

Metody projektowania scalonych układów cyfrowych z wykorzystaniem języków VHDL i Verilog HDL[®]

***Opracował:
Bogdan Pankiewicz,
Marek Wójcikowski***

Gdańsk, 1999r.

Spis treści

Spis treści	1
CZĘŚĆ 1. VHDL	4
1. Zakres materiału	4
2. Geneza powstania VHDL	4
3. Zastosowania języka VHDL	4
4. Poziomy opisu	5
4.1. Abstrakcyjny poziom behawioralny (ang. <i>behavior</i> – zachowanie)	5
4.2. Poziom przesłań międzyrejestrów (ang. <i>register-transfer level RTL</i>)	5
4.3. Poziom strukturalny (ang. <i>structural level</i>)	6
5. Komentarze, konwencja używania nazw, słowa zarezerwowane, literały	6
5.1. Konwencja używania nazw, słowa zarezerwowane	6
5.2. Literały	7
6. Jednostki projektowe (ENTITY)	8
6.1. Komentarze	9
7. Sygnały	10
8. Architektury jednostek (ARCHITECTURE)	12
9. Operacja przypisania wartości sygnału ($\leq$)	15
9.1. Operacje współbieżne oraz czasowe	16
10. Operatory	17
11. Obiekty	18
12. Typy, deklarowanie własnych typów (TYPE)	18
12.1. Deklaracja niekompletna typu	19
12.2. Typy skalarne	19
12.3. Typy złożone	19
12.4. Podtypy	19
13. Atrybuty	20
14. Instrukcje współbieżne i sekwencyjne	20
15. Organizacja bibliotek, biblioteka STD, zakres dostępności bibliotek	21
16. Biblioteka 1164, typ MVL9, przykłady zrealizowanych funkcji	22
17. Procesy (PROCESS), zegarowane, zatraskowe, kombinacyjne	23
17.1. Procesy zegarowane	23
17.2. Procesy niezegarowane (kombinacyjne)	25
17.3. Procesy zatraskiwane	26
17.4. Przegląd mechanizmów ustalania poziomów sygnałów	26
17.5. Zdarzenia zegarowane i niezegarowane	28
18. Wyrażenia warunkowe (IF, CASE, WHEN, SELECT)	29
18.1. Wyrażenie warunkowe IF	29
18.2. Wyrażenie warunkowe WHEN	30
18.3. Wyrażenie warunkowe SELECT	30
18.4. Wyrażenie warunkowe CASE	31
19. Tablice (ARRAY)	32
20. Funkcje i Procedury	33

20.1. Istota funkcji	33
20.2. Funkcja arbitrażowa	36
20.3. Istota procedury	36
21. Synteza układów sekwencyjnych	37
21.1. Podstawowe wiadomości o licznikach	37
21.2. Podstawowy licznik binarny	38
21.3. Jaki jest wynik procesu syntezy logicznej?	39
21.4. Projektowanie licznika w języku VHDL	40
21.5. Dodanie wejść ENABLE oraz LOAD (zezwalającego oraz ładującego)	41
21.6. Logika kontrolna i maszyny stanów.	44
21.7. Podstawy projektowania logiki kontrolnej.	44
21.8. Sekwencer bezwarunkowy	45
21.9. Proces z wielokrotnym zegarem	47
21.10. Detektor zbocza narastającego	48
21.11. Generator impulsów wyzwalany wielokrotnie	51
22. Opis na poziomie strukturalnym	55
22.1. Projektowanie hierarchiczne	59
23. Symulacje i testowanie	60
23.1. Struktura TEST BENCH	61
23.2. Generacja sygnałów	62
23.3. Instrukcja kontroli	66
23.4. Kompletny przykład TEST BENCH	66
24. Uwagi dotyczące kodu VHDL przeznaczonego do syntezy.	68
25. Dodatek: rozwinięcie deklaracji ENTITY, pętle, konfiguracje, przeładowanie nazw, pakiety i operacje na plikach.	69
25.1. Rozwinięcie deklaracji ENTITY (generic, buffer, linkage)	69
25.2. Pętle	71
25.3. Blok konfiguracji (configuration)	73
25.4. Przeładowanie podprogramów i operatorów (over loading)	75
25.5. Pakiety (packages)	76
25.6. Podstawowe operacje wej/wyj – operacje na plikach (I/O)	77
26. Literatura	82
CZĘŚĆ 2. VERILOG	83
1. WSTĘP	83
1.1. Ewolucja systemów CAD	83
1.2. Rozwój języków HDL	83
1.3. Typowy przebieg projektowania	83
1.4. Znaczenie języków HDL	84
1.5. Popularność Verilog HDL	84
1.6. Kierunki rozwoju języków HDL	84
2. KONCEPCJA MODELOWANIA HIERARCHICZNEGO	86
2.1. Metodologia projektowania	86
2.2. Licznik 4-bitowy z przeniesieniem	86
2.3. Moduły	87
2.4. Obiekty	88
2.5. Symulacja	88
2.6. Przykład	89
3. PODSTAWOWE POJĘCIA	92
3.1. Składnia	92
3.2. Typy danych	94
3.3. Zadania systemowe i dyrektywy kompilatora	97
4. MODUŁY I PORTY	99

4.1. Moduły	99
4.2. Porty	100
5. MODELOWANIE NA POZIOMIE BRAMEK LOGICZNYCH	102
5.1. Typy bramek	102
5.2. Opóźnienia w bramkach	105
6. MODELOWANIE NA POZIOMIE REJESTRÓW (Dataflow)	108
6.1. Przypisanie ciągle	108
6.2. Opóźnienia	108
6.3. Wyrażenia, operandy i operatory	109
6.4. Operatory	110
6.5. Przykłady	114
7. MODELOWANIE NA POZIOMIE BEHAWIORALNYM	121
7.1. Procedury strukturalne	121
7.2. Przypisanie proceduralne	122
7.3. Sterowanie wykonywaniem instrukcji	124
7.4. Wyrażenia warunkowe	127
7.5. Wyrażenia typu case	127
7.6. Pętle	128
7.7. Bloki sekwencyjne i równoległe	129
8. ZADANIA I FUNKCJE	131
8.1. Różnice pomiędzy zadaniami i funkcjami	131
8.2. Zadania	131
8.3. Funkcje	132
9. TECHNIKI MODELOWANIA	134
9.1. Proceduralne przypisanie ciągle	134
9.2. Definiowanie parametrów	135
9.3. Warunkowa kompilacja i uruchamianie	136
9.4. Skala czasu	137
9.5. Zadania systemowe	137
10. Literatura	139

CZĘŚĆ 1. VHDL¹

1. Zakres materiału

Niniejsze opracowanie zawiera wiadomości podstawowe o języku opisu sprzętu VHDL. Zakres materiału został tak dobrany aby wystarczał do rozpoczęcia projektowania systemów cyfrowych z użyciem języka VHDL. Przedstawione są liczne przykłady opisu podsystemów cyfrowych pozwalające w sposób przystępny zrozumieć główne zasady i reguły języka z szczególnym naciskiem na wykorzystanie w celach projektowych.

2. Geneza powstania VHDL

Język VHDL (*VHSIC Hardware Description Language*, gdzie VHSIC oznacza *Very High Speed Integrated Circuit*) powstał w wyniku prac badawczych prowadzonych przez amerykańskie firmy IBM, Intermetics oraz Texas Instruments, a sponsorowanych przez Departament Obrony USA. Prace rozpoczęto w lipcu 1983, a zakończono w roku 1985. W roku 1987 język VHDL został przyjęty jako standard (o nr 1076-1987) przez organizację IEEE (*Institute of Electrical and Electronics Engineers*), a w roku 1993 znowelizowano standard, który otrzymał numer 1076-1993. Standard 1076-1987 stanowi podstawę wszystkich narzędzi do syntezy i symulacji, zmiany wprowadzone w standardzie 1076-1993 są kosmetyczne i zostały uwzględnione w większości dostępnych obecnie narzędzi wspomagających projektowanie.

Standard 1076 opisywał dokładnie składnię języka, nie określał natomiast typów danych, przez co producenci narzędzi wspomagających projektowanie definiowali typy danych wykorzystywane podczas symulacji według własnego uznania. W efekcie wymiana modeli do symulacji pomiędzy różnymi programami do symulacji stała się utrudniona. Aby umożliwić swobodną wymianę modeli symulacyjnych, 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**.

Dodatkowo w roku 1995 zostały powołane standardy 1076.3 oraz 1076.4 rozszerzające możliwości poprzednich implementacji języka. Standard 1076.3 opisuje typy danych wykorzystywane podczas symulacji ale blisko skojarzone z reprezentacją sprzętową. Standard 1076.4 precyzuje sposób opisywania opóźnień występujących w układach cyfrowych.

3. Zastosowania języka VHDL

Język VHDL stosuje się do:

- modelowania i symulacji układów cyfrowych;
- wprowadzania projektu do systemów automatycznej syntezy i projektowania;
- weryfikacji projektu (test-bench);
- opisu układu (netlista);
- sporządzania dokumentacji projektu.

¹ Opracowano na podstawie publikacji [1-3] wyszczególnionych w rozdziale 26.

4. Poziomy opis

Każdy język HDL umożliwia wykorzystanie kilku dostępnych poziomów opisu, przy czym podział taki ma charakter umowny. Poniżej przedstawiony jest podział wg [2].

4.1. Abstrakcyjny poziom behawioralny (ang. *behavior* – zachowanie)

Jest to najwyższy poziom opisu systemu w ramach VHDL. VHDL jest tu zastosowany do opisu zachowania się systemu, czyli tego w jaki sposób wyjście układu reaguje na sygnały wejściowe, jednakże nie jest tu brana pod uwagę realizacja fizyczna tego 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. Opis pracy jest bardziej abstrakcyjny, często przy użyciu możliwości VHDL sporządzenia tzw. opisu sygnałów typu wyliczeniowego (ang. *enumerated*) definiowanych przez użytkownika. Mając powyższe na uwadze, można, dla przykładu, problem objaśnić następująco: pewna operacja może zostać nazwana „ADD” bez potrzeby definiowania kodu binarnego, który ją reprezentuje. 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. Jako przykład można wyobrazić sobie system cyfrowy oczekujący na pewną wiadomość, a gdy się ona pojawi na szynie danych ustawiane są pewne sygnały wyjściowe na okres 80 μ s. To, jak wykonane zostanie opóźnienie o wartości 80 μ s i ustawienie poszczególnych wyjść systemu jest przedmiotem szczegółowej implementacji sprzętowej, ale wyrażenie tego na poziomie behawioralnym opisu VHDL nie jest wymagane.

Ogólnie ujmując, 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.

4.2. Poziom przesłań międzyrejestrowych (ang. *register-transfer level* **RTL**)

Poziom przesłań międzyrejestrowych opisu systemu odpowiada szczegółowemu opisowi blokowemu w tradycyjnej metodzie projektowania, w którym operuje się blokami funkcjonalnymi oraz ich funkcjami, sygnałami wejściowymi i wyjściowymi oraz szynami danych. Muszą być zdefiniowane sygnały zegarowe oraz zerujące (ang. *reset*) a szyny danych oraz pamięci (przerzutniki, liczniki, pamięci) muszą mieć przypisane konkretne liczby bitów. Poziom abstrakcji w opisie funkcjonowania systemu jest powyżej poziomu pojedynczych bramek logicznych konwencjonalnego schematu ideowego i może się zmieniać w zakresie od opisu równaniami boolowskimi aż do wyższego poziomu nazywanego Poziomem Przesłań Międzyrejestrowych (RTL).

Użycie nazwy RTL zmieniło się od momentu pierwszego jej wprowadzenia i obecnie pojęcie to może, ale nie musi, mieć związku z rejestrami. Na poziomie przesłań międzyrejestrowych 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 projektowania układów ASIC, metoda ta może zapewnić bardzo dobre rezultaty z punktu widzenia efektywności projektowania. Projekty z opisem na poziomie RTL są także 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.

4.3. Poziom strukturalny (ang. *structural level*)

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 więc 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 I/O, może być symulowany na poziomie bramek. Możliwa jest także symulacja łącznie z innymi układami ASIC oraz dodatkowymi układami dyskretnymi.

Podczas każdej z faz procesu projektowania możliwości strukturalne VHDL pozwalają na łączenia bloków między sobą, a także użycie programów generacji testów, w tym do dobierania kształtu sygnałów pobudzających oraz śledzenia ich postaci na wyjściu.

5. Komentarze, konwencja używania nazw, słowa zarezerwowane, literały

5.1. Konwencja używania nazw, słowa zarezerwowane

VHDL ma listę słów zarezerwowanych (Tabela 1), które są częścią języka i nie mogą być używane dla nazw bloków ENTITY i sygnałów lub innych identyfikatorów. Dodatkowo zbiór znaków ASCII, który może być używany dla nazw i identyfikatorów 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 obok siebie.
Sig_A and BUS_07 są poprawne, ale SIG__A and BUS_ 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.

Język nie jest czuły wielkość liter wyrażenia ENTITY oraz Entity są traktowane identycznie.

Długość nazw używanych dla sygnałów, procesów, jednostek projektowych etc. jest ważna ze względu na zachowanie zgodności z innymi narzędziami projektowymi, zewnętrznymi względem VHDL. Normalnym zbiorem wyjściowym (po syntezie) jest lista

połączeń, która może być wykorzystana w celu weryfikacji układu ASIC do 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 obciążenia 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.

Tabela 1. 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	

5.2. Literały

Literały są to symbole reprezentujące pewne wartości. W języku VHDL wyróżnia się następujące typy literałów:

- całkowity – liczba całkowita zapisana w postaci liczby o podstawie dziesiętnej, dwójkowej, ósemkowej lub szesnastkowej, przykłady:

podstawa dziesiętna: 123, 2e2

podstawa dwójkowa: 2#101001#
podstawa ósemkowa: 8#1234567#
podstawa szesnastkowa: 16#abC#

- rzeczywisty – liczba rzeczywista zapisana w postaci liczby o podstawie dziesiętnej, dwójkowej, ósemkowej lub szesnastkowej, przykłady:

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

- znakowy – znak alfanumeryczny zamknięty w pojedynczym cudzysłowie, przykłady:

'a' , 'A'

- łańcuchowy – ciąg znaków alfanumerycznych zamkniętych w podwójnym cudzysłowie, przykłady:

"abc" , "ABC"

- łańcuchowo-bitowy – ciąg bitów w postaci liczby o podstawie dwójkowej, ósemkowej lub szesnastkowej, przykłady:

"1001_1001" , B"1001_1001" , 0"167" , X"abC"

- fizyczny – literał rzeczywisty po którym następuje określenie jednostki, przykłady:

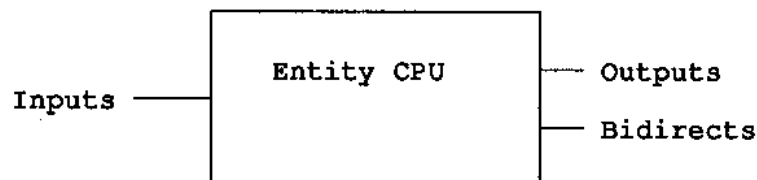
1.5ns , 2kOhm , 60Hz

Dla czytelności zapisu, poszczególne znaki literału mogą być rozdzielone znakiem podkreślenia, nie powoduje on zmiany wartości.

6. Jednostki projektowe (ENTITY)

Gdy chcemy opisać istniejący fizyczny system elektroniczny, można brać pod uwagę zarówno cały system, płytę główną, grupę płytek drukowanych, układy typu MCM (ang. *multi-chip modules*), układy ASIC lub jakąkolwiek część każdego z tych systemów. Niezależnie od wyboru stopnia złożoności zawsze występuje pewna „jednostka” projektowa (*entity*), która komunikuje się z otoczeniem poprzez sygnały. W VHDL taką jednostkę projektową nazywa się **ENTITY**, a jej połączenia (wejściowe, wyjściowe oraz dwukierunkowe) nazywane są **PORTS** tak jak pokazano na rys. 1.

Do bloku **ENTITY** przypisana jest nazwa, w tym przypadku *CPU*, oraz połączenia I/O. Nazwa jednostki pozwala w prosty sposób zidentyfikować tę jednostkę w całym systemie w trakcie jego syntezy.



Rys. 1. Blok ENTITY w języku VHDL

Opis bloku ENTITY obejmuje:

- nazwę,
- nazwy końcówek (PORT),
- kierunek przepływu sygnału (IN, OUT, INOUT) (tzn. czy blok czyta wartość sygnału, czy też ustawia wartość sygnału, czy też może wykonać obie powyższe czynności),

- oznaczenie typu logicznego sygnału (nazwane w przykładzie BIT).

Przykład:

```
Entity CPU is
  Port(CLOCK : In Bit;
        ENAB  : In  Bit;
        RESET : In Bit;
        Gate  : In Bit);
End CPU;
```

Przykład 1. Składnia bloku ENTITY.

Należy zauważyć następujące ogólne cechy składni języka VHDL:

- Nazwy i słowa mogą być pisane zarówno małymi jak i dużymi literami, dla przykładu *RESET*, *reset* oraz *Reset* są traktowane jako ten sam sygnał. Zalecane jest przestrzeganie pewnego porządku, w którym stosuje się pewne reguły, zarówno dla zachowania zgodności z innymi narzędziami CAD jak również dla przejrzystości zapisu.
- Umieszczanie średników, nawiasów i dwukropków jest ważne dla składni – patrz przykład 1. Średniki są używane na końcach linii.
- 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 jest zazwyczaj wyrażeniem In, Out lub InOut dla sygnałów dwukierunkowych, a wyrażenie Typ_logiczny jest objaśnione poniżej. Należy zauważyć, że każda linia definiująca sygnały kończy się średnikiem z wyjątkiem ostatniej, gdzie na końcu jest nawias i średnik.

Sygnały można grupować po kilka w jednej linii, np.:

```
Clock, reset, enab : In Std_Ulogic;
```

Mimo, że jest to całkowicie zgodne z semantyką VHDL, poprzedni sposób definiowania sygnałów - po jednym w każdej linii - jest bardziej przejrzysty.

- Kolejność sygnałów w wyrażeniu PORT nie ma znaczenia dla VHDL, ale dla testów, kontroli i dokumentacji należy wybrać dogodną formę. Dobrym wyborem jest np. grupowanie alfabetyczne sygnałów danych typów.

6.1. Komentarze

Dla zapewnienia przejrzystości zapisu można dodawać komentarze. Są one zawsze poprzedzone dwoma myślnikami (--) i są aktywne aż do końca linii. Oznacza to, że mogą być dodane w tej samej linii co kod VHDL na jej końcu jak np.

```
RESET : In Std_ulogic; --Zerowanie główne o szerokości dwu
-- mikrosekund
```

```
-- to jest linia komentarza. Jeśli potrzebujesz więcej linii
-- komentarza każda musi być poprzedzona przez dwa myślniki.
```

Jeśli potrzeba więcej niż jednej linii dla wyrażenia VHDL kontynuuje się je w tylu liniach ile potrzeba. Znak kończący - zazwyczaj średnik - mówi kompilatorowi VHDL gdzie znajduje się koniec wyrażenia, niezależnie od tego ile linii ono zajmuje.

7. Sygnały

Sygnały były wcześniej tylko wspomniane bez dokładnego ich definiowania. Są one używane w VHDL w ten sam sposób jak w układach ASIC, płytach lub systemach. W typowym projekcie jest zazwyczaj wiele bloków ENTITY, każdy z nich definiuje część całego systemu. Sygnały definiowane przez wyrażenie PORT tworzą połączenia pomiędzy ENTITY, analogicznie jak końcówki (tzw. konektory, ang. *connectors*) łączące poszczególne płyty lub inne moduły. Wewnątrz bloku ENTITY są także używane dodatkowe sygnały zapewniające realizację odpowiedniej funkcji tego bloku, podobnie jak to się dzieje z płytami systemów cyfrowych. Te wewnętrzne sygnały płyt nie są dostępne na portach I/O.

W fizycznej realizacji systemu cyfrowego sygnały mogą być pojedynczymi bitami jak np. *reset* czy *clock*, jak również szynami o pewnej szerokości. Oba te typy mogą być łatwo opisane w VHDL. Język VHDL z natury nie definiuje charakterystyk sygnałów takich jak stan logiczny czy obciążalność. W zamian umożliwia wykonanie tego poprzez pliki VHDL pogrupowane w pakiety (ang. *packages*) zawarte w bibliotekach (ang. *libraries*), które są zazwyczaj dołączane jako część symulatora VHDL lub narzędzia syntezy, oraz mogą być rozwijane przez użytkownika dla specjalnych zastosowań.

Najprostszy sygnał został nazwany **BIT** i jest zdefiniowany tylko dla dwóch stanów logicznych 0 oraz 1. Następny typ logiczny został nazwany **MVL4** (*Multi-Valued Logic*). Zezwala on na posługiwanie się logiką trzystanową oraz ma stan 'nieznany' (ang. *unknown*) używany podczas symulacji (stany 0,1,Z,X). Typ MVL4 wystarczał do opisu podstawowych operacji. Nie wystarczał jednak do modelowania bardziej złożonych przypadków, jak np. logiki typu otwarty kolektor i stanu typu „nieistotny” (ang. *don't care*).

Dla zapewnienia większej uniwersalności, IEEE zaadoptował 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 (we wczesnych latach 90-tych). Znana jest ona jako biblioteka IEEE Standard_Logic_1164 i definiuje logikę 9-cio stanową o nazwie Std_Ulogic oraz system o zbliżonej nazwie Std_Logic. System ten jest znany także jako MVL9.

Używając tej biblioteki sygnały mogą przyjmować:

stany logiczne 0 i 1,

U dla stanu niezainicjowanego	- Uninitialized,
X dla stanu nieznanego	- Unknown,
Z dla trzystanowego	- Tri-state,
W dla niskoobciążalnego	- Weak strength,
H dla stanu wysokiej rezystancji	- High, dla wyjść typu otwarty kolektor
L dla stanu niskiej rezystancji	- LOW, dla wyjść typu otwarty emiter
- dla stanu nieistotnego	- Don't Care.

W przypadkach ogólnych te 9 stanów są wystarczające dla symulacji i opisu najczęściej występujących warunków logicznych. Mają one również dodatkowe możliwości, takie jak stan U dla sygnałów, dla których nigdy nie została przypisana wartość w czasie symulacji, oraz znak - reprezentujący warunki nieistotne.

Biblioteki są specyfikowane przez wyrażenie ogólne `library` oraz słowo `use`, za którym musi być podana specyficzna nazwa biblioteki. Zamieszczony przykład jest typowy dla specyfikacji typu używanej biblioteki dla większości symulatorów i narzędzi syntezy:

```
Library IEEE;  
Use ieee.std_logic_1164.all;
```

Biblioteka jest wygodnym mechanizmem dla przechowywania najczęściej używanych funkcji VHDL oraz definiowania typów danych. Biblioteka zazwyczaj składa się z dwóch części kodu VHDL. Pierwsza z nich nazywana PACKAGE definiuje nazwy oraz wejścia i wyjścia funkcji bibliotecznych. Druga nazywana PACKAGE BODY zawiera kod VHDL opisujący działanie funkcji. Sposoby posługiwania się własnymi bibliotekami będą przedstawione w następnych rozdziałach.

W plikach VHDL wyrażenie biblioteczne poprzedza sekcję ENTITY. Dla potrzeb syntezy, większość dostawców dostarcza dodatkową bibliotekę ze specjalnymi funkcjami operacji logicznych i arytmetycznych oraz dodatkowych typów logicznych. Format drugiej biblioteki jest podobny do pierwszej z przykładu, ale ma inną nazwę i także poprzedza wyrażenie ENTITY.

Sygnały są deklarowane poprzez podanie ich Nazwy oraz Typu logicznego, jak to poniżej przedstawiono:

```
SIGNAL CLOCK : Std_Ulogic;
```

Ten szczególny sygnał posiada nazwę *CLOCK* i jest typu *Std_Ulogic*. Jak to będzie przedstawione w dalszej części książki, nazwa sygnału oraz jego typ są deklarowane tylko raz i następnie mogą być używane w dalszej części kodu VHDL jako *CLOCK*.

W systemie IEEE MVL9 szyny, rejestry oraz wszystkie sygnały wielobitowe są wektorami (*vectors*) typu *Std_Ulogic_Vector*, gdzie bity mogą być definiowane na dwa następujące sposoby:

- gdy w danych binarnych najbardziej znaczący bit znajduje się na lewej pozycji słowa binarnego, wówczas używamy słowa „downto”:

```
Signal name : Std_Ulogic_Vector (M Downto N);
```

 np. dla szyny ośmiobitowej

```
Signal Databus : Std_Ulogic_Vector(7 downto 0);
```
- gdy w danych binarnych najbardziej znaczący bit znajduje się na prawej pozycji słowa binarnego, wówczas używamy słowa „to” gdzie $n < m$.:

```
Signal name : Std_Ulogic_Vector( n to m.);
```

 np. dla szyny 10-cio bitowej gdzie najbardziej znaczący bit leży po prawej stronie słowa:

```
Signal Inst_Reg : Std_Ulogic_Vector(0 to 9);
```

Poszczególne bity w szynie danych są identyfikowane przez ich numer bitu np.:

```
Databus(4); --dla bitu nr 4
```

Implikacją definiowania rejestru 8 bitowego jako *REG_A* (7 downto 0) lub *REG_B*(0 to 7) jest to, że jeśli binarną zawartością jest liczba (00000001), 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.

REG_A								REG_B							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

Użyta konwencja wyznacza sposób widzenia wartości poszczególnych bitów w całym słowie. W REG_A bit 5 ma wartość $2^5=32$ podczas gdy w rejestrze REG_B bit piąty ma wartość $2^2=4$. Nie ma nieścisłości w przypadku operacji przypisania wartości obu rejestrów takich jak REG_A<=Reg_B, wynik takiego działania byłby dokładnie taki jak opisany wcześniej dla przypadku z liczbą binarną (00000001).

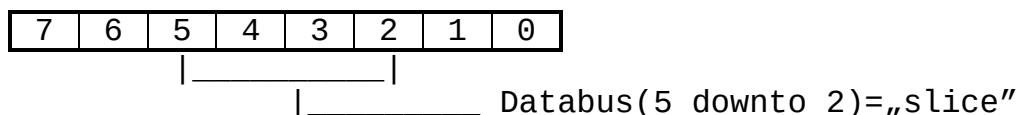
Chociaż większość aplikacji oraz arytmetyki binarnej używa formatu gdzie MSB (najbardziej znaczący bit) odpowiada numerowi bitu, w VHDL istnieje możliwość używania przypadku odwrotnego. Większość przykładów będzie prezentowana z zachowaniem MSB na pozycji najbardziej na lewo, jak w przypadku rejestru REG_A z użyciem słowa „downto”.

Bardzo często zachodzi potrzeba użycia tylko części szyny danych tzw. segmentu (ang. slice), np. w zastosowaniach do przetwarzania sygnałów. W takiej sytuacji segment szyny lub rejestru jest dostępny poprzez następującą składnię:

Databus(5 downto 2); - czterobitowy segment szyny danych

Signal *Databus* : Std_Ulogic_Vector(7 downto 0);

i nazywana jest „slice”.



Należy zauważyć, że użyta konwencja „downto” lub „to” w definicji szyny oraz w „slice” musi być taka sama.

Podsumujemy: dotąd opisano kilka typów sygnałów oraz sposób ich użycia. Mogą być także używane sygnały innego typu niż Std_Ulogic. Te i inne sposoby użycia będą omawiane na przykładach zamieszczonych w dalszych rozdziałach. Od tej chwili ogólne wprowadzenie dotyczące definiowania bloków ENTITY oraz sygnałów, jakie zostało dokonane, jest wystarczające do dalszych rozważań. Zanim jednak przejdziemy do przedstawienia sposobu opisu zachowania się bloku ENTITY, spójrzmy na typowy system cyfrowy czy układ ASIC oraz zastanówmy się jak zaplanować proces projektowania.

Jeśli podczas deklaracji sygnału nie zdefiniowano jego wartości początkowej przyjmuje się ją jako wartość najbardziej na lewo z danego zakresu typu sygnału.

8. Architektury jednostek (ARCHITECTURE)

Wyrażenie ARCHITECTURE jest w języku VHDL używane do definiowania jak powinny zachowywać się lub z czego powinny się składać bloki Entity. Opis Architektury w języku VHDL może być dokonany na poziomie behawioralnym, na poziomie przesłań międzyrejestrowych RTL lub na poziomie strukturalnym, chociaż, z powodu możliwości syntezy, poziom behawioralny RTL jest najczęściej stosowany.

W każdym procesie symulacji lub syntezy występuje jeden opis ARCHITECTURE powiązany z każdym blokiem Entity. Wspominamy o tym w tym miejscu, ponieważ w projektowaniu hierarchicznym ‘top down’, może pojawić się w danym momencie tj. w aktualnym katalogu plików, kilka dostępnych dla danego Entity Architektur. Język

VHDL posiada mechanizm zapewniający możliwość wyboru jednej z nich poprzez wybór CONFIGURATION.

Architektura ma zawsze określony 'typ' oraz związany z nią blok Entity, jak to przedstawiono poniżej:

```
ARCHITECTURE RTL of CPU is
```

```
    ... wiele innych linii
```

```
End RTL;
```

Typ architektury jest w powyższym przypadku RTL i może być dowolną nazwą nie kolidującą z zarezerwowanymi dla kompilatora. Zaleca się, aby nazwa architektury wskazywała na działanie komórki chociaż tak być nie musi.

Innymi nazwami architektur, które występują w VHDL są np.: BEHAVIORAL, STRUCTURAL, SCHEMATIC, TEST_BENCH. Nazwy te wyraźnie wskazują na typ projektu VHDL, który jest przedmiotem modelowania. Duże litery w nazwach zostały użyte jedynie dla przejrzystości i uwydatnienia.

Przypomnijmy, że układ ASIC 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. Ponadto, dla każdej ARCHITECTURE definiuje się inne sygnały dostępne tylko wewnątrz niej a które będą przedmiotem syntezy jako rejestry, liczniki i bramki wchodzące w skład ARCHITEKTURY.

Przykłady deklarowania sygnałów, nazwanych tu CLOCK, LOAD, 16-to bitowego rejestru instrukcji oraz 8-mio bitowego licznika, są następujące:

```
Architecture RTL of CPU is
```

```
Signal CLOCK      : Std_Ulogic;  
Signal Load       : Std_Ulogic;  
Signal Instr_Reg   : Std_Ulogic(15 downto 0);  
Signal Counter     : Std_Ulogic(7 downto 0);
```

Powyższy zapis jest Deklaracją Sygnałów (*Signal Declarations*). Sygnały są zawsze na początku bloku ARCHITECTURE w tzw. sekcji deklaracji.

Dobłą praktyką jest dodawanie komentarzy, co służy celom dokumentacyjnym i ułatwia późniejszą analizę:

```
Signal Clock : Std_Ulogic; --10MHz system clock
```

Zasadą ogólną syntezy jest to, że każdemu elementowi pamięciowemu sterowanemu sygnałem zegara musi być nadana nazwa i przypisany typ logiczny. Dobrą praktyką jest używanie tych samych nazw, jakie zostały użyte na schemacie blokowym - ułatwia to przeglądanie oraz rozumienie projektu. Pokażemy w dalszym ciągu, że dla układów kombinacyjnych nie zawsze jest konieczne nazywanie sygnałów, jeśli są one używane tylko jako wejścia elementów zegarowanych.

Podobnie jak sygnały można definiować stałe **Constants**:

```
Constant ALL_ONES : Std_Ulogic_Vector:="11111";
```

Deklaracje Stałych są także umieszczane w sekcji deklaracji na początku ARCHITECTURE, a dobrą pozycją jest miejsce zaraz po **Deklaracji Sygnałów**. Należy zauważyć, że podobnie jak w przypadku Deklaracji Sygnałów, Deklaracja Stałych zawiera typ logiczny oraz dodatkowo symbol przypisania wartości „:=”. Wartości binarne wektorów są zawsze podawane w podwójnym cudzysłowie górnym i są interpretowane jako łańcuchy bitów. Należy zauważyć, że w przeciwieństwie do deklaracji sygnałów nie ma tu konieczności deklarowania długości wektora tak jak w składni „4 downto 0”.

Po deklaracjach sygnałów i stałych może wystąpić kilka innych linii które zostaną objaśnione później.

Jest zasadą, że słowo **BEGIN** rozpoczyna opis funkcjonalny ARCHITECTURE i wskazuje miejsce, w którym rozpoczyna się opis jej funkcji. Przykładowa składnia całego kodu VHDL prostego układu CPU i jego architektury może wyglądać następująco:

```
-- Biblioteki
Library IEEE;
  Use ieee.std_logic_1164.all;

Library Synthesis; -- Ta nazwa zależy od narzędzia syntezy
  Use synthesis.synth.all

-- Blok ENTITY
Entity CPU is
  Port ( CLK33M :In Std_Ulogic;
        ENAB    :In Std_Ulogic;
        RESET   :In Std_Ulogic;
        Datain  :In Std_Ulogic_Vector(15 downto 0);
        Dataout  :Out Std_Ulogic_Vector(15 downto 0);
        Gate    :Out Std_Ulogic);
End CPU;

-- Architektura
Architecture RTL of CPU is

  Signal CLOCK      :Std_Ulogic;
  Signal LOAD       :Std_Ulogic;
  Signal Instr_Reg  :Std_Ulogic_Vector(15 downto 0);
  Signal Counter    :Std_Ulogic_Vector(5 downto 0);
  Constant ALL_ones :Std_Ulogic_Vector:="111111";
Begin

-- Reszta opisu funkcjonalnego

End RTL;
```

„Reszta opisu funkcjonalnego” stanowi część kodu VHDL, który definiuje jak powyższy blok układu cyfrowego zachowuje się w odpowiedzi na sygnały wejściowe.

Zanim przejdziemy dalej musimy wyjaśnić dwa zagadnienia. Otóż w ogólności, każdy system cyfrowy można opisać przez stany sygnałów wyjściowych uzależnione od wejść oraz przez czas, w ciągu którego wyjścia uzyskują te stany. Dlatego istotne jest zrozumienie, w jaki sposób język VHDL operuje zdarzeniami w czasie.

9. Operacja przypisania wartości sygnału (<=)

Po zadeklarowaniu portów sygnałowych I/O bloku ENTITY oraz wewnętrznych sygnałów wewnątrz ARCHITECTURE, następnym krokiem jest prześledzenie metody ustawiania pożądaných wartości tych stanów. Terminem oznaczającym ustawianie stanów w VHDL jest „**Signal Assignment**”, a jego składnia jest następująca:

```
SIGNAL<=VALUE;
```

gdzie VALUE może być stałą, sygnałem lub wynikiem działań logicznych lub arytmetycznych na stałych i sygnałach. Symbol „<=” wyraża przeniesienie stanu VALUE na SIGNAL. Jako prosty przykład można podać:

```
Counter <= ALL_ONES;
```

co oznacza „Ustaw stan licznika na ALL_ONES” tj. na binarną wartość „111111”, jak to zostało wcześniej zadeklarowane w części deklaracji stałych. Ten sam wynik można także otrzymać przez:

```
Counter <="111111";
```

Symbol „<=” **jest używany w odniesieniu do sygnałów** i oznacza przeniesienie wartości z prawej strony symbolu na lewą. W powyższym przypadku sygnał Counter został wcześniej zadeklarowany jako 6-cio bitowy (Std_Ulogic_Vector (5 downto 0)). Wartość po prawej stronie musi być tego samego typu i długości, w przeciwnym wypadku kompilator VHDL zasygnalizuje błąd. Jak widać stała ALL_ONES jest zgodna co do długości, wartość „111111” ma także 6 bitów.

Przypisanie wartości może być także wykonane na pojedynczych sygnałach lub bitach np.:

```
Clock <= CLK33M;  
lub  
Load <= '1';  
lub  
Load <= Enab AND Instr_Reg(3);
```

Należy zauważyć, że wartości pojedynczych bitów są umieszczane w pojedynczym cudzysłowie, podczas gdy wielobitowe (szyn i wektorów) w podwójnym cudzysłowie („character literal”). Wyrażenie po prawej stronie może być bardzo złożone w porównaniu z powyższymi przykładami - może ono zawierać operacje matematyczne i logiczne z użyciem wielu sygnałów. Będzie o tym mowa w dalszych rozdziałach.

Drugim aspektem zależności sygnałów od czasu jest ich opóźnienie, czyli potrzeba określenia czasu po którym przyjmują one zadane stany. Cechą VHDL jest to, że pozwala on na dokładne określenie operacji logicznych, ma wbudowane operacje z użyciem

parametrów takich jak czas i opóźnienie czasowe oraz posiada możliwość obsługi operacji równoczesnych.

9.1. Operacje współbieżne oraz czasowe

Rozważmy typowy projekt cyfrowy zawierający różne rejestry, liczniki, multipleksery oraz układy sterujące, wszystkie sterowane synchronicznie sygnałem zegarowym. System cyfrowy może składać się z niezależnych, oddzielnych funkcji albo może realizować funkcje, które są od siebie zależne, ale cały ten system działa równocześnie w tym samym czasie, lub innymi słowy - współbieżnie. Każde z wyjść bramek lub przerzutników zależy wyłącznie od stanu ich wejść, pełnionej funkcji oraz chwili czasu, w której obserwujemy wyjścia. Dlatego system opisu/symulacji sprzętu musi być zdolny do wykonywania operacji równoczesnych oraz obsługi ich opóźnień na wyjściach.

Wbudowane własności obsługi zdarzeń czasowych i współbieżności są dwiema głównymi różnicami pomiędzy VHDL oraz standardowymi językami programowania.

