МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

КАФЕДРА ТЕОРЕТИЧНОЇ ЕЛЕКТРОТЕХНІКИ ТА ЕЛЕКТРОННИХ СИСТЕМ

РЯБЕНЬКИЙ В.М., УШКАРЕНКО О.О.

VERILOG

ПРОЕКТУВАННЯ ЦИФРОВИХ ПРИСТРОЇВ

Миколаїв – 2007

УДК 681.3
ББК 32.973.26-018.2

Рецензенти: докт. техн. наук, професор Смірнов В.С.
докт. техн. наук, професор Терещенко Г.А.

Автори: докт. техн. наук, проф. Рябенький В.М., зав. кафедрою Теоретичної електротехніки та електронних систем Національного університету кораблебудування ім. адм. Макарова
ст. викладач каф. ТЕЕС НУК Ушкаренко О.О.

Рябенький В.М., Ушкаренко О.О.
VERILOG. Проектування цифрових пристроїв.

В навчальному посібнику детально описуються основи проектування цифрових пристроїв на ПЛІС фірми ALTERA на основі САПР MAX+plus II та Quartus з використанням високорівневої мови опису апаратури (Hardware Description Language – HDL) – Verilog. Ця мова є міжнародним стандартом та використовується як системами аналізу (моделювання), так і системами синтезу цифрової апаратури. Приводиться велика кількість прикладів опису цифрових пристроїв, починаючи з реалізації логічних функцій і закінчуючи побудовою RISC-мікропроцесора. Посібник призначений для студентів, що досконало вивчають цифрову електроніку, а також буде корисним розробникам електронних пристроїв на сучасній елементній базі.

Зміст

ВСТУП

При розробці цифрових пристроїв широке поширення знайшли програмовані логічні інтегральні схеми, які дуже вдало доповнили мікропроцесорні засоби. Це зручна елементна база, яка використовується в системах цифрової обробки сигналів, телекомунікації, для побудови різноманітних систем керування та ін. Існує можливість створювати на кристалі спеціалізовані процесорні ядра, що дозоляє вирішувати будь-які науково-практичні задачі. Однак схемна реалізація подібних систем – задача досить складна, а іноді і неможлива. Для вирішення подібних задач може бути використана мова опису апаратури (HDL), однією з яких є Verilog. Він вигідно відрізняється загальноалгоритмічною базою, гнучкістю, широким діапазоном охоплення систем та рівнів їх опису. Проектувальник може скласти функціональний HDL-опис проектованої системи, і використовуючи систему моделювання САПР, перевірити її роботу. Verilog є офіційно признаним стандартом опису цифрової апаратури.

В навчальному посібнику розглядається велика кількість прикладів цифрових моделей, описаних за допомогою Verilog, які можуть бути синтезовані та запрограмовані в мікросхему. Детально описуються принципи проектування цифрових пристроїв на ПЛІС фірми ALTERA на основі САПР MAX+plus II Baseline та Quartus. Приводиться опис цифрових пристроїв з використанням різних стилів (структурний, поведінковий). Навчальний посібник призначений для студентів вищих навчальних закладів, які займаються проектуванням електронних пристроїв на основі програмованої логіки. Книга буде корисною для широкого кола спеціалістів в області цифрової електроніки, які починають знайомитися з мовами опису апаратури (HDL).

УМОВНІ СКОРОЧЕННЯ

ПЛІС (Програмовані логічні інтегральні схеми) – елементна база для побудови цифрової апаратури різного призначення. До переваг можна віднести: універсальність, порівняно низьку вартість, високу швидкодію та надійність, різне конструктивне виконання, різноманітність у виборі напруг живлення і параметрів сигналів, наявність розвинених та ефективних програмних засобів автоматизованого проектування.

САПР – Системи автоматизованого проектування – основний інструмент проектування цифрових систем на мікросхемах програмованої логіки. Можливості САПР визначають методику проектування та верифікації (перевірки) систем. В залежності від складності проекту, типу обробляємої інформації, способу її обробки, технічної бази, зручності роботи, вартості, вибирається та чи інша САПР.

HDL (Hardware Description Language) – мови опису апаратури. Програма на одній з таких мов може розглядатися як знакова модель дискретного пристрою, тобто така модель, яка відображає властивості об'єкту з використанням прийнятих формальних позначень. HDL–мови дозволяють будувати моделі як за описом закону функціонування, не вдаючись до конкретної фізичної реалізації, так і детальним представленням проекту на рівні вентилів та макрокомірок. Найбільш розповсюдженими та популярними є мови VHDL та Verilog.

АЛП, ALU (Арифметично-логічні пристрої) – це спеціалізовані мікросхеми, які виконують арифметичні та логічні операції у відповідності до двійкового коду, який подається на керуючі входи мікросхеми. При використанні логічних операцій АЛП всі зазначені в таблиці функції виконує порозрядно.

ПЗП – постійний запам'ятовуючий пристрій – пам'ять лише для зчитування, в яку інформація записується один раз на етапі виготовлення мікросхеми. Інформація в пам'яті не знищується при вимиканні живлення, тому її ще називають енергонезалежною пам'яттю

ОЗП – оперативний запам'ятовуючий пристрій – пам'ять, запис інформації в яку найбільш простий і може бути організований користувачем будь-яку кількість разів протягом всього періоду роботи мікросхеми. Інформація в пам'яті знищується при вимиканні живлення.

ЦСАР – цифрова система автоматичного регулювання. Представляє собою дискретну систему і характеризується наявністю квантування керуючих сигналів у часі та за рівнем.

BIN, HEX, OCT, DEC – формати представлення числових даних в проекті (двійковий, восьмирічний, шістнадцятирічний, десятковий). Найчастіше використовуються при описі проектів на одній з мов HDL. При роботі в Графічному редакторі, використовуються при створенні файлів ініціалізації пам'яті.

1. ОСНОВИ VERILOG

1.1. HDL-опис пристроїв

В загальному випадку опис пристрою за допомогою однієї з HDL-мов представляє собою текст, що зберігається в одному або декількох файлах, які в сукупності складають уявлення розробника про проект. Тексти описів на більшості мов проектування цифрових пристроїв по складу синтаксичних конструкцій схожі із традиційними мовами програмування. Як правило, створюючи програму розробник абстрагується від конкретної фізичної реалізації, виділяючи насамперед алгоритм функціонування проектованого пристрою. На основі моделі може бути побудований фізичний пристрій. В сучасних умовах основну роботу з інтерпретації моделі у фізичний об'єкт виконує комп'ютер. Як буде продемонстровано, сучасні мови дозволяють будувати моделі з різним ступенем наближення до реалізації – від опису закону функціонування до детального представлення проекту на рівні вентилів.

Вхідна інформація в дискретних пристроях – цифрові сигнали, які повинні надходити в систему через контакти і далі рухатися по ланцюгах до блоків, що виконують ті або інші перетворення. Джерелу сигналу, який подається на деякий ланцюг, в HDL-мовах зіставляється оператор, який присвоює значення змінній. Ця змінна представляє сигнал на цій лінії зв’язку. Такі оператори називаються драйверами. Основним типом в мовах є дані сигнального типу, по властивостях близькі до звичайних логічних даних. Основною відмінністю даних у HDL-мовах від логічних даних мов програмування є набір припустимих значень. Крім того враховується той факт, що сигнал має часові характеристики, зокрема, зміна значення змінної може відбуватися не відразу після присвоєння нової величини. Дані можуть поєднуватися в масиви, вектори. Крім того, застосовується ряд службових типів даних. Найбільш важливими з них є дані арифметичного типу.

Модель, що відображає об'єкт проектування у формі правил перетворення вхідних даних у вихідні, називається поведінковою. Структурна модель описує проект у вигляді сукупності модулів, кожен з яких реалізує певну частину завдання, і набору зв'язків між ними. Структурна модель є загальним способом створення складних проектів, опис яких доцільно виконати по ієрархічному принципу. В програмному модулі вищого рівня ієрархії присутні оголошення портів і внутрішніх зв'язків, а також декларації входжень модулів. Це ілюструє приклад асинхронного передавача. Як правило, першим етапом проектування є створення поведінкової моделі вищого рівня ієрархії. Часто може знадобитися кілька корекцій до одержання відповідності поведінкової моделі заданим вимогам. В процесі реалізації деякого проекту повна декомпозиція потрібна не завжди. На деякому етапі може виявитися, що

сукупність операторів, яка відтворюється деяким модулем, присутня у бібліотеці, що поставляється із САПР. В такому випадку досить у відповідному структурному тілі послатися на цей бібліотечний модуль. Загальна структура проекту зберігається і в цьому випадку. Опис модуля, який підключається, поведінковий або структурний, від розробника може бути приховано.

Однією з найважливіших проектних процедур є тестування. Проектування передбачає об'єднання найпростіших модулів в більш складну структуру. Тестування цих модулів дозволить виявити помилки в алгоритмах їх роботи на ранніх етапах реалізації проекту і у перспективі зменшити час на налагодження пристрою. Тестування розроблених модулів зручно виконувати за допомогою редактора часових діаграм.

Вибір стилю програмування багато в чому визначається досвідом розробника, але потрібно мати на увазі, що стиль може істотно впливати на апаратну реалізацію. Послідовний стиль опису відповідає традиційним підходам до написання комп'ютерних програм. Розташування і порядок виконання операторів в тексті опису відповідає порядку проходження даних через них. Паралельний стиль припускає асинхронне, одночасне виконання операторів в реалізованому пристрої. Блок, що реалізує деякий оператор, безпосередньо реагує на виконання дії будь-яким із його попередників. Для представлення паралельних процесів використовуються такі форми запису, які передбачають виконання кожного оператора після виконання хоча б однієї з ініціюючих подій. Порядок запису паралельних операторів байдужний. Припустимі зворотні зв'язки і петлі. Потоковий стиль відрізняється тим, що оператор виконується після готовності всіх попередників даного оператора. В чистому вигляді потоковий принцип досить складний в реалізації. Автоматний стиль опирається не на модель передачі даних, а на модель перемикання станів. Він заснований на визначенні деякої множини станів проектованого пристрою і правил переходу з одного стану в інший залежно від вхідних сигналів. Кожному стану або переходу відповідає певний набір дій. Такий підхід найбільш ефективний при описі пристроїв, що характеризуються циклічним виконанням однотипних послідовностей перетворень, зокрема керуючих пристроїв, процесорних модулів. Стиль створення проекту і використані засоби мови визначаються структурними особливостями проектованого пристрою. В реальних проектах, як правило, присутні структурні фрагменти різних типів, які описані різними способами (текстовий, графічний, у вигляді часових діаграм) і стилями програмування.

Відомі два протилежних підходи до побудови систем моделювання дискретних пристроїв – наскрізне моделювання і моделювання по подіям. При наскрізному моделюванні час ділиться на кванти, тривалість яких вибирається як найбільший час затримки компонента. Недоліки такого моделювання – нераціональні витрати машинного часу, викликані

потребою в дискретизації часу і необхідність відтворення на кожному кроці поведінки всіх компонентів, у тому числі і тих, на яких не відбувається зміна сигналу. Інші системи моделювання використовують подійний підхід, або дискретну модель подій. На кожному кроці моделювання переобчислюються стани тільки тих компонентів, на вході яких у цей момент відбуваються зміни. Зміни логічних змінних називають подіями. Будь-яка подія може викликати ланцюжок інших подій. Важливою характеристикою методу моделювання цифрових пристроїв є кількість помітних станів сигналу. Кожному стану сигналу зіставляється індивідуальний символ. Сукупність символів становить алфавіт моделювання. Кожний стан специфічно сприймається приймачем сигналів, тому в системі моделювання визначається набір правил перетворення сигналів типовими цифровими елементами. Найпростіший алфавіт – двійковий, який складається з 0 та 1. Функціонування елементів описується за правилами алгебри логіки. Однак неможливо описати шинну логіку, яка має високоімпедансний стан Z. Досить розповсюджений алфавіт із чотирьох символів {'0', 'X', '1', 'Z'}. X позначає невизначений стан. Такий сигнал присвоюється виходу логічного елемента під час перехідного процесу. Невизначений стан приймає тригер після подачі синхронізуючого сигналу при забороненій комбінації сигналів на інформаційних входах. Мова Verilog використовує саме цей алфавіт в якості базового. Однак проектувальник може присвоїти сигналу рівень сили. Різниця між слабким і активним станами полягає в тому, що слабкий сигнал формується від джерел з підвищеним вихідним опором в порівнянні з активними джерелами. Приклад елемента, що генерує слабку одиницю, – буфер з відкритим колектором.

Програми складаються з лексичних елементів, до яких відносять ідентифікатори, ключові слова, спеціальні символи та ін. Лексичні елементи поєднуються в синтаксичні конструкції – вирази. Вирази можуть містити декларації та оператори. Головне призначення декларацій полягає у визначенні змісту деяких ідентифікаторів (наприклад, вхідний або вихідний вивід, тип змінної). Оператори визначають дії, передбачені описаним алгоритмом. Для запису ідентифікаторів і ключових слів допускається використання тільки латинських літер. Кирилиця може використовуватися лише в коментарях.

1.2. Структура модулів Verilog. Коментарі та ідентифікатори

Verilog HDL – це мова опису апаратури, яка використовується для опису цифрових систем, наприклад комп'ютерних модулів. Компілятори Verilog, поряд з VHDL-компіляторами, включені в більшість САПР інтегральних схем. Синтаксис мови Verilog нагадує синтаксис мови програмування С. У Verilog менша в порівнянні з VHDL кількість службових слів, що спрощує його вивчення. Verilog дозволяє ефективно

виконувати опис і проводити моделювання та синтез цифрових схем завдяки вбудованим примітивам, засобам часового контролю, моделюванню затримок сигналів від входу до виходу. Крім того, через свої розширені можливості VHDL програє Verilog по ефективності, тобто для опису однієї і тієї ж конструкції на Verilog потрібно в кілька разів менше символів, чим в VHDL.

Говорячи про синтаксис мови Verilog слід зазначити, що він є контекстно-залежною мовою, тобто рядкові і прописні букви розрізняються. Всі ключові слова задаються в нижньому регістрі. Жоден проект не обходиться без його описання. Хороша документація означає не тільки багато сторінок тексту, але і наявність коментарів в тексті програми. В Verilog підтримуються два типи коментарів. Вони аналогічні прийнятим в C++. Якщо треба закоментувати один рядок, то використовуються дві косі риси на початку:

// це коментар

Щоб закоментувати кілька рядків, використовується наступна конструкція:

/ це*
*коментарі... */*

Коментарі істотно підвищують читання програмного коду. Під час компіляції коментарі ігноруються. Їх можна застосовувати для тимчасового відключення частини коду в процесі налагодження. Можливе використання комбінації різних типів коментарів. В складних проектах коментарі дозволяють швидше зрозуміти призначення тієї чи іншої конструкції.

Базовою одиницею мови Verilog є проектний модуль, або просто модуль, який інтегрує як визначення інтерфейсу, так і правила функціонування блоку. Кожен модуль починається з ключового слова **module**, за яким розташована його назва та перелік портів. Опис модуля закінчується ключовим словом **endmodule**. Між цими дома ключовими словами розташоване описання інтерфейсу – оголошення портів, перелік параметрів. При описі портів вказуються їхні імена, розрядність, напрям. Кожен порт може мати один з трьох напрямів:

- вхідний – данні зчитуються модулем з його оточення (інших модулів) через вхідні порти. Неможливо записати дані в такий порт;
- вихідний – дані відсилаються модулем з його оточення через вихідні порти. Зчитування даних з такого порту неправильне;
- двонапрямлений – дані можуть бути як зчитані, так і записані в такий порт.

Далі розташоване тіло модуля – його внутрішній вміст. Тіло модуля може складатися з оголошення змінних, операторів безперервного або процедурного присвоєння, задач та функцій. Відмітимо, що в тілі модуля

не може бути використана змінна, якщо вона не оголошена. Також всередині модуля може бути розташована директива компілятора **‘include**, за допомогою якої відбувається підстановка текстового фрагменту з іншого файлу.

Концепції мови дозволяють описувати пристрої з використанням різних рівнів абстракції. Модулі можуть містити оператори включення інших модулів, що дозволяє створювати ієрархічні проекти. Текст програми може містити довільний набір проектних модулів, кожний з яких представлений назвою, описом портів і тілом модуля. Модуль не може містити в собі опис іншого модуля. Загальна структура модуля наведена нижче:

```
module «ім'я_модуля»(«список_портів_модуля»); // заголовок модуля
    // опис портів модуля;
    // реалізація модуля:
    // оголошення сигналів, шин;
    // підключення інших модулів;
    // функціональний опис пристрою;
    // підпрограми;
endmodule  // кінець опису модуля
```

Ім'я модуля – ідентифікатор, що дозволяє використовувати модуль в інших модулях. Список портів модуля – перелік виводів, без вказівки їх напрямку і типу переданої інформації, за допомогою яких модуль підключається до інших частин схеми. Перелік портів розташовується в дужках після назви модуля. Опис портів модуля – це перше, що потрібно зробити при описанні модуля. Він представляє собою вказівку типу порту, його розрядності та назви. Після цього розташовується оголошення сигналів та шин. Ці сигнали визначаються в реалізації модуля – структурі модуля або операторів мови, які описують його поведінку. Розглянемо опис модуля, який реалізує прості логічні функції.

```
module primer(A1, A2, B1, B2);      // заголовок модуля
input A1, A2;                       // опис портів модуля: вхідні порти
output B1, B2;                      // вихідні порти
wire w1=A1;                         // оголошення сигналів
wire w2=A2;                         // що підключені до кожного з входів
    nand(B2,w1,w2);                 // B2=~(A1&A2) – елемент 2I-HI
    assign B1=A1 & A2;              // функціональний опис: B1=A1&A2
endmodule
```

В описі модуля ідентифікатори A1 та A2 – вхідні сигнали, B1 та B2 – вихідні сигнали. За допомогою ключового слова wire створюються два додаткових ланцюги w1 і w2, які підключені до вхідних портів A1 та A2 відповідно. За допомогою цих ланцюгів підключається елемент **nand**, який реалізує логічну функцію 2I-HI. Вихід логічного елементу підключений до виходу B2 описаного модуля. Сигнал на виході B1 представляє собою результат виконання логічної операції I над вхідними сигналами. Для цього був використаний оператор „&”. Більш детальний опис операторів приводиться нижче. За допомогою ключових слів **input** (вхідний) та **output** (вихідний) вказуються типи портів.

Ідентифікатори можуть складатися з латинських букв будь-якого регістра, цифр, знака підкреслення і знака долара. Ідентифікатор може починатися з букви або знаку підкреслення. Деякі ідентифікатори зарезервовані компілятором і використовуються для побудови мовних конструкцій. Вони називаються ключовими словами. У табл.1.1. приводиться список зарезервованих слів Verilog.

Таблиця 1.1. Зарезервовані слова Verilog

always	endmodule	large	reg	
and	ndprimitive	macromodule	release	tranif0
assign	endspecify	nand	repeat	tranif1
attribute	endtable	negedge	rnmos	tri
begin	endtask	nmos	rpmos	tri0
buf	event	nor	rtran	tri1
bufif0	for	not	rtranif0	triand
bufif1	force	notif0	rtranif1	trior
case	forever	notif1	scalared	trireg
casex	fork	or	signed	unsigned
casez	function	output	small	vectored
cmos	ighz0	parameter	specify	wait
deassign	highz1	pmos	specparam	wand
default	if	posedge	strength	weak0
defparam	ifnone	primitive	strong0	weak1
disable	initial	pull0	strong1	while
edge	inout	pull1	supply0	wire
else	input	pulldown	supply1	wor
end	integer	pullup	table	xnor
endattribute	join	rcmos	task	xor
endcase	medium	real	time	
endfunction	module	realtime	tran	

В Verilog використовуються складені імена – це послідовність імен, розділених крапками (з їхньою допомогою здійснюється доступ до елементів модулів, наприклад параметрів). У складних випадках, при великій кількості портів модуля і довільним порядком їх перерахунку в

оголошенні портів, використовується ключовий принцип: список портів утворює пари з назви порту модуля і назви приєднаного до нього ланцюга:

dff dff1(.d (data), .q (q_out), .clk (clock), .clrn (clearn), .prn (presetn));

Основна форма представлення інформації з якою оперує Verilog – це сигнальні дані, що відображають стан змінних і ланцюгів в пристрої, який описується. Крім того, використовуються службові дані, що служать для задання конфігурації, керування моделюванням і компіляцією, а також відображення результатів у процесі моделювання. До службових типів відносяться час, строчки. Тип даних задає діапазон можливих значень і операцій над об'єктом. Мова не дозволяє користувачам визначати свої власні типи даних.

1.3. Числа в Verilog

Verilog підтримує наступні «стандартні» типи даних: ціле – **integer** (32-бітове зі знаком) і **real** – число із плаваючою крапкою. При розробці синтезованих моделей з перерахованих типів використовується тільки **integer**. Тип **real** не підтримується при синтезі.

У реальному пристрої дані представлені двійковими кодами, а моделююча програма розуміє їх як сукупність незалежних елементів, кожний з яких може мати чотири можливі стани. Теоретично для представлення елементу досить двох бітів, але практична програмна інтерпретація може бути різною. Для задання вихідних значень змінних використовуються форми представлення у вигляді чисел. Варіанти синтаксису для представлення чисел мають вигляд:

1. <розрядність>'<основа><число> – повний опис числа.

2. <основа><число> – використовується розрядність представлення, яка задана в системі, але не менша 32 біт.

3. <число> – використовується десяткове представлення.

Основа системи счислення може бути однією з чотирьох видів (табл.1.2).

Таблиця 1.2. Основи систем счислення

Основа	Символ	Значення
двійкове	b або B	0, 1, x, X, z, Z, ?, _
вісімкове	o або O	0-7, x, X, z, Z, ?, _
десяткове	d або D	0-9, _
шістнадцяткове	h або H	0-9, a-f, A-F, x, X, z, Z, ?, _

При запису числових констант прописні і малі літери еквівалентні, наприклад:

```
8'b01101011          // 8-бітне число у двійковій системі
12'h5B               // 12-бітне число в шістнадцятковій системі
6'o43                // 6-бітне число у вісімковій системі
```

Від'ємні числа задаються за допомогою символу «–», який записується перед розрядністю числа. Приклади від'ємних чисел:

```
–8'h7C               // 8-бітне від'ємне число з основою 16_10
6'o–52               // неправильно
```

Варто звернути увагу на спосіб представлення від'ємних чисел: використовується доповнюючий код (доповнення до 2). Наприклад, якщо присвоїти деякій 16-розрядній змінній значення –12, то в ній буде зберігатися код 16'hfff4. Для відокремлення розрядів можна використовувати символ підкреслення.

```
16'b0101_1101_0110_1001
8'b_1101_0111              // неправильно
```

Для опису змінної цілого типу (**integer**) використовується наступна конструкція:

integer __integer_name;

Змінні цілого типу можуть бути використані для програмного обчислення конфігураційних параметрів проекту, але частіше застосовуються у фрагментах програм, в яких виконуються арифметичні перетворення. Якщо тип **integer** присвоєний сигнальній змінній, то передбачається, що вона представлена в реалізації 32-розрядним кодом, і її поведінка подібна до даних регістрового типу. Істотна відмінність цілих даних від регістрових – неможливість присвоєння їм невизначених значень. Крім того код, що зберігається в регістровій змінній, інтерпретується як позитивне число. Цілі ж представлені додатковим кодом.

Числа типу **real** можуть бути представлені або в десятковому виді, або в стандартній формі із плаваючою крапкою. Приклади дійсних чисел:

```
2.6
5.7e10
9.3e-8
7.          // неправильно
```

Змінні цього типу також мають властивості, подібні до регістрових даних, але в мові вводиться ряд обмежень на набір припустимих операцій

над ними. Фактично внутрішнє представлення (розрядність) визначається комп'ютером.

Тип даних часу у Verilog – це специфічний носій інформації, що має ціле значення. Над даними цього типу можна виконувати будь-які операції. Поточний відлік модельного часу зберігається в спеціальному регістрі часу і його значення може присвоюватися змінній, яка має тип "час".

Строчкові дані найчастіше використовуються для організації видачі повідомлень про виникнення визначених ситуацій в процесі моделювання, які передбачає розробник. Якщо строчка при виконанні програми може піддаватися модифікації, то вводитися змінна регістрового типу – бітовий вектор, число елементів якого достатнє для збереження будь-якого можливого значення строчкової змінної. Кожен символ представлений 8-бітовим кодом ASCII. Для відображення рядка на екрані викликається системна функція **$display()**. Наприклад:

```
reg [8·15:1] string;              // вектор довжиною 120 біт
initial                           // блок ініціалізації змінних
begin                             // початок блоку
      string="World!";            // присвоєння значення змінній
      $display ("\n Hello, %s!\n", string);   // вивід у вікно повідомлень
end                               // кінець блоку
```

Над рядками можна робити такі ж дії, як і над будь-якими іншими даними. Для конкатенації строчок використовуються фігурні дужки. В САПР MAX+plus II та Quartus тип даних **real** не підтримується.

1.4. Невизначений і високоімпедансний стан

Найбільш просте представлення про рівень сигналу дає двійковий (булевий) алфавіт, в якому сигнал може мати високий (TRUE, 1) або низький (FALSE, 0) рівні. Однак для практичного використання це занадто грубе представлення чисел. В Verilog використовується чотирьохзначний алфавіт з додатковими значеннями.

Символ „x" використовується для задання невизначеного стану (наприклад, вихід невстановленого тригера позначається як 1'bx), символ „z" показує високоімпедансний стан (наприклад обрив лінії). При використанні в якості цифри замість „z" у числах можна використовувати символ „?".

```
6'oz4      // запис числа z4 у восьмеричному форматі
8'h7x      // запис числа 7x у шістнадцятковій формі
12'dz      // запис числа в десятковій формі
```

Таблиці істинності логічних функцій І, АБО, ВИКЛ.АБО в чотирьохрозрядному алфавіті досить громіздкі. Однак немає необхідності пам'ятати їх. Можна використати принципи мажорування: в функції І мажорує 0 (якщо один з аргументів 0, то і результат також 0), у функції АБО – 1. Якщо в аргументах функції І немає 0 (а в АБО немає 1), але є X або Z, то результат дорівнює невизначеному значенню X. Наступний приклад демонструє це.

```
module major(d_and, d_or, d_xor, in1, in2);// заголовок модуля
output d_and, d_or, d_xor;          // вихідні сигнали
input in1, in2;                     // вхідні сигнали
      assign d_and=in1 & in2;       // логічна операція І
      assign d_or=in1 | in2;        // логічна операція АБО
      assign d_xor=in1 ^ in2;       // логічна операція ВИКЛ.АБО
endmodule
```

Описаний модуль має два входи та три виходи. Назва вихідного порту відповідає логічній операції, яка виконується над вхідними значеннями сигналів, а саме І, АБО, ВИКЛ.АБО. Діаграма на рис.1.1 демонструє поведінку модуля при різних значеннях сигналів на входах.

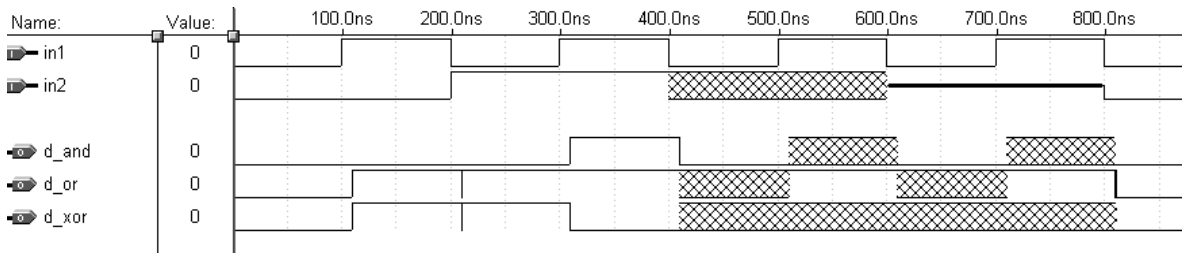

Рис.1.1. Демонстрація принципів мажорування

На часовій діаграмі високоімпедансний стан позначений товстою лінією. В такому стані знаходиться вхід in2 в проміжку часу від 600нс до 800нс. При виконанні логічних операцій це призводить до появи невизначеності (заштрихований фрагмент діаграми) на вихідних портах модуля. В момент часу 200 нс, при зміні сигналів на входах, на виходах виникають «голки». Причиною їх появи є неоднаковість затримок сигналів кожного з входів на вихід.

Всі сигнали в Verilog поділяються на 2 класи: ланцюги (**wire**) та регістри (**register**). Ланцюг представляє собою фізичне з'єднання між елементами системи. Стан ланцюга визначається значенням, яке приймає драйвер (джерело сигналу). В разі відсутності джерела сигналу ланцюг переходить у високоімпедансний стан.

1.5. Ланцюги і регістри

Сигнали представляються в чотирьохзначному алфавіті {0, 1, X, Z}. Однак для декларації ліній зв'язку, до яких підключаються джерела, що характеризуються специфічними електричними параметрами, драйверам сигналу можуть бути надані додаткові атрибути (рівні сили), які довизначають спосіб обчислення дійсного значення сигналу на лінії.

Введено два основних типи для представлення сигналів: «ланцюги» і «регістри». Кожній групі відповідають специфічні способи збереження присвоєних значень, і в остаточному підсумку, різні схемні рішення для їхньої реалізації в мікросхемі. Найпоширеніші з них описуються ключовими словами **wire** та **reg** відповідно. Однак варто пам'ятати, що засіб синтезу не завжди реалізує регістр у вигляді тригера. Відмінність ланцюга від регістра полягає в тому, що регістр здатний зберігати присвоєне значення (працює як змінна в мовах програмування), а для ланцюга потрібне безперервне джерело сигналу – драйвер (driver). Відмітимо, що концепція регістрового типу даних в Verilog відрізняється від цифрових регістрів, які складаються з тригерів, що тактуються. Регістри в Verilog не потребують наявності тактового сигналу. Сам по собі ланцюг не зберігає стан. Тобто ланцюг моделює провідник, що переходить у високоімпедансний стан при відключенні драйвера. Драйвер відповідає оператору безперервного присвоєння, причому назва драйвера така сама як і назва ланцюга. Це і відбивається в прийнятому в мові Verilog назві операторів цього типу – безперервні, які постійно впливають на ланцюг. Регістр в Verilog просто позначає сигнал, який не вимагає наявності драйвера. Якщо драйвер (джерело) сигналу має деяке значення, то і ланцюг приймає таке ж саме значення. Якщо драйвери ланцюга приймають різні значення, ланцюг приймає значення найбільш «сильного» сигналу. Якщо ж «сила» кожного з сигналів рівнозначна, то ланцюг приймає невизначений стан „x”. Ланцюги забезпечують безперервну модифікацію сигналів на виходах цифрової схеми у відповідності до зміни сигналів на їх входах. Значення регістра повинне бути призначене явно. Регістр зберігає записану в нього величину доти, доки не відбудеться нове призначення сигналу. Оператор присвоєння значення регістру діє як сигнал установки нового значення. В Verilog визначений широкий набір конструкцій для задання умов зміни станів регістрів і призначення різних способів керування тригерними пристроями. Якщо в декларації портів або змінних відсутнє визначення діапазону, то відповідна назва привласнюється скалярній змінній, що представляє стан одновихідного логічного елемента або однієї лінії з'єднання. В протилежному випадку передбачається оголошення бітового вектора. Для регістра початкова величина дорівнює „X”. Для моделювання джерел живлення використовуються ключові слова **supply0** та **supply1**.

Розглянемо приклад опису тригера з входом дозволу:

```
module DFF(q, data, en, rst, clk);      // заголовок програми
output q;                   // оголошення вихідного порту
input data, en, rst, clk;   // оголошення вхідних портів
reg q;                      // вихід має регістровий тип
always @(posedge clk)  // по фронту тактового сигналу
      if (rst==0)           // якщо сигнал сбросу встановлений в 0
            q=1'b0;         // на виході також установлюється 0
      else if (en==1)       // якщо сигнал  дозволу встановлений в 1
            q=data;         // на виході з'являється значення входу data
endmodule                   // кінець опису модуля
```

В приведеному описі вихід тригера q має регістровий тип. Це дозволило використовувати операцію присвоєння в процедурному блоці. На відміну від попередніх прикладів значення вихідного сигналу зберігається до наступного присвоєння, а не змінюється разом зі зміною вхідних сигналів. Операція присвоєння, а отже і установка на виході нового значення, відбувається по фронту сигналу clk. При цьому на вході дозволу en повинний бути сигнал лог.1. В протилежному випадку стан виходу залишається незмінним. Якщо в момент появи фронту тактового сигналу на вході rst був сигнал низького рівня, на виході q тригера встановлюється значення 0. Тобто тригер має інвертований вхід синхронного сбросу. Перевірка стану сигналів дозволу en та сбросу rst відбувається за допомогою оператора **if**.

Ключові слова **wand**, **wor**, **tri0**, **tri1**, **triand**, **trior** і **trireg** (це ланцюг, а не регістр) можуть бути використані для моделювання різних типів ланцюгів (**wand** – wired and – монтажне І, **tri0** – резистор до 0, **trireg** – ємність і т.п.), але такі ланцюги зустрічаються рідко. Вони використовуються для представлення різних варіантів монтажної і комутаційної логіки, які не характерні для ПЛІС.

Типи ланцюгів **wire** і **tri** з погляду програмної інтерпретації еквівалентні. Тип **wire** використовується для ланцюгів, підключених до одного джерела, а тип **tri** – для представлення зв'язків, які керуються декількома джерелами.

Приклад:

```
reg r1;             //  регістрова змінна
wire w1, w2;        // декларація двох ланцюгів
```

Дані регістрового типу (**reg**) і з'єднання (**wire**) допускають скаляри і вектори (одномірні масиви) багатозначного типу (значення 0, 1, x, z).

```
reg [7:0] rv;       // 8-розрядний векторний регістр
tri [3:0] bus3;     // 4-розрядна шина з 3 станами
```

Сигнали одного типу можуть бути оголошені одночасно. В такому випадку їх ідентифікатори повинні бути відокремлені комами.

Розглянемо приклад. Необхідно реалізувати логічну функцію:

$$y = x_3 x_2 x_1 x_0 + \overline{x}_3 x_2 x_1 x_0 + x_3 \overline{x}_2 \overline{x}_1 x_0 + \overline{x}_3 x_2 \overline{x}_1 \overline{x}_0 + x_3 x_2 x_1 \overline{x}_0 .$$

Складаємо схему з базових елементів (І, АБО, НІ) для цієї функції, не мінімізуючи її. Для цього потрібно запустити Графічний редактор системи MAX+plus II і ввести схему, вигляд якої представлений на рис.1.2.

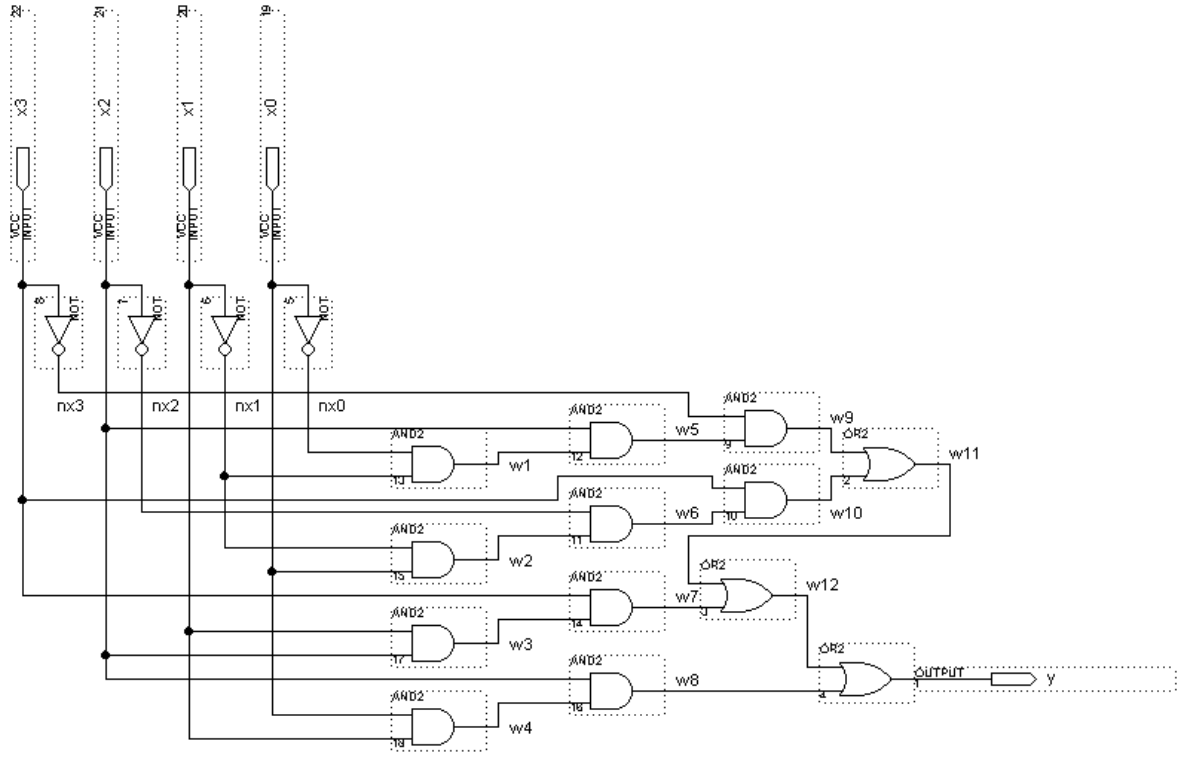

Рис.1.2. Схема реалізації логічної функції

Для реалізації схеми знадобилася велика кількість логічних елементів. Крім того, при розробці великої схеми можна дуже легко заплутатися. Часто в такому випадку розбивають схему на окремі функціональні блоки. Тепер проведемо аналіз часових затримок. Скористаємося для цього Timing Analyzer і Simulator. У таблиці на рис.1.3. приводяться значення часових затримок.

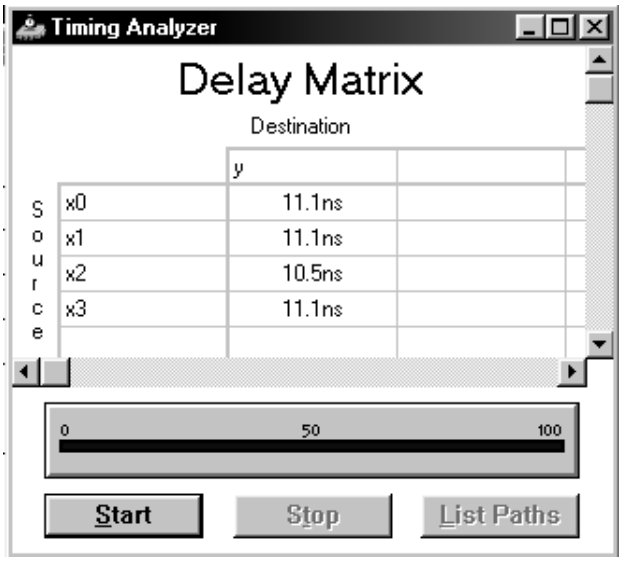

Рис.1.3. Значення тимчасових затримок

Як бачимо, максимальна затримка від входу до виходу становить 11,1 нс. Часова діаграма для логічної функції приводиться на рис.1.4.

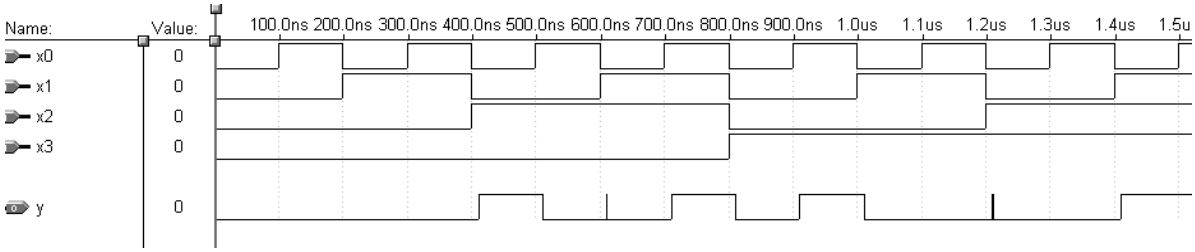

Рис.1.4. Часова діаграма схеми

На діаграмі, у моменти часу 600нс і 1,2 мкс, виникають паразитні імпульси. Причиною їх виникнення є різниця у величинах затримок сигналів для кожного із входів. Тривалість цих імпульсів становить 6нс.

Тепер спробуємо реалізувати цю логічну функцію за допомогою текстового опису на Verilog. В MAX+plus II створюється новий проект з назвою LogicFunction. У Текстовому редакторі (Text Editor) системи потрібно набрати опис схеми:

```
module LogicFunction(y,x0,x1,x2,x3,);      // заголовок модуля
output y;                                  // оголошення вихідних портів
input x0,x1,x2,x3;                         // оголошення вхідних портів
wire nx0,nx1,nx2,nx3,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12;
      // створення декількох ланцюгів
      not (nx3, x3);
      and (w1, nx0, nx1);                  // двовходовий елемент I
      and (w2, x0, nx1);
      and (w3, x1, x2);
```

```
    and (w4, x0, x1);
    and (w5, w1, x2);
    and (w6, w2, nx2);
    and (w7, w3, x3);
    and (w8, w4, x2);
    and (w9, w5, nx3);
    and (w10, w6, x3);
    or (w11, w9, w10);          // елемент АБО
    or (w12, w11, w7);
    or (y, w8, w12);
endmodule                       // кінець опису модуля
```

У даній реалізації, як бачимо, досить багато ланцюгів (**wire**) w1-w12. Їх назви відповідають позначенням на рис.1.2. Ланцюги використовуються як з'єднання між окремими логічними елементами. Однак велика кількість ланцюгів вносить деяку плутанину. Як буде продемонстровано пізніше, подібні завдання можна вирішувати не менш вдало, якщо звернутись до іншої форми опису схеми. Як видно з прикладу, прийняті в літературі алгебраїчні та інші форми задання систем булевих функцій, які описують функціонування комбінаційних схем, легко представити мовою Verilog. Зрозуміло, система булевих функцій може бути мінімізована в класі ДНФ і логічні вирази будуть спрощені.

Слід зазначити, назва модуля повинна бути такою самою, як ім'я файлу. Створений у Текстовому редакторі файл потрібно записати на жорсткий диск, змінивши розширення з .tdf на .v (рис.1.5.).

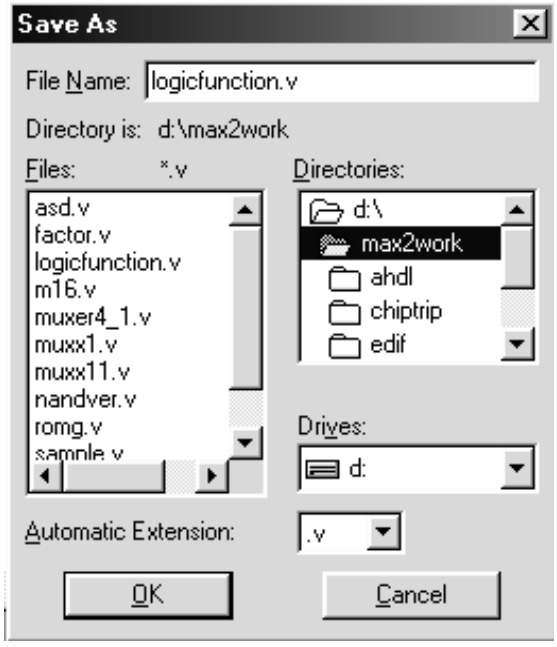

Рис.1.5. Вікно збереження файлу

Для цього з меню File необхідно вибрати меню Project і у ньому команду Save & Check, або нажати комбінацію клавіш Ctrl+K. Якщо в описі модуля не містяться синтаксичні і логічні помилки, система видасть відповідне повідомлення (рис.1.6.).

Рис.1.6. Повідомлення про вдалу перевірку проекту

Після компіляції проекту можна проводити моделювання схеми та аналіз часових затримок. Проаналізувавши часові затримки, що виникають у схемі при текстовому описі, можна прийти до висновку про те, що вони абсолютно однакові в порівнянні з графічним вводом.

Спробуємо мінімізувати логічну функцію. Це просто зробити за допомогою карти Карно. Вигляд мінімізованої функції:

$$y = \overline{x}_3 x_2 \overline{x}_1 \overline{x}_0 + x_3 \overline{x}_2 \overline{x}_1 x_0 + x_3 x_2 x_1 + x_2 x_1 x_0 .$$

Для її реалізації потрібні два елементи 4І, два елементи 3І, чотири інвертори (НІ) і один елемент 4АБО. Опис модуля має вигляд:

```
module LogicFunction(y, x0, x1, x2, x3);   // заголовок модуля
output y;                                  // опис вихідних сигналів
input x0, x1, x2, x3;                      // вхідні порти
wire nx0, nx1, nx2, nx3, w1, w2, w3, w4;   // оголошення ланцюгів
      not (nx0, x0);                       // реалізація логічної функції
      not (nx1, x1);                       // інвертор
      not (nx2, x2);
      not (nx3, x3);
      and (w1, nx0, nx1, x2, nx3);         // логічний елемент І
      and (w2, x0, nx1, nx2, x3);
      and (w3, x1, x2, x3);
      and (w4, x0, x1, x2);
      or (y,w1, w2, w3, w4);               // елемент АБО
endmodule                                  // кінець описання  модуля
```

В описі модуля, після його назви, в дужках вказуються спочатку виходи, потім входи схеми. Загалом, цей порядок не має принципового значення. Після цього за допомогою слів **input** (вхід) та **output** (вихід)

вказується тип виводу. Для з’єднання елементів всередині модуля необхідно використати декілька ланцюгів. За допомогою ліній nx0, nx1, nx2, nx3, w1, w2, w3, w4 проміжні результати з виходів одних логічних елементів подаються на входи інших. Логіка роботи пристрою може бути описана як на вентильному, так і на функціональному рівнях. В такий спосіб надається альтернатива рішення подібних завдань. При схемній реалізації комбінаційних схем, остання не містить елементів пам'яті. Цього варто уникати і при текстовому описанні комбінаційних схем. Бажано вживати змінні типу **wire** і утримуватися від використати змінних регістрового (**reg**) типу.

Система MAX+plus II містить бібліотеку шаблонів для опису проектів на таких мовах, як Verilog, VHDL, AHDL. Це полегшує деяким чином завдання розробнику, дозволяючи уникати синтаксичних помилок, а також зосереджувати увагу на самому проекті, а не деталях реалізації певних конструкцій мови. Для того, щоб скористатися бібліотекою шаблонів, на текстовому полі слід нажати праву кнопку миші. З'явиться контекстне меню, вигляд якого приводиться на рис.1.7.

