerujących.

W powyższym procesie występuje następujące przypisanie sygnałów:

```
reg8 <= data8;
```

Program syntezy logicznej automatycznie wygeneruje rejestr, w którym liczba jego bitów została wcześniej podana w definicji sygnałów, np.

```
signal reg8 : std_logic_vector(7 downto 0);
```

Jeśli zostanie zwiększona szerokość `reg8` do 32 bitów, to wówczas otrzymamy rejestr 32-bitowy. Ta pojedyncza linia kodu może być użyta do generacji rejestru niezależnie od tego czy ma on jeden, czy też *N* bitów. Ten fakt jest jednym z powodów wysokiej efektywności syntezy logicznej, realizowanej zarówno w stosunku do układów sekwencyjnych jak i kombinacyjnych.

Rozważmy teraz w jaki sposób dodać asynchroniczne wejście kasowania, aktywowane poziomem wysokim. Będzie to wymagało umieszczenia dwóch sygnałów na liście czułości. Ponieważ są dwa sygnały, to musimy zdecydować, który z nich będzie miał wyższy priorytet i w momencie wystąpienia obu naraz będzie decydował o końcowym stanie procesu. W praktyce sygnał zerowania ma wyższy priorytet, więc i tu zostanie to zaimplementowane:

```
process (clk_i, rst_i)
begin
  if rst_i='1' then
    reg8 <= "00000000"; --wyczyść rejestr reg8
  elsif rising_edge(clk_i) then
    reg8 <= data8;
  end if;
end process;
```

Wyrażenie `elsif`, będące skrótem od `else if`, jest aktywne tylko wtedy, jeśli pierwszy warunek wyrażenia `if` jest fałszem, tj. gdy `rst_i` ma stan niski. Należy zauważyć, że priorytet jest ustalany poprzez wyrażenie `if/elsif`. W rzeczywistości część `elsif` oznacza „tylko wtedy gdy `rst_i='0'` sprawdź akcję uruchamianą zboczem zegarowym”. Innymi słowami mówiąc, jeśli oba warunki pojawią się jednocześnie, priorytet ma sygnał `rst_i`. Podsumowując, oto najważniejsze zasady konstruowania procesu sekwencyjnego:

- zarówno sygnał zegara jak i *resetu* są umieszczone na liście czułości, na której może być umieszczony tylko jeden zegar i jeden *reset* w procesie,
- za wyrażeniem `if rst_i` ... następuje opis akcji, która powinna nastąpić gdy pojawi się *reset*,
- wyrażenie `elsif rising_edge` ... służy do obsługi zdarzeń związanych z sygnałem zegara.

Procesy kombinacyjne

Logika kombinacyjna może zostać opisana poza procesem lub poprzez proces kombinacyjny spełniający odpowiednie wymagania. Dla zilustrowania, rozważmy prosty multiplekser z wejściami `a`, `b` oraz `sel` i wyjściem `y`. Wyjście `y` będzie równe `a` kiedy `sel` ma stan niski oraz równe `b` kiedy `sel` jest w stanie wysokim. Aby w wyniku syntezy powstał układ kombinacyjny, należy na liście czułości procesu umieścić wszystkie sygnały wejściowe (w tym przypadku `a`, `b` i `sel`). Jakakolwiek zmiana któregoś z sygnałów wejściowych powoduje aktywację procesu i obliczenie stanu wyjściowego `y`.

```
process(a, b, sel) -- multiplekser 2 na 1
begin
  if sel='0' then
    y<=a;
  else
    y<=b;
  end if;
end process;
```

Ważną obserwacją jest to, że nie ma tu żadnej operacji typu `rising_edge` lub `falling_edge`, tak jak to było w procesach sekwencyjnych. Dodatkowo, powyższy zapis umożliwia obliczenie wartości `y` dla wszystkich kombinacji wartości sygnałów wejściowych. W rezultacie taki proces będzie prowadził do logiki kombinacyjnej.

Sygnały `a`, `b` i `y` mogą być pojedynczymi bitami lub też słowami 32 bitowymi, w zależności od tego jak zostały one zadeklarowane w architekturze. Jest to kolejny przykład na to, że liczba linii w kodzie VHDL nie musi być proporcjonalna do liczby syntezowanych bramek. Układ 32 bitowy byłby z pewnością 32 razy większy niż jednobitowy, a opis w VHDL jest podobny, różniący się jedynie deklaracją sygnałów.