W standardowym języku programowania takim jak PASCAL, C lub Fortran wszystkie operacje wykonywane są sekwencyjnie krok po kroku. Ponieważ te języki nie zostały rozwinięte dla modelowania rzeczywistych procesów czasowych, nie są one przydatne do symulacji i projektowania systemów cyfrowych.

Operacje czasowe i współbieżne są integralną częścią składową języka VHDL i dlatego nadaje się on do modelowania rzeczywistego zachowania się systemów cyfrowych. Własności VHDL upodabniają go do pracy rzeczywistych systemów cyfrowych.

Wróćmy do przedstawionego wcześniej sposobu przyporządkowania sygnałów:

```
Load <= Enab AND Instr_Reg(3);
```

gdzie sygnał Load jest wyjściem iloczynu logicznego sygnału Enab oraz linii nr 3 z Instr_Reg. Opóźnienie sygnału Load o 5 nanosekund odbywa się zgodnie z następującą składnią:

```
Load <= Enab AND Instr_Reg(3) after 5 nsec;
```

Jest to zupełnie intuicyjna składnia, która zostanie bardziej szczegółowo omówiona w dalszej części.

Ale jeśli powyższy kod wprowadza 5nsek opóźnień, to co stanie się jeśli takowego nie wprowadzimy? Oczywiście w praktyce nie ma zerowych czasów opóźnień. W VHDL wprowadzona jest technika nieskończenie małych przyrostów czasowych, w których następują zmiany stanów logicznych. Taka technika umożliwia symulatorowi VHDL wyznaczenie stanów logicznych w systemie w danym punkcie czasowym T, obliczenie zmian stanów logicznych oraz wprowadzenie ich w momencie T + delta T. Rzeczywiste nadawanie stanów logicznych jest wykonywane tylko wtedy, gdy nastąpiła zmiana sygnału na przynajmniej jednym z wejść do systemu lub kiedy kończy się proces opóźnieniowy jak np. w powyższym przypadku po 5 nanosekundach.

W warunkach biernych lub w stanie ustalonym systemu, w każdej chwili czasu wszystkie z wejść i wyjść bramek oraz elementów pamięciowych są w konkretnych stanach logicznych, które są nazywane aktualnym stanem systemu (ang. Present State). Symulator VHDL oczekuje na mające się pojawić zmiany stanów wejściowych. Jeśli się one pojawią w chwili T0, wówczas symulator oblicza czy mają nastąpić zmiany wyjść bramek i rejestrów i czeka z wprowadzeniem ewentualnych zmian zgodnie z ich opóźnieniami. Wykonywane jest to poprzez przeglądanie całej architektury od słowa

BEGIN do słowa END, poprzez badanie poszczególnych stanów w chwili T0. W rezultacie wyznaczony zostanie nowy stany logiczny systemu, który zostanie wprowadzony w momencie T0 + delta T. Delta T, nazywane także „time tick”, jest małym przyrostem czasu, w którym następuje ustawienie stanów logicznych, jeśli nie mają one podanych szczegółowych opóźnień czasowych. Natomiast w przypadku sygnałów, które mają podane czasy opóźnień, stany logiczne poszczególnych wyjść są obliczane po upływie tych opóźnień. Po przejściu „tick time” uaktualniany jest stan systemu, a proces analizy systemu znowu oczekuje na następne zmiany sygnałów, które mogą się pojawić.

Należy zauważyć, że powyższy mechanizm jest bardzo ważnym składnikiem VHDL, ponieważ daje to równoczesne zmiany wszystkich sygnałów niezależnie od pozycji na jakiej są one umieszczone w kodzie VHDL.

Sposób w jaki VHDL oszacowuje wartości sygnałów powinien być znany projektantom układów logicznych, gdyż sposób ten jest analogiczny do tego w jaki zmieniane są stany elementów pamięciowych, a te są ustawiane w momencie pojawienia się sygnału zegara. Wszystkie wejścia do przerzutników, liczników, rejestrów są wyznaczane oraz przenoszone na wyjścia zaraz po sygnale zegara. Gdy pojawi się zbocze zegarowe oraz minie czas propagacji zmiany pojawiają się na wyjściach systemu. Kilka dodatkowych przykładów, w jaki sposób sygnały są obsługiwane, zostanie podanych poniżej w części ‘Procesy’.

Wszystkie operacje umieszczone wewnątrz bloku ARCHITECTURE pomiędzy słowami kluczowymi BEGIN oraz END są wykonywane współbieżnie (jednocześnie) niezależnie od pozycji ich umieszczenia w kodzie VHDL. Dodatkowo możliwy jest opis operacji sekwencyjnych, jednakże te muszą być umieszczone wewnątrz procesu, procedury lub funkcji. Wywołanie procesu, funkcji lub procedury jest wykonywane współbieżnie natomiast operacje wykonywane w ich wnętrzach są przetwarzane sekwencyjnie.

10. Operatory

Poniższa tabela przedstawia listę dostępnych operatorów z podziałem na ich typy.

Tabela 2. Operatory dostępne w VHDL.

Typ	Symbol	Operacja
Arytmetyczne	+	Dodawanie
	-	Odejmowanie
Znak	-	Znak minus
Logiczne	And	Mnożenie logiczne
	Or	Suma logiczna
	Nand	Logiczne NAND
	Nor	Logiczne NOR
	Xor	Logiczne Exclusive-or
	Not	Zaprzeczenie
Relacyjne	=	Równe
	/=	Nierówne
	<	Mniejsze niż
	>	Większe niż
	<=	Mniejsze niż lub równe
	>=	Większe niż lub równe
Konkatenacja	&	Łączenie danych,

		sklejanie
--	--	-----------

11. Obiekty

Obiekty w języku VHDL są to elementy posiadające własność posiadania wartości. Są 4 klasy obiektów:

- stałe (*constant*),
- sygnały (*signal*),
- zmienne (*variable*),
- pliki.

Każdy z obiektów musi mieć przyporządkowany typ oraz może mieć nadaną wartość początkową. Dwie pierwsze klasy obiektów zostały omówione wstępnie w poprzednich rozdziałach.

Zmienne (*variable*) są podobne w działaniu i zastosowaniu do zmiennych w językach wysokiego poziomu takich jak PASCAL czy C. Do tej pory skupiliśmy się na rozważaniach dotyczących operacji na sygnałach, należy mieć jednak na uwadze, że VHDL ma również możliwość deklaracji zmiennych, przy czym trzy zasadnicze cechy odróżniają zmienne od sygnałów:

- a. Zmienne przyjmują wartości w momencie ich przypisania, dokładnie tak jak zmienne w PASCALu lub innym standardowym języku programowania. To oznacza, że inaczej niż w przypadku sygnałów, kolejność przypisywania zmiennych ma istotne znaczenie dla ich funkcjonowania.
- b. Zmienne mogą być używane tylko wewnątrz procesów, funkcji i procedur i występują tylko lokalnie w tym bloku. Wartość zmiennej nie może przejść poza proces i nie może być bezpośrednio użyta przez inny proces lub operację równoczesną.
- c. Do przypisania wartości zmiennej używa się symbolu ":", natomiast dla sygnału "<=".

W projektowaniu z użyciem narzędzi syntezy nie ma powodu, dla którego miałyby być stosowane wyłącznie sygnały, za wyjątkiem używania zmiennych w funkcjach i procedurach VHDL (co będzie omawiane w dalszym ciągu). Aby sprawa była jasna do końca - stosowanie zmiennych zostanie przedstawione na konkretnych przykładach, ale w ogólnym przypadku należy postępować ostrożnie przy ich używaniu, szczególnie w dużych procesach. Ponieważ kolejność występowania zmiennych jest istotna dla działania układu, wprowadzanie zmian wymaga kontroli całego mechanizmu ustawiania kolejnych ich wartości. Jest to nieco podobne do projektowania logiki sterowanej dwoma zboczami sygnału zegara - jest to możliwe, ale zazwyczaj jest to trudne do konserwacji i modyfikowania. Z tego powodu w tym opracowaniu stosowanie sygnałów będzie uprzywilejowane w stosunku do zmiennych, chociaż zmienne mogą być efektywnie użyte do testów, w funkcjach i procedurach.

Piki są obiektami w postaci zbiorów dyskowych w których mogą być przechowywane wartości określonego typu. Nie będą one wykorzystywane w dalszej części opracowania.

12. Typy, deklarowanie własnych typów (TYPE)

Typ w VHDL stanowi zbiór wartości posiadających własną nazwę i często posiadających wspólne cechy. Wszystkie obiekty muszą mieć określony typ przez co przechodzą na nie podstawowe właściwości związane z danym typem. Oprócz typów zdefiniowanych w pakiecie STANDARD (patrz rozdział 15) można dodawać własne. Deklaracja typu może występować w 2 formach tzw. niekompletna (gdzie zdefiniowany

jest wyłącznie identyfikator typu) oraz pełna zawierająca identyfikator typu oraz zbiór przypisanych wartości. Typy można podzielić na skalarne, złożone, dostępu oraz plikowe. Poniżej przedstawione są przykłady deklaracji niektórych typów.

12.1. Deklaracja niekompletna typu

Deklaracja niekompletna typu ma postać:

```
TYPE identyfikator_typu;
```

12.2. Typy skalarne

Do typów skalarnych zaliczamy typ wyliczeniowy, całkowity, rzeczywisty, fizyczny:

- wyliczeniowy

```
type BIT is ('0' , '1');  
type KOLORY_PODS is (ZIELONY , CZERWONY, NIEBIESKI);
```

Każdy z elementów typu wyliczeniowego ma przyporządkowany identyfikator w postaci liczby całkowitej przy czym pierwszy z nich ma identyfikator równy 0 a każdy następny kolejno 1,2...

- całkowity jest zestawem liczb całkowitych ograniczonych do pewnego zakresu

```
type LICZBA is range 1 to 31;  
type CYFRA is range 31 downto -23;
```

- rzeczywisty jest typem REAL ograniczonym do pewnego zakresu

```
type LICZBA_r is range -1.0 to 31.0;  
type CYFRA_r is range 31.5 downto -23.2;
```

- fizyczny, z dodaną jednostką podstawową oraz jej mnożnikami

```
type NAPIECIE is range 0 to 3.2  
units
```

```
V;
```

```
kV=1000V;
```

```
V=1000mV;
```

```
mV=1000uV;
```

```
end units NAPIECIE;
```

12.3. Typy złożone

Typy złożone to:

- typ tablicowy reprezentuje grupę elementów tego samego typu. Każdy element ma przyporządkowany indeks (jedno lub wielowymiarowy). Dostęp do elementu tablicy następuje poprzez indeks elementu. Są dwa rodzaje deklaracji tablicy: ograniczona i nieograniczona. Przykłady:

```
type TAB_A is array (0 to 7) of integer;
```

tablica 8-miu elementów typu całkowitego,

```
type TAB_B is array (0 to 15) of Bit_Vector(7 downto 0);
```

tablica 16-tu elementów typu Bit_Vector,

```
type TAB_C is array (positive range <>) of integer;
```

tablica nieokreślonej liczby elementów typu całkowitego,

```
type T_Dual is array(0 to 15,BIT) of Bit_Vector(7 downto 0);
```

tablica 2-u wymiarowa 16-tu elementów typu Bit_Vector,

- typ rekordowy – nie będzie omawiany,

12.4. Podtypy

Podtypy są podzbiorem wartości danego typu. Poniżej przedstawione są przykłady:

```
type T_A is Bit_Vector(7 downto 0);
subtype T_B is T_A(6 downto 1);
```

13. Atrybuty

Atrybut jest dodatkową informacją związaną z elementami języka. W zależności od typu elementu języka rozróżnia się następujące atrybuty: typów, sygnałów, nazw. Atrybuty są wywoływane z użyciem składni *nazwa_elementu'nazwa_atrybutu*. Poniżej przedstawione są wybrane predefiniowane atrybuty VHDL.

A) Atrybuty typów prostych, nazwa typu jest podana jako ABC .

base

ABC'base – typ bazowy dla ABC przykład:

```
type TABLICA is array ( 0 to 7) of Integer;
subtype TABLICZKA is TABLICA (1 to 3);
```

atrybut TABLICZKA'base da w wyniku wartość TABLICA

ABC'left – wartość najbardziej na lewo typu ABC,

ABC'right – wartość najbardziej na prawo typu ABC,

ABC'high – wartość największa indeksu typu ABC,

ABC'low – wartość najmniejsza indeksu typu ABC,

ABC'ascending – wartość TRUE jeśli typ ABC ma charakter rosnący lub FALSE w przeciwnym wypadku.

B) Atrybuty typów i obiektów tablicowych, nazwa typu jest podana jako ABC .

ABC'left(n) – wartość najbardziej na lewo typu ABC dla indeksu z zakresu *n*,

ABC'right(n) – wartość najbardziej na prawo typu ABC dla indeksu z zakresu *n*,

ABC'high(n) – wartość największa typu ABC dla indeksu z zakresu *n*,

ABC'low(n) – wartość najmniejsza typu ABC dla indeksu z zakresu *n*,

C) Atrybuty sygnałów, SIG oznacza nazwę sygnału.

SIG'event – wynik TRUE jeśli wystąpiło zdarzenie na sygnale SIG w czasie obecnym symulacji, FALSE w przeciwnym przypadku,

SIG'stable(t) – wynikiem jest nowy sygnał o wartości TRUE jeśli nie wystąpiło zdarzenie przez okres czasu *t*, FALSE w przeciwnym przypadku,

SIG'delayed(t) – wynikiem jest nowy sygnał opóźniony o *t* względem SIG,

14. 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: proces, funkcja i procedura, które to grupują wewnątrz swego ciała szereg instrukcji wykonywanych sekwencyjnie. Same instrukcje procesu, procedury oraz funkcji wywoływane są równocześnie z innymi instrukcjami umieszczonymi w ARCHITEKTURZE, ich wnętrza natomiast przetwarzane są sekwencyjnie. Poniżej podany jest podział instrukcji na współbieżne i sekwencyjne, przy czym 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.

A) Instrukcje współbieżne.

- przypisania wartości sygnału,
- kontroli,
- procesu,
- procedury,
- funkcji,
- blokowa,
- wstawiania składnika (konkretyzacji),
- powielania.

B) Instrukcje sekwencyjne.

- oczekiwania,
- przypisania wartości sygnału,
- przypisania wartości zmiennej,
- kontroli,
- raportu,
- wywołania procedury,
- instrukcje warunkowe (IF, CASE),
- instrukcje pętli,
- instrukcja wyjścia,
- instrukcja powrotu,
- instrukcja pusta.

15. Organizacja bibliotek, biblioteka STD, zakres dostępności bibliotek

Każdy blok ENTITY, CONFIGURATION, PACKAGE jest analizowany (kompilowany) i umieszczany w bibliotece bieżącego projektu o nazwie WORK. W ogólności biblioteki są implementowane jako katalogi dyskowe i dostępne przez ich logiczną nazwę. W środowisku VHDL 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 zadeklarować 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.

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 oraz elementy danego pakietu mają być widoczne w bieżącym projekcie. Wykonuje się to za pomocą wyrażenia use tak jak w poniższym przykładzie:

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_unsigned.all;
      ^      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^      ^
```

		nazwa pakietu	
nazwa biblioteki		wyrażenie "all" oznaczające wszystkie elementy pakietu	
		lub nazwa konkretnego składnika pakietu	

Widzialność bibliotek i pakietów jest przenoszona wyłącznie do najbliższego bloku ENTITY/PACKAGE/CONFIGURATION występującego bezpośrednio poniżej deklaracji library/use.

Zawartość pakietu STANDARD (przykłady):

- zadeklarowane TYPY
- BOOLEAN wartości logiczne FALSE i TRUE,
- BIT wartości '0' i '1',
- BIT_VECTOR tablica elementów typu BIT,
- CHARACTER znaki kodu ASCII,
- STRING typ tablicowy ze znaków typu CHARACTER,
- INTEGER liczby całkowite w zakresie zależnym od implementacji,
- NATURAL liczby całkowite większe lub równe zero,
- POSITIVE liczby całkowite większe od zera,
- REAL liczby z punktem dziesiętnym w zakresie zależnym od implementacji,
- TIME typ definiujący czas,

16. Biblioteka 1164, typ MVL9, przykłady zrealizowanych funkcji

W roku 1987 organizacja IEEE opublikowała bibliotekę o nr 1164 zawierającą min typy STD_LOGIC i STD_ULOGIC (nazywane również MVL9) oraz ich reprezentacje wektorowe STD_LOGIC_VECTOR oraz STD_ULOGIC_VECTOR. Typy te zawierają 9 wartości i mają na celu zwiększenie uniwersalności w projektowaniu i syntezie układów cyfrowych. Typy te są zadeklarowane następująco:

type STD_LOGIC is ('U','0','1','X','Z','W','H','L','-');

Poszczególne wartości mają następującą interpretację: (&&&)

- 'U' – wartość niezainicjowana,
- '0' – zero logiczne,
- '1' – 1 logiczne,
- 'X' – stan nieznan
- 'Z' – stan wysokiej impedancji,
- 'W' – stan słabego wysterowania,
- 'H' – typu otwarty kolektor,
- 'L' – typu otwarty emiter,
- '-' – stan nieistotny.

Dodanie kilku możliwych wartości dostępnych dla obiektów typu STD_LOGIC znacznie ułatwiło opis VHDL systemów cyfrowych zawierających szyny danych / adresów oraz funkcje iloczynów i sum logicznych „na drutach”. Dla typu STD_LOGIC dodatkowo zdefiniowana funkcja arbitrażowa natomiast dla typu STD_ULOGIC nie ma takiej definicji.

Poza tym biblioteka zawiera funkcję dodawania wektorów typu STD_LOGIC (i wersji ULOGIC), brak jest ich odpowiednika w bibliotece STD dla typu BIT_VECTOR.

17. Procesy (PROCESS), zegarowane, zatraskowe, kombinacyjne

17.1. Procesy zegarowane

Procesy zegarowane są używane do syntezy układów z przerzutnikami, rejestrami, zatraskami lub dowolnymi innymi elementami zegarowanymi. Istnieje także część kombinacyjna w takim procesie, jednakże będzie ona wewnętrzna w stosunku do procesu przetwarzania układu - jako wejścia do przerzutników. Głównym punktem jest to, że procesy zegarowane służą do syntezy układów zarówno kombinacyjnych jak i sekwencyjnych z tą różnicą, że jak zaprezentowano wcześniej, każdy sygnał, któremu zostanie przypisana wartość, zostanie potraktowany jako element pamięciowy. Zagadnienie to zostanie rozwinięte w dalszej części na przykładach.

W procesach zegarowanych lista czułości „Sensitivity List” zawiera zawsze sygnał zegara i zazwyczaj asynchroniczny sygnał resetu (jest to reguła projektowania układów ASIC, a nie VHDL). W procesie może wystąpić tylko jeden sygnał zegarowy oraz jeden sygnał zerowania. Jeśli jest więcej niż jeden reset, to powinny być one zsumowane logicznie (OR) poza procesem.

Rozważmy najprostszy przypadek, kiedy nie ma sygnału zerowania (reset) oraz założmy, że nasza biblioteka zawiera przerzutniki wyzwalane zboczem narastającym. Proces będzie oczekiwał na narastające zbocze sygnału i wtedy ładował zawartość szyny danych Databus do rejestru REG, o których założmy, że były wcześniej zdefiniowane jako wektory ośmiobitowe. Proces będzie wyglądał następująco:

```
Loadreg : Process (Clock)
Begin
  If Rising_Edge(Clock) then
    REG<=Databus;
  End if;
End Process Loadreg;
```

Użycie wyrażenia If-then, pokazanego powyżej, jest intuicyjne i oznacza: jeśli pojawi się narastające zbocze, wówczas przepis zawartość szyny Databus do rejestru REG, w przeciwnym wypadku nic nie rób. Taki kod zostanie zsyntezowany jako prosty rejestr złożony z przerzutników typu D bez wejść zerujących.

Przypomnijmy, że proces jest aktywowany kiedy tylko następuje zmiana w jakimś sygnale umieszczonym na liście „Sensitivity List”. W przypadku naszego zegara (Clock), jedyne zmiany zachodzą w czasie narastającego (rising, leading edge) lub opadającego (falling, trailing edge) zbocza. W przypadku kodu przedstawionego powyżej, testowane jest tylko zbocze narastające, dając w rezultacie zmianę stanów. Podczas zbocza opadającego warunek IF Rising_Edge(CLOCK) nie jest spełniony, a proces zostaje zakończony lub raczej zawieszony do następnej zmiany sygnałów z listy „Sensitivity List”.

W powyższym procesie występuje następujące przypisanie sygnałów:

```
Reg<=Databus;
```

Program syntezy logicznej automatycznie wygeneruje rejestr, w którym liczba jego bitów została wcześniej podana w definicji sygnałów, np.

```
SIGNAL REG : Std_Ulogic_Vectr(7 downto 0);
```

Jeśli zostanie zwiększona wielkość REG do 23 bitów wówczas otrzymamy rejestr 23-bitowy! Ta pojedyncza linia kodu może być użyta do generacji rejestru niezależnie od tego czy ma on jeden czy też N bitów, bez konieczności dodatkowych prac. 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. Proces będzie wymagał umieszczenia dwóch sygnałów na liście czułości „Sensitivity List”. 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:

```
Loadreg : Process (Clock, Reset)
Begin
  If Reset='1' then
    REG <= "00000000"; --wyczyść rejestr REG
  ElsIf Rising_Edge(Clock) then
    REG <= Databus;
  End if;
End Process Loadreg;
```

Wyrażenie ELSIF, będące skrótem od ELSE IF, jest aktywne tylko wtedy, jeśli pierwszy warunek wyrażenia IF jest fałszem tj. jeśli Reset ma stan niski. Należy zauważyć, że priorytet jest ustalany poprzez wyrażenie IF - ELSIF. W rzeczywistości część ELSIF oznacza „tylko wtedy gdy nie ma sygnału Reset (tj. tylko wtedy gdy Reset='0') sprawdź akcję uruchamianą z boczem zegarowym”. Innymi słowami mówiąc, jeśli oba warunki pojawią się jednocześnie priorytet ma Reset. Można także zauważyć, że w odróżnieniu od bezpośredniego resetu akcja następuje jedynie w momentach pojawienia się zboczy sygnałów, co jest powodem, tego, że wszystkie sygnały, których wartość zostaje przypisana wewnątrz procesów zegarowanych, stają się przerzutnikami lub rejestrami w syntezyowanym układzie. Jeśli zachodzi potrzeba kombinacyjnego dekodowania niektórych stanów rejestrów lub liczników, to muszą być dodane odpowiednie wyrażenia poza procesem.

Prezentowana tu składnia VHDL jest standardowym formatem dla logiki zegarowanej z bezpośrednim resetem:

- Zarówno sygnał zegara jak i resetu są umieszczone na liście „Sensitivity List”, na której może być umieszczony tylko jeden zegar i jeden reset w procesie.
- Za wyrażeniem IF Reset ... następuje opis akcji, która powinna nastąpić gdy pojawi się reset.
- Wyrażenie Elsif Rising_Edge ... służy do obsługi zdarzeń związanych z sygnałem zegara.

Przykłady projektów w dalszej części książki pokażą, że operacje zegarowane mogą być bardziej wydajne niż proste ładowanie rejestrów. Jednak podstawowa struktura procesu zegarowanego pozostaje bez zmian. Decyzja o tym, jak dużo logiki należy umieścić wewnątrz procesu zegarowanego oraz ile procesów należy utworzyć zostanie omówiona później.

Słowo ostrzeżenia. W pojedynczym projekcie można mieć pewne części, które posiadają reset asynchroniczny, a inne nie posiadają. Większość dostawców układów ASIC woli umieszczać reset w każdym przerzutniku, dla polepszenia testowalności oraz możliwości inicjalizacji zarówno podczas symulacji jak i podczas testów. Na przykład, można mieć rejestry kolejkowane (pipelined data registers), które zostaną ustawione do znanego stanu wcześniej podczas symulacji lub testów. Jeśli projektant decyduje się na nie wykonywanie resetu dla tej części logiki, to należy być ostrożnym i umieścić te części w oddzielnych procesach nie zawierających resetów. Powodem jest to, że podczas symulacji, jeśli reset jest uaktywniony logika zegarowana (`Elsif Rising_Edge ...`) nie będzie przetwarzana i żaden z przerzutników czy rejestrów tego procesu nie zmieni stanu. Jednakże w rzeczywistym ASICu, jeśli generator zegarowy nadal działa, to te przerzutniki, które nie są resetowane, są zegarowane i mogą zmienić stan. **Ten fakt może spowodować inne działanie fizycznie wykonanego układu ASIC niż jego wcześniejsze symulacje i może powodować niekiedy poważne problemy.** Takiej sytuacji można uniknąć poprzez separację tej części układu, która nie zawiera resetów w oddzielne procesy lub poprzez użycie asynchronicznego resetu na całą część logiki zegarowanej.

17.2. Procesy niezegarowane (kombinacyjne)

Logika kombinacyjna może zostać opisana poza procesem lub poprzez proces niezegarowany. Podstawową różnicą jest składnia językowa opisu obu tych procesów. Proces niezegarowany ma podobną składnię do zegarowanego.

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. Tak jak w procesie chcemy mieć A, B i SEL na liście „Sensitivity List”, tak i tu jakakolwiek zmiana któregoś z sygnałów wejściowych powoduje aktywację procesu i obliczenie stanu wyjściowego Y.

```
MUX2_2 : Proces(A, B, SEL) -- multiplekser 2 na 1
Begin
  If SEL='0' then Y<=A;
  Else Y<=B;
  End If;
End Process MUX2_1;
```

Ważną obserwacją jest to, że nie ma tu żadnej operacji typu `Rising_Edge` lub `Falling_Edge`, tak jak to było w procesach zegarowanych. W rezultacie taki proces będzie prowadził do logiki kombinacyjnej.

Zwraca uwagę to, że 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 jednakowy; różnią się jedynie deklaracje długości wektorów sygnałów.

Powyższy przykład wykorzystuje inną składnię wyrażenia IF, a mianowicie IF-THEN-ELSE, która może zostać uogólniona do postaci:

```
IF warunek_jest_prawdą THEN
    akcja
ELSE inna_akcja
END IF;
```

Akcja po wyrażeniu ELSE jest wykonywana tylko wtedy gdy pierwszy warunek jest fałszem. Składnia IF-THEN-ELSE umożliwia uniknięcie konieczności stosowania oddzielnych wyrażeń IF dla testowania warunków prawdy i fałszu tak jak:

```
IF warunek_jest_prawdą THEN
    akcja_1
END IF;
IF warunek_jest_fałszem THEN
    akcja_2
END IF;
```

Wyrażenie IF jest używane szeroko podczas modelowania systemów i będzie ilustrowane w trakcie książki. Każde IF wymaga wyrażenia END IF. Składnia IF może być używana tylko wewnątrz procesów (lub też funkcji i procedur). Inną składnię stosuje się do wprowadzania warunków poza procesem, co zostanie omówione później.

17.3. Procesy zatraskiwane

Trzecia odmiana procesów jest używana do tworzenia zatrasków, tzn. elementów pamięciowych sterowanych poziomem. Nazywane są one także przezroczystymi lub przepływowymi i są podobne do przerzutników z wyjątkiem tego, że ich wyjścia są identyczne jak wejścia w momencie 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 zatraskowego jest podobna do zegarowanego z tym, że tutaj na liście „Sensitivity List” znajdują się wszystkie sygnały wejściowe. Zakładając, że zatrask ma sygnał zezwolenia ENABLE aktywowany stanem wysokim proces taki może być następujący:

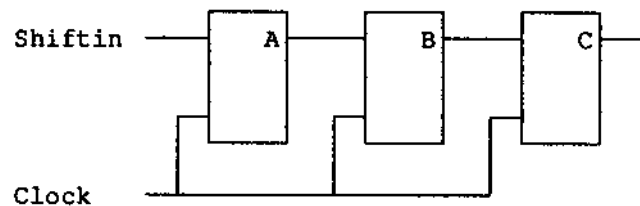
```
Load_latch : Process(Enable, Reset, Databus)
Begin
    If Reset='1' then --zerowanie zatrasku
        LATCH<="00000000";
    ElseIf Enable='1' then --zatrask aktywny
        LATCH<=Databus;
    End If;
End Process Load_latch;
```

Poza zerowaniem (Reset) proces jest aktywowany jakąkolwiek zmianą sygnału Enable lub wejść Databus. Dodanie szyny danych Databus do listy „Sensitivity List” zapewnia, że jeśli nawet stan wysoki jest na Enable stały i niezmienny, zmiany wejść przenoszą się na wyjście.

17.4. Przegląd mechanizmów ustalania poziomów sygnałów

Po omówieniu procesów i nauczaniu się mechanizmu tworzenia przerzutnika w procesie zegarowanym, prześledźmy poniższy przykład w celu wyjaśnienia mechanizmu

uaktualniania stanów logicznych. Rozważmy trzy przerzutniki w połączeniu prostego rejestru przesuwanego jak na rys. 3-5.



Rys. 3-5. Prosty trzystopniowy rejestr przesuwany

Powyższy rejestr może być opisany wewnątrz procesu zegarowanego jako:

```
Shiftreg : Process(Clock, Reset)
Begin
  If Reset='1' then
    A<='0';
    B<='0';
    C<='0';

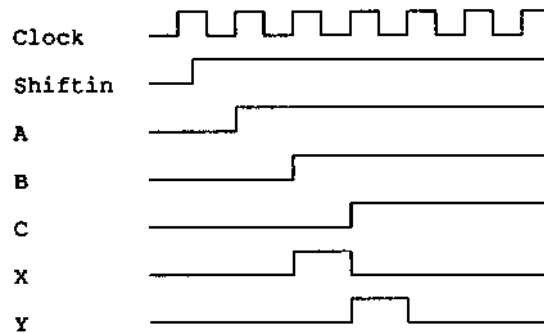
    ElsIf Rising_Edge(Clock) then
      A<=Shiftin;
      B<=A;
      C<=B;

    End If;
  End Process Shiftreg;
```

W momencie pojawienia się zbocza zegarowego stan układu jest określany na podstawie stanów przerzutników A, B i C oraz stanu wejścia Shiftin. Załóżmy, że system miał ustalone wartości logiczne równe 0 wszystkich czterech sygnałów. Jeśli na wejściu Shiftin pojawi się '1' wówczas przy następnym zboczach 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 Shiftin=1, A=0 oraz B=0. Nowe stany logiczne będą dlatego 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;
C<=B;
A<=Shiftin;
```

ponieważ tylko warunki w momencie czasu T (Shiftin=1, A, B, C=0) są używane do obliczania nowego stanu. Rysunek 3-6 przedstawia stany przerzutników po kilku okresach zegara.



Rys. 3-6. Przebiegi czasowe sygnałów rejestru przesuwanego.

Porównując działanie z konwencjonalnymi językami programowania, w których nie tylko od kolejności linii kodu zależą stany wyjściowe ale także pojawiają się one kolejno w momentach ich wywołania, można stwierdzić, że kod VHDL jest wygodniejszy do opisu układów logicznych. Z tego powodu w języku takim jak Pascal kod

```
A<=Shiftin;
B<=A;
C<=B;
```

ustawiłby A=1, następnie B=A (tj. =1), następnie C=B (tj.=1). W tym momencie staje się jasne, że pomysły zdarzeń czasowych i równoczesnych przypisań wartości sygnałów są istotne w opisie działania operacji logicznych.

17.5. Zdarzenia zegarowane i niezegarowane

Dla zilustrowania różnicy pomiędzy ustawianiem sygnałów wewnątrz i na zewnątrz procesu zegarowanego utwórzmy następujący sygnał

```
B and (not C);
```

który powinien być prostą bramką AND, przy założeniu, że dostępne są obie polaryzacje wyjść przerzutnika C. Niech sygnałowi Y przypisana zostanie powyższa funkcja wewnątrz procesu zegarowanego, natomiast sygnałowi X poza nim.

```
Shifreg : Process(Clock, Reset)
Begin
  If Reset='1' then
    A<='0';
    B<='0';
    C<='0';
  ElseIf Rising_Edge(Clock) then
    A<=Shiftin;
    B<=A;
    C<=B;
    Y<=B and not(C); -- wewnątrz procesu
  End If;
End Process Shifreg;

X<=B and not(C); -- poza procesem
```

Jak można zobaczyć na rys. 3-6, sygnał **X** przyjmuje swoją wartość gdy tylko zmieni się **B** lub **C**, podczas gdy **Y** musi czekać na następne zbocze zegarowe do przyjęcia swojej nowej wartości. Jak można zauważyć **X** jest bezpośrednio iloczynem logicznym (AND) sygnałów **B** oraz **not C**, podczas gdy **Y** jest opóźnione o okres zegara. Dlatego sygnał **Y** jest generowany przez przerzutnik poprzedzony bramką AND ponieważ jest on częścią procesu zegarowanego.

Reguła jest następująca :

Wszystkie sygnały, którym wartości są nadawane wewnątrz procesu zegarowanego są interpretowane w trakcie syntezy jako przerzutniki lub inne elementy pamięciowe.

Sygnały, którym nowe wartości są nadawane poza procesem lub w procesie niezegarowanym są interpretowane jako układy kombinacyjne.

18. Wyrażenia warunkowe (IF, CASE, WHEN, SELECT)

Testowanie warunków logicznych można wykonać poprzez wyrażenia: IF, WHILE, WHEN, CASE oraz SELECT. Wyrażenia If oraz Case są używane tylko wewnątrz procesów, natomiast Select i When są używane tylko poza procesami i w związku z tym są interpretowane w syntezie logicznej jako układy kombinacyjne.

Należy być ostrożnym przy ustawianiu warunków porównywania, takich jak równe, mniejsze niż, większe niż etc. i zapewnić równość rozmiarów obu stron relacji. W przypadku porównania wektora 8-mio bitowego z wektorem o innym rozmiarze nie otrzymamy zamierzonych rezultatów. Na przykład, jeśli Reg_A jest rejestrem 6-cio bitowym wówczas porównanie:

If Reg_A="000101"; będzie prawdą, jeśli Reg_A jest równy 5, ale wyrażenie

If Reg_A="0101";

gdzie Reg_A ma 6 bitów a "0101" ma 4 bity nigdy nie będzie prawdą. Dla uniknięcia takich problemów należy się upewnić, czy długości porównywanych ciągów są takie same.

18.1. Wyrażenie warunkowe IF

a.

```
If warunek then
    akcja;
End If;
```

b.

```
If warunek1 then
    akcja1;
Elsif warunek2 then
    akcja2;
End If;
```

c.

```
If warunek1 then
```

```

    akcja1;
Elsif warunek2 then
    akcja2;
Else
    akcja3;
End If;

```

d.

```

If warunek then
    akcja1;
else
    akcja 2
End If;

```

18.2. Wyrażenie warunkowe WHEN

Jest to warunkowe przypisanie sygnałów prowadzące w syntezie do logiki kombinacyjnej.

a.

```

SIGNAL<='1' WHEN Databus>Register Else '0';

```

(w syntezie odpowiada temu zapisowi N bitowy komparator 'większy niż', zależny od liczby bitów w „Databus”)

b.

```

DBus<=Geg_A when (Control='1') or (XYZ='0') or (Bus_C="0101")
else Reg_B;

```

(w syntezie odpowiada temu zapisowi multiplexer sterowany przez trzy rozłączne warunki)

c.

```

Sum<=A+B when Select='1' else C+D;

```

(w syntezie odpowiada temu zapisowi sumator multipleksowany)

18.3. Wyrażenie warunkowe SELECT

Wyrażenie warunkowe SELECT oznacza „Selected Signal Assignment” - selektywne przypisane wartości, w syntezie logicznej interpretowane jako układ kombinacyjny.

Składnia:

```

With Sygnał_testowany SELECT

Target_output <=
    Waveform_1 when warunek1,
    Waveform_2 when warunek2,
    ...
    Waveform_N when warunekN;

```

--sygnał wyjściowy
--kształt sygnału 1
--kształt sygnału 2
--kształt sygnału N

Przykład:

a.

```

With SigA Select
    Output<=
    DataA when "00",
    DataB when "01",

```

```
DataC when "10",  
DataD when "11";
```

(w syntezie logicznej zapis ten odpowiada multiplexerowi 4 na 1 sterowanemu przez SigA. Sygnał sterujący jest dwubitowy, przy czym Output jest przełączany na jedną z czterech szyn danych)

Znak pionowej kreski '|' może być użyty do wskazania wielokrotnych warunków lub wyborów umieszczonych na liście, które powodują tę samą akcję. Na przykład zamiast:

```
DataA when "00",  
DataA when "01",
```

można użyć prostszego zapisu:

```
DataA when "00" | "01",
```

Taki skrót może być użyty dla operacji warunkowych Select i Case.

18.4. Wyrażenie warunkowe CASE

Wyrażenie warunkowe CASE służy do selekcjonowanego przypisania wartości wewnątrz procesów.

Jest to wyrażenie podobne do SELECT z przykładów powyżej, ale dla każdego z warunków może wystąpić wielokrotna akcja.

a.

```
CASE SigA IS  
  When "00" => Output <= DataA;  
  When "01" => Output <= DataB;  
  When "10" => Output <= DataC;  
  When "11" => Output <= DataD;
```

(zapis ten umieszczony wewnątrz procesu niezegarowanego daje w wyniku syntezy logicznej taki sam rezultat jak przykład z zastosowaniem wyrażenia SELECT)

b.

```
Case State is  
  When S0 => Output <= A+B;  
             Sig1 <= '1';  
             State <= S1;  
  When S1 => Pulse <= '0';  
             State <= S0;  
End CASE;
```

(wyrażenie CASE sprawdza sygnał nazwany State i w zależności od tego, czy ma on wartość S0 czy S1, podejmowane są odpowiednie akcje).