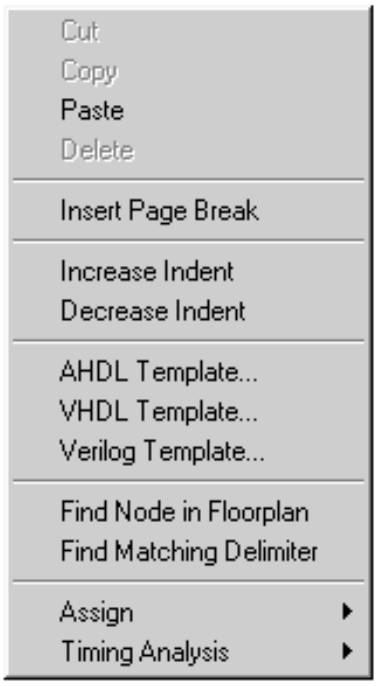

Рис.1.7. Контекстне меню Текстового редактора

Для відкриття бібліотеки шаблонів потрібно вибрати команду Verilog Template... З'явиться вікно, показане на рис.1.8.

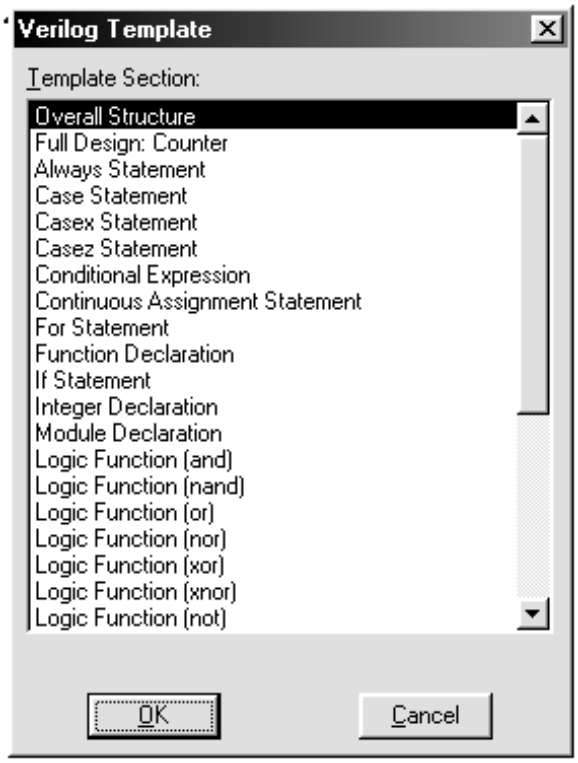

Рис.1.8. Шаблони конструкцій мови Verilog

У цій бібліотеці міститься велика кількість шаблонів, починаючи з шаблону всього модуля (Overall Structure), і закінчуючи шаблонами окремих конструкцій (Case Statement, For Statement, If Statement), логічних функцій (Logic Function (and), Logic Function (xnor) і ін.), оголошення змінних (Integer Declaration), оголошення ланцюгів і регістрів та ін. Наприклад, шаблон логічної функції І має вигляд:

and (__output_signal, __input_signal, __input_signal, ..);

Розробнику слід вказати замість __output_signal та __input_signal назви вихідних і вхідних сигналів, які використовуються в проекті.

Розглянемо, яким чином можна описати подібний елемент. Лістинг програми має вигляд:

```
module AND(out, in1, in2);          // заголовок модуля
// структурна модель вентиля І, складена із двох елементів 2І-НІ
output out;                         // вихідний сигнал
input in1, in2;                     // вхідні сигнали
wire w1;                            // декларація ланцюга
      nand (w1, in1, in2);          // два модуля 2І-НІ
      nand (out, w1, w1);
endmodule                           // кінець опису модуля
```

В свою чергу, елемент 2I-HI можна описати в такий спосіб:

```
module NAND(out, in1, in2);          // початок модуля
output out;                          // вихідний порт
input in1, in2;                      // декларація вхідних портів
      assign out = ~(in1 & in2);     // безперервне присвоєння
endmodule                            // кінець опису модуля
```

Подібні перетворення, в принципі, немає необхідності виконувати. Наведені приклади демонструють, яким чином можна одержати базові логічні елементи з декількома входами (більше двох), а також зробити потрібні входи інверсними (у правій частині оператора **assign**, перед ідентифікатором вхідного порту, поставити знак „~” – унарна операція інверсії (див. нижче)).

При описі портів (вхідних і вихідних), у загальному випадку, може бути використана наступна конструкція:

```
input  __input_name;                          // вхідний сигнал
input  [__input_range_msb:__input_range_lsb] __input_name; // вхідна шина
output __output_name;                         // вихідний сигнал
inout  __inout_name;                          // двонапрямлений вивід
```

Замість __input_range_msb та __input_range_lsb вказуються старший і молодший індекси масиву. Різниця цих значень визначає розрядність шини.

Засоби HDL для відображення структур цифрових систем базуються на уявленні про те, що об'єкт проекту представляє собою структуру з більш простих об'єктів – модулів, які з’єднуються один з одним лініями зв'язку (ланцюгами – **wire**). Кожен модуль, в свою чергу, є об'єктом і може складатися з модулів нижчого рівня (ієрархія об'єктів). Взаємодіють об'єкти шляхом передачі сигналів по лініях зв'язку. Лінії зв'язку підключаються до вхідних (**input**) і вихідних (**output**) портів створюваних модулів.

1.6. Вектори, масиви та пам'ять

До цього моменту ми розглядали тільки прості ланцюги, які представлені однією лінією. Сигнал на цій лінії в кожен момент часу може приймати один визначений стан. Однак в багатьох проектах потрібні багаторозрядні ланцюги, які називаються шинами, або векторами, по яким можуть одночасно передаватися різні значення сигналів. Хорошим прикладом є 32-розрядний мікропроцесор. По шині даних такого мікропроцесора в кожний момент часу передається 32 біти двійкової

інформації. Всі біти впорядковані і розташовуються, як правило, від молодшого розряду праворуч до старшого розряду ліворуч. Оголошення векторів, які мають тип **reg** подібно до оголошення векторів з'єднань **wire**. Приклади оголошень векторів представлені нижче:

```
wire [7:0] input_signal;          // 8-розрядний ланцюг
reg [15:0] out_data;              // 16-розрядний регістр
out_data [7:0] = input_signal;    // часткове призначення сигналу
```

При оголошенні вектору його розрядність вказується за допомогою індексів, які розташовуються в квадратних дужках ('[', ']'). Індекси розділені за допомогою двокрапки (:), і розташовуються між вказівкою типу сигналу та його ідентифікатором. Регістри, цілі числа і часові (**time**) типи даних можна оголошувати як масиви:

```
reg temp_data [31:0]              // 32 1-розрядних елемента
integer [3:0] data [15:0]         // 16 4-розрядних елементів
```

Двовимірний масив бітів в Verilog представляється винятково як пронумерована сукупність бітових векторів і називається пам'яттю. Відмінність між оголошеннями вектору та масивами полягає в розташуванні діапазону індексів: для векторів він розташовується між типом змінної та її ідентифікатором, а для масивів – після ідентифікатору. Для позначення пам'яті бажано використовувати інформативні ідентифікатори:

```
reg [7:0] mem8_512 [511:0];       // регістровий файл 512 × 8
```

Вектор – це один об'єкт, який складається з декількох бітів, а масив – це набір об'єктів, які можуть складатися з одного чи більше бітів. Під об'єктом в даному випадку мається на увазі ланцюг або регістр. Вектору може бути привласнене нове значення однією операцією (оператор „="), що неможливо для масиву. Багатовимірні масиви в мові не визначені. Для того, щоб отримати значення деякого біта (або декількох бітів) в пам'яті, потрібно скопіювати потрібне слово в додаткову змінну, а потім адресуватися до окремого розряду.

Наступний приклад демонструє використання векторів для вказівки розрядності ланцюга. Модуль представляє з себе інвертор 8-розрядної шини.

```
module inv(dout, din);   // заголовок модуля
output [7:0] dout;       // декларація вихідної шини
input [7:0] din;         // вхідна шина
```

assign dout = ~din; *// передача інвертованого значення на вихід*
endmodule *// кінець опису модуля*

На рис.1.9 представлена часова діаграма, отримана в результаті моделювання роботи інвертора.

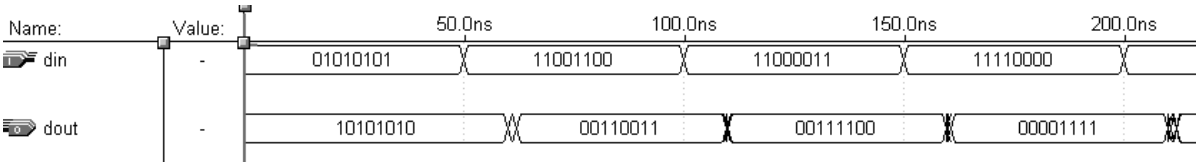

Рис.1.9. Діаграма роботи інвертора

З аналізу діаграми можна зробити висновки про правильність функціонування модуля інвертора. На виході dout з деякою затримкою з'являється інвертований код, який присутній на вході din.

Вектори можна використовувати разом з примітивами. Припустимо, є наступний фрагмент коду:

reg [3:0] a, b; *// оголошення двох 4-розрядних змінних регістрового типу*
wire [3:0] y; *// оголошення 4-розрядної лінії*
and g[3:0](y, a, b)*;// використання логічного елементу І*

Такий запис еквівалентний наступному:

and g3 (y[3], a[3], b[3]); *// використання логічного елементу 2І*
and g2 (y[2], a[2], b[2]);
and g1 (y[1], a[1], b[1]);
and g0 (y[0], a[0], b[0]);

Скалярні сигнали можна розглядати як окремий випадок вектору, який складається з однієї лінії, де старший розряд одночасно є молодшим розрядом.

1.7. Примітиви у Verilog

Verilog має велику кількість вбудованих примітивів (**primitive**) – логічні елементи І, АБО та інші, а також дозволяє розробнику створювати власні – примітиви користувача (UDP – User Defined Primitives). Примітиви представляють з себе базові функціональні блоки, за допомогою яких виконується структурний опис системи. Вони розділяються на чотири групи: з багатьма входами, з багатьма виходами, буфери з трьома станами та буфери, що підтягують. До вентилів з багатьма

входами відносяться наступні: **and** (І), **nand** (І-НІ), **or** (АБО), **nor** (АБО-НІ), **xor** (ВИКЛ.АБО) та **xnor** (ВИКЛ.АБО-НІ). До групи з багатьма виходами відносяться лише два вентилі: **not** (НІ) та **buf** (буфер). До групи буферів з трьома станами відносяться наступні: **bufif0**, **bufif1**, **notif0**, **notif1**. Керування буфером здійснюється за допомогою додаткового входу. В залежності від рівня сигналу на цьому вході (0 або 1), вихід буфера може приймати значення сигналу на вході чи знаходитися у стані високого імпедансу. На рис.1.10 представлене графічне зображення визначених у Verilog примітивів.

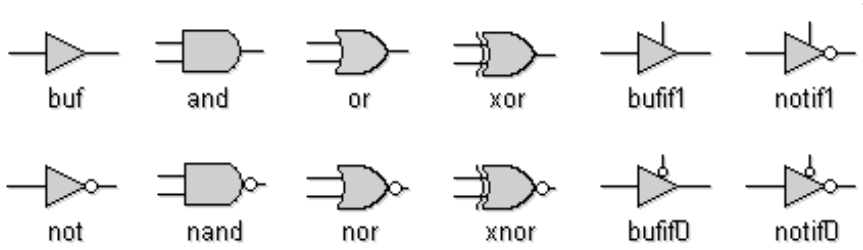

Рис.1.10. Примітиви Verilog

Хоча всі цифрові пристрої можуть бути побудовані з логічних елементів, це не завжди зручно на практиці. Наприклад найпростіший тригер буде представлений двома логічними елементами (вентилями) з перехресними зв'язками. Для того, щоб подолати цю незручність, Verilog дозволяє створювати власні примітиви користувача. Призначення UDP – моделювання та синтез логіки, яка представлена у вигляді таблиці істинності. Примітиви можуть бути двох типів: комбінаторні та послідовністні. В загальному випадку UDP представляють з себе більш складні елементи, ніж базові логічні елементи. Однак вони відрізняються від модулів як за описом, так і за способом описання. Примітиви користувача можуть мати лише один вихід. Вхідні і вихідні порти не повинні бути векторами. Нижче приводиться приклад створення і використання примітиву користувача 2І-НІ, а також спосіб його використання в системі Quartus. В САПР MAX+plus II не підтримується можливість створення таких примітивів.

```
module udp(y1,y2,a,b,c);        // заголовок модуля
output y1, y2;                  // вихідні порти
input a,b,c;                    // вхідні сигнали
wire w1;                        // додатковий ланцюг
      xor(w1,a,b);              // вихід примітиву – перший у списку
      and(y2,w1,c);             // логічний елемент І
      INE2(y1,a,b);             // підключення примітиву користувача
endmodule
```

```
                                    // опис примітиву користувача
primitive INE2(Y,X1,X2);
output Y;          // єдиний вихід повинен бути першим в списку параметрів
input X1,X2;       // всі порти (до 10) повинні бути однобітовими
      table        // таблиця станів комбінаційної схеми
            0 ? : 1;          // ? – байдуже значення аргументу
            ? 0 : 1;
            1 1 : 0;
      endtable                // кінець опису таблиці станів
endprimitive                  // кінець опису модуля
```

Початок опису примітиву починається із зарезервованого слова **primitive**, за яким розташовується його назва та перелік портів. Опис примітиву повинен закінчуватися ключовим словом **endprimitive**. Перелік портів завжди починається з назви одного вихідного порту, за яким через кому перелічені всі вхідні порти. Вихідний порт примітиву послідовністного типу повинен мати регістровий тип. Таблиця станів розташована між зарезервованими словами **table** та **endtable**. Якщо в таблиці визначені не всі можливі стани, в такому разі вихід перейде у невизначений стан. При описі примітивів послідовністного типу в таблиці істинності повинні бути вказані як поточний стан виходу, так і наступний. В таблиці істинності спочатку вказуються стани вхідних сигналів в тому порядку, в якому вони перелічені в переліку портів. Далі через двокрапку вказується значення, яке встановиться на виході у відповідності до вхідних сигналів. В кінці кожного рядка повинна ставитися крапка з комою. Кількість пробілів при описанні таблиці істинності не має значення. Результати моделювання наведеного опису представлені на рис.1.11.

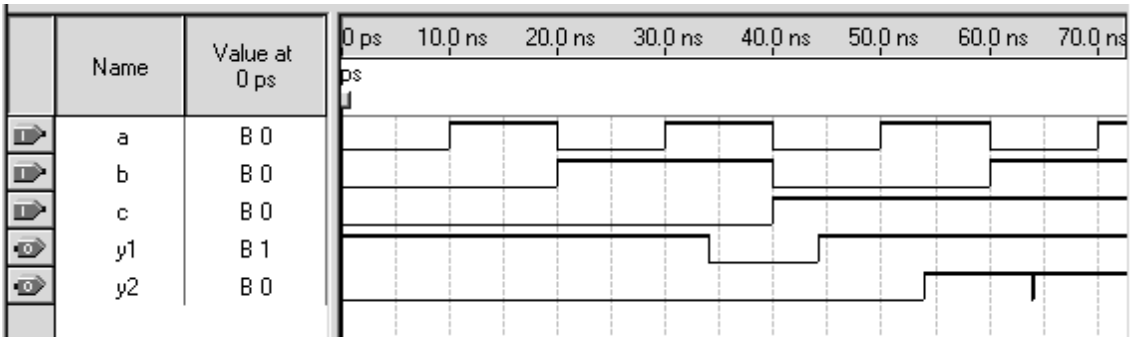

Рис.1.11. Діаграма роботи модуля

Як видно з діаграми, на виході y1 встановлюється сигнал високого рівня тоді, коли лише на одному з входів присутня лог.1. Описаний примітив представляє з себе логічний елемент 2I-HI. Можливість

використання великої кількості вхідних портів при описі примітивів дозволяє реалізовувати логічні функції, які представлені у вигляді таблиці станів.

Розглянемо приклад опису послідовністного примітиву Т-тригера.

```
primitive TFF(Q, Clk, Clr);     // початок опису примітиву
output Q;                       // вихідний порт
reg Q;                          // вихід має регістровий тип
input Clk, Clr;                 // тактовий сигнал та сигнал сбросу
initial Q=0;                    // початкове значення виходу
table                           // початок опису таблиці істинності
// Clk Clr : Q : Q+
   ?   1 : ? :  0 ;             // асинхронний сброс
   r   0 : 0 :  1 ;             // перемикання стану по
   r   0 : 1 :  0 ;             // фронту тактового сигналу
   f   0 : ? :  – ;             // ігноруємо спад сигналу Clk
   ?   f : ? :  0 ;             // ігноруємо спад сигналу Clk
endtable                        // кінець опису таблиці істинності
endprimitive                    // кінець опису примітиву
```

Елементи таблиці істинності можуть приймати значення, які приводяться в табл.1.3.

Таблиця 1.3. Можливі значення елементів таблиці станів

0	Логічний 0	
1	Логічна 1	
x	Не визначене значення	
?	Будь-яке значення	Не може бути використано в якості вихідного значення
b	Ітерація 0 і 1	Не може бути використано в якості вихідного значення
–	Без зміни	Може бути використано на виході послідовного UDP
vw	Значення, яке змінюється від v до w	v і w можуть бути 0, 1, x, ? та b
*	Те ж саме що і ??	Будь-яка зміна значення на вході
r	Перехід 01	Фронт на вході
f	Перехід 10	Спад на вході
p	Ітерація (01), (0x) і (x1)	Фронт на вході
n	Ітерація (10), (1x) і (x0)	Спад на вході

САПР MAX+plus II та Quartus містять в своїх бібліотеках велику кількість примітивів – буфери, тригери, логічні функції. В наступному прикладі демонструється використання простого D-тригера, примітиву LCELL та примітиву виводу з відкритим колектором OPNDRN.

```
module vprim (indata, outdata, clock);        // початок опису модуля
input indata, clock;                          // входи даних і тактовий
output outdata;                               // вихідний порт
reg out_dff, out_lcell;                       // додаткові змінні
      dff d1(.d(indata), .q(out_dff), .clk(clock));   // D-тригер
      lcell l1(.in(out_dff), .out(out_lcell));  // примітив LCELL
      opndrn o1(.in(out_lcell), .out(outdata));       // примітив OPNDRN
endmodule
```

За допомогою складених ідентифікаторів виконується з'єднання примітивів з портами модуля. Схема, представлена на рис.1.12, відповідає наведеному опису.

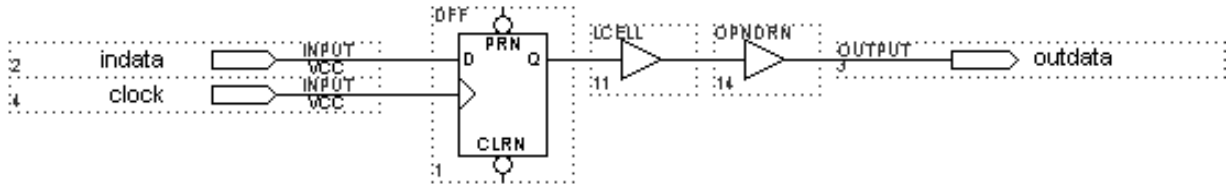

Рис.1.12. Приклад використання примітивів

Примітив OPNDRN подібний до буферу з трьома станами, однак має один вхід та один вихід. Якщо на вході присутній сигнал низького рівня, на виході також буде сигнал низького рівня. Якщо на вході встановлена логічна одиниця, вихід переходить у стан високого імпедансу, що демонструє часова діаграма на рис.1.13.

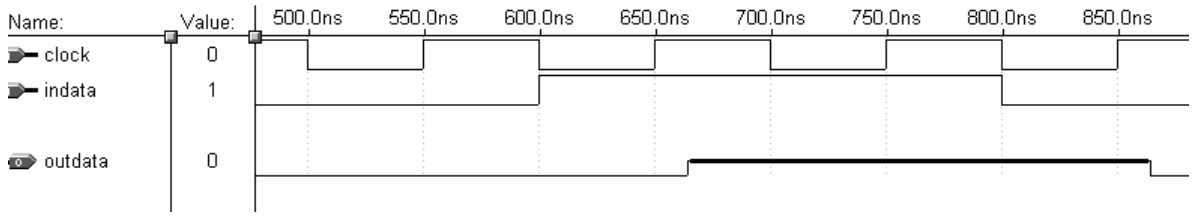

Рис.1.13. Діаграма роботи схеми

Буфер LCELL розподіляє логічні комірки в проекті. Він завжди використовує одну логічну комірку і враховується при синтезі. Розглянемо ще один приклад використання примітивів та буферу LCELL. В наступному модулі описаний генератор імпульсів.

```
module pulsar(outclk, en);          // заголовок модуля
output outclk;                      // вихідний сигнал
input en;                           // вхідний сигнал
wire w1, w2;                        // внутрішні ланцюги
        and(w1, en, outclk);        // логічний елемент I
        not(w2,w1);                 // інвертор
        lcell l1(w2,outclk);        // буфер lcell
endmodule
```

Наведеному опису відповідає схема, яка представлена на рис.1.14. На схемі позначені назви ланцюгів, які були використані при описі модуля.

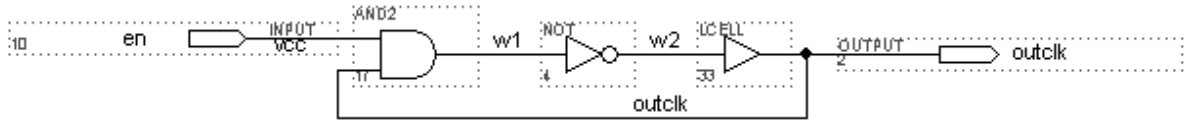

Рис.1.14. Схема генератора

При низькому рівні сигналу на вході en, на виході логічного елемента I також буде сигнал лог.0. Буфер LCELL працює як повторювач, тому на виході, завдяки присутньому в схемі інвертору, установиться значення лог.1, яке подається на другий вхід елемента I. При подачі високого рівня сигналу на вхід en, на виході елемента I також з'явиться 1, оскільки він буд працювати як повторювач. Після інверсії, з деякою затримкою через наявність в схемі елемента LCELL, на другому вході елемента I з'явиться рівень лог.0, що переведе вихід логічного елемента в низький стан. Таким чином цикл буде повторюватися і на виході схеми будуть присутні прямокутні імпульси доти, поки на вході en не з'явиться сигнал низького рівня. Це демонструє часова діаграма, представлена на рис.1.15.

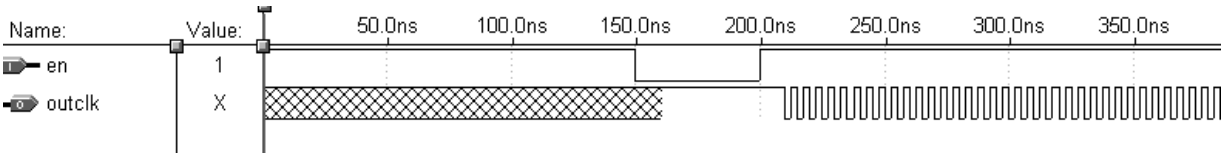

Рис.1.15. Діаграма роботи генератора

Збільшення кількості буферів LCELL в схемі призведе до збільшення затримки сигналу від входу до виходу, а отже зменшиться частота імпульсів на виході схеми. Збільшення кількості інверторів не призведе до подібного ефекту. Незважаючи на таку особливість роботи буферів LCELL, не рекомендується використовувати цей примітив для формування

внутрішньої затримки. Це пов’язано з тим, що часова затримка, яка вноситься цим елементом, залежить від температури, напруги живлення та технологічного процесу виготовлення мікросхеми. Вплив цих факторів може призвести до ненадійної роботи схеми.

Отже, користувальницькі примітиви значно розширюють набір стандартних вентилів, особливо послідовністного типу. Однак є обмеження на лише один вихідний порт. Це робить неможливим створення примітивів повного суматора або регістра, які повинні бути визначені як звичайні модулі. На відміну від модулів примітиви не можуть містити в собі екземпляри інших примітивів.

1.8. Оператори і вирази

В Verilog існують три типи операторів – з одним, двома та трьома операндами. Унарні оператори розташовуються ліворуч від операнду, бінарні – між двома операндами, а тернарний оператор розділяє три операнди двома операторами. Для формування результату у виразі операнди поєднуються знаками операції. В якості операндів можуть використовуватися числа, імена змінних (ланцюгів, регістрів), окремі біти і групи бітів, виклики функцій, які повертають значення кожного з перерахованих типів. Найпростіше – це скористатися записом арифметичного виразу за допомогою операторів, які доступні в мові Verilog. Крім загальноприйнятих арифметичних операцій та операцій відносин введена велика кількість логічних операцій. У табл.1.4. приводиться перелік символів операцій Verilog.

Таблиця 1.4. Символи операцій

Група операцій	Символи операцій	Найменування
Об'єднання	{}	
Арифметичні операції	+, – +, –, *, / %	Унарні арифметичні Бінарні арифметичні Модуль числа
Операції зсуву	>>, <<	Зрушення вправо або вліво на задане число розрядів
Операції відношення	>, >= <, <=	Більше, більше або дорівнює Менше, менше або дорівнює
Операції порівняння	==, != ===, !==	Дорівнює, не дорівнює
Операції згортки	&, ~& |, ~| ^ ^~, ~^	Згортка по І або І-НІ Згортка по АБО або АБО-НІ Згортка по «ВИКЛ.АБО» Згортка по «ВИКЛ.АБО-НІ»
Поразрядні операції	~	Інверсія

	&, ~& |, ~| ^ ^~, ~^	Порозрядне І (І-НІ) Порозрядне АБО (АБО-НІ) Порозрядне «ВИКЛ.АБО» Порозрядне «ВИКЛ.АБО-НІ»
Логічні операції	! && ||	Логічна інверсія Логічне І Логічне АБО
Умовна операція	?:	Якщо <умова> ? <значення 1> : <значення 2>

Оператори, які розташовані в таблиці вище, мають більший пріоритет у порівнянні з групами операцій, які розташовані в таблиці нижче. Для вказівки порядку обчислень рекомендується користуватися дужками. При використанні дужок більш високий пріоритет будуть мати вирази в дужках. Тип виконуваної операції визначається по положенню оператора у виразі.

Оператор об'єднання {} дозволяє збільшити розрядність ланцюгів і регістрів, об’єднує декілька операндів в один. Використовуючи цей оператор до вихідних сигналів можна однією операцією присвоєння встановити потрібні значення на виходах, що в свою чергу зменшує об’єм коду.

```
A = 8'b01011010;
B = {A[3:0] | A[7:4], 4'b0000}; // B призначається вектор 8'b11110000
C = {2{4'b1011}};               // C призначається вектор 8'b10111011
D = {{4{A[4]}}, A[7:4]};        // D призначається вектор 8'b11110101
```

Операнди, які підлягають об’єднанню, записуються у фігурних дужках і розділяються комами.

Проектування систем обробки інформації неможливо без реалізації операцій добутку, суми, різниці, ділення. Нижче приводиться приклад використання арифметичних операторів.

```
module arithmetic(dataA, dataB, add, sub, mul);
input [7:0] dataA, dataB;       // вхідні порти
output [8:0] add;               // вихідний порт суми
output [8:0] sub;               // вихідний порт різниці
output [15:0] mul;              // вихідний порт добутку
assign add=dataA+dataB;         // призначення вихідним портам
assign sub=dataA-dataB;         // результату виконання арифметичних
assign mul=dataA*dataB;         // операцій
endmodule
```

На рис.1.16 представлена часова діаграма, отримана в результаті моделювання.

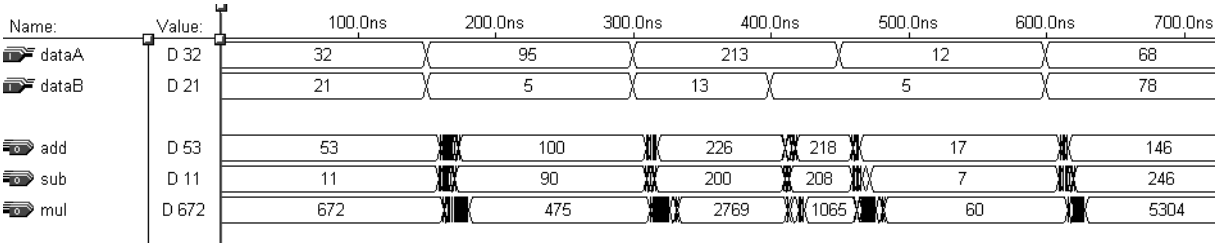

Рис.1.16. Діаграма роботи модуля

З діаграми видно, що результати операції множення дані встановлюються на вихідному порту з більшою затримкою, ніж для операцій додавання та віднімання. Дані регістрового типу дані розглядаються як ціле число у двійковому беззнаковому представленні. Якщо присвоїти змінній регістрового типу від'ємне число, то ця змінна буде представлена доповнюючим кодом:

```
reg [11:0] data;     // оголошення 12-розрядної шини  регістрового типу
   data=-12;            // data =  12'b111111110100
```

Нажаль, відсутні оператори „++", „– –" і всі оператори модифікації, які мають вигляд (операція) =, наприклад „*=", „^=" і т.п.

Оператори зсуву дозволяють здійснити зсув операнду як вправо, так і вліво. Цей оператор часто застосовують для реалізації регістрів зсуву, алгоритмів ділення та множення.

Результатом операції відношення буде скалярна величина, яка приймає значення 1, якщо елементи ліворуч і праворуч від символу операції задовольняють знаку операції, і 0 якщо не задовольняють. Якщо в коді кожного з операндів хоч в одному розряді є значення "x" або "z", результат приймає значення "x".

```
// regA=4'b1100   regB=4'b0011
regA==regB;       // оцінюється як 0
regA!=regB;       // оцінюється як 1
// regA=4'b1xz0   regB=4'b1100
regA==regB;       // оцінюється як x
regA!=regB;       // оцінюється як x
// regA=4'b1xz0   regB=4'b1xz0
regA==regB;       // оцінюється як x
regA!=regB;       // оцінюється як x
// regA=4'b1100   regB=4'b1100
regA==regB;       // оцінюється як 1
regA!=regB;       // оцінюється як 0
```

Для операцій ”===” та ”!==”, навіть якщо в кодах перебувають символи невизначеності, коди вважаються рівними, а при будь-яких відмінностях – нерівними. Результатом цих операцій може бути тільки 0 або 1.

```
// regA=4'b1100   regB=4'b0011
regA===regB;     // оцінюється як 0
regA!==regB;     // оцінюється як 1
// regA=4'b1xz0   regB=4'b1100
regA===regB;     // оцінюється як 0
regA!==regB;     // оцінюється як 1
// regA=4'b1xz0   regB=4'b1xz0
regA===regB;     // оцінюється як 1
regA!==regB;     // оцінюється як 0
// regA=4'b1100   regB=4'b1100
regA===regB;     // оцінюється як 1
regA!==regB;     // оцінюється як 0
```

При моделюванні комбінаторна логіка звичайно сприяє поширенню „x”, але якщо, наприклад, на один із входів елемента І (**and**) поданий 0, то незалежно від значення на інших входах, на виході буде 0 (принцип мажорування).

Результатом логічної операції може бути значення „1”, „0” і „x”. Найчастіше логічні операції використовуються для скалярних операндів, зокрема для об'єднання в одній умові результатів операцій відношення і порівняння. У відмінності від логічних операцій, порозрядні операції виконуються над парами однойменних розрядів операндів. Ці часткові результати збираються в тому ж порядку, що і в операндах, формуючи результуючий код. Якщо операнди мають різну розрядність, то більш короткий операнд доповнюється нулями з боку старших розрядів. Довжина коду дорівнює довжині більшого операнда.

Операції згортки мають ті ж позначення, що і порозрядні, але є унарними і дають скалярний результат. Всі розряди операнду розглядаються як скалярні аргументи відповідної багатомісної операції. При виконанні операції згортки по АБО розряди операнду аналізуються послідовно, один за одним, причому щоразу виконується операція АБО над значенням нового розряду і попереднім результатом. При виконанні операції згортки по І результат буде нульовим, якщо хоч в одному з розрядів міститься 0. Якщо ж у якому-небудь розряді буде невизначений або високоімпедансний стан, результат приймає невизначене значення у випадку, коли в операнді немає 0.

```
// regA=4'b0101   regB=4'b1011
&regA       // 0 & 1 & 0 & 1,  результат  0
&regB       // 1 & 0 & 1 & 1,  результат  0
|regA       // 0 | 1 | 0 | 1, результат 1
|regB       // 1 | 0 | 1 | 1, результат 1
^regA       // 0 ^ 1 ^ 0 ^ 1, результат 0
^regB       // 1 ^ 0 ^ 1 ^ 1, результат 1
```

У Verilog є умовний оператор ?:, який працює так само, як і у мові програмування С. Якщо при обчисленні умовного виразу отримане значення "істинно", то як вираз використовується <значення 1>, а якщо "хибно", то <значення 2> (див.табл.1.3.) Таким чином, найпростішим записом мультиплексора з 2 в 1 є:

assign Y=(SEL)?A:B;

Застосовуватися оператори можуть як до ланцюгів (**wire**), так і до сигналів регістрового типу (**reg**) і змінних. При цьому використовуються різні типи присвоєння. Для ланцюгів, які є моделлю фізичного з'єднання (провідника), потрібна наявність джерела сигналу, що виконується безперервним (continuous) присвоєнням. Значення ж регістрів і змінних можуть змінюватися в результаті процедурних дій і зберігатися між наступними призначеннями (так само, як і змінні процедурної мови програмування). Безперервне присвоєння вживається поза процедурними блоками **initial** або **begin** і використовується або в описі ланцюга, або із ключовим словом **assign** (існують також процедурні безперервні присвоєння в блоках **initial** або **always** із ключовими словами **assign** та **deassign**). Ліворуч від оператора безперервного присвоєння (=) повинен перебувати об'єкт типу ланцюг. При зміні значення якого-небудь із об'єктів, що входять у вираз праворуч від „=", даний вираз буде обчислено призначене нове значення сигналу. Оператор ініціалізації **initial** запускається єдиний раз при моделюванні і використовується для задання початкових станів моделюємого пристрою. Оскільки в системі MAX+plus II для моделювання використовується Сигнальний редактор (Waveform Editor) і Симулятор (Simulator), компілятором ігнорується ключове слово **initial** і всі операції, які містяться в цьому блоці.

Вираз – це конструкція, що поєднує операнди і знаки операції для формування результату. У мові Verilog в якості операнду можуть використовуватися числа, імена змінних (ланцюгів, регістрів, цілих змінних), виділені біти і групи бітів регістрів і ланцюгів, а також виклики функцій, які повертають значення одного з перерахованих типів.

За рідкісним винятком дані різних типів сумісні в одному виразі. Навіть якщо операнди мають різну розрядність, їх можна поєднувати в одному виразі. При цьому відбувається розширення розрядності більш короткого операнда.

1.9. Оператори initial та always. Контроль подій

В Verilog розрізняють два види операторів – паралельні і послідовні. Послідовні оператори виконуються один за одним, в порядку запису. Паралельні оператори виконуються при будь-якій зміні сигналу, який використовується як аргумент. Послідовні оператори повинні бути в тілі складених операторів.

Оператор **initial** запускається до виконання один раз при початку моделювання і використовується для задання початкових станів у пристрої, який моделюється. Оператори ініціалізації починають виконуватися одночасно (їх результати недоступні для операторів, що мають нульову позначку часу). У системі MAX+plus II, як було вказано вище, для моделювання використовуються Симулятор (Simulator) і Сигнальний редактор (Waveform Editor). У Сигнальному редакторі є засоби для керування процесом моделювання. Існує можливість задавати крок моделювання, початкові значення на вхідних портах. Оператор **initial** ігнорується компілятором при синтезі логічних кіл. Такий блок служить для підтримки функціонування моделі, як, наприклад, для формування початкових значень і тестових послідовностей.

Оператор постійного повторення **always** перший раз виконується при початку моделювання, але потім повторюється після завершення вкладеного оператора. У зв'язку із циклічним характером цього оператора, він може мати сенс при наявності в його тілі конструкції, що передбачають його перехід у режим очікування. Найчастіше це очікування яких-небудь подій у системі. Елементарною формою події є зміна значення ланцюга або регістра, яка відслідковується оператором очікування події @.

```
reg data_out;
always @(a or b or c)    // очікування зміни сигналів a, b та c
if (b)                   // якщо b – істина, то data_out = a
        data_out = a;
else                     // інакше data_out = c
        data_out = c;
```

Припустимо, є наступний опис:

```
always @ (posedge CLK) a=b;      // по фронту CLK виконується
always @ (posedge CLK) b=a;      // оператор присвоєння
```

Нехай a і b – регістри одиничної довжини, і до моменту позитивного фронту тактового сигналу CLK a = 0, b = 1. Яке значення будуть мати ці змінні після проходження фронту? Це не визначено в мові і залежить від того, у якій послідовності операції присвоєння потрапляють у список. Тобто виходить конструкція, поводження якої непередбачене. Це означає,

що обидві ці змінні приймуть значення 0 або 1. Конкретний результат залежить від симулятора і від послідовності, в якій будуть оброблені рядки вихідного файлу. «Блокуюче», або «блокове» присвоєння (=) забороняє виконання інших послідовних операцій доти, доки не буде виконана задана операція. Використання операції блокуючого присвоєння у подібній конструкції небажано і його варто уникати. Але в той же час, якщо в блоці потрібно провести послідовне виконання операторів, варто застосовувати даний тип присвоєння.

В якості активізуючої події може використовуватися фронт або спад сигналу. Конструкція

```
always @ (posedge CLK)          // початок процедурного блоку
begin
      a=0;                      // a=0
      b=a;                      // b=0
end
```

гарантує, що після фронту CLK обидві змінні – a та b – будуть мати значення 0. Один з фундаментальних принципів Verilog – будь-який об'єкт, якому присвоюється значення в процедурному блоці **always**, повинен мати регістровий (**reg**) тип.

Ознакою контролю події є символ @. У розглянутих раніше прикладах контроль подій використовувався в блоках **always**. Відмінність Verilog від VHDL в цьому випадку полягає в тому, що для опису фронтів і спадів сигналів використовуються не спеціалізовані атрибути сигналу, а спеціальна конструкція мови. Контроль подій використовується в процедурних блоках так само, як і часовий контроль. При цьому затримка виконання відбувається не на фіксований часовий інтервал, а доти, доки не відбудеться потрібна подія.

Події бувають наступних типів:

- @ (name) – зміна name, при цьому name може бути ланцюгом, регістром;
- @ (**posedge** A) або @ (**negedge** A) – фронт або спад сигналу A, при цьому A – однобітовий регістр або ланцюг;
- комбінація перерахованих подій із ключовим словом «**or**», наприклад @(**posedge** CLK **or posedge** SET); в цьому випадку варто розрізняти «**or**» з однойменною логічною операцією; у конструкції контролю подій «**or**» означає, що очікується будь-яка з перерахованих подій, а не певний результат логічної операції.

Таким чином список чутливості, що містить слово **posedge**, забезпечує виконання оператора тільки при зміні результату обчислення виразу, записаного в дужках, з нульового стану в одиничне. Аналогічно, ключове слово **negedge** задає чутливість тільки до зміни сигналу з

одиничного стану в нульовий. У префіксах з вказаними ключовими словами вирази повинні бути скалярними (розрядність 1 біт). Отже, коли відбувається виконання деякого виразу при виникненні події, то цей вираз є чуттєвим до цієї події. Verilog дозволяє визначати вирази, які будуть виконані при виникненні однієї з декількох подій. В такому разі назви сигналів повинні бути відокремлені словом or, а вирази знаходитися в блоці **begin** – **end**. Контроль подій дозволяє зупинити, або навпаки, відновити виконання нескінченного оператора **always**.

В мові Verilog порядок виконання операторів визначається не тільки порядком їхнього запису. Передбачено широкий набір засобів, що визначає умови, при яких оператор буде виконаний. Ці умови оформляються у вигляді префіксів операторів і виразів мови. У прикладах ми не один раз використовували блокове представлення програми. Складені оператори, оператори повторення являються простими прикладами блоків. Блоки поєднують оператори, зв'язані загальними правилами ініціалізації. Розрізняють послідовні і паралельні блоки. Такі блоки без обмежень можуть бути вкладеними один в інший. Хорошою можливістю для покращення структурованості програм є використання іменованих блоків – складних операторів, яким присвоюється унікальний ідентифікатор. Синтаксис послідовного іменованого блоку має вигляд:

```
begin : block      // відкриваюче слово і назва блоку
      …            // оператор
end                // закриваюче слово
```

Послідовний блок обмежується відкриваючим і закриваючим словами **begin** та **end**. Оператори в послідовному блоці виконуються один за одним, у порядку запису. Виконання завершується після реалізації останнього оператора в блоці. Наступний приклад демонструє використання операторів відношення, а також конструкції **always** для контролю зміни вхідних сигналів. Модуль представляє собою компаратор, який має два 4-розрядних інформаційних входи A та B, а також чотири виходи Q1 – Q4.

```
module rational(A,B,Q1,Q2,Q3,Q4); // заголовок модуля
input [3:0] A,B;          // вхідні дані
output Q1, Q2, Q3, Q4;    // перелік вихідних сигналів
reg Q1, Q2, Q3, Q4;       // виходи мають регістровий тип
always @(A or B)          // процедурний блок
      begin               // початок блоку
            Q1=A>B;       // на кожному з виходів встановлюється
            Q2=A<B;       // значення сигналу відповідного до
            Q3=A>=B;      // результату виконання операції
```

```
        Q4=A==B;
    end                 // кінець блоку
endmodule               // кінець опису модуля
```

На рис.1.17 приводиться часова діаграма, отримана після моделювання.

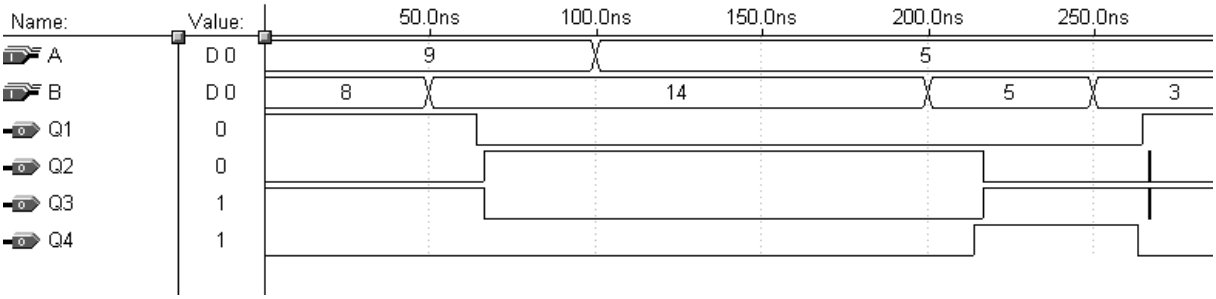

Рис.1.17. Діаграма роботи модуля

В залежності від кодів, які присутні на вхідних портах модуля, на вихідних портах встановлюються логічні рівні, відповідні до результату виконання операції відношення. Рівні сигналів на виходах переобчислюються щоразу при зміні коду на будь-якому з вхідних портів.

Якщо необхідно, щоб всі оператори в блоці виконувалися одночасно, їх необхідно розташовувати між словами **fork** та **join**. Порядок запису операторів в такому блоці не має значення. Зміни значень сигналів, які виробляються в інших блоках перед виконанням кожного оператора, враховуються в результатах обчислення виразів. В цілому, зручніше описувати поводження в такому блоці за допомогою безперервних або блокуючих присвоєнь.

В Verilog існує можливість переривання виконання іменованого блоку за допомогою команди **disable**. Наступний приклад демонструє використання команди **disable**.

```
module couner(clk, count);          // заголовок модуля
input clk;                          // вхідний тактовий сигнал
output [7:0] count;                 // вихідний порт
reg [7:0] count;                    // вихід має регістровий тип
initial
    begin
    count=10;                       // початкове значення лічильника
        begin: counting             // іменований блок
        forever
        begin
            @(posedge clk)          // по фронту тактового сигналу
            count=count+1;          // інкремент значення лічильника
```

```
                    if (count==71)      // якщо значення лічильника 71
                    disable counting;   // переривання виконання блоку
                    end                 // forever
              end                       // counting
        end                             // initial
endmodule                               // кінець опису модуля
```

Описаний модуль представляє з себе лічильник, який починає рахувати з count=10 і закінчує при досягненні значення count=71. Іменований блок містить нескінченний цикл **forever**. За допомогою оператора **if** відбувається перевірка умови виходу. Якщо умова виходу дотримана, то виконується команда **disable** і виконання циклу зупиняється. Оператор **disable** також може бути використаний в циклах, задачах (**task**). Загалом, він використовується разом з оператором умови, що дозволяє створити компактний та гнучкий механізм керування виконанням блоків.

Контроль подій і список чутливості найчастіше використовується в блоці **always**. В Verilog існує інша форма подій, яка може бути створена за допомогою ключового слова **event**. Такі події називаються іменованими подіями (named events) і не містять в собі яких-небудь даних. Вони можуть бути ініційовані явно і використовуватися для контролю послідовності виконання поведінкових блоків. Наступний приклад демонструє використання подій. Модуль виконує функцію контролю парності вхідного слова.

```
module events(control, din);    // заголовок модуля
output control;                 //  вихідний порт
input [3:0] din;                // вхідне слово
reg control;                    // вихід має регістровий тип
event setone, setzero;          // оголошення подій
always @(din)                   // очікування зміни вхідного слова
      begin
      if(^din==1) ->setone;// якщо в слові непарна кількість одиниць
                                // ініціюється подія setone
      else ->setzero;           // інакше виникає подія setzero
      end
always @(setone)                // очікування події setone
      begin
            control=1'b1;       // установка на виході 1
      end
always @(setzero)               // очікування події setzero
      begin
            control=1'b0;       // установка на виході 0
      end
```

endmodule *// кінець опису модуля*

В модулі за допомогою оператора згортки по ВИКЛ.АБО визначається кількість одиниць у вхідному слові, і в залежності від цього відбувається активізація однієї з подій оператором −>. За допомогою конструкції **always** відслідковується кожна з двох подій і на виході встановлюється потрібний рівень сигналу. Іменовані події можуть використовуватися для керування активізацією подій, і не можуть бути використані для передачі інформації.

