- 4 -
Засоби VHDL для моделювання реальних об’єктів

```
entity RS_Trigger is
   port(S, R:        in IEEE.Std_logic_1164.Std_Logic;
            D, Inv_D:        out IEEE.Std_logic_1164.Std_Logic;
end entity RS_Trigger;
```

є адекватним запису

```
library IEEE;
use IEEE.Std_logic_1164.Std_Logic;
entity RS_Trigger is
   port(S, R:        in Std_Logic;
            D, Inv_D:        out Std_Logic;
end entity RS_Trigger;
```

Якщо є необхідність у застосуванні всіх програмних елементів, що входять до складу пакету, то замість *ідентифікатора програмного елементу* слід поставити ключове слово **all**. Так, при автоматизованому створенні файла VHDL-коду САПР Active-HDL додає до програми наступні рядки:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
```

що дозволяють застосовувати всі типи та підпрограми пакету стандартної логіки.

4.1. Опис об’єктів у VHDL (Entity)

VHDL є мовою опису реальних об’єктів та процесів, що протікають в цих об’єктах. Для такого типу описів VHDL містить ряд спеціалізованих конструкцій.

Опис об’єкта в VHDL починається з визначення зв’язків між об’єктом та зовнішнім середовищем. Повний перелік таких зв’язків називається *інтерфейсом*.

Синтаксис опису інтерфейсу об’єктів:

entity *identifier* **is**
port (*port_interface_list*);
entity_declarative_item1;
entity_declarative_item2;
…
entity_declarative_item;
end entity *identifier*;

де *identifier* – ім’я об’єкта; *port_interface_list* – інтерфейс об’єкта; *entity_declarative_items* – необов’язковий список об’явлень, що може містити описання констант, типів, сигналів (більш детально буде розглянуто нижче); *port_interface_list* містить повний перелік всіх портів (вхідних та вихідних сигналів) об’єкта з означенням їх типу та напряму (вхідний, вихідний, двонаправлений).

Синтаксис опису портів:

(*identifier1*: **mode** *type* **: =** expression1;
identifier2: **mode** *type* **: =** expression2;
…);

де *identifier1, identifier2* – імена портів; *mode* – параметр, що вказує на напрям порту і може приймати одне з трьох значень: **in** – вхідний, **out** – вихідний та **inout** – двонаправлений; *type* – тип порту; *expression* – початкове значення порту.

Для прикладу розглянемо опис інтерфейсу чотирирозрядного суматора з двома вхідними та одним вихідним портами:

entity *adder* **is**
  port(*a*: **in** *Std_Logic_Vector* (3 **downto** 0);
        b: **in** *Std_Logic_Vector* (3 **downto** 0);
        c: **out** *Std_Logic_Vector* (3 **downto** 0) := “0000”);
end entity *adder*;

В одному рядку можна об'явити кілька однакових за типом та напрямом портів. Так, попередній приклад можна переписати у вигляді

entity *adder* **is**
  port(*a, b*: **in** *Std_Logic_Vector* (3 **downto** 0);
        c: **out** *Std_Logic_Vector* (3 **downto** 0) := "0000");
end entity *adder*;

4.2. Архітектура

Після опису інтерфейсу об'єкта необхідно описати його архітектуру. *Архітектурою* називається внутрішня структура об'єкта, що визначає його поведінку. Саме архітектура визначає, яким чином вхідні порти об'єкта поєднуються з вихідними. Синтаксис архітектури:

architecture identifier **of** entity_name **is**
Block_Declarative_Item;
begin
  Concurrent_Statements
end architecture identifier;

де *identifier* – ім'я архітектури; *entity_name* – ім'я об'єкта, поведінку якого описує дана архітектура; *Block_Declarative_Item* – список елементів програми, що будуть доступні в межах архітектури; в цьому розділі можна оголошувати константи, типи, компоненти та сигнали.

Поведінка об'єкта описується всередині архітектури за допомогою конкурентних операторів (*Concurrent_Statements*). В послідовних мовах програмування, таких як Pascal та C++, поняття конкурентних операторів відсутнє. Ці оператори призначаються для реалізації паралельних процесів і виконуються одночасно. До конкурентних операторів належать процеси (**process**), компоненти та деякі інші.