Wyrażenie CASE musi sprawdzić wszystkie możliwe wartości sygnału testowanego. Jeżeli jednak nie wszystkie wartości są ważne, to można zastosować następującą składnię:

```
When Others => Akcja or 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.

19. Tablice (ARRAY)

Sygnaly i dane mogą zostać zorganizowane w postaci tablic wartości, co ma swój odpowiednik sprzętowy w postaci pamięci ROM lub RAM. Tablice są częścią klasy VHDL nazywanej typami złożonymi (COMPOSITE TYPES), gdzie każdy element tablicy ma ten sam typ. Specyfikacja tablic jest podobna do specyfikacji pamięci, która składa się z pewnej liczby rzędów danych, z których każdy posiada pewną długość. Np. pamięć 32 x 8 bitów może zostać opisana w VHDL następująco:

```
Type Data_File is Array (0 to 31) of Std_Ulogic(7 downto 0);
```

lub w sposób ogólny

```
Type Nazwa_typu is Array (liczba wartości) of Type;
```

Liczba wartości w tablicy jest jej zakresem (**Range**) i może być wartością stałą jak np. (31 downto 0) lub mieć niezdefiniowaną wielkość. Pierwszy przypadek jest nazywany tablicą o ustalonej wielkości (constrained), drugi typ nie ma z góry określonej wielkości tablicy, a rzeczywisty jej wymiar jest wyznaczany przez kolejne do niej odwoływania. Np. pamięć ROM składająca się z 8-miu 4-ro bitowych słów może być zdefiniowana następująco:

```
Type LOOKUP is Array (0 to 7) of Integer Range 0 to 15;
```

```
Constant ROM_Table_1:LOOKUP:=(12,4,6,14,1,7,12,0);
```

```
Constant ROM_Table_2:LOOKUP:=(7,5,3,1,6,4,2,0);
```

Tablica LOOKUP o ustalonej wielkości jest zdefiniowana jeden raz jako 8 wartości typu INTEGER, każda z nich z zakresu od 0 do 15. Tablica LOOKUP jest używana dwukrotnie jako ROM_Table_1 i ROM_Table_2 - każda zawiera inne elementy. Liczba elementów tablicy musi być zgodna z podaną w jej definicji, w przeciwnym wypadku zostanie wygenerowany błąd kompilatora, co daje możliwość sprawdzenia zgodności wymiarów.

Taka sama tablica może zostać zdefiniowana bez określonej z góry liczby jej elementów z użyciem składni (natural range <>), co oznacza, że zakres tablicy jest liczbą naturalną bez specjalnego ograniczenia górnego zakresu. Np.:

```
Type LOOKUP is Array (natural range <>) of Integer (range 0 to 15);
```

```
Constant ROM_Table:LOOKUP:=(1,3,5,7,9,11,13,15,17);
```

gdzie wymiar tablicy jest określony przez liczbę wylistowanych wartości, w tym przypadku 9. Zaletą takiej tablicy jest to, że wprowadzanie zmian w liczbie elementów może być wykonane bez zmiany jej definicji. Przykłady zastosowań w projektowaniu układów logicznych zostaną zaprezentowane w następnych rozdziałach.

20. Funkcje i Procedury

Funkcje i procedury w VHDL są podprogramami używanymi w celu uproszczenia kodowania operacji wykorzystywanych wielokrotnie i powszechnie. Jeśli pewien blok kodu jest używany wielokrotnie może zostać umieszczony w funkcji lub procedurze jednokrotnie, a następnie być wywoływany poprzez jej nazwę.

Interesującą rzeczą jest to, że syntezywany układ może być w rzeczywistości różny dla różnych typów wejść. Np. dla zastosowań w filtrach cyfrowych, gdzie mnożona jest zawartość pewnego rejestru przez wartość innego rejestru, należy spodziewać się standardowej wersji mnożnika. Jeśli jednak mnożona będzie zawartość pewnego rejestru przez stałą - wynikowy układ będzie prostszy. Dlatego ten sam podprogram będzie tworzył dwa różne układy.

20.1. Istota funkcji

Definicja funkcji składa się z: nazwy, specyfikacji wejść i typu wyjścia. Funkcje nie mogą zmieniać wartości parametrów umieszczonych na jej liście (procedury mogą). Parametry na liście funkcji mogą być tylko trybu IN. Z tego powodu nie ma konieczności podawania trybu parametrów funkcji – są one domyślnie przyjmowane jako IN.

Przykład deklaracji funkcji

```
function rising_edge(signal clk: in 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 stałe. W takim przypadku parametr aktualny może być zmienną, sygnałem, stałą lub wyrażeniem. Jeśli dla parametru wejściowego nie sprecyzowano klasy domyślnie jest przyjmowany jako stała. Wyrażenia Wait nie są dopuszczalne w ciele funkcji (procedury dopuszczają) i

dlatego są wykonywane w zerowym czasie wykonania. Z tego powodu wyrażenie `wait` nie może znajdować się w procedurze wywoływanej przez funkcję.

VHDL'93 obsługuje 2 typy funkcji: czyste i nieczyste (pure and impure). Pierwsza z nich daje taki sam wynik gdy wywoływana jest z tymi samymi parametrami. Funkcje nieczyste mogą dać różne wyniki przy wywołaniu ich z tymi samymi wartościami parametrów. Jest to możliwe z powodu tego, że funkcje mogą widzieć sygnały nie umieszczone na liście parametrów funkcji, np. lista sygnałów bloku entity. Funkcje czyste są najczęściej stosowane do konwersji typów oraz jako funkcje arbitrażowe. W dalszej części skupimy się tylko na funkcjach czystych.

20.1.1. Przykład 1. Detekcja zbocza narastającego sygnału.

Poniższy przykład przedstawia wykorzystanie funkcji jako detekcji zbocza narastającego sygnału. Proszę zauważyć miejsce zdeklarowania funkcji – wewnątrz części deklaracyjnej architektury. Oprócz tego miejsca funkcje mogą być deklarowane w części deklaracyjnej procesów tj. pomiędzy słowami `process` a `begin`. Różnica w deklaracjach jest taka, że w tym drugim przypadku zdeklarowana funkcja może być wywołana tylko wewnątrz procesu w którym została zdeklarowana. W praktyce funkcje i procedury najczęściej deklaruje się w pakietach- jednostkach projektowych które zostaną opisane w dalszej części.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is port (D, Clk : in std_logic;
                   Q, Qbar: out std_logic);
end entity dff;

architecture behavioral of dff is

function rising_edge (signal clock : std_logic) return boolean is
variable edge : boolean:= FALSE;
begin
edge := (clock = '1' and clock'event);
return (edge);
end function rising_edge;

begin
output: process
begin
wait until (rising_edge(Clk));
Q <= D after 5 ns;
Qbar <= not D after 5 ns;
end process output;
end architecture behavioral;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is port (D, Clk : in std_logic;
                   Q, Qbar: out std_logic);
end entity dff;
architecture behavioral of dff is

function rising_edge (signal clock : std_logic) return boolean
is
variable edge : boolean:= FALSE;
```

```

begin
    edge := (clock = '1' and clock'event);
    return (edge);
end function rising_edge;
begin
    output: process
    begin
        wait until (rising_edge(Clk));
        Q <= D after 5 ns;
        Qbar <= not D after 5 ns;
    end process output;
end architecture behavioral;

```

20.1.2. Przykład 2

Konwersja typów jest innym częstym przypadkiem stosowania funkcji. Rozważmy model pamięci jako tablica jednowymiarowa:

```
type T_RAM is array(0 to 255) of std_logic_vector(7 downto 0);
```

Ta tablica jest indeksowana poprzez obiekt typu całkowitego, jednakże sygnały szyn danych i adresów najczęściej są reprezentowane jako `std_logic_vector`, będziemy musieli więc wykorzystać funkcję konwersji typów z `std_logic_vector` do typu całkowitego. Z drugiej strony może zdarzyć się sytuacja w której chcemy wykorzystać modele bloków entity wykonane przez inne osoby. Aby wstawić je do własnego projektu może okazać się konieczna konwersja typów portów wej/wyj. Poniżej przedstawiony jest przykład zamiany `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(svalue:std_logic_vector) return
bit_vector is
variable outvalue:bit_vector(svalue'length-1 downto 0);
begin
    for i in svalue'range loop
        case svalue(i) is
            when '0' => outvalue<='0';
            when '1' => outvalue<='1';
            when others => outvalue<='0';
        end case;
    end loop;
    return (outvalue);
end function to_bitvector;

```

Inne funkcje konwersji typów można znaleźć w pakietach `std_logic_1164` oraz `std_logic_arith`. Poniżej zamieszczono kilka przykładów:

```

function CONV_INTEGER(ARG: SIGNED) return INTEGER is

function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT is

function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return
UNSIGNED is

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)

```

```
return STD_LOGIC_VECTOR is
```

20.2. Funkcja arbitrażowa

Funkcja arbitrażowa obejmuje specjalną klasę funkcji definiującą wartość sygnału w przypadku wielokrotnego jego sterowania.

20.3. Istota procedury

Procedury są podprogramami które mogą zmieniać wartość jednego lub więcej parametrów wejściowych. Poniższa deklaracja procedury przedstawia interfejs procedury.

```
procedure read_v1d (variable fname: in text; v: out
std_logic_vector);
```

Procedura powyższa czyta zawartość pliku fname. Pierwszą różnicą w stosunku do funkcji jest to, że parametry procedury mogą być trybu OUT. Podobnie możemy stosować parametry trybu INOUT. Podobnie jak w przypadku funkcji typ parametru formalnego musi być zgodny z typem parametru aktualnego wstawianego w czasie wywoływania procedury. Jeśli klasa i tryb parametru formalnego nie są zadeklarowane wówczas domyślnie przyjmuje on tryb IN oraz klasę constant, podczas gdy parametry trybu OUT oraz INOUT są domyślnie uważane jako zmienne. Zmienne zadeklarowane w procedurze są inicjowane w czasie każdego wywołania procedury a ich wartości nie przechodzą do innych wywołań tej samej procedury.

Przykład 1 – Interfejs pamięci.

Rozważmy prosty model procesora zawierający jedynie 2 elementy CPU i pamięć. CPU musi mieć możliwość zarówno odczytu jak i zapisu elementów pamięci. Zostaną stworzone 2 procedury: jedna do odczytu i druga do zapisu z pamięci

```
library IEEE;
use IEEE.std_logic164.all;
entity CPU is
port (write_data : out Std_logic_vector (31 downto 0); -- data from memory
      ADDR :out std_logic_vector (2 downto 0); -- CPU generated address
      MemRead, MemWrite: out std_logic; -- read and write control signals from CPU B
      read_data : in Std_logic_vector(31 downto 0); -- data to memory
      S : in std_logic);
end entity CPU;

architecture behavioral of CPU is

procedure mread (address : in std_logic_vector (2 downto 0);
                signal R : out std_logic;
                signal S : in std_logic;
                signal ADDR: out std_logic_vector (2 downto 0);
                signal data : out std_logic_vector (31 downto 0)) is
begin
  ADDR <= address;
  R<='1';
  wait until S ='1';
  data <= read_data;
  R<='0';
end procedure mread;

procedure mwrite (address : in std_logic_vector (2 downto 0);
```

```

        signal data : in std_logic_vector (31 downto 0);

        signal ADDR : out std_logic_vector (2 downto 0);

        signal W : out Std_logic;
        signal DO : out std_logic_vector (31 downto 0)) is
begin
    ADDR <= address;
    DO<= data;
    W<='1';
    wait until S = ' 1';
    W<='0';
end procedure mwrite;

```

20.3.1. Używanie procedur

Sygnały nie mogą być deklarowane w procedurach, jednakże 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 porty są widzialne dla procesu. O takiej procedurze mówi się, że ma efekt boczny gdyż zmienia wartości sygnału nie podanego na liście parametrów. Bardziej czytelne jest stosowanie przekazywania poprzez listę parametrów. Jeśli klasa parametru nie jest zadeklarowana a tryb jest out lub inout wtedy klasą domyślną jest zmienna. Jeśli tryb jest in wówczas klasą domyślną jest stała.

Procedury mogą być umieszczane w części deklaracyjnej procesu. Wiemy, że proces nie może mieć równocześnie listy czułości jak i wyrażenia typu wait. Z tego powodu wynika, że proces w którym następuje wywołanie procedury zawierającej wyrażenie wait nie może posiadać listy czułości.

21. Synteza układów sekwencyjnych

W rozdziale 17 przedstawione zostały zasady ogólne używania procesów zegarowanych oraz przykładowe ich użycia w logice sekwencyjnej. Ten rozdział jako pierwszy omawia temat projektowania układów sekwencyjnych z wykorzystaniem VHDL. Przedstawione są liczne przykłady liczników. Zasygnalizowane zostaną także problemy testowalności tych układów.

Liczniki są podstawowymi blokami funkcjonalnymi wielu typów układów sekwencyjnych i z powodu ich dużej roli w technice cyfrowej zostaną omówione szczegółowo. Techniki projektowania prezentowane są stopniowo, poprzez różne przykłady, począwszy od najprostszych rozwiązań aż do wykonań z wieloma funkcjami dodatkowymi, które są często spotykane w systemach praktycznych.

W ciągu następnych przykładów okaże się, że doświadczenie projektanta jest bardzo ważnym czynnikiem umożliwiającym swobodną kontrolę nad przebiegiem projektowania, pozostawiając szczegółową obsługę projektu na poziomie bramek logicznych narzędziom syntezy umożliwia skupienie się na ogólnej funkcjonalności układu.

21.1. Podstawowe wiadomości o licznikach

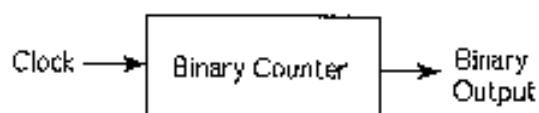
Obecnie w użyciu jest wiele typów liczników np. liczących w górę, w dół lub w obie strony. W większości zastosowań liczniki zbudowane są w ten sposób, że zarówno sygnał sterujący (CLOCK) oraz sygnał zezwalający (ENABLE) jak i sygnały zerowania (RESET) oraz ustawiania licznika na żadaną wartość (LOAD), są od siebie

odseparowane. Najczęściej stosuje się zliczanie binarne lub BCD (Binary Coded Decimal), chociaż spotyka się także liczniki pracujące w kodzie Graya, Johnsona lub pseudolosowe. Wszystkie te typy są łatwo realizowalne w technice VHDL.

Poprzedni rozdział zawierał opis ogólnych zasad VHDL. Różni dostawcy narzędzi syntezy VHDL dołączają różniące się zestawy bibliotek i wbudowanych operacji takich jak: zamiana typów, „przeładowywanie” operatorów (ang. *overloading*), oraz atrybuty specjalne, co czyni język VHDL prostszym i łatwiejszym do zrozumienia. Szczegóły dotyczące wykorzystania tych właściwości będą omawiane przy ich pierwszym użyciu. Zwracamy uwagę, że chociaż w VHDL można posługiwać się różnymi typami sygnałów, np. `BIT`, `Std_Logic`, to jednak w dalszych rozważaniach stosowany będzie typ `Std_Ulogic` ze względu na lepsze możliwości symulacyjne i większą uniwersalność w porównaniu z innymi typami.

21.2. Podstawowy licznik binarny

Rozważymy najpierw proste liczniki binarne bez dodatkowych wejść funkcyjnych. Schemat blokowy takiego układu jest przedstawiony na rys. 4-1.



Rys. 4-1. Podstawowy licznik binarny.

W opisie VHDL N-bitowy licznik jest zazwyczaj przypisany do sygnału typu `Std_Ulogic_Vector` lub `Integer`, a jego działanie opisane jest wewnątrz procesu zegarowanego.

Na przykład, dla licznika ośmiobitowego deklaracja sygnału może być następująca:

```
Signal Upcount:Std_Ulogic_Vector(7 downto 0);  
lub  
Signal Upcount: Integer Range 0 to 255;
```

lub przez podobny typ sygnału zależny od możliwych do stosowania przez konkretne narzędzie syntezy.

Wykonanie operacji zliczania wewnątrz procesu może być następujące:

```
Upcount<=Upcount+'1'    -- gdy Upcount jest wektorem oraz dostępny jest  
                        -- overloading typów lub  
Upcount<=Upcount+1      -- gdy Upcount jest typu Integer.
```

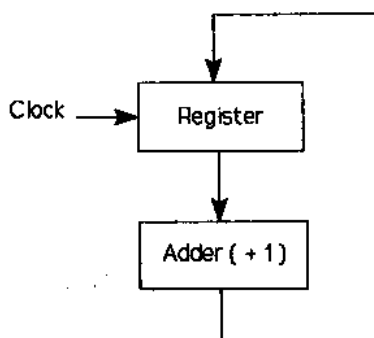
Powyższa operacja będzie wykonana w momencie pojawienia się zbocza zegarowego lub w odpowiedzi na inne sygnały. Oczywiście dla wprowadzenia sygnału resetu asynchronicznego należy dodać odpowiedni fragment kodu. Należy zauważyć, że poprzez opis licznika na poziomie RTL oraz dzięki użyciu syntezy logicznej, specyfikowana jest funkcja licznika, a nie szczegóły dotyczące jego budowy. Należy również zauważyć, że w powyższym przykładzie zastosowano *overloading*, który umożliwia dodanie do wektora typu `Std_Ulogic` pojedynczego bitu '1'. Bez takiej możliwości należałoby użyć składni:

```
Upcount<=Upcount+"00000001";
```

aby kompilator VHDL nie sygnalizował błędu składni. Jest to jeden z powodów, dla których należy dbać o kompletność modułów dostarczanych przez dostawcę systemu projektowego; czyni to proces kodowania w języku VHDL bardziej przejrzystym i łatwiejszym w użyciu w trakcie projektowania logicznego.

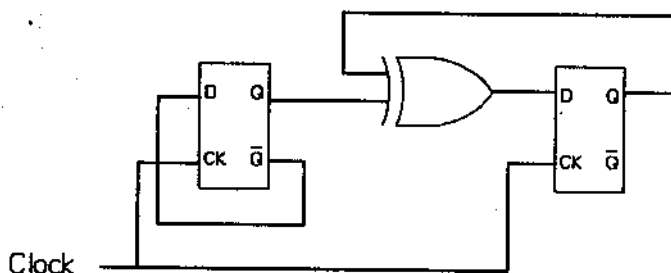
21.3. Jaki jest wynik procesu syntezy logicznej?

W przypadku syntezy logicznej licznika omawianego powyżej, interesujące jest spostrzeżenie, że jeśli układ nie zawiera ograniczeń czasowych, to można go uznać za rejestr oraz prosty sumator, który dodaje 1 do zawartości rejestru w takt zegara, czyli można go przedstawić jak na rys. 4-2.



Rys. 4-2. Rejestr = Rejestr + 1

Biorąc pod uwagę składnię VHDL wykorzystaną do opisu licznika schemat z rys. 4-2 wydaje się sensownym rozwiązaniem. Z drugiej strony wiadomo, że układy licznikowe realizuje się w praktyce w odmienny sposób. W przypadku dokładnej analizy równań logicznych okazuje się jednak, że dla każdego przerzutnika w liczniku bazującym na układzie sumatora, są one takie same jak dla licznika projektowanego konwencjonalnie.



Rys. 4-3. Schemat logiczny podstawowego licznika binarnego

Syntezerowane projekty są zazwyczaj budowane z wykorzystaniem przerzutników typu D (a nie J-K, czy innych). Rzeczywisty układ, będący rezultatem syntezy logicznej na poziomie bramek logicznych, zależy również od wyrażeń czasowych umiejscowionych w projekcie: jeśli ich nie ma, to układ będzie optymalizowany ze względu na minimalną liczbę użytych bramek i uzyskamy szeregowy łańcuch bramek. W przypadku syntezy z dodatkowymi wymaganiami co do częstotliwości zegara, w miejsce prostej sieci bramek pojawi się automatycznie logika typu "carry look-ahead" - jeśli tylko jest tam niezbędna. Nadal jednak minimalizowana jest liczba bramek, układ typu "look-ahead" jest stosowany tylko tam, gdzie jest niezbędny!

Drugim spostrzeżeniem jest to, że układ z powyższego rysunku wykorzystuje funkcję EX-OR, co w praktyce zostanie prawdopodobnie zrealizowane inaczej, zależnie od zestawu komórek dostępnych w bibliotece dostawcy układów ASIC. Na przykład, seria 100k układów wielkiej skali integracji LSI, ma kilka funkcji EX-OR jak również kilka kombinacji bramek AND-OR o różnej równoważnej liczbie bramek oraz różnych czasach propagacji. Program syntezy zazwyczaj wybiera bramkę AO2, która jest złożeniem dwóch bramek AND oraz jednej NOR i ma równoważną liczbę bramek równą 2. Natomiast bramka EX-OR (EO2) ma 3 równoważne bramki i jest nieco wolniejsza. Dalszym uproszczeniem, jak to przedstawiono na Rys. 4-3, jest to, że z większym prawdopodobieństwem zostanie zastosowane wyjście zanegowane przerzutnika niż dodatkowe negowanie sygnału na jego wejściu. Program syntezy bowiem wie, że jeśli zanegowane zostanie wejście D, to wyjście Q będzie również odwrócone i w związku z tym – wyjście Nie-Q będzie miało właściwą polaryzację.

21.4. Projektowanie licznika w języku VHDL

Rozważmy jaka składnia VHDL jest potrzebna do opisu licznika binarnego zliczającego w górę z asynchronicznym sygnałem resetu. W analizowanym przypadku zakładamy, że system wymaga zegara 32 MHz. W pewnym miejscu projektu sygnały CLOCK oraz RESET będą deklarowane albo poprzez wejścia PORT pewnej jednostki Entity albo jako SIGNAL wewnątrz ARCHITEKTURY. Załóżmy, że sygnały te będą miały następujące nazwy: CLK32M oraz MRESET.

Ogólne zasady syntezy VHDL układów licznikowych są takie same jak układów sekwencyjnych, a ich przypomnienie przedstawione jest w punktach poniższych:

- Procesy VHDL opisujące układy sekwencyjne są procesami 'zegarowanymi'. W VHDL oznacza to po prostu, że układ używa sygnału zegara tzn. czeka na odpowiednie zbocze zegara dla podjęcia opisanych w nim akcji. Proces opisujący nasz licznik zawiera na swojej liście (sensitivity list) zarówno sygnał zegara jak i sygnał zerujący.
- Do opisu operacji w procesach zegarowanych stosuje się składnię IF-THEN-ELSE. Kod warunków zerowania oraz zegarowania umieszczony jest w oddzielnych pętlach warunkowych IF; podana tam struktura VHDL powinna być traktowana jako normalna technika projektowania układów sekwencyjnych.
- Dla detekcji zbocza użyto funkcji Rising_Edge (w rodzinach układów ASIC zegarowanych przerzutnikami reagującymi na zbocze narastające; jeśli jest inaczej, to należy posłużyć się funkcją Falling_Edge). Obie te funkcje pochodzą ze standardowej biblioteki IEEE 1164 obsługującej logikę 9-cio stanową.

Alternatywą dla operacji Rising_Edge jest obowiązująca w pewnych systemach syntezy składnia:

```
(CLK32M'event and CLK32M='1'); -- taka sama jak Rising_Edge
```

Biorąc powyższe pod uwagę, kod opisujący prosty licznik binarny zliczający w górę może być następujący:

```
-- Deklaracje sygnałów i stałych
```

```
SIGNAL CLK32M, MRESET: Std_Ulogic;
SIGNAL Upcount: Std_Ulogic_Vector( 5 downto 0);
Begin
```

```
--Definicja procesu

Upcounter:  Process(CLK33M,MRESET)
BEGIN
  IF MRESET='1' THEN
    Upcount<="000000";
  ELSIF Rising_Edge(CLK33M) THEN
    Upcount<=Upcount+'1';
  End IF;
End Process Upcounter;
```

Przykład. 4-1. Prosty licznik binarny z asynchronicznym zerowaniem.

Jest kilka właściwości procesów VHDL, które należy w tym miejscu przypomnieć. Po pierwsze, proces wykonuje się od wyrażenia BEGIN do wyrażenia END, jeśli tylko wystąpiła zmiana sygnału umieszczonego na liście procesu (sensitivity list) tj. w powyższym przypadku CLK32M lub MRESET. Sygnały, których wartości są ustalane wewnątrz procesu, są uaktualniane wszystkie jednocześnie na koniec procesu. Dla projektanta układów cyfrowych jest to identyczne jak operacje z użyciem przerzutników typu D, gdzie jego wejście może zmieniać się wcześniej niż zbocze zegara, ale wyjście zostanie ustalone dokładnie w momencie odpowiedniego zbocza zegarowego. W powyższym przykładzie, jeżeli stan licznika jest Upcount="000100", w momencie rozpoczęcia procesu pozostanie on taki sam w czasie jego trwania, a nowa wartość Upcount="000101" zostanie przyjęta na końcu wykonywania procesu.

Licznik z powyższego przykładu ma bezpośrednie wejście zerowania MRESET podłączone bezpośrednio do każdego z przerzutników. W przypadku gdy sygnał zegara jest ciągły, w momencie kiedy licznik osiągnie stan sześciu jedynek, następne zbocze zegarowe ustawi same zera, a cykl zliczania zostanie powtórzony.

21.5. Dodanie wejść ENABLE oraz LOAD (zezwalającego oraz ładującego)

Powyższy przykład licznika realizuje tylko podstawową funkcję zliczania, podczas gdy bardzo często potrzebne są możliwości synchronicznego ładowania pewnego stanu oraz wejście zezwalające na sterowanie procesem zliczania. Alternatywą dla tego typu licznika, zbudowanego w standardzie TTL, są czterobitowe liczniki rodziny 54163, których synteza jest znacznie bardziej elastyczna, bo pozwala na zastosowanie tej samej techniki do projektowania liczników o dowolnej liczbie bitów.

Rozszerzmy zatem poprzednią wersję przez dodanie sygnału zezwalającego ENABLE oraz ładującego LOAD, który umożliwi ustawienie licznika na wartość pobraną z szyny danych DATABUS. Dodatkowo założmy, że przy wartości logicznej '1' sygnał ENABLE zapewnia zliczanie oraz ładowanie wyjść zawartością szyny DATABUS.

```
-- Deklaracje sygnałów i stałych

SIGNAL CLK32M,MRESET,ENABLE,LOAD: Std_Ulogic;
SIGNAL Upcount,DATABUS: Std_Ulogic_Vector(5 downto 0);

Begin

-- Definicja procesu
```


Zauważmy, że operacja ładowania zrealizowana jest jako wyrażenie IF wewnątrz innego IF, który sprawdza obecność sygnału zezwalającego. W takim przypadku operacja ładowania będzie wykonywana tylko wówczas, gdy sygnał ENABLE jest w stanie wysokim. Natomiast w przeciwnym przypadku zostanie zignorowany. W przypadku gdy chcemy aby ładowanie było zawsze dostępne, wystarczy w następujący sposób zmodyfikować kod VHDL:

```
    If LOAD = '1' then
        Upcount<= DATABUS;

    Elsif ENABLE = '1' then
        Upcount<= Upcount + '1';

    End if;
```

Użycie składni IF-ELSE oraz IF-ELIF jest ważne ponieważ w trakcie syntezy istotny jest **priorytet** wykonywanych operacji. Rozważmy sytuację, w której operacje zostaną zakodowane w sposób niezależny, a więc:

```
    If LOAD = '1' then
        Upcount <= DATABUS;
    End If;

    If ENABLE = '1' then
        Upcount <= Upcount + '1';
    End If;
```

Reguły VHDL są takie, że jeśli sygnałowi wartość nadawana jest kilka razy, np. w taki sposób:

```
    Upcount <= DATABUS, a następnie Upcount <= Upcount + '1',
```

to wówczas ostatnie przyporządkowanie ustala wartość logiczną sygnału, a poprzedni przydział jest ignorowany. Algorytm syntezy wykorzysta ten fakt w taki sposób, że nada zliczaniu wyższy priorytet niż ładowaniu, co może być niezgodne z intencją projektanta. Natomiast odwrotny porządek tych wyrażen prowadzi do innego układu.

Dążąc zatem do wyeliminowania błędów w projektowaniu, należy przed rozpoczęciem kodowania w VHDL podjąć decyzję, w jaki sposób układ powinien reagować na sygnały wejściowe wielokrotne.

W tym miejscu można sformułować następującą regułę:

Operacje, które mają pierwszeństwo należy umieścić w warunku IF-THEN-ELSE, a następnie pozostałe funkcje.

Z przykładu 4-2 wynika, że jeśli sygnał LOAD jest aktywny, to taka operacja może spowodować, że licznik przestanie spełniać swoją funkcję normalnego zliczania.

21.6. Logika kontrolna i maszyny stanów.

Niemalże wszystkie systemy logiczne wymagają logiki kontrolnej do sterowania operacjami. Projektowanie takiego typu logiki, która jest także nazywana sekwencerami lub skończonymi maszynami stanów używa nieco innego podejścia. Ten rozdział poświęcony jest podstawowym typom logiki kontrolnej, metodom analizy oraz praktycznymi technikami projektowania VHDL. Przedstawiony zostanie często stosowany w takich przypadkach typ wyliczeniowy oraz wyrażenie warunkowe typu CASE.

21.7. Podstawy projektowania logiki kontrolnej.

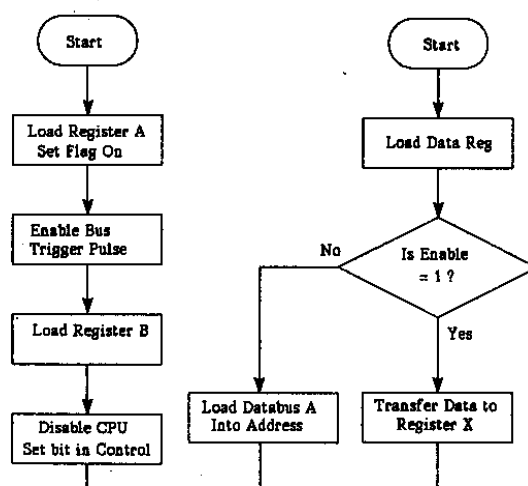
Praca podstawowych bloków logicznych systemu jest nadzorowana przez różne typy logiki kontrolnej. Poprzednie rozdziały przedstawiały samodzielnie pracujące bloki liczników, generatorów, rejestrów itp. W kompletnym systemie musi być dodatkowa logika kontrolna sterująca przepływem sygnałów. Termin logika kontrolna może być używany zarówno do centralnego sterowania wielu funkcji systemowych jak również do sterowania pracą pojedynczej funkcji w większym bloku. Podstawowym założeniem jest to, że logika kontrolna jest z natury układem sekwencyjnym a jej działanie zależy zarówno od sygnałów wejściowych jak i od stanów wewnętrznych.

Funkcje kontrolne wykonywane są sekwencyjnie poprzez bloki nazywane sekwencerami. Sekwencery umożliwiają wykonanie operacji logicznych w zależności od obecnego stanu systemu jak i stanów pewnych sygnałów wejściowych. To połączenie jest całkowicie deterministyczne - dla każdej kombinacji wejść i stanów wewnętrznych istnieje tylko jedna odpowiedź.

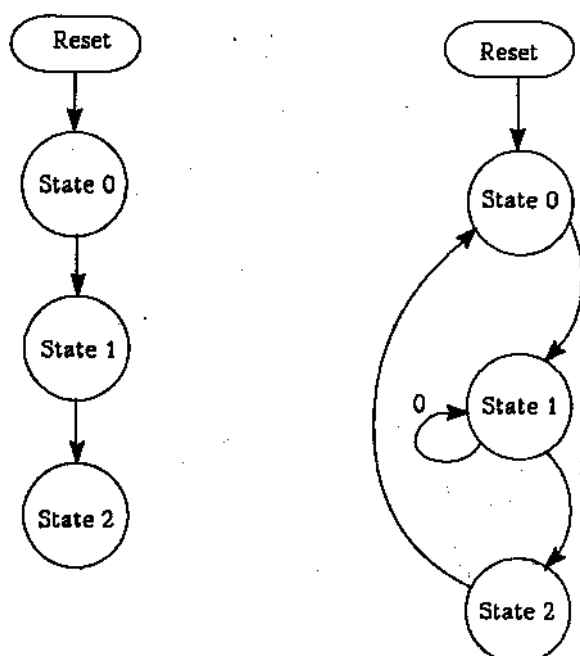
Maszyna stanów jest formą sekwencera który jest zorganizowany jako skończony zestaw stanów logicznych. Każdy ze stanów reprezentuje jeden zestaw akcji takich jak np. zezwolenie zliczania czy generacja potwierdzenia. Niemalże wszystkie stany zawierają metodę przeniesienia sterowania do innego stanu, bazując na pewnych stanach obecnych. Każdy stan który nie ma stanu następnego jest ostatnim stanem maszyny stanów a system pozostanie w tym stanie na zawsze (chyba, że w międzyczasie pojawi się „reset” asynchroniczny). Przejście pomiędzy poszczególnymi stanami może być warunkowe (bazujące na innych sygnałach systemu) lub bezwarunkowe.

Projektowanie i analiza sekwencerów bardzo często wykorzystuje graficzne formy takie jak **schemat przejść** (flowchart) lub **wykres stanów** (State Diagram). Schemat przejść jest typem schematu używanego do określenia wykonania procesów łącznie z warunkami i operacjami zapewniającymi rozwiązanie problemu logicznego. Schemat przejść nie jest jednak ograniczony do zastosowań w projektowaniu problemów logicznych ale może przedstawiać każdy ogólny proces. Odpowiednikiem logicznym schematu przejść jest wykres stanów, który każdemu stanowi przyporządkowuje okrąg lub prostokąt. Wewnątrz wykresu każdy ze stanów określa zarówno wykonywaną operację jak i warunki przebiegu i rozgałęziania przejść pomiędzy stanami. Przypisanie następnego stanu może być warunkowe (wykonywane na podstawie stanu obecnego i warunków na sygnały wejściowe) lub bezwarunkowe (tylko na podstawie stanu obecnego). Owale reprezentujące „reset” jest użyty do wskazania stanu początkowego, który zazwyczaj sprzętowo jest wykonywany jako sygnał asynchroniczny. Podczas symulacji, inicjuje się wszystkie nieznane wyjścia do stanów znanych. W całym systemie funkcja inicjalizacji umożliwia wykonywanie całkowicie deterministycznych symulacji systemów cyfrowych.

Sekwencery / maszyny stanów są używane w szerokim zakresie zastosowań. W ogólności mogą one być podzielone na warunkowe i bezwarunkowe. Rys. 6-1 oraz 6-2 przedstawiają przykładowy schemat przejść i wykres stanów.



Rys. 6-1. Schemat przejść sekwencera bezwarunkowego i warunkowego.

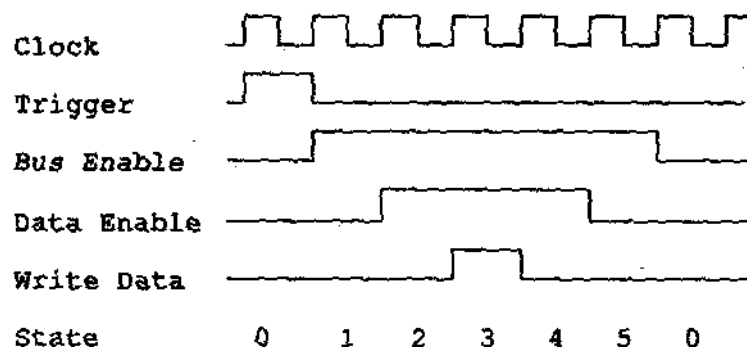


Rys. 6-2. Wykres stanów sekwencera bezwarunkowego i warunkowego.

21.8. Sekwencer bezwarunkowy

Najprostszą formą sekwencera jest sekwencer bezwarunkowy, który przechodzi do następnych stanów bez sprawdzania żadnych dodatkowych warunków które mogłyby zmienić przebieg. Jest to przedstawione po lewej stronie rys. 6-1 i 6-2. Najczęstszym zastosowaniem jest sekwencer czasowy lub sterownik dający ustalone serie sygnałów wykonujących pewne akcje. Jako przykład rozważmy prosty kontroler szyny dla podsystemu pamięci i założmy, że dane i adres pamięci pochodzą z różnych źródeł. Aby sterować zapis do pamięci kontroler musi dezaktywować szyny wejściowe, następnie wybrać jedno ze źródeł, zapisać dane do pamięci i zwolnić szynę danych dla kolejnej operacji. Wymagane przebiegi sygnałów przedstawione są na rys. 6-3, są one przesunięte

w czasie dla uniknięcia problemów czasowych a cały proces zapisu zabiera 6 okresów zegara.



Rys. 6-3. Przebiegi czasowe kontrolera pamięci.

Dla tak prostego kontrolera jak w powyższym przypadku istnieje wiele metod generacji wymaganych sygnałów. Np. można użyć licznik binarny i dekodować odpowiednie stany dla generacji przebiegów sygnałowych. Spójrzmy jednak na ten problem od strony projektu sekwencera, w którym będziemy kodować odpowiednią akcję w języku VHDL i pozwolimy syntezerowi na wybór najlepszej implementacji układowej.

Ponieważ sekwencery / maszyny stanów są funkcjami zegarowanymi zostaną wykonane w procesie, z uwzględnieniem reguł z rozdziału poprzedniego. Każdy ze stanów maszyny będzie wykonywał pewną szczególną akcję, w powyższym przypadku generację sygnałów kontrolera pamięci, a następnie przechodził do kolejnego stanu.