Контроль подій має відношення до зміни стану ланцюга чи змінної (регістру). Але іноді важливий факт не самої зміни сигналу чи змінної, а значення, яке вона приймає, наприклад рівень сигналу дозволу в буфері з трьома станами. В мові Verilog крім оператору @, який дозволяє визначати зміну сигналів, існує також оператор wait, який призначений для реагування на досягнення сигналом визначеного значення. Конструкція **wait** починається з ключового слова **wait**, за яким в дужках розташована умова. Далі записується вираз, який виконується в залежності від умови:

wait(en) A=B+C; *// якщо en=1, виконується оператор додавання*
wait(A==8'h7D) C=A-B;*// якщо $A=7D_{16}$, виконується оператор віднімання*

Виконання блоку призупиняється доти, доки вираз у дужках не прийме значення „істино" (true). Наступний приклад, що представляє собою модуль суматора, демонструє використання оператора **wait**.

```
module counter(dAin, dBin, go, done, dout, cout); // заголовок модуля
input [7:0] dAin;          // вхідний порт
input [7:0] dBin;          // вхідний порт
output [7:0] dout;         // вихідний порт
input go;                  // вхід запуску роботи модуля
output done, cout;         // сигнали готовності даних та переносу
reg [7:0] dout;            // виходи мають регістровий тип
reg done;
reg cout;
always                     // цей блок виконується нескінченно
      begin
      done = 0;            // установка в 0 сигналу готовності даних
      wait (go);           // очікування сигналу запуску
            {cout,dout}=dAin+dBin;// визначення суми
            done=1;        // установка сигналу готовності даних
      wait (~go);          // очікування низького рівня сигналу запуску
      end
endmodule                  // кінець опису модуля
```

В наведеному прикладі оператор **wait** використовується для зупинки виконання блоку **always** доти, доки сигнал go не прийме значення 1. Якщо це сталося, відбувається визначення суми значень, що присутні на вхідних портах dAin, dBin. Для зручності використаний оператор об'єднання {}, що дозволило відслідковувати переповнення і встановлювати сигнал cout. Коли значення суми визначене, на вихідному порту done встановлюється 1.

1.10. Оператори блокуючого та неблокуючого присвоєння

Операція присвоєння може входити в оператори паралельного присвоєння або трактуватися як окремий оператор. При виконанні присвоєння операнд, що записується ліворуч від знаку рівності, приймає значення, яке одержується при обчисленні виразу, записаного праворуч від знаку рівності. Приймачем може бути проста змінна, елемент вектора, група розрядів вектора, а також об'єднання зазначених типів. Якщо приймач – сукупність об'єктів, то її елементи записуються у фігурних дужках через коми. У випадку групового приймача розряди результату присвоюються розрядам елементів приймача в порядку їхнього запису в списку елементів приймача. В Verilog визначені два основних типи операторів присвоєння – безперервні і процедурні. Безперервні оператори завжди розглядаються як паралельні, тобто виконуються при зміні будь-який змінної, присутньої в правій частині операції присвоєння. Процедурні виконуються тільки за певних умов, які задаються спеціальними конструкціями. Вони можуть бути послідовними, тобто виконуватися один за одним в порядку запису, якщо локалізовані в послідовних блоках, або паралельними, якщо включені в паралельні блоки.

Приймачем в операторі безперервного присвоєння може бути тільки змінна типу "ланцюг", скалярна або векторна. Шаблон оператора паралельного присвоєння має вигляд:

```
assign __identify_name = __value;
       __identify_name = __value;
```

де __identify_name – ім'я змінної типу **wire**, __value – значення.

Оператор виконується щоразу при зміні значення __value, яка присутня в правій частині оператора присвоєння. Поки не виконані всі оператори, ініційовані загальною подією, аргументами перетворень є стани сигналів на момент початку відпрацювання цієї події.

```
module contacc(out1,out2,out3,in);   // початок модуля
input in;                            // опис вхідних портів
output out1, out2, out3;             // опис вихідних портів
wire out1, out2, out3;               // вихідні порти мають тип "ланцюг"
      assign out1=in;                // безперервне призначення сигналів
```

```
    assign out2=out1;
    assign out3=in;
endmodule                    // кінець опису модуля
```

На рис.1.18 показана часова діаграма роботи модуля.

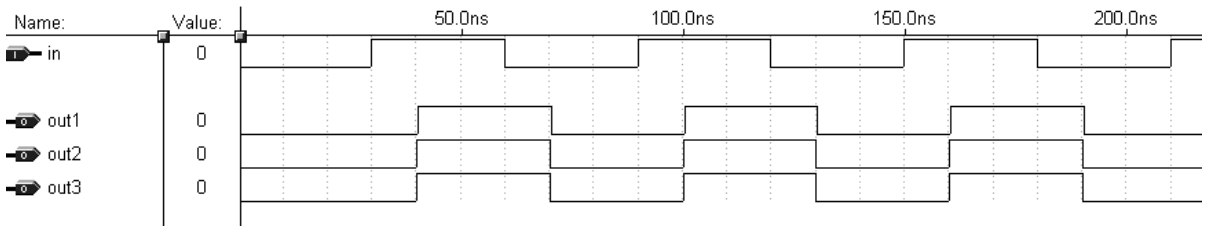

Рис.1.18. Часова діаграма, що демонструє
роботу оператора безперервного присвоювання

У випадках, коли розробника цікавить час затримки для окремих розрядів індивідуально, варто записувати не групове присвоєння, а індивідуальне присвоєння для розрядів.

Мова Verilog дозволяє описувати пристрої, у яких кілька джерел працюють на одну лінію, при цьому кожному джерелу зіставляється власний оператор безперервного присвоєння змінної, що представляє сигнал на цій лінії. Якщо при цьому лише одне з таких джерел може бути в активному стані, то ситуація описується відносно просто: якщо хоч одне джерело перебуває в активному стані, передане їм значення, як більш сильне, приглушує слабкі сигнали, що представляють джерела у відключеному стані. Однак у практиці може бути необхідним виявлення при моделюванні збійних ситуацій, пов'язаних з некоректною активізацією декількох драйверів, підключення до шин схем з відкритим колектором і подібними компонентами, вихідний опір яких буде неоднаковим, а виходить, і вплив на результат також буде відрізнятися. Силою драйвера називається конструкція, що визначає ступінь такого впливу. Якщо на сигнал діє кілька драйверів, які мають різний рівень сили, то сигналу присвоюється те значення, яке має найбільшу силу.

Послідовні оператори присвоєння локалізуються в послідовних блоках. Послідовний оператор – це послідовність операторів, що розташована між парою ключових слів **begin** та **end**. Послідовні оператори виконуються один за одним у порядку запису, а приймачем у них може бути змінна тільки регістрового типу. У мові Verilog введені специфічні механізми керування доступністю, які задаються формою запису оператора: оператори блокуючого і не блокуючого присвоєння.

Оператор блокуючого присвоєння забороняє виконання інших операцій до свого завершення. Це гарантує послідовне виконання операторів у блоці. Крім того, таке визначення робить результат присвоєння безпосередньо

доступним будь-якому наступному операторові в поточному програмному блоці.

Оператор неблокуючого присвоєння відображається сполученням символів <= і дозволяє виконання інших операторів ще до власного завершення. Якщо сукупність неблокуючих операторів ініційована загальною подією, то ці оператори починають виконуватися одночасно, тобто всі вони використовують значення операндів до початку виконання всієї сукупності подій. Оператори неблокуючого присвоєння надають можливість опису регістрових пристроїв із зворотними зв'язками. Розглянемо приклад, в якому використовуються оператори блокуючого та неблокуючого присвоєння.

```
module procass(out1, out2, out3, out4, clk, en, inp);
// початок опису модуля
input clk;                          // опис вхідних портів
input en;
input inp;
output out1,out2,out3,out4;     // опис вихідних портів
reg out1;                   // всі вихідні порти мають тип "регістр"
reg out2, out3, out4;
 always @ (posedge clk) // по фронту тактового сигналу
        if (en==1)          // перевіряємо, є чи на вході en сигнал лог.1
                begin               // якщо є
                        out1=1'b0;  // установлюємо рівні лог.0
                        out2=1'b0;  // на всіх вихідних портах
                        out3=1'b0;
                        out4=1'b0;
                end
        else                        // якщо на вході en сигнал лог.0
                begin
                        out1=inp;   // виконуємо блокуюче присвоєння
                        out2<=out1;// виконуємо неблокуюче присвоєння
                        out3=out1;
                        out4=out2;
                end
endmodule                           // кінець опису модуля
```

Часова діаграма, яка демонструє роботу модуля, приводиться на рис.1.19.

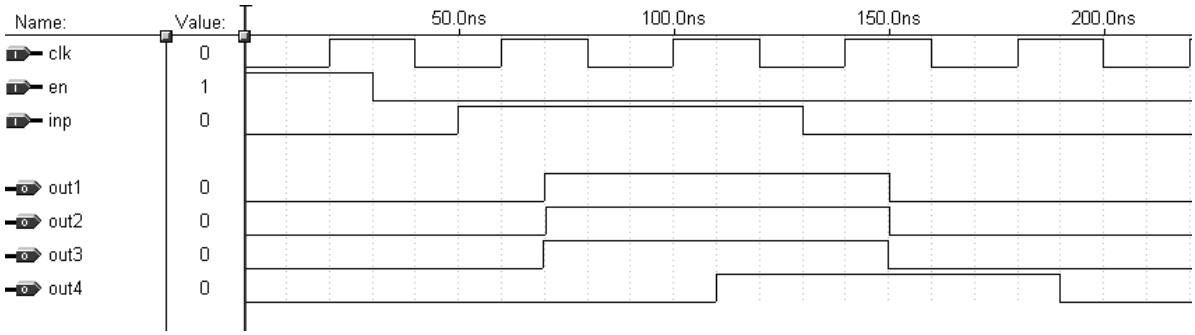

Рис.1.19. Результати моделювання роботи пристрою

По фронту тактового сигналу, на виході out1 установлюється значення сигналу на вході inp. У цей же момент часу (з урахуванням затримки) виходи out2 і out3 приймають точно такі ж значення сигналу, як і вихід out1. При цьому був використаний оператор блокуючого присвоєння. Сигнал на виході out4 установлюється в 1 тільки в наступному такті. У такий спосіб моделюється лінія затримки на час, що дорівнює рівно одному модельному інтервалу.

Блокуюче присвоєння еквівалентно присвоєнню значення змінній, а не блокуюче – послідовному сигнальному присвоєнню.

Ще один приклад, який демонструє використання операторів блокуючого і неблокуючого присвоєння приводиться нижче.

```
module nbtest(dout, din);          // початок опису модуля
output [7:0] dout;                 // опис вихідного порту
input[7:0] din;                    // опис вхідного порту
reg [7:0] dout;
reg [7:0] tmp;                     // додатковий регістр
always @(din)                      // початок процедурного блоку
      begin
            tmp=din;
            // присвоюємо додатковому регістру значення входу
            tmp<=tmp+1; // збільшення значення змінної на 1
            dout<=tmp; // установка значення на виході

      end                          // кінець процедурного блоку
endmodule                          // кінець опису модуля
```

При зміні значення сигналу на вході модуля (din) це значення за допомогою оператора блокуючого присвоєння записується в змінну tmp. Далі відбувається операція інкременту внутрішньої змінної і установка нового значення на виході. Часова діаграма, отримана в результаті моделювання, представлена на рис.1.20.

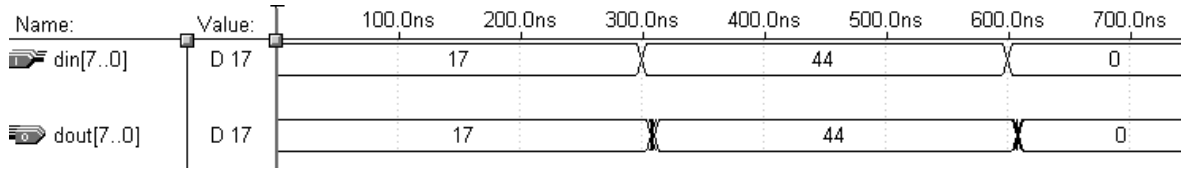

Рис.1.20. Діаграма роботи модуля

Зміна порядку запису рядків «tmp<=tmp+1;» і «dout<=tmp;» не вплине на роботу модуля. Змінимо опис модуля наступним чином:

```
…
begin
        tmp=din;
        tmp=tmp+1;
        dout=tmp;
end
…
```

Якщо замість оператора неблокуючого присвоєння «<=» записати оператор блокуючого присвоєння «=», то поводження модуля зміниться. На виході встановиться значення змінної tmp, збільшене на 1. Це демонструє часова діаграма на рис.1.21.

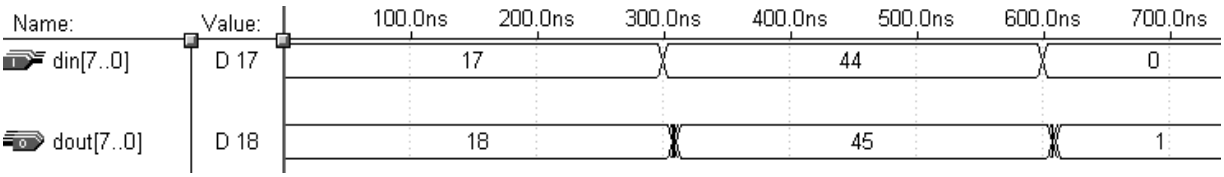

Рис.1.21. Результат використання оператора блокуючого присвоєння

Таким чином, на відміну від попереднього прикладу, операції присвоєння виконувались послідовно. Були використані значення, отримані в результаті виконання попереднього (за порядком запису) виразу.

1.11. Оператори прийняття рішень

Оператор умови (**if**) і оператор варіанту (**case**) дозволяють вибрати один з можливих шляхів виконання алгоритму в залежності від поточних умов. У процесі моделювання вони відносяться до класу послідовних (процедурних) операторів і виконуються слідом за оператором, що передує їм у програмному блоці. При синтезі пристрою відтворюються блоки, що виконують всі описані альтернативи, з яких або ініціюється в кожний момент тільки один блок, або вихідні сигнали зчитуються тільки з одного із блоків. Конструкція **if** записується в такий спосіб:

if (<умова>)
    <оператори>
else
    <оператори>

Для вибору з декількох варіантів можуть бути застосовані вкладені оператори **if**.

if (<умова>)
    <оператори>
else if (<умова>)
    <оператори>
else if (<умова>)
    <оператори>
else

Тут <умова> – будь-який вираз мови, а <оператор> – оператор або група операторів, які розташовані між **begin** та **end**. Ключове слово **else** може бути відсутнім, але якщо є вкладені **if** (як у прикладі), то **else** відноситься до найближчого **if**. Якщо одержане у виразі <умова> значення не дорівнює 0 і не є невизначеною (x або z), то виконується гілка **if**, інакше – **else**. Варто пам'ятати, що так само, як у мові програмування C, операція порівняння записується як == (два знаки «=»), на відміну від операції присвоєння = (один знак). Але операції порівняння при невизначених операндах повертають невизначене значення (x). Тому в моделюванні (не приймається засобами синтезу) можуть використовуватися оператори === (три знаки «=») і !==. Ці операції дозволяють виконати літеральне порівняння певних бітів у виразі. Ще раз звертаємо увагу на те, що вираз <умова> не є виразом якого-небудь спеціального типу (булевого), а є будь-яким виразом, що може бути приведене до типу **integer**. Тут простежується аналогія з мовою C, єдина відмінність від якої полягає в тому, що в Verilog тип integer може приймати невизначені значення (x або z). У цьому випадку виконується гілка **else**. Оператор **if** можна реалізувати за допомогою мультиплексора – якщо на керуючому вході логічний 0, то на виході буде сигнал x_0. У противному випадку на виході буде сигнал x_2. В деяких випадках при схемній реалізації оператора **if** виходить комбінаційна схема. Наступний приклад демонструє використання умовного оператора **if** для опису регістра з входом дозволу en та асинхронним сбросом reset, в який записуються дані по фронту тактового сигналу clk.

```
module reg_if(dout, clk, en, reset, din);      // заголовок модуля
output [3:0] dout;                             // вихідний 4-розрядний порт
input clk, en, reset;                          // вхідні сигнали
input [3:0] din;                               // вхід даних
reg [3:0] dout;                                // вихід має регістровий тип
always @(posedge clk or posedge reset)
                                               // по фронту одного з сигналів
if (reset) din=0;                              // якщо сброс – на виході 0
else if (en) dout=din; // якщо встановлено дозвіл, на виході вхідні дані
else dout=0;                  //при невиконанні попередніх умов, на виході 0
endmodule
```

Діаграма на рис.1.22 демонструє роботу модуля.

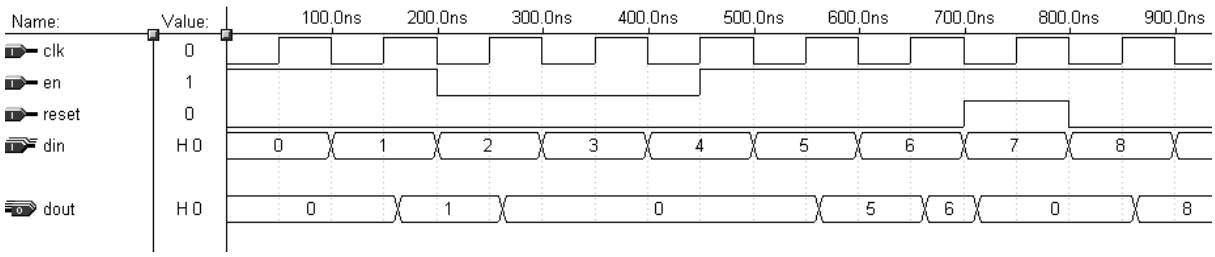

Рис.1.22. Діаграма роботи паралельного регістра

Для вибору з декількох варіантів також застосовується оператор **case**. Наприклад, дана конструкція реалізує дешифратор, подібний до К155ИД3:

```
module decoder(result, din);    // заголовок модуля
output [9:0] result;            // вихідний 10-розрядний порт
input [3:0] din;                // вхідний 4-розрядний порт
reg [9:0] result;               // вихід має регістровий тип
      always @(din)             // процедурний блок
      case (din)                // вибираємо одну з альтернатив
                                // в залежності від вхідного коду
            4'd0: result = 10'b0111111111;
            4'd1: result = 10'b1011111111;
            4'd2: result = 10'b1101111111;
            4'd3: result = 10'b1110111111;
            4'd4: result = 10'b1111011111;
            4'd5: result = 10'b1111101111;
            4'd6: result = 10'b1111110111;
            4'd7: result = 10'b1111111011;
            4'd8: result = 10'b1111111101;
```

```
            4'd9: result = 10'b1111111110;
            default: result = 0;
        endcase                 // кінець конструкції вибору
endmodule                       // кінець опису модуля
```

Оператор **case** в Verilog, на відміну від оператора switch мови C, гарантує виконання однієї з альтернатив. У випадку, якщо жодна з умов не збігається, виконується гілка **default**. Оператор **case** часто використовується в синтезованому коді для опису дешифраторів і мультиплексорів. При цьому в несинтезованих модулях (а в деяких засобах синтезу і у синтезованих) в конструкції **case** можуть використовуватися літерали з невизначеними значеннями. Для моделювання використовуються оператори **casez** та **casex**, які особливим образом обробляють невизначені стани. Синтаксис **casez** і **casex** подібний до синтаксису **case**. При цьому додається символ «?», який використовується у двійковому записі літералу для того, щоб замаскувати біти, які не повинні впливати на прийняте рішення.

Шаблон оператора **case** представлений нижче.

```
case (__expression)      // початок оператора
    __constant_value : // значення
    begin
     __statement;       // якщо __expression == __constant_value , тоді
    end                 // виконується цей оператор
    __constant_value :
    begin
         __statement;
    end
    default :           // якщо в тілі не знайдено необхідної величини
    begin               // виконується цей оператор
         __statement;
    end
endcase                 // кінець оператора
```

Якщо навпроти однієї з альтернатив міститься лише один оператор, то в записі конструкції **begin** – **end** немає необхідності.

Версії оператора варіанту із ключовими словами **casex** та **casez** дозволяють задавати сукупність альтернатив, які виконуються при збігу хоча б деяких розрядів, а значення в інших розряди при цьому не враховується. В операторі **casez** не враховуються при перевірці на співпадання розряди, що мають значення z як у записі одного з варіантів , так і у результаті обчислення ключового виразу. Так, коди 7'b00zz101 і 7'bzz1010z в операторі **casez** збігаються. Оператор із ключовим словом

casex розглядає розряди, установлені в стан високого імпедансу або невизначений стан, як несуттєві. При цьому символи невизначеності можуть отримуватися як в результаті обчислення виразу, так і у записі варіанту, причому в записі варіанту замість символів x та z можна використати знак питання.

Шаблон оператора **casex** представлений нижче.

```
casex (__expression)          // початок оператора
    __constant_value :        // значення
    begin
        __statement;  // якщо __expression == __constant_value, то
    end               // виконується цей оператор
    __constant_value :
    begin
        __statement;
    end
    default :     // якщо в тілі не міститься необхідної величини
    begin         // виконується цей оператор
        __statement;
    end
endcase           // кінець оператора
```

Шаблон оператора **casez** має такий самий вигляд, тому немає необхідності приводити його. Оператор **case** підтримується при синтезі.

1.12. Параметри

При описі цифрових схем виникає необхідність у константах, що визначають які-небудь фіксовані значення. Ці константи можуть описувати затримки в системі, розрядність шин або будь-які інші характеристики, що не змінюється під час симулювання моделі і відомі на момент компіляції. Але в той же час при використанні моделі того самого модуля в різних технологічних умовах або включенні його різними способами у модулі вищого рівня дані константи повинні мати можливість змінюватися. В Verilog для оголошення константних значень використовуються параметри. Тип параметра збігається з типом константного виразу.

```
parameter word = 32;          // word = 32
input [word-1:0] addr;        // 32-розрядна шина
```

Незважаючи на те, що значення параметрів всередині модуля, в якому вони визначені, є константами, при включенні в ієрархічні проекти їм може бути присвоєне значення, відмінне від початкового. Змінивши значення параметра, зміниться розрядність ланцюга, регістра або пам'яті.

Таким чином модуль, що розробляється, може оперувати з даними будь-якої розрядності.

Розглянемо простий приклад. Необхідно розробити дільник частоти. Для того, щоб мати можливість змінювати коефіцієнт перерахунку, введемо в опис модуля параметр. Текст опису модуля приводиться нижче.

```
module param(clkout, clkin, en); // заголовок модуля
output clkout;                   // опис вихідних портів
input clkin, en;                 // опис вхідних портів
reg [22:0] temp;                 // опис додаткового регістра
reg clkout;                      // вихідний сигнал має тип reg
parameter div = 4000000;         // значення коефіцієнта перерахунку
always @ (posedge clkin)  // по кожному фронту тактового сигналу
      begin
      if (en==1)                 // якщо на вході  дозволу сигнал лог.1
       begin
            temp = 0;            // обнуління додаткового регістра
            clkout = 0;          // установлюємо лог.0 на виході
       end                       // якщо на вході  дозволу сигнал лог.0
      else temp = temp+1;// інкремент значення додаткового регістра
      if (temp == div)
       begin
           clkout = ~clkout;     // після кожних 4000000 тактів
           temp=0; // на виході сигнал змінюється на протилежний
       end                       // відбувається обнуління лічильника
      end
endmodule                       // кінець модуля
```

В модулі введена додаткова змінна temp, призначення якої полягає в підрахунку кількості сигналів, що надійшли на вхід clkin. Модуль має інверсний вхід дозволу роботи en, високий рівень сигналу на якому забороняє роботу модуля. В процесі роботи модуля значення змінної temp порівнюється з параметром div, який визначає коефіцієнт ділення частоти. Якщо ці значення рівні, відбувається зміна сигналу на виході clkout на протилежний.

Подібні дільники використовуються в багатьох пристроях. Наприклад, для створення годинника потрібний тактовий сигнал з періодом 1с. Загалом використовується кварцовий генератор частотою кілька десятків кілогерц. Знаючи частоту кварцового резонатора, можна задати необхідний коефіцієнт ділення і одержати на виході модуля імпульси заданого періоду. На рис.1.23 приводиться часова діаграма роботи такого дільника з коефіцієнтом ділення, що дорівнює 11.

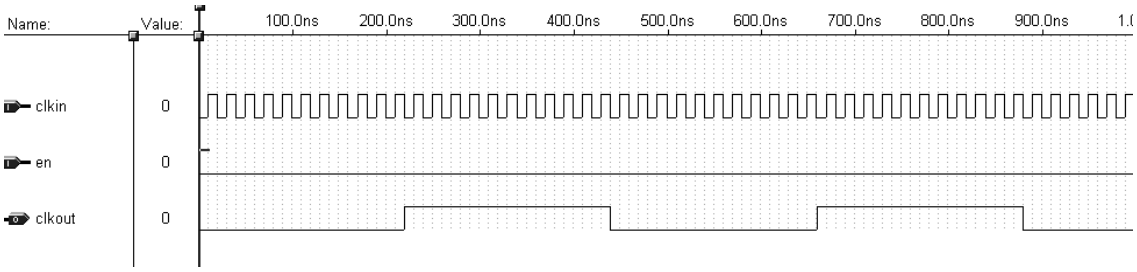

Рис.1.23. Часова діаграма роботи дільника частоти

Застосування такого роду конструкцій дозволяє уникнути використання лічильників (в графічному редакторі), їх нарощування та ”обв'язку” додатковими елементами для одержання необхідного коефіцієнту ділення. При розробці, наприклад, універсального асинхронного приймача-передавача (UART) взагалі не обійтися без дільника частоти, оскільки потрібно забезпечити кілька швидкостей для обміну даними. Запропонований модуль дільника частоти може бути використаний разом з генератором імпульсів, який розглядався раніше.

Спробуємо визначити максимальну частоту вхідного тактового сигналу, при якому схема буде виконувати свої функції. Для цього скористаємося Аналізатором часових затримок. На рис.1.24. приводяться отримані результати.

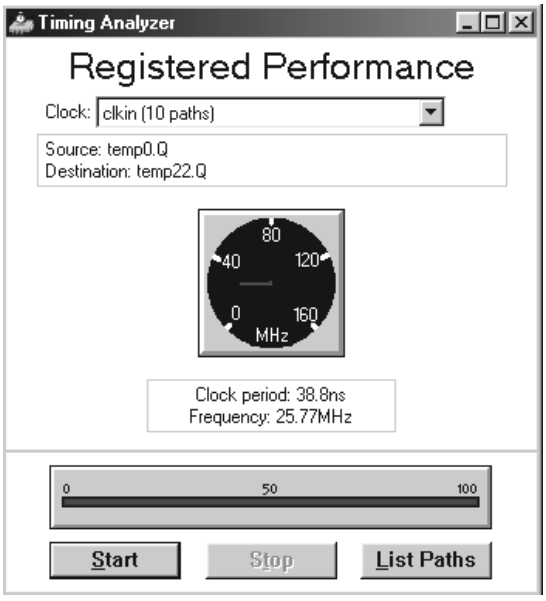

Рис.1.24. Результати аналізу часових затримок дільника частоти

Легко побачити, що частота, при якій пристрій буде зберігати свою працездатність, становить 25,77 МГц. Цей результат отриманий для мікросхеми EPF8282ALC84-2 сімейства FLEX8000.

При використанні бібліотеки шаблонів Verilog для опису параметру, буде запропонована наступна конструкція:

parameter __parameter_name = _parameter_value;

Розробнику потрібно змінити ідентифікатор параметра і вказати його значення. Нагадаємо, для того, щоб скористатися бібліотекою шаблонами Verilog-конструкцій, необхідно в текстовому редакторі викликати на екран контекстне меню шляхом натискання правої кнопки миші, а потім вибрати в цьому меню команду Verilog Template...

Наряду з механізмом параметрів в Verilog існує механізм (препроцесор), який дозволяє проводити попередню обробку тексту до того, як він буде оброблений засобом синтезу. Даний механізм забезпечує умовну компіляцію, виконання макропідстановок. Найбільш часто використовуються директиви **`define**, **`include**, **`ifdef**, **`else**, **`endif**. Для включення тексту з одного файлу в інший фикористовується директива **`include** „назва файлу”. Наступний приклад демонструє використання **`define**.

```
`define pc @(posedge clk)        // використання директиви
module testdef(dout, clk, din); // заголовок модуля
output dout;                     // вихідний порт
input clk, din;                  // вхідні сигнали
reg dout;                        // вихід має регістровий тип
always pc                        // підстановка значення pc
        dout=din;                // передача вхідних даних на вихід
endmodule                        // кінець опису модуля
```

Наведений опис представляє з себе модуль D-тригера, який тактується по фронту сигналу clk.

1.13. Оператори повторення

У мові Verilog визначені чотири форми операторів повторення: **forever**, **repeat**, **while**, **for**. У всіх операторах повторення вкладений оператор може бути простим або складовим, тобто містити сукупність послідовних операторів, які розташовані між ключовими словами **begin** та **end**. По синтаксису і по інтерпретації записаних дій, оператори повторення практично не відрізняються від таких самих операторів мови програмування С, за винятком оператора **forever**. Цей оператор повторюється нескінченно. Він починається з ключового слова **forever**, за яким розташований оператор, що буде постійно повторюватися. Це робить його подібним до блоку **always**. Якщо операторів декілька, вони розташовуються в блоці **begin** – **end**. В циклі forever обов’язково повинен

бути присутній часовий контроль. В протилежному випадку симулятор постійно буде виконувати тільки цей оператор, забороняючи моделювання інших частин модуля. Найчастіше цикл **forever** використовується для генерування тактового сигналу.

Виконання циклу **for** контролюється трьома параметрами, які розташовуються в дужках після ключового слова **for**: початкове значення, умова для переривання циклу та алгоритм зміни лічильника циклу між ітераціями. Розглянемо приклад використання циклу **for**.

```
module forr (d, q);          // початок опису модуля
input  [2:0] d;       // декларація вхідних портів та їх розрядності
output [1:0] q;       // декларація вихідних портів та  їх розрядності
integer num_bits;            // оголошення додаткових змінних,
                             // необхідних для роботи алгоритму
always @ (d)                 // початок блоку  контролю подій
    begin: block
    integer i;               // оголошення змінної цілого типу
    num_bits = 0;
    for (i = 0; i < 3; i = i + 1)    // початок оператора повторення
    if (d[i] == 1)
    // перевірка на рівність 1 кожного біта вхідного масиву
    num_bits = num_bits + 1;   // інкремент лічильника, якщо умова
                                        // виконується
    end                                 // block
    assign q = num_bits;
    // установка на виході підрахованого значення бітів
endmodule                    // кінець опису модуля
```

Ця програма описує алгоритм підрахунку числа одиниць у вхідному слові. Вхідний масив d використовується як змінна в конструкції **always** для контролю появи події. Таким чином, зміна цієї змінної приводить до виконання блоку процедурного призначення. Якщо біт у вхідному масиві даних дорівнює 1, збільшується на 1 значення лічильника бітів, оскільки виконується умова. Цей алгоритм виконується доти, поки значення змінної „i” менше 3. Після цього відбувається установка значення кількості одиниць у вхідному слові на вихідному порту q.

В циклі **while** не обмежується кількість виконуваних ітерацій. Оператори в циклі виконуються доти, доки логічний вираз приймає значення „істинно”. В протилежному випадку виконується наступний за циклом оператор. Синтаксис циклу **while** має вигляд:

while(«лог.вираз») «оператор»;

Результат логічного виразу переобчислюється щоразу при старті циклу. Наступний приклад, який створений в САПР Quartus, представляє собою модуль, який виконує перестановку бітів місцями, тобто перший біт становиться восьмим (для 8-розрядних слів), другий – сьомим і т.д. При спробі компіляції цього прикладу в САПР MAX+plus II компілятор видасть помилку, оскільки цикл **while** не підтримується, хоча і є у шаблонах Verilog.

```
module reverse(din, dout);          // заголовок модуля
parameter n=8;                      // параметр, що визначає розрядність
даних
output [n-1:0] dout;                // вихідний порт
input [n-1:0] din;                  // вхідний порт
reg [n-1:0] temp;                   // додаткова змінна регістрового типу
integer i;                          // змінна для використання в циклі
always @(din)                       // процедурний блок
begin
        i=0;                        // установка значення змінної в 0
        while (i<n-1)       // початок циклу; якщо умова виконується
          begin                     // виконується тіло циклу
                temp[i]=din[n-i-1];         // перестановка бітів
                i=i+1;              // збільшення на 1 значення змінної
          end                       // кінець тіла циклу
end                                 // кінець процедурного блоку
assign dout=temp;                   // установка значення на виході
endmodule                           // кінець опису модуля
```

В тілі циклу **while** відбувається копіювання бітів в додаткову змінну temp в зворотньому порядку. Разом з цим збільшується значення змінної „i”. Цикл продовжує виконуватися доти, поки результатом операції порівняння i<n є значення „істинно”. Після виконання циклу значення змінної temp установлюється на вихідному порту dout. В модулі використаний параметр n, змінивши значення якого можна змінити розрядність даних. Нове значення на виході переобчислюється щоразу при зміні значення на вході. Часова діаграма на рис.1.25 демонструє роботу модуля.

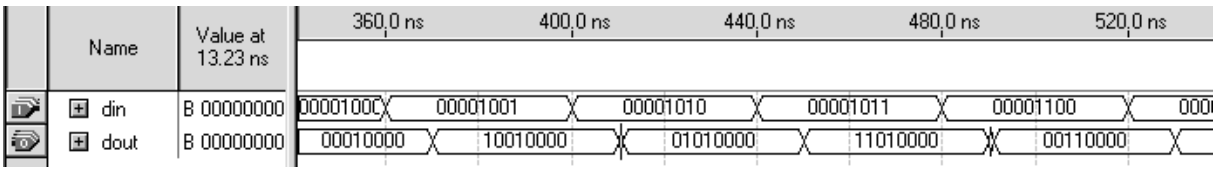

Рис.1.25. Діаграма роботи модуля reverse

Ще один приклад використання циклу **for** приводиться в поведінковому описі n-розрядного суматора з послідовним переносом. Також в цьому описі використовується параметр n, значення якого дорівнює 10. Досить змінити це число і користувач одержить суматор, що оперує числами необхідної розрядності.

```
module adder_n(a,b,cin,sum,cout);    // заголовок модуля суматора
parameter n=10;                      // визначення параметру
input [n:1] a,b;                     // вхідні шини даних розрядністю n
input cin;                           // вхід переносу
output [n:1] sum;                    // вихідний n-розрядний порт
output cout;                         // вихід переносу
integer i;                           // оголошення змінної
reg [n:1] vsum;                      // проміжний результат додавання
reg carry;                           // для зберігання значення переносу
always @(a or b or cin) // відслідковується зміна вхідних сигналів
begin
        carry=cin;
        for(i=1;i<=n;i=i+1)          // в циклі виконуємо операцію додавання
        begin
        vsum[i]=(a[i]^b[i])^carry;       // значення і-го розряду суми
        carry=(a[i] & b[i]) | (carry & (a[i] | b[i])); // значення переносу
        end
end
        assign sum=vsum;  // на виході встановлюється значення суми
        assign cout=carry;           // та переносу
endmodule
```

Після перевірки роботи модуля суматора була отримана часова діаграма, представлена на рис.1.26.

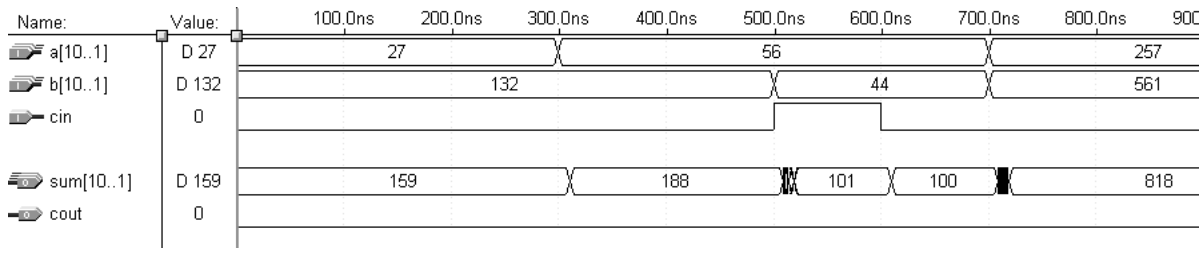

Рис.1.26. Діаграма роботи суматора з послідовним переносом

Розкид затримок сигналів від входу до виходу лежить у межах від 10нс до 38нс. Із цих причин виникає нестабільність значення на виході суматора при зміні вхідних сигналів, що легко побачити на часовій

діаграмі. Зі збільшенням розрядності суматора значення часу затримки сигналів також буде зростати.

Цикл **repeat** використовується для повторення виконання деякого виразу визначену кількість разів. Число повторень вказується в дужках після ключового слова **repeat** і повинно бути константою або змінною чи сигналом. В останніх двох випадках величина розраховується один раз, на початку виконання циклу. Якщо в результаті розрахунку з'являється невизначений результат, то він обертається до 0 і цикл не виконується жодного разу. Кількість ітерацій не може бути змінена протягом виконання циклу. Цикл **repeat** дуже зручний у випадках, коли кількість ітерацій відома, наприклад при ініціалізації вектору або пам'яті. Наступний приклад представляє собою модуль дешифратора. При описанні був використаний цикл **repeat**. Проект створений в САПР Quartus.

```
module decoder(out, data, en); // заголовок модуля
output [15:0] out;          // вихідний порт
input en;                   // вхід дозволу
input [3:0] data;           // вхід даних
reg [15:0] out;             // вихід має регістровий тип
reg [3:0] temp;             // додаткова змінна
integer i;                  // змінна для задання індексу
always @(data or en)        // процедурний блок
begin
i=16;                       // присвоєння змінній і значення 16
temp=data;                  // запис у додаткову змінну вхідного значення
if(!en) out=0;              // якщо en=0, на виході встановлено 0
else                        // інакше
        repeat(15)          // повторюємо цикл 15 разів
        begin
        if(temp==i-1)       // якщо виконується умова
                out[i-1]=1; // встановлюємо в 1 одну з вихідних ліній
        else                // в іншому випадку
                out[i-1]=0; // на виході встановлюється 0
                i=i-1;      // зменшення значення індексу
        end                 // repeat
end                         // always
endmodule                   // кінець опису модуля
```

В циклі **repeat**, який виконується 15 разів, відбувається порівняння вхідного коду з лічильником ітерацій temp. Якщо ці значення рівні, на вихідному порту з'являється лог.1 на тому біті, номер якого дорівнює вхідному коду.

1.14. Задачі та функції

Як і у традиційних мовах програмування, використання підпрограм дозволяє забезпечити структуризацію, отже, краще розуміння програм, а також заощаджує час проектувальника, дозволяючи однократно описувати однотипні фрагменти пристроїв і алгоритми їх функціонування. В Verilog розрізняють два типи підпрограм, що відрізняються по способу повернення результату: задачі (**task**) і функції (**function**). Задача повертає результат через зіставлення формальних і дійсних об'єктів, а функція повертає одне значення через назву підпрограми на місці її виклику у виразі. Набір припустимих операцій такий самий, як і в програмному модулі. Обмеження зводяться до того, що в декларації функції не може бути вихідних об'єктів і повинен бути хоч один вхідний формальний об'єкт.

Відмінності задачі від функції:

- Задача (**task**) може містити опції контролю подій, тобто для задачі можуть передбачатися умови ініціалізації і передбачається її виконання впродовж деякого часу. Функція виконується з нульовою затримкою. У функціях заборонено використання конструкції контролю подій (**always** @) або часове керування (при моделюванні).
- У задачах допускається виклик інших задач і функцій. Функція може викликати іншу функцію, але не задачу.
- У розділі операторів повинне міститися присвоєння значення імені функції.

При виклику функції необхідно записати її назву, а в дужках фактичні об'єкти. Причому порядок запису фактичних об'єктів повинен строго відповідати порядку появи імен формальних об'єктів в декларації підпрограми. При реалізації в апаратурі кожному виклику функції зіставляється регістр для збереження результату, стан якого може змінитися в момент виконання відповідного оператора. Синтаксис опису функції має вигляд:

```
function [__range_msb:__range_lsb] __function_name;
        // Input_port Declaration - опис вхідних портів
        // Reg Declaration - опис регістрів
         // Wire Declaration - опис ланцюгів
         begin
        __statements;       // алгоритм роботи функції
        end
endfunction                 // кінець опису функції
```

Опис функції розміщений між ключовими словами **function** та **endfunction**. Ідентифікатор функції повинен бути унікальним. Зверніть

увагу на відсутність опису вихідних портів. Як говорилося вище, значення повертається з функції через її ідентифікатор. Розрядність результату, що повертається, вказується у визначенні функції в квадратних дужках, між словами **function** та назвою функції. Розглянемо невеликий приклад використання функцій.

```
module function1(outf, A, B); // початок модуля
output outf;                  // декларація вихідних портів
input A, B;                   // декларація вхідних портів
wire [7:0] A;                 // вказівка типу вхідних сигналів, їх
розрядності
wire [7:0] B;
reg [7:0] outf;               // вказівка типу вихідного сигналу та
розрядності
function [7:0] min;           // початок опису функції
      input [7:0] C;          // оголошення розрядності переданих
                              // фактичних об'єктів та їх  ідентифікаторів
      input [7:0] D;
            begin                       // опис алгоритму
                  if(C>D) min=D;
                  if(C<D) min=C;
                  if(C==D) min=1;
             end
 endfunction                            // кінець опису функції
      always @(A or B)                  // оператор контролю подій
            begin
                  outf=min(A,B);        // на виході установлюється
                                        значення
                                        // мінімальне із вхідних величин
            end
 endmodule                  // кінець опису модуля
```

При описі функції не вказувалася кількість і розрядність переданих формальних об'єктів. В цьому прикладі при виклику функції в її тіло передаються значення A та B. В тілі функції цим значенням будуть відповідати C і D (в порядку опису в тілі функції). Розрядність регістра в який записується значення, що повертається функцією, повинна бути такою самою, як розрядність результату. На вихідному порту описаного модуля встановлюється мінімальне зі значень, які присутні на входах. Якщо значення кодів на вхідних портах однакові, тоді на виході встановлюється значення 1. Часова діаграма, що демонструє роботу модуля, приводиться на рис.1.27.

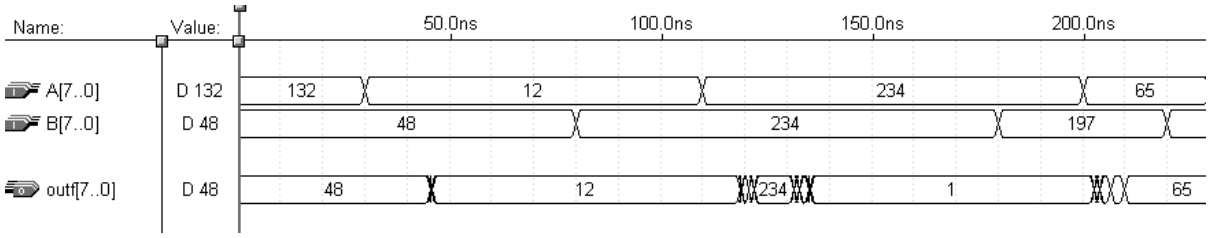

Рис.1.27. Часова діаграма роботи модуля

Моделювання синтезованої схеми показало, що її поводження відповідає алгоритмічному опису. В моменти зміни вхідних сигналів, на виході виникають перехідні процеси, і мінімальне зі значень установлюється через деякий проміжок часу. Це обумовлено різними величинами затримок передачі сигналу від входу до виходу для різних виводів мікросхеми. Максимальна затримка за результатами моделювання становить близько 20 нс. Для усунення цього ефекту можна додатково ввести в схему паралельний регістр і синхросигнал. Розглянемо ще один приклад. В описанні наступного модуля є функція Shift, яка виконує зсув 16-розрядного слова data на один розряд ліворуч (0) або праворуч (1), в залежності від значення сигналу direction, який є другим параметром, що передається в функцію. На виході Rdata з'явиться вхідний код, зсунутий на один розряд праворуч, а на виході Ldata – зсунутий на один розряд ліворуч.

```
module function2(data, Ldata, Rdata);        // заголовок модуля
parameter WIDTH=16;                          // розрядність вхідного порту
input [WIDTH-1:0] data;                      // вхідний порт
output [WIDTH-1:0] Ldata, Rdata;             // вихідні порти
reg [WIDTH-1:0] Ldata, Rdata;                // виходи мають регістровий тип
function [WIDTH-1:0] Shift;                  // заголовок функції
      input [WIDTH-1:0] din;                 // перший параметр – вхідний код
      input direction;                 // другий параметр – напрям зсуву
if(direction==0) Shift=din<<1;  // в залежності від умови виконується
else if(direction==1) Shift=din>>1;    // зсув ліворуч або праворуч
endfunction                                  // кінець опису функії
always @(data)                               // щоразу при зміні вхідного коду
begin                                        // виконується послідовний блок
      Ldata=Shift(data,0);                   // виклик функції
      Rdata=Shift(data,1);
end                                          // always
endmodule                                    // кінець опису модуля
```

На відміну від функцій, в задачі може бути декілька вихідних портів. Опис задачі починається з ключового слова **task**, за яким слідує її назва.

Опис задачі закінчується ключовим словом **endtask**. Між цими словами розташовується описання портів (інтерфейсу) і тіла задачі. Повернення результату виконується не через ідентифікатор задачі, а через один або декілька портів, які мають тип output або **inout**. Для демонстрації прикладу використання задач і синтаксичних особливостей зазначеної конструкції була використана система проектування Quartus. Опис модуля має вигляд:

```
module task1(out1, in1, in2);          // початок модуля
output out1;                           // оголошення вихідного порту
input in1;                             // оголошення входів
input in2;
      task xor_task;                   // опис задачі
            output out_t;              // перелік портів
            input in_t1;
            input in_t2;
            out_t=in_t1^in_t2;         // операція ВИКЛ. АБО
     endtask                           // кінець опису задачі
always @ (in1 or in2)          // при зміні одного із вхідних сигналів
      begin
            xor_task(out1,in1,in2);    // виконуємо задачу
      end
endmodule                              // кінець опису модуля
```

Цей простий приклад показує використання в задачі (**task**) операції ВИКЛ.АБО. При зміні будь-якого із сигналів (in1 або in2) відбувається їх об'єднання по ВИКЛ.АБО, і результат цієї операції передається на вихід модуля. Зверніть увагу на те, що при оголошенні задачі (**task**) в блоці **always**, послідовність переданих об'єктів у задачу відповідає послідовності їх оголошення при описі самої задачі. Використання задач можливо тільки в конструкціях контролю подій. Часова діаграма роботи модуля приводиться на рис.1.28.