Należy zauważyć, że współbieżne przypisanie ciągłe:

```
y <= a and b;
```

jest równoważne następującemu procesowi kombinacyjnemu:

```
process (a, b)
begin
  y <= a and b;
end process;
```

Procesy zatrzaskowe

Trzecia odmiana procesów jest używana do tworzenia zatrzasków (ang. *latch*), tzn. elementów pamięciowych sterowanych poziomem. Zatrzaski są "przezroczyste" w czasie, gdy sygnał zezwalający np. `enable` (lub sygnał zegarowy) jest w stanie aktywnym, w przeciwnym wypadku zapamiętywany jest ostatni stan wyjść, gdy sygnał `enable` był jeszcze aktywny. Forma składniowa procesu zatrzaskowego jest podobna do sekwencyjnego z tym, że na liście „*sensitivity list*” znajdują się

wszystkie sygnały wejściowe. Zakładając, że zatrzask ma sygnał zezwolenia **enable** aktywowany stanem wysokim proces taki może wyglądać następująco:

```
process(enable, rst_i, data8)
begin
    if rst_i='1' then --zerowanie zatrzasku
          latch8 <= "00000000";
    elsif enable='1' then --zatrzask aktywny
          latch8=data8;
    end if;
end process;
```

Poza zerowaniem (**rst_i**), proces jest aktywowany dowolną zmianą sygnału **enable** lub wejścia **data8**. Dodanie szyny danych **data8** do listy czułości procesu zapewnia, że jeśli sygnał **enable** jest w stanie wysokim, to zmiany wejść przenoszą się na wyjście.

5.4.7. Opóźnienia typu *wait*

Słowo kluczowe **wait** umożliwia wstrzymanie działania procesu lub procedury przez określony czas. Istnieją następujące różne wersje polecenia **wait**: **wait for**, **wait until**, **wait on** i **wait**.

Polecenie *wait for*

wait for <czas> – powoduje wstrzymanie procesu przez określony czas, np.:

```
wait for 10 ns;
```

Polecenie *wait until*

wait until <warunek_sygnału> – powoduje wstrzymanie procesu, dopóki nie zmieni się wartość sygnału występującego w warunku **<warunek_sygnału>**. Proces będzie kontynuowany, gdy po zmianie sygnału okaże się, że warunek **<warunek_sygnału>** jest spełniony.

```
wait until enable='1';
```

Przykład:

```
architecture rtl of test_wait is
signal a, x: std_logic;
begin
    process
    begin
          x <= '0';
          wait until a='1';
          x <= '1';
          wait for 5 ns;
    end process;
end architecture;
```

Polecenie *wait on*

`wait on <lista_czułości>` – powoduje wstrzymanie procesu, aż do momentu gdy zmieni się przynajmniej jeden z sygnałów umieszczonych na liście `<lista_czułości>`.

```
wait on S1, S2;
```

Polecenie *wait*

`wait` – samo polecenie `wait` powoduje wstrzymanie procesu w nieskończoność:

```
wait;
```

Polecenie `wait` wstrzymujące proces w nieskończoność jest często stosowane w procesach generujących sygnały testowe *test bench* podczas symulacji. Jeśli polecenie `wait` występuje w procesie, to proces nie może posiadać listy czułości.

Różne postaci polecenia `wait` można łączyć, np.:

```
architecture rtl of test_wait_mixed is
signal a, b, x: std_logic;
begin
  process
  begin
    x <= '0';
    wait on b until a='1';
    x <= '1';
    wait for 5 ns;
  end process;
end architecture;
```

5.5. Funkcje i procedury

Funkcje i procedury w VHDL są podprogramami używanymi w celu uproszczenia kodowania operacji wykorzystywanych wielokrotnie. Jeśli pewien blok kodu jest używany wielokrotnie, może zostać umieszczony w funkcji lub procedurze jednokrotnie, a następnie być wywoływany poprzez jej nazwę.

5.5.1. Funkcje

Definicja funkcji składa się z:

- nazwy,
- specyfikacji wejść,
- specyfikacji typu wyjścia.

Funkcje nie mogą zmieniać wartości parametrów umieszczonych na jej liście (procedury mogą). Parametry na liście funkcji mogą należeć do klasy m.in. `constant` lub `signal` i mogą mieć tylko kierunek `in`. Z tego powodu nie ma konieczności podawania kierunku parametrów funkcji – są one

domyślnie przyjmowane jako `in`. Jeżeli nie podano klasy parametru, domyślnie przyjmuje się klasę `constant`.