Sekwencer ma 6 stanów, tak jak pokazano na rys. 6-3, które mogą być zdefiniowane od 0 do 5 lub zdefiniowane w VHDL jako S0 do S5. W tym przypadku założmy, że stany będą binarne wyrażone explicitly przez liczby od 0 do 5

Podstawową zasadą w przykładzie 6-1 jest to, że dla zdefiniowanych stanów sygnałów każda wartość stanu jest testowana (poprzez wyrażenie CASE) a następnie wprowadzany jest stan następny.

```
Signal Clock, Reset :Std_Ulogic;
Signal Trigger,Bus_en, Data_en, Wr_data:Std_Ulogic;
Signal State: Integer range 0 to 5;
```

```
Begin
Bus_cont: Process(Clock, Reset)
Begin
If reset='1' then
State<=0;
Bus_en<='0';
Data_en<='0';
Wr_data<='0';
Elsif Rising_Edge(Clock) then
Case State is - w jakim jest stanie?
When 0 =>
Bus_en <= after 5ns;
State <= 1;
When 1 =>
Data_en <= '1' after 5ns;
State <= 2;
When 2 =>
```

```

        Wr_data <= '1' after 5ns;
        State <= 3;
    When 3 =>
        Wr_data <= '0' after 5ns;
        State <= 4;
    When 4 =>
        Data_en <= '0' after 5ns;
        State <= 5;
    When 5 =>
        Bus_en <= '0' after 5ns;
        State <= 0;
    End Case;
End If;
End Process Bus_cont;

```

Przykład 6-1. Kod VHDL sekwencera bez rozgałęzień.

Powyższy przykład jest bardzo prostym sekwencerem bez rozgałęzień (bezw warunkowym), który działa ciągle tzn. po stanie 5 jest stan 0.

Niektóre narzędzia syntezy pozwalają na stosowanie prostszej techniki dla tego typu sekwencerów. Jest to wykonywane przez użycie procesu z „zegarem wielokrotnym”, w którym poszczególne stany nie są definiowane bezpośrednio.

21.9. Proces z wielokrotnym zegarem

Główną różnicą w stosunku do standardowego procesu jest to, że proces nie ma listy czułości a w zamian, poprzez stosowanie wielokrotnych wyrażeń WAIT, testowane są sygnały zegarowe. Wewnątrz procesu użyta jest pętla, sygnał „resetu” jest badany dla każdego sygnału zegarowego. W przypadku pojawienia się sygnału „resetu” wyrażenie NEXT wraca do początku pętli i zeruje logikę. Użycie tej metody daje następujący kod VHDL:

```

Signal Clock, Reset :Std_Ulogic;
Signal Trigger, Bus_en, Data_en, Wr_data :Std_Ulogic;
Signal State :Integer Range 0 to 5;

Begin

Bus_Cont: Process

Begin
Reset_loop:Loop
    If Reset='1' then
        State <=0;
        Bus_en <='0';
        Data_en <='0';
        Wr_data <='0';
    End IF;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';
    Bus_en<='1' after 5ns;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';
    Data_en<='1' after 5ns;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';

```

```

    Wr_data<='1' after 5ns;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';
    Wr_data<='0' after 5ns;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';
    Data_en<='0' after 5ns;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';
    Bus_en<='0' after 5ns;
    Wait until Rising_Edge(Clock); Next Reset_loop when Reset='1';

    End Loop Reset_loop;
    End process Bus_cont;

```

Przykład 6-2. Sekwencer z wielokrotnym zegarem.

Należy zauważyć, że sekwencer jest, w powyższym przykładzie, utworzony przez wielokrotne użycie wyrażenia WAIT UNTIL Clock i wyrażenia NEXT ale nie ma wyraźnie wyszczególnionej maszyny stanów. Takie podejście umożliwia syntezerowi realizację układu minimalnego. Powyższy kod ma „reset” synchroniczny bo jest testowany przy każdym narastającym zboczach sygnału zegara. Jedną z możliwych modyfikacji (dla wprowadzenia „resetu” asynchronicznego) jest poniższy fragment kodu VHDL:

```

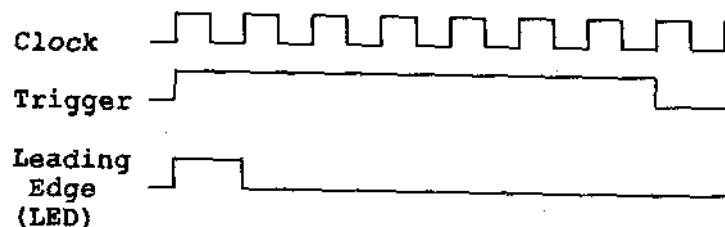
    Wait until Rising_Edge(Clock) or Reset='1';
    Next Reset_loop when Reset='1';

```

Wielokrotny zegar nie jest możliwy do zastosowania w każdym zestawie narzędzi VHDL, ale dostarcza wygodnego sposobu opisu maszyny stanów.

21.10. Detektor zbocza narastającego

Przy okazji przejścia do przykładów sekwencerów z rozgałęzieniami prześledźmy przykład z poprzedniego rozdziału - detektor zbocza narastającego. Jest on przedstawiony na rys. 6-4. Pożądanym wyjściem jest sygnał o długości jednego cyklu zegarowego pojawiającym się po narastającym zboczach sygnału wejściowego Trigger. Funkcja taka może być traktowana jako dwustanowa - stan pierwszy, który oczekuje na pojawienie się zbocza i kiedy się on pojawi ustawiający wyjście LED oraz stan drugi, który ustawia LED do stanu niskiego i następnie oczekujący na zmianę do stanu niskiego wejścia Trigger.

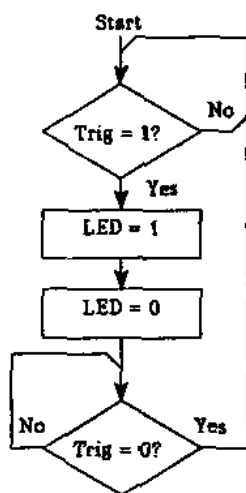


Rys. 6-4. Przebiegi sygnałów detektora zbocza narastającego.

Działanie rozpatrywanej maszyny stanów zostało przedstawione na rysunkach 6-5 oraz 6-6. Należy zauważyć (rys. 6-5) dwa węzły oznaczające sterowanie przepływem funkcji oraz dwa bloki oznaczające działanie akcji. Wiadomo, że każdy z bloków podejmuje akcje równocześnie z pojawiającym się zboczem zegara, przy czym każdy z innym zboczem.

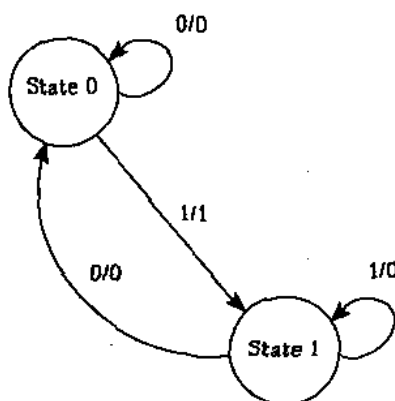
Wielu projektantów woli używać wykresu stanów do opisu funkcji logicznych, które przedstawiają działanie układu w innej formie. W takim przypadku projektant bezpośrednio identyfikuje stany logiczne, warunki które powodują zmianę stanu oraz wyjście dla każdego ze stanów. Wykres stanów detektora zbocza narastającego jest przedstawiony na rys. 6-6. W tym przypadku oba stany są reprezentowane przez nazwy State0, i State1. Dodatkowo założono synchroniczne działanie układu. Kiedy układ jest w stanie State0, jeśli Trigg jest niski wówczas stan nie zmienia się. Gdy Trigg zmienia stan na wysoki wówczas LED zmienia się na stan wysoki a układ przechodzi do stanu State1. Wówczas działania są wykonywane w stanie wysokim.

State 0 -stan kiedy układ oczekuje na pojawienie się zbocza narastającego sygnału Trigg,
State 1-stan kiedy układ oczekuje na zmianę sygnału Trigg na stan niski.



Rys. 6-5. Schemat przejść detektora zbocza narastającego.

TRIG Input / LED Output :



Rys. 6-6. Wykres stanów detektora zbocza narastającego.

Bazując na schemacie przejść detektora zbocza dodatniego z rys. 6-5 łatwo można stworzyć odpowiadający kod VHDL, który zawarty jest w przykładzie 6-3.

```

Entity LEDDET Is
  Port(Clock    : In Std_Ulogic;
        Reset   : In Std_Ulogic;
  
```

```

        Trigger    : In Std_Ulogic;
        LED        : Out Std_Ulogic);
End LEDDET;
Architecture RTL of LEDDET is
Signal State : Std_Ulogic;
Begin
    LED_EX:Process(Clock, Reset)
    Begin
        If Reset='1' then
            LED<='0';
            State<='0';
        Elsif Rising_Edge(Clock) then
            Case State is -- w jakim jestem stanie?
                When '0'=>
                    If Trigger='1' then
                        LED<='1' after 5ns;
                        State<='0';
                    End If;
                When '1'=>
                    LED<='0' after 5ns
                    If Trigger='0' then
                        State<='0';
                    End If;
                When Others=> null;
            End Case;
        End If;
    End Process LED_EX;
End RTL;

```

Przykład 6-3. Kod VHDL dla maszyny stanów detektora zbocza narastającego.

Sygnal LED jest opóźniony o 5nsek i jest to używane tylko podczas symulacji, podczas syntezy jest to ignorowane.

W przypadku bardziej złożonych sekwencerów korzystne jest zdefiniowanie możliwych stanów przy użyciu typu wyliczeniowego, np. w postaci:

```
Type Seq_state is (S0, S1);
```

Następnie można zdefiniować sygnał nazwany STATE używający tego typu:

```
Signal STATE: Seq_state;
```

Wynikająca z tego architektura jest przedstawiona w przykładzie 6-4. Należy zauważyć, że teraz testowane są stany S0 oraz S1, które to stany mogą zostać przez syntezer zrealizowane w dogodnej formie sprzętowej, dla bardziej wydajnych struktur logicznych.

```

Entity LEDDET Is
    Port( Clock : In Std_Ulogic;
          Reset : In Std_Ulogic;
          Trigger: In Std_Ulogic;
          LED : Out Std_Ulogic);
End LEDDET;
Architecture RTL of LEDDET is

Type Seq_state is (S0, S1);
Signal State: Seq_state;

```

```

Begin
  LED_EX:Process(Clock, Reset)
  Begin
    If Reset='1' then
      LED<='0';
      State<='0';
    Elsif Rising_Edge(Clock) then
      Case State is -- w jakim jestem stanie?
        When S0=>
          If Trigger='1' then
            LED<='1' after 5ns;
            State<='0';
          End If;
        When S1=>
          LED<='0' AFTER 5ns;
          If Trigger='0' then
            State<='0';
          End If;
        End Case;
      End If;
    End Process LED_EX;
  End RTL;

```

Przykład 6-4. Stany S0 i S1 w detektorze zbocza narastającego.

W ogólności warunkowe rozgałęzianie się sekwencerów jest implementowane w VHDL poprzez wyrażenia CASE oraz IF, podczas gdy sekwencery bezwarunkowe są realizowane poprzez przypisanie sygnałów.

21.11. Generator impulsów wyzwalany wielokrotnie

Ten przykład jest maszyną stanów będącą generatorem impulsów wyzwalanych wielokrotnie. Jego pożądane działanie przedstawione jest na rys. 6-7. Podobny generator został zaprojektowany w poprzednim rozdziale stosując technikę detektora zbocza sygnału. Podstawowy problem generacji impulsów o ustalonej długości może zostać także rozwiązany przez maszynę stanów gdzie wejściami są: Clock, Reset i Trigger, oraz sygnały wewnętrzne dekodujące koniec impulsu. Załóżmy, że jeśli drugi impuls Trigger pojawi się w tym samym czasie co koniec impulsu wówczas następuje ponowne przedłużenie trwania impulsu.

Układ wykorzystujący wyrażenia warunkowe z przerzutnikami został zaprezentowany w rozdziale poprzednim. Równoważna wersja z użyciem maszyny stanów wymaga oddzielnych stanów dla detekcji przejścia na wejściu Trigger oraz oddzielnych stanów dla zerowania licznika i przejścia wyjścia impulsu. Wykres stanów jest pokazany na rys. 6-8. Jest tam zdefiniowanych 5 różnych stanów, dlatego syntezer logiczny użyje przynajmniej 3 przerzutników.

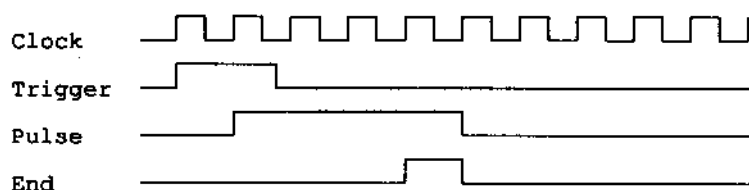
Stosownie do diagramu stanów z rys. 6-8 „reset” ustawia stan IDLE. W tym momencie licznik jest wyzerowany a wyjście jest w stanie niskim. Wejście Trigger jest monitorowane i nic nie dzieje się do czasu zmiany Trigger do stanu wysokiego. Kiedy Trigger='1' wówczas sekwencer przechodzi do stanu Gen_Pulse_A. To powoduje zmianę stanu wyjścia do wysokiego i zezwala licznikowi na zwiększanie zawartości. Ten stan czeka na jedno z dwóch następujących zdarzeń: zmiana poziomu wejścia Trigger lub osiągnięcie predefiniowanego stanu licznika (width). Dopóki żadne z tych zdarzeń nie ma

miejsca sekwencer pozostaje w stanie Gen_Pulse_A a licznik jest zwiększany. Jest to uwidocznione na rys. 6-8 przez strzałkę zaczynającą się i kończącą w okręgu reprezentującym stan Gen_Pulse_A.

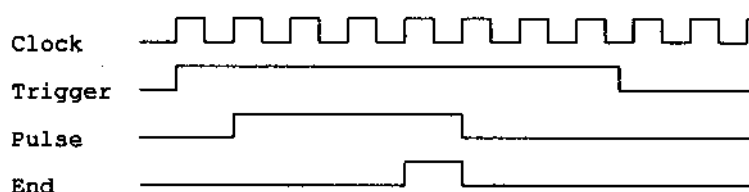
Rozważmy co się stanie gdy licznik osiągnie width. Wówczas sekwencer przechodzi do stanu End_Pulse. W tym momencie wyjście jest ustawione do stanu niskiego. Sekwencer pozostaje w tym stanie aż do Trigger='0', co zapobiega powtórnemu wyzwoleniu jeśli czas trwania impulsu Trigger jest dłuższy niż generowany na wyjściu. Kiedy w końcu Trigger='0' sekwencer powraca do stanu Idle i oczekuje na ponowne wyzwolenie.

W przypadku gdy $\text{count} < \text{width}$ a Trigger zmienia stan na '0' wtedy ze stanu Gen_Pulse_A przechodzi do stanu Gen_Pulse_B. Podczas tego stanu licznik zwiększa zawartość a sekwencer oczekuje na : powtórne wyzwolenie Trigger='1' lub osiągnięcie przez licznik stanu $\text{count} = \text{width}$. W drugim przypadku maszyna stanów przechodzi do stanu Idle. Sygnał wyjściowy jest ustawiany do stanu niskiego a cały układ czeka na zbocze sygnału Trigger. W przeciwnym wypadku tzn. gdy nastąpi powtórne wyzwolenie Trigger='1' sekwencer przejdzie do stanu Retrigger, który zeruje licznik a następnie przechodzi do stanu Gen_Pulse_A. Sekwencer znowu czeka na warunek $\text{count} = \text{width}$ lub zmianę na wejściu Trigger.

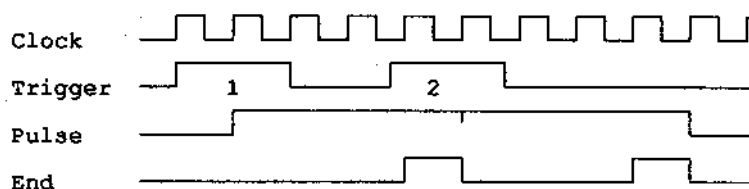
Case A. (Short trigger-width)



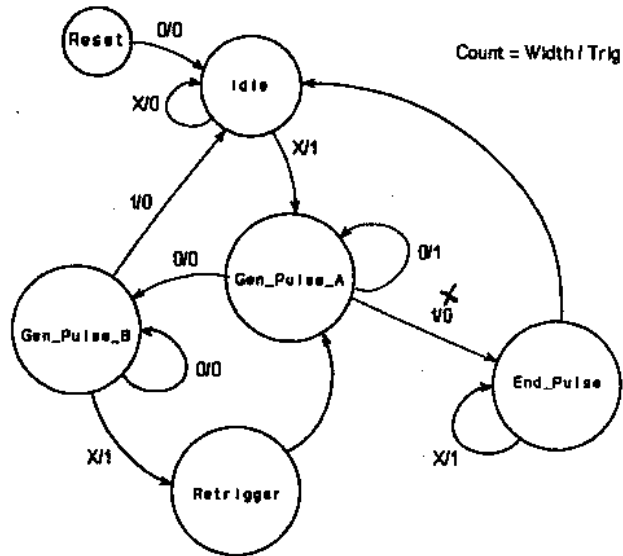
Case B. (Trigger-width independent)



Case C. Retriggering (ie, extend pulse after each trigger)



Rys. 6-7. Przebiegi czasowe generatora impulsów wyzwalanego wielokrotnie.



Rys. 6-8. Wykres stanów generatora impulsów wyzwalanego wielokrotnie.

Kod VHDL opisujący wyzwalany wielokrotnie generator impulsów w wersji maszyny stanów przedstawiony jest w przykładzie 6-5. Blok Entity jest nazywany PULSEGEN i ma trzy wejścia: TRIG, CLK i RESET oraz wyjście nazwane PULSE.

```
-- Generator impulsów wyzwalany wielokrotnie
library IEEE;
use IEEE.std_logic_1164.all
Entity PULSEGEN Is
  Port ( Clk :In Std_Ulogic;
         RESET :In Std_Ulogic;
         TRIG :In Std_Ulogic;
         PULSE :Out Std_Ulogic);
End PULSEGEN;
```

Architecture STATE_MACHINE of PULSEGEN is

```
-- definicja typów
Type PULSEGEN_STATE_TYPE is
  (Idle,Gen_Pulse_A,Gen_Pulse_B,End_Pulse,Retrigger);
Signal Current_State, Next_State: PULSEGEN_STATE_TYPE;
Signal Count: Integer range 0 to 31;
Constant Width: Integer range 0 to 31:=5;
```

Begin

```
-- maszyna stanów
```

```
State_Mach_Proc:Process (CURRENT_STATE,TRIG,COUNT)
```

```
Begin
```

```
Case CURRENT_STATE is
```

```
  when Idle => if TRIG='1' then
                  Next_State<=Gen_Pulse_A;
```

```
                Else
```

```
                  Next_State<=Idle;
```

```
                End If;
```

```
  when Gen_Pulse_A=> if COUNT=WIDTH then
                      Next_State<=End_Pulse;
```

```

        Elsif TRIG='0' then
            Next_State<=Gen_Pulse_B;
        Else
            Next_State<=Gen_Pulse_A;
        End If;
    when End_Pulse=> if TRIG='0' then
        Next_State<=Idle;
    Else
        Next_State<=End_Pulse;
    End If;
    when Gen_Pulse_B=> if TRIG='1' then
        Next_State<=Retrigger;
    Elsif COUNT=WIDTH then
        Next_State<=Idle;
    Else
        Next_State<=Gen_Pulse_B;
    End If;
    when Retrigger=> Next_State<=Gen_Pulse_A;
End Case;
End Process State_Mach_Proc;
Pulse_PROC: Process(CLK,RESET)
Begin
    IF RESET='1' then
        PULSE<='0';
        COUNT<=0;
        Current_State<=Idle;
    Elsif Rising_Edge(CLK) then
-- zapis obecnego stanu
        Current_State<=Next_State
-- ustawienie stanów wyjść
        Case Next_State is
            when Idle=>
PULSE<='0';
                COUNT<=0;
            when Gen_Pulse_A=>
PULSE<='1';
                COUNT<=COUNT+1;
            when End_Pulse=>
PULSE<='0';
                COUNT<=0;
            when Gen_Pulse_B=>
PULSE<='1';
                COUNT<=COUNT+1;
            when Retrigger=>
                COUNT<=0;
        End Case;
    End If;
End Process PULSE_GEN;
End STATE_MACHINE;

```

Przykład 6-5. Generator impulsów-wersja oparta na maszynie stanów.

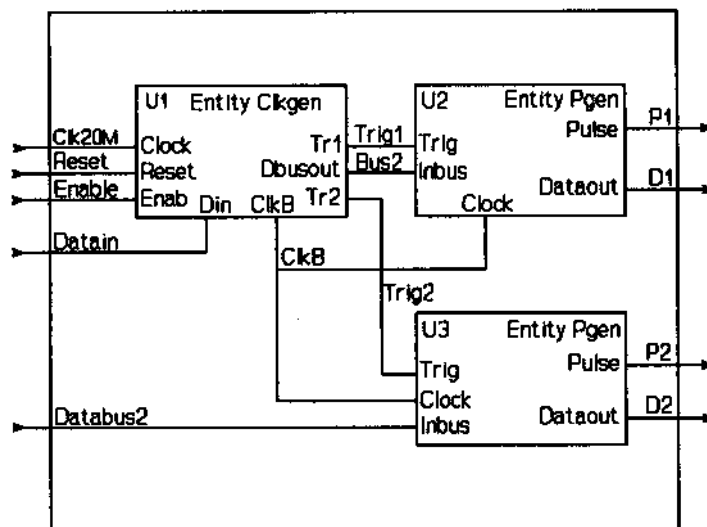
Wnętrze architektury STATE_MACHINE zawiera dwa procesy. Proces STATE_MACHINE_PROC jest procesem kombinacyjnym. Bada on stany sygnałów

CURRENT_STATE, TRIG oraz COUNT w celu określenia stanu następnego. Proces PULSE_GEN jest zegarowany. Warunek na „reset” inicjalizuje wartości wyjścia i sygnałów wewnętrznych.

22. Opis na poziomie strukturalnym

Rozważmy schemat większego układu który składa się z kilku bloków jak jest to przedstawione na rys. 9-1.

W projekcie występują dwa różne bloki ENTITY nazwane CLKGEN oraz PGEN oznaczone jako składniki U1, U2 i U3. Należy zauważyć, że nazwy sygnałów na zewnątrz bloków ENTITY mogą mieć inne nazwy niż używane wewnątrz nazwy zadeklarowane wyrażeniem PORT.



Rys. 9-1. Schemat blokowy większego projektu.

Zakładając, że te dwa bloki ENTITY zostały wcześniej zaprojektowane deklaracje ich mogą być następujące:

```
ENTITY Clkgen Is
  Port(
    Clock      :In Std_Ulogic;
    Reset      :In Std_Ulogic;
    Enab       :In Std_Ulogic;
    Din        :In Std_Ulogic_Vector(15 downto 0);
    ClkB       :Out Std_Ulogic;
    Tr1        :Out Std_Ulogic;
    Tr2        :Out Std_Ulogic;
    Dbusout    :Out Std_Ulogic_Vector(15 downto 0)
  );
END Clkgen;

ENTITY Pgen Is
  Port(
    Clock      :In Std_Ulogic;
    Reset      :In Std_Ulogic;
    Trig       :In Std_Ulogic;
    Inbus      :In Std_Ulogic_Vector(15 downto 0);
  );
END Pgen;
```

```

    Pulse      :Out   Std_Ulogic;
    Dataout     :Out   Std_Ulogic_Vector(7 downto 0)
  );
END Pgen;

```

Cały projekt na najwyższym poziomie hierarchii składa się z innego bloku ENTITY nazwanego Top_gen którego porty I/O są przedstawione na rys. 9-1.

```

ENTITY Top_gen Is
  Port(
    Clk20M      :In Std_Ulogic;
    Reset       :In Std_Ulogic;
    Enable      :In Std_Ulogic;
    Datain      :In Std_Ulogic_Vector(15 downto 0);
    Databus2    :In Std_Ulogic_Vector(15 downto 0);
    P1          :Out   Std_Ulogic;
    D1          :Out   Std_Ulogic_Vector(7 downto 0);
    P2          :Out   Std_Ulogic;
    D2          :Out   Std_Ulogic_Vector(7 downto 0)
  );
END Top_gen;

```

Architektura bloku Top_gen będzie typu strukturalnego, który określa połączenia pomiędzy składnikami - blokami ENTITY. Architektura składa się z dwóch części. Sekcja deklaracji zawiera:

- deklarację sygnałów (Signal declaration) np. dla sygnałów takich jak Trig2 który jest sygnałem wewnętrznym dla Top_gen,
- deklarację składników (Component declaration) dla użytych bloków ENTITY,
- deklarację stałych.
- sekcję deklarującą połączenia pomiędzy zamiennikami.

Sekcja połączeń zamienników (Instantiation) zawiera listę połączeń pomiędzy poszczególnymi składnikami.

Architektura bloku Top_gen będzie następująca:

Architecture Schematic of Top_gen Is

```

Signal Trig1      :Std_Ulogic;
Signal Trig2      :Std_Ulogic;
Signal ClkB       :Std_Ulogic;
Signal Bus2       :Std_Ulogic_Vector(15 downto 0);

```

Deklaracja składników dla każdego bloku ENTITY używanego w Top_gen po prostu odnosi się do nazwy bloku ENTITY oraz jego sygnałów I/O.

```

Component Clkgen
  Port(
    Clock      :In Std_Ulogic;
    Reset      :In Std_Ulogic;
    Enab       :In Std_Ulogic;
    Din        :In Std_Ulogic_Vector(15 downto 0);
    ClkB       :Out   Std_Ulogic;
    Tr1        :Out   Std_Ulogic;
    Tr2        :Out   Std_Ulogic;

```

```

        Dbusout      :Out   Std_Ulogic_Vector(15 downto 0)
    );
END Component; --Clkgen

Component Pgen
Port(
    Clock      :In Std_Ulogic;
    Reset      :In Std_Ulogic;
    Trig       :In Std_Ulogic;
    Inbus      :In Std_Ulogic_Vector(15 downto 0);
    Pulse      :Out   Std_Ulogic;
    Dataout     :Out   Std_Ulogic_Vector(7 downto 0)
);
END Component; --Pgen

```

Nazwa zamiennika lub jego etykieta dla każdego układu takiego jak U1, U2 itp. Zamiennik bloku Entity jest podobny do wielokrotnego pojawiania się pewnego elementu na schemacie elektrycznym, gdzie każdy element ma swój identyfikator oraz sygnały które łączą się z sygnałami zewnętrznymi wg list nazywanej Port Map. Możliwość mapowania nazw sygnałów do portów I/O komponentów umożliwia wielokrotne użycie tego samego typu komponentu. Składnia połączenia sygnałów na liście Port Map jest następująca:

Nazwa portu w bloku ENTITY=>Nazwa sygnału na schemacie

Wyrażenie BEGIN wskazuje początek opisu architektury.

Begin

```

U1:Clkgen
Port Map(
    Clock =>CLK20M, -- mapowanie przez nazwę
    Reset =>Reset,
    Enab  =>Enable,
    Din   =>Datain,
    ClkB  =>ClkB,
    Tr1   =>Trig1,
    Tr2   =>Trig2,
    Dbusout =>Bus2
); --koniec Port Map
U2:Pgen
Port Map(
    Clock =>ClkB,
    Reset =>Reset,
    Trig  =>Trig1,
    Inbus =>Bus2,
    Pulse =>P1,
    Dataout =>D1
); --koniec Port Map
U3:Pgen
Port Map(
    Clock =>ClkB,
    Reset =>Reset,
    Trig  =>Trig2,

```

```

Inbus =>Databus2,
Pulse =>P2,
Dataout =>D2
); --koniec Port Map

```

Składnia typu *Nazwa portu w bloku ENTITY=>Nazwa sygnału na schemacie* jest nazywana Mapowaniem z nazwą (ang. Named Port Mapping) i ma tą zaletę, że lista sygnałów nie zależy od ich uporządkowania. Można również stosować mapowanie pozycyjne jak w poniższym przykładzie:

```

U3:Pgen
Port Map(
    Clock, -- mapowanie pozycyjne
    Reset,
    Trig,
    Inbus,
    Pulse,
    Dataout
); --koniec Port Map

```

gdzie kolejność sygnałów musi być taka sama jak w liście deklaracji bloku ENTITY.

Łącząc wszystkie poprzednie sekcje razem kompletny opis projektu z rys. 9-1 może być następujący:

```

ENTITY Top_gen Is
Port(
    Clk20M :In Std_Ulogic;
    Reset  :In Std_Ulogic;
    Enable :In Std_Ulogic;
    Datain  :In Std_Ulogic_Vector(15 downto 0);
    Databus2 :In Std_Ulogic_Vector(15 downto 0);
    P1      :Out Std_Ulogic;
    D1      :Out Std_Ulogic_Vector(7 downto 0);
    P2      :Out Std_Ulogic;
    D2      :Out Std_Ulogic_Vector(7 downto 0)
);
END Top_gen;

```

Architecture Schematic of Top_gen Is

```

Signal Trig1 :Std_Ulogic;
Signal Trig2 :Std_Ulogic;
Signal ClkB  :Std_Ulogic;
Signal Bus2  :Std_Ulogic_Vector(15 downto 0);

```

```

Component Clkgen
Port(
    Clock :In Std_Ulogic;
    Reset :In Std_Ulogic;
    Enab   :In Std_Ulogic;
    Din    :In Std_Ulogic_Vector(15 downto 0);
    ClkB   :Out Std_Ulogic;
    Tr1    :Out Std_Ulogic;

```

```

        Tr2      :Out   Std_Ulogic;
        Dbusout   :Out   Std_Ulogic_Vector(15 downto 0)
    );
END Component; --Clkgen

Component Pgen
Port(
    Clock   :In Std_Ulogic;
    Reset   :In Std_Ulogic;
    Trig    :In Std_Ulogic;
    Inbus   :In Std_Ulogic_Vector(15 downto 0);
    Pulse   :Out   Std_Ulogic;
    Dataout  :Out   Std_Ulogic_Vector(7 downto 0)
);
END Component; --Pgen

Begin
U1:Clkgen
Port Map(
    Clock =>CLK20M,
    Reset =>Reset,
    Enab   =>Enable,
    Din    =>Datain,
    ClkB   =>ClkB,
    Tr1    =>Trig1,
    Tr2    =>Trig2,
    Dbusout =>Bus2
); --koniec Port Map
U2:Pgen
Port Map(
    Clock =>ClkB,
    Reset =>Reset,
    Trig  =>Trig1,
    Inbus =>Bus2,
    Pulse =>P1,
    Dataout =>D1
); --koniec Port Map
U3:Pgen
Port Map(
    Clock =>ClkB,
    Reset =>Reset,
    Trig  =>Trig2,
    Inbus =>Databus2,
    Pulse =>P2,
    Dataout =>D2
); --koniec Port Map
End Schematic; --architektura bloku Top_gen

```

22.1. Projektowanie hierarchiczne

Blok ENTITY taki jak z powyższego przykładu Top_gen może reprezentować cały układ ASIC lub też funkcjonalny podblok całego projektu. Podczas fazy projektowej można symulować pojedyncze bloki ENTITY jak również funkcjonalne grupy bloków i ewentualnie cały projekt układu ASIC. Składnia strukturalna może być użyta do połączeń pomiędzy

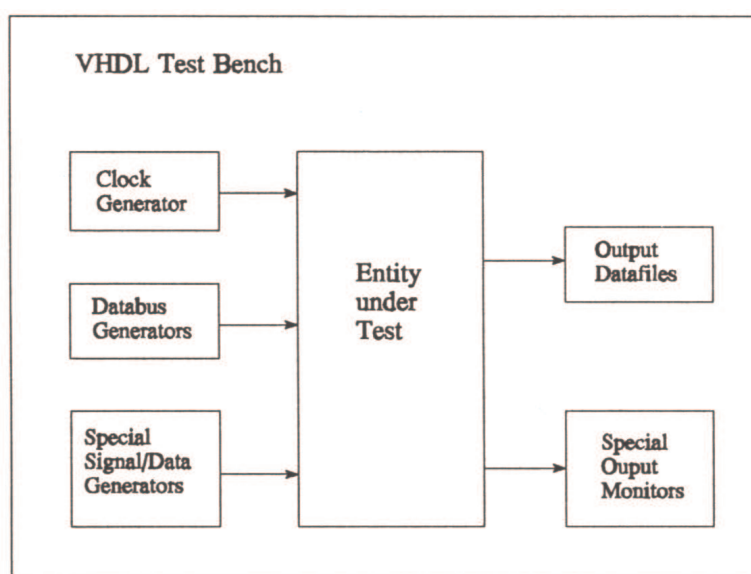
pojedynczymi blokami ENTITY. Jak można zobaczyć w powyższym przykładzie w przypadku używania zbyt wielu podbloków mamy pewien nadmiar stosowania składni VHDL. Z drugiej strony (patrz rozdział. 2) rozmiar bloku ENTITY musi być ograniczony do kilku tysięcy bramek ze względu na proces syntezy i symulacji.

23. Symulacje i testowanie

W ciągu procesu projektowego jednostki projektowe ENTITY są testowane na poziomie RTL z użyciem symulatora VHDL. Jak w każdym innym symulatorze potrzebny jest zarówno opis badanego układu jak i wejściowych sygnałów pobudzających łącznie z procedurą badania wyników symulacji. W środowisku VHDL sygnały wejściowe do testowanych bloków ENTITY są wytwarzane w innych blokach ENTITY nazywanych TEST BENCH. Pomysł TEST BENCH jest powszechnie stosowany w symulacji, testowaniu i dokumentacji systemów cyfrowych zarówno w VHDL jak i innych językach projektowych. Para bloków ENTITY opisujących badany system oraz wektory sygnałów testowych stanowi podstawę do projektowania i dokumentacji typu TOP-DOWN. Oddzielne bloki TEST BENCH mogą być stworzone dla poszczególnych podbloków systemu cyfrowego jak i dla kompletnego układu ASIC.

Rys. 9-2 przedstawia TEST BENCH w którym generowane są sygnały do bloku objętego testowaniem jak również prowadzone jest badanie sygnałów wyjściowych i węzłów wewnętrznych badanego układu. Należy zauważyć, że kompletny TEST BENCH jest blokiem ENTITY nie zawierającym PORTÓW zewnętrznych.

Z punktu widzenia VHDL TEST BENCH jest opisem hierarchicznym w którym badany blok połączony jest z blokami generującymi i monitorującymi sygnały zewnętrzne. Organizacja TEST BENCHu jako oddzielnych bloków testowanego oraz generującego/monitorującego wynika z łatwej syntezy tak opisanego projektu poprzez odrzucenie części niesynteżowalnej. Dodatkową zaletą takiego podejścia jest opis całości w jednym środowisku – VHDL. Wyniki symulacji mogą zawierać przebieg czasowy stanów logicznych, wiadomości monitorujące spełnienie określonych warunków oraz sterować inne bloki projektowe. Niektóre symulatory VHDL umożliwiają również graficzną prezentację wyników symulacji.



Rys. 9-2. Podstawowy TEST BENCH VHDL.

Jako przykład wykonamy opis środowiska testowego TEST BENCH układu z rys. 9-1. Wejście zawiera sygnał zegara 20MHz, główny RESET, sygnał Enable oraz dwie szyny danych.

23.1. Struktura TEST BENCH

Blok ENTITY opisujący TEST BENCH nie zawiera sygnałów zewnętrznych, jego deklaracja jest następująca:

```
ENTITY TB_Top_gen IS
  End TB_Top_gen;
```

Architektura zazwyczaj składa się z deklaracji sygnałów, komponentów oraz generatorów pobudzających:

1. Lista sygnałów łączących testowane bloki ENTITY. Jest to zwykła deklaracja sygnałów, można opcjonalnie przypisać początkowy stan sygnału używając operatora **:=wartość**

```
SIGNAL Clk20M      : Std_Ulogic:= '1'; - wartość początkowa
SIGNAL Reset       : Std_Ulogic;
SIGNAL Enable      : Std_Ulogic;
SIGNAL Datain      : Std_Ulogic_Vector(15 downto 0);
SIGNAL Databus2    : Std_Ulogic_Vector(15 downto 0)
SIGNAL P1          : Std_Ulogic;
SIGNAL D1          : Std_Ulogic_Vector(7 downto 0);
SIGNAL P2          : Std_Ulogic;
SIGNAL D2          : Std_Ulogic_Vector(7 downto 0);
```

W przypadku gdy testowany jest tylko jeden blok ENTITY, wygodnie jest nadać nazwy sygnałów tak samo jak ich odpowiedników wewnątrz badanego bloku. Podczas symulacji sygnały którym nie nadano wartości początkowej przyjmują wartości najbardziej na lewo danego typu. Np. dla sygnałów zadeklarowanych jako Std_Ulogic jest to stan „U” – interpretowany jako niezainicjowany. Dobrą praktyką jest inicjowanie wartości sygnałów generowanych w TEST BENCHach do np. '0' lub '1', gdyż projektowany blok sam w sobie będzie zawierał sygnały 'U' aż do pojawienia się pierwszego RESETU lub sygnału zegara..