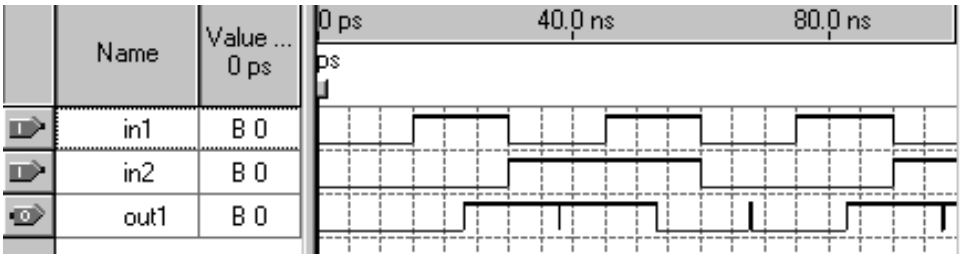

Рис.1.28. Часова діаграма роботи модуля

Аналіз діаграми показує, що модуль поводить себе як логічний елемент ВИКЛ.АБО. На виході встановлюється (з деякою затримкою)

сигнал високого рівня коли лише на одному з входів присутній сигнал високого рівня. В наступному прикладі демонструється використання задачі (**task**) для підрахунку кількості нулів у вхідному слові.

```
module tasks2(ones, data);        // заголовок модуля
output [3:0] ones;                // вихідний порт
input [7:0] data;                 // вхідний порт
task OnesCounter;                 // початок опису задачі
input [7:0] Din;                  // перший параметр – вхідний
output [3:0] Dout;                // другий параметр – результат
integer i;                        // додаткова змінна
reg [3:0] count;                  // змінна для підрахунку нулів
begin                             // тіло задачі
      count=0;                    // обнуління змінної
for(i=0;i<8;i=i+1) // в циклі перевіряється значення кожного біту
      if (Din[i]==0) count=count+1;  // і при виконанні умови
                                  // інкрементується лічильник нулів
      Dout=count;                 // передача значення на вихід
end                               // кінець опису тіла задачі
endtask                           // кінець опису задачі
always @(data)                    // контроль зміни вхідного коду
      OnesCounter(data,ones); // виклик задачі
endmodule                         // кінець опису модуля
```

Передача аргументів в задачу виконується за допомогою круглих дужок, в яких через кому записані всі сигнали в тому порядку, в якому вони були розташовані при описанні задачі, в розділі опису інтерфейсу. Якщо в якості аргументів виступають вектори, то їх розрядність в задачі вказується так само, як це робиться при описанні модуля. На рис.1.29 представлений результат моделювання модуля.

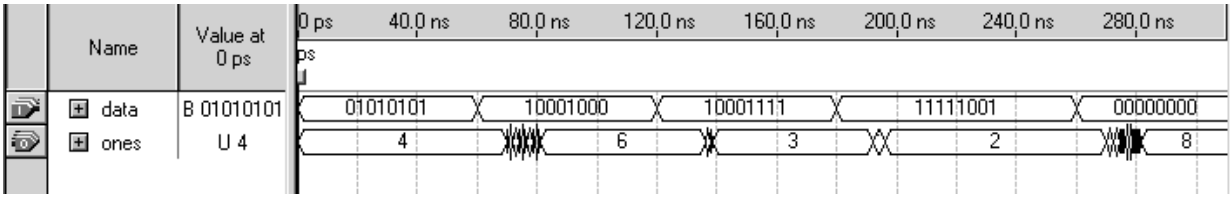

Рис.1.29. Діаграма роботи модуля підрахунку нулів

Загальні для сценаріїв та функцій властивості полягають в наступному:

- задачі та функції можуть мати локальні змінні та регістри;
- ні задачі, ні функції не можуть містити локальних ланцюгів (**wire**);

- задачі та функції можуть містити лише поведінкові оператори і викликатися з інших поведінкових операторів (**always**, **initial**). Функції також можуть бути викликані з оператору безперервного присвоєння **assign**.

1.15. Використання двонапрямлених портів

При розробці складних систем часто виникає необхідність роботи різних вузлів на загальну шину. При роботі якого-небудь пристрою на одну шину необхідно забезпечити високий імпеданс з боку виходів інших вузлів системи. Крім того, для зменшення числа з'єднань між мікросхемами також зручно використовувати двонапрямлену шину. Розглянемо невеликий приклад, що демонструє принцип роботи із двонапрямленими виводами. Опис модуля має вигляд:

```
module bidirec(oe, clk, inp, outp, bidir);      // заголовок модуля
input oe;                                       // вхід дозволу
input clk;                                      // вхід тактового сигналу
input [7:0] inp;                                // вхідний порт
output [7:0] outp;                              // вихідний порт
inout [7:0] bidir;                              // двонапрямлена шина
reg [7:0] a;                          // додаткові змінні регістрового типу
reg [7:0] b;
        assign bidir = oe ? a : 8'bZ; // перевід шини у відповідний стан
        assign outp  = b;
        always @ (posedge clk)        // по фронту тактового сигналу
        begin
                b <= bidir;           // записуємо значення на шині
                a <= inp;             // записуємо дані з вхідного порту
        end
endmodule                             // кінець опису модуля
```

При моделюванні двонапрямленої шини в Сигнальному редакторі не можна одночасно додавати двонапрямлений вивід, який налаштовано на вхід і на вихід. Інакше це призведе до некоректних результатів моделювання. В такому випадку потрібно двічі промоделювати схему, спочатку із двонапрямленою шиною, налаштованою на вихід, а потім налаштованою на вхід (порядок моделювання не має значення). В цьому випадку отримані результати будуть відповідати дійсному поводженню системи. На рис.1.30 приводиться часова діаграма роботи модуля при режимі роботи двонапрямленої шини на вихід.

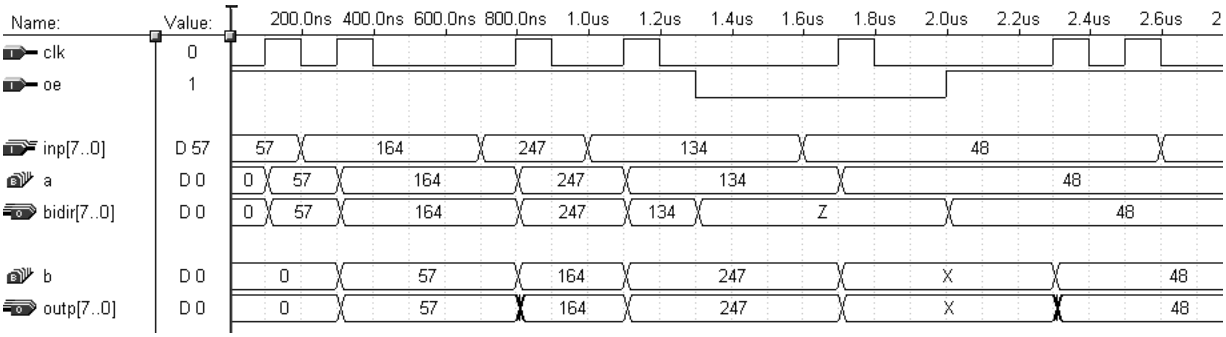

Рис.1.30. Часова діаграма модуля при роботі шини на вихід

При високому рівні сигналу на вході oe (output enable), по фронту тактового сигналу дані, які присутні на вхідній шині inp, записуються у внутрішню змінну „a” регістрового типу, і разом із цим встановлюються на виході bidir. По приходу наступного фронту тактового сигналу дані на шині bidir записуються у внутрішню змінну b і встановлюються на вихідній шині outp. У цей же час оновлюється значення змінної “a” та стан виходу bidir.

З появою сигналу низького рівня на вході oe шина bidir переходить у стан високого імпедансу. По фронту тактового сигналу в змінну “b” записується невизначене значення. Невідоме значення встановлюється на виході outp. Після синтезу цього модуля безпосередньо в мікросхему, значення змінної “b” і виходу outp будуть цілком визначеними, а саме значенню на шині bidir. Це демонструє часова діаграма роботи модуля при настроюванні двонапрямленої шини на вхід (рис.1.31).

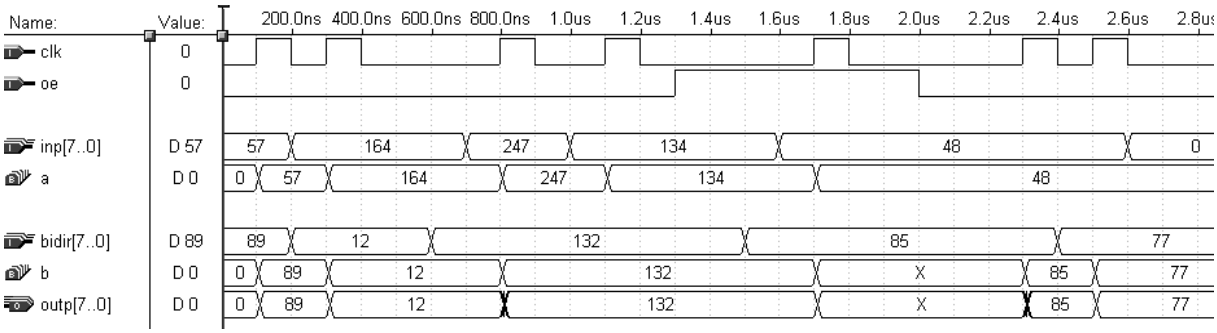

Рис.1.31. Часова діаграма модуля при роботі шини на вхід

При низькому рівні сигналу на вході oe шина bidir працює як вхідна. По фронту тактового сигналу дані на цій шині записуються у внутрішню змінну “b” і встановлюються на виході outp. При високому рівні сигналу на вході oe, шина працює як вихідна і приймає значення змінної “a” (рис.1.31).

1.16. Структурний опис проекту

Структурне представлення проектів забезпечується можливістю використання в проекті інших модулів, причому використовуються традиційні для алгоритмічних мов конструкції, аналогічні виклику відповідних підпрограм. Ці модулі з'єднані між собою і можуть обмінюватися сигналами. Більшість сучасних мов проектування апаратури підтримують можливість опису проекту у вигляді сукупності заздалегідь описаних компонентів і зв'язків між ними, і мова Verilog не є виключенням. Текст програми може містити довільний набір проектних модулів, кожен з яких представлений назвою, оголошенням портів і тілом модуля. У тілі модуля паралельні оператори і декларації розташовуються в довільному порядку, хоча будь-який об'єкт декларується раніше його використання в операторах. Вся ієрархія представлена головним проектним модулем, що називають вершиною проекту, і сукупності додаткових проектних модулів. Вершина проекту містить оператори входження компонентів, тобто представляє собою структурний опис. Додатковий проектний модуль може бути, в свою чергу, вершиною наступної ієрархії. В загальному випадку, проектні модулі незалежні в тому розумінні, що кожен з них може бути використаний самостійно в різних конструкціях і бути вершиною проекту. Проектні модулі, що відносяться до одного проекту, можуть перебувати в одному файлі або бути представлені декількома файлами. Входження проектного модуля в проект вищого ієрархічного рівня відображається оператором входження. При описі входження модуля в ієрархічний проект використовуються дві назви: ім'я входження та ім'я модуля. Кілька входжень можуть бути представлені однотипними підсхемами, тобто посилатися на однаковий програмний модуль, але кожен екземпляр має власне ім'я.

Оператор входження може розглядатися як виклик підпрограм із проблемно-орієнтованим інтерфейсом виклику. Відмінність полягає в тому, що внутрішні змінні модуля, що підключається, визначені як статичні, і зберігають значення між ініціалізаціями. Оператор входження модуля обов'язково паралельний. Це означає, що перетворення, які визначені в операторній частини модуля, що підключається, виконуються щоразу, коли змінюються фактичні об'єкти оператора входження. Порядок запису фактичних значень параметрів строго відповідає порядку оголошення формальних імен параметрів в описі модуля, що підключається. Розглянемо приклад.

```
module compinst (data, clock, clearn, presetn, a, b, c, gn, d, q_out, y, wn);
input  data, clock, clearn, presetn, a, b, c, gn;  // перелік вхідних сигналів
input  [7:0] d;                                    // вихідний 8-розрядний порт
output q_out, y, wn;                               // вихідні сигнали
```

```
dff dff1 (.d (data), .q (q_out), .clk (clock), .clrn (clearn), .prn (presetn));
//D-тригер
\74151b mux (c, b, a, d, gn, y, wn);    // підключення мультиплексора
endmodule
```

В наведеному прикладі змінні в описі модуля, з'єднуються з кожним з портів, вказаних у прототипі функції 74151b. Крім того, використаний D-тригер. Слід зазначити, що назви портів і назви функцій, що збігаються із ключовими словами мови Verilog, не можуть бути використаними. Для використання ідентифікаторів, схожих на ключові слова мови, можна використати префікс (наприклад a_). Логічні функції або порти, що починаються цифрою, повинні мати префікс слеш (\). Це продемонстровано в наведеному прикладі. Прототип функції 74151b можна знайти в довідковій системі. Він має вигляд:

```
FUNCTION 74151b (c, b, a, d[7..0], gn)
RETURNS (y, wn);
```

Цей мультиплексор описаний мовою AHDL, однак можна його використати і в описах на Verilog. У дужках після назви функції перераховані вхідні порти, до яких підключаються ланцюги. Після слова RETURN вказуються вихідні сигнали.

Розглянемо приклад використання в проектах параметризованих модулів, створених розробником. В цьому модулі використовується восьмирозрядний регістр, створений на базі п’ятирозрядного шляхом зміни значення параметра.

```
module params(out1,in1,clk); // початок опису модуля
output [7:0] out1;           // вихідний 8-розрядний  порт
input [7:0] in1;             // вхідний 8-розрядний порт
input clk;                   // тактовий сигнал
reg [7:0] out1;
      reg4 reg8(out1, in1, clk); // використання модуля
      defparam reg8.n=7;     // перевизначення параметра
endmodule                    // кінець опису модуля
// параметризований модуль 5-розрядного паралельного регістра.
module reg4(out_r, in_r, clk_r);     // опис регістра
parameter n=4;                       // визначення параметра
output [n:0] out_r; // оголошення вихідного порту розрядністю n=4
input [n:0] in_r;   // оголошення вхідного порту розрядністю n=4
input clk_r;              // тактовий сигнал
reg [n:0] out_r;          // вихід має тип регістр
      always @(posedge clk_r)      // по фронту тактового сигналу
```

```
    begin
    out_r=in_r;   // передаємо на вихід значення вхідних сигналів
    end
endmodule         // кінець опису модуля
```

Часова діаграма представлена на рис.1.32.

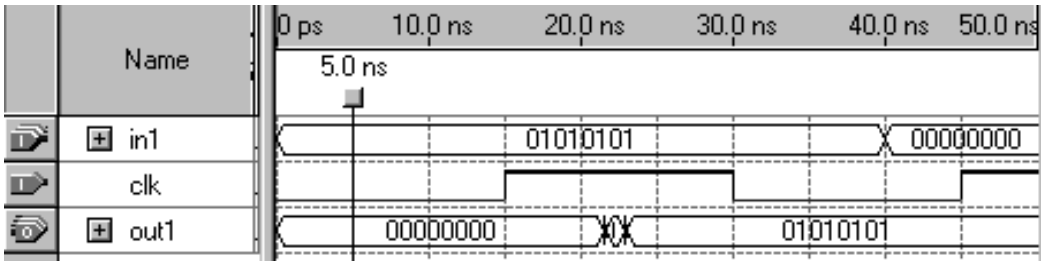

Рис.1.32 Часова діаграма роботи восьмирозрядного паралельного регістра

З аналізу діаграми видно, що описаний модуль представляє з себе паралельний регістр. Якщо в якості параметра вказати число, розрядність якого більша за розрядність порту (вхідні порти виявилися надлишковими), то сигнали, що надходять на дані порти, не беруть участь у формуванні вихідного сигналу. Однак в інтерфейсі синтезованої схеми вони присутні. Інтерфейс синтезованої схеми визначається інтерфейсом Verilog-опису, по якому схема синтезується. Варто уважно стежити, які вхідні сигнали беруть участь у формуванні вихідних сигналів і не залишати надлишкові. Якщо в проекті присутні невикористані порти, при компіляції проекту будуть видані попередження.

Структурний опис включає типи і назви компонентів з яких складається схема, а також зв'язки між ними. Функції, які реалізуються компонентами (модулями) при структурному описі не вказуються. Повний опис модулів повинен бути розміщений в проектній бібліотеці, яка підключена до структурного опису. Назви модулів, примітивів є глобальними іменами, тобто вони доступні у всьому середовищі проекту. Локальні імена видні в своїй області визначення (якщо не використовувати ієрархічні імена). Область видимості даних локалізована в таких конструкціях, як модуль, опис функції, задача (**task**), іменована група операторів (**begin-end**, **fork-join**). САПР шукає опис даних спочатку в локальному блоці, а потім у зовнішніх. При використанні ієрархічних назв, можуть становитися видимими зовні імена даних внутрішніх блоків, і з одного модулю внутрішні дані іншого (наприклад, параметри).

1.17. Рекомендації по створенню проектів

Наступні рекомендації допоможуть Вам найбільш ефективно розробляти свої проекти в системах MAX+plus II і Quartus.

- Використовуйте в Текстовому редакторі (Text Editor) виділення кольором ключових слів мови. Це допоможе уникнути синтаксичних помилок, а також легко виділити різні блоки і конструкції в проекті. Для включення цієї опції на початку роботи запишіть файл із розширенням Verilog (.v).
- Намагайтеся у своїх проектах використовувати смислові ідентифікатори. Це допоможе уникнути плутанини і скоротити час на пошук змінної. Зручно буде вказувати в ідентифікаторі змінної її розрядність.
- Використовуйте відступи при описі язикових конструкцій. Програма буде більш наочною і зручно читатися. Не буде проблем із пошуком вкладених блоків.
- Там, де можливо, заміняйте конструкцію **if** оператором вибору **case**.
- Мова Verilog є регістрозалежною. Однак у системі MAX+plus II рядкові і прописні букви не розрізняються компілятором. Тому не використовуйте ідентифікатори, які відрізняються лише регістром букв.
- Контроль подій повинен бути описаний після ключового слова **always** у конструкції **always**.
- Змінні при процедурному призначенні повинні бути описані за допомогою ключового слова **reg** на початку опису модуля, після його визначення.
- Коли Ви використовуєте Текстовий редактор системи MAX+plus II для створення файлів опису проекту мовою Verilog, уникайте рядків довжиною більше 255 символів. Оптимальна довжина рядка – горизонтальний розмір екрана. Використовуйте Enter для початку нового рядка.
- Ключові слова, ідентифікатори і числа повинні бути розділені операторами або пробілами.
- Однорядкові коментарі починаються із символів //. Багаторядкові коментарі заключаються в блок /* */.
- Для з'єднання примітивів використовуйте тільки дозволені з'єднання. Не всі примітиви можуть бути з'єднані.
- Перед початком розробки проекту вкажіть сімейство і тип мікросхеми.
- Забороняється в одному модулі поєднувати тригери, які керуються як фронтом, так і зрізом тактового сигналу.

Часто розробник відразу виконує опис проекту на синтезуємій підмножині HDL-мов. Відхилення від цих правил в більшості випадків виявляється системою синтезу.

1.18. Конструкції мови Verilog, що підтримуються САПР MAX+plus II

Поняття синтезованої підмножини мови Verilog є центральним для розуміння всієї книги в цілому. Синтезатори, у відмінності від систем моделювання, підтримують не всі конструкції мови Verilog, а тільки деяку підмножину цієї мови, що прийнято називати синтезованою. Щоб скористатися системою MAX+plus II, яка дозволяє одержати опис апаратури, що реалізує необхідну поведінку проектованої схеми, розробник повинен написати Verilog-код на синтезованій підмножині цієї мови.

Таблиця 1.5. Підмножина Verilog в САПР MAX+plus II

Типи даних	
Registers (регістри)	підтримуються
Vectors (вектори)	підтримуються
Strengths (сили)	не підтримуються
Charge Strength (пріоритет сили)	не підтримується
Drive Strength (сила драйвера)	не підтримується
Implicit Declarations	підтримується
Net Initialization	не підтримується
wire and tri Nets (ланцюги)	підтримуються
tri0 and tri1 Nets	не підтримуються
supply0 and supply1 Nets (живлення)	не підтримуються
Memories (пам'ять)	не підтримується
integers, Reals, Times (цілий, дійсний тип даних і час)	підтримуються тип integer у конструкції for
Parameters (параметри)	підтримуються
Оператори	
Арифметичні	підтримуються
Відносини	підтримуються
Рівності	підтримуються, крім === та !==
Логічні	підтримуються
Бітові	підтримуються
Згортки	підтримуються
Зсуву	Підтримуються, якщо правий операнд – константа
Умовний	підтримується

Об'єднання	підтримується, крім вихідних портів в описі модуля
АБО в контролі подій	підтримується
Адресація пам'яті	не підтримується
Строчкові	не підтримуються
Присвоєння	
Безперервне (Continuous Assignments)	підтримується
Процедурне (Procedural Assignments)	підтримується
Вентилі та перемикачі	
Вентилі and, nand, nor, or, xor, xnor	підтримуються (не більше 12 входів)
вентилі buf та not	підтримуються
bufif1, bufif0, notif1, notif0	підтримуються
MOS перемикачі	не підтримуються
CMOS перемикачі	не підтримуються
Джерела	не підтримуються
Поведінкове моделювання	
Примітиви, певні користувачем	не підтримуються
Блокуюче та неблокуюче процедурне присвоєння (Blocking & Nonblocking Procedural Assignments)	підтримується
Процедурне безперервне присвоєння (Procedural Continuous Assignments)	не підтримується
Assign and Deassign	не підтримується
Force and Release Procedural Continuous Assignments	не підтримується
Конструкції if-else, case	підтримуються
Цикли	підтримується тільки цикл for
Контроль подій (Event Controls)	підтримується
Часовий контроль (Delay Control)	не підтримуються
Конструкція begin-end	підтримується
Конструкція fork-join	не підтримується
Оператор initial	підтримується
Оператор always	підтримується
Завдання (tasks)	підтримуються
Функції (functions)	підтримуються
Конструкція Disable	не підтримується
Ієрархічні структури	
Модулі (module)	підтримуються
Конструкція Defparam	підтримується
Порти (Ports)	підтримуються

Спеціальні блоки (Specify Blocks)	не підтримуються
Директиви компілятора	
`define , `undef, `ifdef, `else, `endif, `include	підтримуються
`celldefine, `endcelldefine, `default_nettype, `resetall, `timescale, `unconnected_drive, `nounconnected_drive	не підтримуються
Системні завдання і функції (system tasks and functions)	не підтримуються

Основними проблемами синтезу є:

- перетворення (кодування) даних, які використовуються в алгоритмічних описах у дані для опису результуючих логічних схем;
- реалізація операторів і конструкцій мови високого рівня в логічні підсхеми;
- оптимізація апаратних витрат в рамках критеріїв ”складність - швидкодія”.

Термін ”синтезований Verilog-код” свідчить про те, що по даному алгоритмічному опису автоматично, за допомогою синтезатора, може бути побудована логічна схема, яка реалізує необхідну поведінку проектованої системи. В тексті жирним шрифтом відзначені зарезервовані слова, обрані проектувальником – розробником проектів цифрових схем мовою Verilog, або програміста, який пише програми мовою Verilog.

Оскільки електронні схеми будуються найчастіше на основі принципу двійкового представлення інформації, то дані будь-яких типів кодуються двійковими кодами. Кожен синтезований оператор або конструкція мови Verilog замінюється при автоматичному синтезі своєю підсхемою. Наприклад, оператор додавання при синтезі може бути замінено на суматор, розрядність якого визначається розрядністю операндів.

Схеми, що реалізують стандартні оператори (додавання, віднімання, множення) назвемо модулями, які генеруються. Модулі, які генеруються – параметризовані. Автоматичний синтез умовно можна розділити на два етапи: високорівневий і логічний синтез. Після високорівневого синтезу схема представляється у вигляді модулів, що генеруються і комбінаційних двовходових логічних елементів, після логічного синтезу виходить результуюча схема з елементів цільової бібліотеки синтезу. Наприклад, відносно суматорів можна поступити наступним чином – описати каскадну схему багаторозрядного суматора у вигляді послідовно з'єднаних однорозрядних суматорів. Досить компактний опис однорозрядного суматора можна зберігати в бібліотеці системи синтезу. Одержання логічних функцій для багаторозрядного суматора зведеться до генерації рекурентних логічних виразів.

Ще один варіант – генерація рівнянь із таблиць істинності. Такий підхід придатний для схем комбінаційної логіки. Наприклад, можна

описати таблицю істинності для n-розрядного суматора, потім одержати алгебраїчні вирази логічних функцій і представити їх у вигляді комбінаційної схеми із двовходових логічних елементів.

Різні синтезатори можуть працювати з різними синтезованими підмножинами цієї мови. Наприклад, система Quartus підтримує більшу кількість конструкцій мови в порівнянні з MAX+plus II (можливість створення примітивів користувача, підтримка даних типу пам'ять). В даній книзі розглядається синтезована підмножина мови.

Синтезована підмножина визначається через вказівку заборонених конструкцій мови (табл.1.5). Якщо проектувальник використав в Verilog-коді тип real, то синтезатор видасть повідомлення про помилку і схема не буде побудована. Не підтримуються ініціалізація значення сигналів у розділі декларації сигналів. Від'ємні числа при синтезі представлені в додатковому коді. Допускається використання змінних при написанні синтезованих Verilog-кодів.

У якості однією з бібліотек синтезу обрана бібліотека мікросхем серії 74, а також бібліотека параметризованих модулів. У бібліотеці мікросхем 74 серії містяться різні елементи, серед яких інвертори, мультиплексори, вентилі та ін.

Ми розглянули реалізацію декількох операторів мови Verilog. Якщо в алгоритмічному описі є безліч операторів, то на етапі високорівневого синтезу здійснюється заміна конструкцій мови Verilog логічними схемами – тригерами і двовходовими комбінаційними логічними схемами. Нагадаємо, що вихідний Verilog-опис повинен бути синтезованим.

Для моделювання цифрової схеми, в цілому, потрібне написання спеціальної тестової програми. Її призначення:

- організувати подачу вхідних сигналів;
- одержати реакцію тестуємої системи;
- зрівняти, якщо потрібно, реакцію системи з очікуваною.

Для зручності порівняння отриманої реакції схеми з очікуваною реакцією, двійковий формат вихідних сигналів перетворюється у формат цілих чисел. Саме такий формат найбільш часто використовується в алгоритмічному описі. Реакція системи – це значення вихідних сигналів схеми, яка моделюється. В системах MAX+plus II і Quartus для моделювання використовується Редактор часових діаграм (Waveform Editor) та Симулятор (Simulator).

2. ВИКОРИСТАННЯ VERILOG ДЛЯ ОПИСУ ЦИФРОВИХ ПРИСТРОЇВ

2.1. Реалізація логічних функцій

В першому розділі був розглянутий приклад реалізації логічної функції за допомогою базових логічних елементів. Це досить просто реалізувати, оскільки запис функції в нормальній диз'юнктивній формі полегшує завдання розробнику. Проте, опис цієї конструкції досить великий за розмірами і така реалізація, незважаючи на очевидну простоту, позбавлена деякої наочності. Нагадаємо, функція в мінімізованому виді має вигляд:

$$y = \bar{x}_3 x_2 \bar{x}_1 \bar{x}_0 + x_3 \bar{x}_2 \bar{x}_1 x_0 + x_3 x_2 x_1 + x_2 x_1 x_0 .$$

Розробимо модуль, який реалізує цю функцію. Опис модуля має вигляд:

```
module logfunc(y, x);     // назва модуля і список портів
output y;                 // декларація вихідного  порту
input [3:0] x;            // декларація 4-розрядної вхідної шини
assign y=((~x[3] & x[2] & ~x[1] & ~x[0]) | (x[3] & ~x[2] & ~x[1] &
x[0]) |
        (x[3] & x[2] & x[1]) | (x[2] & x[1] & x[0]));
endmodule                 // закінчення опису модуля
```

За допомогою конструкції **assign** здійснюється безперервне призначення вихідному сигналу результату виконання логічного виразу. В цьому випадку значення вихідного сигналу змінюється при зміні будь-якого з операндів (вхідних сигналів). В реалізації функції були використані унарний оператор інверсії, бінарні оператори І та АБО.

Відкомпілювавши проект, ми можемо перевірити роботу модуля реалізації логічної функції. Результати моделювання абсолютно аналогічні результатам, які були отримані при реалізації схеми в графічному редакторі.

Розглянемо реалізацію цієї логічної функції за допомогою оператора вибору. З опису модуля видно, що цей спосіб найбільш простий і найменш трудомісткий.

```
module logfunc(y,x);      // заголовок модуля
output y;                 // опис виходу
input [3:0] x;            // створення вектора вхідних сигналів
reg y;                    // повідомляємо вихід як регістр
always @(x)               // при зміні сигналу на вході
case (x)                  // перевіряємо відповідність константам
      4'b0100 : y=1;      // встановлюємо на виході 1 при
```

```
        4'b1001 : y=1;      // рівності значення x визначеній константі
        4'b1110 : y=1;
        4'b0111: y=1;
        default : y=0;      // якщо жодна з альтернатив не підходить
    endcase                 // на виході встановлюється 0
    endmodule               // закінчення опису модуля
```

За допомогою оператора **case** вибирається одна з альтернатив і виконується відповідний оператор. Вибір альтернативи визначається кодом, який присутній на вхідному порту. Вираз повинен бути дискретного типу, або мати тип одновимірного масиву, значення елементів якого може бути представлене як рядок бітів. Всі можливі варіанти зміни вхідного коду повинні бути враховані в конструкції вибору.

Часові затримки, що виникають в схемі при такому описі, не відрізняються від затримок, отриманих раніше. Однак, при моделюванні схеми в Сигнальному редакторі, зникли «паразитні» імпульси в моменти зміни сигналів на входах (рис.2.1). Це обумовлено тим, що вихід має тип «регістр».

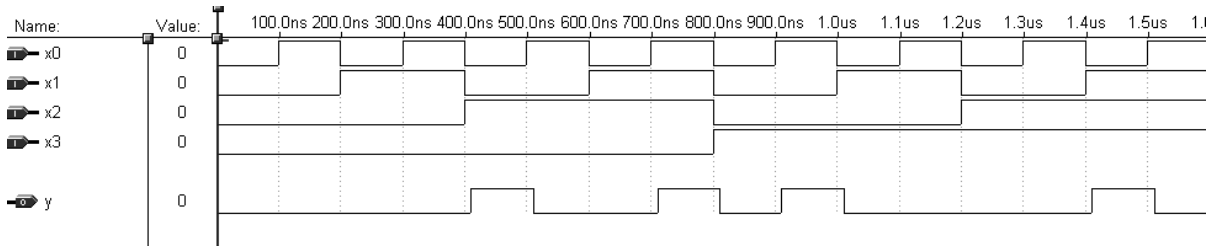

Рис.2.1. Часова діаграма роботи модуля

В принципі, єдина відмінність від попередніх реалізацій – вихід *y* визначений як регістр. Це зроблено для того, щоб його значення призначалося явно. Таке призначення сигналу називається процедурним. Дані типу «ланцюг» не можуть бути призначені явно, вони мають потребу в сигналі-драйвері. Таке призначення називається безперервним. Значення функції в даному прикладі обчислюється завжди при зміні хоча б одного входу – система постійно їх відслідковує.

Для повної картини реалізуємо задану функцію з використанням умовного оператора. В принципі, нічого нового ми вже не бачимо – зрозуміло, що в нашому випадку умовний оператор програє оператору case в наочності представлення, однак синтезовані реалізації ідентичні.

```
module logfunc(y, in);  // початок модуля
output y;               // опис виходу
input [3:0] in;         // опис входу
assign y=(in==4'b0100)?(1):((in==4'b1001)?(1):(in==4'b111x)?
```

```
    (1):(in==4'bx111)?(1):(0));      // реалізація функції
endmodule                            // закінчення опису модуля
```

В приведеному описі вхідний порт in визначений як 4-бітний вектор. Установка нового значення вихідного сигналу “y” відбувається щоразу при зміні даних на вході модуля.

2.2. Проектування мультиплексорів

Мультиплексором називається цифровий комбінаційний пристрій, призначений для прийому інформації з декількох паралельних ліній і передачі його в одну вихідну лінію у вигляді впорядкованої послідовності. Вибір вхідної лінії та її гальванічне з'єднання з вихідною забезпечується відповідним адресним кодом. При наявності n адресних входів комбінаційною схемою мультиплексора забезпечується комбінація 2^n ліній передачі інформації. Робота мультиплексора для n=2 описується логічною функцією:

$$y = \overline{v}(\overline{a}_0 \cdot \overline{a}_1 \cdot d_0 + \overline{a}_0 \cdot a_1 \cdot d_1 + a_0 \cdot a_1 \cdot d_2 + a_0 \cdot \overline{a}_1 \cdot d_3).$$

З формули витікає, що зміна кодів на входах a_0, a_1, яка відбувається з частотою генератора адресних сигналів, може приводити до часового розділення інформаційних сигналів d_0–d_3 на виході y (часто сигнали d_0–d_3 називають “даними”). Мультиплексор може виконувати і дещо іншу функцію – цілеспрямовано вибирати дані по окремому, або декількох окремих адресах. В такому випадку подібні пристрої називаються селекторами (від select – вибирати).

Логічна схема мультиплексора може бути реалізована в базисі І-НІ-АБО. Опис модуля мультиплексора на рівні базових логічних елементів приводиться нижче.

```
module mux_4_1_base (out, in1, in2, in3, in4, cntrl1, cntrl2);
// початок модуля
input in1, in2, in3, in4, cntrl1, cntrl2; // опис входів даних та адресу
output out;                               // опис виходу
wire notcntrl1, notcntrl2, w, x, y, z;    // визначаємо додаткові ланцюги
      not (notcntrl1, cntrl1);            // інверсія керуючих сигналів
      not (notcntrl2, cntrl2);            // за допомогою інвертора
      and (w,in1, notcntrl1, notcntrl2);  // реалізація пристрою
      and (x,in2, notcntrl1, cntrl2);     // логічна операція І
      and (y,in3, cntrl1, notcntrl2);
      and (x,in4, cntrl1, cntrl2);
      or (out, w, x, y, z);               // логічна операція АБО
endmodule                                 // закінчення опису модуля
```

Перший рядок будь-якого модуля починається із ключового слова **module**. В заголовку модуля вказується назва, по якій він буде викликатися. Порядок опису портів наступний: спочатку описуються вихідні порти, потім вхідні. Опис типу портів подібний до оголошення змінних у мові С, за винятком відсутності можливості присвоїти портам початкові значення. Всі порти в розділі оголошення повинні бути оголошені як **input** (вхід), **output** (вихід) або **inout** (двонапрямлений).

В розглянутому прикладі в якості інформаційних використовуються порти in1-in4, адресні входи – порти cntrl1-cntrl2. В описі модуля немає нічого складного.

Опис вентиля НІ (інвертора) відбувається наступним чином:

not (notcntrl1, cntrl1);

Вихідний порт називається notcntrl1, вхідний – cntrl1. Базові вентилі визначені в мові як ключові слова, тому для їх використання не обов'язково присвоювати кожному елементу певного імені. Значення на виході змінюється автоматично кожного разу при зміні рівня сигналу на вході. Закінчення модуля супроводжується ключовим словом **endmodule**.

Розглянемо інший варіант реалізації мультиплексора. Спробуємо спроектувати мультиплексор «4 : 1» (рис.2.2) з використанням структури **case**:

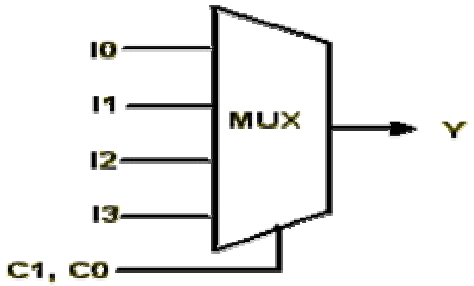

Рис.2.2. Мультиплексор «4 : 1»

Опис модуля мультиплексора має вигляд:

```
module mux_4_1_case (Y,I0,I1,I2,I3,C0,C1); // початок модуля
input I0, I1, I2, I3, C0, C1; // опис інформаційних та адресних входів
output Y;                     // опис виходу
reg Y;                        // визначаємо вихід як регістр
always @ (I0 or I1 or I2 or I3 or C0 or C1)
// при зміні сигналу на входах
begin                         // початок вкладеної конструкції
      case ({C1,C0})          // поєднуємо адресні входи
            2'b00 : Y <= I0 ;  // залежно від стану адресних входів
            2'b01 : Y <= I1 ;  // передаємо на вихід відповідний
```

```
            2'b10 : Y <= I2 ;   // інформаційний сигнал
            2'b11 : Y <= I3 ;
        endcase                 // завершення конструкції вибору
end                             // кінець вкладеної конструкції
endmodule                       // кінець модуля
```

В цьому прикладі також демонструється використання оператору об'єднання сигналів {}. Це виконується для того, щоб перевіряти значення керуючих сигналів не окремо, а разом. У такий спосіб запис 2'b10 означає установку на виході мультиплексора значення сигналу, який присутній на вході I2, за умови, що на адресних входах C1 і C0 встановлені логічні рівні 1 та 0 відповідно. В даному прикладі при описі портів можна оголосити один адресний порт, наприклад з ідентифікатором Addr, як дворозрядний вектор:

```
…
output [1:0] Addr;
…
```

Розглянемо інший спосіб – опис мультиплексора з використанням оператора **if**. Спроектуємо мультиплексор «4 : 1» (рис.2.3), з використанням такої конструкції.

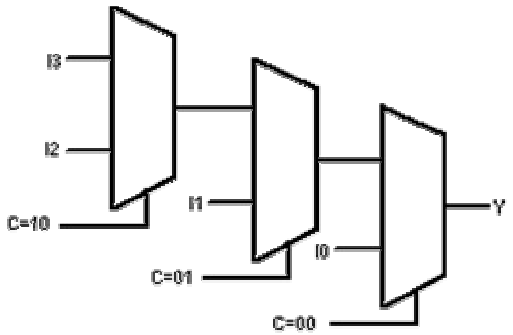

Рис.2.3. Каскадне включення мультиплексорів

Опис модуля мультиплексора має вид:

```
module mux_4_1_if(Y,I0,I1,I2,I3,C0,C1);          // початок модуля
input I0,I1,I2,I3,C0,C1;          // опис вхідних портів
output Y;                         // опис виходу
reg Y;                            // визначаємо вихід як тип «регістр»
always @ (I0 or I1 or I2 or I3 or C0 or C1)       // при зміні сигналу
                                  // на кожному із входів
        begin                     // початок вкладеної конструкції
        if ({C1,C0} == 2'b00) Y = I0;  // залежно від сигналу
        else if ({C1,C0} == 2'b01) Y = I1;     // на адресних входах
        else if ({C1,C0} == 2'b10) Y = I2;     // установлюємо вихід в
```

```
        else if ({C1,C0} == 2'b11) Y = I3;     // певний стан
        end                                    // кінець вкладеної конструкції
endmodule                                      // кінець опису модуля
```

Умова в операторі **if** повинна бути булевого типу. Сигнал на виході мультиплексора визначається адресним кодом C1-C0 і рівнем сигналу на відповідному інформаційному вході.

Існує ще один спосіб реалізації мультиплексора: використання оператора **assign**:

```
module mux4_1_assign(Y,I0,I1,I2,I3,C0,C1); // початок модуля
input I0,I1,I2,I3,C0,C1;                   // опис входів
output Y;                                  // опис виходу
wire a_0, a_1, a_2, a_3;                   // внутрішні з'єднання
wire c0_n, c1_n;                           // опис «ланцюгів»
wire Y;
        assign c0_n = ~C0;                  // інвертуємо адресні сигнали
        assign c1_n = ~C1;
        assign a_0 = I0 & c1_n & c0_n; // за допомогою оператора I
        assign a_1 = I1 & c1_n & C0;   // визначаємо стан виходів
        assign a_2 = I2 & C1 & c0_n;
        assign a_3 = I3 & C1 & C0;
        assign Y = a_0 | a_1 | a_2 | a_3;  // поєднуємо виходи через АБО
endmodule
```

В описі модуля оголошені чотири додаткових ланцюги, рівні сигналів на яких визначаються за допомогою оператора I. По суті, представлений опис реалізує чотири логічних функції, результати яких об'єднуються оператором АБО і представляють з себе вихідний сигнал мультиплексора. На рис.2.4 представлені результати моделювання роботи мультиплексора.

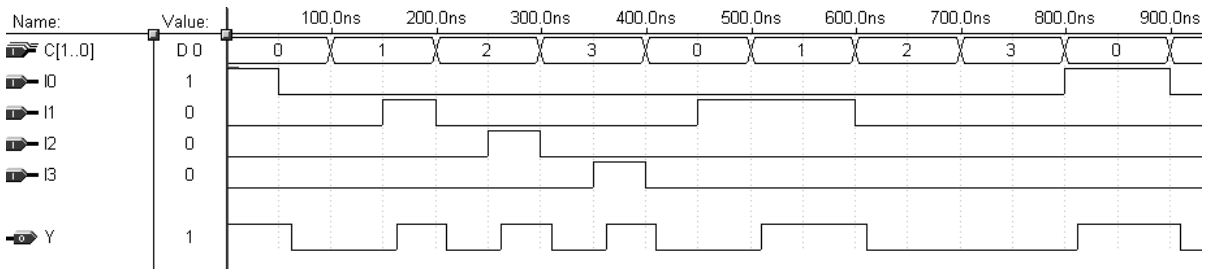

Рис.2.4. Діаграма роботи мультиплексора

З наведених прикладів видно, що пристрій можна описати за допомогою різних конструкцій. Безумовно, деякі з них позбавлені наочності, і з першого погляду буває важко зрозуміти алгоритм роботи

модуля. Розробнику надається право вибору стилю опису пристроїв. Це дозволяє найбільш оптимально побудувати модель проектованої системи. Крім того, деякі оператори мови дозволяють значно скоротити час розробки і витрачені сили на проектування багатьох стандартних алгоритмів (наприклад, процедура множення).

Суттєво розширити можливості мультиплексора можна, якщо ввести дозволяючий вхід V. Він дозволяє синхронізувати роботу мультиплексора з іншими схемами, а також використовується для нарощування розрядності адресних сигналів. Для того, щоб зробити це, потрібно перед оператором **always** (**case**, **assign**) написати наступну конструкцію:

```
if (V)          // якщо на дозволяючому вході лог.1
      begin
      ……;       // виконується визначення сигналу на виході
      end;
else Y=0;       // інакше на виході сигнал низького рівня
```

Зрозуміло, що в описанні портів, необхідно добавити рядок: input V.

Так як мультиплексор є комбінаційною схемою, то на його базі можуть бути реалізовані різні логічні функції. В кожному із прикладів реалізації мультиплексорів, при моделюванні і аналізі часових затримок, були отримані аналогічні результати.

Для використання мультиплексора в схемах, розроблених у Графічному редакторі (Graphic Editor) системи MAX+plus II, можна зробити символ з описаного на Verilog (VHDL, AHDL) модуля. Для цього з меню File системи вибрати команду Create Default Symbol (рис.2.5).

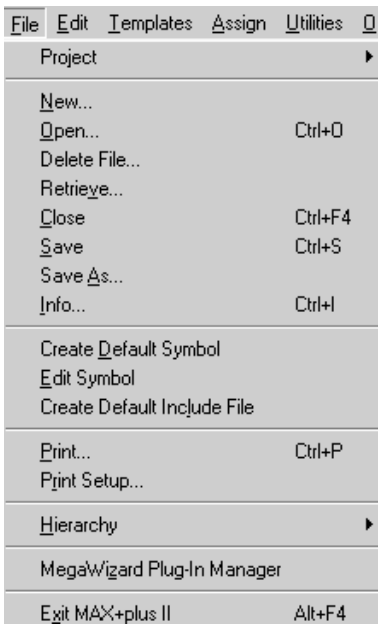

Рис.2.5. Меню File системи

Після цього можна використовувати створений елемент мультиплексора в інших проектах. При цьому текстовий файл із описом модуля буде автоматично підключений до проекту. Вигляд створеного символу мультиплексора приводиться на рис.2.6.

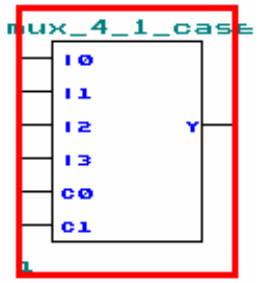

Рис.2.6. Створений символ мультиплексора «4 : 1»

Досить часто виникає завдання мультиплексування шин. При використанні графічного опису схеми розробник повинен буде використати велику кількість мультиплексорів. Незручності такого підходу очевидні: відсутність гнучкості проекту і використання великої площі робочої області. Розробка такого мультиплексора займе велику кількість часу. Текстовий опис подібного модуля досить простий. Використання параметрів для вказівки розрядності шин і кількості входів дозволить адаптувати модуль для різних проектів. САПР MAX+plus II та Quartus містять бібліотеки параметризованих модулів, які можна використати у власних розробках. Однак при використанні параметризованого модуля мультиплексора в САПР MAX+plus II розробник може зіткнутися з деякими труднощами. Проблема полягає в тому, що не підтримується можливість використання двовимірних (two-dimensional) портів. Прототип функції мультиплексора на AHDL має вигляд:

```
FUNCTION lpm_mux (data[LPM_SIZE-1..0][LPM_WIDTH-1..0],
sel[LPM_WIDTHS-1..0], clock, aclr)
WITH (LPM_WIDTH, LPM_SIZE, LPM_WIDTHS, LPM_PIPELINE)
RETURNS (result[LPM_WIDTH-1..0]);
```

Перший параметр (data) потребує вказівки кількості і розрядності шин даних. Для використання мультиплексора шин можна створити модуль за допомогою MegaWizard Plug-In Manager, і використати його при структурному описі схеми. На рис.2.7 приводиться приклад модуля мультиплексора 24-розрядних шин, який має 8 входів.

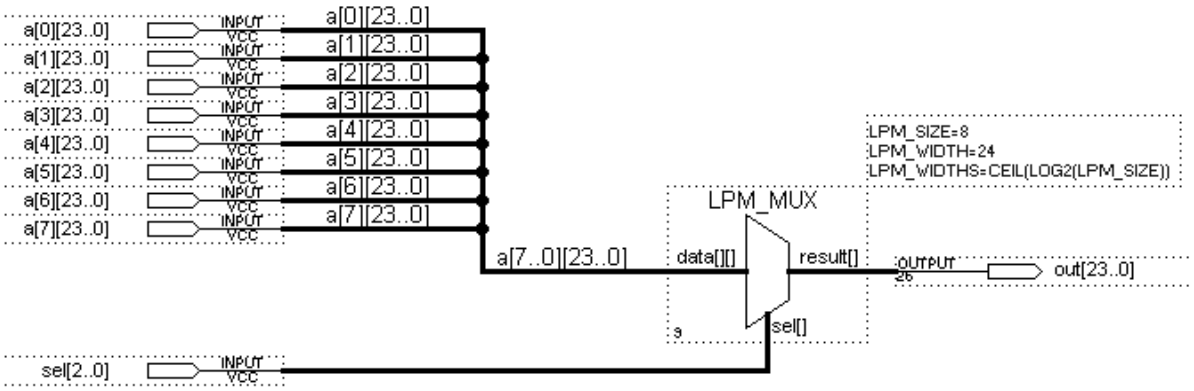

Рис.2.7. Мультиплексор шин

Шляхом зміни значення параметра LPM_SIZE розробник може вказати кількість шин. Параметр LPM_WIDTH визначає розрядність шин. Altera рекомендує використовувати параметризовані модулі в розроблених проектах замість мікросхем 74XXX серії.

Незважаючи на відсутність підтримки двовимірних портів, існує можливість описувати на Verilog модулі мультиплексорів шин. Однак при цьому як параметр можна вказати тільки розрядність шин. Кількість вхідних шин вказується при описі портів модуля. Наступний приклад представляє собою модуль мультиплексора «2:1» для 24-розрядних шин.

```
module busmux(din1, din2, sel, dout);      // назва модуля та перелік сигналів
parameter WIDTH=24;                        // параметр розрядності шини
input [WIDTH-1:0] din1;                    // перший вхід даних
input [WIDTH-1:0] din2;                    // другий вхід даних
input sel;                                 // адресний вхід
output [WIDTH-1:0] dout;                   // вихідний порт
      assign dout=(sel==0)?din1:din2;      // вибір відповідної шини
endmodule                                  // кінець модуля
```

В даному прикладі використовується тернарний оператор разом з конструкцією безперервного присвоєння. На рис.2.8 показаний приклад використання мультиплексора, створеного за допомогою MegaWizard PlugInManager. Цей мультиплексор відповідає наведеному текстовому опису.