4.3. Сигнали

Сигнали використовуються для зв'язків між різними об'єктами (**entity**) та процесами, що протікають в цих об'єктах. Фізично сигнал можна представити у вигляді лінії зв'язку, яка поєднує об'єкти або

структурні складові всередині об'єктів. Крім того, така лінія зв'язку може нескінченно довго зберігати подане на неї значення сигналу. В реальних об'єктах будь-які процеси не можуть перебігати миттєво (стрибкоподібно), тому важливою особливістю *сигналу*, як моделі реального процесу передачі інформації, є те, що сигнал не може змінювати свого значення миттєво. Тобто між командою на зміну значення сигналу та реальною зміною значення завжди пролягає певний проміжок модельного часу.

Перед застосуванням сигнал необхідно попередньо об'явити. Синтаксис об'явлення сигналу має вигляд

signal *Ідентифікатор*:

Тип_Сигналу := *Початкове_Значення*;

де *Ідентифікатор* – ім'я сигналу, яке сформоване за правилами формування ідентифікаторів; *Тип_Сигналу* – може задаватися будь-яким типом чи підтипом; *Початкове_Значення* – значення сигналу в момент часу 0 с. Якщо початкове значення не задається явно, то в такому випадку сигнал набуває крайнього лівого значення підтипу *Тип_Сигналу*.

Кілька однотипних сигналів з однаковим початковим значенням можуть об'являтися одним оператором **signal**. У цьому випадку їх ідентифікатори перераховуються через кому, наприклад:

signal *S1, S2, S3*: *Std_Logic* := '1';

Оператори об'явлення сигналів можуть розміщуватися в розділі оголошень архітектури: тоді кожний з об'явлених сигналів буде доступний всім процесам поточної архітектури. Крім того, сигнал можна об'явити в розділі оголошень **entity**: в цьому випадку сигнал буде доступний всім введеним архітектурам даного об'єкта.

4.4. Оператор направлення сигналу та способи реалізації затримки в VHDL

4.4.1. Оператор направлення сигналу

Для зміни значення сигналу використовується оператор

направлення, його синтаксис має вигляд

Target_Signal <= *Source* **after** *delay*;

де *Target_Signal* – ідентифікатор сигналу, значення якого має змінитися; *Source* – джерело нового значення (може застосовуватися змінна, константа, вираз або сигнал); *delay* – часова затримка.

Мова VHDL є спеціалізованою мовою для описання та моделювання пристроїв, реальні сигнали в яких не можуть змінювати свого значення миттєво. Для врахування цієї особливості в VHDL застосовується механізм транзакцій.

Транзакцією називається відкладена дія призначення сигналу. Кожна транзакція містить ім'я сигналу, що буде змінюватись, нове значення сигналу та значення модельного часу, в якому буде проведено зміну значення сигналу.

Отже, якщо при моделюванні об'єкта зустрічається оператор направлення сигналу, то його значення не змінюється. Замість цього формується транзакція, в яку заноситься ім'я сигналу (*Target_Signal*), нове значення (*Source*) та значення модельного часу зміни сигналу, що формується як сума поточного значення модельного часу та часової затримки *delay* (у випадку, коли *delay* не задано явно, час затримки визначається кроком модельного часу). Сформована нова транзакція додається до списку транзакцій, який перевіряється системою на кожному кроці зростання модельного часу. Якщо ж настає момент зміни сигналу, то йому присвоюється нове значення. Зміна значення сигналу називається *подією* на сигналі. Такий алгоритм призводить до того, що сигнал не змінює свого значення миттєво, і мінімальна часова затримка дорівнює кроку модельного часу.

Для більш точного опису різноманітних умов передачі сигналів в реальних об'єктах у VHDL введено два типи часової затримки: транспортна та інерційна.

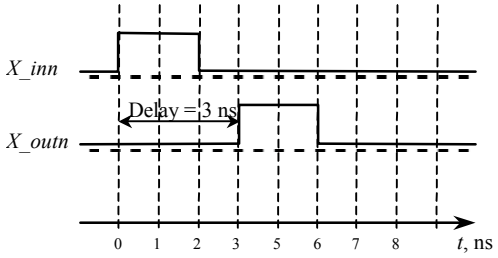
4.4.2. Транспортна затримка сигналу