2. Deklaracja komponentów. Deklaracja bloku testowanego:

```
COMPONENT Top_gen
  PORT(
    SIGNAL Clk20M      :IN Std_Ulogic:= '1'; - wartość początkowa
    SIGNAL Reset       :IN Std_Ulogic;
    SIGNAL Enable      :IN Std_Ulogic;
    SIGNAL Datain      :IN Std_Ulogic_Vector(15 downto 0);
    SIGNAL Databus2    :IN Std_Ulogic_Vector(15 downto 0)
    SIGNAL P1          :OUT Std_Ulogic;
    SIGNAL D1          :OUT Std_Ulogic_Vector(7 downto 0);
    SIGNAL P2          :OUT Std_Ulogic;
    SIGNAL D2          :OUT Std_Ulogic_Vector(7 downto 0)
  ); -- END OF PORT
END COMPONENT;
```

3. Początek bloku architektury i wstawienie badanego bloku

```
U1: Top_gen
```

```

PORT MAP(
    Clk20M => Clk20M,  -- mapowanie bezpośrednie
    Reset,             -- mapowanie pozycyjne
    Enable,
    Datain,
    Databus2,
    P1,
    D1,
    P2,
    D2,
); -- END OF PORT MAP
END COMPONENT;

```

4. Kod VHDL generujący sygnały sterujące. Jak widać na rys. 9-1 sygnały mogą być pojedyncze jak i wielokrotne w postaci szyn sygnałów. Sygnały dyskretne mogą zawierać przebiegi okresowe jak np. zegar, pojedyncze impulsy jak np. reset, czy sygnały zezwalające Enable, lub też ciągi bitów o nieokresowym charakterze. Szyny danych mogą przyjmować zarówno proste zera czy jedynki jak i być ciągiem złożonych słów. W wielu przypadkach sygnały te mogą być generowane przy użyciu kodu VHDL podczas gdy dla innych można sygnały generować przez modelowanie hardware'owe lub poprzez odczyt wartości z plików danych.

23.2. Generacja sygnałów

Jako przykłady rozważmy w jaki sposób można wygenerować kilka prostych typów sygnałów

23.2.1. Sygnały stałe:

Sygnały stałe mają stałe wartości podczas całego procesu symulacji, przykłady:

```

SigA<='1';
Data <= "11001";

```

23.2.2. Sygnały powtarzalne

Są to np. sygnały zegara lub inne przebiegi prostokątne, przykłady:

```

Clk20M <= NOT (Clk20M) after 25ns;

```

Zakładając, że Clk20M zostało zainicjowane jako '0' lub '1' sygnał jest po prostu negowany co każde 25 ns. Lepiej jest jednak zdefiniować stałą określającą okres zegara

```

CONSTANT CLK_PERIOD: Time:=50ns;

```

a następnie wygenerować zegar:

```

Clk20M <= NOT (Clk20M) after CLK_PERIOD/2;

```

23.2.3. Niesymetryczne sygnały powtarzalne

Aby wygenerować niesymetryczne sygnały powtarzalne należy użyć procesu oraz wyrażenia WAIT. Wygenerujemy przebieg o okresie 100us gdzie stan '1' pojawia się okresowo i trwa 1us

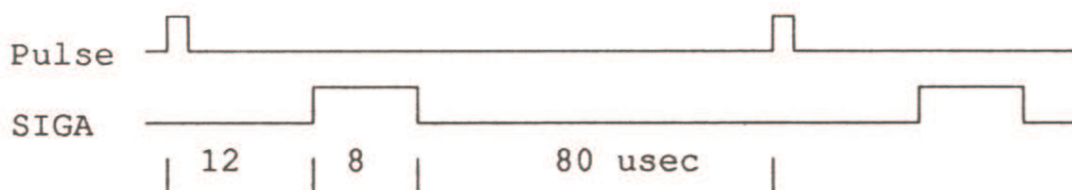
```
CONSTANT Zero_Time : Time:=99us;
```

```
Pulse1: Process
Begin
  PULSE <='1';
  WAIT FOR 1us;
  PULSE <='0';
  WAIT FOR Zero_Time;
End Pulse1;
```



Czas podany w wyrażeniu WAIT nie musi być liczbą całkowitą. Każdy niezależny ciąg impulsów z unikalnym okresem musi być generowany w oddzielnym procesie i może być całkowicie niesynchronizowany względem zegara. Można jednak wygenerować kilka sygnałów w jednym procesie jeśli są powiązane ze sobą w czasie i mają ten sam okres, np.:

```
Pulse1: Process
Begin
  PULSE <='1';
  WAIT FOR 1us;
  PULSE <='0';
  WAIT FOR 11us;
  SIGA <='1';
  WAIT FOR 8us;
  SIGA <='0';
  WAIT FOR 80us; -- całkowity okres wynosi 100us
End Pulse1;
```



Technika ta może być rozszerzona na wiele sygnałów ale jest czasochłonna dla wielu skomplikowanych sygnałów.

23.2.4. Proste sygnały niepowtarzalne

Proste sygnały niepowtarzalne występują tylko jednokrotnie (np. RESET)

```
CONSTANT Reset_Width : Time:=140ns;

Reset <= '0', '1' after Reset_Width; -- reset aktywowany
-- zerem
```

W powyższym przykładzie pierwsze przypisanie do wartości 0 logicznego pojawia się w czasie 0, następnie po czasie 140ns sygnał osiąga wartość '1'. Można rozszerzyć tą techniką dla sygnałów bardziej złożonych np.:

```
En<='1','0' after 27us,'1' after 78us,'Z' after 112us,'0' after 89ns;
```

Należy zauważyć, że 'Z' oznacza stan wysokiej impedancji.

23.2.5. Regularne wzory danych

Czasami potrzebne jest wysterowanie szyny danych sygnałem regularnym takim jak np.: wartości zwiększające się liniowo, funkcje arytmetyczne, bity przesuwające się itp. W poprzednich rozdziałach opisane zostały techniki opisu VHDL systemów cyfrowych przeznaczone do syntezy logicznej, opis dla celów testowania jest jednak prostszy. Rozważmy generację danych zwiększających się o jeden co 75ns

```
SIGNAL Databus : Std_Ulogic_Vector(7 downto 0):="00000000";  
  
Databus <= Databus+'1' after 75ns;
```

Funkcja powyższa realizowana jest w jednej linii a nie jak licznik w procesie zegarowanym. Dla przejrzystości kodu można wprowadzić stałą będącą okresem zmian:

```
SIGNAL Databus : Std_Ulogic_Vector(7 downto 0):="00000000";  
CONSTANT DelayTime : Time:=75ns  
  
Databus <= Databus+'1' after DelayTime;
```

Wzór kroczącej jedynki można uzyskać poprzez dodawanie wartości do samej siebie lub inaczej mnożenie przez 2:

```
Walk1: PROCESS  
Begin  
  DBUS<="00000001";  
  Loop1:  
    For i in 0 to DBUS'length loop  
      DBUS <= DBUS+DBUS;  
    End loop Loop1;  
End Walk1;
```

W powyższym przykładzie zastosowano atrybut 'LENGTH w ten sposób process automatycznie dostosowuje się do długości szyny danych DBUS.

23.2.6. Wzory danych odczytywane z tablic.

W wielu zastosowaniach dane wejściowe muszą przybierać pewne specyficzne wartości. Prostym sposobem do zapewnienia takiej możliwości jest stworzenie tablicy zawierającej pożądane dane a następnie odczytywanie jej zawartości. Poniższy przykład przedstawia tablicę zawierającą liczby typu integer oznaczające częstotliwości, które zostaną następnie wyprowadzone na wyjście jako sygnały DBUS (integer) oraz DBUS_V (Std_Ulogic_Vector)

Deklaracja typów i stałych może znajdować się zarówno w bloku ENTITY jak i w architekturze lub pakiecie zewnętrznym.

```
TYPE Freq is array (1 to 6) of Integer
CONSTANT Freqval:Freq:=(6820,
                        10230,
                        10231,
                        10232,
                        10233,
                        10234,
                        10235);

SIGNAL DBUS    : Integer;
SIGNAL DBUS_V  : Std_Ulogic_Vector(15 downto 0)
CONSTANT Bus_delay: Time:=75ns;

Begin
  Table_gen:PROCESS
  Begin
    Loop1: For i in 1 to 6 loop
      DBUS <= Freqval(i);
      DBUS_V <=To_SUV(Freqval(i));
      WAIT FOR Bus_delay;
    End loop Loop1;
  End PROCESS;
```

Należy zauważyć, że wartości typu integer zostają zamienione do Std_Ulogic_Vector za pomocą funkcji konwersji To_SUV. Kod VHDL tej funkcji jest przedstawiony w dalszej części rozdziału, chociaż często takie procedury zawarte są w pakiecie dostarczanym od producenta oprogramowania VHDL. Bez tej funkcji należałoby zadeklarować tablicę typu STD_ULOGIC_VECTOR, która mogłaby być trudniejsza do interpretacji w niektórych przypadkach.

Powyższa tablica jest dodatkowo ograniczona ustalonym jej rozmiarem. W przypadku potrzeby rozszerzenia rozmiaru tablicy należy również zwiększyć zakres pętli. W celu uniknięcia takich kłopotów można użyć tablicy o nieokreślonym rozmiarze, a kompilator VHDL powinien sam dobrać zakres pętli. W tym celu należy zmienić dwie następujące linie:

```
-- TYPE Freq is array (1 to 6) of Integer;
  TYPE Freq is array (natural range <>) of Integer
--Loop1: For i in 1 to 6 loop
Loop1: For i in Freqval'low to Freqval'high loop
```

Gdzie 'low i 'high są predefiniowanymi atrybutami niezbędnymi do obliczenia dolnej i górnej granicy tablicy Freqval.

23.2.7. Generowanie sygnałów zorientowane projektowo

Niektóre typy sygnałów muszą być generowane podobnie do tych użytych do projektowania logicznego. Na przykład trudno wygenerować ręcznie dane przebiegające w systemach komunikacyjnych. W takim przypadku dane są pobierane w słowach a wprowadzane w postaci szeregowej. Generowanie takich danych w VHDLu odbywa się w podobny sposób.

Założmy, że sygnał DataBus jest 16-to bitowy oraz częstotliwość przesunięć powinna wynosić 4MHz (250ns/bit).

```
SIGNAL DataBus : Std_Ulogic_Vector(15 downto 0);
SIGNAL Serial  : Std_Ulogic;
CONSTANT B_delay: Time:=250ns;

Begin
  Serial1:PROCESS
  Begin
    Loop1: For i in 0 to DataBus'High loop
      Serial<=DataBus(i);
      WAIT FOR _delay;
    End loop Loop1;
  End PROCESS;
```

Rozszerzając można użyć drugiej pętli dla generowania innego sygnału szeregowego. Możliwe jest także generowanie złożonych sygnałów zmiennych w czasie poprzez modyfikację opóźnień.

```
SIGNAL Delay_Time: Time:=3us;

-- NASTĘPNIE W PĘTLI:

SigA <= '1';
Wait for 100ns;
SigA <= '0';
Wait for Delay_Time;
Delay_Time<=Delay_Time+50ns;
IF Delay_Time=12us then
  Delay_Time<=3us;
End If;
```

Powyższy przykład przedstawia impuls którego czas przerwy zmienia się cyklicznie od 3us do 12us.

23.3. Instrukcja kontroli

Instrukcja kontroli służy do generacji wiadomości tekstowej podczas symulacji jeśli wystąpią pewnych zdarzenia logiczne. Instrukcja składa się z warunku, tekstu wiadomości oraz ważności komunikatu. Treść wiadomości oraz poziom ważności są przekazywane do terminala w przypadku gdy warunek wyrażenia nie jest spełniony, przykład:

```
ASSERT Pulse='0'
  Report „Pulse has gone high”
  Severity warning;
```

Typy ważności komunikatu mogą być Failure, Error, Warning i Note.

23.4. Kompletny przykład TEST BENCH

```
Library IEEE;
Use ieee.std_logic_1164.all;
```

```

Library Synth;
    Use synth.packages.all;

ENTITY TB_Top_gen IS
End TB_Top_gen;

Architecture Test_bench OF TB_Top IS

-- deklaracja sygnałów
SIGNAL Clk20M      : Std_Ulogic:='1'; -- wartość początkowa
SIGNAL Reset       : Std_Ulogic;
SIGNAL Enable      : Std_Ulogic;
SIGNAL Datain      : Std_Ulogic_Vector(15 downto 0);
SIGNAL Databus2    : Std_Ulogic_Vector(15 downto 0);
SIGNAL P1          : Std_Ulogic;
SIGNAL D1          : Std_Ulogic_Vector(7 downto 0);
SIGNAL P2          : Std_Ulogic;
SIGNAL D2          : Std_Ulogic_Vector(7 downto 0);

CONSTANT CLK_PERIOD: Time:=50ns;
CONSTANT Reset_Width: Time:=140ns;

-- deklaracja komponentów
COMPONENT Top_gen
    PORT(
        SIGNAL Clk20M      :IN  Std_Ulogic:='1'; -- wartość
                                                --początkowa
        SIGNAL Reset       :IN  Std_Ulogic;
        SIGNAL Enable      :IN  Std_Ulogic;
        SIGNAL Datain      :IN  Std_Ulogic_Vector(15 downto 0);
        SIGNAL Databus2    :IN  Std_Ulogic_Vector(15 downto 0);
        SIGNAL P1          :OUT  Std_Ulogic;
        SIGNAL D1          :OUT  Std_Ulogic_Vector(7 downto 0);
        SIGNAL P2          :OUT  Std_Ulogic;
        SIGNAL D2          :OUT  Std_Ulogic_Vector(7 downto 0)
    ); -- END OF PORT
END COMPONENT;

BEGIN

-- wstawienie komponentu
U1: Top_gen
    PORT MAP(
        Clk20M      => Clk20M, -- mapowanie bezpośrednie
        Reset,      -- mapowanie pozycyjne
        Enable,
        Datain,
        Databus2,
        P1,
        D1,
        P2,
        D2,
    ); -- END OF PORT MAP

TB: Block
Begin

```

```

Clk20M <= NOT (Clk20M) after CLK_PERIOD/2; -- zegar 20MHz
Reset <= '1' after 10ns, '0' after Reset_Width;
Enable <='1' after 520ns, '0' after 840ns, '1' after 3010ns;
-- uwaga: sygnał Enable nie został zainicjowany aż do czasu 520ns, jego
-- wartość do tego momentu wynosi 'U'

Datain <= "0000010100110010";
Databus2 <= "0000010100110010", "0010010100110010" after 2us;
End Block TB;
End Test_Bench;

```

24. Uwagi dotyczące kodu VHDL przeznaczonego do syntezy.

Opis strukturalny czy przypisania ciągle? Zarówno wstawienie komponentu jak i przypisania ciągle wewnątrz bloku architektury są synteżowalne, jednakże prowadzą do różnych wyników. Ogólnie mówiąc opis typu DATA FLOW jest bardziej abstrakcyjny niż strukturalny i trudniejszy do optymalizacji i implementacji. Jest to szczególnie ważne dla bloków arytmetycznych, które są często bardziej zwarte w przypadku syntezy z poziomu strukturalnego - korzystając z bibliotek producenta-dostawcy. Z drugiej strony używanie bibliotek producenta powoduje często trudności z przenoszeniem projektu do innych technologii i wymusza na projektancie wykonanie dodatkowych czynności.

IF czy CASE? Wyrażenia typu IF-THEN-ELSE są najczęściej synteżowane jako multipleksery z sygnałem sterującym zawartym w warunku instrukcji. Jednakże, jeżeli użyjemy zagnieżdżonych instrukcji IF-THEN-ELSE nie prowadzi to do wykorzystania dużego multipleksera ze złożonym słowem sterującym, zamiast tego użyty jest łańcuch małych multiplekserów mniej więcej oddający kolejność kodu VHDL. Wyrażenie CASE również prowadzi do syntezy multipleksera jednakże w tym przypadku będzie to jeden element niezależnie od liczby rozgałęzień. Z tego powodu jeśli chcemy mieć jeden duży multiplekser (prawdopodobnie szybki) powinniśmy użyć wyrażenia CASE. Jeśli jednakże prędkość działania układu nie jest kluczowa, użycie zagnieżdżonych instrukcji IF jest łatwiejsze do rozmieszczenia i trasowania (może dać mniejszą użytą powierzchnię).

IF czy wstawiony multiplekser? Wybór pomiędzy tymi opcjami jest podobny do wyboru z pkt. 1. Wstawione bloki multiplekserów są zazwyczaj bardziej zwarte po implementacji, jednakże równocześnie są zależne od producenta. Dodatkowo opis strukturalny jest dłuższy niż behawioralny, więc projektant musi spędzić więcej czasu na wykonanie opisu strukturalnego niż behawioralnego. Z drugiej strony synteza z opisu funkcjonalnego powoduje prostszy kod VHDL i ułatwia projektowanie powodując, że proces syntezy jest dużo szybszy. Jest również łatwiej zrozumieć i użyć ponownie fragment kodu. Korzyści te związane są z mniej optymalną implementacją.

Operacje arytmetyczne czy wstawienie komponentów? Chociaż w tym punkcie można zastosować obserwacje z pkt. poprzednich, może okazać się korzystne zastosowanie komponentów w miejsce opisu funkcjonalnego (mimo, że wygląda to jak cofanie się o jeden krok do tyłu). Powodem tego jest to, że operacje arytmetyczne takie jak mnożenie, dzielenie i modulo są bardzo kosztowne wyrażając je w liczbie koniecznych bramek logicznych do ich realizacji. Z tego powodu może okazać się niezwykle ważne aby wykorzystać te bloki jako możliwie najmniejsze.

Podział operacji. Większość operatorów jest dwuargumentowa i narzędzia syntezy logicznej biorą to pod uwagę. Stosownie do tego wielokrotne operacje takie jak $A+B+C+D$ prowadzą do syntezy jako sekwencja sumatorów tj. najpierw dodawane jest A do B następnie do tej sumy C i do tego wyniku D. W ten sposób stworzony jest łańcuch 3 sumatorów, a jeśli użyto by nawiasów $(A+B)+(C+D)$ łańcuch wynosiłby 2 i układ byłby szybszy. Również odpowiedni podział struktur regularnych może prowadzić do poprawy

właściwości czasowych jak i zajmowanej powierzchni implementowanego układu. Ogólną zasadą jest próbowanie wykorzystywania wszelkich regularności struktur.

Przypisywanie wartości do wszystkich wyjść we wszystkich warunkach. Stosownie do zasady - „Jeśli w procesie zegarowanym dla danego warunku nie specyfikuje się wartości sygnału oznacza to zapamiętanie jego poprzedniego stanu czyli w zsyntezowanym układzie wstawienie przerzutnika” można łatwo spowodować wstawienie niepotrzebnych przerzutników lub nawet doprowadzić do nieprawidłowej pracy układu.

25. Dodatek: rozwinięcie deklaracji ENTITY, pętle, konfiguracje, przeładowanie nazw, pakiety i operacje na plikach.

25.1. Rozwinięcie deklaracji ENTITY (generic, buffer, linkage)

Przykład:

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_unsigned.all;

entity Counter is
    generic (N:integer range 1 to 32 :=1;
            Delay: Time:=10ns);
    port (QQ: out std_logic_vector(N-1 downto 0);
          Reset,Clock,En : std_logic); -- domyślnie tryb IN
end entity Counter;

architecture RTL of Counter is
    signal int_QQ : std_logic_vector(N-1 downto 0);
begin
    QQ<=int_QQ;
    process(Reset,Clock)
    begin
        if Reset='1' then
            L1: for i in int_QQ'range loop
                int_QQ(i)<='0' after Delay;
            end loop L1;
            -- int_QQ<=(others =>'0');
        elsif Rising_Edge(Clock) then
            if En='1' then
                int_QQ<=int_QQ+'1' after Delay;
            end if;
        end if;
    end process;
end RTL;

library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_unsigned.all;

entity TB is
    generic (A: integer range 1 to 16 :=16); -- wartość domyślna (jeśli nie podana to
    -- pierwsza z deklarowanego typu)
end entity TB;

architecture TB of TB is
    -- deklaracja sygnałów wewnętrznych
    signal Reset,Clock,En : std_logic :='0';
    signal QQ:std_logic_vector(A-1 downto 0);

    -- deklaracja komponentów
    component Counter
        generic (N:integer range 1 to 32 :=1; -- 1 miejsce do zmiany wartości domyślnej GENERIC
                Delay: Time:=10ns);
        port (QQ: out std_logic_vector(N-1 downto 0);
              Reset,Clock,En : std_logic); -- domyślnie tryb IN
    end component ;

begin
    U1: Counter
        generic map(A)
        -- 2 miejsce do zmiany wartości domyślnej GENERIC
```

```

port map(QQ, Reset, Clk, En);

-- Stimuli
Reset<='1', '0' after 10ms;
En<='0', '1' after 500ms, '0' after 1000ms, '1' after 2sec;
Clk<= not Clk after 0.5ms; -- Clk=1kHz
end TB;

```

Deklaracja portów może zawierać tryby: IN, OUT, INOUT, BUFFER, LINKAGE.

Tryb IN – wartość obiektu aktualnego podstawionego w miejsce obiektu formalnego może być tylko odczytywana, nie może być zmieniana wewnątrz konstrukcji posiadającej dany parametr. Dla obiektów sygnałowych mogą być odczytywane atrybuty z wyjątkami: STABLE, QUIET, DELAYED, TRANSACTION.

Tryb OUT – wartość obiektu aktualnego jest wyliczana wewnątrz danej jednostki i stanowi wartość wyjściową z danej jednostki, nie można wewnątrz jednostki odczytywać jej wartości. Dla obiektów sygnałowych mogą być odczytywane atrybuty z wyjątkami: STABLE, QUIET, DELAYED, TRANSACTION, EVENT, ACTIVE, LAST_EVENT, LAST_ACTIVE, LAST_VALUE.

Tryb INOUT – oznacza, że wartość obiektu aktualnego może być odczytywana i uaktualniana wewnątrz danej konstrukcji języka. Dla obiektów sygnałowych mogą być odczytywane atrybuty z wyjątkami: STABLE, QUIET, DELAYED, TRANSACTION.

Tryb BUFFER – oznacza, że wartość obiektu aktualnego może być odczytywana i uaktualniana wewnątrz danej konstrukcji językowej. Można także odczytać wszystkie atrybuty obiektu. Rodzaj przesyłu sygnału ma podobne własności do rodzaju INOUT dla parametrów sygnałowych, różni się przede wszystkim tym, że port trybu INOUT może być uaktualniany przez więcej niż jedno źródło, tzn. sygnał reprezentujący port może być zmieniany w kilku procesach niezależnie, natomiast port trybu BUFFER może być uaktualniany tylko przez jedno źródło.

Tryb LINKAGE – jest przeznaczony do komunikacji z jednostkami projektowymi definiowanymi poza językiem VHDL. Wartość obiektu aktualnego może być odczytywana i uaktualniana, wtedy obiekt aktualny musi być również trybu LINKAGE.

GENERIC – jest to statyczna (niezmienna w czasie symulacji) wartość parametru przekazywana do wnętrza powoływanego bloku ENTITY. Wartość aktualna GENERIC jest ustalana na podstawie: wartości domyślnej, wartości domyślnej wstawianego komponentu, polecenia GENERIC MAP (od najsłabszego do największego priorytetu). Wartości GENERIC mogą być przekazywane do niższych poziomów hierarchii. Rozważmy sumator przedstawiony w poniższym przykładzie. Zdefiniowana jest tam wartość `gate_delay` dla wszystkich modułów niższej hierarchii, która przechodzi do bloków niższej hierarchii poprzez podstawienie wartości przy użyciu konstrukcji `generic map`. Chociaż wszystkie komponenty mają zdefiniowaną wartość domyślną parametru zostają one nadpisane nowymi wartościami uzyskanymi przez `generic map(gate_delay=>gate_delay)`.

```

library IEEE;
use IEEE.stdJogic.1164.all;
entity half_adder is

generic
    (gate_delay:Time:= 3 ns);
port
    (a, b : in std_logic;
    sum, carry: out std_logic);
end entity half_adder;

```

```

architecture generic_delay2 of half_adder is

  component xor2 is
    generic
      (gate_delay: Time);
    port
      (x,y : in std_logic;
      z: out stdJogic);
  end component xor2;

  component and2 is
    generic
      (gate_delay: Time);
    port
      (x, y: in stdJogic;
      z : out stdJogic);
  end component and2;

  begin
  EX1: xor2
  generic map
    (gate_delay => gate_delay)
  port map
    (x => x, y => y, z => sum);
  A1: and2
  generic map
    (gate_delay => gate_delay)
  port map
    (x=> x, y=> y, z=> carry);

end architecture generic_delay2;

```

GENERIC jest deklarowany obok PORTu, jednakże nie ma on fizycznej interpretacji. GENERIC jest dodatkowym środkiem do przekazywania informacji w głąb hierarchii projektu i przez to umożliwia projektowanie układów parametryzowanych. GENERIC są elementami o stałej wartości i dlatego nie mogą być zapisywane a tylko odczytywane. Wartość GENRIC musi być znana przed momentem rozpoczęcia symulacji (syntezy).

25.2. Pętle

Instrukcja pętli składa się z instrukcji sekwencyjnych, które są uruchamiane wielokrotnie. Jest kilka form składni pętli, pierwsza z nich przedstawia pętlą nieskończoną:

```

library IEEE;
use IEEE.std_logic_1164.all;

entity ppp is
end ppp;

architecture ppp of ppp is
  signal Clk:std_logic:='0';
begin
  process
  begin
    L1:loop
      Clk<= not Clk after 10ns;
      wait for 100ns;
    end loop L1;
  end process;
end ppp;
--generacja sygnału o okresie 200ns z 10ns opóźnieniem zboczy względem zera --czasowego

```

Pętla while:

```

library IEEE;
use IEEE.std_logic_1164.all;

entity ppp is
end ppp;

architecture ppp of ppp is
  signal Clk:std_logic:='0';
begin
  process
    variable n: integer range 0 to 101 :=1;
  begin
    n:=1;
    L2: while n<=100 loop
      Clk<= not Clk after 3ns;

```

```

                n:=n+1;
                wait for 10ns;
            end loop L2;
            wait for 1us;
        end process;
end ppp;
--generacja 50 impulsów o okresie 20ns z 3ns opóźnieniem zboczy względem
--zera następnie 1us przerwa, następnie cykl się powtarza.

```

W pętli L2 instrukcje są wykonywane dopóki warunek pętli $n \leq 100$ jest spełniony. Pętla samodzielnie nie zmienia zmiennej n ani jej nie modyfikuje.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity ppp is
end ppp;

architecture ppp of ppp is
    signal Clk:std_logic:='0';
begin
    process
    begin
        L3: for i in 1 to 100 loop
            Clk<= not Clk after 3ns;
            wait for i*10ns;
        end loop L3;
        wait for 1us;
    end process;
end ppp;
-- generacja 50 impulsów o czasach 1/0 rosnących od 20ns do 20*100 z 3ns
-- opóźnieniem zboczy względem zera czasowego,
-- następnie 1us przerwa, następnie cykl się powtarza.

```

W pętli L3 wykorzystany jest schemat FOR. W pętli niejawnie zdefiniowany jest identyfikator zakresu dyskretnego i a jego wartości są przyjmowane jako kolejne ze zadeklarowanego zakresu w czasie kolejnych przejść pętli, począwszy od wartości lewej do prawej. Parametr i może być wykorzystywany jedynie w instrukcjach sekwencyjnych występujących wewnątrz pętli w której został zadeklarowany. Nie można zmieniać jego wartości a jedynie ją odczytywać. Parametr i przykrywa ewentualnie zadeklarowane zmienne o tej samej nazwie w zakresie instrukcji pętli.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity ppp is
end ppp;

architecture ppp of ppp is
    signal Clk:std_logic:='0';
begin
    process
    begin
        L4: for i in 1 to 100 loop
            Clk<= not Clk after 3ns;
            wait for i*10ns;
            exit L4 when i=50;
        end loop L4;
        wait for 1us;
    end process;
end ppp;
-- zmniejszenie z 50 do 25 liczby generowanych impulsów (w stosunku do
-- poprzedniego przykładu) o czasach 1/0 rosnących od 20ns do 20*50 z 3ns
-- opóźnieniem zboczy względem zera czasowego,
-- następnie 1us przerwa, następnie cykl się powtarza.

```

Instrukcja `exit` umożliwia wcześniejsze zakończenie pętli niż to wynika z warunków `for`, `while` lub zakończenie pętli nieskończonej. Występuje w 2 formach tj. z warunkiem i bezwarunkowo. W przypadku formy z warunkiem wyjście następuje jeśli warunek jest spełniony. Etykieta (jak w przykładzie L4) instrukcji wyjścia jest opcjonalna.

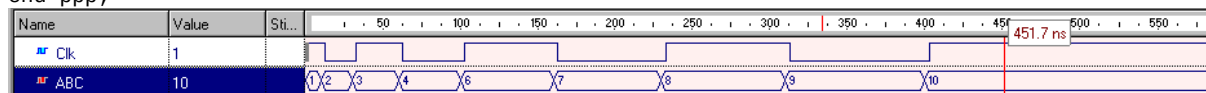
```

library IEEE;
    use IEEE.std_logic_1164.all;

entity ppp is
end ppp;

architecture ppp of ppp is
    signal Clk:std_logic:='0';
    signal ABC: integer :=1;
begin
    process
    begin
        L4: for i in 1 to 10 loop
            ABC<=i;
            Clk<= not Clk after 3ns;
            next L4 when i=5;
            wait for i*10ns;
        end loop L4;
        wait for 1us;
    end process;
end ppp;

```



Specyfikacja konfiguracji jest używana do identyfikacji związków pomiędzy komponentami a ich parami bloków: ENTITY-ARCHITECTURE. Dokładna lokalizacja pary: ENTITY-ARCHITECTURE może odbyć się poprzez podanie biblioteki, nazwy bloku ENTITY oraz aktywnej architektury. Rozważmy poniższy przykład. Blok ENTITY Counter ma dwie architektury nazwane tutaj `Active_1` oraz `Active_0`. W przypadku gdy nie jest użyta żadna dodatkowa składnia VHDL wyróżniająca aktywną architekturę, domyślnie uaktywniana jest architektura ostatnio kompilowana (w naszym przypadku `Active_0`). W bloku TB powołany jest komponent o nazwie Licznik. W przypadku gdy nie jest użyta składnia CONFIGURATION, kompilator poszukuje bloku ENTITY o nazwie zgodnej z podaną w bloku COMPONENT a jeśli jej nie znajdzie w żadnym z dołączonych plików i bibliotek, wskaże na błąd kompilacji. W przypadku gdy użyta jest składnia CONFIGURATION można zmienić nazwę bloku ENTITY przypisanego do deklarowanego komponentu oraz podać która konkretnie architektura zostanie uaktywniona (nazwa architektury podana jest w nawiasie). W przypadku braku specyfikacji architektury aktywna jest architektura ostatnio kompilowana. Gdy chcemy dla wszystkich komponentów o tej samej nazwie użyć tego samego bloku ENTITY i tej samej architektury można użyć składni:

W powyższej linii komponentowi o nazwie Licznik przypisany jest blok ENTITY o nazwie Counter pobrany z biblioteki WORK (czyli z bieżącego projektu) a aktywna architektura ma nazwę Active 1.

```
for L2: Licznik use entity WORK.Counter(Active_1);
```

```
for all: Licznik use entity WORK.Counter(Active_1);
```

Nie każdy syntezer umożliwia podawanie konfiguracji, wówczas aktywne jest domyślne przypisywanie architektur i komponentów.

Przykład licznika o 2 architekturach, z wejściami aktywnymi stanem niskim (Active_0) i wysokim (Active_1).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity Counter is
    generic (N: integer range 1 to 32 :=32);
    port ( Q: out std_logic_vector(N-1 downto 0);
          Reset,Clk,En: std_logic
        );
end entity Counter;

architecture Active_1 of Counter is -- input and output signals are '1' - active
    signal int_Q: std_logic_vector(N-1 downto 0);
begin
    Count:
    process(Clk,Reset)
    begin
        if Reset='1' then
            int_Q<=(others=>'0');
        elsif Rising_Edge(Clk) then
            if En='1' then
                int_Q<=int_Q+'1';
            end if;
        end if;
    end process    Count;
    Q<=int_Q;
end Active_1;

architecture Active_0 of Counter is -- input and output signals are '0' - active
    signal int_Q: std_logic_vector(N-1 downto 0);
begin
    Count:
    process(Clk,Reset)
    begin
        if Reset='0' then
            int_Q<=(others=>'1');
        elsif Rising_Edge(Clk) then
            if En='0' then
                int_Q<=int_Q-'1';
            end if;
        end if;
    end process    Count;
    Q<=int_Q;
end Active_0;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity TB is
    generic (N: integer range 1 to 32 :=32);
end entity TB;

architecture TB of TB is
    signal Q1,Q2 : std_logic_vector(N-1 downto 0);
    signal Clk,Reset,En: std_logic :='0';
    component Licznik
        generic (N: integer range 1 to 32);
        port ( Q: out std_logic_vector(N-1 downto 0);
              Reset,Clk,En: std_logic);
    end component Licznik;
    -- Configuration statement
    for L1: Licznik use entity WORK.Counter; -- default binding is Active_0
    for L2: Licznik use entity WORK.Counter(Active_1);
begin
    -- Signal generation
    Clk<= not Clk after 50ns;
    Reset<='1', '0' after 200ns, '1' after 2us;
    En<='0', '1' after 500ns, '0' after 2.1us;
    -- Component instantiation
    L1: Licznik
        generic map(N)
```

```

        port map(Q1,Reset,Clk,En);
        L2: Licznik
        generic map(N)
        port map(Q2,Reset,Clk,En);
end TB;

```

Deklaracja konfiguracji może być także wykonana poza architekturą w oddzielnym bloku i jeśli trzeba w oddzielnym pliku. Poniżej przedstawiony jest przykład deklaracji configuration. Podana jest nazwa konfiguracji oraz nazwa bloku ENTITY którego dotyczy. Druga linia identyfikuje konkretną nazwę architektury której dotyczy. Następnie wstawione są bloki for – end for deklarujące powiązania pomiędzy wstawianymi komponentami a konkretnymi ich architekturami. Jeśli użyty jest oddzielny blok configuration wówczas nie można używać: ani wyrażenia for all: licznik use entity counter ..., ani konfiguracji tych samych komponentów równocześnie bezpośrednio w konfigurowanej architekturze jak i bloku configuration.

```

configuration Config_1 of tb is -- name of configuration and associated ENTITY
    for TB
        for L1: Licznik use entity WORK.Counter; -- default binding is Active_0
        end for;
        for L2: Licznik use entity WORK.Counter(Active_1);
        end for;
        -- for all: Licznik use entity WORK.Counter(Active_1);
    -- using of statement FOR ALL is not possible in separate CONFIGURATION BLOCK
    end for;
end configuration Config_1;

```

Z blokiem configuration powiązane są zaawansowane funkcje języka takie jak: połączenie komponentów i architektur, bezpośrednie wstawienie, połączenie inkrementalne, łączenie do bloków konfiguracji (zamiast do architektur), nie będą one tu omówione.

25.4. Przeładowanie podprogramów i operatorów (over loading)

Bardzo użyteczną funkcją języka VHDL jest możliwość „przeładowania nazw” podprogramów oraz operatorów. Jest wiele modeli opisujących przerzutnik D. Wyobraźmy, że mamy do napisania model systemu zawierającego przerzutniki D. Możemy w tym celu użyć procedur jak w poniższym przykładzie

```

procedure dff(signal d,clk,reset,in std_logic; signal q,qbar: out std_logic) is
begin
    if reset='1' then
        q<='0';
        qbar<='1';
    elsif Rising_Edge(clk) then
        q<=d;
        qbar<= not d;
    end if;
end procedure dff;

```

Jeśli chcemy użyć przerzutnika zarówno z asynchronicznym zerowaniem jak i ustawianiem musimy użyć innej procedury (jak w przykładzie poniżej), której nazwa może być taka sama jak procedury powyższej a wybór aktywnej procedury jest wykonywany na podstawie aktualnych parametrów ją wywołujących (tj. liczby parametrów oraz ich typów logicznych).

```

procedure dff(signal d,clk,reset,set,in std_logic; signal q,qbar: out std_logic) is
begin
    if reset='1' then
        q<='0';
        qbar<='1';
    else
        if set='1' then
            q<='1';

```

```

        qbar<='0';
    elsif Rising_Edge(clk) then
        q<=d;
        qbar<= not d;
    end if;
end if;
end procedure dff;

dff(d,clk,reset,q,qbar);          -- przerzutnik bez asynchronicznego ustawiania
dff(d,clk,reset,set,q,qbar);      -- przerzutnik z asynchronicznym ustawianiem

```

Podobnie jak na podprogramach „przeładowanie” może być wykonane na operatorach np. „*”, „+”. W tym przypadku wykorzystuje się możliwość identycznego nazwania operatorów dla różnych typów logicznych operandów, co jest bardzo często wykorzystywane w pakiecie STD_LOGIC_1164.

Funkcje i procedury są elementami koniecznymi dla zapewnienia czytelności kodu VHDL, szczególnie dla większych projektów. Jak powyżej przedstawiono, można je także używać do zwiększenia bogactwa językowego poprzez zawieranie operacji na różnych typach operandów pod tą samą nazwą funkcji/procedury/operatora.