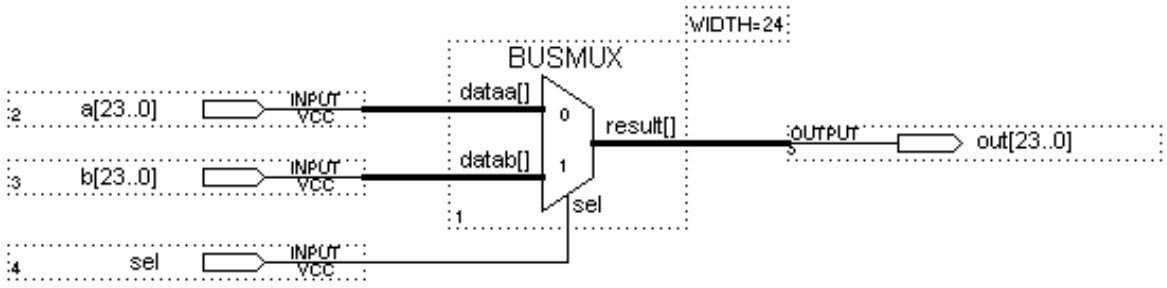

Рис.2.8. Мультиплексор «2 в 1» 24-розрядних шин

Змінивши в описі модуля значення параметра WIDTH, змінюється розрядність шин даних din1 та din2.

Окрім звичайного оператора вибору, в мові опису апаратури Verilog використовується оператор вибору **casex**. Опис модуля мультиплексора «4 : 1» з використанням цього оператору приводиться нижче.

```
module decoder(q, a, b, c, sel); // початок модуля
parameter WIDTH=8;          // розрядність даних
output [WIDTH-1:0] q;       // вихідний порт
input [WIDTH-1:0] a,b,c;    // вхідні інформаційні шини
input [1:0] sel;            // адресні входи
reg [WIDTH-1:0] q;          // вихід має регістровий тип
always@(a or b or c)        // початок процедурного блоку
      casex(sel)            // в залежності від адресу
      2'b00 : q<=a;         // на вихідний порт передаються
      2'b01 : q<=b;         // дані з визначеного вхідного порту
      2'b1x : q<=c;         // значення молодшого біту не важливо
      default: q<=c;
endcase                     // кінець конструкції вибору
endmodule
```

В цьому модулі використовується параметр для визначення розрядності даних, з якими працює мультиплексор. При аналізі конструкції вибору видно, що на виході q встановиться рівень сигналу, який присутній на вхідному порту „c”, якщо старший біт адресу sel має значення 1. Значення молодшого біту адресу не враховується.

Після моделювання роботи мультиплексора була отримана часова діаграма, яка приводиться на рис.2.9.

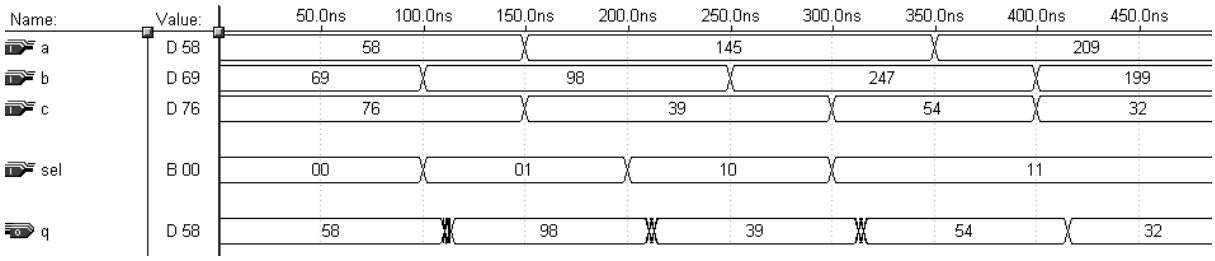

Рис.2.9. Діаграма роботи мультиплексора шин

З діаграми видно, що в залежності від значення на адресному вході, на вихід передаються дані, які присутні на одному з вхідних портів. Тристабільні шини популярні при проектуванні плат, оскільки вони зменшують кількість провідників. Однак на рівні кристалу часто рекомендується використовувати мультиплексні шини, тобто ставити мультиплексори на вході кожного приймача, які з'єднані з усіма необхідними джерелами сигналів.

Розглянемо приклад створення комбінаторного примітиву мультиплексора. В САПР Quartus опис примітиву мультиплексора та приклад його використання в модулі представлені нижче.

```
module muxprim(y,sel,A,B);   // заголовок модуля
output y;                    // вихідний сигнал
input A,B,sel;               // інформаційні входи (A,B) та керуючий вхід
mux_2_1(y,sel,A,B);          // підключення примітиву мультиплексора
endmodule                    // кінець опису модуля
// опис комбінаторного примітиву користувача
primitive mux_2_1(out, control, dataA, dataB);   //заголовок примітиву
output out;                  // вихідний сигнал
input control, dataA, dataB; // вхід керування та інформаційні входи
// control dataA dataB out – порядок опису сигналів в таблиці станів
table                        // початок опису таблиці станів
      0 1 ? : 1;             // значення на вході dataB неважливе
      0 0 ? : 0;
      1 ? 1 : 1;             // значення на вході dataA неважливе
      1 ? 0 : 0;
      x 0 0 : 0;             // незалежно від стану сигналу на керуючому
      x 1 1 : 1;             // вході, на виході встановиться 0 або 1
endtable                     // кінець таблиці опису станів
endprimitive                 // кінець опису примітиву
```

2.3. Розробка дешифраторів і демультиплексорів

Велика різноманітність кодів, які використовуються при представленні і обробці інформації, вимагає від схемотехніки спеціальних

пристроїв для перетворення інформації з одної форми до іншої. Для розв'язання таких завдань і використовуються перетворювачі кодів, або дешифратори. Демультиплексори – це цифрові комбінаційні пристрої, функціональне призначення яких протилежне мультиплексорам. У них сигнали з одного інформаційного входу „x" розподіляються на 2^n виходів y_n, які комутуються *n* адресними входами. Демультиплексори здебільшого поєднуються з дешифраторами (дешифратори-демультиплексори) і виконують функції як дешифрації, так і демультиплексування. Бібліотека MAX+plus II містить широкий набір приладів, що серійно виробляються. Серед них перетворювачі двійкового коду в код семисегментних індикаторів (7446–7449, 74246–74248, перетворювач коду Грея в десятковий код (7444) та інші.

Найпростішим способом створення модуля дешифратора на Verilog є використання конструкції вибору **case**, в списку альтернатив якої потрібно врахувати всі можливі комбінації вхідного коду (тобто задати таблицю станів). Очевидно, при великій розрядності вхідного коду кількість альтернатив в конструкції **case** збільшується пропорційно квадрату цього значення. Тому зручніше буде використовувати при описі модуля дешифратора адресацію бітів. Наступний приклад представляє з себе параметризований модуль дешифратора «N:M». Значення N та M – це розрядності вхідної та вихідної шини.

```
module decNM(dout,en,din);   // заголовок модуля дешифратора
parameter N=4;               // розрядність вхідної шини
parameter M=16;              // розрядність вихідної шини
output [M-1:0] dout;         // вихідний порт
input en;                    // вхід дозволу роботи
input [N-1:0] din;           // вхідний порт
reg [M-1:0] dout;            // вихід має регістровий тип
integer i;                   // змінна для використання в циклі
always @(din or en)
if(en==1)                    // якщо на вході дозволу 1
begin
      dout=0;                // обнуління значення вихідних сигналів
      for(i=0;i<M;i=i+1)     // виконуємо цикл M разів
      dout[i]=(din==i)?1:0;  // якщо виконується умова, на
      //відповідному виході встановлюється високий рівень сигналу
      // для САПР Quartus розкоментувати наступний рядок
// dout[din]=1; // на потрібному виводі встановлюється сигнал лог.1
end
else
dout=0;                  // якщо на вході дозволу 0, на виході також 0
endmodule                // кінець опису модуля
```

В САПР MAX+plus II неможна адресуватися до окремого розряду у векторі якщо номер біту не є константою. Тому використовується цикл, в якому перевіряється рівність значення змінної „i” та вхідного коду. Якщо вони рівні, тоді на відповідному виході встановлюється лог.1. В САПР Quartus підтримується можливість використовувати адресацію бітів, тому цикл можна замінити наступним образом: dout[din]=1;. Звертаємо увагу на необхідність обнуління значення на виході dout перед циклом. Вихід має регістровий тип і значення на ньому зберігається до наступного присвоєння. Оскільки в даному випадку змінюються значення окремих бітів, то необхідно встановлювати значення 0 на виході при дешифрації наступного слова. В протилежному випадку, при переборі всіх можливих вхідних значень, на всіх виходах модуля будуть сигнали високого рівня, тобто робота модуля дешифратора буде некоректною. Значення N та M пов’язані співвідношенням: $M=2^N$.

Розглянемо приклад демультиплексора «1 : 8». Опис модуля на Verilog представлений нижче.

```
module demuxer(y,in,cntrl);     // початок модуля
output [7:0] y;                 // опис виходу
input in, cntrl;                // опис входів
wire [2:0] cntrl; // вхід, визначений як 3-бітний вектор типу «ланцюг»
reg [7:0] y;        // вихід, визначений як «регістр»
        always @(cntrl or in)   // очікується зміна сигналів на вході
        begin                   // початок тіла always
                y=8'd0;         // призначаємо виходу низький рівень
                case (cntrl)    // вибираємо адресний сигнал з наступних
                        3'b000: y[0]=in;
                        3'b001: y[1]=in;
                        3'b010: y[2]=in;
                        3'b011: y[3]=in;
                        3'b100: y[4]=in;
                        3'b101: y[5]=in;
                        3'b110: y[6]=in;
                        3'b111: y[7]=in;
                default: y=0;
                endcase         // завершення конструкції вибору
        end                     // завершення тіла always
endmodule                       // кінець модуля
```

На рис.2.10 наведені результати моделювання описаного демультиплексора.

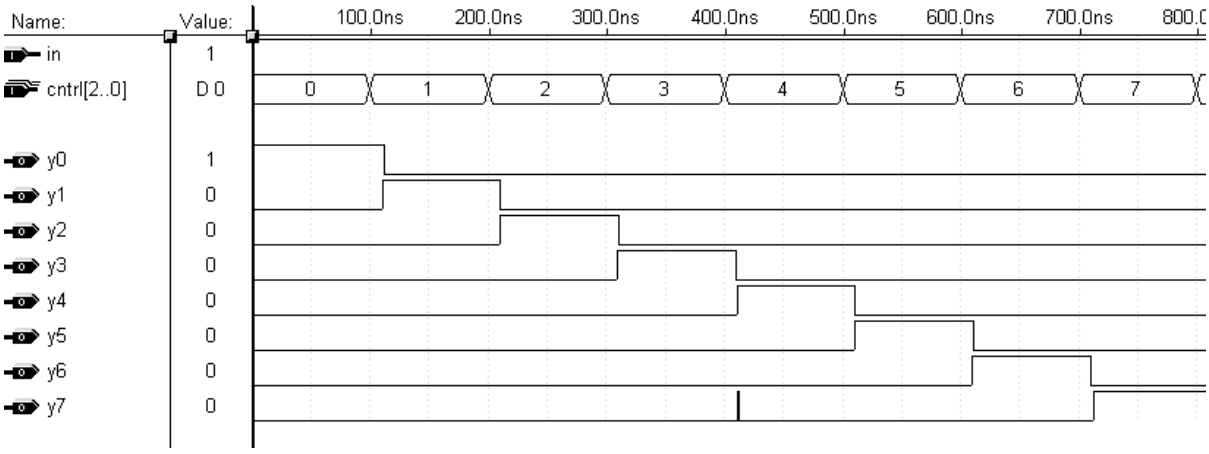

Рис.2.10. Результати моделювання роботи демультиплексора

В кожному рядку оператора **case** (тобто на кожному наборі аргументів) явно задане значення тільки одного виходу модуля. На інших виводах при даному наборі аргументів будуть логічні нулі. Це забезпечується шляхом установки на виході сигналів низького рівня перед конструкцією вибору **case**.

Можна сказати, що дешифратор реалізує окремий випадок демультиплексора. Отже, схема дешифратора може бути отримана зі схеми демультиплексора при виключенні з неї інформаційного входу та використанні адресних виходів у якості інформаційних. В наступних прикладах демонструється використання конструкції вибору (**case**) для реалізації дешифраторів.

Розглянемо дешифратор двійкового коду в код семисегментних індикаторів. Умовне розташування світлодіодів матриці семисегментного індикатора наведено на рис.2.11. Відповідно до нього створюємо таблицю відповідності двійкового коду до коду семисегментних індикаторів (табл.2.1), приписуючи значення «1» тому світлодіоду, який засвічується при відображенні цифри, та нуль – у протилежному випадку.

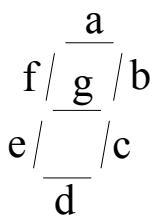

Рис.2.11.

Таблиця 2.1. Таблиця відповідності

№	x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0

8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
A	1	0	1	0	1	1	1	0	1	1	0
B	1	0	1	1	0	0	1	1	1	1	1
C	1	1	0	0	1	0	0	1	1	1	0
D	1	1	0	1	0	1	1	1	1	0	1
E	1	1	1	0	1	0	0	1	1	1	1
F	1	1	1	1	1	0	0	0	1	1	1

З табл.2.1. можемо записати функції роботи кожного світлодіода:

$$a=\Sigma 0,2,3,5,6,7,8,9;$$
$$b=\Sigma 0,1,2,3,4,7,8,9;$$
$$c=\Sigma 0,1,3,4,5,6,7,8,9;$$
$$\ldots$$
$$g=\Sigma 2,3,4,5,6,8,9.$$

Опис модуля перетворювача двійкового коду в код семисегментних індикаторів має наступний вигляд:

```
module bin_to_7(y, in); // початок опису модуля
output [6:0] y;         // оголошення вихідного порту
input [3:0] in;         // вхідний порт має розрядність 4
reg [6:0] y;            // вихідний порт має тип регістр
always @ (in)           // конструкція контролю подій
      begin             // при зміні вхідного сигналу виконується цей
блок
            case (in)   // вибираємо одну з  альтернатив
                  4'b0000: y=7'b0111110;
                  4'b0001: y=7'b0000110;
                  4'b0010: y=7'b1011011;
                  4'b0011: y=7'b1001111;
                  4'b0100: y=7'b1100110;
                  4'b0101: y=7'b1101101;
                  4'b0110: y=7'b1111101;
                  4'b0111: y=7'b0000111;
                  4'b1000: y=7'b1111111;
                  4'b1001: y=7'b1101111;
                  4'b1010: y=7'b0110111;
                  4'b1011: y=7'b1111100;
                  4'b1100: y=7'b0111001;
                  4'b1101: y=7'b1011110;
                  4'b1110: y=7'b1111001;
```

```
                4'b1111: y=7'b1110001;
                default: y=0;
            endcase                         // кінець конструкції вибору
        end                                 // always
    endmodule                               // кінець опису модуля
```

Представлений перетворювач коду реагує на будь-яку зміну вхідного сигналу in. Після цього відбувається вибір однієї з альтернатив у конструкції вибору **case** і на виході модуля з'являється відповідний код.

Код Грея, який часто називається циклічним, має ту особливість, що при переході з одного десяткового числа до іншого, сусіднього, у ньому відбувається зміна "0" на "1" або навпаки тільки в одному розряді. Як видно з таблиці код, який представляється двома, трьома або чотирма розрядами, завжди створює циклічну послідовність, тобто адекватну можливість переходу від найстаршого кодового значення числа до наймолодшого. Ця особливість дає можливість використовувати його при кодуванні кутових переміщень в перетворювачах кута повороту в цифровий код. Код Грея знаходить також широке використання в різних перетворювачах "аналог-код", де його властивість дає можливість звести до одиниці молодшого розряду похибку неоднозначності при зчитуванні інформації. Побудувати код Грея безпосередньо з двійкового коду можна, якщо використати наступне правило: і-й біт коду Грея кодового слова встановлюється в нуль, якщо і-й та (і+1)-й біти відповідного двійкового коду однакові. У протилежному випадку біт і=1. В тому випадку, коли (і+1)-й біт виходить за рамки розрядності двійкового коду, його значення приймається рівним нулю. Нижче приводиться текстовий опис перетворювача двійкового коду в код Грея.

```
module decoder_gray(y,in);      // початок модуля
output [3:0] y;                 // визначаємо вихідний порт
input [3:0] in;                 // визначаємо вхідний порт
reg [3:0] y;                    // вхід має тип регістр
always @ (in)                   // початок конструкції контролю подій
    begin                       // при зміні вхідного значення
        case (in)               // вибирається одна з альтернатив
            4'b0000: y=4'd0;
            4'b0001: y=4'd1;
            4'b0010: y=4'd3;
            4'b0011: y=4'd2;
            4'b0100: y=4'd7;
            4'b0101: y=4'd6;
            4'b0110: y=4'd4;
            4'b0111: y=4'd5;
```

```
            4'b1000: y=4'd15;
            4'b1001: y=4'd14;
            4'b1010: y=4'd12;
            4'b1011: y=4'd13;
            4'b1100: y=4'd8;
            4'b1101: y=4'd9;
            4'b1110: y=4'd11;
            4'b1111: y=4'd10;
        endcase          // кінець конструкції вибору
    end
endmodule                // кінець опису модуля
```

В кожному рядку оператора case явно задане значення тільки одного входу модуля. На виході формується сигнал у відповідності до вхідного коду. Оскільки в конструкції перераховані всі можливі комбінації вхідного коду, тому не описується варіант **default**. Аналогічним чином можна описати дешифратор двійкового коду в двійково-десятковий і навпаки. Структура модуля в цілому не зміниться. Потрібно лише замінити значення, які встановлюються на виході модуля у відповідності до таблиці станів. Однак використання такого способу неможливе для багаторозрядних чисел. Наприклад, для 10-бітного слова в конструкції **case** потрібно буде описати 1024 можливі варіанти. Тому необхідно вміти виконувати перетворення двійкового коду будь-якої розрядності у двійково-десятковий. Виконується ця операція у такій послідовності. Оскільки, наприклад, однобайтовим двійковим словом може бути закодована цифра, яка у десятковому коді містить три знаки (до 255), то таке слово повинно бути переміщене з двійкового регістру R відповідними засобами у три чотирьохбітні регістри – регістр одиниць R_O, регістр десятків $R_Д$ і регістр сотень R_C:

$$R_C \leftarrow R_Д \leftarrow R_O \leftarrow R .$$

Виходячи з правил двійкової арифметики, при переміщенні кожного розряду з регістра R у регістри, розміщені зліва, необхідно контролювати значення чисел, що з'являються в регістрах R_C, $R_Д$, R_O. Якщо ці числа дорівнюють або перевищують $5_{10} = 0101_2$, то до них необхідно додати число $3_{10} = 0011_2$. Контролюючи у такий спосіб вміст кожного регістра, за вісім тактів операції зсуву виконується повний цикл перетворення двійкового числа з регістру R у відповідні регістри R_C, $R_Д$, R_O. В табл.2.2 наведено приклад перетворення числа $217_{10} = 1101\ 1001_2$ у двійково-десятковий код шляхом використання операцій зсуву та додавання числа 3_{10}.

Таблиця 2.2. Перетворення у двійково-десятковий код

№ етапу	Регістр сотень $R_С$	Регістр десятків $R_Д$	Регістр одиниць $R_О$	Задане число, регістр R
	0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 1 1 0 0 1
1	0 0 0 0	0 0 0 0	0 0 0 1	
2	0 0 0 0	0 0 0 0	0 0 1 1	
3	0 0 0 0	0 0 0 0	0 1 1 0	
			1 1	
	0 0 0 0	0 0 0 0	1 0 0 1	
4	0 0 0 0	0 0 0 1	0 0 1 1	
5	0 0 0 0	0 0 1 0	0 1 1 1	
			1 1	
	0 0 0 0	0 0 1 0	1 0 1 0	
6	0 0 0 0	0 1 0 1	0 1 0 0	
		1 1		
	0 0 0 0	1 0 0 0	0 1 0 0	
7	0 0 0 1	0 0 0 0	1 0 0 0	
			1 1	
	0 0 0 1	0 0 0 0	1 0 1 1	
8	0 0 1 0	0 0 0 1	0 1 1 1	
	2_{10}	**1_{10}**	**7_{10}**	

Опис модуля дешифратора двійкового коду в двійково-десятковий код для 14-розрядних чисел представлений нижче.

```
module bintobcd(dout0, dout1, dout2, dout3, din); // заголовок модуля
output [3:0] dout0, dout1, dout2, dout3;          // вихідні дані
input [13:0] din;                                 // інформаційний вхід
reg [3:0] dout0, dout1, dout2, dout3; // виході мають регістровий тип
integer i;
reg [29:0] shreg;                     // змінна для виконання алгоритму
```

```
    reg [3:0] temp0, temp1, temp2, temp3; // тимчасові регістри
    always @(din)          // при зміні вхідного коду
    begin                  // виконується алгоритм перетворення
    shreg=0;               // обнуління змінних
    shreg[13:0]=din;       // запис вхідного слова в молодші 14 розрядів
    for(i=0;i<14;i=i+1)    // в циклі виконується операція зсуву
    begin
    shreg=shreg<<1;        // зсув даних на 1 розряд ліворуч
    temp0=shreg[17:14];    // переміщення слова в чотирьохрозрядні
регістри
    temp1=shreg[21:18];
    temp2=shreg[25:22];
    temp3=shreg[29:26];
    if (i<13)              // якщо виконано менше 14 операцій зсуву
    begin                  // перевірка виконання наступних умов
        if (temp0>=5) temp0=temp0+3; // якщо значення регістра більше
        if (temp1>=5) temp1=temp1+3; // або дорівнює 5, додаємо 3
        if (temp2>=5) temp2=temp2+3;
        if (temp3>=5) temp3=temp3+3;
        shreg[17:14]=temp0;        // привласнюємо нові значення
        shreg[29:26]=temp3;        // відповідним бітам у регістрі
        shreg[25:22]=temp2;
        shreg[21:18]=temp1;
    end
    end                            // кінець тіла циклу
    // на виході установлюються дані в двійково-десятковому коді
{dout3,dout2,dout1,dout0}<={shreg[29:26],shreg[25:22],shreg[21:18],shreg[17:
14]};
    end
    endmodule      // кінець опису модуля дешифратора
```

Для виконання операцій зсуву в тілі циклу використовується багаторозрядна регістрова змінна shreg. Оскільки вхідне слово має розрядність 14, а чотири 4-розрядні виходи в сумі дають 16 бітів, то розрядність змінної shreg дорівнює 30. Завдяки доступу до окремих бітів в зазначеному регістрі формуються чотири 4-бітні слова: temp0, temp1, temp2, temp3. Значення в цих змінних порівнюються з 5 і при необхідності до них додається цифра 3. Відкориговані дані встановлюються у відповідні позиції змінної shreg. Після виконання перетворення в двійково-десятковий код, дані встановлюються на виходах модуля. Для компактного запису операції присвоєння був використаний оператор об'єднання (фігурні дужки {}). На вихідних портах dout0, dout1, dout2, dout3 встановлюються значення одиниць, десятків, сотень і тисяч відповідно. На

рис.2.12 представлена часова діаграма, яка демонструє роботу описаного дешифратора.

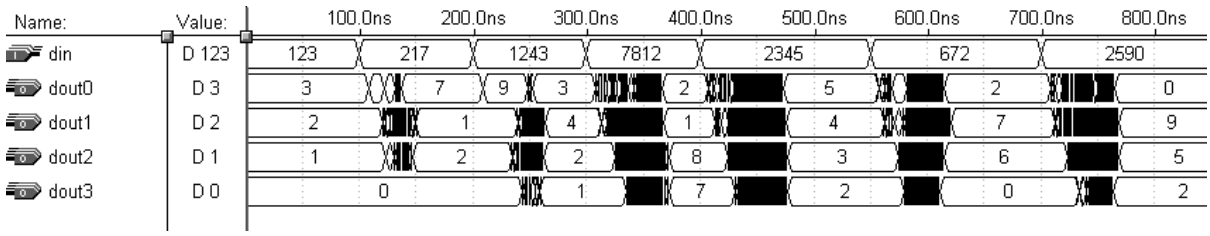

Рис.2.12. Діаграма роботи дешифратора

На діаграмі видно, що під час зміни вхідного коду виникає довготривалий перехідний процес. Якщо керування світлодіодами індикаторами відбувається безпосередньо програмованою матрицею, цей перехідний процес буде непомітним. Однак якщо на виході є паралельні регістри, то подібні явища можуть призвести до неправильного відображення даних. Тому бажано мати тактовий сигнал, по якому в регістри буде записуватися дешифроване слово, і цей тактовий сигнал повинен бути затриманий від моменту зміни вхідного коду модуля на проміжок часу, що дорівнює тривалості перехідного процесу. Недоліком також є те, що такий дешифратор потребує використання великої кількості ресурсів мікросхеми (69% логічних комірок для мікросхеми EPF8282ALC84-2 сімейства FLEX8000). В параграфі „Оптимізація синтезу” приводиться приклад розробки подібного дешифратора, який має дещо кращі характеристики.

Широке використання дешифратори знаходять в цифрових пристроях криптографічного захисту інформації. Спосіб перестановки за допомогою внутрішніх комутаторів дає можливість для *n*-входового перетворювача створити *n*! можливих варіантів. Зворотне перетворення інформації забезпечується блоком зі зворотнім законом перестановки. В наступному прикладі описаний модуль, який забезпечує шифрацію та дешифрацію даних шляхом перестановки бітів вхідного слова.

```
module coder(dout, clk, en, iscode, din); // заголовок модуля шифратора
    output [7:0] dout;                    // вихідний порт
    input en, iscode, clk;                // вхідні сигнали керування
    input [7:0] din;                      // вхідний інформаційний вхід
    reg [7:0] dout;                       // вихід має регістровий тип
    always @(posedge clk)                 // процедурний блок
    if(en && iscode)                      // якщо en=1 та iscode=1
    begin                                 // кодування інформації шляхом
        dout[0]<=din[3];                  // перестановки бітів
```

```
        dout[1]<=din[5];
        dout[2]<=din[7];
        dout[3]<=din[1];
        dout[4]<=din[2];
        dout[5]<=din[6];
        dout[6]<=din[0];
        dout[7]<=din[4];
end
else if(en)                 // якщо en=1 та iscode=0
begin                       // декодування інформації перестановкою
        dout[3]<=din[0];    // бітів в зворотному порядку
        dout[5]<=din[1];
        dout[7]<=din[2];
        dout[1]<=din[3];
        dout[2]<=din[4];
        dout[6]<=din[5];
        dout[0]<=din[6];
        dout[4]<=din[7];
end
else if(en==0)              // якщо en=0
        dout<=0;            // на виході встановлюється 0
endmodule                   // кінець опису модуля
```

Модуль має вхід тактового сигналу, по фронту якого відбувається шифрація або дешифрація даних в залежності від рівня сигналу на вході iscode (1 та 0 відповідно). Робота модуля можлива при високому рівні сигналу на вході дозволу en. В протилежному випадку на виході встановлюється значення 0. Часова діаграма, яка представлена на рис.2.13 демонструє роботу модуля в різних режимах.

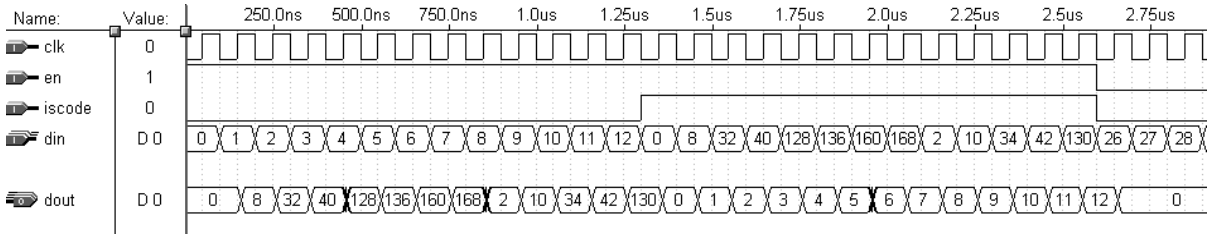

Рис.2.13. Діаграма роботи модуля кодування даних

Спочатку модуль працює в режимі шифратора (iscode=0). На вхід модуля подаються дані для кодування (0, 1, 2, ...). На виході з'являються закодовані дані. При досягненні значення 12 на вході din, модуль перемикається в режим дешифрації і на його вхід подаються закодовані

дані в тій самій послідовності. На виході модуля з'явиться початкова послідовність.

Використання подібних пристроїв дає можливість відкрито передавати і зберігати конфіденційну інформацію.

2.4. Арифметичні пристрої та схеми контролю

До арифметичних пристроїв відносяться суматори, компаратори, арифметично-логічні пристрої, а також схеми для математичної обробки інформації на їх основі. В практиці цифрової схемотехніки існує декілька типів арифметичних пристроїв, наприклад, паралельного, послідовного типу, пристрої із збереженням інформації та інші. В цьому розділі будуть розглядатись лише схеми комбінаційного типу.

2.4.1. Суматори

Суматором називається комбінаційний логічний пристрій, призначений для виконання операції арифметичного додавання чисел, представлених у вигляді двійкових кодів. Суматор є одним з основних вузлів арифметично-логічного пристрою. Класифікація суматорів може бути виконана по різних ознаках. За кількістю виводів розрізняють напівсуматори, однорозрядні суматори, багаторозрядні суматори. Напівсуматором називається пристрій, призначений для додавання двох однорозрядних кодів, який має два входи і два виходи та формуючий із вхідних сигналів сигнали суми і переносу в старший розряд. Багаторозрядним суматором називається пристрій, призначений для додавання двох багаторозрядних кодів, який формує на виході код суми і сигнал переносу у випадку, якщо результат додавання не може бути представлений кодом, розрядність якого збігається з розрядністю кодів доданків.

Розглянемо приклад суматора двох чотирьохрозрядних слів. Текст програми приводиться нижче.

```
module summator(sum,pout,a,b,pin);// початок опису модуля
output [3:0] sum;              // оголошення вихідних портів
output pout;
input [3:0] a;                 // оголошення вхідних портів
input [3:0] b;
input pin;                     // вхід переносу
reg [3:0] sum;
reg pout;
always @(a or b or pin)        // очікуємо зміни сигналів на вході
      begin
      sum = a + b + pin;       // виконуємо операцію додавання
      if (a+b<16) pout = 0;    // якщо сума менша за 16, то
```

```
                              //значення переносу дорівнює 0
        else pout = 1;        // інакше встановлюємо сигнал переносу в 1
        end
endmodule                     // кінець опису модуля
```

В описаному суматорі входами є порти a, b – чотирьохрозрядні слова для виконання операції додавання, pin – вхід переносу. Вихідними є порти sum – сума чисел a та b, pout – вихід переносу. При реалізації віднімання даний вихід може служити знаковим розрядом результату. Часова діаграма роботи модуля суматора приводиться на рис.2.14.

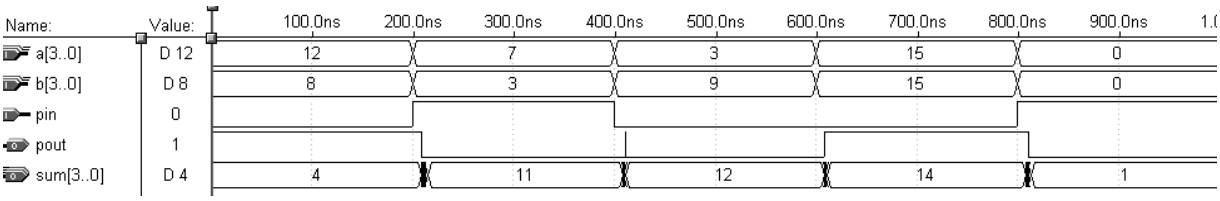

Рис.2.14. Часова діаграма роботи суматора

Проаналізуємо роботу модуля. Якщо на входах a та b присутні числа 12 і 8 відповідно, то при їх додаванні виходить число 20. Однак для представлення цього числа у двійковому коді потрібно 5 розрядів. Тому, на виході встановлюється значення 4 ($12+18-2^4$), і встановлюється біт переносу (pout). Якщо сума менша 15, то значення біту переносу дорівнює 0. В описі модуля відслідковується зміна кожного із сигналів на вході, і при зміні відбувається обчислення суми вхідних слів. На рис.2.15 приводиться таблиця з часовими затримками, які виникають в суматорі.

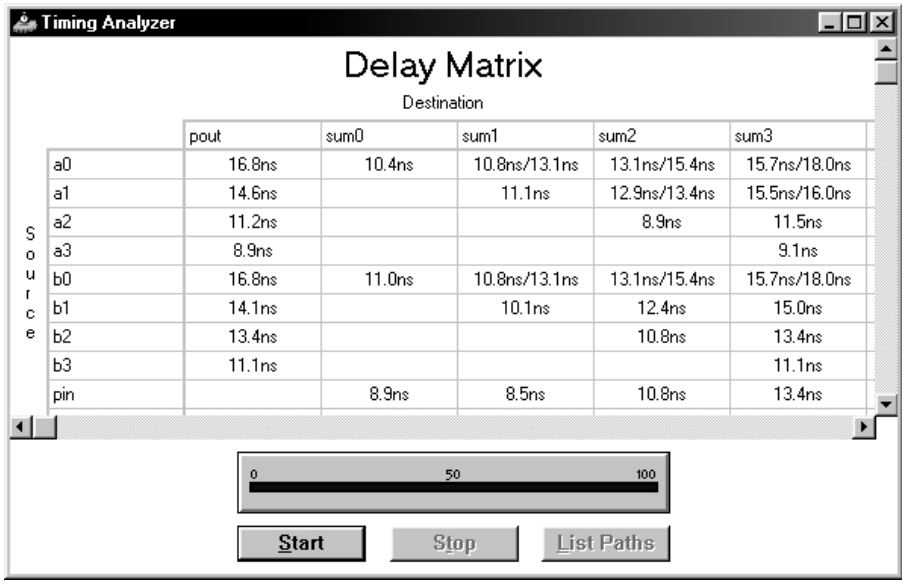

Рис.2.15. Часові затримки при роботі суматора

З таблиці бачимо нерівність значень затримок сигналів від різних входів на вихід. Це призведе до того, що при черговому підрахунку суми на виході буде ”тремтіння” окремих бітів. Для запобігання цього явища рекомендується ставити на вихід суматора паралельний регістр, а в суматор додати вихід готовності результату.

Зверніть увагу на простоту опису суматора. Справа в тому, що ми описали лише поводження модуля, і нічого не сказали про його внутрішню організацію, поклавши це завдання на компілятор системи MAX+plus II. При компіляції проекту система створює (а точніше, підключає до проекту) ще кілька файлів. Це файли опису суматора з бібліотеки параметризованих модулів (LPM) системи, а також опис схем прискореного переносу для забезпечення мінімальних часових затримок (рис.2.16). Багаторозрядні суматори поділяються на послідовні та паралельні. У послідовних суматорах операція додавання виконується послідовно, розряд за розрядом, починаючи з молодшого. У паралельних суматорах всі розряди вхідних кодів додаються одночасно. Розрізняють комбінаційні суматори – пристрої, які не мають власної пам'яті, та накопичуючі суматори з власною внутрішньою пам'яттю, у якій зберігаються результати виконаної операції. При цьому кожний наступний доданок додається до того значення, що містилось у пристрої. По способу тактування розрізняють синхронні та асинхронні суматори. У синхронних суматорах час виконання операції арифметичного додавання двох кодів не залежить від виду самих кодів і завжди залишається постійним. В асинхронних суматорах час виконання операції залежить від вигляду доданків. Тому по завершенню виконання операції додавання необхідно виробляти спеціальний сигнал завершення операції.

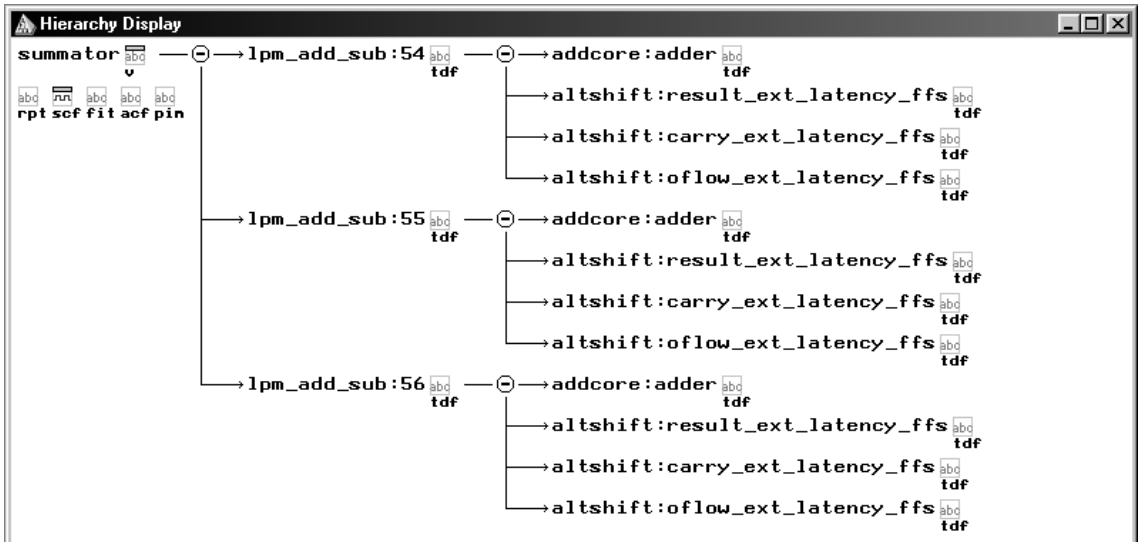

Рис.2.16. Вікно Редактора ієрархії проекту

Verilog допускає велику різноманітність стилів опису об'єктів проекту. В попередньому прикладі був використаний поведінковий опис суматора. Однак існують інші способи функціонального опису об'єктів, більш близькі до традиційних мов програмування. Розглянемо приклад поведінкового опису *n*-розрядного суматора з послідовним переносом.

```
module adder(a,b,cin,sum,cout);        // заголовок модуля
parameter n=10;                        // розрядність вхідних даних
input [n:1] a,b;                       // вхідні порти
input cin;                             // вхід попереднього переносу
output [n:1] sum;                      // вихід суми
output cout;                           // вихід переносу
integer i;                             // змінна для використання в циклі
reg [n:1] vsum;                        // додатковий регістр
reg carry;                             // змінна для зберігання біту переносу
always @(a or b or cin)                // процедурний блок
begin
      carry=cin;                       // запам'ятовуємо попередній перенос
      for (i=1;i<=n;i=i+1)             // виконується цикл n разів
      begin
      vsum[i]=(a[i]^b[i])^carry;       // формування кожного біту суми
      carry=(a[i] & b[i])|(carry & (a[i]|b[i])); // формування переносу
end
end
assign sum=vsum;                       // установка на виході значення суми
assign cout=carry;                     // установка переносу
endmodule                              // кінець опису модуля
```

В наведеному описі використовується цикл, в тілі якого відбувається додавання по модулю 2 (операція ВИКЛ.АБО над бітами) окремих бітів кожного з вхідних слів a та b, з урахуванням біту переносу cin. За межами процедурного блоку **always** відбувається установка обчисленого значення суми на вихідному порту. Також формується біт переносу. Параметр n визначає розрядність вхідних слів.

Аналогічний суматор можна отримати при використанні оператора об'єднання. При цьому значно скорочується об'єм коду:

```
module adder(a,b,cin,sum,cout);        // заголовок модуля суматора
parameter n=10;                        // параметр – розрядність
вхідних даних
input [n:1] a,b;                       // вхідні дані
input cin;                             // вхід переносу
output [n:1] sum;                      // вихідний порт суми двох слів
```

```
output cout;                    // вихід переносу
assign {cout,sum}=a+b+cin;      // визначення суми та значення біту
переносу
endmodule                       // кінець опису модуля
```

В цьому прикладі описаний *n*-розрядний суматор. Що стосується переносу, то в даному випадку важко сказати який він – паралельний чи послідовний. Суматор відрізняється дещо гіршими часовими характеристиками, а при синтезі використовує більше ресурсів мікросхеми в порівнянні з попереднім прикладом.

2.4.2. Компаратори

Компаратори виконують порівняння двох чисел, представлених у вигляді двійкових кодів. Мікросхеми компараторів визначають не тільки рівність, але і нерівність двох чисел. Для цього мікросхема має три виходи: “A>B”, “A<B” і “A=B”, на одному з яких в залежності від співвідношення величин A=$a_3a_2a_1a_0$ та B=$b_3b_2b_1b_0$ з'являється активний рівень сигналу. При описі компаратора на Verilog розробник може включати в свій модуль такі виводи, які йому необхідні (наприклад вихід, логічна одиниця на якому встановлюється якщо на вході значення слів розрізняються на 1, тобто майже рівні). Пристрої порівняння на рівність будуються на основі порозрядних операцій над однойменними розрядами обох слів. Побудувати багаторозрядний компаратор можна на основі суматора. У відповідності до законів арифметики, при A=B на виходах всіх розрядів суматора s_0, s_1, s_2, s_3 буде 0 при $p_i+1=1$. При A>B значення 1 буде як на виході переносу $p_i+1=1$, так і хоча б на одному виході s_i. Тому ознакою A>B може бути функція $y=(p_i+1)\cdot(\sum s_i)$. При A>B результат ознаки переносу $p_i+1=0$.

Компаратори широко використовуються в інформаційних системах для виділення необхідного слова в потоці цифрової інформації, для відмітки часу в часових пристроях, для виконання умовних переходів в обчислювальних пристроях. В пристроях автоматики компаратори використовуються для контролю виходу величин за припустимі межі, в приводах слідкуючих систем автоматики і т.д.

Розглянемо опис компаратора, який забезпечує порівняння двох 8-розрядних чисел по трьох ознаках.

```
module comparator(Agt, Alt, Aeq, A, B);   // початок опису модуля
output Agt, Alt, Aeq;             // оголошення вихідних портів
input [7:0] A;                    // оголошення вхідних портів
input [7:0] B;
reg Agt, Alt, Aeq;                // вихідні порти мають тип регістр
      always @(A or B)            // очікування зміни вхідних сигналів
      begin
```

```
        if (A>B)                    // якщо виконується умова A>B
        begin
        Agt = 1'b1;   // установлюємо потрібні значення сигналів на
        Alt = 1'b0;                 // відповідних виходах
        Aeq = 1'b0;
        end
        else if (A<B)               // якщо виконується умова A<B
        begin
        Agt = 1'b0;
        Alt = 1'b1;                 // установлюємо в 1  сигнал „менше”
        Aeq = 1'b0;
        end
        else                        // якщо A=B
        begin
        Agt = 1'b0;
        Alt = 1'b0;
        Aeq = 1'b1;                 // установлюємо в 1  сигнал „рівні”
        end
        end
    endmodule                       // кінець опису модуля
```

Проаналізувавши програму можна бачити, що ми, як і при описі суматора, використали поведінковий опис пристрою. В даному прикладі демонструється використання операторів порівняння в конструкції **if**. Вихідні порти мають тип регістр, тому при призначенні їм відповідних значень не пишеться ключове слово **assign** (якщо його написати, то воно просто буде проігнороване компілятором). Діаграма на рис.2.17 демонструє роботу приведеного опису компаратора.

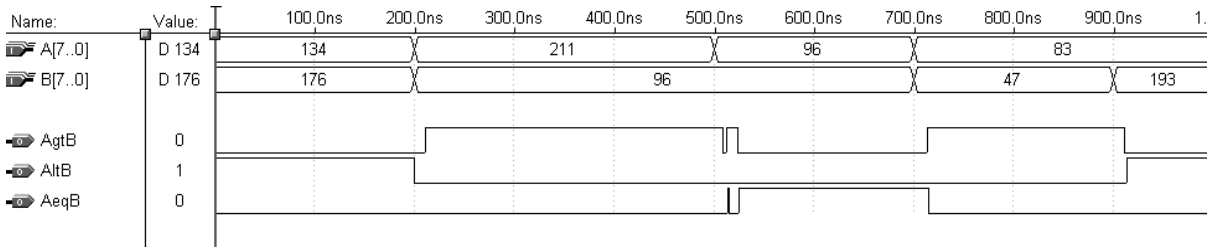

Рис.2.17. Часова діаграма роботи компаратора

Використовуючи параметризовані модулі, розробник зможе вибирати розрядність суматора, а також необхідні ознаки порівняння. На приведеній часовій діаграмі в моменти зміни сигналів на входах компаратора, на виходах виникають паразитні імпульси. Один з шляхів вирішення цієї проблеми полягає у введені синхронізуючого імпульсу.

Наступний приклад представляє собою модуль компаратора з тактуючим входом.

```
module synccomp(aeb, agb, alb, clk, A, B); // заголовок модуля
parameter WIDTH=8;                  // розрядність даних
output aeb,agb,alb;                 // вихідні сигнали
input clk;                          // тактовий сигнал
input [WIDTH-1:0] A;                // вхідний порт A
input [WIDTH-1:0] B;                // вхідний порт B
reg aeb,agb,alb;                    // виходи мають регістровий тип
always @(posedge clk)               // по фронту сигналу clk
      begin
      // на виходах встановлюємо відповідні рівні сигналів
      {aeb,agb,alb}=(A>B)?(3'b010):((A==B)?3'b100:3'b001);
      end
endmodule                           // кінець опису модуля
```

По функціональним можливостям приведений опис модуля компаратора не відрізняється від попереднього. Однак введення параметра WIDTH придає гнучкості модулю, оскільки компаратор може порівнювати значення вхідних кодів довільної розрядності (при зміні значення цього параметру). Шляхом використання оператора об'єднання {} та тернарного оператора ?: вдалося скоротити розмір опису безпосередньо алгоритму роботи компаратора до однієї строчки. Після моделювання роботи модуля на часовій діаграмі (рис.2.18) відсутні паразитні викиди сигналів.