Przykład szkieletu definicji funkcji:

```
function rising_edge(signal CLK: std_logic) return boolean is
-- część deklaracyjna
begin
  -- ciało funkcji
return (VALUE);
end function rising_edge:
```

Przykład wywołania funkcji

```
rising_edge(ENABLE);
```

Parametry na liście deklaracyjnej funkcji nazywają się parametrami formalnymi. Ich wartości są zastępowane parametrami aktualnymi w czasie wywołania funkcji. Na przykład w powyższym wywołaniu sygnał `ENABLE` staje się parametrem aktualnym i zastępuje sygnał `CLK` we wnętrzu funkcji. Klasa parametru formalnego i aktualnego musi być taka sama z wyjątkiem parametrów formalnych zadeklarowanych jako `constant`. W takim przypadku parametr aktualny może być zmienną, sygnałem, stałą lub wyrażeniem.

Funkcje i procedury mogą być zagnieżdżone. Wyrażenia `wait` nie są dopuszczalne w ciele funkcji (jest to możliwe w procedurze, patrz rozdział 5.5.2) i dlatego funkcje są wykonywane w zerowym czasie. Z tego powodu wyrażenie `wait` nie może znajdować się również w procedurze wywoływanej przez funkcję.

Funkcje mogą być wywoływane rekurencyjnie. Można definiować funkcje o nazwach zarezerwowanych dla operatorów lub innych funkcji – nazywa się to przeciążaniem operatora lub funkcji.

Przykład funkcji

Poniżej przedstawiony jest przykład funkcji zamieniającej wartości typu `std_logic_vector` na `bit_vector`. Deklaracja funkcji nie zawiera szerokości parametru wejściowego ani wartości wyjściowej. Jest to ustalane podczas wywoływania funkcji, w czasie wstawienia parametrów aktualnych.

```
function to_bitvector(IN_VAL:std_logic_vector) return bit_vector is
    variable RET_VAL:bit_vector(IN_VAL'range);
begin
    for I in IN_VAL'range loop
          case IN_VAL(I) is
                when '0' => RET_VAL(i) := '0';
                when '1' => RET_VAL(i) := '1';
                when others => RET_VAL(i) := '0';
```

```
            end case;
        end loop;
        return (RET_VAL);
    end function to_bitvector;
```

5.5.2. Procedury

Procedury są podprogramami, które mogą zmieniać wartość jednego lub więcej parametrów wejściowych. Parametry procedury mogą należeć do klasy: `constant`, `signal` lub `variable`. Pierwszą różnicą w stosunku do funkcji jest to, że parametry procedury mogą mieć kierunek `in`, `out` lub `inout`. Jeśli klasa i kierunek parametru formalnego nie są zdeklarowane, wówczas domyślnie przyjmuje się kierunek `in` oraz klasę `constant`.

Tak jak w przypadku funkcji, typ parametru formalnego musi być zgodny z typem parametru aktualnego wstawianego w czasie wywoływania procedury. Zmienne zdeklarowane w procedurze są inicjowane w czasie każdego wywołania procedury, a ich wartości nie przechodzą do innych wywołań tej samej procedury.

Procedura może nie posiadać żadnych parametrów. Można przeciążać procedury, tj. w programie może występować jednocześnie kilka procedur o takich samych nazwach, ale o różnych parametrach. Procedury można zagnieżdżać oraz wywoływać rekurencyjnie. Procedury są zazwyczaj syntezowalne, dopóki nie zawierają poleceń `wait`.

Poniżej zamieszczono przykład procedury, wykonującej odczyt z pamięci QDR:

```
procedure QDR_read (
          read_addr: in std_logic_vector(7 downto 0);
          signal read_phase  : in std_logic;
          signal write_phase : in std_logic;
          signal QDR_rps_n_i : out std_logic;
          signal QDR_a_read: out std_logic_vector(23 downto 0)) is
begin
          QDR_rps_n_i<='Z';
          QDR_a_read  <= (others => 'Z');
          wait until read_phase='1';
          QDR_rps_n_i<='0';
          QDR_a_read  <= (others => '0');
          QDR_a_read(7 downto 0) <=  read_addr;
          wait until read_phase='0';
          QDR_rps_n_i <= 'Z';
          QDR_a_read  <= (others => 'Z');
end procedure;
```