25.5. Pakiety (packages)

Większe grupy funkcji i procedur używanych do modelowania VHDL mogą zostać zgrupowane w pakietach. Pakiety umożliwiają grupowanie deklaracji typów, funkcji i procedur. W ten sposób mogą być one używane wielokrotnie w różnych projektach VHDL. Pakiety składają się z 2 części: deklaracyjnej oraz opisowej. W części deklaracyjnej wyszczególnione są deklaracje typów, funkcji, procedur, funkcji – co łącznie tworzy interfejs usług oferowanych przez pakiet dostępnych dla jego otoczenia. Poniżej przedstawiony jest przykład wykorzystania własnego pakietu o nazwie my_pkg z własnej biblioteki my_lib.

```

-- ver 1.0 3/1/2001
-- function table
--
-- |Control| QQ | Procedure name |
-- |-----|---|-----|
-- | 000 | QQ | none |
-- | 001 | QQ+1 | add1 |
-- | 010 | QQ-1 | sub1 |
-- | 011 | Data | none |
-- | 100 | QQ+Data | dinp |
-- | 101 | QQ-Data | dinm |
-- | 110 | QQ<<1 | shl1 |
-- | 111 | QQ>>1 | shr1 |
-- |-----|---|-----|

```

```

library IEEE;
use IEEE.std_logic_1164.all;
library MY_LIB;
use MY_LIB.my_pkg.all;

entity counter is -- counts up, down or make other functions depending on signal Control state (see
    -- above table)
    generic (N:positive range 1 to 32 :=8);
    port(Clk,Reset: in std_logic;
        Control: in std_logic_vector(2 downto 0);
        Data: in std_logic_vector(N-1 downto 0);
        QQ: out std_logic_vector(N-1 downto 0)
    );
end entity counter;
architecture RTL of counter is
    signal tmp_QQ:std_logic_vector(N-1 downto 0);
begin

```

```

QQ<=tmp_QQ;

process (Clk, Reset)
begin
    if Reset='1' then
        tmp_QQ<=(QQ(N-1 downto N-2)<="11" ,others '0');
    elsif Rising_Edge(Clk) then
        case Control is
            when "001"    => add1(tmp_QQ);
            when "010"    => sub1(tmp_QQ);
            when "011"    => tmp_QQ<=Data;
            when "100"    => dinp(tmp_QQ, Data);
            when "101"    => dinm(tmp_QQ, Data);
            when "110"    => shl1(tmp_QQ);
            when "111"    => shr1(tmp_QQ);
            when others => null;
        end case;
    end if;
end process;
end RTL;

-- file with individually declared package
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package my_pkg is
    procedure add1(signal Qin: inout std_logic_vector);
    procedure sub1(signal Qin: inout std_logic_vector);
    procedure dinp(signal Qin: inout std_logic_vector; signal Data : in std_logic_vector);
    procedure dinm(signal Qin: inout std_logic_vector; signal Data : in std_logic_vector);
    procedure shl1(signal Qin: inout std_logic_vector);
    procedure shr1(signal Qin: inout std_logic_vector);
end package my_pkg;

package body my_pkg is

    procedure add1(signal Qin: inout std_logic_vector) is
    begin
        Qin<=Qin+'1';
    end procedure add1;

    procedure sub1(signal Qin: inout std_logic_vector) is
    begin
        Qin<=Qin-'1';
    end procedure sub1;

    procedure dinp(signal Qin: inout std_logic_vector; signal Data : in std_logic_vector) is
    begin
        Qin<=Qin+Data;
    end procedure dinp;

    procedure dinm(signal Qin: inout std_logic_vector; signal Data : in std_logic_vector) is
    begin
        Qin<=Qin-Data;
    end procedure dinm;

    procedure shl1(signal Qin: inout std_logic_vector) is
    begin
        Qin<=Qin(Qin'left-1 downto 0)&Qin(Qin'left);
    end procedure shl1;

    procedure shr1(signal Qin: inout std_logic_vector) is
    begin
        Qin<=Qin(0) & Qin(Qin'left downto 1);
    end procedure shr1;
end package body my_pkg;

```

25.6. Podstawowe operacje wej/wyj – operacje na plikach (I/O)

Do tej pory używaliśmy języka VHDL do opisu 3 typów obiektów: sygnałów, zmiennych i stałych. Zasady ich używania są one do siebie podobne. Pliki są odmiennym typem obiektu i nie mogą służyć jako wyprowadzenia I/O bloku ENTITY. Są one przeznaczone do komunikacji pomiędzy językiem VHDL a środowiskiem w którym jest on zaimplementowany. Używanie plików jest całkowicie inne niż pozostałych 3 rodzajów obiektów. Dodatkowo są wykorzystywane specjalne operacje do zapisu, odczytu i modyfikacji zawartości plików. Informacje przesyłane przez pliki nie mogą być syntezywane. Jako jedno z zastosowań pliki są często wykorzystywane do konstrukcji wzorów testowych (testbench) – programów VHDL służących do symulacji modeli VHDL.

Podstawowe operacje na plikach obejmują: deklarację pliku i jego typu, operację otwierania i zamykania plików określonego typu, odczyt i zapis z/do pliku.

Deklaracja pliku składa się z deklaracji zawartości pliku oraz deklaracji obiektu plikowego tak jak to przedstawiono w poniższym przykładzie:

```
type TEXT is file of string;
type INTF is file of integer;
file Integer_file :INTF;
file Input_file   :TEXT;
```

Pierwsza linia deklaruje typ plikowy który składa się ze znaków ASCII i jest plikiem czytelnym dla człowieka. W drugiej linii deklarowany jest plik zawierający elementy składowe typu INTEGER. Pliki takiego typu są przechowywane jako ciągi binarne i nie są czytelne dla człowieka. Oba typy są zdefiniowane w pakiecie STANDARD. W ogólności pliki mogą przechowywać ciągi danych o bardziej złożonej strukturze, jednakże tutaj zostaną przedstawione prostsze przypadki dla predefiniowanych typów danych. Linie 3 i 4 powyższego przykładu zawierają deklaracje obiektów plików poszczególnych typów. Należy dodać, że typ plikowy TEXT jest predefiniowany w pakiecie TEXTIO biblioteki STD. Obiekty Integer_file oraz Input_file możemy traktować jako wskaźniki do plików zawierających ciągi wartości typu INTEGER oraz znaków ASCII. Kiedy procedury VHDL są użyte do odczytu lub zapisu z tych plików, typy plików umożliwiają prawidłową interpretację przechowywanych wartości.

Otwieranie i zamykanie plików poprzedza (kończy) proces zapisu i odczytu z plików. W VHDL'93 następujące procedury służą do otwierania i zamykania plików

```
procedure FILE_OPEN (file file_handle: FILE_TYPE;
                     File_Name: in STRING;
                     Open_Kind: in File_Open_Kind:=READ_MODE);
procedure FILE_OPEN (File_Status: out FILE_OPEN_STATUS;
                     file file_handle: FILE_TYPE;
                     File_Name: in STRING;
                     Open_Kind: in File_Open_Kind:=READ_MODE);
procedure FILE_CLOSE (file f: FILE_TYPE);
```

Rozważmy pierwszą procedurę. Pierwszym argumentem jest wskaźnik do pliku. Drugi argument jest nazwą fizyczną pliku skojarzoną ze wskaźnikiem do pliku np. *input_file.txt*. Trzeci argument opisuje tryb otwarcia pliku i mogą to być: READ_MODE, WRITE_MODE, APPEND_MODE. Druga procedura jest identyczna jak pierwsza z dodaniem parametru wyjściowego FILE_STATUS, którego wartości oznaczają:

OPEN_OK	- operacja otwarcia pliku powiodła się;
STATUS_ERROR	- próba otwarcia pliku już otwartego;
NAME_ERROR	- plik nie znaleziony;
MODE_ERROR	- plik nie może być otworzony w tym trybie.

Procedura FILE_CLOSE zamyka plik (zapisuje zawartość/zmiany pliku na dysku). Chociaż istnieje jawna procedura FILE_CLOSE jednakże nie zawsze konieczne jest jej uruchamianie (gdyż zawsze jest ona wywoływana niejawnie na zakończenie pracy środowiska VHDL). W rzeczywistości również nie zawsze konieczne jest jawne otwieranie plików – można to wykonać w czasie deklaracji pliku jak to pokazano poniżej.

```
-- jawne otwieranie pliku niejawnie zamykanie pliku
type IntegerFileType is file of integer;
--
--
process is
file datain: IntegerFileType;      -- typ danych przechowywanych w pliku
variable: fstatus:File_Open_Status; -- status otwarcia pliku
--
begin
    file_open (fstatus,datain,"myfile.txt",read_mode); -- jawne otwarcie pliku
--
```

```

--
end process; -- niejawne wywołanie FILE_CLOSE

-- niejawne otwieranie i zamykanie pliku
type IntegerFileType is file of integer;
--
--
process is
file datain: IntegerFileType open read_mode is "myfile.txt"; -- niejawne
-- otwarcie pliku
begin
--
--
end process; -- niejawne wywołanie FILE_CLOSE

```

Czytanie i zapis do pliku odbywa się po wstępnym otwarciu pliku. Standard VHDL zawiera poniższe procedury:

```

procedure READ( file file_handle: FILE_TYPE; value: out type);
procedure WRITE( file file_handle: FILE_TYPE; value: in type);
function ENDFILE( file file_handle: FILE_TYPE) return boolean;

```

Procedury WRITE oraz READ służą odpowiednio do zapisu lub odczytu plików podczas gdy funkcja ENDFILE testuje koniec pliku w czasie jego odczytu. Większość symulatorów VHDL umożliwia zapis i odczyt plików zawierających wartości: CHARACTER, INTEGER, REAL i inne. Pakiet standardowy TEXTIO dostarcza dodatkowe procedury odczytu i zapisu plików tekstowych.

Przykład: Plik binarny. Utworzono plik "file2.int" i wpisano do niego 8 wartości typu INTEGER

```

entity f_w_1 is
end entity f_w_1;
architecture fff of f_w_1 is
begin
    process is
    type IntegerFileType is file of integer;
    file dataout: IntegerFileType;
    variable count: integer:=0;
    variable fstatus:FILE_OPEN_STATUS;
    begin
        file_open(fstatus,dataout,"file2.int",write_mode);
        for j in 1 to 8 loop
            write(dataout,count);
            count:=count+12;
        end loop;
        file_close(dataout);
        wait;
    end process;
end fff;

```

Pakiet TEXTIO jest pakietem zawartym w niemal każdym symulatorze VHDL. Wchodzi on w skład biblioteki STD i zawiera zestaw typów plikowych oraz procedur je obsługujących. Przykładowo pakiet ten zawiera typ plikowy TEXT oraz procedury READ,WRITE dla typów logicznych bit, bit_vector, character oraz string. W przypadku użycia pakietu TEXTIO operacje na plikach przeprowadzane są pośrednio przez bufor (nazwany tutaj BUF). BUF można przyrównać do pomostu łączącego plik z językiem VHDL. Procedury read oraz write umożliwiają dostęp do BUF poprzez jego odczyt/zapis. Procedury readline oraz writeline przenoszą zawartość BUF z lub do pliku. Są 2 specjalne rodzaje bufora BUF nazwane INPUT oraz OUTPUT, predefiniowane w pakiecie TEXTIO. Bufory te są zazwyczaj skojarzone ze STD_INPUT oraz STD_OUTPUT środowiska projektowego i często jest to konsola symulatora.

Rozważmy poniższy przykład. Plik jest zadeklarowany jako TEXT.

```

use STD.textio.all;
entity f_w_2 is

```

```

end entity f_w_2;
architecture fff of f_w_2 is
begin
    process is
        file outfile: Text;
        variable count: integer:=5;
        variable fstatus:FILE_OPEN_STATUS;
        variable value: bit_vector( 3 downto 0) :=x"6";
        variable buf:line;
        begin
            file_open(fstatus,outfile,"file_3.txt",write_mode);
            write(buf, "This is an example of text line");
            writeline(outfile,buf);
            write(buf, ' ');
            write(buf,value);
            writeline(outfile,buf);
            write(buf, "... and so on ...");
            writeline(outfile,buf);
            writeline(outfile,buf);
            -- previous call of procedure "writeline" cause
            -- that "buf" is empty
            -- so this line only introduces new empty line
            -- into the output file
            write(buf,"An example text line No 2");
            writeline(outfile,buf);
            file_close(outfile);
            wait; -- suspend of process for infinity
        end process;
    end fff;

```

Wynikiem wykonania powyższego procesu jest plik tekstowy o nazwie *file_3.txt* i zawartości przedstawionej poniżej.

```

This is an example of text line
0110
... and so on ...

```

An example text line No 2

Zmienna *buf* jest typu *line* (specjalny logiczny typ wskaźnikowy) i tylko taki typ logiczny jest dozwolony jako drugi argument procedur *writeline* oraz *readline*. Procedury *write* oraz *read* są przeładowane i jako drugi argument umożliwiają kilka typów logicznych np. *bit_vector* oraz *integer*.

Następny przykład przedstawia wykorzystanie procedur *read* oraz *readline*. W jaki jednak sposób można obsłużyć typ logiczny *std_logic_vector* nie zawarty w pakiecie STD? Aby to wykonać należy wykorzystać funkcję konwersji typów logicznych. Poniżej przedstawiony jest przykład pakietu zawierającego taką funkcję, a następnie wykorzystania go do odczytu i zapisu danych logicznych typu *std_logic_vector*.

```

--
library IEEE;
    use IEEE.std_logic_1164.all;
    use STD.textio.all;
package classio is
    procedure read_v1d(variable f: in text; v: out std_logic_vector);
-- v is variable by default
    procedure write_v1d(variable f:out text; v: in std_logic_vector);
end package classio;
package body classio is
    procedure read_v1d(variable f: in text; v: out std_logic_vector) is
        variable buf:line;
        variable c:character;
    begin
        readline(f,buf);
        for i in v'range loop
            read(buf,c);
            case c is
                when 'X' => v(i):='X';
                when 'U' => v(i):='U';
                when 'Z' => v(i):='Z';
                when '0' => v(i):='0';
                when '1' => v(i):='1';
            end case;
        end loop;
    end read_v1d;
    procedure write_v1d(variable f:out text; v: in std_logic_vector) is
        variable buf:line;
        variable c:character;
    begin
        for i in v'range loop
            c:=v(i);
            write(buf,c);
            if i=v'last then
                writeline(f,buf);
            else
                write(buf," ");
            end if;
        end loop;
        file_close(f);
    end write_v1d;
end package body classio;

```

```

        when '-' => v(i):='-';
        when 'W' => v(i):='W';
        when 'L' => v(i):='L';
        when 'H' => v(i):='H';
        when others => v(i):='0';
    end case;
end loop;
end procedure read_v1d;

procedure write_v1d(variable f:out text; v: in std_logic_vector) is
    variable buf:line;
    variable c:character;
begin
    for i in v'range loop
        case v(i) is
            when 'X' => write(buf,'X');
            when 'U' => write(buf,'U');
            when 'Z' => write(buf,'Z');
            when '0' => write(buf,'0');
            when '1' => write(buf,'1');
            when '-' => write(buf,'-');
            when 'W' => write(buf,'W');
            when 'H' => write(buf,'H');
            when 'L' => write(buf,'L');
            when OTHERS => write(buf,'^');
        end case;
    end loop;
    writeline(f,buf);
end procedure write_v1d;
end package body classio;

-- An example of use of above declared procedures
library IEEE;
    use IEEE.std_logic_1164.all;
    use STD.textio.all;
    use WORK.classio.all;
entity check is
end entity check;

architecture beh of check is
begin
    IOoper:
    process is
        file infile: text open read_mode is "infile.txt"; -- implicit file open
        file outfile: text open write_mode is "outfile.txt"; -- implicit file open
        variable ck : std_logic_vector(15 downto 0):=x"0008";
    begin
        -- copy the input file to the output file (but only the first 16 characters)
        while not (endfile(infile)) loop
            read_v1d(infile,ck);
            write_v1d(outfile,ck);
        end loop;
        file_close(outfile); -- close file and update on the disc
        file_close(infile); -- relase file for other applications
        wait; -- artificial end of process
    end process;
end architecture beh;

```

Jeśli plik *infile.txt* zawiera:

```

01010101010101010101010101
121342
001100110011001100110011
asqq23441327946764587364785c47
xchi1248732klewnfd
cas1
0000111100001111 - tttt

```

wówczas po wykonaniu procesu IOoper : zawartość pliku *outfile.txt* będzie następująca:

```

0101010101010101
1010000000000000
0011001100110011
0000000010000000
0000100000000000
0001111111111111
0000111100001111

```

26. Literatura

- [1] „VHDL język opisu i projektowania układów cyfrowych” W. Wrona, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, 1998.
- [2] „A designer guide to VHDL Synthesis” autorstwa D.E. Ott, T.J. Wilderotter,
- [3] „VHDL made easy” D. Pellerin, D. Taylor, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

CZĘŚĆ 2. VERILOG²

1. WSTĘP

1.1. Ewolucja systemów CAD

Cyfrowe układy przeżywają gwałtowny rozwój od prawie 30 lat. Pierwsze cyfrowe układy scalone zawierały niewielką liczbę tranzystorów, stąd ich nazwa SSI (Small Scale Integration). Wraz z rozwojem technologii, zwiększała się ilość tranzystorów zwarta w jednym chipie. Powstały układy średniej skali integracji (MSI - Medium Scale Integration). Powstanie układów LSI oznaczało możliwość umieszczenia w jednym układzie tysięcy tranzystorów i od tego momentu proces projektowania zaczynał być bardzo skomplikowany. Potrzebne było zautomatyzowanie tego procesu - zaczęły rozwijać się systemy CAD. Projektanci zaczęli używać symulatorów mogących symulować pojedyncze bloki układowe składające się z około 100 tranzystorów, całe układy były jednak testowane "na żywo", a layout robiony był na papierze lub ręcznie na terminalu graficznym. Wraz z początkiem rozwoju układów VLSI, projektanci mogli umieścić ponad 100,000 tranzystorów w jednym układzie scalonym. Z powodu takiej złożoności, nie było już możliwe przetestowanie takiego układu w sposób tradycyjny. Systemy CAD stały się niezbędne do projektowania. Projektanci składali cały układ z pojedynczych bramek, tworzyli większe bloki, aż do zbudowania całego układu. Cały układ był testowany w symulatorze.

1.2. Rozwój języków HDL

Do opisu programów komputerowych używa się języków, takich jak FORTRAN, Pascal czy C. Języki te opisują działanie programów komputerowych w naturalny dla nich, sekwencyjny, sposób. Projektanci układów scalonych potrzebowali też języka opisującego działanie sprzętu w sposób równoległy, tak, jak to ma miejsce w rzeczywistych układach. W ten sposób powstały języki Verilog i VHDL. Verilog powstał w 1983 roku w firmie Gateway Design Automation, trochę później powstał język VHDL. Na początku, pomimo opisu w języku HDL, projektanci musieli ręcznie tłumaczyć układ do postaci schematu połączeń pomiędzy bramkami. Dopiero w późnych latach 80-tych powstały pierwsze systemy automatycznej syntezy układów cyfrowych. Zaczęto wtedy operować pojęciami na poziomie przesłań pomiędzy rejestrami (Register Transfer Level - tzw. RTL), czyli na takim poziomie abstrakcji, na którym interesuje nas, jak przepływają dane. Szczegóły budowy poszczególnych bramek oraz połączenia między nimi były rozważane w sposób automatyczny.

W ten sposób języki HDL stały się pierwszym ogniwem w syntezie układów - od opisu w języku HDL zaczynało się projektowanie układu. Języki HDL używane są również do projektów i symulacji układów FPGA, PAL, magistral połączeniowych itp.

1.3. Typowy przebieg projektowania

Każdy proces projektowania jest poprzedzony sporządzeniem wymagań. Wymagania precyzują w sposób abstrakcyjny funkcjonowanie układu, jego interfejs i ogólną architekturę. W tym momencie projektant nie musi myśleć o tym, jak zrealizuje

² Opracowano na podstawie: S.Palnitkar, "Verilog[®] HDL, A Guide to Digital Design and Synthesis", SunSoft Press, 1996

poszczególne części układu. Następnie wykonuje się opis behawioralny, aby dokonać analizy układu pod względem funkcjonalnym, sprawdzić jego działanie, zgodność ze standardami oraz innymi względami na wysokim poziomie abstrakcji. Opis behawioralny może już być wykonany przy użyciu języków HDL.

Opis behawioralny jest następnie ręcznie przekształcany do opisu na poziomie RTL. Projektant musi opisać przepływ poszczególnych sygnałów w układzie. Od tego momentu projektowanie może odbywać się przy użyciu systemów CAD.

Synteza logiczna dokonuje zamiany opisu RTL na listę połączeń poszczególnych bramek. Lista połączeń wprowadzana jest do układu dokonującego rozmieszczenia i trasowania połączeń, czyli na jej podstawie otrzymywany jest *layout* układu scalonego, który to następnie jest sprawdzany i wysyłany do produkcji.

Jak widać z powyższego opisu, najwięcej wysiłku projektant przeznacza na ręczną optymalizację opisu na poziomie RTL.

1.4. Znaczenie języków HDL

W porównaniu do tradycyjnych metod projektowania opartych o schemat połączeń, języki HDL mają wiele zalet:

- Projekty mogą być opisane na bardzo wysokim poziomie abstrakcji. Projektant może zapisać projekt na poziomie RTL bez decydowania o technologii wykonania układu. Narzędzia syntezy logicznej automatycznie dostosują projekt do danej technologii. W przypadku pojawienia się nowej technologii, projektanci nie muszą przeprojektowywać układu - po prostu system automatycznej syntezy tworzy nową listę połączeń korzystając z bramek wykonanych w nowej technologii.
- Dzięki opisowi w języku HDL, sprawdzenie działania układu może nastąpić na bardzo wczesnym etapie projektowania. Dużo łatwiej jest znaleźć błąd w opisie na poziomie RTL niż w połączeniach bramek.
- Projektowanie w języku HDL przypomina programowanie w języku komputerowym. Opis tekstowy wraz z komentarzem dużo łatwiej zrozumieć i śledzić. Opis na poziomie połączeń pomiędzy bramkami dla dużych układów jest zupełnie nieczytelny.

1.5. Popularność Verilog HDL

Język Verilog HDL, w skrócie Verilog, posiada następujące cechy:

- Jest językiem ogólnego stosowania, łatwym do nauczenia i używania. W składni podobny jest do języka C.
- Umożliwia opis układu na różnych poziomach abstrakcji jednocześnie, projektant może opisywać sprzęt posługując się opisami na poziomie poszczególnych kluczy, bramek, RTL lub behawioralnym. Wystarcza też do opisu pobudzenia oraz projektowania w sposób hierarchiczny.
- Większość popularnych systemów syntezy logicznej posiada implementację Verilogu.
- Wszyscy producenci układów scalonych dostarczają biblioteki do symulacji po syntezy logicznej.
- Programming Language Interface (PLI) umożliwia dopisywanie w języku C procedur współdziałających z wewnętrznymi strukturami Verilog, umożliwiając dopasowanie symulatora Verilog HDL do indywidualnych potrzeb.

1.6. Kierunki rozwoju języków HDL

- Najbardziej popularne poziomy abstrakcji to poziom RTL, gdyż systemy syntezy logicznej umożliwiają przejście z poziomu RTL do listy połączeń.

Rozwijają się również techniki formalnej weryfikacji służące do kontroli opisu Verilog oraz

jego zgodności opisu na poziomie RTL z listą połączeń bramek.

Dla bardzo skomplikowanych układów, takich jak mikroprocesory, lista połączeń bramek nie jest optymalna - w takim wypadku stosuje się projektowanie części układu na poziomie RTL, aby osiągnąć lepszy rezultat.

Często też stosuje się metodę projektowania tzw. mieszane "*bottom-up*"- projektant korzysta z bloków zaprojektowanych wcześniej. Pozwala to na szybkie złożenie z takich "klocków" gotowego układu. Np. podczas projektowania nowej generacji mikroprocesora, projektanci mikroprocesorów budują go korzystając z poziomu RTL, ale do opisu innych układów, takich jak układ grafiki, czy magistrala systemowa, wykorzystują modele behawioralne. Dzięki temu sam procesor może powstać bardzo szybko, nawet wcześniej, niż opis RTL układów dodatkowych.

2. KONCEPCJA MODELOWANIA HIERARCHICZNEGO

2.1. Metodologia projektowania

Zasadniczo wyróżnia się dwa podejścia do programowania:

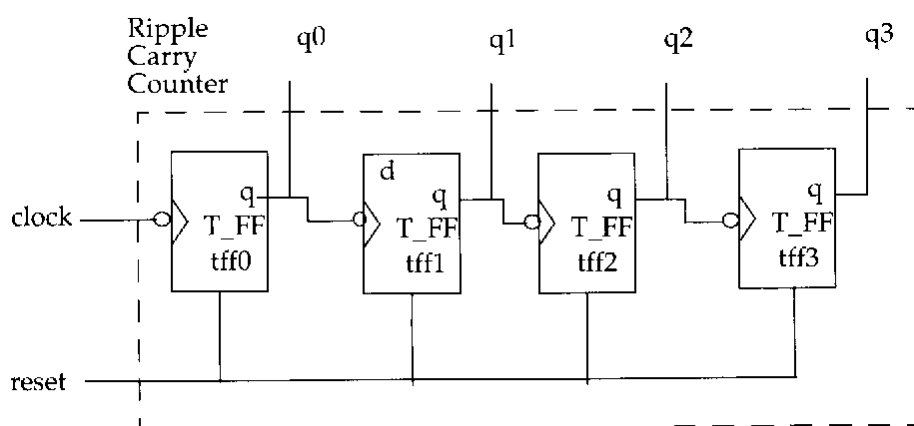
- *top-down*
- *bottom-up*

W metodologii *top-down* najpierw definiuje się blok główny a następnie opracowuje pod-bloki, aż do zejścia do poziomu bloków podstawowych.

W metodologii *bottom-up* najpierw rozpoznaje się podstawowe bloki, jakie ma się do dyspozycji, z nich składa się trochę większe bloki, aż do uzyskania całego układu.

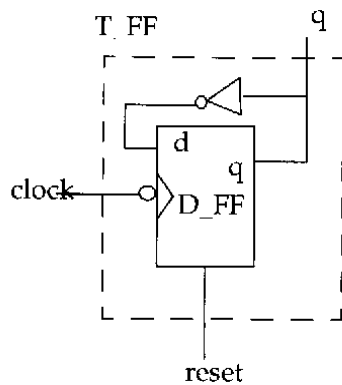
W praktyce stosuje się kombinację tych metodologii.

2.2. Licznik 4-bitowy z przeniesieniem

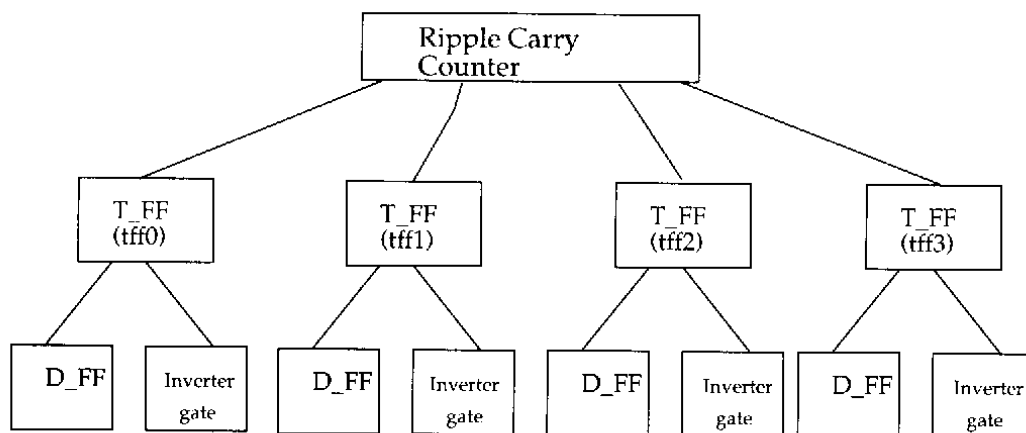


Powyższy licznik [1] składa się z przerzutników T_FF. Każdy licznik składa się z przerzutników D wyzwalanych ujemnym zboczem oraz inwerterów (zakładając, że wyjście q_{bar} nie jest osiągalne w przerzutnikach D).

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



W ten sposób projekt przybrał postać hierarchiczną [1]:



2.3. Moduły

Moduł to podstawowy blok w Verilog'u.

Moduł posiada interfejs wej./wyj. Pozwala to na zmiany we wnętrzu układu, bez zmiany w pozostałych częściach systemu.

W przykładzie z licznikiem, modułami były bloki T_FF i D_FF.

```
module <nazwa_modułu> (<lista_końcówek_modułu>);
...
<wnętrze modułu>
...
endmodule
```

Verilog jest językiem zarówno strukturalnym jak i behawioralnym. Wnętrze każdego modułu może być zdefiniowane przy użyciu 4 poziomów abstrakcji:

- poziom behawioralny (algorytmiczny) - najwyższy poziom abstrakcji w Verilog'u. Moduł projektuje się na podstawie algorytmu działania, nie precyzując szczegółów implementacji.
- poziom *dataflow* - na tym poziomie moduły definiuje się poprzez opis przepływu danych. Projektant musi wiedzieć jak dane są przetwarzane oraz przepisywane pomiędzy rejestrami.
- poziom bramek - moduł projektuje się definiując bramki logiczne oraz ich połączenia

(podobnie jak na schemacie).

- poziom kluczy - jest to najniższy poziom oferowany w Verilog'u.

Verilog umożliwia mieszanie wszystkich czterech poziomów abstrakcji w jednym projekcie. Popularny poziom abstrakcji zwany RTL (register transfer level) stanowi połączenie opisu behawioralnego i typu *dataflow*.

2.4. Obiekty

Moduł opisuje szablon wykorzystywany przy tworzeniu poszczególnych egzemplarzy danego modułu.

```
module ripple_carry_counter (q, clk, reset);
  output [3:0] q;
  input clk, reset;

  T_FF tff0(q[0], clk, reset) ;
  T_FF tff1(q[1], q[0] , reset) ;
  T_FF tff2(q[2], q[1] , reset) ;
  T_FF tff3(q[3], q[2] , reset) ;
endmodule

module T_FF(q, clk, reset);
  output q;
  input clk, reset;
  wire d;

  D_FF dff0(q, d, clk, reset);
  not n1(d,q);
endmodule
```

Nie wolno zagnieżdżać definicji modułów.

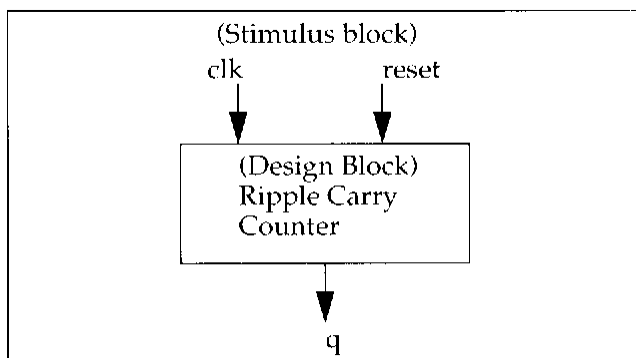
Moduł tylko opisuje działanie bloku układu oraz definiuje jego końcówki. Aby wykorzystać taki blok, należy go powołać do istnienia za pomocą deklaracji.

2.5. Symulacja

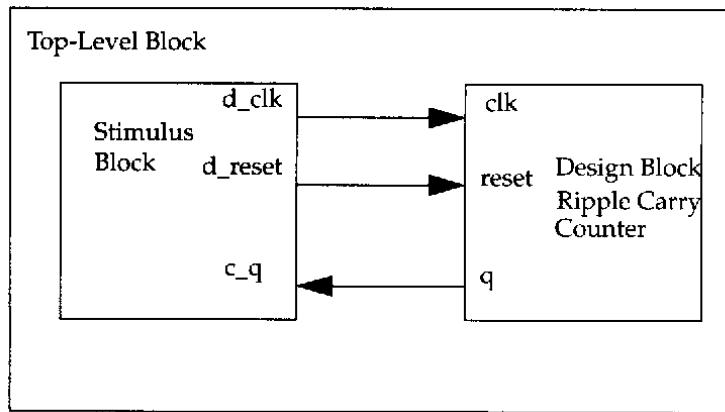
Po zdefiniowaniu modułu, dobrze jest go przetestować. Aby przetestować blok należy zdefiniować sygnały pobudzające. Można to zrobić w Verilog'u.

Stosuje się dwa podejścia:

- blok sygnałów powołuje do istnienia moduł testowany [1];



- wykorzystuje się pusty moduł, w którym osadza się zarówno blok badany jak i blok z sygnałami testującymi [1].



2.6. Przykład

Wykorzystując podejście typu *top-down*, zdefiniujemy w Verilog'u licznik.

2.6.1. Blok układu

```

module ripple_carry_counter(q, clk, reset);

    output [3:0] q;
    input clk, reset;
    T_FF tff0(q[0], clk, reset);
    T_FF tff1(q[1], q[0], reset);
    T_FF tff2(q[2], q[1], reset);
    T_FF tff3(q[3], q[2], reset);
endmodule

```

W powyższym przykładzie powołuje się do życia cztery obiekty T_FF (przerzutniki typu T). Należy je zdefiniować:

```

module T_FF(q, clk, reset);

    output q;
    input clock, reset;
    wire d;

    D_FF dff0(q, d, clk, reset);
    not n1(d, q);
endmodule

```

Ponieważ T_FF powołuje do życia obiekty D_FF, należy również opisać działanie modułu D_FF:

```

module D_FF(q, d, clk, reset);

    output q;
    input d, clk, reset;
    reg q;

    always @(posedge reset or negedge clk)
        if (reset)

```

```

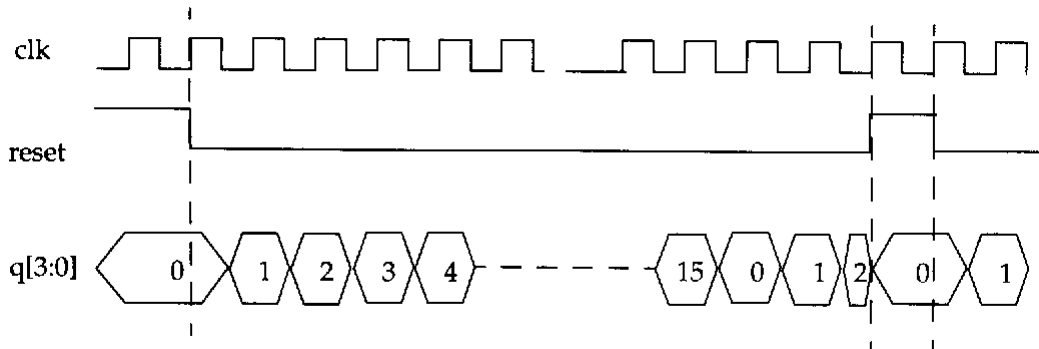
    q = 1'b0;
else
    q = d;

endmodule

```

2.6.2. Blok sygnałów pobudzających

Następnie należy zdefiniować blok sygnałów pobudzających, za ich pomocą stwierdzimy, czy licznik działa poprawnie. W bloku tym osadzony zostanie układ licznika. Wykorzystane zostaną następujące sygnały [1]:



```

module stimulus;

reg clk;
reg reset;
wire [3:0] q;

ripple_carry_counter r1(q, clk, reset);

initial
    clk = 1'b0;
always
    #5 clk = ~clk;

initial
begin
    reset = 1'b1;
    #15 reset = 1'b0;
    #180 reset = 1'b1;
    #10 reset = 1'b0;
    #20 $finish;
end

initial
    $monitor($time, " Output q=%d", q);

endmodule

```

Wynik działania układu:

```

0    Output q= 0
20   Output q= 1
30   Output q= 2
40   Output q= 3

```

50	Output q=	4
60	Output q=	5
70	Output q=	6
80	Output q=	7
90	Output q=	8
100	Output q=	9
110	Output q=	10
120	Output q=	11
130	Output q=	12
140	Output q=	13
150	Output q=	14
160	Output q=	15
170	Output q=	0
180	Output q=	1
190	Output q=	2
195	Output q=	0
210	Output q=	1
220	Output q=	2

3. PODSTAWOWE POJĘCIA

3.1. Składnia

Podstawowe zasady składni są podobne do zasad stosowanych w języku C.

Składnikami języka mogą być: komentarze, ograniczniki, liczby, łańcuchy, identyfikatory i słowa kluczowe.

Verilog rozdziela małe i duże litery.

Wszystkie słowa kluczowe pisane są małymi literami.

3.1.1. Biała spacja

Białe spacje: (spacje \b, tabulatory \t, koniec linii \n) są ignorowane, chyba, że oddzielają od siebie poszczególne składniki języka. Białe spacje nie są natomiast ignorowane w łańcuchach.

3.1.2. Komentarze

Komentarze jednoliniowe:

```
// dshdghsd
```

Komentarze na wiele linii:

```
/* fghds */
```

3.1.3. Operatory

Operatory mogą być następujących typów:

```
a = ~b;           // 1 argument
a = b && c;        // 2 argumenty
a = b ? c : d;     // 3 argumenty
```

3.1.4. Liczby

Liczby o określonym rozmiarze:

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

<rozmiar> może być określony tylko liczbą dziesiętną i określa ilość bitów w reprezentacji liczby.

<podstawa> dziesiętna 'd lub 'D

szesnastkowa 'h lub 'H

binarna 'b lub 'B

ósemkowa 'o lub 'O

przykłady:

```
4'b1111 // to jest 4-bitowa liczba dwójkowa
12'habc // to jest 12-bitowa liczba szesnastkowa
16'd255 // to jest 16-bitowa liczba dziesiętna
```

Liczby o nie określonej podstawie uznawane są za dziesiętne.

Liczby bez określonego rozmiaru uznawane są za 32-bitowe (zależne też od komputera).

```
23456 // liczba dziesiętna
'hc3  // 32-bitowa liczba szesnastkowa
'o21  // 32-bitowa liczba ósemkowa
```

x oznacza stan nieokreślony

z oznacza stan wysokiej impedancji

```
12'h13x // 12-bitowa liczba szesnastkowa,
          // 4 najmniej znaczące bity są nieokreślone
6'hx     // 6-bitowa liczba szesnastkowa
32'bz    // 32-bitowa liczba o wysokiej impedancji
```

x lub z oznacza 4 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.

```
-6'd3 // 8-bitowa liczba ujemna przechowywana jako uzupełnienie
dwójkowe.
```

W liczbach, aby zwiększyć ich czytelność, można stosować znak podkreślenia (byle nie na początku liczby).