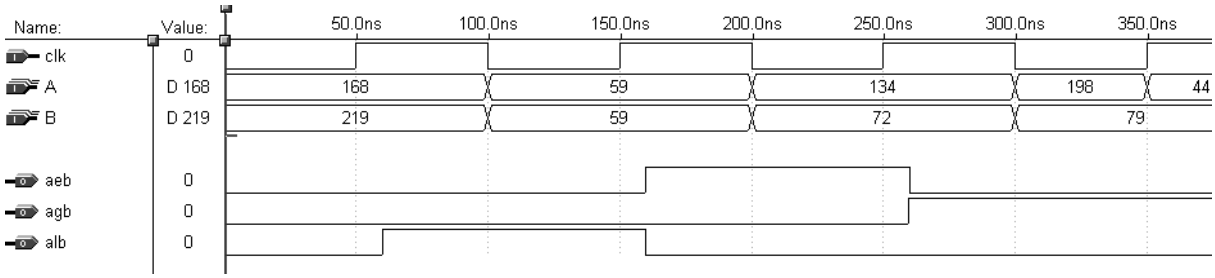

Рис.2.18. Діаграма роботи компаратора

З часової діаграми видно, що на виходах змінюються сигнали по фронту тактового сигналу clk. Якщо значення на вході A більше за B, та виході agb встановлюється сигнал високого рівня. При рівності цих значень високий рівень буде мати вихід aeb. Якщо ж значення B більше за A, на виході alb встановиться лог.1.

Наступний приклад демонструє використання параметризованого модуля компаратора з бібліотеки САПР.

```
module comptest (A, B, eq, gt, lt);     // заголовок модуля
input [7:0]  A;                         // вхідні порти
input [7:0]  B;
output eq, gt, lt;                      // виходи модуля
wire weq, wgt, wlt;                     // додаткові ланцюги
wire eq = weq;                          // з'єднання ланцюгів з виходами порту
wire gt = wgt;
wire lt = wlt;
lpm_compare lpm_compare_component (.dataa (A),.datab (B),
      .aeb(weq), .agb(wgt), .alb(wlt));     // підключення компаратора
defparam                                    // визначення параметрів
lpm_compare_component.lpm_width = 8,    // розрядність шини даних
lpm_compare_component.lpm_representation = "UNSIGNED",
lpm_compare_component.lpm_pipeline = 0,
lpm_compare_component.lpm_hint = "UNUSED",
lpm_compare_component.one_input_is_constant = "NO";
endmodule                               // кінець опису модуля
```

Параметризований модуль компаратора має шість виходів, але в наведеному прикладі використані лише три сигнали: „A=B” (eq), „A>B” (gt), „A<B” (lt). Доступ до відповідного порту виконується за допомогою складених ідентифікаторів. Після підключення модуля вказуються значення параметрів, опис яких приводиться в табл.2.3.

Таблиця 2.3. Опис параметрів компаратора

Найменування параметру	Опис параметру
lpm_width	Розрядність портів dataa і datab.
lpm_representation	Форма представлення: SIGNED" – зі знаком, "UNSIGNED" – без знаку, або "UNUSED" – не використовується.
lpm_pipeline	Визначає кількість тактів генератора при використанні режиму конвеєра. Якщо дорівнює 0, то синтезується комбінаційна схема.
lpm_hint	Дозволяє задати параметри VHDL, специфічні для САПР.
one_input_is_constant	На одному з входів встановлено постійне значення. Можливі значення „YES”, „NO” або „UNUSED”.

Часова діаграма роботи приведеного опису компаратора приводиться на рис.2.19.

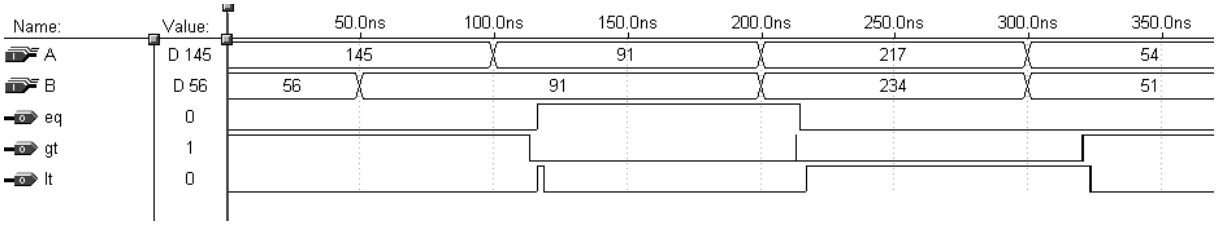

Рис.2.19. Діаграма роботи компаратора

З діаграми роботи компаратора видно, що в залежності від значень кодів на входах, на виходах встановлюються сигнали відповідних рівнів.

2.4.3. Арифметично-логічні пристрої (АЛП)

Арифметично-логічні пристрої – це спеціалізовані мікросхеми, які виконують арифметичні та логічні операції у відповідності до двійкового коду, який подається на керуючі входи мікросхеми. Описуючи арифметично-логічний пристрій мовою Verilog, розробник має можливість описати тільки ті операції, які необхідні для побудови конкретної системи. Таким чином, не будуть використовуватися "дарма" ресурси програмованої мікросхеми. Це вигідно відрізняє текстовий опис модуля від використання готових елементів з бібліотеки, які мають значні можливості, які часом зовсім не використовуються.

Розглянемо невеликий приклад. Запропонований арифметично-логічний пристрій буде оперувати чотирьохрозрядними словами і мати три входи вибору операції. Пристрій виконує чотири логічні та чотири арифметичні операції. Опис модуля АЛП представлений нижче.

```
module alu(F, Pout, Sel, A, B);  // початок опису пристрою
output [3:0] F;                  // опис вихідних портів
output Pout;                     // вихід переносу
input [2:0] Sel;                 // опис вхідних портів
input [3:0] A;                   // вхід A даних
input [3:0] B;                   // вхід B даних
reg [3:0] F;                     // вихідні порти мають тип "регістр"
reg Pout;
always @ (A or B or Sel)         // стежимо за зміною вхідних сигналів
begin
case (Sel)                       // початок конструкції вибору
        4'b000 : F=~A;           // інверсія значення A
        4'b001 : F=A&B;          // операція побітового I
        4'b010 : F=~(A^B);       // інверсія
        4'b011 : F=&B;           // згортка по I
```

```
        4'b100 : {Pout,F}=A+B; // сума
        4'b101 : {Pout,F}=A+(~B);
        4'b110 : {Pout,F}=(A^B)+(A-B);
        4'b111 : {Pout,F}=2*B;  // зсув вліво на один розряд
        default : F=0;
    endcase
    end
    endmodule                    // кінець опису модуля
```

У конструкції case відбувається вибір однієї з альтернатив в залежності від стану сигналів Sel[2], Sel[1], Sel[0]. Розробник у кожній з альтернатив може описати той алгоритм, який йому потрібний. Тобто можна побудувати пристрій АЛП під конкретну задачу. У програмі також використаний оператор об'єднання розрядів вихідних портів. Це зроблено для зручності призначення цим портам сигналів. Замість оператору об’єднання можна було б використати наступну форму запису:

```
…
  4'b100 :
  begin                          // початок блоку
  F = A + B;                     // операція додавання
  if ((A + B) > 15) Pout = 1;    // при виконанні умови встановлюємо перенос
  else Pout = 0;                 // очистка переносу
  end
…
```

Однак такий опис трохи громіздкий. На рис.2.20 приводиться часова діаграма роботи АЛП.

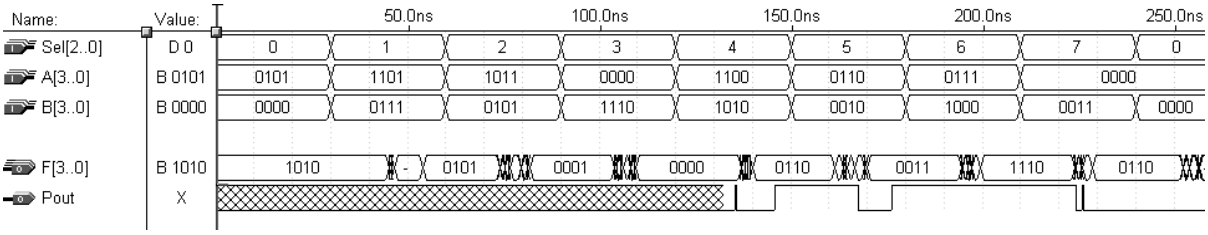

Рис.2.20. Часова діаграма роботи АЛП

Проаналізувавши діаграму, можна переконатися в правильності роботи пристрою. Дійсно, коли на вході вибору Sel присутній 0, на виході встановлюється інвертоване значення A. Код 011 на вході Sel викликає виконання операції згортки по I над значенням B. Оскільки тільки при умові рівності 1 всіх розрядів числа B результатом операції згортки по I буде лог.1, то в цьому випадку на виході встановлюється значення 0. Значення Pout до першого присвоєння в програмі залишається

невизначеним. Для того, щоб сигнал переносу був встановлений в 0 при виконанні операцій, які не впливають на стан цього виводу, перед початком конструкції case потрібно добавити строчку наступного вигляду: Pout = 0;. Тип виконуваної операції відповідає коду на вході Sel.

2.4.4. Контроль парності

Незважаючи на високу надійність та перешкодостійкість цифрових систем передачі інформації, вірогідність появи помилок завжди існує. Тому всі високонадійні канали передачі інформації забезпечуються допоміжними схемами, які дають можливість впевнитись у відсутності помилок в прийнятих даних. В будь-якій структурі каналу передачі інформації з контролем помилок повинна бути надмірність каналу. Наприклад, якщо передається код $0101_2=5_{10}$ і на виході каналу з'являється $1101_2=13_{10}$, то в загальному плані помилку без спеціальних перевірок визначити неможливо. Але якщо ми знаємо, що інформація передається у двійково-десятковому коді, то одержаний результат хибний. Тобто, наявність шести надлишкових станів дає можливість виявити деякі помилки.

Простий спосіб визначення помилок у словах базується на припущенні, що найбільша вірогідність збою можлива тільки в одному біті, тобто в появі помилкової одиниці або нуля. Тому для визначення наявності такого збою використовують контроль парності або непарності одиниць у слові, що передається (parity check). В основі цього способу лежить операція знаходження суми по модулю 2 всіх двійкових розрядів контрольованого слова. При парній кількості одиниць вказана сума дорівнює 0, а при непарній – 1. Рівняння, за яким підраховується кількість одиниць, має вигляд:

$$Y = x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus v = y \oplus v\,.$$

З аналізу рівняння витікає, що при парній кількості одиниць у слові $x_3x_2x_1x_0$ значення виходу y=0. При непарній – y=1. Опис модуля визначення парності має вигляд:

```
module parity(y, v, in);  // початок опису модуля
output y;                 // оголошення вихідних портів
input v;                  // оголошення вхідних портів
input [7:0] in;
reg y;
always @ (v or in)        // очікуємо зміни сигналів на вхідних портах
      begin
      if(v)          // якщо v  має високий рівень
      y=(^in);       // виконуємо операцію згортки по ВИКЛ.АБО
      else
      y=~(^in);      // інакше встановлюємо на виході інверсне значення
```

```
                    // результатату операції згортки
        end
endmodule           // кінець опису модуля
```

При зміні будь-якого сигналу на вході відбувається операція згортки вхідного байту по ВИКЛ.АБО. Значення „*y*” залежить від сигналу „*v*”, який задає режим контролю – контроль парності або контроль непарності. При контролі парності задається *v*=0. Якщо відбудеться контроль по непарній кількості одиниць, то задається *v*=1. Вихід „*y*” називається контрольним бітом. При прийомі інформації одержане слово знову перевіряється на парність або непарність. Якщо при контролі парності в приймальному контрольному пристрої з'явилась 1, то це означає, що або в інформаційних даних, або в контрольному біті виникла помилка. Як бачимо, такий простий контроль не дає можливості виправити помилку, але він дає можливість визначитись з хибною інформацією, щоб потім або використати її, або відкоригувати.

В практиці контролю переважно використовують спосіб контролю по непарності. Пов’язано це з тим, що при появі слова з усіма нулями при контролі по парності цю ситуацію неможливо відрізнити від обриву лінії зв’язку.

Приведена схема не може визначити подвійної помилки і будь-якої кількості парних помилок. Пояснюється це лише малою надмірністю лінії зв’язку. Тому для більш глибокого контролю необхідно мати і більшу її надмірність.

При побудові мереж доводиться кодувати передані дані тим або іншим способом. Це обумовлено декількома причинами. Однією з них є присутність у переданій послідовності постійної складової. Тому елементи гальванічної розв'язки (наприклад, трансформатори) не зможуть передати довгу послідовність нулів або одиниць. Крім того, бажано забезпечити дані деякою надмірністю для виявлення помилок (більш ніж в одному розряді). Інакше кажучи, необхідно виконати логічне кодування. Тут варто підкреслити, що на етапі логічного кодування вже не формується форма сигналів. Тут просто борються з недоліками методів фізичного цифрового кодування – відсутність синхронізації, наявність постійної складової. Таким чином, спочатку за допомогою засобів логічного кодування формуються виправлені послідовності двійкових даних, які потім за допомогою методів фізичного кодування передаються по лініях зв'язку. Логічне кодування має на увазі заміну біт вхідної інформації новою послідовністю біт, що несе таку саму інформацію, але крім цього має додаткові властивості, наприклад можливістю для приймальної сторони виявляти помилки в прийнятих даних. Супроводження кожного байта вихідної інформації одним бітом парності – це приклад способу логічного кодування який часто використовується при передачі даних за допомогою

модемів. Іншим прикладом логічного кодування може служити шифрация даних, що забезпечує їх конфіденційність при передачі через загальні канали зв'язку. Одним з методів логічного кодування є застосування надлишкових кодів. Надлишкові коди засновані на розбивці вихідної послідовності біт на порції, які часто називають символами. Потім кожен вихідний символ замінюється на новий, який має більшу кількість біт. Один з прикладів надлишкового коду – логічний код 4B/5B. Логічний код 4B/5B заміняє вихідні символи довжиною в 4 біти на символи довжиною в 5 біт. Оскільки результуючі символи містять надлишкові біти, то загальна кількість бітових комбінацій в них більша, ніж у вхідній. Таким чином, п’ятибітова схема дає 32 (2^5) дворозрядних літерно-цифрових символів, які мають значення в десятковому коді від 0 до 31, в той час як вхідні дані можуть містити тільки чотири біти або 16 (2^4) символів. Тому в результуючому коді можна підібрати 16 таких комбінацій, які не містять великої кількості нулів, а інші вважати забороненими кодами (code violation). Очевидно, що в такому випадку довгі послідовності нулів перериваються. Зникає постійна складова, звужується спектр сигналу. Але цей метод знижує корисну пропускну здатність лінії, тому що надлишкові біти інформації не несуть, і тільки "забирають ефірний час". Надлишкові коди дозволяють приймачу розпізнавати помилкові біти. Якщо приймач приймає заборонений код, то це означає, що на лінії відбулося пошкодження даних. Комбінації, які мають більше трьох нулів (01–00001, 02–00010, 03–00011, 08–01000, 16–10000), інтерпретуються символом V і командою приймача VIOLATION – збій. Команда означає наявність помилки через високий рівень перешкод або збій передавача. Єдина комбінація з п'яти нулів (00–00000) ставиться до службових сигналів, означає символ Q і має статус QUIET – відсутність сигналу в лінії.

Таке кодування даних вирішує два завдання – синхронізації і поліпшення перешкодостійкості. Синхронізація відбувається за рахунок виключення послідовності більше трьох нулів, а висока перешкодостійкість досягається приймачем даних на п’ятибітовому інтервалі. Ціна за ці переваги при такому способі кодування даних – зниження швидкості передачі корисної інформації. Наприклад, в результаті додавання одного надлишкового біта на чотири інформаційних, ефективність використання смуги частот в протоколах з кодом MLT-3 і кодуванням даних 4B/5B зменшується відповідно на 25%. Схема кодування 4B/5B представлена в табл.2.4.

Таблиця 2.4. Схема кодування 4B/5B

Двійковий код 4B	Результуючий код 5B	Двійковий код 4B	Результуючий код 5B
0000	11110	1000	10010

0001	01001	1001	10011
0010	10100	1010	10110
0011	10101	1011	10111
0100	01010	1100	11010
0101	01011	1101	11011
0110	01110	1110	11100
0111	01111	1111	11101

Отже, відповідно до цієї таблиці формується код 4B/5B, потім передається по лінії за допомогою фізичного кодування по одному з методів потенційного кодування, чутливому тільки до довгих послідовностей нулів – наприклад, за допомогою цифрового коду NRZI. Символи коду 4B/5B довжиною 5 біт гарантують, що при будь-якому їхньому сполученні на лінії не можуть зустрітися більше трьох нулів підряд. Спробуємо описати пристрій, який виконує логічне кодування даних. Опис модуля має вигляд:

```
module coder(y, v, in);          // початок опису модуля
output [4:0] y;     // оголошення вихідних портів
input v;            // оголошення вхідних портів
input [3:0] in;
reg [4:0] y;
always @ (y or v)  // очікуємо зміну сигналів на входах
        begin
        if(v)          // якщо встановлено сигнал v
        case (in)      // вибираємо значення з декількох альтернатив
        4'b0000 : y=5'b11110;
        4'b0001 : y=5'b01001;
        4'b0010 : y=5'b10100;
        4'b0011 : y=5'b10101;
        4'b0100 : y=5'b01010;
        4'b0101 : y=5'b01011;
        4'b0110 : y=5'b01110;
        4'b0111 : y=5'b01111;
        4'b1000 : y=5'b10010;
        4'b1001 : y=5'b10011;
        4'b1010 : y=5'b10110;
        4'b1011 : y=5'b10111;
        4'b1100 : y=5'b11010;
        4'b1101 : y=5'b11011;
```

```
        4'b1110 : y=5'b11100;
        4'b1111 : y=5'b11101;
        endcase          // кінець конструкції вибору
        else y=0;        // встановлюємо на виході 0 якщо v=0
end
endmodule                // кінець опису модуля
```

Описаний пристрій працює по вже знайомому нам принципу: при зміні сигналів на вході відбувається заміна чотирьохрозрядного вхідного слова на п’ятирозрядний відповідно до таблиці. Як бачимо, алгоритм роботи такого пристрою досить простий.

2.4.5. Перемножувачі та поділювачі

Ідеологія перемноження двох бінарних слів полягає у використанні операцій додавання і зсуву проміжної суми. Реальні чотирьохрозрядні перемножувачі використовують просту технологію, відповідно до якої для двох слів A та B, які необхідно перемножити, створюється таблиця істинності з вихідним словом C подвійної довжини. Кожен розряд c_i слова C виступає логічною функцією з логічними змінними слів A та B. Тому реалізація чотирьохрозрядного перемножувача є простою реалізацією 8-ми логічних функцій. При необхідності реалізувати перемножувач двох восьмирозрядних слів кожне з них розбивається на групи по 4 біти і з кожною з груп виконуються операції, як з однією змінною, за принципом знаходження проміжної суми з послідуючим виконання операції зсуву. Розглянемо приклад. Необхідно розробити модуль підрахунку факторіала. Найпростіше було б використати згадану технологію побудови перемножувачів, тобто створити таблицю істинності і за допомогою оператора вибору **case** реалізувати модуль. Але для демонстрації деяких можливостей мови вирішимо це завдання дещо інакше. Опис модуля підрахунку факторіалу має вигляд:

```
module factor(result, ready, in, clk, start);   // початок опису модуля
output [7:0] result;            // оголошення вихідної шини
output ready;                   // вихід готовності даних
input [2:0] in;                 // вхід даних
input clk, start;               // входи тактового сигналу та запуску
reg [7:0] result;               // вихід має регістровий тип
reg ready;
reg [2:0] temp;                 // додаткові змінні
reg [7:0] count;
always @ (posedge clk) begin
// очікування фронту тактового сигналу
        if (start==1)           // якщо умова виконується
```

```
        begin
        temp=in;            // присвоюємо тимчасовому регістру
                            // значення на вході
        count=8'b00000001;// установлюємо значення змінної
        ready=0;            // обнуління сигналу готовності результату
        end
        else begin
        count=count*temp;       // операція множення
        temp=temp-1;        // зменшення значення тимчасового регістра
        if (temp<3'b010)  // якщо значення регістра менше 2
              begin
              result=count;     // установка результату на виході
              ready=1;          // установка сигналу готовності
              end
              end
              end
endmodule                   // кінець опису модуля
```

Модуль працює наступним чином. При появі фронту синхроімпульсу виконується перевірка входу start. При високому рівні сигналу на цьому вході в регістр temp записується значення того числа, факторіал якого необхідно визначити. На виході result зберігається попереднє значення. Сигнал готовності результату ready має низький рівень. Якщо на вході start присутній лог.0, то починається визначення факторіалу. Особливість роботи модуля полягає в тому, що для підрахунку результату знадобиться кількість тактів генератора, яка дорівнює значенню вхідного слова. Тобто для підрахунку значення факторіалу цифри 5 знадобиться п'ять тактів. Коли результат готовий, вихід ready встановлюється в рівень лог.1. Роботу модуля демонструє часова діаграма, що приводитися на рис.2.21.

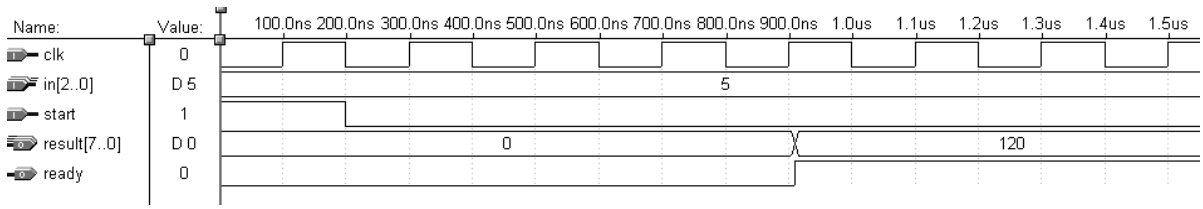

Рис.2.21. Підрахунок факторіалу

Спробуємо описати модуль, який підраховував би значення наступної функції:

$$y=a\cdot x+b\cdot x.$$

Опис модуля має наступний вигляд:

```
module func(y, x, r);    // початок опису модуля
output y;                // оголошення вихідних портів
input x,r;               // оголошення вхідних портів
wire [3:0] x;
reg [3:0] a;  // оголошення додаткових змінних регістрового типу
reg [3:0] b;
reg [11:0] y;            // розрядність вихідного порту дорівнює 12
      always @ (x or r) // очікуємо зміни сигналів на вході
      begin
      a=4'd3;       // присвоюємо змінній значення 3
      b=4'd7;       // присвоюємо змінній значення 7
     if (r==1) y=a*x+b*x; // якщо r=1 обчислюємо значення функції
     end
endmodule                // кінець опису модуля
```

У конструкції контролю подій **always** відбувається стеження за зміною кожного з сигналів на вході. Коли відбувається зміна стану будь-якого з вказаних сигналів, на виході „y” встановлюється значення функції при умові, що на вході r присутня лог.1. Взагалі, змінні „a” та „b” можна в даному прикладі зробити змінними типу **integer**, оскільки вони використовуються лише в тілі модуля.

Цей приклад демонструє зручність опису подібних конструкції, не вдаючись в особливості їх апаратної реалізації. Це завдання виконує компілятор. Для мікросхеми сімейства FLEX8000 EPF8282ALC84-2, на реалізацію описаної функції знадобилося 14% логічних комірок, а для реалізації модуля підрахунку факторіалу цей показник становить 37%.

У бібліотеці параметризованих модулів систем MAX+plus II та Quartus є перемножувач lpm_mult. Цей модуль зручно використовувати якщо відбувається проектування як за допомогою опису проекту на одній з HDL-мов, так і у Графічному редакторі. Використання параметризованого модуля в якості перемножувача двох п'ятирозрядних слів демонструє наступний приклад.

```
module multipler4x4 (dataa, datab, result);  // початок опису модуля
input [4:0]  dataa;                    // вхідні шини даних
input [4:0]  datab;
output [9:0]  result;                  // результат множення
wire [9:0] sub_wire0;                  // додатковий ланцюг
wire [9:0] result = sub_wire0[9:0];// з'єднання шини даних з ланцюгом
lpm_mult     lpm_mult_component (.dataa (dataa), .datab (datab),
.result (sub_wire0));
// використовуємо параметризований модуль з бібліотеки
defparam           // встановлення значень параметрів
```

```
        lpm_mult_component.lpm_widtha = 5,
        lpm_mult_component.lpm_widthb = 5,
        lpm_mult_component.lpm_widthp = 10,
        lpm_mult_component.lpm_widths = 10,
        lpm_mult_component.input_b_is_constant = "NO",
        lpm_mult_component.lpm_representation = "UNSIGNED",
        lpm_mult_component.use_eab = "OFF";
endmodule          // кінець опису модуля
```

Для використання модуля потрібно встановити деякі параметри. Для доступу до параметрів модуля використовуються складені імена. В табл.2.5 приводиться опис параметрів модуля перемножувача.

Таблиця 2.5. Опис параметрів модуля перемножувача

Найменування параметру	Опис параметру
lpm_widtha, lpm_widthb	Розрядність вхідних портів dataa та datab відповідно
lpm_widthp	Розрядність вихідних даних (порт result)
lpm_widths	Розрядність порту sum. Його потрібно вказувати навіть якщо цей порт не використовується
input_b_is_constant	Установка значення “YES” дозволяє перемножувати дані на константу. Це дозволяє оптимізувати модуль по швидкості та зайнятим ресурсам.
lpm_representation	Тип перемноження – знаковий (SIGNED) або беззнаковий (UNSUGNED).
use_eab	Установка параметра в “ON” дозволить компілятору використовувати вбудовані блоки при синтезі на мікросхемах сімейства ACEX 1K.
maximize_speed	Може приймати значення від 0 до 10. Якщо цей параметр використовується, компілятор намагається виконати оптимізацію модуля по швидкості, не зважаючи на розмір використаних ресурсів

	мікросхеми.
lpm_pipeline	Дозволяє використовувати тактовий сигнал. Результат встановиться на виході після деякої кількості фронтів тактового сигналу.

Використання параметризованих модулів (а також параметрів у власно розроблених модулях) придає гнучкості в зміні розрядності вхідних та вихідних портів, дозволяє задавати різні режими роботи модуля, використовувати оптимізацію синтезу під конкретну архітектуру мікросхеми.

Часова діаграма, що демонструє роботу перемножувача, приводитися на рис.2.22.

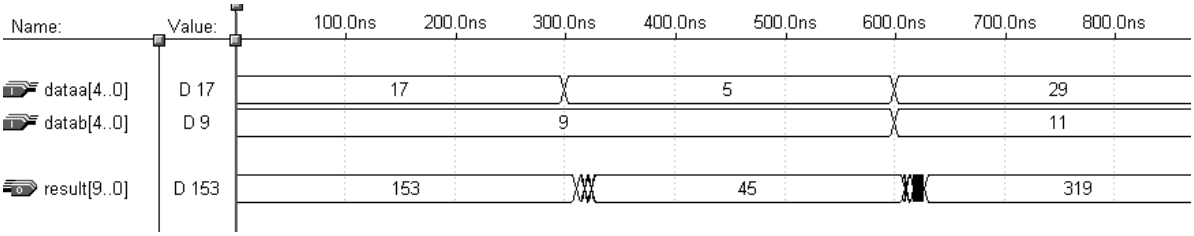

Рис.2.22. Часова діаграма роботи перемножувача

На діаграмі видно, що при зміні даних на вхідних портах, на виході протягом деякого часу відбувається перехідний процес. Це відбувається внаслідок неоднаковості проміжків часу встановлення даних на кожному з розрядів вихідного порту. Для запобігання цього явища, як вже говорилося, можна використовувати регістр на вихідному порту, або тактовий сигнал для самого модуля перемноження.

Наступний приклад описує модуль перемноження з накопиченням.

```
module mult_accum (dataout, dataa, datab, clk, aclr, clken);   // початок опису
input [7:0] dataa;                    // вхідні 8-розрядні шини даних
input [7:0] datab;
input clk;                            // вхід тактового сигналу
input aclr;                           // вхід асинхронного сбросу
input clken;                          // вхід дозволу тактування
output [31:0] dataout;                // вихідна шина даних
reg [31:0] dataout;                   // вихід має регістровий тип
reg [7:0] dataa_reg;                  // додаткові 8-бітні регістри
reg [7:0] datab_reg;
reg [15:0] multa_reg;
wire [15:0] multa;
```

```
wire [31:0] adder_out;
assign multa = dataa_reg * datab_reg; // призначаємо значення добутку
assign adder_out = multa_reg + dataout;      // накопичення даних
always @(posedge clk or posedge aclr)
// по фронту тактового сигналу або сигналу асинхронного сбросу
begin
        if (aclr)               // якщо aclr=1
        begin
        dataa_reg <= 0;     // встановлюємо значення всіх регістрів в 0
        datab_reg <= 0;
        multa_reg <= 0;
        dataout <= 0;
        end
        else if (clken)             // якщо дозволений сигнал тактування
        begin
        dataa_reg <= dataa;         // запам'ятовуємо вхідні дані
        datab_reg <= datab;
        multa_reg <= multa;
        dataout <= adder_out;
        // установлюємо на виході поточне значення
        end
end
endmodule                           // кінець опису модуля
```

По фронту сигналу aclr всі розряди вихідного порту приймають значення нуль. Також обнуляються значення внутрішніх змінних модуля. По фронту тактового сигналу у внутрішні регістри dataa_reg та datab_reg записуються значення на вхідних портах dataa та datab. Значення multa при першому такті, як і значення накопичувача adder_out, дорівнює нулю. При наступному фронті значення multa дорівнює добутку значень на вхідних портах. Це ж значення присвоюється adder_out. По третьому фронту тактового сигналу на виході встановлюється значення adder_out. З кожним фронтом тактового сигналу описана послідовність повторюється. До значення на виході додається величина добутку слів на вхідних портах. Для того, щоб зробити вхід обнуління синхронним, потрібно убрати «**posedge** aclr» зі списку чутливості в конструкції **always**. Часова діаграма на рис.2.23 демонструє роботу модуля перемножувача з накопиченням.

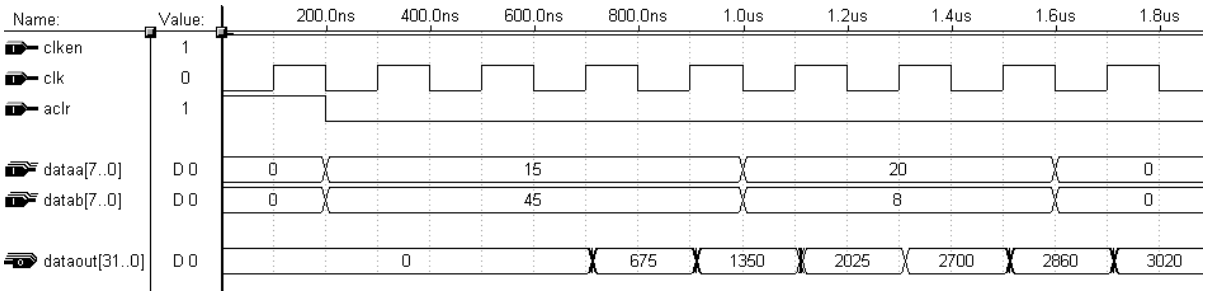

Рис.2.23. Діаграма роботи перемножувача з накопиченням

Часто виникає необхідність визначення частки від ділення двох чисел. Наприклад, при визначенні похідної, або при виконанні апроксимації чи інтерполяції. Оператор ділення (/) не підтримується підмножиною мови Verilog у системах MAX+plus II та Quartus. Але це не означає, що неможливо виконати операцію ділення. Для цього може бути використаний модуль поділювача з бібліотеки системи. Розглянемо приклад використання параметризованого модуля поділювача. Опис модулю має наступний вигляд.

```
module divider (numerator, denominator, quotient, remainder);
input [19:0]  numerator;          // ділене
input [6:0]  denominator;         // дільник
output [19:0]  quotient;          // частне
output [6:0]  remainder;          // залишок
wire [6:0] sub_wire0;             // додаткові ланцюги для підключення
wire [19:0] sub_wire1;            // модуля поділювача
wire [6:0] remainder = sub_wire0[6:0];
wire [19:0] quotient = sub_wire1[19:0];
divide divider1 (.numerator (numerator),     .denominator (denominator),
.remainder (sub_wire0), .quotient (sub_wire1));    // модуль дільника
defparam                          // перевизначення параметрів
        divider1.width_n = 20,
        divider1.width_d = 7,
        divider1.width_q = 20,
        divider1.width_r = 7,
        divider1.width_d_min = 1,
        divider1.lpm_pipeline = 0,
        divider1.pipeline_delay = 0;
endmodule                         // кінець модуля
```

Як і для модуля перемножувача, який було розглянуто вище, для модуля ділення також потрібно вказувати параметри. В табл.2.6 приводитися опис параметрів модуля поділювача.

Таблиця 2.6. Опис параметрів модуля дільника

Найменування параметру	Опис параметру
width_n	Розрядність дільника
width_d	Розрядність поділювача
width_q	Розрядність результату
width_r	Розрядність залишку
width_d_min	Мінімальна розрядність порту поділювача
lpm_pipeline	Дозволяє використовувати тактовий сигнал. Значення на вихідному порту встановиться після визначеної кількості тактів
pipeline_delay	Переміщення конвеєру по етапах. Значення 0 запускає на останньому рівні. Значення WIDTH_Q-1 запускає на першому рівні.

На рис.2.24 приводиться часова діаграма, яка демонструє роботу модуля поділювача.

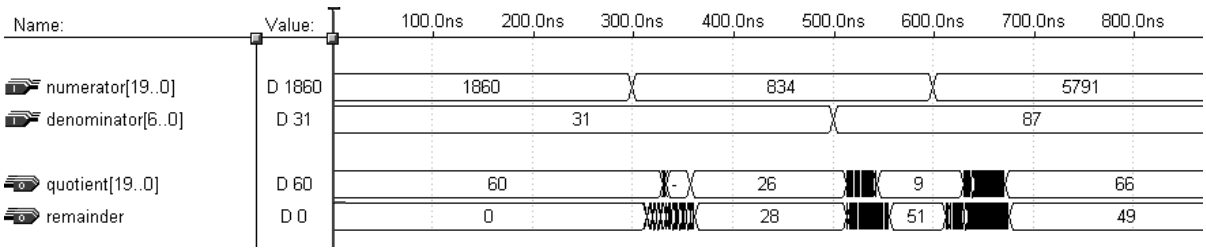

Рис.2.24. Робота модуля поділювача

Як видно з діаграми, при зміні сигналів на вхідних портах, на виході виникає нестабільний стан. Час встановлення значення результату ділення двох чисел та залишку за результатами моделювання складає більш ніж 80 нс для мікросхеми сімейства FLEX10K. Це накладає деякі обмеження на використання модуля.

Операція ділення потребує великих витрат ресурсів мікросхеми. Фрагмент з файлу звіту (divider.rpt), який створюється при компіляції модулю для мікросхеми EPF10K10ATC100-1, приводитися нижче.

Total dedicated input pins used:	6/6	(100%)
Total I/O pins used:	48/60	(80%)

Total logic cells used:	**242/576**	**(42%)**
Total embedded cells used:	0/24	(0%)
Total EABs used:	0/3	(0%)
Average fan-in:	3.40/4	(85%)
Total fan-in:	823/2304	(35%)

Для реалізації модуля поділювана знадобилося 242 логічних комірки. Це складає 42% від загальної кількості комірок для вибраної мікросхеми.

2.5. Тригери

Комбінаторна логіка синтезується з наступних конструкцій:
1) безперервне присвоєння:

```
assign a=b+c&d;
wire b={e,f} | g[1:0];
```

2) сигнали, описані у функціях;
3) сигнали, описані наступною або подібною конструкцією:

```
reg data_out;
always @(a or b or c)    // процедурний блок
if (b==1)                // перевірка умови
        data_out = a;
else
        data_out = c;
```

В наведеному прикладі використовується блок **always**, у списку чутливості якого перераховані всі вхідні сигнали. При зміні будь-якого з них відбувається виконання процедурного блоку.

Як можна бачити в третьому (а іноді і у другому) випадку, оголошення сигналу із ключовим словом **reg** не приводить до створення регістра. Також комбінаторна логіка синтезується при використанні конструкції **case**, якщо описані всі можливі значення сигналу. При цьому виходить не пріоритетний набір (як у моделюванні), а набір паралельних конструкцій. Крім того **case**, як було розглянуто раніше, використовується для синтезу мультиплексорів. В результаті синтезу будь-який елемент мови може бути синтезований, ігнорований, або може викликати помилку. Конструкції мови можуть підтримуватися повністю, частково або не підтримуватися зовсім. Часовий контроль #nn ігнорується в синтезованій моделі. Контроль подій підтримується в конструкції **always**.

В цифрових пристроях використовується велика кількість різноманітних тригерів, але всі вони мають в своєму складі елемент пам’яті та систему керування. За кількістю інформаційних входів тригери

можуть бути одновходові, двовходові і багатовходові. Найбільше поширення отримали одно- та двовходові тригери. Слід розрізняти кількість інформаційних входів з кількістю фактичних входів, на які надходять інформаційні сигнали, тому що реально діючий інформаційний вхід у структурі тригера може бути кон'юнкцією, диз'юнкцією або будь-якою функцією декількох логічних змінних, діючих на інформаційних входах. В зарубіжній інженерній практиці всі тригерні схеми розділяються на дві групи. Перша з них – flip-flop – характеризується тим, що вибірка вхідних сигналів і відповідна зміна виходів визначається в моменти дії тактових сигналів (синхронні тригери). Особливість другої групи схем – latch – полягає в тому, що вони змінюють свій стан при появі вхідних сигналів незалежно від наявності чи відсутності тактових сигналів.

Для опису регістра-засувки (latch) застосовується наступна конструкція:

```
reg data_out;
always @(data_in or enable)   // при зміні вхідних сигналів
if (enable)                   // при наявності сигналу дозволу
      data_out = data_in;     // передача вхідного сигналу на вихід
```

При зміні будь-якого з вхідних сигналів перевіряється стан входу дозволу enable. Якщо на цьому вході присутній сигнал високого рівня, на виході встановлюється значення сигналу на інформаційному вході data_in.

Для регістрів, що працюють по фронту/зрізу сигналу, застосовується опис із конструкціями **posedge** (фронт) або **negedge** (спад):

```
reg data_out;
always @(posedge clock)       // по фронту тактового сигналу
      data_out = data_in;     // встановлюється значення на виході
```

Щоб додати синхронний сброс, потрібно доповнити модуль сигналом установки/обнуління, але не вносити ці сигнали в список чутливості:

```
reg data_out;
always @(posedge clock)       // по фронту тактового сигналу
if (set_sig)                  // по сигналу установки
      data_out = 1'b1;        // на виході лог.1
else if (reset_sig)           // якщо reset_sig=1
      data_out = 1'b0;        // на виході лог.0
      else                    // інакше
      data_out = data_in;     // на виході значення сигналу на вході
```

Для асинхронної установки/обнуління конструкція повинна бути змінена так, щоб ці сигнали потрапили в список чутливості:

```
reg data_out;
always @(posedge clock or posedge set_sig or posedge reset_sig)
// по фронту будь-якого з сигналів у списку чутливості
if (set_sig)                  // по сигналу установки
        data_out = 1'b1;      // на виході 1
else if (reset_sig)           // якщо reset_sig=1
        data_out = 1'b0;      // на виході 0
else                          // по тактовому сигналу
        data_out = data_in; // на виході встановлюється рівень вхідного
        сигналу
```

Користуючись цими принципами, можна описати тригер будь-якого типу.

Класифікація тригерів може проводитися по різним ознакам. По способу організації логічних зв'язків розрізняють тригери з роздільною установкою станів «0» та «1» (RS-тригери), з лічильним входом (T-тригери), універсальні з роздільною установкою станів «0» і «1» (J-K-тригери), з прийомом інформації по одному входу (D-тригери), універсальні з керованим прийомом інформації по одному входу (DV-тригери), комбіновані (RST-, JKRS-, DRS-тригери), зі складною вхідною логікою.

2.5.1. RS-тригери

Особливість RS-тригера на відміну від комбінаційної схемотехніки полягає в тому, що будь-який із станів є стійким при відсутності вхідних сигналів. Комбінація S="1" та R="1" для тригера, побудованого на вентилях АБО-НІ, і S="0" та R="0" для тригера, побудованого на вентилях І-НІ, на входах вважається забороненою і переводить тригер в невизначений стан. Опис модуля асинхронного RS-тригера приводитися нижче.

```
module RS_trig(q, s, r);      // початок опису модуля тригера
output q;                     // оголошення вихідних портів
input s, r;                   // оголошення вхідних портів
reg q;                        // вихідні порти мають тип регістр
always @ (s or r)             // при зміні сигналів на вході
      if (s && !r)            // якщо на вході s високий рівень
            q=1'b1;           // установлюємо на виході 1
      else if (!s && r)
            q=1'b0;           // інакше встановлюємо на виході 0
      else if (s && r)
```

```
        q=1'b0;
endmodule                    // кінець опису модуля
```

Часова діаграма, яка демонструє роботу тригера, приводиться на рис.2.25.

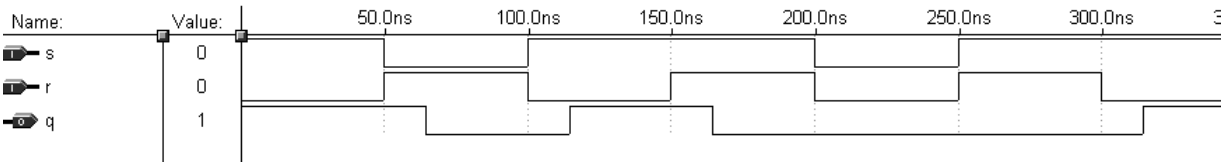

Рис.2.25. Діаграма роботи RS-тригера

Описаний тригер працює в такий спосіб. З появою високого рівня на вході установки s (set) вихід q приймає значення лог.1. Як видно з діаграми, з появою одиниці на вході r, незалежно від стану входу s, на виході буде 0. І тільки за умови відсутності високого рівня на вході r, при низькому рівні сигналу на вході r, вихід прийме значення лог.1. Таким чином, при роботі тригера не виникає невизначених станів виходу. Змінивши потрібним чином умови в описанні модуля, можна змінити логіку роботи тригера (наприклад, зробити значення рівнів сигналів інвертованими).

Розглянемо приклад створення примітиву RS-тригера послідовністного типу. Опис примітиву тригера та модуля з його використанням має вигляд:

```
module rsprim(q,s,r);      // заголовок модуля
output q;                  // вихідний сигнал
input s,r;                 // входи установки та сбросу
RSTRIG(q,s,r);             // підключення примітиву RS-тригера
endmodule                  // кінець опису модуля
primitive RSTRIG(Q, S, R); // заголовок примітиву
output Q;                  // вихідний порт
input S, R;                // вхідні сигнали
reg Q;                     // вихід має регістровий тип
initial Q=1'b1;            // початкове значення стану виходу
table                      // опис таблиці станів
// S R:Q:Q+  порядок опису сигналів в таблиці; „Q+”– новий стан
      1 0 : ? : 1;   // при S=1 на виході встановлюються 1
      f 0 : 1 : -;   // при переході сигналу з 1 в 0 стан не змінюється
      0 r : ? : 0;   // по фронту сигналу сбросу R на виході 0
      0 f : 0 : -;   // при переході сигналу з 1 в 0 стан не змінюється
      1 1 : ? : 0;   // якщо S=R=1, на виході встановлюється 0
endtable
endprimitive       // кінець опису примітиву
```

Використання ключового слова **initial** дозволяє визначити початковий стан виходу тригера. Це значення буде використано при моделюванні роботи. Вихідний порт має регістровий тип, тобто стан виходу тригера зберігається протягом деякого часу між змінами сигналів на вході. Незважаючи на простоту, RS-тригери в чистому вигляді використовуються обмежено. RS-тригер є здебільшого лише елементом пам'яті для різних типів тригерних систем.

2.5.2. D-тригери

Функціональна особливість тригерів цього типу полягає в тому, що сигнал на виході Q в n+1 такті повторює значення сигналу на вході D в n-му такті. Мінімізуючи логічну функцію D-тригера, находимо рівняння:

$$Q_{n+1} = C_n D_n + C_n \overline{Q}_n .$$

Враховуючи той факт, що друга складова рівняння характеризує лише режим зберігання інформації, закон функціонування тригера може бути описаний формулою:

$$Q_{n+1} = C_n D_n.$$

З точки зору логіки роботи тригера, він затримує проходження сигналу, який поступає на вхід D, на один такт періоду синхросигналу (delay-затримка).

Опис модуля D-тригера з інформаційним входом data і тактовим сигналом clk приводиться нижче.

```
module D_trig (q, clock, data);        // початок опису модуля тригера
output q;                              // оголошення вихідного порту
input clock, data;                     // вхідні порти
reg q;                                 // вихідний порт має тип регістр
      always @ (negedge clock)         // по спаду тактового сигналу
      q = data;            // установлюємо на виході значення  входу
endmodule                              // кінець модуля
```

В описі модуля використовується конструкція контролю подій. Щоразу при зміні синхронізуючого сигналу clock з високого рівня на низький відбувається передача вхідного сигналу data на вихід q.

Часова діаграма, що демонструє роботу приведеного опису D-тригера, приводиться на рис.2.26.

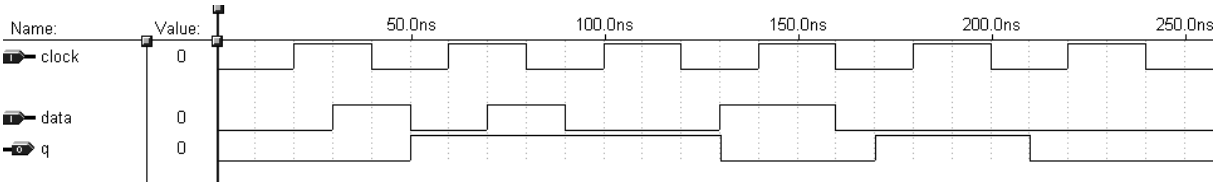

Рис.2.26. Діаграма роботи D-тригера

В реальності, всі тригери мають часові затримки встановлення вихідних сигналів, а також мають часові вимоги до вхідних сигналів, при порушенні яких будь-який тригер буде працювати нестійко, або не буде працювати взагалі. При описі на мові Verilog розробник має можливість запобігти появи таких ситуацій шляхом опису реакції модуля на всі можливі зміни вхідних сигналів. В першому наближенні можна вважати, що мінімально припустимі часові інтервали між вхідними сигналами повинні дорівнювати 1-2 затримкам відповідної мікросхеми. Також не повинна бути занадто малою тривалість фронту тактового сигналу, інакше тригер може перемикатися нестійко. Якщо сигнал має розмірність більшу одиниці, то пристрої синтезуються для кожного біта, а існуюча логіка в подібних конструкціях синтезується в комбінаторну схему. Розглянемо приклад опису тригера, який тактується фронтом, з асинхронним інверсним обнуліннням і установкою.

```
module dffe_async(reset, preset, data, clk, q); // заголовок модуля
input reset, preset, data, clk;                 // перелік вхідних сигналів
output q;                                       // вихідний сигнал
reg q;
always @(posedge clk or negedge reset or posedge preset)
// процедурний блок
        if (~reset)             // по спаду сигналу сбросу
                q=1'b0;         // на виході встановлюється 0
        else if (preset)        // по сигналу установки
                q=1'b1;         // на виході встановлюється 1
        else q=data; // на виході встановлюється рівень вхідного сигналу
endmodule
```

На рис.2.27 представлена часова діаграма, отримана в результаті моделювання. В розглянутому прикладі вхід reset інвертований, тобто обнуління тригера відбувається при низькому рівні сигналу на цьому вході.