Przykład wykorzystania powyższej procedury:

```
process
```

```
...
begin
  ...
  for i in 0 to 3 loop
    QDR_read (address_v, read_phase, write_phase,
              QDR_rps_n_i, QDR_a_read);
    address_v := address_v + 1;
    random_delay(seed_v);
  end loop;
  ...
end process;
```

Sygnały nie mogą być deklarowane w procedurach, ale mogą być do nich przekazywane jako parametry. Stosownie do zasad widzialności obiektów, procedury mogą zmieniać wartości sygnałów nie wyszczególnionych w liście parametrów. Na przykład procedura zadeklarowana w procesie może zmieniać sygnały zadeklarowane w bloku `entity`. Jest to możliwe, gdyż sygnały i porty są widoczne we wnętrzu procedury. O takiej procedurze mówi się, że ma efekt uboczny, gdyż zmienia wartości sygnałów nie wymienionych na liście parametrów. Bardziej czytelne jest stosowanie przekazywania informacji poprzez listę parametrów.

Procedury mogą być umieszczane w części deklaracyjnej procesu. Jak wiadomo, proces nie może mieć równocześnie listy czułości jak i wyrażenia typu `wait`. Z tego wynika, że proces, w którym następuje wywołanie procedury zawierającej wyrażenie `wait`, nie może posiadać listy czułości.

6. Podsumowanie

Niniejszy skrypt przeznaczony jest jako pomoc dydaktyczna dla studentów kierunku Inżynieria Biomedyczna prowadzonego na Wydziale Elektroniki Telekomunikacji i Informatyki Politechniki Gdańskiej do przedmiotu p.t. „Języki modelowania i symulacji”. Oprócz tego skryptu, dostępne są również slajdy do wykładu oraz materiały pomocnicze do laboratorium. Zakres przedstawionego w skrypcie materiału z nadwyżką pokrywa treści wymagane do zaliczenia przedmiotu. Znajomość metod opisu i symulacji analogowych i cyfrowych układów elektronicznych jest bardzo ważna dla inżynierów i projektantów urządzeń elektronicznych. Autorzy żywią nadzieję, że niniejsze opracowanie przyczyni się do znacznego poszerzenia wiedzy czytelników w tym zakresie.

Literatura

[1] J. Izydorczyk, „PSpice komputerowa symulacja układów elektronicznych”, Helion, 1993.

[2] Cadence Design Systems, „PSPICE reference guide”, Cadence PCB Design Systems, 2001.

[3] P. Antognetti, Massobrio, „Semiconductor Device Modeling with SPICE”, McGraw-Hill, 1988.

[4] S. Palnitkar, „VERILOG HDL a Guide to Digital Design and Synthesis”, SunSoft Press, 1996.

[5] K. Coffman, „Real Word FPGA Design with Verilog”, Prentice-Hall, 2000.

[6] M. G. Arnold, „Verilog Digital Computer Design, Algorithms into Hardware”, Prentice-Hall, 1999.

[7] A. Yalamanchili, „Introductory VHDL: From Simulation to Synthesis”, Prentice-Hall, 2001.

[8] K. Skahill, „Język VHDL, Projektowanie programowalnych układów logicznych”, WNT, 2001.

[9] D. Pellerin, D. Taylor, „VHDL Made Easy!”, Prentice Hall, 1997.

[10] D. E. Ott, T. J. Wilderotter, „A Designer’s Guide to VHDL Synthesis”, Kluwer Academic Publishers, 1994.

Załącznik 1. Lista ważniejszych skrótów

ASIC	ang. *Application Specific Integrated Circuit*
CAD	ang. *Computer Aided Design*
CPLD	ang. *Complex Programmable Logic Device*
FPGA	ang. *Field Programmable Gate Array*
IEEE	ang. *Institute of Electrical and Electronics Engineers*
MVL4	ang. *Multi-Valued Logic*
PSPICE	ang. *Personal Computer Simulation Program with Integrated Circuit Emphasis*
RTL	ang. *Register Transfer Level*
SI	franc. *Système international d'unités*
SPICE	ang. *Simulation Program with Integrated Circuit Emphasis*
THD	ang. *Total Harmonic Distortion*
UUT	ang. *Unit Under Test*
VHDL	ang. *Very High Speed Integrated Circuit Hardware Description Language*
VHDL-AMS	ang. *VHDL - Analog Mixed Signal*
VHPI	ang. *VHDL Procedural Interface*
VLSI	ang. *Very Large Scale of Integration*