Znak zapytania w liczbie jest odpowiednikiem stanu wysokiej impedancji z.

3.1.5. Łańcuchy

Łańcuch musi być zdefiniowany w jednej linii - nie dopuszcza się znaków CR w łańcuchu.

3.1.6. Identyfikatory i słowa kluczowe

Wszystkie słowa kluczowe pisane są małymi literami.

Identyfikatory określają wcześniej zdefiniowane obiekty. Identyfikatory mogą składać się tylko ze znaków alfanumerycznych, znaku podkreślenia "_" oraz znaku "\$". Identyfikatory nie mogą zaczynać się od "\$" ani od cyfry (od "\$" zaczynają się zadania systemowe).

```
reg value;
input clk;
```

3.1.7. Identyfikatory zaczynające się od \

Jeśli identyfikator zaczyna się od "\", to może zawierać dowolne znaki ACSII (koniec identyfikatora - biała spacja). Znak "\" oraz biała spacja nie są uznawane za część identyfikatora.

```
\a+b-c
\**my_name**
```

3.2. Typy danych

3.2.1. Zestaw wartości

Verilog obsługuje 4 wartości i osiem poziomów siły sygnałów.

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

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

Jeśli dwa sygnały o różnych siłach sterują jednym przewodem (*wire*), to wygrywa silniejszy sygnał.

Jeśli dwa sygnały mają równe siły, to na przewodzie jest sygnał nieokreślony **x**.

3.2.2. Sieci

Sieć reprezentuje połączenia pomiędzy elementami układu.

Sieci definiuje się słowem kluczowym *wire*. Standardowo przyjmują one długość 1-bitową, chyba, że zdefiniuje się je jako wektory. Standardową domyślną wartością dla sieci jest stan **z**.

```
wire a;  
wire b, c;  
wire d=1'b0 // siec d ma wartość 0 podczas deklaracji
```

3.2.3. Rejestry

Rejestry reprezentują elementy pamiętające. W Verilog'u rejestry oznaczają zmienne, które pamiętają ostatnio zapisaną wartość.

```
reg reset; // deklaracja zmiennej reset  
initial // będzie wyjaśnione później  
begin  
    reset=1'b1; // ustaw reset=1  
    #100 reset=1'b0; po 100 jednostkach czasu ustaw reset=0  
end
```

3.2.4. Wektory

Sieci (*wire*) lub rejestry (*reg*) mogą być deklarowane jako wektory (domyślnie są skalarami).

```

wire a; // skalar
wire [7:0] bus; // 8-bitowa magistrala
wire [31:0] busA, busB, busC;
reg [0:40] addr;

```

Wektory mogą być deklarowane jako *[max:min]* lub jako *[min:max]*, ale zawsze lewa liczba oznacza najbardziej znaczący bit.

Użycie części wektorów:

```

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

```

3.2.5. Liczby całkowite, rzeczywiste oraz czas

Liczby całkowite

```

integer counter;
initial
    counter = -1;

```

- deklaracja zmiennej, reprezentowanej jako (standardowa długość) 32-bitowa (zależnie od maszyny) liczba ze znakiem (można używać deklaracji **reg**, ale wtedy jest ona przechowywana jako liczba bez znaku).

Liczby rzeczywiste

Słowo kluczowe: *real*.

Mogą być definiowane w notacji dziesiętnej (3.14) i naukowej (3e6). Nie można deklarować zakresu, wartość domyślna 0.

Kiedy wartość rzeczywista jest przypisywana do całkowitej, jest ona zaokrąglana do najbliższej wartości całkowitej.

```

real data;
initial
begin
    delta=4e10;
end
integer i;
initial
    i=delta;

```

Czas

Symulacja w Verilog'u dokonywana jest względem czasu symulacji. Zmienne czasowe przechowywane są w rejestrach typu **time**. Szerokość reprezentacji takiej zmiennej zależy od implementacji, ale ma minimum 64 bity.

Aby uzyskać aktualną wartość czasu należy wywołać systemową funkcję **\$time**.

```

time save_sim_time;
initial
    save_sim_time=$time;

```

Symulacja mierzona jest w *sekundach symulacyjnych*. Jednak ich relacja do czasu rzeczywistego zależy już od użytkownika.

3.2.6. Tablice

Można definiować tablice dla następujących typów:

reg, **integer**, **time**. Nie wolno tego robić dla zmiennych typu **real**!

Dostęp do elementu macierzy odbywa się następująco:

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

Nie wolno definiować tablic wielowymiarowych.

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

count[5] // 5-ty element tablicy count
port<>id[3] // 3-ci element zmiennej (5-bitowej)
```

Nie należy mylić tablic z wektorami!

Wektor to pojedynczy element o szerokości n -bitów.

Tablica to zestaw wielu elementów o szerokości 1 lub n bitów.

3.2.7. Pamięci

Pamięci modeluje się jako tablice rejestrów. Każdy element tablicy to słowo. Każde słowo może mieć długość 1 lub więcej bitów.

3.2.8. Parametry

Verilog pozwala zdefiniować stałe jako parametry.

Parametry nie mogą być używane jako zmienne.

Wartości parametrów każdego obiektu mogą być nadane indywidualnie podczas kompilacji.

```
parameter port_id = 5;
```

Definicje modułów mogą być pisane w oparciu o parametry. Ich wartości można zmieniać przy powoływaniu obiektu do życia lub przy wykorzystaniu słowa kluczowego **defparam** (będzie omówione później).

3.2.9. Łańcuchy

Łańcuchy mogą być przechowywane w zmiennych typu **reg**. Szerokość rejestru musi być wystarczająca do przechowania łańcucha.

Każdy znak w łańcuchu zajmuje 8 bitów.

Jeśli łańcuch jest krótszy, Verilog uzupełnia bity z lewej strony zerami.

Jeśli łańcuch jest za długi, to Verilog obcina lewą część łańcucha.

```
reg[8*18:1] string_value; // zmienna o szer. 18 bajtów
initial
    string_value="Hello Verilog Word";
```

W łańcuchu można umieszczać znaki specjalne poprzedzone znakiem \:

```

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

```

3.3. Zadania systemowe i dyrektywy kompilatora

3.3.1. Zadania systemowe

Oznacza się je: **\$<słowo_kluczowe>** i wykonują one następujące operacje:

- wyświetlanie na ekranie;
- monitorowanie wartości sieci;
- pauzowanie;
- zatrzymywanie programu.

Wyświetlanie na ekran

\$display(*p1, p2, p3, . . . , pn*) - wyświetla zmienną lub łańcuch na ekranie.

Dyrektywa **\$display** na koniec wstawia znak nowej linii.

Skłania jest podobna do *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 (np. 2.13)
%g lub %G	wyświetl zmienną w postaci naukowej lub zmiennoprzecinkowej, w zależności od tego, która jest prostsza.

```

$display("Hello Verilog Word");
-- Hello Verilog Word

```

```

$display($time);
-- 230

```

```

$display("At time %d address is %h", $time, virtual_addr)
-- At time 200 address is 1fe0000001c

```

```

$display("ID of the port is %b", port_id);
-- ID of the port is 0010

```

```

$display("This string is displayed from %m level of hierarchy");
-- This string is displayed from top.pl level of hierarchy

```

```
$display("This is a \n multiline string with a %% sign");
-- This is a
-- multiline string with a % sign
```

Monitorowanie zmiennych

Po każdej zmianie sygnału będzie drukowana jego wartość.

\$monitor(*p1, p2, p3, ..., pn*);

Parametry podaje się w sposób podobny jak w poleceniu **\$display**. Różnicą w stosunku do **\$display** jest, że **\$monitor** wystarczy podać tylko raz.

Jeśli występuje więcej niż jedno polecenie **\$monitor**, to aktywne jest tylko ostatnie.

\$monitoron; - załącza monitorowanie,

\$monitoroff; - wyłącza monitorowanie.

Pauzowanie i koniec symulacji

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

\$finish; - kończy symulację.

3.3.2. Dyrektywy kompilatora

**`define** definiuje makro tekstowe. Verilog zastępuje każde wystąpienie *`<nazwa_makra>* określonym tekstem.

```
`define FALSE 1'b0
`define WORD_SIZE 32
`define S $stop;
`define WORD_REG reg [31:0]
```

Wykorzystanie makra:

```
assign out = `FALSE;
```

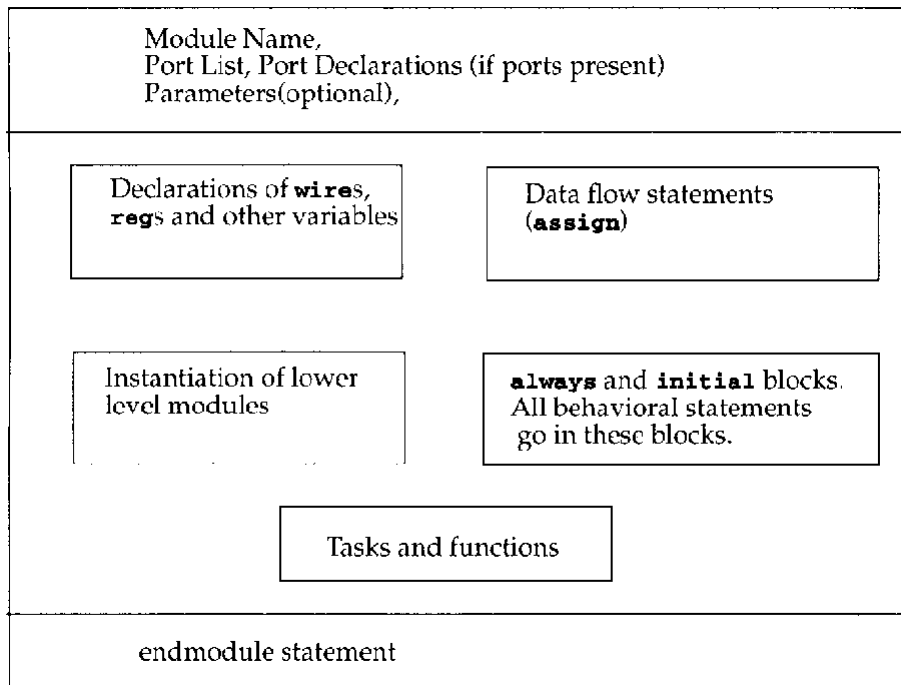
`include - pozwala dołączyć dowolny plik:

```
`include header.v
```

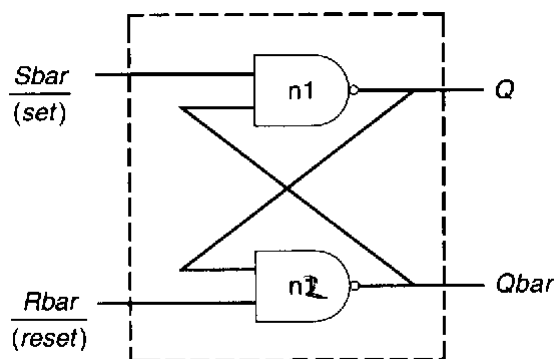
4. MODUŁY I PORTY

4.1. Moduły

Moduł składa się z części pokazanych na rysunku [1]:



Przykład opisu przerzutnika [1]:



```
// nazwa modułu i lista portów
module SR_latch(Q, Qbar, Sbar, Rbar);

// deklaracje portów
output Q, Qbar;
input Sbar, Rbar;

// powołanie do życia innych obiektów:
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

// słowo kluczowe endmodule
```

```

endmodule

// Moduł sygnałów:
module Top;
wire q, qbar;
reg set, reset;

//utworzenie modułów niższego poziomu:
SR_latch m1(q,qbar, ~set, ~reset);

// blok behawioralny
initial
begin
    $monitor($time, "set=%b, reset=%b, q=%b\n", set, reset, q);
    set=0; reset=0;
    #5 reset =1;
    #5 reset=0;
    #5 set=1:
end

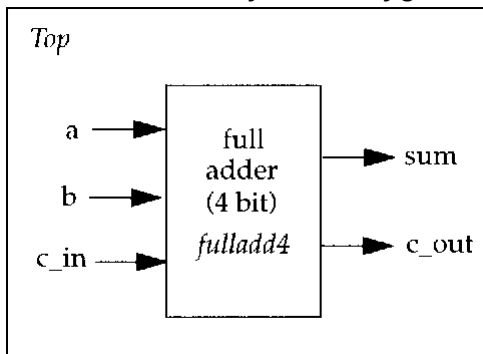
// słowo kluczowe endmodule:
endmodule

```

4.2. Porty

4.2.1. 4.2.1 Lista portów

Jeśli moduł nie wymienia sygnałów z otoczeniem, lista portów nie jest potrzebna [1].



Moduł *Top* jest modulem podstawowym. Moduł *fulladd4* jest tworzony wewnątrz modułu *Top*.

Moduł *fulladd4* posiada wejścia *a*, *b*, i *c_in* oraz wyjścia *sum* i *c_out*.

Moduł *Top* nie posiada wejść ani wyjść.

Lista portów dla modułów (moduł *Top* nie posiada listy portów):

```

module fulladd4(sum, c_out, a, b, c_in);
module Top;

```

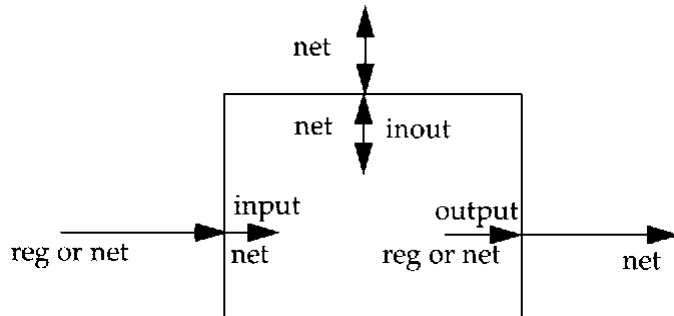
4.2.2. 4.2.2 Deklaracja portów

Wszystkie porty muszą zostać zadeklarowane jako **input**, **output** lub **inout**.

Deklaracje te niejawnie inicjują port jako **wire**. Zazwyczaj jest to wystarczające dla portów

typu **input** i **inout**. Port typu **output** można dodatkowo zadeklarować jako **reg**, aby mógł pamiętać wartość.

4.2.3. 4.2.3 Reguły łączenia portów [1]



Input

Wewnętrznie, porty **input** muszą być typu **wire**. Zewnętrznie można łączyć je ze zmiennymi typu **reg** i **wire**.

Output

Wewnętrznie porty **output** mogą być typu **reg** lub **wire**. Zewnętrznie muszą być podłączone do sieci typu **wire**. Nie mogą być podłączone do **reg**.

Inout

Wewnętrznie, takie porty muszą być typu **wire**. Zewnętrznie muszą być podłączone do sieci typu **wire**.

Dopasowanie szerokości

Można łączyć magistrale o różnej szerokości bitowej. Kompilator wygeneruje tylko ostrzeżenie.

Nie podłączone porty

Można zostawiać porty nie podłączone.

```
fulladd4 fa0(SUM, ,A, B, C_IN); // port c_out jest nie
podłączony
```

4.2.4. 4.3.4 Podłączanie portów do sygnałów zewnętrznych

Są dwie metody - nie mogą być stosowane jednocześnie.

I. Lista uporządkowana:

Podczas inicjacji obiektu porty muszą występować w tej samej kolejności jak w definicji modułu.

```
fulladd4 fa1 (SUM, C_OUT, A, B, C_IN);
```

II. Łączenie poprzez podanie nazwy:

Gdy w module występuje wiele portów, trudno ustalić ich kolejność - stosuje się wtedy inną metodę:

```
fulladd4 fa2(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN),
.a(A), );
```

5. MODELOWANIE NA POZIOMIE BRAMEK LOGICZNYCH

5.1. Typy bramek

Verilog posiada wbudowane definicje podstawowych bramek *and/or* i *buf/not*.

5.1.1. Bramki and/or

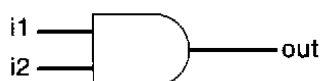
Bramki te posiadają jedno wyjście skalarne i wiele skalnych wejść. Pierwszą końcówką w liście portów bramki jest wyjście, po nim występują wejścia.

Wyjście bramki obliczane jest zaraz po tym, jak zmieni się jedno z wejść.

Dostępne bramki typu and/or:

and
nand
or
nor
xor
xnor

Standardowe symbole logiczne tych bramek przedstawia rysunek poniżej [1]:



and



nand



or



nor



xor



xnor

Bramki o większej ilości wejść inicjalizuje się poprzez podanie po prostu większej ilości wejść w liście portów.

```
wire OUT, IN1, IN2;  
and a1(OUT, IN1, IN2);  
nand na3(OUT, IN1, IN2, IN3);  
and (OUT, IN1, IN2) //można nie podawać nazwy bramki
```

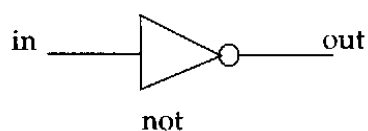
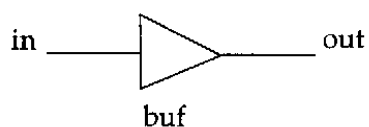
Tablice prawdy:

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

5.1.2. Bramki typu buf/ not [1]



Dwie takie bramki: **buf** i **not**. Mogą one mieć jedno wejście i wiele wyjść.

buf:

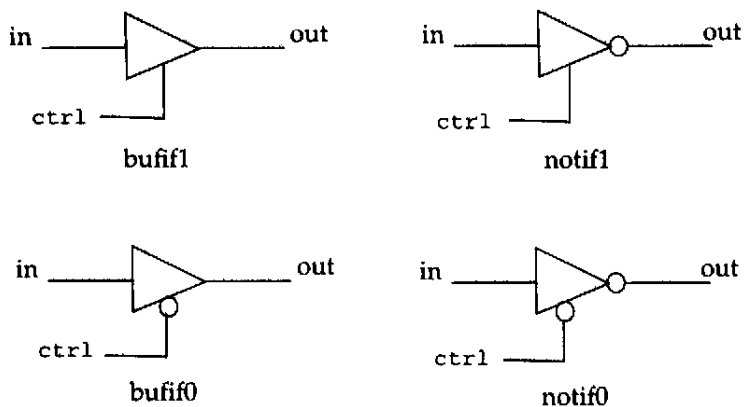
in	out
0	0
1	1
x	x
z	x

not

in	out
0	1
1	0
x	x
z	x

```
buf b1(OUT1, IN);
not n1(OUT1, IN);
buf b2(OUT1, OUT2, IN); // wiele wyjść
not (OUT1, IN); // nie trzeba podawać nazwy
```

Istnieją jeszcze bramki **bufif** i **notif**. Posiadają one dodatkowe wejście sterujące *ctrl* [1].



Ich tabele prawdy wyglądają następująco:

bufif1		ctrl			
in		0	1	x	z
	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

bufif0		ctrl			
in		0	1	x	z
	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

notif1		ctrl			
in		0	1	x	z
	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

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

(oznaczenia: L=0,z, H=1,z)

```

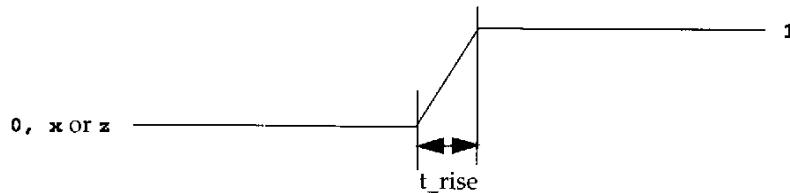
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);

```

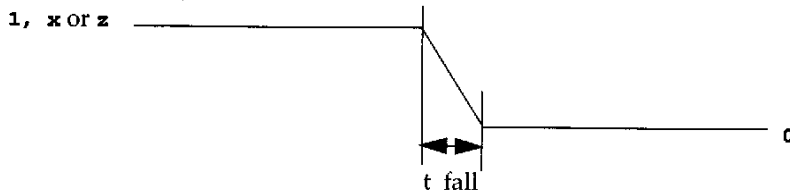
5.2. Opóźnienia w bramkach

5.2.1. Czas narastania, opadania i wyłączenia

Czas narastania jest to czas w jakim wyjście osiągnie stan 1 z dowolnego innego stanu [1].



Czas opadania jest to czas w jakim wyjście osiągnie stan 0 z dowolnego innego stanu [1].



Czas wyłączenia jest to 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.

Są trzy możliwości określenia czasów opóźnień:

- podaje się jedną wartość - jest ona używana jako czas wszystkich trzech przejść;
- podaje się dwie wartości - oznaczają one czasy narastania i opadania, czas wyłączenia jest równy mniejszej z tych liczb.
- podaje się trzy wartości - określają one 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 #(delay_time) a1(out, i1,i2);

// określenie czasu narastania i opadania:
and #(rise_val, fall_val) a2(out, i1,i2);

// określenie trzech czasów:
bufif0 #(rise_val, fall_val, off_val) b1(out, in, control);
```

5.2.2. Wartości minimalne, typowe i maksymalne

Dla każdego typu opóźnień można zdefiniować dodatkowo trzy wartości: *min*, *typ* i *max*. Określają one rozrzuty parametrów bramek podczas produkcji układu scalonego.

Sposób określania, którą z tych wartości wybrać, może być różny dla różnych symulatorów. Np. dla symulatora Verilog-XL podaje się odpowiednie słowo kluczowe: **+maxdelays**, **+typdelay** lub **+mindelays** podczas symulacji.

Przykład:

```
// jedno opóźnienie:
// jeśli +mindelays to opóźnienie=4
// jeśli +typdelay to opóźnienie=5
```

```
// jeśli +maxdelays to opóźnienie=6
and #(4:5:6) a1(out, i1,i2);

// dwa opóźnienia:
// jeśli +mindelays to czas narast.=3, czas opad.=5, czas
wyl.=min(3,5)
// jeśli +typdelay to czas narast.=4, czas opad.=6, czas
wyl.=min(4,6)
// jeśli +maxdelays to czas narast.=5, czas opad.=7, czas
wyl.=min(5,7)
and #(3:4:5, 5:6:7) a2(out, i1, i2);

// trzy opóźnienia:
// jeśli +mindelays to czas narast.=2, czas opad.=3, czas
wyl.=4
// jeśli +typdelay to czas narast.=3, czas opad.=4, czas wyl.=5
// jeśli +maxdelays to czas narast.=4, czas opad.=5, czas
wyl.=6
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1, i2);

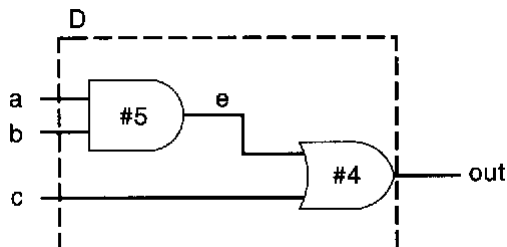
and #(4:5:6) a1(out, i1,i2);
```

5.2.3. Przykład układu z opóźnieniami

Moduł D ma realizować funkcję:

$$out=(a*b)+c$$

Implementacja została pokazana na rysunku poniżej [1]:



```
// definicja modułu D:
module D(out, a, b, c);

// deklaracje portów wej/wyj
output out;
input a, b, c;

// sieć wewnętrzna:
wire e;

// bramki:
and #(5) a1(e, a, b);
or #(4) o1(out, e, c);
endmodule
```

Plik do testowania modułu D:

```

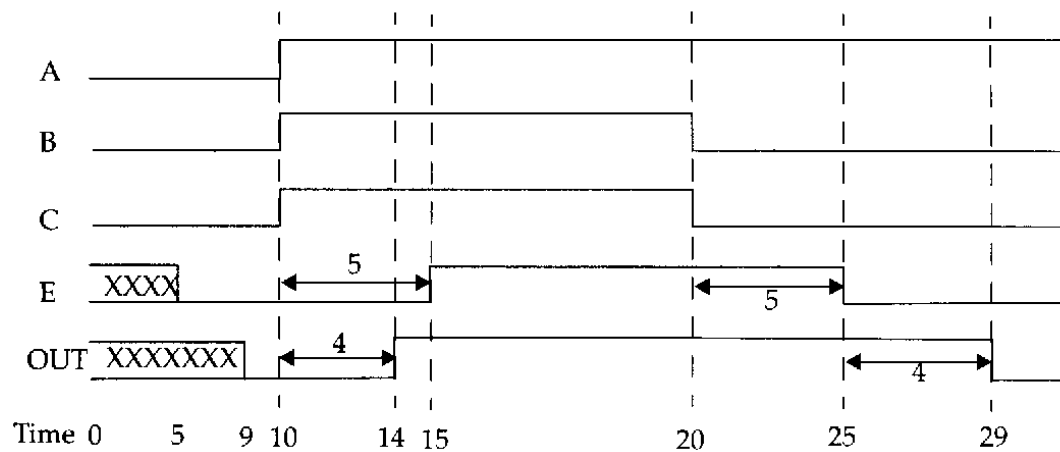
module stimulus;

// deklaracja zmiennych:
reg A, B, C;
wire OUT;

// powołanie modułu D:
D d1(OUT, A, B, C);

// pobudzanie wejść, koniec symulacji po 40 jednostkach
czasowych
initial
begin
    A= 1'b0; B= 1'b0; C= 1'b0;
    #10 A= 1'b1; B= 1'b1; C= 1'b1;
    #10 A= 1'b1; B= 1'b0; C= 1'b0;
    #20 $finish;
end
endmodule

```



6. MODELOWANIE NA POZIOMIE REJESTRÓW (Dataflow)

6.1. Przypisanie ciągłe

Przypisanie ciągłe to najpowszechniej używane polecenie służące do sterowania sieci jakąś wartością logiczną. Przypisanie zastępuje bramki i opisuje układ na wyższym poziomie abstrakcji.

Składnia:

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

Definiowanie siły sygnału podczas przypisania ciągłego jest opcjonalne (domyślnie **strong1** i **strong0**) tak, jak i opóźnienia (podaje się je jak w bramkach).

Cechy przypisania ciągłego:

1. Lewa strona musi być siecią (skalar lub wektor), albo musi być efektem konkatenacji sieci skalarych lub wektorowych.
2. Przypisanie ciągłe jest zawsze aktywne. Wyrażenie obliczane jest zawsze, gdy któraś ze zmiennych z prawej strony zmieni swą wartość. Wynik jest natychmiast przesyłany do sieci określonej z lewej strony.
3. Wyrażenia z prawej strony mogą być rejestrami, sieciami lub wywołaniami funkcji. Rejestry lub sieci mogą być skalarami lub wektorami.

Przykłady:

```
assign out = i1 & i2;  
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];  
// nawiasy klamrowe oznaczają łączenie (konkatenacja)  
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

6.1.1. Niejawne przypisanie ciągłe

Zamiast najpierw deklarować sieć a potem wykonywać do niej operację przypisania

```
wire out;  
assign out = in1 & in2;
```

można to zrobić krócej:

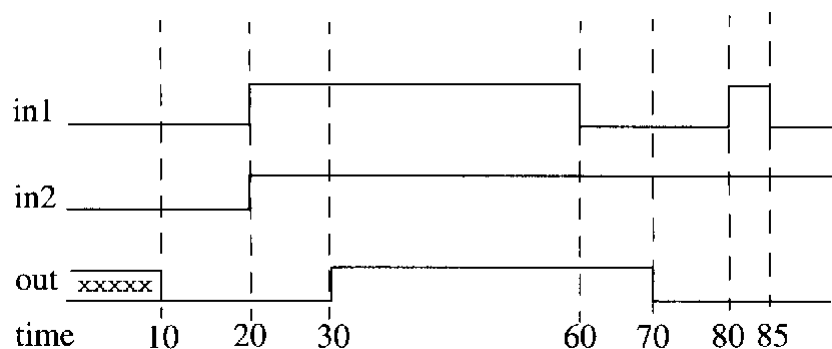
```
wire out = in1 & in2;
```

6.2. Opóźnienia

Opóźnienia określają czas, po którym nastąpi uaktualnienie lewej strony przypisania spowodowane zmianą prawej strony. Są trzy metody definiowania opóźnień.

6.2.1. Opóźnienie w instrukcji przypisania

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



Uwaga: W powyższym przykładzie [1] impulsy krótsze niż 10 jedn. czasowych nie przechodzą na wyjście.

6.2.2. Niejawne określenie opóźnienia przypisania

Równoważną metodą określania opóźnień do:

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

jest określenie:

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

6.2.3. Określenie opóźnienia sieci

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

Opis:

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

jest równoważny następującemu:

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

6.3. Wyrażenia, operandy i operatory

6.3.1. Wyrażenia

Wyrażenia są konstrukcjami zawierającymi operatory i operandy, zwracającymi jakiś wynik. Przykłady wyrażeń:

```
a^b
addr[20:17]+addr2[20:17]
in1 | in2
```

6.3.2. Operandy

Operandy mogą być jednym z typów danych wymienionych w rozdziale 3.2. Operandy mogą być *stałymi*, *liczbami całkowitymi*, *liczbami rzeczywistymi*, *sieciami*, *rejestrami*, *czasem*, *zakresem bitów* (pojedynczym bitem lub zakresem), *pamięciami* lub *wywołaniami funkcji*

6.3.3. Operatory

Operatory działają na operandach w celu obliczenia wyniku. Lista operatorów podana jest w następnym rozdziale.

6.4. 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
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

6.4.1. Operatory arytmetyczne

Operatory binarne:

```
A= 4'b0011; B= 4'b0100;  
D=6; E= 4;
```

```
A*B    // mnożenie, wynik: 4'b1100
```

```
D/E    // dzielenie, wynik: 1 (część ułamkowa jest ucięta)
A+B    // dodawanie, wynik: 4'b0111
B-A    // odejmowanie, wynik: 4'b0001
```

Jeśli jakiś argument ma wartość **x**, to wynik też będzie miał wartość **x**.
Operator modulo zwraca resztę z dzielenia dwóch liczb (działa podobnie jak w C).

```
13%3   // wynik: 1
16%4   // wynik: 0
-7%2   // wynik: -1 (znak z pierwszego argumentu)
7%-2   // wynik: 1 (znak z pierwszego argumentu)
```

Operatory jednoargumentowe

```
-4 // MINUS 4
+5 // PLUS 5
```

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).

```
-'d10 / 5 // tj.  $(2^{32}-10)/5$ 
```

6.4.2. Operatory logiczne

&& - 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).

```
A=3; B=0;
A&&B // wynik: 0
A||B // wynik: 1
!A // wynik: 0
!B // wynik: 1
```

```
A=2'b0z; B=2'b10
A&&B // wynik: x
(a==2)&&(b==3) // wynik: 1 jeśli a=2 oraz b=3, w przeciwnym
wypadku wynik: 0
```

6.4.3. Operatory porównania

Jeśli w wyrażeniu użyto operatorów porównania, wyrażenie zwraca wartość logiczne 1, logiczne 0 lub **x**.

```
// A=4, B=3
// X=4'b1010, Y=4'b1101, Z=4'b1xxx
```

$A \leq B$ // wynik: 0
 $A > 0$ // wynik: 1
 $Y \geq X$ // wynik: 1
 $Y < Z$ // wynik: x

6.4.4. 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.

6.4.5. Operatory bitowe

$\sim$	negacja bit po bicie
$\&$	and bit po bicie
$ $	or bit po bicie
$\wedge$	xor bit po bicie
$\wedge \sim$ lub $\sim \wedge$	xnor bit po bicie

Operacje wykonywane są według tabel podanych poniżej. Wartość z jest traktowana tak jak x.

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

6.4.6. Operatory redukujące

Operatory redukujące (& - and, ~& - nand, | - or, ~| - nor, ^ - xor, ^~ , ~^ - xnor) wymagają tylko jednego argumentu. Operują na bitach wykonując odpowiednie operacje według tabel jak dla operatorów bitowych, działają od prawej do lewej.

```
// X=4'b1010
&X // 1&0&1&0 = 1'b0
|X // 1|0|1|0 = 1'b1
^X // 1^0^1^0 = 1'b0
```

Redukcja **xor** lub **xnor** może być wykorzystana przy obliczaniu parzystej lub nieparzystej ilości bitów w wektorze.

6.4.7. Operatory przesuwania

>> - przesunięcie w prawo

<< - przesunięcie w lewo

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

```
Y=X<<2 //przesuń X 2 bity w lewo
```

6.4.8. 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.

```
// A=1'b1, B = 2'b00 C = 2'b10, D = 3'b110
Y={B, C} // wynik: Y=4'B0010
Y={A, B, C, D, 3'B001} // wynik: Y=11'b100100110001
Y={A, B[0], C[1]} // wynik: 3'b101
```

6.4.9. Operator replikacji

Wielokrotne łączenie tego samego argumentu 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'b00; C=2'b10;
```

```
Y={4{A}} // wynik: Y = 4'b1111
Y={4{A, 2{B}}} // wynik: Y=8'b11110000
Y={4{{A}, 2{B}, C}} // wynik: Y=8'b1111000010
```

6.4.10. Operator warunkowy

Operator warunkowy wymaga podania trzech argumentów:

```
warunek ? wyrażenie_true : wyrażenie_false;
```

Najpierw obliczany jest *warunek*, jeśli wynik obliczeń jest prawdą (*true* - logiczna 1), to wykonywane jest *wyrażenie_true*, w przeciwnym wypadku wykonywane jest *wyrażenie_false*. Jeśli wynik wyrażenia jest *x*, to obliczane są obydwa wyrażenia *wyrażenie_true* oraz *wyrażenie_false*, a następnie są porównywane bit-po-bicie. Jeśli bity się nie zgadzają, to w wyniku na tym miejscu będzie wartość bitu *x*.

Operator ten przypomina w pracy multiplexer lub polecenie *if-then-else*. Wykorzystuje się go często do przypisań warunkowych.

```
// model bufora trójstanowego
assign addr_buf = drive_enable ? addr_out : 36'bz;

// model multiplexera 2-do-1
assign out = control ? in1 : in0;
```

Operatory warunkowe mogą być zagnieżdżane:

```
assign out = (A==3) ? (control ? x : y) : (control ? m : n);
```

6.4.11. Kolejność operatorów

Priorytet operatorów (od największego):

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

6.5. Przykłady

Poprzednie przykłady multiplexera 4-do-1 oraz sumatora opierały się o schemat połączeń bramek. W tym rozdziale podane zostaną przykłady opisane na poziomie *data-flow*.