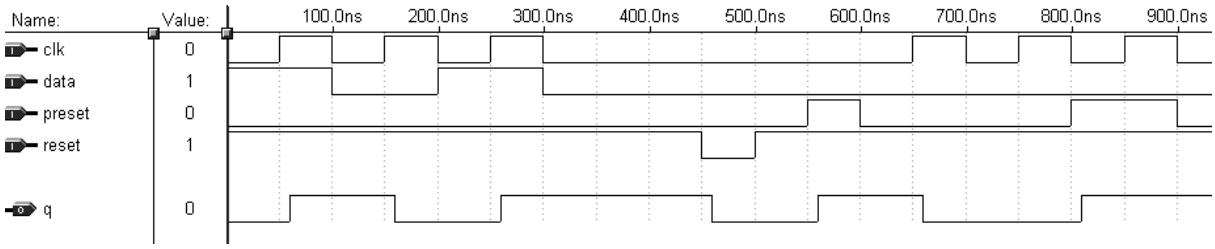

Рис.2.27. Діаграма роботи тригера

На основі наведеного вище опису можна створити тригер, який тактується фронтом, із синхронними обнулінням і установкою. Для цього досить у конструкції **always**@ залишити тільки **posedge** clk. По фронту тактового сигналу будуть перевірятися стани входів установки (preset) і обнуління (reset). Безпосередньо в момент зміни вказаних сигналів тригер не змінить свого стану. Шляхом зміни ключового слова **posedge** на **negedge** можна створити тригер, що буде спрацьовувати по спаду тактового сигналу. Розглянемо опис модуля тригера, який тактується переднім фронтом, із синхронним сбросом і дозволом тактового сигналу.

```
module dffe_sync(q, reset, en, data, clk);// початок опису модуля
input reset, en, data, clk;        // перелік вхідних сигналів
output q;                          // вихідний сигнал
reg q;                             // вихід має регістровий тип
always @(posedge clk)              // по фронту тактового сигналу
      if (~reset)                  // якщо reset=0
            q=1'b0;                // на виході 0
      else if(en)                  // якщо встановлена 1 на вході дозволу
            q=data;                // на виході значення вхідного сигналу
endmodule                          // кінець опису модуля
```

Після моделювання роботи описаного тригера була отримана діаграма, що представлена на рис.2.28.

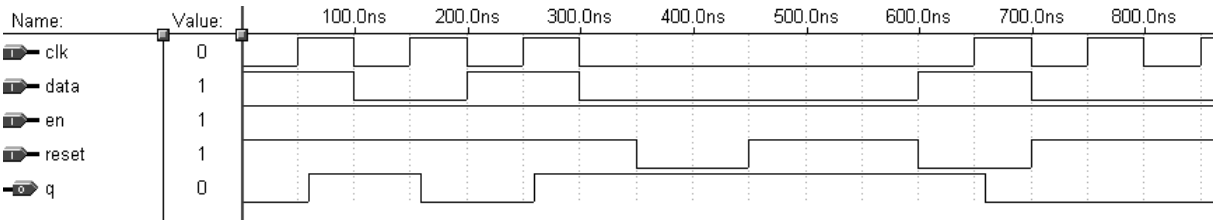

Рис.2.28. Діаграма роботи тригера

Як видно з діаграми, з появою низького рівня на вході reset стан тригера не змінився, оскільки на цьому проміжку часу був відсутній фронт тактового сигналу.

Розглянемо приклад засувки (latch) на основі D-тригера. Нижче приводиться опис засувки з дозволом виходу.

```
module latch(enable, data, q);  // початок опису модуля засувки
input enable, data;             // вхідні сигнали
output q;                       // вихідний сигнал
reg q;                          // вихід має регістровий тип
always @(enable or data)        // при зміні будь-якого з вхідних сигналів
        if (enable)             // якщо сигнал дозволу має рівень 1
                q=data;         // на виході встановлюється значення
data
endmodule
```

Часова діаграма роботи модуля представлена на рис.2.29.

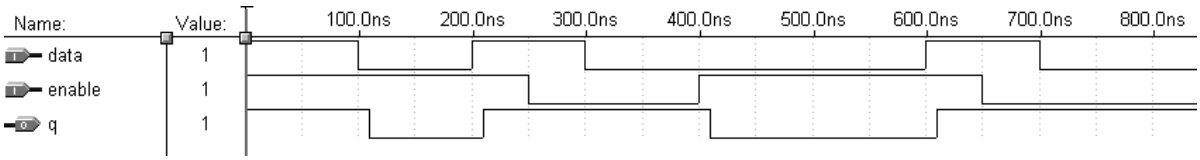

Рис.2.29. Діаграма роботи «засувки»

З діаграми видно, що зміна сигналу на виході q відбувається разом зі зміною сигналу на вході data при високому рівні сигналу на вході дозволу enable. Якщо ж на вході дозволу присутній логічний 0, стан виходу тригера не змінюється.

Розглянемо приклад опису D-тригера як примітиву користувача в САПР Quartus.

```
primitive UDFF(q,d,clk,nrst);  // опис примітиву D-тригера
output q;                      // вихідний порт
input d,clk,nrst;              // вхідні сигнали
reg q;                         // вихід має регістровий тип
table                          // опис таблиці станів тригера
//d  clk  nrst : q : next_q  порядок опису сигналів в таблиці
0 r 1 : ? : 0;                 // прийом входу d по фронту clk
1 r 1 : ? : 1;                 // прийом входу d по фронту clk
? ? 0 : ? : 0;                 // асинхронне обнуління
? f ? : ? : -;
// по спаду тактового сигналу стан тригера  не змінюється
endtable
endprimitive                   // кінець опису примітиву
```

Тригер має інформаційний вхід, асинхронне обнуління та вхід тактового сигналу. За допомогою таблиці описані стани тригера. В описанні примітиву символ „r” означає фронт сигналу, „f” – спад сигналу, „?” – будь-яке значення, „-” – попередній стан. В наступному описі модуля демонструється приклад використання створеного примітиву.

```
module dtrigpr(q,data,clock,nres);     // модуль D-тригера
output q;                              // вихідний сигнал
input data,clock,nres;                 // входи
UDFF(q,data,clock,nres);               // підключення примітиву користувача
endmodule                              // кінець опису модуля
```

Після моделювання була отримана діаграма, що приводиться на рис.2.30.

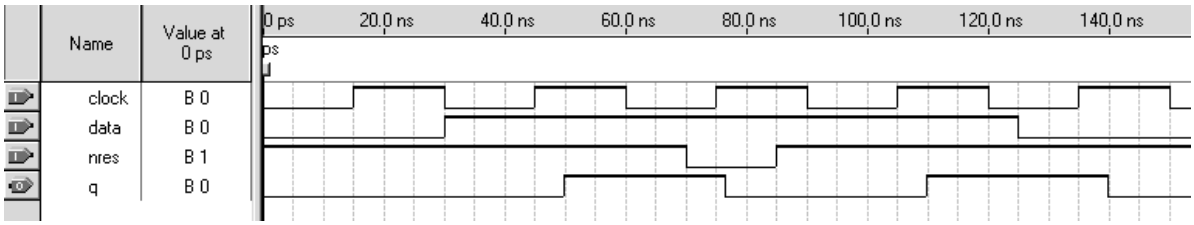

Рис.2.30. Діаграма роботи примітиву D-тригера

По фронту тактового сигналу clock на виході q встановлюється рівень сигналу, який присутній на вході data. При переході в низький стан сигналу на вході сбросу nres, на виході модуля встановлюється рівень лог.0.

2.5.3. J-K тригери

Цей тип тригерів за логікою роботи подібний до R-S тригерів, але на відміну від них не має невизначених переходів. Однотактні тригери відрізняються наявністю зворотних зв’язків з виходів на вхід, а також елементами часової затримки. Стан виходів J-K тригера залежить не тільки від сигналів на входах J та K, але і від логічно пов'язаних з ними сигналів з виходів Q та $\overline{Q}$. Поява комбінації J=K=1 у кожному такті призводить до зміни стану тригера на протилежний. При J=K=0 на виходах будуть логічні одиниці, які не впливають на стан тригера. При одиниці на вході J і нулі на вході K по фронту тактового сигналу clock прямий вхід встановлюється в одиницю (інверсний – в нуль). При нулі на вході J та одиниці на вході K, по фронту сигналу clock прямий вхід встановлюється в нуль (інверсний – в одиницю). Опис алгоритму реалізований в наступному модулі.

```
module JK_trig (q, nq, clock, j, k); // початок модуля
output q;                            // оголошення вихідних портів
reg    q;                            // вихід має тип регістр
output nq;                           // інверсний вихід
reg nq;
input  clock;                        // оголошення вхідних портів
input j, k;
   always @(posedge clock)           // по фронту тактового сигналу
    begin                        // перевіряємо умови
          if (j && !k)           // при наявності 1 на вході j
    begin
          q<=1;                  // на виході встановлюється 1
          nq<=0;                 // на інверсному виході 0
    end
          else if (!j && k)      // при наявності 1 на вході k
    begin
          q<=0;                  // на виході встановлюється 0
          nq<=1;                 // на інверсному виході 1
    end
          else if (j && k)       // при високому рівні на обох входах
    begin
          q<=~q;          // інвертуються значення сигналів на виходах
          nq<=~nq;
    end
          else                   // при низькому рівні на обох входах
          begin
                q<=1;            // на обох виходах сигнал лог.1
                nq<=1;
          end
          end
endmodule                        // кінець модуля
```

На рис.2.31 приводяться результати моделювання роботи модуля JK_trig.

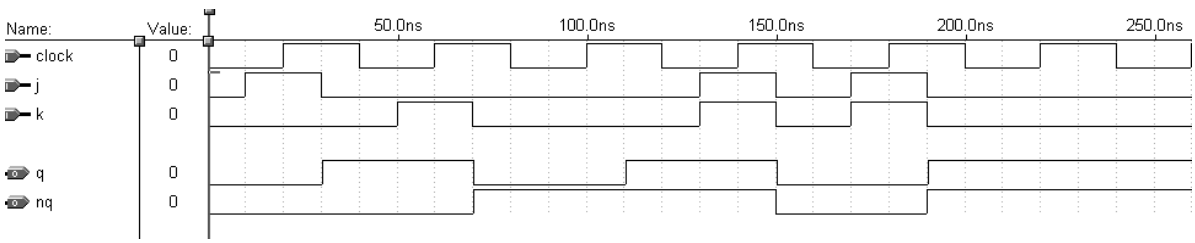

Рис.2.31. Робота J-K-тригера

Розглянемо ще один варіант опису J-K-тригера. Цей приклад представляє з себе поведінковий опис, демонструється визначення режиму перемикання. Використані конструкція вибору та оператор об'єднання. Опис модуля представлений нижче.

```
module JKFF(q, clock, reset, j, k);      // початок модуля
output q;                                // вихідний сигнал
input clock, reset, j, k; // вхідні сигнали: тактовий, обнуління, керуючі
reg q;
always @ (posedge clock or posedge reset) // процедурний блок
begin
      if (reset) q<=0;      // по сигналу reset встановлюємо на виході 0
      else
       case ({j,k})                 // в залежності від стану входів
            2'b00: q<=1'b0;    // встановлюємо на виході
            2'b01: q<=1'b0;    // потрібне значення
            2'b10: q<=1'b1;
            2'b11: q<=~q;
      endcase                       // кінець конструкції вибору
end
endmodule                           // кінець модуля
```

В цьому модулі описаний J-K-тригер з динамічним керуванням і асинхронним сбросом. В конструкції вибору **case** відбувається порівняння значень сигналів на входах з визначеним набором констант і при їх рівності відбувається установка на виході q потрібного рівня сигналу. Проаналізувавши часову діаграму (рис.2.32), отриману при моделюванні роботи тригера, можемо переконатися в правильності його роботи.

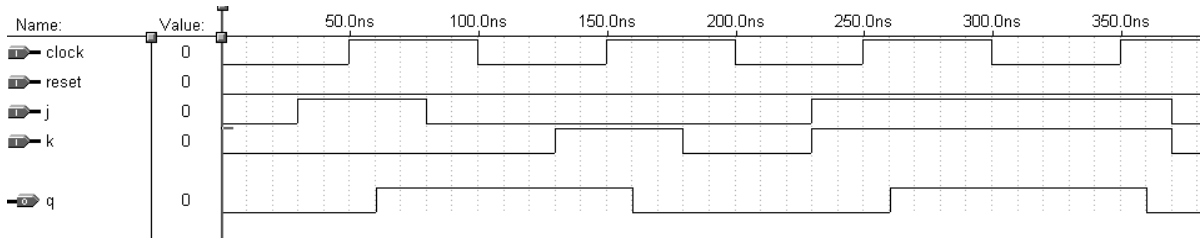

Рис.2.32. Моделювання роботи тригера.

2.5.4. T- та TV- тригери

До тригерів Т-типу відносяться такі тригери, які по сигналу на Т-вході перемикаються на протилежний стан. Це тригери з динамічним Т-входом, або з динамічним С-входом і статичним Т-входом. У зв'язку з їх широким використанням у лічильниках імпульсів, Т-тригери з динамічним Т-входом часто називаються тригерами з лічильним входом, або лічильними тригерами. В залежності від характеру дії Т-входу, вони часто

діляться на Т-тригери, які спрацьовують по фронту Т-імпульсу, та Т-тригери, що спрацьовують по зрізу Т-імпульсу. Т-тригер виконує операцію додавання по модулю 2 вхідної змінної. Одиничний Т-тригер розділяє на 2 частоту вхідної послідовності імпульсів. Послідовне включення m таких тригерів дає можливість ділити вхідну послідовність імпульсів у 2^m разів, або утворює лічильник. Найпростіший опис модуля Т-тригера на мові Verilog має вигляд:

```
module T_trig (q, nq, t); // початок опису модуля
output q;                 // оголошення вихідного порту
reg q;                    // вихід має регістровий тип
output nq;                // інверсний вихід
reg nq;
input t;                  // вхідний порт
always @(posedge t)       // по фронту сигналу на вході t
      begin
            q=~q;         // інвертуємо значення на виходах тригера
            nq=~q;
      end
endmodule                 // кінець опису Т-тригера
```

Тригер спрацьовує по фронту сигнала t. Виходи визначені як ті, що мають регістровий тип. Це дозволило використати оператор присвоєння, а також зберігати стан тригера. Часова діаграма, яка демонструє роботу Т-тригера, представлена на рис.2.33.

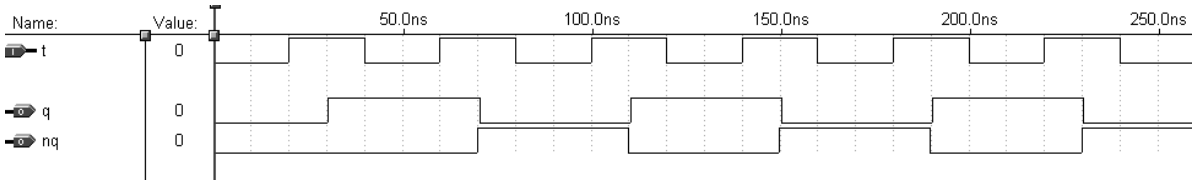

Рис.2.33. Часова діаграма роботи Т-тригера

Приклад опису примітиву Т-тригера був розглянутий в першому розділі. Для реалізації лічильного режиму найчастіше використовуються не тригери, а лічильники, які будуть розглянуті далі.

За допомогою RS-тригера дуже просто вирішується проблема тремтіння контактів механічних перемикачів. Правда в такому випадку необхідний перемикач з трьома виводами. Основне призначення тригери знаходять у тих випадках, коли потрібно сформувати сигнал, тривалість якого відповідає дії операції, що виконується. Вихідний сигнал тригера при цьому може дозволяти цей процес, або інформувати інші вузли пристрою про те, що процес має місце. Пізніше буде наведено приклад використання тригера саме в цій ролі. Друга важлива область застосування

тригерів – це синхронізація сигналів. Наприклад, тригер дозволяє найбільш просто позбавитися від паразитних коротких імпульсів на виходах комбінаційних схем, які виникають при майже одночасній зміні декількох вхідних сигналів. Для синхронізації в даному випадку необхідно мати синхросигнал, який супроводжує вхідні інформаційні сигнали і затриманий відносно зміни цих сигналів на деякий час, більший за час затримки комбінаційної схеми.

2.6. Розробка скінченних автоматів

Скінченними автоматами або послідовністними машинами називаються цифрові схеми, стан виходів яких залежить не тільки від значень вхідних сигналів у даний момент часу, а і від внутрішнього стану схеми в попередній момент часу. З цього визначення випливає, що автомат має декілька елементів пам'яті, інформація в яких залежить як від комбінації вхідних сигналів, так і від значення цієї інформації в попередні моменти часу. Аналогічний взаємозв'язок має місце і для вихідних сигналів. Таким чином, однією з особливостей цієї схеми є те, що вона має свій внутрішній стан, від якого залежить реакція на вхідні сигнали. Як відомо, найчастіше розглядають два типи автоматів – автомати Мілі та Мура. Вихід автомату Мура є функцією тільки потокового стану, у той час як вихід автомату Мілі – функція як потокового стану, так і зовнішньої події.

Загалом скінченний автомат складається з декількох основних частин: регістр стану, логіка переходів, логіка формування виходу. Наявність елементів пам'яті, які можуть бути синхронними або асинхронними, визначає і тип скінченного автомату, який може бути синхронним і працювати строго у відповідності з тактовими сигналами зовнішнього генератора, або асинхронним, і працювати у відповідності до зовнішніх сигналів. В якості елементу пам'яті можуть бути використані D-тригери. Загалом, скінченні автомати дещо складніші від розглянутих вище тригерів. Тому дуже зручно реалізовувати автомати на програмному рівні. Скінченний автомат може знаходитися в кожний конкретний момент часу лише в одному стані. Кожний тактовий імпульс, або зміна вхідних сигналів викликає перехід автомату з одного стану в інший. Правила переходу визначаються комбінаційною схемою, яка називається логікою переходів. Формування вихідного сигналу автомата відбувається за допомогою логіки формування виходу.

Як приклад, розглянемо опис скінченного автомату синхронного типу. В контексті мови Verilog елементом пам'яті, в якому буде зберігатися внутрішній стан автомату, є змінна state регістрового типу.

```
module mach(clk, in, reset, out);      // модуль скінченого автомату
input   clk, in, reset;                // вхідні сигнали
```

```
output  out;                    // вихід
reg     out;                    // вихід має регістровий тип
reg     state;                  // змінна для збереження стану автомату
parameter S0 = 0, S1 = 1;       // параметри
always @ (state)                //при зміні внутрішнього стану автомату
begin
        case (state)            // визначаємо новий стан
                S0: out = 0;    // на виході 0
                S1: out = 1;    // на виході 1
        default: out = 0;
        endcase
end
                                // по фронту тактового сигналу
always @ (posedge clk or posedge reset)
                                // або сигналу сбросу
begin
        if (reset) state = S0;  // при reset=1 автомат переходить в S0
        else
        case (state)            // визначаємо поточний стан автомату
                S0: state = S1; // виконується перехід в інший стан
                S1: if (in) state = S0; // в залежності від сигналу на вході
                else state = S1;        // автомат переходить в S0 або S1
        endcase
end
endmodule                       // кінець опису модуля автомату
```

По фронту сигналів clk або reset відбувається зміна значення state (власне, внутрішнього стану автомата). Якщо на вході reset присутній 0, значення змінної state приймає таке ж значення (тобто 0). Якщо ж на вході reset сигнал високого рівня, відбувається аналіз стану входу in. В цьому випадку змінній state присвоюється інверсне значення сигналу на вході in. Це демонструє діаграма, яка представлена на рис.2.34.

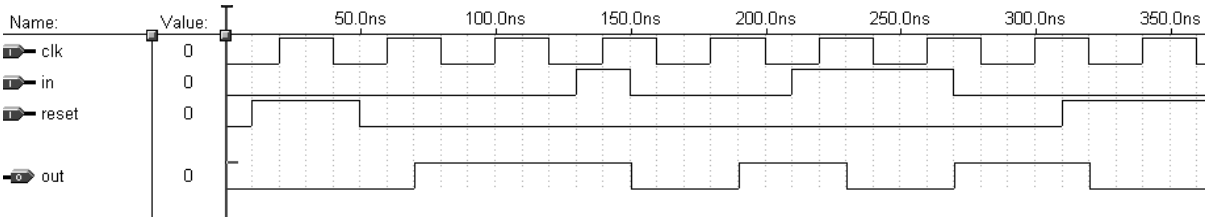

Рис.2.34. Часова діаграма роботи автомата

Розглянутий приклад цікавий також тим, що в ньому показаний спосіб обробки подій всередині модуля. Змінна state оголошена в самому модулі. За допомогою конструкції очікування події **always**@ відбувається

контроль зміни внутрішнього стану автомата. При зміні значення state виконується відповідний блок і на виході встановлюються відповідні сигнали.

2.7. Лічильники імпульсів

Лічильником називається функціональний вузол, призначений для підрахунку кількості імпульсів вхідної імпульсної послідовності. Основним елементом будь-якого лічильника є Т-тригер, який використовується для роботи в лічильному режимі. Т-тригери можуть зберігати свій стан та додавати до нього по модулю 2 вхідний сигнал. Одиночний Т-тригер розділяє на 2 частоту вхідної послідовності імпульсів. При структурному описі каскадне включення n таких тригерів утворює лічильник з коефіцієнтом перерахунку (модуль рахунку, ємність лічильника) 2^n. Тригери лічильника послідовно перебирають всі свої стани та повертаються в початковий стан, видаючи на вихід лічильника один імпульс. Якщо стан виходів тригерів лічильника змінюється у відповідності з послідовністю двійкових чисел, лічильник називається двійковим. Лічильник може працювати на збільшення вихідного коду по кожному вхідному імпульсу. Це основний режим, що є у всіх лічильників. Він називається режимом прямого рахунку. Код, що визначається вихідними станами тригерів, змінюється в зростаючому напрямку (додаючі лічильники). Лічильники також можуть працювати на зменшення вихідного коду по кожному вхідному імпульсу. Це режим зворотнього або інверсного рахунку. При зворотній зміні станів лічильники називаються віднімаючими, а якщо напрямок перебору може змінюватися, то лічильник називається реверсивним. Напрямок рахунку визначається як динамічними властивостями тригерів (перемикання по фронту або спаду), так і способом з’єднання виходів попереднього тригера з входом наступного. Якщо, наприклад, у лічильнику використовуються тригери, що змінюють свій стан по фронту синхроімпульсу, то для організації додаючого лічильника необхідно з’єднати вхід наступного тригеру з інверсним виходом попереднього, а для організації віднімаючого лічильника для зв'язку між тригерами використовується прямий вихід тригерів. Для використання тригерів, що спрацьовують по спаду синхроімпульсу, вказані зв'язки повинні бути протилежними. Виходи лічильника представляють собою виходи цих тригерів. Для чотирьохрозрядного додаючого лічильника з коефіцієнтом перерахунку $К=2^4=16$ початковим кодом, записаним на прямих виходах тригерів, є 0000, а кінцевим, після якого наступає переповнення, код 1111. Цей же код є початковим для віднімаючого лічильника. У випадку, коли в якості початкового коду віднімаючого лічильника прийнятий стан 0000, поточний спад тригерів лічильника відображає від'ємне число зчитаних імпульсів, яке представлене в доповнюючому коді. Що стосується поведінкового опису лічильника, то

для реалізації реверсивного режиму використовуються оператори прийняття рішень.

Нижче приводитися структурний опис двійкового чотирьохрозрядного лічильника.

```
module counter_4(count, reset, clock,en); // опис лічильника
output [3:0] count;                       // вихідний порт
input reset;                              // вхід обнуління
input clock;                              // тактовий сигнал
input en;                                 // вхід дозволу
    t_ff t1(count[0],~clock,en,reset);    // послідовне з'єднання
    t_ff t2(count[1],~count[0],en,reset); // Т-тригерів
    t_ff t3(count[2],~count[1],en,reset);
    t_ff t4(count[3],~count[2],en,reset);
endmodule                                 // кінець опису лічильника
```

Модуль Т-тригера, який використовується при структурному описі лічильника, має вигляд:

```
module t_ff(q, clk, enable, res);   // заголовок модуля Т-тригера
output q;                           // вихідний порт
input clk;                          // тактовий сигнал
input enable;                       // вхід дозволу
input res;                          // вхід обнуління
reg q;
always @ (posedge clk or posedge res)
                                    // по фронту тактового сигналу
                                    // або сигналу res
    begin
        if (res) q=0;               // якщо res=1, на виході 0
    else
        if (enable) q=~q;   // інакше інвертуємо значення виходу
         end
endmodule                           // кінець опису модуля
```

У програмі використовуються чотири Т-тригери. Інверсний вихід попереднього тригеру з'єднаний з входом наступного тригера. Лічильник має вхід сбросу та вхід дозволу рахунку. Кожного разу по фронту імпульсу clock відбувається збільшення на 1 вихідного значення лічильника. Демонструє роботу описаного модуля часова діаграма, яка приводитися на рис.2.35.

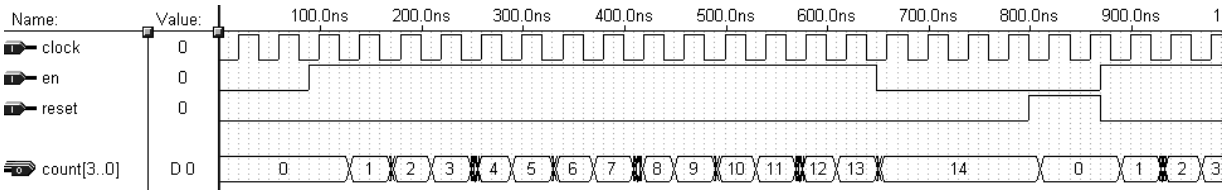

Рис.2.35.Часова діаграма роботи лічильника

При описі цього лічильника ми використали тригери, тобто проводили опис модуля на вентильному рівні. При описі наступного лічильника ми також використовуємо тригер, який тактується спадом сигналу clock.

```
module counter16bit (value, clock, fifteen, altFifteen);
output [3:0] value;                    // вихідний 4-розрядний порт
output fifteen,  altFifteen;
input clock;                           // тактуючий сигнал
        dEdgeFF a (value[0], clock, ~value[0]),
        // з'єднання 4 D-тригерів
        b (value[1], clock, value[1] ^  value[0]),
        c (value[2], clock, value[2] ^ &value[1:0]),
        d (value[3], clock, value[3] ^ &value[2:0]);
        assign fifteen = value[0] & value[1] & value[2] & value[3];
        assign altFifteen = &value;
endmodule                              // кінець модуля
```

Описання модуля тригера приводиться нижче.

```
module dEdgeFF (q, clock, data);      // опис тригера
output q;                             // вихідний порт
input  clock, data;                   // вхідні сигнали
reg    q;
always @ (negedge clock)              // по спаду тактового сигналу
        q = data;                     // передаємо вхідне значення на вихід
endmodule                             // кінець опису тригера
```

Модуль лічильника має виходи fifteen та altFifteen. Високий рівень сигналу на цих виходах з'являється при досягненні лічильником значення 15. В першому випадку для визначення сигналу використовується логічний оператор I, а в другому – оператор згортки по I. Результати виконання цих операторів абсолютно однакові. Часова діаграма, яка представлена на рис.2.36, демонструє роботу описаного модуля.

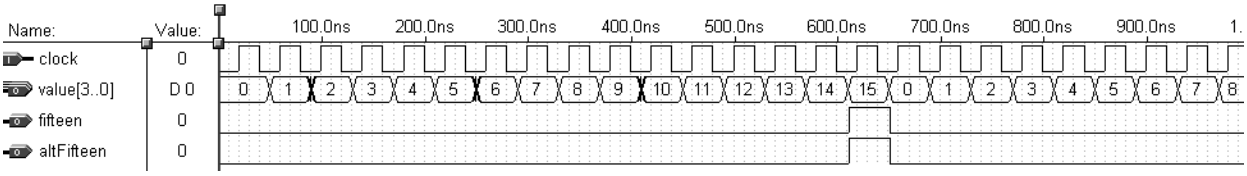

Рис.2.36. Часова діаграма роботи лічильника

Опис на поведінковому рівні – це найпростіший і найбільш загальний спосіб задання алгоритму роботи лічильника. Проілюструємо це на наступному прикладі. Програма реверсивного лічильника, описаного на Verilog з використанням оператора умови **if** має вигляд:

```
module counters(count, dir, load, data, en, clk, rst);// початок модуля
parameter WIDTH=4;                  // розрядність лічильника
output  [WIDTH-1:0] count;          // вихід
input dir, load, en, clk, rst, data;    // вхідні сигнали
reg [WIDTH-1:0] count;
wire [WIDTH-1:0] data;
always @ (posedge clk)              // по фронту тактового сигналу
begin                               // виконуємо цей блок
    if (load & en & !rst) count=data; // перевіряємо виконання умов
    else if (en & dir & !rst) count=count+1; // інкремент значення
    else if (en & !dir & !rst) count=count-1;// декремент
    else if (rst) count=0;  // по сигналу сбросу обнуління лічильника
end
endmodule                           // кінець опису модуля
```

Зверніть увагу на те, що в даному прикладі при описі алгоритму роботи не використовувались, як раніше, тригери. Лічильник описаний за допомогою арифметичних операторів додавання та віднімання. Крім того, можна встановлювати попереднє значення виходу лічильника. Це відбувається по фронту тактового сигналу clk при наявності високого рівня сигналу на вході load. Рівень сигналу на вході dir визначає напрямок роботи лічильника (збільшення чи зменшення значення). Робота лічильника можлива при високому рівні сигналу на вході en. Можна змінити величину, на яку буде збільшуватися (або зменшуватися) значення лічильника. Сброс відбувається при низькому рівні сигналу на вході rst, оскільки при описі модуля від вказаний як інверсний (оператор „!”). Модуль лічильника містить параметр WIDTH, який визначає розрядність даних. Змінивши значення цього параметру можна змінити коефіцієнт перерахунку.

Розглянемо приклад побудови параметризованого лічильника з попереднім завантаженням даних, входами дозволу рахунку і асинхронним сбросом. В якості параметру виступає розрядність шини даних.

```
module counter8(out, cout, data, load, clk, en, reset);
parameter WIDTH=8;          // розрядність даних
output [WIDTH-1:0] out;     // вихідний порт
output cout;                // вихід переповнення
input [WIDTH-1:0] data;     // вхід даних для завантаження
input load, clk, en, reset; // входи керування
reg [WIDTH-1:0] out;        // вихід має регістровий тип
always @(posedge clk or negedge reset)  // процедурний блок
    if(!reset)              // якщо reset=1
        out=0;              // на виході 0
    else if(load)           // якщо завантаження
        out=data;           // установка значення лічильника
    else if(en)             // якщо встановлений сигнал дозволу
        out=out+1;          // збільшуємо значення лічильника
    assign cout=&out;
    // встановлюємо значення на виході переповнення
endmodule                   // кінець опису модуля
```

Сигнал переповнення лічильника cout визначається за допомогою оператора згортки по I над вихідними даними. Сигнал високого рівня установиться на цьому виході тільки тоді, коли всі біти на виході out лічильника будуть встановлені в 1. Звертаємо увагу на те, що вихід переповнення cout в розділі декларації портів не визначений як регістр. Це дозволило застосувати оператор безперервного присвоєння **assign** за межами процедурного блоку **always**. Вхід reset інвертований і обнуління виконується по спаду сигналу. Модуль має синхронний вхід завантаження даних load. По фронту тактового сигналу і високому рівні на вході load дані, які присутні на вхідному порту data будуть завантажені у лічильник. Збільшення значення лічильника відбувається якщо на вході дозволу en присутній сигнал високого рівня. На рис.2.37 приводиться часова діаграма роботи описаного модуля.

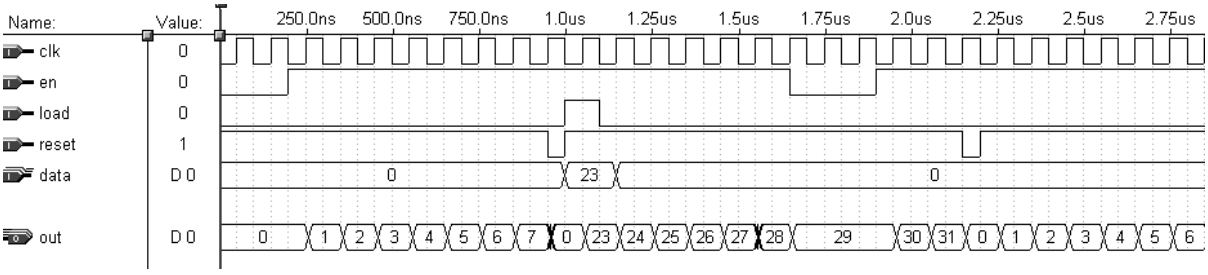

Рис.2.37. Діаграма роботи лічильника

Опис пристрою, який може бути використаний в якості формувача імпульсів с заданою тривалістю, приводитися нижче.

```
module pwm(out, count, clk, run, compare); // початок модуля
output out;                    // оголошення вихідних портів
output [7:0] count;
input clk, run;                // оголошення вхідних портів
input [7:0] compare;           // вхід визначення тривалості імпульсу
reg flag, temp;
reg [7:0] count;               // розрядність вихідної шини дорівнює 8
reg [7:0] compare1;
reg out;
reg run1;
      always @ (posedge clk) // по фронту тактового сигналу
      begin
      run1=run;
      if (run1==1)             // якщо умова виконується, змінюємо
                               // скважність імпульсів
      begin
            flag=0;            // обнуління внутрішніх змінних
            count=0;           // значення лічильника дорівнює 0
            temp=1;
            out=0;             // на виході низький рівень сигналу
            compare1=compare;  // установлюємо нове значення
                               // регістра порівняння
      end
      if (!flag && temp==1)
            begin
            if (count>=compare1) out=1;
// порівнюємо поточне значення лічильника з регістром порівняння і якщо
//  рівні, установлюємо out  в 1
            count=count+1;
            if (count==20) flag=1;    // зміна напрямку рахунку
            end
      else if (flag && temp==1)
            begin
            if (count<compare1) out=0; // установлюємо  вихід out в 0
            count=count-1;             // зменшуємо значення лічильника
            if (count==0) flag=0;      // змінюємо напрямок рахунку при
                                       // досягненні 0
            end
      end
endmodule                              // кінець опису модуля
```

Виконавши моделювання розробленого модулю, можемо переконатися в правильності його роботи. Часова діаграма приводиться на рис.2.38.

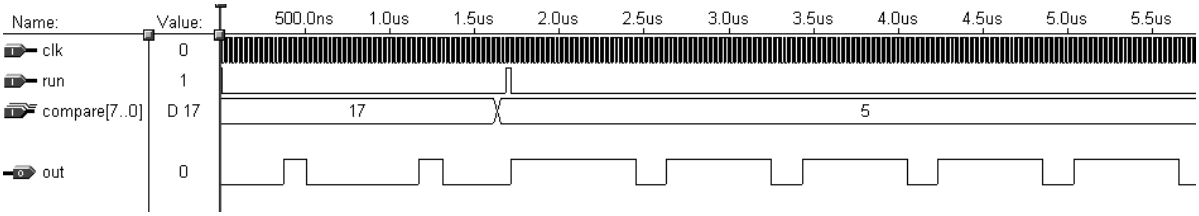

Рис.2.38. Діаграма роботи формувача імпульсів

З появою на вході clk фронту тактового сигналу до поточного значення накопичуючого суматора count додається 1. Додавання одиниці до значення в накопичуючому суматорі після певної кількості тактів призведе до переповнення суматора, і у всіх розрядах лічильника будуть нулі. В даному прикладі відбувається порівняння значення накопичуючого суматора із значенням змінної compare1. Розробник має можливість установити це значення, подавши на вхід compare двійковий код і встановивши на вході run логічну 1. При досягненні значення 20 у накопичуючому суматорі встановлюється прапор flag і з суматора по кожному фронту синхросигналу віднімається 1. При досягненні значення 0 у лічильнику змінна flag обнуляється і напрямок рахунку змінюється. По такому алгоритму також працює широтно-імпульсний модулятор. Період імпульсів постійний: змінюється їх скважність. Розглянемо приклад параметризованого модуля керованого дільника частоти, у якого буде змінюватися період імпульсів в залежності від вхідного коду, а скважність буде залишатися постійною. Опис модуля приводиться нижче.

```
module dcnt(outclk, clk, en, din);      // заголовок модуля
parameter WIDTH_DIN=4;                  // розрядність вхідної шини
output outclk;                          // вихід імпульсів
input clk, en;                          // тактовий вхід та вхід дозволу
input [WIDTH_DIN-1:0] din;              // вхідна шина даних
reg outclk;                             // вихід має регістровий тип
reg [WIDTH_DIN-1:0] temp;               // внутрішній лічильник імпульсів
reg [WIDTH_DIN-1:0] compare;// змінна для порівняння з лічильником
always @(posedge clk)                   // по фронту тактового сигналу
      if(en==0)                         // якщо на вході дозволу 0
      begin
            outclk<=0;                  // на виході низький рівень сигналу
            compare<=0;                 // обнуління внутрішньої змінної
      end
      else                              // якщо на вході дозволу 1
```

```
        begin
            temp<=temp+1;    // збільшення внутрішнього лічильника
            if(temp>=compare)        // якщо виконується умова
            begin
                outclk<=~outclk;  // інверсія значення на виході
                temp<=0;    // обнуління внутрішнього лічильника
                compare<=din;      // запис даних з вхідної шини
            end
        end
    endmodule                    // кінець опису модуля
```

В наведеному прикладі для керування частотою імпульсів на виході модуля використовується внутрішній лічильник temp, значення якого порівнюється зі значенням змінної регістрового типу compare. Запис даних з вхідної шини в цю змінну відбувається лише при нульовому значенні внутрішнього лічильника. Це дозволило запобігти появі „хибного" імпульсу в момент зміни вхідного коду. Інакше тривалість такого імпульсу була б випадковою величиною і залежала від поточного значення внутрішнього лічильника. Період імпульсів $T_{вих}$ на виході модуля визначається виразом:

$$T_{вих}=T_{вх}\cdot(din+1),$$

де $T_{вх}$ – період вхідних імпульсів, din – значення коду на вході din. Робота модуля можлива при високому рівні сигналу на вході дозволу en. Параметр WIDTH_DIN визначає розрядність вхідної шини та внутрішніх змінних. Параметризований опис лічильників дозволить побудувати модулі під конкретне завдання, що додає їм універсальності. На рис.2.39 приводиться діаграма роботи керованого дільника частоти.

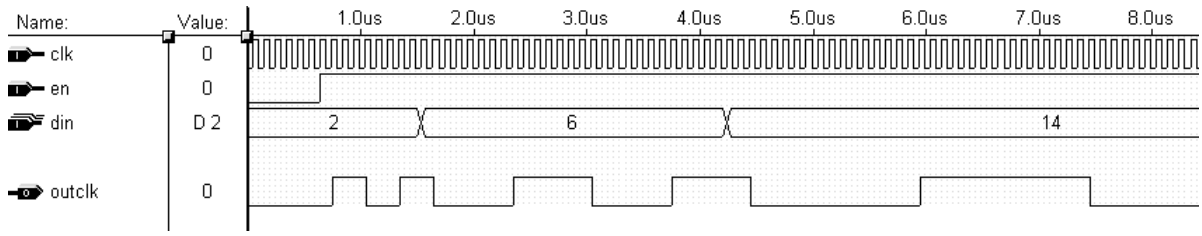

Рис.2.39. Діаграма роботи керованого дільника частоти

Для роботи багатьох пристроїв потрібний тактовий сигнал визначеної частоти. Наприклад, для роботи годинника потрібен генератор з періодом імпульсів 1с (частота 1Гц). Для одержання такої частоти можна використати кварцовий генератор і дільник частоти. Для ділення частоти на 2 може бути використаний Т-тригер. Послідовне з'єднання декількох тригерів дозволить отримати бажаний коефіцієнт ділення. Для вирішення цієї задачі також може бути використаний лічильник імпульсів з

додатковою схемою обнуління по досягненню визначеного значення. В бібліотеках САПР MAX+plus II і Quartus є модуль lpm_constant, який представляє з себе генератор константи. В наступному прикладі приводиться опис модуля дільника частоти, в якому використовується вказаний генератор константи.

```
module oscil(clk, en, oclk);      // початок опису модуля
parameter DIV=12;                 // параметр значення дільника частоти
parameter WIDTHD=8;               // розрядність дільника
input clk, en;                    // входи тактування та дозволу
output oclk;                      // вихідний порт
reg oclk;
reg [WIDTHD-1:0] cnt;             // змінна для підрахунку імпульсів
wire [WIDTHD-1:0] tmp;            // додатковий ланцюг
lpm_constant const(.result(tmp)); // модуль генератора константи
      defparam                    // перевизначення параметрів
            const.LPM_WIDTH=WIDTHD,
            const.LPM_CVALUE=DIV;
always@(posedge clk)              // по фронту тактового сигналу
      begin
      if (en==0)                  // якщо відсутній сигнал дозволу
            oclk=0;               // на виході 0
      else if (tmp==cnt) // якщо кількість імпульсів дорівнює дільнику
      begin
            cnt=0;                // обнуління значення лічильника
            oclk=~oclk;           // інвертуємо сигнал на виході
      end
      else
            cnt=cnt+1;            // збільшення на 1 значення cnt
      end
endmodule                         // кінець опису модуля
```

На рис.2.40 представлені результати моделювання описаного дільника частоти.

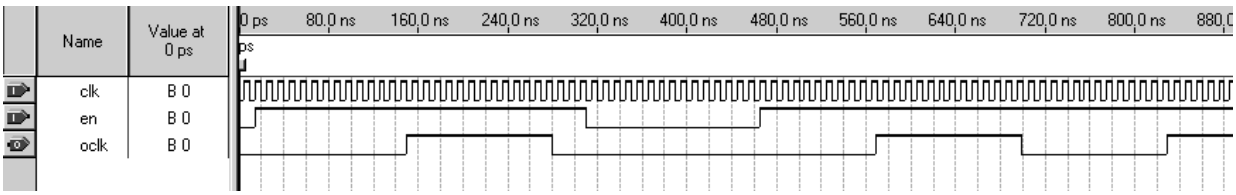

Рис.2.40. Діаграма роботи дільника частоти

Модуль lpm_constant можна використовувати в якості коефіцієнту множення або ділення при виконанні відповідних операцій. Наприклад, при розробці цифрових фільтрів, коефіцієнти фільтру мають дробні значення. Для синтезу фільтра потрібно виконувати масштабування, тобто потрібно перемножити коефіцієнти фільтру на коефіцієнт (наприклад 1000). Після обробки даних вихідний результат також потрібно масштабувати (розділити на 1000). Цей модуль також можна використати в якості вхідного коду керованого дільника частоти, який було розглянуто вище. Модуль lpm_constant має два параметри: LPM_WIDTH – кількість розрядів для представлення константи та LPM_CVALUE – значення константи. Саме цими параметрами вказується значення числа, яке буде встановлено на виході модуля.

2.8. Регістри

Регістрами називаються послідовністні пристрої, призначені для збереження та перетворення інформації, поданої у вигляді багаторозрядних двійкових чисел. Регістри, по суті, представляють собою декілька D-тригерів, з'єднаних між собою тим чи іншим способом. Тому принципової різниці між ними та D-тригерами не існує. Правда, тригери, що входять до складу регістрів, не мають такої кількості різноманітних керуючих входів, як одиночні тригери. Кількість тригерів визначається кількістю розрядів числа, яке може бути записане в регістр. Регістри звичайно будуються таким чином, щоб нарощування їх розрядності не викликало складності.

В залежності від функціональних можливостей, регістри поділяються на два типи: накопичувальні (регістри пам'яті) та послідовні (регістри зсуву). Існують також регістри інших типів, але вони застосовуються значно рідше, ніж регістри пам'яті і регістри зсуву, оскільки мають спеціальне призначення. В свою чергу послідовні регістри поділяються: по способу вводу та виводу інформації – на паралельні, послідовні та комбіновані; за напрямком зсуву (передачі) інформації – на однонапрямлені та реверсивні.

В паралельних регістрах кожен з тригерів має свій незалежний інформаційний вхід і свій незалежний інформаційний вихід. Тактові входи всіх тригерів з'єднані між собою. Тому паралельний регістр представляє собою багаторозрядний, багатовходовий тригер. Розглянемо один з варіантів опису паралельного регістра на мові Verilog.

```
module register(r_out, r_in, enable, load); // початок модуля
output [7:0] r_out;       // оголошення 8-розрядного вихідного порту
input [7:0] r_in;         // оголошення вхідного порту
input enable;             // вхід дозволу запису в регістр
input load;               // сигнал запису даних
```

```
reg [7:0] r_out;                    // вихід має тип регістр
always @ (posedge load)             // по фронту сигналу запису
begin
    if (enable) r_out<=r_in;        // записуємо дані в регістр
    else r_out<=0;  // якщо сигнал en має низький рівень, на виході 0
end
endmodule                           // кінець опису модуля
```

Результати моделювання описаного модуля демонструє часова діаграма на рис.2.41.

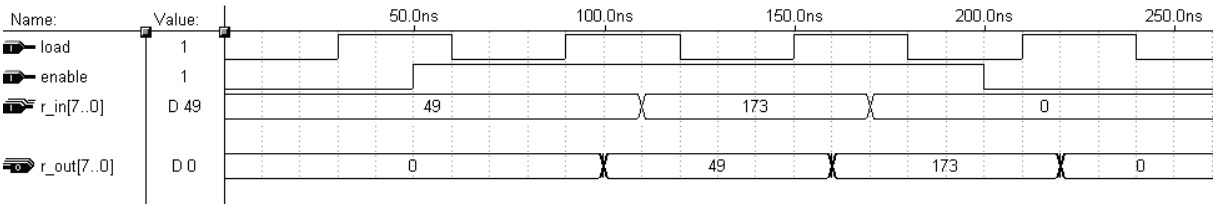

Рис.2.41. Моделювання роботи регістра

Аналогічним образом описується D-тригер. Різниця полягає в тому, що розрядність вхідного і вихідного портів дорівнює 8. Таким чином, ми описали 8 D-тригерів, які мають загальні сигнал дозволу і сигнал завантаження даних. По фронту сигналу load кожен з виходів r_out регістра встановлюється в той рівень, який був в цей момент на відповідному вході r_in, і зберігається до наступного фронту сигналу load. Тобто якщо тригер запам'ятовує один сигнал, то регістр запам'ятовує відразу декілька (4, 6, 8, 16) сигналів (декілька розрядів, бітів). Як бачимо, досить просто описати паралельний регістр із необхідною розрядністю. Для цього необхідно в оголошенні портів змінити всього лише одну цифру.

Паралельні регістри розділяються на дві групи:

- регістри, що спрацьовують по фронту керуючого сигналу (або тактовані регістри);
- регістри, що спрацьовують по рівню керуючого сигналу (або стробуємі регістри).