Транспортна затримка сигналу (ТЗ) в загальному випадку моделює просту затримку в передачі вхідного сигналу на чітко визначений час. У загальному вигляді транспортна затримка може бути описана в VHDL як

X_out <= **transport** *X_in* **after** time_value;

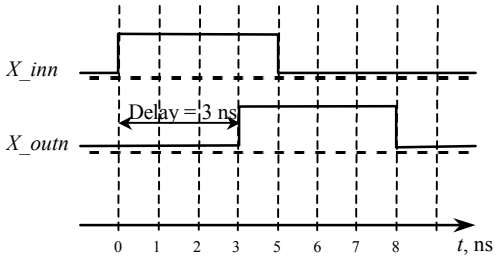
Приклад, наведений на рис. 4.1, ілюструє вплив транспортної затримки на вихідний сигнал при різній тривалості вхідного сигналу.

X_out<= **transport** *X_in* **after** 3 ns;



а

X_out<= **transport** *X_in* **after** 3 ns;



б

Рис. 4.1. Моделювання транспортної затримки сигналу: **а –** тривалість вхідного сигналу *X_in* менше значення затримки 3 ns; **б –** тривалість вхідного сигналу *X_in* більше або дорівнює значенню затримки 3 ns

4.4.3. Інерційна затримка сигналу

Інерційна затримка дозволяє моделювати роботу елементів схеми, які не пропускають короткі імпульси. Тобто, якщо будь-який з елементів схеми має інерційну затримку *T* (як, наприклад, логічні елементи **ТАК**, **АБО, НІ**), то вхідний сигнал буде затримуватися на час *T*, однак імпульси, тривалість яких менше *T*, взагалі передаватися

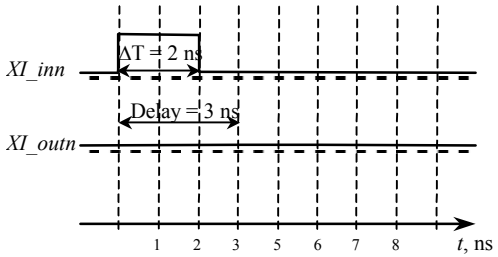
не будуть.

Загальна форма запису сигналу з інерційною затримкою має вигляд

XI_out <= **inertial** *XI_in* **after** *time_value*;

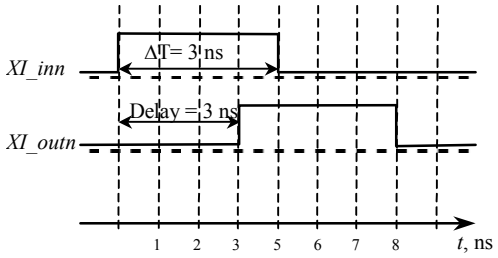
Наступний приклад, наведений на рис. 4.2, демонструє роботу пристрою з інерційною затримкою при різній тривалості вхідного сигналу.

XI_out <= **inertial** *XI_in* **after** 3 ns;



а

XI_out <= **inertial** *XI_in* **after** 3 ns;



б

Рис. 4.2. Моделювання інерційної затримки сигналу: **а** – тривалість вхідного сигналу менше значення затримки; **б** – тривалість вхідного сигналу більше або дорівнює значенню затримки

Зазначимо, що часто в реальних електронних пристроях мінімальна тривалість імпульсів, що повинні пропускатися, менша за значення інерційної затримки. У цих випадках форма запису даного

Таблиця 4.1

Ідентифікатор	Функція
S'delayed(*T*)	Приймає таке ж значення, як і *S*, але з затримкою на час *T*. Може застосовуватися для визначення попередніх значень сигналів
S'stable(*T*)	Повертає булеве значення, що дорівнює *True,* якщо на сигналі *S* не відбувалося *подій* (змін значення) впродовж проміжку часу *T* до поточного значення модельного часу
S'quiet(*T*)	Повертає булеве значення, що дорівнює *True,* якщо на сигналі *S* не виникало *транзакцій* за проміжок часу *T* перед поточним значенням модельного часу
S'transaction	Сигнал типу *bit*, що змінює своє значення з '0' на '1' (і навпаки) кожного разу, коли на сигналі *S* з'являється транзакція
S'event	Булевий сигнал, що дорівнює *True,* якщо на сигналі *S* в поточній ітерації моделювання відбувалася подія, і дорівнює *False* в іншому випадку
S'active	Сигнал дорівнює *True,* якщо на поточній ітерації моделювання спрацювала транзакція для сигналу *S*
S'last_event	Повертає значення типу *time* – проміжок часу до останньої події на сигналі *S*
S'last_active	Повертає значення типу *time* – проміжок часу до останньої транзакції на сигналі *S*
S'last_value	Повертає останнє перед поточним значення сигналу *S*

типу затримки зміниться таким чином:

X_out <= **reject** *impuls_length*
inertial *X_val* **after** *time_value*;

Приклад:

XI_out <= **reject** 1 ns **inertial** *XI_in* **after** 3 ns;

Якщо тип затримки не вказується явно, то автоматично "по умовчанню" вибирається інерційний тип затримки сигналу.

4.4.4. Аналогії та відмінності між сигналами та змінними

Порівнюючи сигнали із змінними, можна помітити, що головна відмінність між ними – це час зміни значення. На відміну від сигналів, змінна набуває нового значення миттєво. З *прикладу* добре видно різницю, між результатами виконання одних і тих же операцій над змінними (а) і над сигналами (б):

а) операції над змінними:

```
architecture var of EntA is
 signal InS, Sum: integer :=0;
begin
 process (InS)
 variable var1, var3: integer :=0;
 variable var2 : integer :=1;
 begin
 var1:=var2+1;
 var2:=var1-2;
 var3:=var1+var2;
 Sum<=var1+var2+var3;
 end process;
end var;
```

Результат виконання процесу:
Sum = **4**.

б) операції над сигналами:

```
architecture var of EntA is
 signal InS, Sum: integer :=0;
 signal var1, var3: integer :=0;
 signal var2 : integer :=1;
begin
 process (InS)
 begin
 var1 <= var2+1;
 var2 <= var1-2;
 var3 <= var1+var2;
 Sum <= var1+var2+var3;                    --(*)
```

```
 end process;
end var;
```

Результат виконання процесу:
Sum = **1**.

Відмінності в результатах пояснюються тим, що сигнали *var1, var2, var3* в прикладі б) не змінюють свого значення на тому ж циклі моделювання, на якому виконується процес. Таким чином, при виконанні рядка програми, позначеного знаком (*), перелічені сигнали ще мають старі значення, і саме на їх основі обчислюється результат виразу правої частини оператора призначення сигналу.

4.5. Атрибути сигналів

Сигнали в VHDL – це гнучкий і потужний засіб для моделювання реальних каналів обміну інформацією як всередині електронної системи, так і між електронною системою та зовнішнім середовищем. За допомогою *атрибутів* можна будувати програми, що реагують не лише на поточні значення сигналів, а й на “історію” їх подій і транзакцій.

Нехай визначено сигнал *S* і вираз *T* типу *time*. В табл. (4.1) наведено повний перелік атрибутів сигналів з описом їх функцій.

Слід звернути увагу на відмінності між застосуванням атрибутів *‘Event* та *‘Active*. Перший з них приймає значення *True* в тих циклах моделювання, де на відповідному сигналі відбулася подія (змінилося значення сигналу), а другий – в тих циклах, коли відбулася транзакція.

На рис. 4.3 проілюстровано цю відмінність.

```
process is
begin
   S<='0';
   wait for 10 ns;
   S<='1';
   wait for 30 ns;
   S<='1';
   wait;
```

end process;

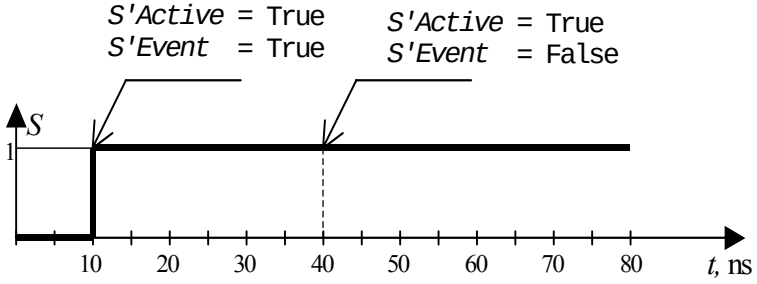


Рис. 4.3. Результат виконання процесу

Тут у момент часу t = 10 ns значення сигналу S змінилось з '0' на '1'. Відповідно на цьому циклі моделювання спрацювала транзакція (*S'Active* = True) і відбулася подія (*S'Event* = True). В момент часу t = 40 ns транзакція спрацювала (*S'Active* = True). Разом з тим, оскільки в момент часу t = 40 ns фактичної зміни значення сигналу *S* не відбулося (значення змінилося з '1' на '1'), то події на сигналі не було і, відповідно, *S'Event* = False.

Незважаючи на те, що при апаратній реалізації VHDL-проектів застосування більшості атрибутів сигналів є ускладненим (так, наприклад, було б дуже важко визначити останнє попереднє значення напруги на провіднику чи час її зміни), використання інформації атрибутів є дуже корисним на етапі комп'ютерного моделювання VHDL-проекту, що розробляється або вже є розробленим. За допомогою механізму атрибутів можна більш детально аналізувати внутрішні процеси, що протікають в електронних пристроях, і, відповідно, виявляти значну кількість проектних помилок та відхилень.

Область застосування атрибутів не обмежується лише задачами моделювання та тестування VHDL-проектів. Атрибути *'Event* та *'Active* підтримуються засобами синтезу ПЛІС, однак в цьому випадку їх можна використовувати лише в операторах **wait** та **if** для визначення подій на сигналах тактової частоти з метою синхронізації окремих частин проекту.

4.6. Процес

Одним з ключових елементів мови VHDL є процес (**process**). Процес є конкурентним оператором, тому поміщається в розділ *Concurrent_Statements* архітектури.

Синтаксис процесу має вигляд

Мітка_Процесу: **process** (*Список_Чутливості*) **is**
Розділ_Оголошень;
begin
Послідовні_Оператори;
end process *Мітка_Процесу*;

У наведеному прикладі позначено:

- *Мітка_Процесу* – сформований за правилами ідентифікатор процесу; *Розділ_Оголошень* – розділ, в якому можна об'являти локальні (доступні лише в поточному процесі) константи, змінні, підтипи, а також підпрограми (процедури та функції);
- *Список_Чутливості* – список сигналів, при виникненні подій на яких (зміна сигналу) здійсниться запуск процесу. Список чутливості може бути відсутнім, але в цьому випадку обов'язкова присутність у процесі хоча б одного оператора **wait** (розглядається нижче);
- *Послідовні_Оператори* – оператори, що виконуються (на відміну від конкурентних) послідовно один за одним. До послідовних операторів відносяться *умовний оператор, оператор вибору, циклічні оператори, оператор присвоювання, оператор пересилання значення сигналу* та інші. Для аналогії зазначимо, що всі оператори таких мов, як Pascal та C++ є послідовними.

Розглянемо роботу процесів на прикладі RS-тригера:

entity *RSTrigger* **is**
  port(S, *R*: **in** *Std_Logic*;
    D, *Inv_D*: **out** *Std_Logic;*
end entity *RSTrigger*;

architecture *RSTrigger* **of** *RSTrigger* **is**