6.5.1. Multiplexer 4-do-1

Sposób 1: równania logiczne

```
// Lista portów taka sama, jak w przykładzie opartym na
// schemacie połączeń bramek.
```

```

module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Deklaracja portów:
output out;
input i0, i1, i2, i3;
input s1, s0;

// Równania logiczne:
assign out = (~s1 & ~s0 & i0) |
              (~s1 & s0 & i1) |
              (s1 & ~s0 & i2) |
              (s1 & s0 & i3);

endmodule

```

Sposób 2: operator warunkowy

```

// Lista portów taka sama, jak w przykładzie opartym na
// schemacie połączeń bramek.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Deklaracja portów:
output out;
input i0, i1, i2, i3;
input s1, s0;

// Wykorzystanie zagnieżdżonych operatorów warunkowych:
assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule

```

6.5.2. Sumator 4-bitowy

Sposób 1: operatory *dataflow*

```

// module fulladd4 (sum, c_out, a, b, c_in);

// Porty I/O:
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;

// Określenie funkcji układu:
assign {c_out, sum} = a + b + c_in;

endmodule

```

Sposób 2: Sumator „carry lookahead”

W tym sumatorze przepełnienie pojawia się na wyjściu szybciej niż dane.

```

module fulladd4(sum, c_out, a, b, c_in);

// Porty I/O:
output [3:0] sum;
output c_out;

```

```

input [3:0] a, b;
input c_in;

// linie wewnętrzne:
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c4, c3, c2, c1;

// Obliczanie p dla każdego stopnia:
assign p0 = a[0] ^ b[0],
       p1 = a[1] ^ b[1],
       p2 = a[2] ^ b[2],
       p3 = a[3] ^ b[3];

// Obliczanie g dla każdego stopnia:
assign g0 = a[0] & b[0],
       g1 = a[1] & b[1],
       g2 = a[2] & b[2],
       g3 = a[3] & b[3];

// Obliczanie przeniesienia dla każdego stopnia:
assign c1 = g0 | (p0 & c_in),
       c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) |
           (p2 & p1 & p0 & c_in),
       c4 = g3 | (p3 & g2) | (p3 & p2 & g1) |
           (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & c_in);

// Obliczenie sumy:
assign sum[0] = p0 ^ c_in,
       sum[1] = p1 ^ c1,
       sum[2] = p2 ^ c2,
       sum[3] = p3 ^ c3;

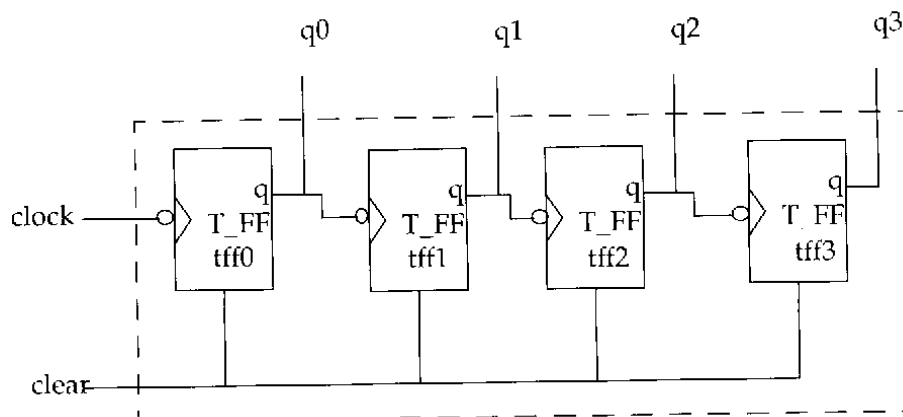
// Przypisanie przeniesienia:
assign c_out = c4;

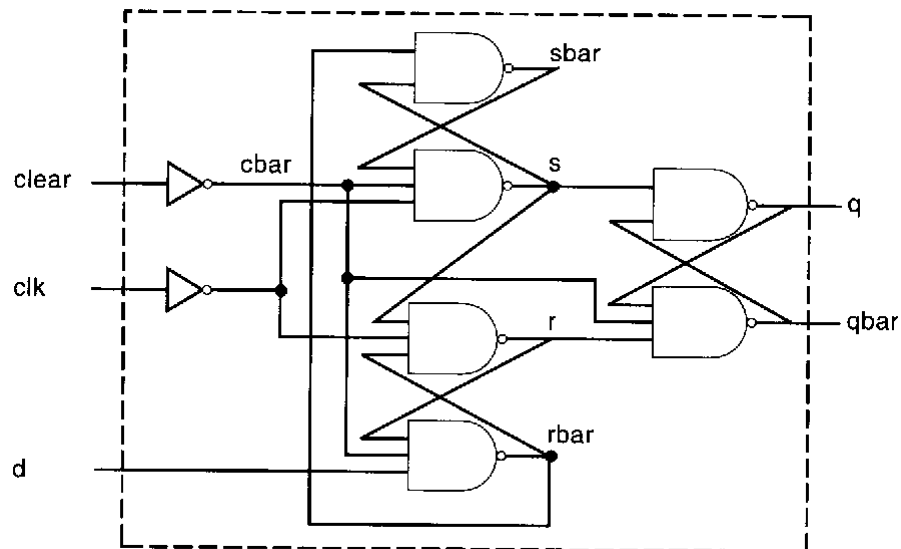
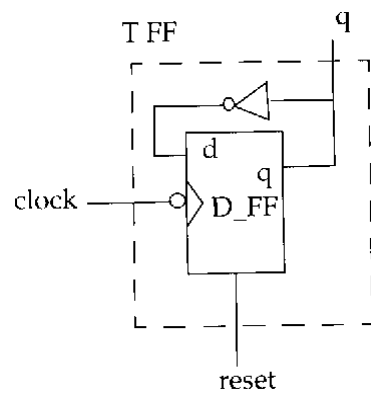
endmodule

```

6.5.3. Licznik

Przykładowy projekt 4-bitowego licznika z wykorzystaniem przerzutników wyzwalanych zboczem opadającym [1]:





```
// Ripple counter
module counter(Q , clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```

```

// Edge-triggered T-flipflop. Toggles every clock
// cycle.
module T_FF(q, clk, clear);

// I/O ports
output q;
input clk, clear;

// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ~q, clk, clear);

endmodule

```

```

// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);

// Inputs and outputs
output q,qbar;
input d, clk, clear;

// Internal variables
wire s, sbar, r, rbar,cbar;

// dataflow statements
//Create a complement of signal clear
assign cbar = ~clear;

// Input latches; A latch is level sensitive. An edge-sensitive
// flip-flop is implemented by using 3 SR latches.

```

```

assign  sbar = ~(rbar & s),
        s = ~(sbar & cbar & ~clk),
        r = ~(rbar & ~clk & s),
        rbar = ~(r & cbar & d);

// Output latch
assign  q = ~(s & qbar),
        qbar = ~(q & r & cbar);

endmodule

```

```

// Top level stimulus module
module stimulus;

// Declare variables for stimulating input
reg CLOCK, CLEAR;
wire [3:0] Q;

initial
    $monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);

// Instantiate the design block counter
counter c1(Q, CLOCK, CLEAR);

// Stimulate the Clear Signal
initial
begin
    CLEAR = 1'b1;
    #34 CLEAR = 1'b0;
    #200 CLEAR = 1'b1;
    #50 CLEAR = 1'b0;
end

// Set up the clock to toggle every 10 time units
initial
begin

```

```

    CLOCK = 1'b0;
    forever #10 CLOCK = ~CLOCK;
end

// Finish the simulation at time 400
initial
begin
    #400 $finish;
end

endmodule

```

```
0 Count Q = 0000 Clear= 1
34 Count Q = 0000 Clear= 0
40 Count Q = 0001 Clear= 0
60 Count Q = 0010 Clear= 0
80 Count Q = 0011 Clear= 0
100 Count Q = 0100 Clear= 0
120 Count Q = 0101 Clear= 0
140 Count Q = 0110 Clear= 0
160 Count Q = 0111 Clear= 0
180 Count Q = 1000 Clear= 0
200 Count Q = 1001 Clear= 0
220 Count Q = 1010 Clear= 0
234 Count Q = 0000 Clear= 1
284 Count Q = 0000 Clear= 0
300 Count Q = 0001 Clear= 0
320 Count Q = 0010 Clear= 0
340 Count Q = 0011 Clear= 0
360 Count Q = 0100 Clear= 0
380 Count Q = 0101 Clear= 0
```

7. MODELOWANIE NA POZIOMIE BEHAWIORALNYM

Wraz ze wzrostem złożoności układu, potrzebna stała się możliwość oceny układu na wczesnym etapie projektowania, bez konieczności rozstrzygania detali implementacyjnych. Projektantom zależało na ocenie działania algorytmu. Do tego służy behawioralny poziom opisu, który bardziej przypomina programowanie w C niż projektowanie *hardware'u*.

7.1. Procedury strukturalne

Rozróżnia się dwa typy procedur strukturalnych: **always** i **initial**, które zazwyczaj zawierają w sobie wszystkie polecenia opisu behawioralnego.

Verilog jest językiem o charakterze równoległym, w przeciwieństwie do sekwencyjnego języka C. Polecenia w Verilog u wykonywane są jednocześnie. Każde z procedur **always** i **initial** reprezentują osobny ciąg poleceń, obydwa rozpoczynają się w czasie 0, nie mogą być zagnieżdżane.

7.1.1. Polecenie initial

Wszystkie polecenia wewnątrz polecenia **initial** tworzą tzw. **initial block**. Taki blok wykonuje się tylko jeden raz podczas symulacji, począwszy od czasu 0. Działanie każdego bloku **initial** kończy się niezależnie od pozostałych. Instrukcje wewnątrz bloku muszą być zgrupowane za pomocą **begin** i **end**. Jeśli w bloku występuje tylko jedno polecenie, grupowanie jest zbędne (podobnie jak w Pascalu *begin* i *end*, a w C { }).

```
module stimulus;
  reg x, y, a, b, m;
  initial
    m = 1'b0;

  initial
  begin
    #5 a=1'b1;
    #25 b=1'b0;
  end

endmodule
```

Polecenia wykonywane są po kolei, a opóźnienie czasowe #<czas> oznacza opóźnienie względem aktualnego czasu symulacji. W powyższym przykładzie występują dwa bloki **initial**, rozpoczynają się one w czasie 0:

czas	wykonane polecenie
0	m=1'b0;
5	a=1'b1;
30	b=1'b0;

Bloki **initial** wykorzystuje się m.in. do inicjalizacji, monitorowania.

7.1.2. Polecenie always

Polecenie **always** wraz z instrukcjami stanowi blok **always**. Wszystkie bloki **always** rozpoczynają działanie równocześnie w czasie 0 i są od siebie niezależne. Wykonują się one w sposób ciągły.

```

module clock_gen;

reg clock;

//inicjalizacja zegara w czasie 0
initial
    clock = 1'b0;

//przełączanie stanu zegara co pół okresu:
always
    #10 clock = ~clock;

initial
    #1000 $finish;
endmodule

```

7.2. 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 **assignment** gdzie przypisanie ma charakter ciągły.

<lvalue> = <wyrażenie>

Lewa część przypisania może być:

- zmienna typu **reg**, **integer**, **real**, **time**, **memory**;
- wybrany bit z powyższych zmiennych;
- zestaw bitów;
- połączenie powyższych elementów.

Wyróżnia się dwa typy przypisań: *blocking* i *nonblocking*.

7.2.1. Przypisanie *blocking* (=)

Przypisania typu *blocking* są wykonywane według kolejności ich umieszczenia w programie. Nie wstrzymują one działania innych, równoległe działających, bloków.

```

reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
// wszystkie wyrażenia behawioralne muszą być
// w bloku initial lub always
initial
begin
    x=0; y=1; z=1; //przypisania skalarów
    count = 0; // przypisanie zmiennej integer
    reg_a = 16'b0; reg_b = reg_a //

    #15 reg_a[2] = 1'b1;
    #10 reg_b[15:13] = {x, y, z};
    count = count + 1;
end

```

W powyższym przykładzie przypisanie $y=1$ odbędzie się dopiero wtedy, gdy wykona się

przypisanie $x=0$. Wykorzystanie instrukcji typu *blocking* powoduje, że blok wykonuje się sekwencyjnie.

7.2.2. Przypisanie *nonblocking* ($\leq$)

Przypisania typu *nonblocking* pozwalają na wykonanie przypisań bez wstrzymywania działania pozostałych instrukcji. Symbol $\leq$ jest taki sam, jak operator warunkowy mniejsze lub równe. Znaczenie symbolu rozstrzygane jest w zależności od kontekstu.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
// wszystkie wyrażenia behawioralne muszą być
// w bloku initial lub always
initial
begin
    x=0; y=1; z=1;
    count = 0;
    reg_a=16'b0; reg_b=reg_a;
    reg_a[2] <= #15 1'b1;
    reg_b[15:13] <= #10 {x, y, z};
    count <= count + 1;
end
```

Wszystkie przypisania typu *nonblocking* będą wykonane równocześnie - tj. zostaną zaplanowane do wykonania po odpowiednim opóźnieniu. Zazwyczaj symulator wykonuje wszystkie przypisania *nonblocking* na końcu danego kroku czasowego.

Zastosowanie przypisań *nonblocking*

Przypisań *nonblocking* używa się do modelowania równoczesnego przepływu kilku strumieni danych po jakimś zdarzeniu. W poniższym przykładzie, po dodatnim zboczu zegara, wykonane zostaną trzy równoległe przepływy informacji:

```
always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2^in3;
    reg3 <= #1 reg1; //stara wartość reg1
end
```

Po każdym narastającym zboczu zegara następują następujące operacje:

1. Odczyt wartości z prawej części przypisań, tj. *in1*, *in2*, *in3* oraz *reg1*. Wyrażenia te są obliczane i zapamiętywane wewnętrznie przez symulator.
2. Zapis do zmiennych określonych po lewej stronie równań zostanie rozdysponowany do wykonania zgodnie z odpowiednim opóźnieniem czasowym.
3. Zapis rozdysponowanych przypisań po opóźnieniu. Kolejność w jakiej się on wykona nie jest istotny, ponieważ zapisywane wartości są pamiętane przez symulator i przebieg programu nie ma na niego wpływu.

Następny przykład pokazuje różnicę pomiędzy przypisaniami typu *blocking* i *nonblocking*.

```
//Przykład I: Dwa równoczesne bloki always
// z przypisaniami blocking
always @(posedge clock)
```

```

a = b;

always @(posedge clock)
    b = a;

//Przykład II: Dwa równoczesne bloki always
// z przypisaniami nonblocking
always @(posedge clock)
    a <= b;

always @(posedge clock)
    b <= a;

```

W przykładzie I występuje zjawisko hazardu. Z zależności od tego które z przypisań $a=b$ czy $b=a$ zostanie wykonane pierwsze, to wynik będzie różny a dodatkowo wartości zmiennych a i b nie zostaną zamienione, lecz będą miały taką samą wartość.

Użycie przypisań *nonblocking* w przykładzie II eliminuje zjawisko hazardu. Obydwie instrukcje będą wykonane równocześnie, ale z wykorzystaniem dwóch cykli: odczytu do zmiennych tymczasowych oraz zapisu. Dzięki temu operacja zamiany wartości zmiennych a i b odbędzie się prawidłowo.

7.3. Sterowanie wykonywaniem instrukcji

7.3.1. Opóźnienia

Opóźnienie określa czas pomiędzy „napotkaniem” wyrażenia przez symulator a jego wykonaniem.

```

# <NUMBER>
# <identyfikator>
#(<wyrażenie mintypmax>: [<wyrażenie mintypmax>: <wyrażenie
mintypmax>])

```

Dla wyrażień proceduralnych wyróżnia się trzy typy opóźnień: zwykłe, wewnętrzne i zerowe.

Opóźnienie zwykłe

```

// definicja parametrów
parameter opoznienie = 20;
parameter delta = 2;

reg x, y, z, p, q;

initial
begin
    x=0; // brak opóźnienia
    #10 y=1; // opóźnienie o 10 jedn.czasowych
    #opoznienie z=0; // opóźnienie określone parametrem
    #(opoznienie+delta) p=1; // opóźnienie określone wyrażeniem

    #y x=x+1 // opóźnienie określone zmienną
    #(4:5:6) q=0 // minimalne, typowe i maksymalne opóźnienie,

```

```

    //szczegóły w rozdziale o opóźnieniach w brawkach
end

```

Dla grupy **begin-end**, opóźnienie jest odliczane zawsze względem czasu rozpoczęcia wykonywania grupy.

Opóźnienie wewnętrzne

Zamiast określać opóźnienie całego wyrażenia, można kontrolować opóźnienie przypisane do prawej części przypisania. Osiąga się w ten sposób trochę inne działanie programu.

```

// definicja rejestrów
reg x, y, z;

//opóźnienia wewnętrzne
initial
begin
    x=0; z=0;
    y = #5 x+z; // pobierz wartość x i z w czasie=0,
                // oblicz x_z i poczekaj 5 jedn.
                // czasu a następnie zapisz wynik do y.
end

// odpowiednik z wykorzystaniem opóźnień zwykłych oraz
// zmiennych tymczasowych
initial
begin
    x=0; z=0;
    temp_xz = x+z;
    #5 y=temp_xz; // pobierz wartość x+z w czasie bieżącym
                  // i zapisz w zmiennej tymczasowej.
end

```

Opóźnienie zerowe

Może się zdarzyć, że wyrażenia proceduralne w różnych blokach **always/initial** będą wykonane w tym samym kroku czasu. Kolejność ich wykonania nie jest w takim przypadku określona. Aby wyeliminować możliwość hazardu, można zadać opóźnienie zerowe, co oznacza, że dana instrukcja wykona się jako ostatnia w danym kroku czasu. Oczywiście, jeśli będzie więcej takich instrukcji, to będą one wykonane jako ostatnie, ale kolejność między nimi nie jest określona.

```

initial
begin
    x = 0;
    y = 0;
end;

initial
begin
    #0 x = 1;
    #0 y = 1;
end

```

Pod koniec czasu 0 zmienne x i y będą miały wartość 1. Kolejność wykonania instrukcji x=1 i y=1 jest nieokreślona.

7.3.2. Sterowanie przy użyciu zdarzeń

Zdarzeniem jest zmiana wartości rejestru lub sieci. Zdarzenia mogą być wykorzystane do wyzwolenia wykonania bloku instrukcji. Są 4 typy sterowania zdarzeniem: *zwykłe sterowanie przy użyciu zdarzeń*, *sterowanie przy użyciu nazwanych zdarzeń*, *sterowanie przy użyciu zdarzeń typu OR*, *sterowanie poziomem sygnału*.

Zwykłe sterowanie przy użyciu zdarzeń

Do sterowania zdarzeniami służy symbol @. Instrukcje mogą reagować na: zmianę wartości sygnału, rosnące (**posedge**) lub opadające (**negedge**) zbocze.

```
@clock q = d; //q=d przy każdej zmianie sygnału clock
@(posedge clock) q = d; // q=d gdy clock zmienia wartość:
                        // 0->1, 0->x, 0->z, x->1, z->1
@(negedge clock) q = d; // q=d gdy clock zmienia wartość:
                        // 1->0, 1->x, 1->z, x->0, z->0
q=@(posedge clock) d; // d jest obliczane
                        // natychmiastowo,
                        // a wartość jest przypisana do q
                        // podczas rosnącego zbocza sygnału
                        //clock.
```

Sterowanie przy użyciu nazwanych zdarzeń

Verilog umożliwia zdeklarowanie jakiegoś zdarzenia, a następnie reagowanie na jego wystąpienie. Wyrażenie nazywa się słowem kluczowym **event**. Zdarzenie wyzwala się za pomocą symbolu ->. Wykrywanie wyzwolenia zdarzenia odbywa się dzięki znakowi @.

```
event received_data; // definicja zdarzenia
always @(posedge clock) // kontrola zdarzenia podczas
                        // dodatniego zbocza
begin
    if(last_data_packet) // jeśli to jest ostatni pakiet
        ->received_data; // to wyzwól zdarzenie
end

always @(received_data) // oczekuj na zdarzenie
                        // „received data”
    data_buf={data_pkt[0], data_pkt[1], data_pkt[2],
data_pkt[3]};
```

Sterowanie przy użyciu zdarzeń typu OR

Czasami wyzwolenie może nastąpić poprzez jeden z kilku sygnałów:

```
// przerzutnik wyzwala się poziomem z asynchronicznym resetem
always @(reset or clock or d)
    // czekaj na zmianę sygnału reset, clock lub d
begin
    if (reset) //jeśli sygnał reset ma stan wysoki, ustaw q=0
        q = 1'b0;
    else if (clock) //jeśli clock ma stan wysoki
        q = d;
end
```

7.3.3. Sterowanie poziomem sygnału

Poprzednie sposoby umożliwiały wykrywanie zmian w sygnale za pomocą symbolu @. Słowo kluczowe **wait** umożliwia oczekiwanie na określoną wartość sygnału.

```
always
    wait (count_enable) #20 count = count + 1;
```

W powyższym przykładzie wartość *count_enable* jest monitorowane cały czas. Jeśli *count_enable* ma wartość **0**, to wyrażenie nie jest wykonywane. Jeśli *count_enable* ma wartość **1** to wyrażenie *count = count + 1* zostanie wykonane po 20 jednostkach czasu i będzie powtarzane co 20 jednostek, dopóki wartość *count_enable* ma wartość **1**.

7.4. Wyrażenia warunkowe

Wyrażenie warunkowe ma postać **if .. else**.

```
// typ I - brak else
if (<wyrażenie>) polecenie_true;

// typ II - jedno else
if (<wyrażenie>) polecenie_true; else polecenie_false;

// typ III - skomplikowane
if (<wyrażenie1>) polecenie_true1;
else if (<wyrażenie2>) polecenie_true2;
else if (<wyrażenie3>) polecenie_true3;
else polecenie_domyślne;
```

7.5. Wyrażenia typu case

7.5.1. Wyrażenie case

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

Wyrażenie *wyrażenie* jest porównywane z poszczególnymi alternatywami *alternatywa_n*. Pierwsze dopasowanie spowoduje uruchomienie odpowiedniego polecenia. Wartości porównywane są bit po bicie, także wartości **x** i **z** są brane pod uwagę. Jeśli nie zgadza się ilość bitów, to krótsze wyrażenie uzupełniane jest zerami, aby długość porównywanych wyrażeń była identyczna. Dopuszcza się oddzielenie przecinkiem kilku alternatyw. Wielokrotne użycie części **default** nie jest dozwolone. Wyrażenia **case** mogą być zagnieżdżone.

Przykład wyrażenia **case**:

```

reg [1:0] alu__control;
...
...
case (alu_control)
    2'd0 : y = x + z;
    2'd1, 2'd2 : y = x - z;
    default : $display(„Invalid ALU control signal);
endcase

```

7.5.2. Wyrażenie casex i casez

Są 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.

7.6. Pętle

7.6.1. Pętla while

Pętla **while** uruchamia się dopóki *wyrażenie* jest prawdziwe. Jeśli *wyrażenie* jest fałszywe od początku, to pętla **while** nie uruchomi się w ogóle.

```

integer count;
while (count < 128)
begin
    $display(„count=%d”, count);
    count = count + 1;
end
...
...
while ((i<16) && continue)
begin
    ...
end

```

7.6.2. Pętla for

Pętla **for** zawiera trzy części:

- warunek początkowy;
- kontrolę zakończenia pętli;
- przypisanie zmieniające zmienną pętli.

```

integer count;

initial
    for (count=0; count<128; count=count+1)
        $display("Count=%d",count);

```

7.6.3. Pętla repeat

Pętla ta po prostu wykonuje polecenie określoną ilość razy. Pętla **repeat** musi zawierać liczbę powtórzeń, która może być stałą, zmienną lub sygnałem. Ilość powtórzeń jest obliczana tylko raz na początku uruchamiania polecenia **repeat**.

```
repeat(128)
begin
    $display("Count=%d", count);
    count=count+1;
end
```

7.6.4. 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 wykonywała by się w nieskończoność, wstrzymując pozostałe polecenia.

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

7.7. Bloki sekwencyjne i równoległe

7.7.1. Typy bloków

Istnieją dwa typy bloków: sekwencyjne i równoległe.

Bloki sekwencyjne

Do zgrupowania poleceń w blok sekwencyjny służą słowa kluczowe **begin** i **end**. Bloki sekwencyjne mają następujące cechy:

- Polecenia w bloku sekwencyjnym wykonywane są w kolejności, w jakiej są tam umieszczone. Następne polecenie jest wykonywane wtedy, gdy zakończy się poprzednie (z wyjątkiem przypisań *nonblocking* w wewnętrznym opóźnieniu).
- Jeśli określone jest opóźnienie lub kontrola wydarzeniem, jest ono określone względem czasu zakończenia poprzedniego polecenia w bloku.

Bloki równoległe

Do zgrupowania poleceń w blok równoległy służą słowa kluczowe **fork** i **join**. Bloki równoległe mają następujące cechy:

- Wyrażenia w bloku równoległym wykonywane są jednocześnie;
- Kolejność wykonywania poleceń jest określona poprzez opóźnienia lub wydarzenia związane z konkretnymi poleceniami.
- Opóźnienia liczone są względem czasu wejścia do bloku.

```
reg x, y;
reg [1:0] z, w;

initial
fork
    x=1'b0;          // wykonuje się w czasie=0
    #5 y=1'b1;       // wykonuje się w czasie=5
    #10 z={x, y};    // wykonuje się w czasie=10
    #20 w={y,x};     // wykonuje się w czasie=20
```

join

Należy uważać na zjawisko hazardu, mogące wystąpić w bloku równoległym.

7.7.2. Cechy specjalne bloków

Zagnieżdżanie

Bloki mogą być zagnieżdżane. Można mieszać bloki sekwencyjne i równoległe.

```
// zagnieżdżanie bloków
initial
begin
    x=1'b0;
    fork
        #5 y=1'b1;
        #10 z={x,y};
    join
    #20 w={y, x};
end
```

Bloki z nazwą

Bloki mogą mieć nadane nazwy.

- Można deklarować zmienne lokalne dla bloków z nazwą.
- Bloki z nazwą są częścią hierarchii projektu. Zmienne bloków mogą być dostępne przy wykorzystaniu pełnej nazwy hierarchicznej.
- Bloki z nazwą mogą być dezaktywowane, tj. ich wykonywanie może być zatrzymane.

```
//bloki z nazwą:
module top;

    initial
    begin: block1 // blok sekwencyjny nazwany block1
        integer i; //zmienna i jest zmienną statyczną i lokalną bloku
                    //dostęp do niej: top.block1.i
        ...
        ...
    end

    initial
    fork: block2 // blok równoległy nazwany block2
        reg i; // rejestr i jest statyczny i lokalny bloku
                // dostęp do niego: top.block2.i
        ...
        ...
    join
```

Dezaktywacja bloków z nazwą

Słowo **disable** powoduje zakończenie wykonywania poleceń bloku. W ten sposób można wychodzić z pętli, obsługiwać błędy, itp. Wyłączenie bloku powoduje przejęcie kontroli nad programem przez polecenia występujące bezpośrednio za blokiem. Wyłączenie bloku jest podobne do polecenia *break* w C. Różnicą jest, że *break* powoduje przerwanie aktualnej pętli, a polecenie **disable** może przerwać wykonywanie dowolnego bloku.

8. ZADANIA I FUNKCJE

8.1. Różnice pomiędzy zadaniami i funkcjami

Funkcje	Zadania
Funkcja może uruchamiać inną funkcję, nie może uruchamiać zadania.	Zadanie może uruchamiać inne zadania i funkcje
Czas wykonywania funkcji jest zawsze zerowy	Zadanie może wykonywać się przez czas niezerowy.
Funkcja nie może zawierać opóźnień, wydarzeń ani innych struktur kontroli czasowej	Zadanie może zawierać opóźnienia, wydarzenia i inne struktury kontroli czasowej
Funkcja musi posiadać przynajmniej jeden argument (może być więcej) typu input .	Zadanie może mieć zero lub więcej argumentów typu input , output lub inout .
Funkcja zawsze zwraca pojedynczą wartość.	Zadanie nie zwraca wartości, ale może zwracać wartości poprzez argumenty typu output lub inout .

Zadania i funkcje muszą być zdefiniowane w module i są lokalne dla danego modułu. Zadania i funkcje mogą zawierać lokalne zmienne, rejestry, zdarzenia itp. Nie mogą zawierać zmiennych typu **wire**. Zadania i funkcje zawierają tylko wyrażenia behawioralne. Nie zawierają bloków **initial/always**, ale często są wywoływane przez te bloki.

8.2. Zadania

Zadania deklaruje 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.

8.2.1. Deklaracja zadania i jego wywołanie

Składnia:

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

Deklaracje: **input**, **output** lub **inout**. Zmienne typu **input** i **inout** są przekazywane do zadania. Zmienne typu **output** i **inout** są zwracane z powrotem po zakończeniu zadania. Zadanie może wywoływać inne zadania lub funkcje. Zadanie może operować na zmiennej typu **reg** zdefiniowanej w module.

8.2.2. Przykład

```
// definicja modułu z zadaniem  
module operation;
```

```

...
...
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B) //gdy A lub b zmienia wartość
begin
    bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
...
...
//definicja zadania bitwise_oper;
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor; //zmienne wyjściowe
input [15:0] a, b; // zmienne wejściowe
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end
endtask
...
endmodule

```

8.3. Funkcje

Funkcje deklaruje 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.

8.3.1. Deklaracja funkcji i jej wywołanie (przykład)

Gdy zadeklarowano funkcję, automatycznie tworzony jest rejestr o takiej samej nazwie jak nazwa funkcji. Wartość funkcji jest zwracana w ten sposób, że ustawiany jest wspomniany rejestr. Funkcja nie może wywoływać innych zadań. Funkcja może wywoływać inne funkcje.

```

module shifter;
...
`define LEFT_SHIFT    1'b0;
`define RIGHT_SHIFT   1'b1;
reg [31:0] addr, left_addr, right_addr;
reg control;

// oblicz wartości przesunięcia w lewo i prawo, gdy
// pojawia się nowa wartość
always @(addr)
begin
    // wywołaj funkcję zdefiniowaną poniżej:
    left_addr = shift(addr, 'LEFT_SHIFT);
    right_addr = shift(addr, 'RIGHT_SHIFT);

```

```

end
...
...
// definicja f-cji shiftf (zwraca 32-bit. wartość)
function [31:0] shiftf;
input [31:0] address;
input control;
begin
    //ustaw odpowiednią wartość
    //w zależności od sygnału control
    shift = (control=='LEFT_SHIFT')?(address<<1):(address>>1);
end
endfunction
...
...
endmodule

```

9. TECHNIKI MODELOWANIA

9.1. Proceduralne przypisanie ciągle

Przypisanie proceduralne (rozdział 7.2) przypisuje wartość do rejestru. Rejestr zachowuje daną wartość, aż do następnego wpisu. Proceduralne przypisanie ciągle działa inaczej - zmienne lub wyrażenia są w sposób ciągły podawane do rejestru przez określony okres czasu.

9.1.1. Typ 1: assign i deassign

Słowa kluczowe **assign** i **deassign** służą do oznaczenia pierwszego rodzaju przypisania ciągłego. Lewa część przypisania może być tylko rejestrem lub połączeniem rejestrów. Nie może być częścią bitów z sieci ani też macierzą rejestrów. Przypisanie ciągle ma większy priorytet niż zwykle przypisania proceduralne. Poniższy przykład pokazuje realizację przerzutnika D z asynchronicznym *resetem* [1].

```
// Negative edge-triggered D-flipflop with asynchronous reset
module edge_dff(q, qbar, d, clk, reset);

// Inputs and outputs
output q,qbar;
input d, clk, reset;
reg q, qbar; //declare q and qbar are registers

always @(negedge clk) //assign value of q&qbar at active edge of clock.
begin
    q = d;
    qbar = ~d;
end

always @(reset) //Override the regular assignments to q and qbar
    //whenever reset goes high. Use of procedural continuous
    //assignments.
    if(reset)
    begin //if reset is high,override regular assignments to q with
        //the new values, using procedural continuous assignment.
        assign q = 1'b0;
        assign qbar = 1'b1;
    end
    else
    begin //If reset goes low, remove the overriding values by
        //deassigning the registers. After this the regular
        //assignments q = d and qbar = ~d will be able to change
        //the registers on the next negative edge of clock.
        deassign q;
        deassign qbar;
    end
end

endmodule
```

9.1.2. Typ 2: force i release

Force i **release** pozwalają zapisywać wartości zarówno do rejestrów jak i do sieci. Polecenia te wykorzystywane są głównie przy *debuggingu*.

Rejestry:

Takie przypisanie do rejestru ma większy priorytet niż przypisania proceduralne czy ciągłe, dopóki nie zostanie wydane polecenie **release**. Po **release** wartość będzie pamiętana, ale będzie już można ją zmienić [1].

```
module stimulus;
...
...
//instantiate the d-flipflop
edge_dff dff(Q, Qbar, D, CLK, RESET);
...
...
initial
begin
    //these statements force value of 1 on dff.q between time 50 and
    //100, regardless of the actual output of the edge_dff.
    #50 force dff.q = 1'b1; //force value of q to 1 at time 50.
    #50 release dff.q; //release the value of q at time 100.
end
...
...
endmodule
```

Sieci:

Przypisanie **force** do sieci nadpisuje jakiegokolwiek inne wartości. Po **release** następuje powrót do normalnej wartości wymuszanej na sieci [1].

```
module top;
...
...
assign out = a & b & c; //continuous assignment on net out
...
initial
    #50 force out = a | b & c;
    #50 release out;
end
...
...
endmodule
```

W czasie od 50 do 100 wyrażenie $a | b \& c$ jest od nowa obliczane i podstawiane do out, gdy któraś ze zmiennych a, b lub c zmieni swą wartość.

9.2. Definiowanie parametrów

Parametry umożliwiają zmianę pewnych wartości w modułach już podczas ich powoływania do życia. Są dwie możliwości podawania parametrów: poprzez *defparam* lub *module instance parameter value*.

9.2.1. Polecenie defparam

Polecenie **defparam** umożliwia zmianę parametrów w dowolnym module [1]:

```
//Define a module hello_world
module hello_world;
parameter id_num = 0; //define a module identification number = 0

initial //display the module identification number
    $display("Displaying hello_world id number = %d", id_num);
endmodule

//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;

//instantiate two hello_world modules
hello_world w1();
hello_world w2();

endmodule
```

W module może być wiele poleceń **defparam**.

9.2.2. Zmiana parametrów podczas powoływania do życia

Poprzedni przykład został trochę zmieniony, nowe parametry są podawane podczas powoływania modułu do życia. Nie potrzebne są już polecenia **defparam** [1].

```
//define top-level module
module top;

//instantiate two hello_world modules; pass new parameter values
hello_world #(1) w1; //pass value 1 to module w1
hello_world #(2) w2; //pass value 2 to module w2

endmodule
```

W przypadku kilku parametrów, należy je wymienić po przecinku, w takiej kolejności, w jakiej zostały one zdefiniowane w definicji modułu.

```
hello_word #(1,4,5) w1();
```

9.3. Warunkowa kompilacja i uruchamianie

9.3.1. Warunkowa kompilacja

Kompilacja warunkowa odbywa się po podaniu słów kluczowych **`ifdef**, **`else** oraz **`endif**. Warunki sprawdzane przez **`ifdef** definiuje się za pomocą polecenia **`define**.

9.3.2. Warunkowe uruchamianie

Kompilowany jest cały program, ale uruchamiane tylko odpowiednie części. Za pomocą słowa kluczowego **\$test\$plusargs** sprawdzane jest, czy zdefiniowana została zmienna:

```
module test;
...
```

```

...
initial
begin
if($test$plusargs("DISPLAY_VAR"))
    $display("Display=%b", {a,b,c} ); // wyświetl tylko,
                                     // gdy flaga
                                     // jest ustawiona
else
    $display("No display");
end
endmodule

```

Zmienną *DISPLAY_VAR* definiuje się podczas uruchamiania za pomocą polecenia: *+DISPLAY_VAR*. Warunkowe uruchamianie nie jest częścią standardu IEEE i może nie działać na niektórych symulatorach.

9.4. Skala czasu

Często zachodzi potrzeba definiowania czasu w jednym module w ns a w innym w μ s. Verilog umożliwia definicję jednostki czasu dla modułu.

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

<jednostka_czasu> określa jednostkę czasu dla czasu symulacji oraz opóźnień. <dokładność> określa z jaką dokładnością zaokrąglane są opóźnienia podczas symulacji. Do określenia obydwu tych wartości można wykorzystać tylko trzy wartości: 1, 10 i 100. Przykłady:

```
'timescale 100 ns / 1 ns
```

```
'timescale 1 us / 10ns
```

9.5. Zadania systemowe

9.5.1. Pliki

Otwieranie plików

Wyjście z Verilogu idzie normalnie na *standard output* oraz do pliku *verilog.log*. Jest możliwość przekierowania wyjścia do innego pliku:

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

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

Zadanie **\$fopen** zwraca 32-bitową wartość zwaną *deskryptorem wielokanałowym*. W deskrytorze tylko jeden bit jest ustawiony. *Standard output* posiada deskryptor z ustawionym pierwszym bitem (tzw. kanał 0), który jest zawsze otwarty. 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.

```
integer handle1, handle2, handle3;
```

```

initial
begin
    handle1=$fopen("file1.out"); //handle1=32'h0000_0002
    handle2=$fopen("file2.out"); //handle1=32'h0000_0004
    handle3=$fopen("file3.out"); //handle1=32'h0000_0008
end

```

Pisanie do pliku

Do pisania do pliku służą następujące zadania: **\$fdisplay**, **\$fmonitor**, **\$fwrite** oraz **\$fstrobe**. Działają one podobnie jak **\$display** i **\$monitor**. Rozważone zostaną tylko **\$fdisplay** i **\$fmonitor**.

```

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

```

gdzie $p1, p2, \dots, pn$ oznaczają zmienne, sygnały czy łańcuchy. Deskryptor może być kombinacją kilku deskryptorów wielokanałowych - Verilog będzie pisał do wszystkich, które mają ustawioną 1 na odpowiednim miejscu.

Zamykanie pliku

```

$fclose(<handle>);

```

9.5.2. Wyświetlanie hierarchii

Hierarchia na dowolnym poziomie może być wydrukowana przy użyciu %m.

9.5.3. Strobe

Polecenie **\$strobe** jest podobne do **\$display**, ale wykonuje się ono zawsze pod koniec danego kroku czasu - jest wtedy pewność, że wykonały się już wszystkie przypisania.

9.5.4. Generator pseudolosowy

```

$random;
$random(<seed>);

```

Podanie wartości <seed> pozwala na uzyskanie takiego samego zestawu wartości za każdym uruchomieniem symulacji.

\$random zwraca 32-bitową liczbę.

9.5.5. Inicjalizacja pamięci z pliku

Deklaracja pamięci została omówiona w punkcie 3.2.7. Do inicjalizacji pamięci służą dwa polecenia: **\$readmemb** i **\$readmemh**.

```

$readmemb("<nazwa_pliku>", <nazwa_pamięci>);
$readmemb("<nazwa_pliku>", <nazwa_pamięci>, <adres_pocz>);
$readmemb("<nazwa_pliku>", <nazwa_pamięci>, <adres_pocz>,
<adr_końc>);

```

Analogicznie wygląda wykorzystanie polecenia **\$readmemh**.

Domyślnie <adres_pocz> jest początkowym indeksem pamięci, a <adr_końc> jest końcem pliku lub pamięci.

```
module test;
reg [7:0] memory[0:7]; //deklaracja pamięci 8-bajtowej
integer i;

initial
begin
    // czytaj z pamięci:
    $readmemb("init.dat", memory);
    // wyświetl:
    for(i=0; i<8; i=i+1);
        $display("Pamięć [%d] = %b, i, memory[i]);
end

endmodule
```

W pliku adres oznacza się poprzez @<adres>. Dane oddziela się białą spacją. Dane mogą zawierać wartości x lub z. Niezainicjowane wartości będą miały wartość x.

```
@002
11111111 01010101
00000000 10101010
```

```
@006
1111zzzz 00001111
```

Efekt działania programu:

```
Pamięć [0] = xxxxxxxx
Pamięć [1] = xxxxxxxx
Pamięć [2] = 11111111
Pamięć [3] = 01010101
Pamięć [4] = 00000000
Pamięć [5] = 10101010
Pamięć [6] = 1111zzzz
Pamięć [7] = 000011
```

10. Literatura

- [1] S. Palnitkar, VERILOG HDL a Guide to Digital Design and Synthesis, SunSoft Press, 1996.
- [2] K. Coffman, Real Word FPGA Design with Verilog, PrenticeBHall, 2000.
- [3] M. G. Arnold, Verilog Digital Computer Design, Algorithms into Hardware, PrenticeBHall, 1999.