```
begin
Flip_Flop:      process (R,S) is
        begin
                if S = '1' and R = '0' then
                        D<='1';
                        Inv_D<='0';
                elseif S = '0' and R = '1' then
                        D<='0';
                        Inv_D<='1';
                else
                        D<='Z';
                        Inv_D<='Z';
                end if;
        end process Flip_Flop;
end architecture RSTrigger;
```

Робота цього процесу протікає таким чином: процес очікує появи події (зміни сигналу) на сигналах *R* чи *S*. Після цього перевіряються значення вхідних сигналів: якщо *R* ='0', *S* = '1', то на вихід тригера *D* подається сигнал логічної '1', а на інверсний вихід – сигнал логічного '0', якщо ж *R* ='1', *S* = '0', то на вихідний та на інверсний вихідний сигнали відповідно подаються логічні '0' та '1'. У всіх інших випадках вихідний сигнал RS-тригера є невизначеним, тому на його виходи подається сигнал 'Z' (високий імпеданс).

4.7. Оператор Wait

Аналіз основних класів електронних пристроїв показує, що вони, як правило, мають безперервний цикл роботи. Тобто після ініціалізації вони виконують визначені для них операції, а потім переходять у режим очікування відповідної *події* нового запуску. Іншими словами, пристрої припиняють свою роботу після завершення поточних задач і поновлюють її знову, як тільки виникає необхідна *подія*. Дана властивість систем описується в VHDL за допомогою службового слова **wait,** яке визначає не тільки момент розриву послідовності операцій, але і необхідні умови їх поновлення. Аналогічні функції виконує *список чутливості* процесу, тому один і той же процес не

може містити одночасно *список чутливості* і оператор **wait.** При цьому процес, у який входить вираз, що містить **wait,** має вигляд

process is
begin
sequential_statement ;
wait_statement;
sequential_statement;
wait_statement;
...
end process;

Даний процес виконує послідовність дій *sequential_statement* доти, доки не зустріне оператор **wait** (*wait_statement*), який припиняє виконання процесу до виникнення події, що зазначена в параметрах оператора **wait**.

Існує три основні і один додатковий тип оператора **wait**:

Основний оператор 1.

wait on *sensitivity_list* – очікування здійснюється доти, доки не зміниться який-небудь з сигналів у списку чутливості (*sensitivity_list*).

Приклад:

wait on CLK;
wait on Ain, Bout;

Основний оператор 2.

wait for *time_expression* – очікування здійснюється, поки не мине час, що вказаний в *time_expression*.

Приклад:

wait for 5 ns;
wait for *CLKperiod*/2;

Основний оператор 3.

wait until *boolean_expression* – очікування виконання умови, що вказана в *boolean_expression*.

Приклад:

wait until *CLK*=‘1’;
wait until *IntData*>15;

Додатковий тип можна визначити як *комбінований тип* запису припинення процесу, який може складатися із двох або трьох різних форм оператора **wait**.

Приклад:

wait on *A* **until** *CLK*=‘1’;

wait until *CLK*=‘1’ **for** 10 ns;

Слід зазначити, що, окрім наведених вище типів запису оператора **wait**, оператор **wait** може використовуватись і без параметрів, наприклад:

```
process is
begin
        list_of_statements;
        wait;
end process;
```

В останньому випадку процес почне свою роботу відразу після початку моделювання і зупинитися, коли будуть виконані всі операції з *list_of_statements*. Такий оператор **wait** без параметрів корисний при створенні випробувальних стендів і використовується для формування задаючих або збурюючих впливів.

4.8. Моделювання у VHDL

Розглянемо детально процедуру моделювання об’єктів у VHDL. Моделювання поведінки об’єктів в ЕОМ передбачає розрахунок значень сигналів об’єкта (вхідних, вихідних та внутрішніх) через рівні (дискретні) проміжки часу. Такі проміжки часу називаються кроком або циклом моделювання. Зрозуміло, що адекватність результатів моделювання реальним процесам обернено-пропорційна величині циклу моделювання.

Нижче наведено основні етапи процедури моделювання об’єкта в VHDL (рис. 4.4):

1. Ініціалізація моделювання:

- поточне значення *модельного часу* встановлюється в 0 (нуль);
- всім сигналам об'єкта присвоюються початкові значення.

2. Запуск на виконання всіх процесів.

Процеси виконуються до їх повної зупинки, тобто для процесів із списком чутливості до кінця (**end process**), або до оператора **wait** в іншому випадку. Лише після зупинки всіх процесів система може перейти до наступного етапу.

3. *Модельний час* збільшується на *Цикл (крок) моделювання*.
4. Перевіряється список транзакцій: якщо є транзакції, що настроєні на поточний *Модельний час,* то відповідні сигнали набувають нових попередньо запрограмованих значень.
5. Перевіряються всі оператори **wait**, що задіяні в даний момент *Модельного часу*: якщо є оператори, для яких виконуються умови запуску, то запускаються відповідні процеси.
6. Якщо *Модельний час* не досягає значення *Кінець_моделювання,* то система повертається до п. 3.

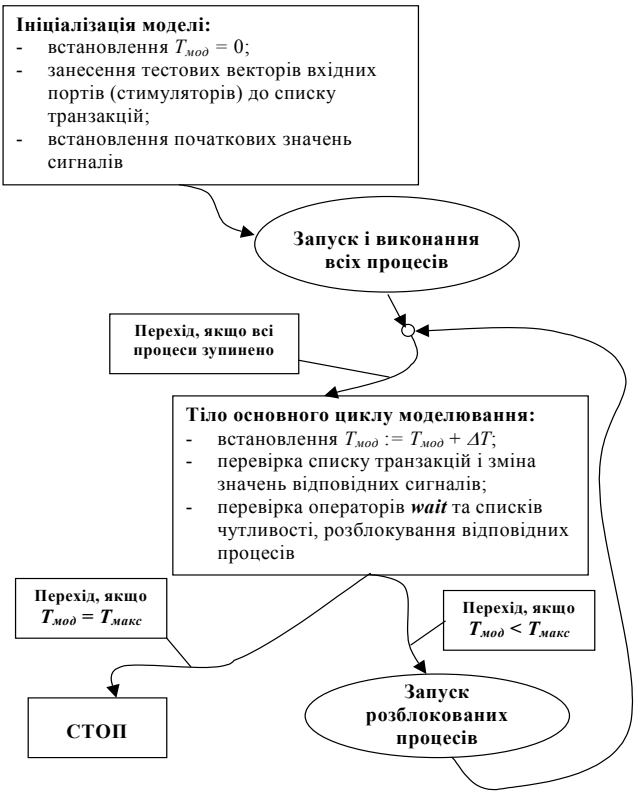


Рис. 4.4. Процес імітаційного моделювання об’єктів у VHDL

4.9. Компонування складних об’єктів із структурних складових

При описанні складних об’єктів часто виникає необхідність в їх декомпозиції (розбитті на структурні складові). Крім того, буває корисною можливість повторного застосування типових елементів об’єкта (наприклад, таких як лічильники, суматори, мультиплексори тощо). Мова VHDL надає програмісту можливості ефективно розв’язувати вказані задачі за допомогою механізму компонентів.

Один з можливих шляхів компонування складного об'єкта з уже розроблених елементів – застосування *оператора включення*. Цей оператор є конкурентним, отже розміщується в тілі архітектури (після слова **begin**).

Синтаксис *оператора включення* має вигляд

Instance_Name: **entity** *work. Entity_Name*
port map(*S1=>A, S2=>B, S3=>C*);

де *Instance_Name* – ім'я включення; *work* – ім'я бібліотеки (ім'я *work* вказує на поточний проект); *Entity_Name* – ім'я об'єкта, що включається.

Після ключових слів **port map** в дужках необхідно вказати, які саме сигнали поточного об'єкта підключаються до входів та виходів об'єкта, що включається (в розглянутому прикладі *A,B,C* – сигнали складного об'єкта; *S1, S2, S3* – сигнали компонента).

Розглянемо приклад.

Необхідно спроектувати арифметико-логічний пристрій (АЛП), при цьому пристрої, що реалізують арифметичні операції, були розроблені попередньо. Опис інтерфейсів цих пристроїв наведено нижче:

а) для суматора

```
entity adder is                    -- Суматор
   port(a, b: in Std_Logic_Vector (7 downto 0);
        c: out Std_Logic_Vector
                    (7 downto 0) := "00000000");
end entity adder;
```

б) помножувача

```
entity mul is                      -- Помножувач
   port(a, b: in Std_Logic_Vector (7 downto 0);
        c: out Std_Logic_Vector
        (7 downto 0) := "00000000");
end entity mul;
```

в) поділювача

```
entity divider is                  -- Поділювач
   port(a, b: in Std_Logic_Vector (7 downto 0);
        c: out Std_Logic_Vector
                    (7 downto 0) := "00000000");
end entity divider;
```

VHDL-програма, що реалізує арифметико-логічний пристрій на

основі декомпозиційного підходу, наведена нижче:

```
entity ALU is
   port(X, Y: in Std_Logic_Vector (7 downto 0);
            operation: in Std_Logic_Vector (1 downto 0);
            en: in Std_Logic;
            Z: out Std_Logic_Vector
                                    (7 downto 0) := "00000000");
end entity ALU;

architecture ALU of ALU is
 signal S1, S2, S3: Std_Logic_Vector (7 downto 0);
 begin
   Op1:   entity work.Adder
          port map(a=>X, b=>Y, c=>S1);
   Op2:   entity work.Mul
          port map(a=>X, b=>Y, c=>S2);
   Op3:   entity work.Divider
          port map(a=>X, b=>Y, c=>S3);

process (En) is
begin
   case Op is
            when "00" => Z <= S1;
            when "01" => Z <= S2;
            when "10" => Z <= S3;
            when others => Z <= "ZZZZZZZZ";
   end case;
end process;

 end architecture ALU;
```

На момент компіляції наведеної програми необхідно, щоб програми, які описують окремі елементи, були попередньо скомпільовані в окремих файлах і, якщо це необхідно, були підключені відповідні бібліотеки.

4.10. Константи generic

При компонуванні складних об'єктів часто виникає необхідність у

зміні параметрів компонентів. Можливість такої зміни без перекомпілювання компоненти дає застосування констант **generic**. Ці константи розміщуються в розділі опису об'єкта **entity.** Синтаксис констант **generic** наведено нижче:

```
entity Entity_Name is
   generic (a,b,c: integer range 0 to 255;
                                                d: Std_Logic);
   port(port_list);
end Entity_Name;
```

Як видно з наведеного прикладу, константам, описаним у розділі **generic**, не присвоюється конкретне значення на етапі компіляції об'єкта, однак вирази, що містять ці константи, є локально-статично визначеними. Це дає можливість застосовувати константи **generic** для задавання діапазонів індексів масивів, підтипів тощо. Конкретне значення цим константам присвоюється при включенні об'єкта як компонента складного об'єкта, *наприклад*:

```
entity Entity_Name is
   generic(a,b,c: integer range 0 to 255;
                                                d: Std_Logic);
   port    (port_list_1);
end Entity_Name;

entity Main_Entity is
   port(port_list_2);
end Main_Entity;

architecture Main_Entity of Main_Entity is
begin
   INST0: entity work.Entity_Name
            generic map(a=> 10, b=> 125, c=> 0, d => '1');
            port map(port_association_list);
   INST1: entity work.Entity_Name
            generic map(a=> 2, b=> 3, c=> 4, d => '0');
            port map(port_association_list);
   …

end Main_Entity;
```

Розглянемо застосування **generic** на прикладі реалізації універсального паралельного регістра.

VHDL-програма, що реалізує 8-бітний паралельний регістр, має вигляд

```
entity Reg is
port(
        X : in Std_Logic_Vector(7 downto 0);
        CLK: in Std_Logic;
        WE : in Std_Logic;
        Y: out Std_Logic_Vector(7 downto 0)
  );
end Reg;

architecture REG of Reg is
begin
process (CLK) is
variable U: Std_Logic_Vector (7 downto 0);
  begin

        if (WE = '1' ) then
                U := X;
  end if;
        Y<=U;
  end process;
end REG;
```

Щоб реалізувати універсальний регістр, необхідно дати користувачу можливість змінювати розрядність регістра без створення нових **entity** та без перекомпіляції регістра. Здійснимо це за допомогою включення необхідної константи:

```
entity Reg is
  Generic (N: integer range 1 to 63);
port(
        X : in Std_Logic_Vector(N downto 0);
        CLK: in Std_Logic;
        WE : in Std_Logic;
        Y: out Std_Logic_Vector(N downto 0)
```

```
    );
end Reg;

architecture REG of Reg is
begin
process (CLK) is
variable U: Std_Logic_Vector (N downto 0);
  begin

        if (WE = '1') then
                U := X;
  end if;
        Y<=U;
  end process;
end REG;
```

Тепер для включення 8-бітного або 16-бітного регістра треба записати відповідний оператор включення компоненти:

а) 8-бітний регістр:

```
  REG8: entity work.Reg
        generic map(N=>7);
        port map(X=>S1,CLK=>S2,WE=>S3,Y=>S4);
```

б) 16-бітний регістр:

```
  REG16:      entity work.Reg
        generic map(N=>15);
        port map(X=>S1,CLK=>S2,WE=>S3,Y=>S4);
```

4.11. Компоненти з попередньо описаним інтерфейсом

Опис *компоненти* представляє собою визначення інтерфейсу елементу системи, включаючи його статичні (константи **generic**) і динамічні (порти) складові. Механізм компонент дозволяє зробити архітектуру повністю незалежною і закінченою в межах даного ієрархічного рівня. Тобто попередній опис інтерфейсу компонентів дає змогу запускати процес аналізу проекту, що не містить архітектур –

реалізацій всіх включених об'єктів **entity**. В результаті відкриваються широкі можливості для розподілу великого проекту між окремими виконавцями.

Синтаксис *оператора оголошення компоненти* наведено нижче*:*

component *Identifier* **is**
  generic (*список_констант_generic*);
  port (*список_портів*);
end comoponent *Identifier*;

Даний оператор розміщується в розділах оголошень архітектури або в пакетах (**package**). В першому випадку компонента доступна в межах даної архітектури, у другому випадку – в межах дії оператора підключення пакету.

Для застосування описаної компоненти застосовується конкурентний оператор включення компоненти:

мітка_включення: **component** *Identifier*
generic map (*список_асоціювання_generic*)
port map (*список_асоціювання_портів*);

Для прикладу розглянемо реалізацію 4-бітного регістра як комбінацію чотирьох простих тригерів.

Інтерфейс регістра можна записати наступним чином:

entity *reg4* **is**
  port (*clk, clr*: **in** *Std_Logic*;
            d: **in** *Std_Logic_Vector* (0 **to** 3);
            q: **out** *Std_Logic_Vector* (0 **to** 3));
end entity *reg4*;

Програмна реалізація архітектури регістра має вигляд

architecture *struct* **of** *reg4* **is**
  component *FlipFlop* **is**
      generic (*Tprop, Tsetup, Thold*: *delay_length*);
      port (*clk*: **in** *Std_Logic*;
            clr: **in** *Std_Logic*;
            d: **in** *Std_Logic*;
            q: **out** *Std_Logic*);
  end component *FlipFlop*;

```
begin
  bit0:   component FlipFlop is
          generic map (Tprop=>2 ns, Tsetup=>2 ns,
                                          Thold=>1 ns)
          port map (clk=>clk, clr=>clr, d=>d(0),
                                          q=>q(0));

  bit1:   component FlipFlop is
          generic map (Tprop=>2 ns, Tsetup=>2 ns,
                                          Thold=>1 ns)
          port map (clk=>clk, clr=>clr, d=>d(1),
                                          q=>q(1));

  bit2:   component FlipFlop is
          generic map (Tprop=>2 ns, Tsetup=>2 ns,
                                          Thold=>1 ns)
          port map (clk=>clk, clr=>clr, d=>d(2),
                                          q=>q(2));

  bit1:   component FlipFlop is
          generic map (Tprop=>2 ns, Tsetup=>2 ns,
                                          Thold=>1 ns)
          port map (clk=>clk, clr=>clr, d=>d(3),
                                          q=>q(3));

end architecture struct;
```

Таким чином, наведений приклад ілюструє механізм реалізації алгоритму функціонування регістра на основі тригерів без реалізації самого тригера.

На відповідному етапі розробки проекту в будь-якому випадку слід описати, включення якого з об’єктів ***entity*** здійснює компонента, а також вказати архітектуру, що реалізує даний ***entity***. Задача асоціювання компоненти з конкретним об’єктом ***entity*** розв’язується оператором ***configuration***.

Синтаксис оператора ***configuration*** наведено нижче:

configuration *Identifier* **of** *Ім’я_entity* **is**

```
    for Ім'я_Архітектури
        for Ім'я_компонента
            use     entity Ім'я_асоційованого_entity1
                                (Архітектура1);
            use     entity Ім'я_асоційованого_entity2
                                (Архітектура2);
        end for;
    end for;
end configuration Identifier;
```

Запишемо оператор конфігурації для регістра, розглянутого у попередньому прикладі. При цьому вважаємо, що тригер реалізовано в пакеті *edge_triggered_Dff* бібліотеки *star_lib*:

```
library star_lib;                   -- Підключення бібліотеки
use
star_lib.edge_trigg_Dff;  -- Включення відповідного
                    -- пакету

configuration reg4_gate_level of reg4 is
    for struct              -- Вказано архітектуру об'єкта reg4
        for bit0: FlipFlop
            use entity
                edge_triggered_Dff (hi_fanout);
        end for;
        for others : FlipFlop
            use entity edge_triggered_Dff (basic);
        end for;
    end for;
end configuration reg4_gate_level;
```

Як видно з даного прикладу, за допомогою операторів ***component/ configuration*** можна призначати для різних включень однієї і тієї ж компоненти різні реалізації архітектур (для включення *bit0* призначено архітектуру *hi_fanout*, а для включень *bit1, bit2, bit3* – архітектуру *basic*). Отже, застосування механізму компонент підвищує рівень абстракції окремих ієрархічних рівнів проекту шляхом "маскування" деталей їх технічної реалізації, що в результаті робить процедуру розуміння проекту більш простою.

4.12. Паралельна робота процесів

Головною особливістю *процесів* є їх здатність працювати паралельно. Тобто, якщо всередині архітектури наявні два або більше *процесів*, *списки чутливості* яких містять один і той же *сигнал*, то при зміні значення цього *сигналу* всі *процеси* запускаються одночасно. При цьому дії всередині кожного з них виконуються послідовно і незалежно від інших паралельних *процесів*.

Для наочності розглянемо приклад:

```
architecture SomeArch of SomeEnt is
signal D, E : bit;
begin
P1: process (A, B, E)
 begin
 some_statement;
 some_statement;
 D<=some_expression;
 end process P1;
P2: process (A, C)
 begin
 some_statement;
 some_statement;
 E <= some_expression;
 end process P2;
P3: proces (D, E)
 begin
 some_statement;
 some_statement;
 end process P3;
end architecture SomeArch
```

Послідовність виконання процесів даного прикладу наведено на рис. 4.5. Якщо змінити значення сигналу *A* (змінені сигнали будемо позначати індексом (*)), два процеси *P1* і *P2* одночасно почнуть свою роботу. Внаслідок їх виконання зміняться значення сигналів *D* і *E*, які, в свою чергу, входять до списків чутливості процесів *P3* і *P1*. В результаті роботи процесу *P1* змінюється сигнал *D*, що призводить до нової ініціалізації процесу *P3*. Після виконання процесу *P3* жодному з