Найчастіше всього в цифрових схемах використовуються регістри, що керуються фронтом або спадом (тобто, тактуємі), хоча і стробуємі регістри мають своє коло застосування, в яких їх нічим не можна замінити. Описаний нами регістр відноситься до тих, які спрацьовують по фронту керуючого сигналу. Для того, щоб описати регістр, який буде спрацьовувати по рівню керуючого сигналу, потрібно змінити опис модуля наступним чином:

```
module register(r_out, r_in, enable, load);   // початок опису модуля
output [7:0] r_out;                 // 8-розрядний вихідний порт
input [7:0] r_in;                   // 8-розрядний вхідний порт
input enable;                       // вхід дозволу
input load;                         // сигнал завантаження даних
reg [7:0] r_out;                    // вихід має регістровий тип
      always @ (load or enable)     // при зміні керуючих сигналів
      begin
      if (enable && load) r_out<=r_in;      // завантажуємо дані
      else if (!enable && !load) r_out<=0;  // інакше на виході
      встановлюємо 0
      end
endmodule          // кінець модулю
```

У конструкції контролю подій **always** відсутнє ключове слово posedge. В такому випадку відслідковується будь-яка зміна рівня сигналу load або enable. Після цього виконується блок **begin**...**end**, в якому є оператор умови. Якщо сигнал на вході load приймає високий рівень і встановлений сигнал дозволу запису enable, відбувається запис даних у регістр. Якщо ці сигнали приймають стан низького рівня, на виході всі розряди встановлюються в 0. При інших можливих комбінаціях вхідних сигналів вихід зберігає свій стан. Часова діаграма, представлена на рис.2.42, демонструє це.

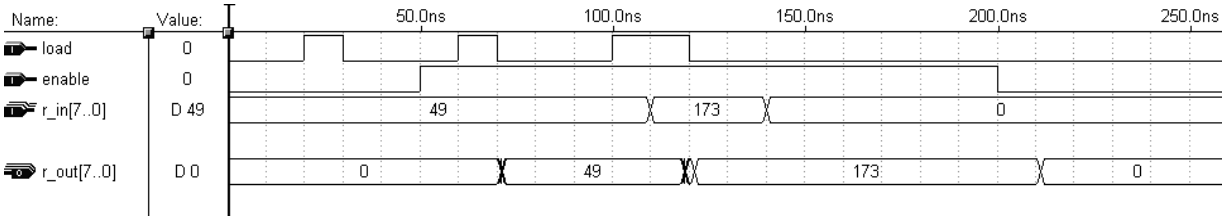

Рис.2.42. Робота паралельного регістра

Всі часові обмеження, що накладаються на вхідні сигнали тригерів, справедливі і для вхідних сигналів регістрів. Наприклад, не повинна бути занадто малою тривалість тактового сигналу, а також не повинна бути занадто малою затримка між встановленням сигналу на виході і приходом фронту сигналу запису. Інакше робота регістра може бути нестабільною, або навіть неправильною.

Одним з основних призначень регістрів є зберігання даних протягом деякого часу. Якщо для роботи частини схеми необхідно мати вхідний код, який можна легко змінювати, то можна використати регістр. Не менш важливе застосування регістрів пов'язано із запам'ятовуванням декількох послідовних значень змінного вхідного коду. Це дозволяє, наприклад, порівнювати попереднє значення коду з наступним значенням цього ж

коду і виконувати арифметичні операції над декількома послідовними значеннями одного і того ж коду. Тобто регістр в даному випадку виступає як елемент лінії затримки, що зберігає в собі історію поведінки вхідного коду. При розробці цифрового КІХ-фільтру ми опишемо регістр, який виконує саме таку функцію. Регістри також широко застосовуються для організації конвеєрної обробки, що дозволяє суттєво підвищити частоту роботи схеми. Прискорення при цьому досягається за рахунок паралельної роботи декількох послідовно увімкнених вузлів схеми. Регістри можуть застосовуватися в складі обчислювачів, виконуючи роль накопичувача результату обчислень. В даному випадку ми вже маємо справу з більш складною обробкою інформації, ніж у випадку з комбінаційними схемами. З кожним тактом у регістрі поновлюється інформація, яка є результатом математичної обробки вхідного коду і результату попереднього обчислення.

В наступному прикладі приводиться опис модуля 24-розрядного регістра, який створено на основі двох параметризованих модулів тригерів „lpm_ff".

```
module reg24(d, clk, q); // початок модулю
input   [23:0] d;          // вхідні дані
input   clk;               // вхід тактового сигналу
output  [23:0] q;          // вихідна шина
                           // створення екземплярів модулів
        lpm_ff reg12a (.q (q[11:0]), .data (d[11:0]), .clock (clk));
        defparam reg12a.lpm_width = 12;    // розрядність даних 12 біт
        lpm_ff reg12b (.q (q[23:12]), .data (d[23:12]), .clock (clk));
        defparam reg12b.lpm_width = 12;
endmodule                  // кінець опису модуля
```

Розроблений модуль має всього три порти: вхід даних d, вхід тактового сигналу clk та вихід даних q. Приклад демонструє одну з можливостей використання шини. Так, вхідна 24-розрядна шина даних розділена на дві 12-розрядні шини, кожна з яких поступає на вхід параметризованого модуля. Змінюючи номери елементів шини можна поміняти порядок бітів на виході модуля. Хоч даний приклад досить простий, використання функції lpm_ff дозволяє наділити модуль додатковими сигналами керування, які відсутні в модулях тригерів в бібліотеці САПР, наприклад синхронний або асинхронний входи обнуління, вхід завантаження даних та інші.

У регістрах зсуву всі тригери з'єднані в послідовний ланцюг (вихід кожного попереднього тригера з'єднаний із входом наступного тригера). Тактові входи всіх тригерів з'єднані між собою. В результаті такий тригер може розглядатися як лінія затримки, вхідний сигнал якої перезаписується

із тригера в тригер по фронту або спаду тактового сигналу. Інформаційні входи та виходи регістра можуть бути виведені зовні, а можуть і не виводитися в залежності від функцій, що виконує регістр. Розглянемо опис регістра зсуву на мові Verilog.

```
module par_to_ser(r_out, all, r_in, clk, enable, load); // початок модуля
output r_out;                    // опис вихідних портів
output all;
input [7:0] r_in;                // опис вхідних портів
input clk, enable, load;
reg r_out;                       // вихід має регістровий тип
reg all;
integer count;                   // оголошення додаткових змінних
reg [7:0] temp;
always @ (posedge clk)           // по фронту тактового сигналу
begin                            // виконуємо процедурний блок
if (load)                        // якщо на вході load високий рівень
        begin
                temp<=r_in;      // записуємо дані в регістр
                count<=0;        // обнуління змінної
                r_out<=0;        // на виході низький рівень сигналу
                all<=0;
        end
if (enable)                      // якщо встановлено сигнал дозволу
        begin
                r_out=temp[0]; // молодший біт установлюємо на виході
                temp=temp>>1;    // виконуємо зсув вправо
                count=count+1;   // збільшуємо значення лічильника
        if (count==8)            // якщо значення лічильника дорівнює 8
                begin
                        count<=0;
                        all<=1;  // встановлюємо на виході 1
                end
        else all=0;   // інакше на виході all встановлюємо низький рівень
        end
else                    // якщо сигнал дозволу має низький рівень
        begin
                count<=0;        // обнуління змінних
                 all<=0;       // на виході встановлюємо низький рівень
                r_out<=0;
        end
end
endmodule                        // кінець модуля
```

Як бачимо, регістр зсуву, що описаний в даному випадку, немає виводів із всіх тригерів, а має вихід лише з останнього. Описаний регістр має вхід load. При високому рівні сигналу на цьому вході виконується запис даних у регістр по фронту тактового сигналу clk. Вихід all приймає високий рівень після 8 тактів сигналу clk. Цей вихід можна використати для завантаження в регістр наступного байту даних, якщо модуль використовується для перетворення коду з паралельного формату в послідовний і передачі цього коду. Результати моделювання роботи регістра представлені на рис.2.43.

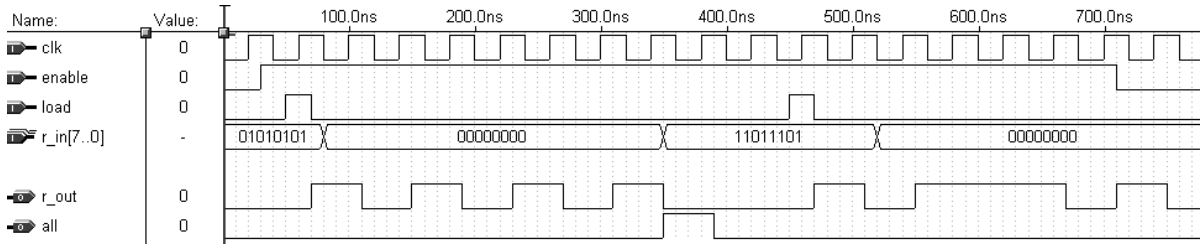

Рис.2.43. Часова діаграма роботи регістра зсуву

Ще одним цікавим типом регістра є регістр зсуву, який перетворює послідовний формат передачі даних у паралельний. Текст опису модуля на мові Verilog представлений нижче.

```
module shiftreg(out_reg, ready, in_reg, shift, enable);
output [7:0] out_reg;          // оголошення вихідних портів
output ready;                  // вихід готовності даних
input in_reg;                  // оголошення вхідних портів
input shift, enable;
reg [7:0] out_reg;             // вихідний 8-розрядний порт має тип регістр
reg ready;
reg [2:0] count;               // оголошення додаткових змінних
reg [7:0] temp;
always @(posedge shift)        // по фронту сигналу shift
begin                          // виконуємо блок
    if (enable)                // якщо встановлено сигнал дозволу
    begin
    temp[0]=in_reg;    // нульовий біт регістра temp приймає
                       // рівень значення на вхідному порту
    out_reg=temp;      // вихід приймає значення регістра temp
    temp=temp<<1;      // зсув даних
    count=count+1;     // збільшуємо на 1 значення лічильника
    if (count==3'b111) ready=1;     // якщо виникло переповнення
```

```
            // встановлюємо високий рівень на виході ready
        else ready=0;  // інакше вихід ready має низький рівень
        end
        else          // якщо сигнал дозволу має низький рівень
        begin
        count=0;      // обнуління значення змінних
        out_reg=0;
        //сигнали на вихідних портах приймають низький рівень
        ready=0;
        end
end
endmodule         // кінець опису модуля
```

Робота регістра дуже проста. На вхід in_reg подається двійковий код у послідовному форматі. На виходах out_reg[7] – out_reg[0] з'являється цей код, причому зсув відбувається по фронту сигналу shift. При наявності низького рівня на вході enable, на виходах регістра встановлюються логічні нулі. Результати моделювання роботи регістра представлені на рис.2.44.

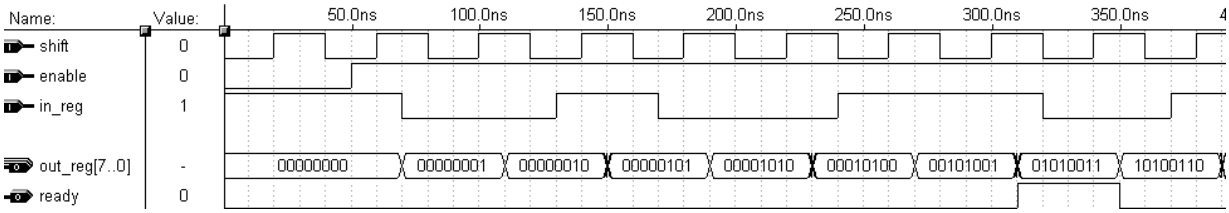

Рис.2.44. Діаграма роботи регістра зсуву

Головне використання всіх регістрів зсуву полягає в перетворенні паралельного коду в послідовний і навпаки. Таке перетворення використовується, наприклад, при передачі інформації на великі відстані (в інформаційних мережах), при запису інформації на магнітні носії, при роботі з телевізійними моніторами та з відеокамерами, а також у багатьох інших випадках. Ще одне застосування регістрів зсуву полягає в організації всеможливих ліній затримок, особливо тих, які мають велику кількість каскадів. За допомогою регістра зсуву можна забезпечити затримку будь-якого сигналу на ціле число тактів. Правда, потрібно враховувати, що тривалість вхідного сигналу буде також передаватися по лінії затримки з точністю до одного такту. Такі лінії затримки можуть застосовуватися для порівняння декількох наступних тактів вхідного сигналу, для виконання арифметичних операцій з декількома тактами вхідного сигналу, для інших подібних цілей. Регістри зсуву також можуть бути використані для множення або ділення двійкових чисел на 2^n, де n –

ціле число. Зсув двійкового числа в бік молодших розрядів на один розряд рівнозначний діленню на 2. Зсув двійкового числа в бік старших розрядів рівнозначний множенню на 2. Для того, щоб регістр перемножував та ділив двійковий код, потрібно записати цей код у регістр і зсунути його на потрібну кількість разів вліво або вправо.

Розробимо в САПР Quartus параметризований модуль, який би виконував операції арифметичного, логічного та циклічного зсуву. При арифметичному зсуві вмісту регістра ліворуч, молодші біти заповнюються нулями, а старші губляться. При арифметичному зсуві праворуч молодші біти губляться, а старші заповнюються розширенням знаку. Логічний зсув відрізняється від арифметичного тим, що при зсуві праворуч старші біти заповнюються не розширенням знаку, а нулями. При зсуві ліворуч арифметичний і логічний зсув виконуються однаково. Циклічний зсув відрізняється від арифметичного і логічного тим, що біт, який „виходить" з однієї сторони, автоматично переміщується на місце біта з протилежної сторони, тим самим утворюючи кільце.

```
module shifter(data, distance, direction, clk , result, overflow, underflow);
    parameter DATA_WIDTH=8;          // розрядність шини даних
    parameter DISTANCE_WIDTH=8; // розрядність величини зсуву
    parameter TYPE="LOGICAL";        // тип зсуву
    input [DATA_WIDTH-1:0] data;     // вхідний порт даних
    input [DISTANCE_WIDTH-1:0] distance; // кількість бітів для зсуву
    input clk, direction;            // тактовий вхід та напрям зсуву
    output [DATA_WIDTH-1:0] result; // результат
    output overflow, underflow;      // виходи переповнення
    integer i;                       // додаткова змінна для циклу
    reg [DATA_WIDTH-1:0] temp;       // додатковий регістр
    reg [DATA_WIDTH-1:0] result;     // вихід має регістровий тип
    reg overflow, underflow;
    reg flag;                        // для збереження знакового біту
    always@(posedge clk)             // по фронту сигналу clk
    begin
        underflow=0;            // виходи переповнення встановлені в 0
        overflow=0;
        case(TYPE)                   // вибір типу зсуву
        "LOGICAL":                   // логічний зсув
        begin
        if (direction==0)   // в залежності від сигналу на вході напряму
        begin                   // виконуємо зсув праворуч або ліворуч
            temp=data;          // в temp записується вхідне слово
        for(i=0;i<8;i=i+1)      // зсув вмісту регістра виконується
```

```
if (i<distance)              // distance разів
begin
underflow=temp[0]|underflow; // визначення сигналу
temp=temp>>1;                // зсув праворуч
temp[DATA_WIDTH-1]=0;// старший біт встановлено в 0
end
overflow<=0;       // встановлення сигналу переповнення в 0
result<=temp;      // передача результату на вихід
end
else
begin
        temp=data;          // запис вхідних даних в temp
        for (i=0;i<8;i=i+1)// в циклі виконується операція зсуву
        if (i<distance)     // при виконанні умови
begin                       // виконується блок
        overflow=temp[DATA_WIDTH-1]|overflow;
        temp=temp<<1;   // зсув ліворуч
        temp[0]=0;          // молодший біт встановлено в 0
end
underflow<=0;      // встановлення сигналу переповнення в 0
result<=temp;      // передача результату на вихід
end
end
"ROTATE":          // циклічний зсув
begin
if (direction==0)  // в залежності від напряму виконується
        begin      // зсув праворуч або ліворуч
        temp=data;
        for (i=0;i<8;i=i+1)
        if (i<distance)
        begin
        underflow=temp[0];      // запам'ятовуємо молодший біт
        temp=temp>>1;           // зсув праворуч
        temp[DATA_WIDTH-1]=underflow;
                // встановлюємо старший біт
        end
        overflow<=0;       // сигнал переповнення має значення 0
        result<=temp;      // передача результату на вихід
        end
        else
        begin
        temp=data;          // записуємо вхідне слово
        for (i=0;i<8;i=i+1)
```

```
        if (i<distance)
        begin
                        // запам'ятовуємо старший біт
        overflow=temp[DATA_WIDTH-1];
        temp=temp<<1;   // зсув ліворуч
        temp[0]=overflow; // встановлення молодшого біту
        end
        underflow<=0;   // сигнал переповнення має рівень 0
        result<=temp;   // передача результату на вихід
        end
        end
"ARITHMETIC":           // арифметичний зсув
begin
flag=data[DATA_WIDTH-1]; // запам'ятовуємо знаковий біт
if (direction==0)       // перевірка напряму зсуву
begin
        temp=data;      // запис вхідного коду в регістр temp
        for (i=0;i<8;i=i+1)
        if (i<distance)
begin
temp=temp>>1;   // зсув праворуч
temp[DATA_WIDTH-1]=flag;//встановлення старшого біту
end
        underflow<=flag;  // установка сигналів на виході
        overflow<=0;
        result<=temp;
end
else
begin
        temp=data;      // записуємо вхідне слово
        for (i=0;i<8;i=i+1)
        if (i<distance)     // якщо виконано менше distance зсувів
begin
        temp=temp<<1;   // зсув ліворуч
        temp[0]=0;      // молодший біт дорівнює 0
end
        underflow<=0;   // результати зсуву встановлюються
        overflow<=0;    // на вихідних портах
        result<=temp;
end
end
default:                // якщо тип зсуву невідомий
        begin
```

```
            result<=data;       // на виході встановлюється вхідне слово
            overflow<=0;        // сигнали переповнення встановлені в 0
            underflow<=0;
                  end
            endcase             // кінець конструкції вибору
      end
      endmodule                 // кінець опису модуля
```

Параметр TYPE визначає тип операції зсуву. За допомогою параметру DATA_WIDTH визначається розрядність слів, з якими працює модуль. В залежності від типу операції зсуву виконується той чи інший блок в конструкції вибору. В циклі **for** встановлено значення умови i<8. Правильніше було б замість жорсткого задання умови встановити значення distance – кількість однобітових операцій зсуву. Однак в циклі неможна задати не константне значення. Для вирішення цієї проблеми введена перевірка значення змінної *i* та кількості операцій зсуву distance.

На рис.2.45 представлена часова діаграма, яка демонструє роботу модуля при логічному зсуві.

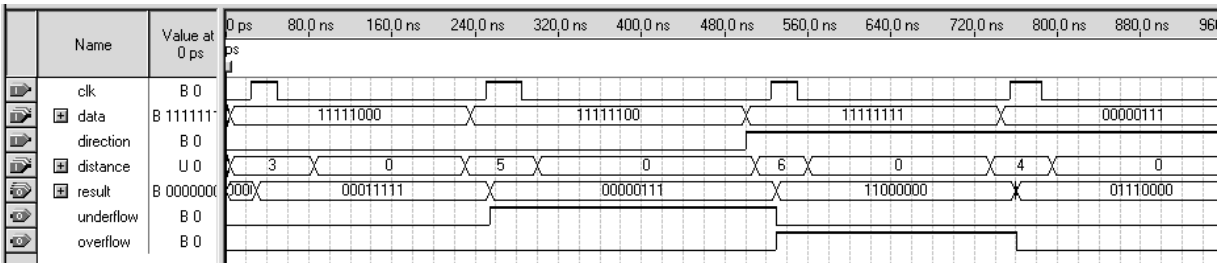

Рис.2.45. Логічний зсув даних

На рис.2.46 представлена часова діаграма, яка демонструє роботу модуля при циклічному зсуві.

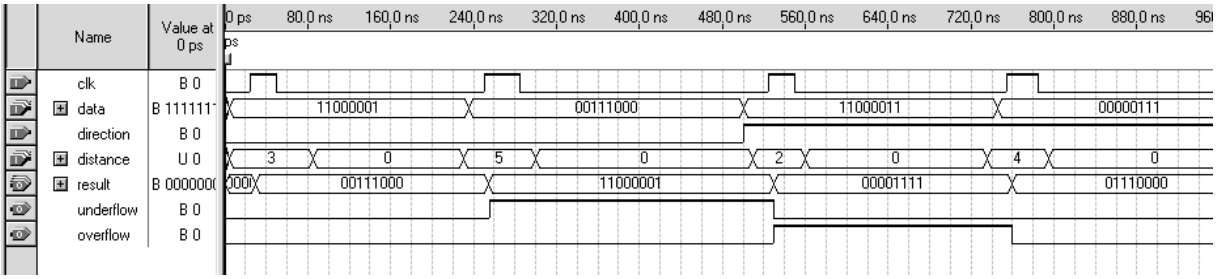

Рис.2.46. Циклічний зсув даних

На рис.2.47 представлена часова діаграма, яка демонструє роботу модуля при арифметичному зсуві.

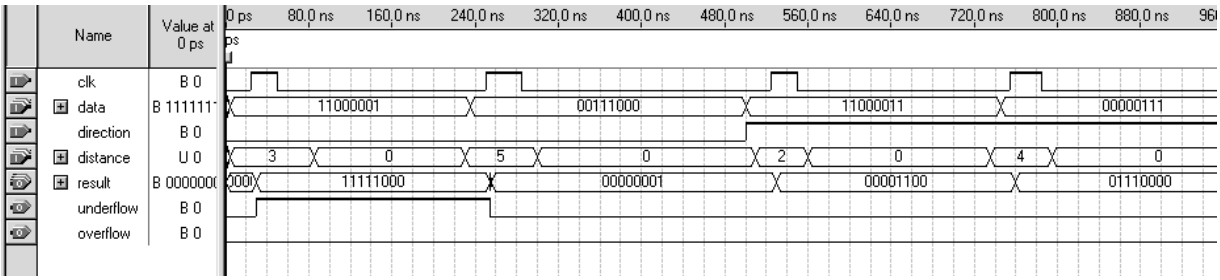

Рис.2.47. Арифметичний зсув даних

Для використання розробленого модуля зручно створити з нього символ. В САПР Quartus для створення графічного символу з текстового опису потрібно в меню File вибрати з підменю Create/Update команду Create Symbol Files for Current File. На рис.2.48 представлений вигляд створеного символу.

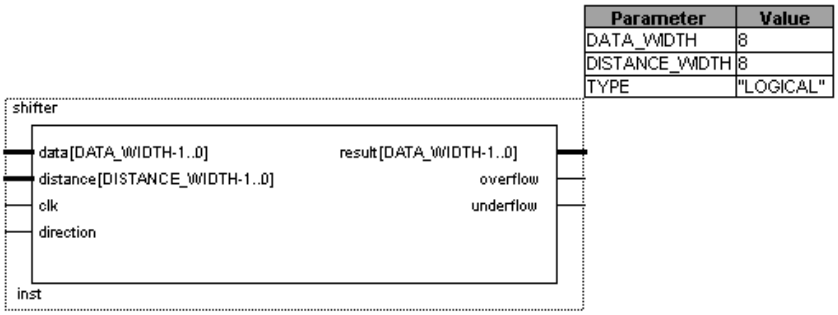

Рис.2.48. Створений символ модуля зсуву

Подвійне натискання лівою кнопкою миші на полі з параметрами призведе до появи діалогового вікна, вигляд якого приводиться на рис.2.49.

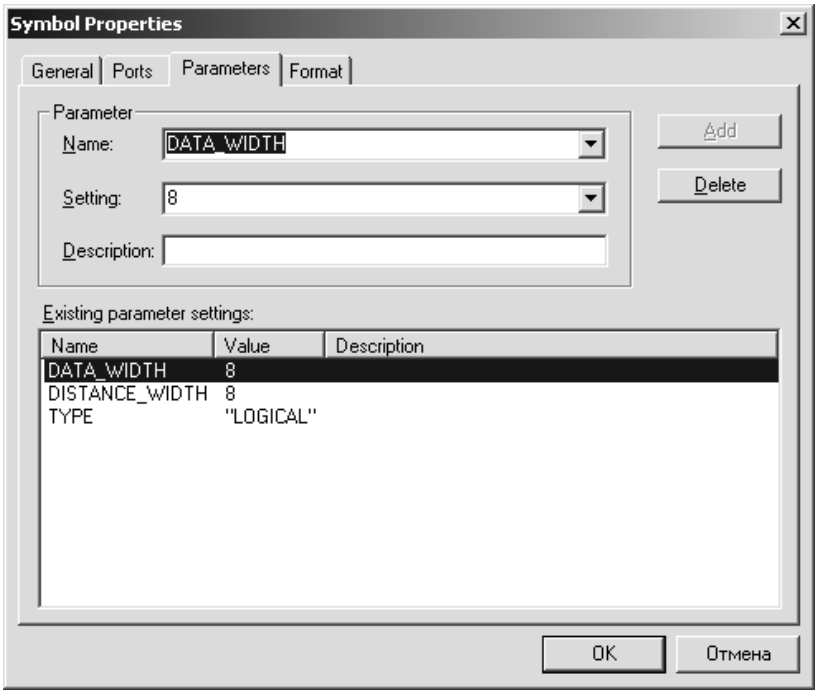

Рис.2.49. Діалогове вікно Symbol Properties

На вкладці Parameters є список з параметрами, яких в розробленому модулі три. В полі вводу Setting вказується значення того параметру, який виділений в списку Existing parameter settings. В полі Description розробник може ввести коментар до параметру, наприклад його призначення. На вкладинці Ports (рис.2.50) розташовані елементи керування портами модуля.

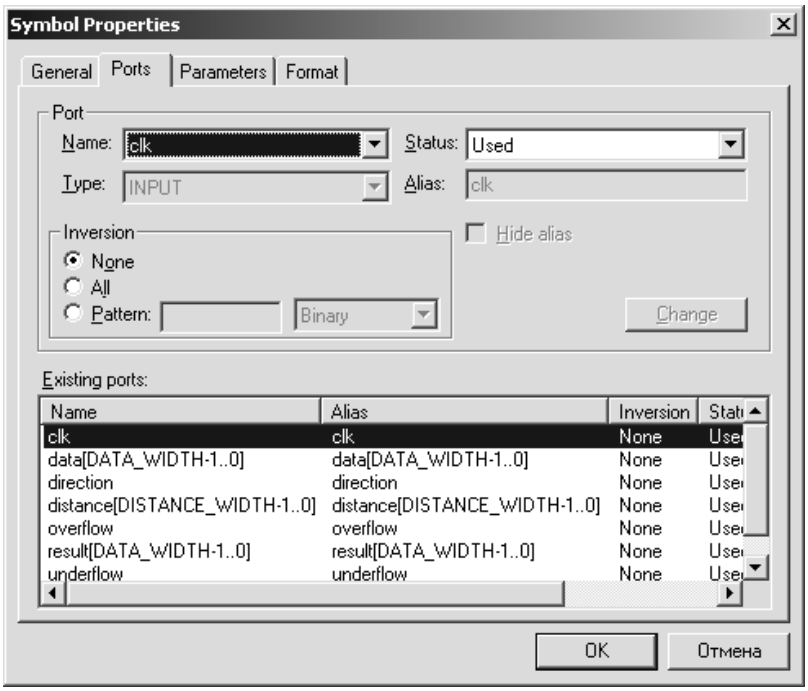

Рис.2.50. Закладка Ports

Виділивши назву виводу в списку Existing ports, можна відключити його. Для цього у списку Status потрібно вибрати значення Unused і натиснути на кнопку Change, яка стане активною. Зображення виводу на символі елемента зникне. Зміною положення перемикача в групі Inversion розробник може встановити інверсію даних на вибраному порту. На вкладці General є одне текстове поле, в якому вказується назва екземпляру модуля. За допомогою елементів керування на вкладці Format діалогового вікна розробник може змінити кольори відображення створеного символу.

В бібліотеці обох САПР є модуль lpm_clshift, який виконує різні операції зсуву вхідного слова. Саме цей модуль потрібно використовувати в MAX+plus II, оскільки в запропонованому варіанті є конструкції, які не підтримуються цією САПР.

Для побудови регістрів зсуву може бути вдало використаний цикл **for**. Крім того зустрічаються задачі, в яких необхідно мати шинний регістр зсуву. В наступному прикладі, який створений у САПР Quartus, приводиться опис 64-бітного регістра зсуву, який працює з 8-розрядними даними.

```
module bushift (clk, shift, sr_in, sr_out, sr_tap_one, sr_tap_two,
sr_tap_three,);
parameter DATA_WIDTH=8; // параметр розрядності даних
input clk, shift;                  // тактовий вхід та вхід дозволу зсуву
input [DATA_WIDTH-1:0] sr_in;      // вхідний порт
output  [DATA_WIDTH-1:0] sr_tap_one, sr_tap_two, sr_tap_three,
sr_out;
reg [DATA_WIDTH-1:0] sr [63:0];
// пам'ять для збереження вхідних даних
integer n;                         // змінна для використання в циклі
        always @(posedge clk)  // процедурний блок
        begin
        if (shift == 1'b1)         // якщо дозволений зсув даних
        begin
                for (n = 63; n>0; n = n-1) // виконується цикл 63 рази
                begin
                sr[n] <= sr[n-1];    // зсув даних
                end
                sr[0] <= sr_in;      // вхідне слово записується в комірку 0
                end
        end
assign sr_tap_one = sr[15];   // на першому виході значення 16 комірки
assign sr_tap_two = sr[31];   // на другому виході значення 32 комірки
assign sr_tap_three = sr[47];// на третьому виході значення 48 комірки
assign sr_out = sr[63];        // на виході значення останньої комірки
endmodule                      // кінець опису модуля
```

Розрядність даних вказується за допомогою параметру DATA_WIDTH, тому при необхідності може бути змінена. Модуль регістра зсуву має рівновіддалені отводи: sr_tap_one, sr_tap_two, sr_tap_three. Вхідний код на цих виводах з'являється через 16, 32 та 48 тактів відповідно. Після 64 тактів дані з'являються на виході sr_out регістра в тому ж порядку. Робота регістра можлива при високому рівні сигналу на вході shift. Часова діаграма, що представлена на рис.2.51, демонструє роботу модуля.

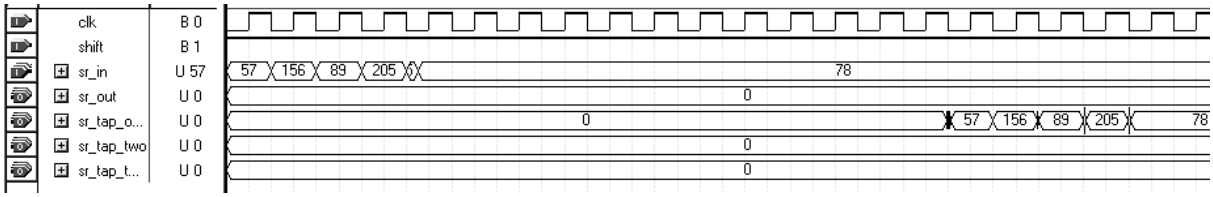

Рис.2.51. Демонстрація роботи регістра зсуву

З приведеного опису регістра зсуву можна створити графічний символ, який в подальшому буде використаний при розробці якого-небудь проекту у графічному редакторі. На рис.2.52 представлений вигляд символу регістра.

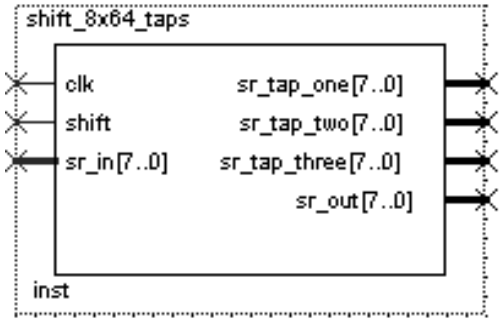

Рис.2.52. Графічний символ регістра зсуву

Нагадаємо, для створення символу в САПР Quartus потрібно в меню File, з підменю Create/Update вибрати команду Create Symbol Files for Current Project. Необхідні файли будуть автоматично створені і розташовані в каталозі проекту.

2.9. Оперативна пам'ять

Пам'ять, як витікає з назви, призначена для запам'ятовування, зберігання масивів інформації, цифрових таблиць, груп цифрових кодів. Кожний код зберігається в окремому елементі пам'яті, що називається коміркою пам'яті. Основна функція будь-якої пам'яті якраз і полягає у видачі цих кодів на виходи мікросхеми по зовнішньому запиту. Основний параметр пам'яті – це її об'єм, тобто кількість кодів, що можуть у ній зберігатися, а також розрядність цих кодів. Синтаксис опису пам'яті на Verilog має наступний вигляд:

reg [<розрядність елементів>] <ідентифікатор> [<кількість елементів>];

Принцип організації пам'яті записується наступним чином: спочатку записується розрядність коду, що зберігається в одній комірці, а потім записується кількість комірок. Нагадаємо, що приведена конструкція пам’яті підтримується компілятором Verilog тільки в САПР Quartus.

В залежності від способу занесення інформації в пам'ять та від способу зберігання інформації мікросхеми пам'яті розділяються на наступні основні типи:

- Постійна пам'ять. Інформація записується один раз на етапі програмування мікросхеми (заноситися користувачем) і не знищується при вимиканні живлення.
- Оперативна пам'ять, запис інформації в яку найбільш простий і може бути організований користувачем будь-яку кількість разів протягом

всього періоду роботи мікросхеми. Інформація в пам'яті знищується при вимиканні живлення.

В загальному випадку будь-яка мікросхема пам'яті має наступні інформаційні виводи: адресні виводи, виводи даних, керуючі виводи. Для демонстрації прикладів з елементами пам'яті буде використана система проектування Quartus. Розглянемо опис на мові Verilog модуля пам'яті на вісім напівбайтових слів. Текст опису модуля приводитися нижче.

```
module memor(out, in, addr, clk, enable);
output [3:0] out;              // визначаємо вихідний порт
input [3:0] in;                // вхідний інформаційний порт
input [2:0] addr;              // вхідний адресний порт
input clk;                     // тактовий сигнал
input enable;                  // сигнал «запис / зчитування»
reg [3:0] temp[7:0];           // пам'ять на зберігання 8 слів
reg [3:0] out;                 // вихідний порт має регістровий тип
always @ (posedge clk)         // по фронту тактового сигналу
begin
if (enable)                    // якщо встановлено сигнал запису
        begin
                case (addr)    // записуємо слово в зазначену комірку
                0: temp[0]=in;
                1: temp[1]=in;
                2: temp[2]=in;
                3: temp[3]=in;
                4: temp[4]=in;
                5: temp[5]=in;
                6: temp[6]=in;
                7: temp[7]=in;
                default: temp[4]=0;
                endcase
        end
else                           // якщо встановлено сигнал читання
        begin
                case (addr)    // зчитуємо зазначену комірку
                        0: out=temp[0];
                        1: out=temp[1];
                        2: out=temp[2];
                        3: out=temp[3];
                        4: out=temp[4];
                        5: out=temp[5];
                        6: out=temp[6];
                        7: out=temp[7];
```

```
            default: out=0;
            endcase
        end
    end
    endmodule                    // кінець опису модуля
```

В описанні модуля було використано конструкцію вибору **case**. Кожного разу по фронту сигналу clk, в залежності від коду на адресних входах, відбувається вибір комірки пам'яті. Рівнем сигналу на вході enable вибирається режим роботи мікросхеми (запис/зчитування). Оперативна пам'ять буває двох основних видів: з розділеними шинами вхідних і вихідних даних (в основному це однорозрядна пам'ять) і з двонапрямленою шиною вхідних і вихідних даних (це багаторозрядна пам'ять). Ми розглянули опис модуля пам'яті з розділеними шинами вхідних та вихідних даних. Часова діаграма, отримана при моделюванні, приводитися на рис.2.53.

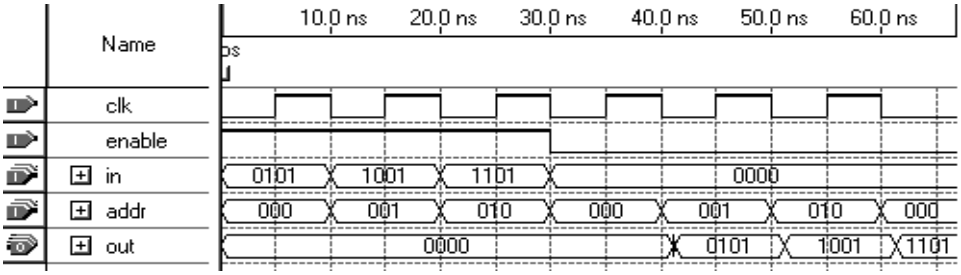

Рис.2.53. Результати моделювання роботи ОЗП

Це лише один із способів опису модуля пам'яті. Легко бачити, що даний спосіб має суттєвий недолік: необхідно описати всі можливі альтернативи в конструкції вибору і при збільшенні об'єму пам'яті пропорційно збільшується об'єм коду. Такий підхід зводить нанівець всю зручність мови Verilog. Розглянемо приклад, в якому доступ до комірки пам'яті відбувається іншим способом. Опис параметризованого модуля ОЗП, створеного у САПР Quartus, представлений нижче.

```
module mem (DataOut ,Addr, DataIn, CS, WE, OE); // заголовок модуля
parameter ADDRESS_SIZE = 4;      // розрядність адресної шини
parameter WORD_SIZE = 8;         // розрядність шини даних
output [WORD_SIZE-1:0] DataOut; // вихідна шина даних
input [ADDRESS_SIZE-1:0] Addr;  // вхідна адресна шина
input [WORD_SIZE-1:0] DataIn;    // вхідна шина даних
input CS, WE, OE;                // входи керування
reg [WORD_SIZE-1:0] DataOut;     // вихід має регістровий тип
reg [WORD_SIZE-1:0] Mem [ADDRESS_SIZE-1:0];
```

```
    always @(CS or WE or OE)    // при зміні сигналів керування
    begin
    if (!CS && !WE) Mem[Addr] = DataIn; // запис даних в ОЗП
    else if (!CS && !OE) DataOut=Mem[Addr]; // зчитування даних
    end
endmodule                       // кінець опису модуля
```

Запис у комірку пам'яті, яка задається значенням на адресному вході Addr, відбувається при низькому рівні сигналів CS та WE (вибір мікросхеми та дозвіл запису). Зчитування даних відбувається при низькому рівні сигналів на входах CS та OE (вихід дозволений). Зміна станів зазначених сигналів відслідковується в конструкції **always**. Доступ до комірки пам'яті Mem відбувається по аналогії з доступом до елементу масиву у традиційних мовах програмування. Діаграма на рис.2.54 демонструє роботу модуля ОЗП.

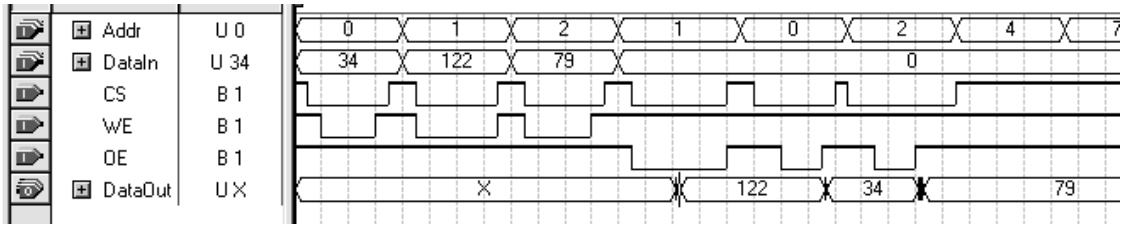

Рис.2.54. Діаграма роботи ОЗП

При роботі з пам'яттю є можливість доступу до визначеної комірки від деякого пристрою або від іншої частини схеми як з операцією запису, так і з операцією зчитування.

Розглянемо ще один приклад використання пам'яті. Наступний модуль представляє з себе детектор максимального (max) та мінімального (min) значень вхідного коду. Крім того, модуль має вихід, величина на якому представляє з себе усереднене значення останніх 9 відліків вхідного коду (din). Ці відліки зберігаються у пам'яті (memory). Опис модуля приводиться нижче.

```
module maxmin(max, min, avg, din, clk, en, reset); // заголовок модуля
parameter WIDTH=10;             // розрядність даних
parameter DEPTH=10;             // кількість комірок пам'яті
output [WIDTH-1:0] max, min, avg; // вихідні порти
input [WIDTH-1:0] din;          // вхід даних
input clk, en, reset;  // тактовий вхід, дозволу роботи та обнуління
reg [WIDTH-1:0] max, min, avg;      // виходи мають регістровий тип
reg [WIDTH:0] tavg;     // змінна для зберігання середнього значення
reg [WIDTH-1:0] memory [DEPTH-1:0];
```

```
                                    // пам'ять на DEPTH-1 елементів
reg [WIDTH-1:0] tmax, tmin;  // регістри для зберігання значень
integer i;                          // змінна для роботи циклу
always @(posedge clk or posedge reset)
                                    // перший процедурний блок
begin
if (reset==1)                       // якщо виконується умова
begin
        tmax=0;             // обнуління змінної
        tmin=10'hFFF;       // присвоєння максимального значення
        tavg=0;             // середнє значення дорівнює 0
for(i=DEPTH-1;i>=0;i=i-1) memory[i]=0; // очищення пам'яті
end   // if
else if(en==1)                      // якщо встановлений сигнал дозволу
begin
        tmax=(tmax>din)?tmax:din; // перевірка максимального значення
        tmin=(tmin<din)?tmin:din; // перевірка мінімального значення
for(i=DEPTH-1;i>0;i=i-1) // в циклі відбувається зсув
begin                               // даних на 1 комірку
        memory[i]=memory[i-1];
        tavg=(tavg+memory[i])>>1;
end                                 // for
        memory[0]=din;
        // нульовій комірці присвоюється значення на вході
end                                         // else if
end
always @(tmax or tmin or tavg)      // другий процедурний блок
begin
        max<=tmax;          // установка на виходах максимального
        min<=tmin;          // мінімального
        avg<=tavg;          // та середнього значень
end
endmodule
```

Параметри WIDTH та DEPTH визначають розрядність вхідних даних і кількість комірок пам'яті. По сигналу сбросу reset відбувається обнуління внутрішніх змінних, а також сигналів на виходах модуля. На виході min встановлюється максимальне значення. По фронту тактового сигналу clk відбувається порівняння вхідних даних зі значеннями, які зберігаються у внутрішніх змінних. При виконанні умови в процедурному блоці, на виходах встановлюються максимальне, мінімальне та середнє значення. За допомогою циклу відбувається зсув вмісту комірок пам'яті. Значення на вході присвоюється комірці з адресою 0. Діаграма, що представлена на

рис.2.55, демонструє роботу детектора максимального та мінімального значень.

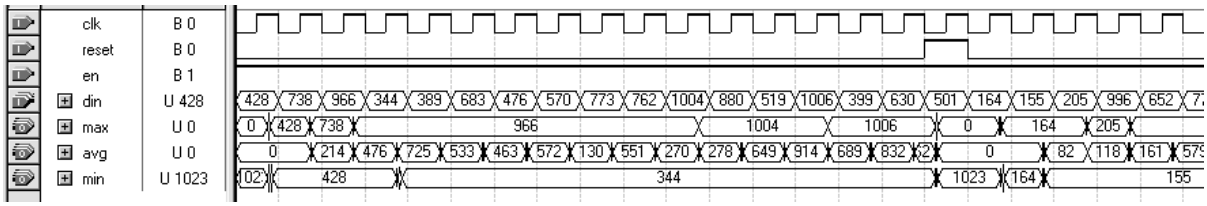

Рис.2.55. Діаграма роботи детектора

В залежності від того, в якому порядку може записуватися або зчитуватися інформація, існують два різновиди ОЗП:

- ОЗП з паралельним довільним доступ (це найбільш універсальна схема);
- ОЗП з послідовним доступом (це більш специфічна схема).

Паралельний (або довільний) доступ найбільш простий і не потребує ніяких додаткових елементів, оскільки саме на цей режим безпосередньо розраховані мікросхеми пам'яті. В цьому режимі можна записувати інформацію по будь-якій адресі ОЗП і зчитувати інформацію з будь-якої адреси ОЗП в довільному порядку. Однак паралельний доступ потребує формування досить складних послідовностей всіх вхідних сигналів пам'яті. Тобто для запису інформації необхідно сформувати код адреси комірки і лише потім подати дані, що супроводжуються керуючими сигналами запису інформації та вибору мікросхеми. Так само необхідно подавати повний код адреси комірки при операції зчитування. Цей режим найчастіше використовується в комп'ютерах та контролерах, де найголовнішими факторами є універсальність і гнучкість використання пам'яті для будь-якої мети.

Послідовний доступ до пам'яті пропонує більш простий порядок звернення до пам'яті. В цьому випадку не потрібно задавати код адреси комірки, в яку відбувається запис або зчитування даних, оскільки адреса пам'яті формується схемою автоматично. Для запису інформації потрібно подати код даних, що записуються і сигнал запису. Для зчитування інформації потрібно подати сигнал зчитування і отримати дані. Автоматичне задання адреси при цьому відбувається внутрішніми лічильниками, що змінюють свій стан при кожному зверненні до пам'яті. Наприклад, десять послідовних циклів запису запишуть інформацію в десять послідовно розташованих комірок пам'яті. Недолік такого підходу зрозумілий: ми не маємо можливості записувати або зчитувати дані з комірок з довільними адресами в будь-якому порядку. Однак суттєво спрощується і прискорюється процедура обміну з пам'яттю.

Можна виділити три основні типи оперативної пам'яті з послідовним доступом:

- пам'ять магазинного типу, стекового типу, що працює за принципом “останній увійшов – перший вийшов” (LIFO, Last In – First Out);
- пам'ять типу “перший увійшов – перший вийшов” (FIFO, First In – First Out);
- пам'ять для зберігання масивів даних.

Розглянемо опис модуля пам'яті типу LIFO у САПР Quartus. Текст програми має наступний вигляд:

```
module lifo_buf(out, full, empty, ind, clk, in, enable, push_pop);
output [3:0] out;              // вихідний порт
output full;                   // сигнал переповнення
output empty;                  // сигнал „буфер порожній”
output [3:0] ind;              // кількість елементів у буфері
input [3:0] in;                // вхідний порт
input enable;                  // вхід дозволу
input clk;                     // тактовий сигнал
input push_pop;                // запис(зчитування) в (з) буфера
reg [3:0] temp[7:0]; // пам'ять для зберігання 8 напівбайтових слів
reg [3:0] out;            // вихідні порти мають регістровий тип
reg full;
reg empty;
reg [3:0] ind;
integer index;                 // лічильник слів у буфері
always @(posedge clk)          // по фронту тактового сигналу
begin
      if (enable)              // якщо є присутнім сигнал дозволу
      begin
      if (push_pop==1)         // перевіряємо стан вхідного порту
            begin
                  if (index==7) full=1;     // якщо буфер заповнений,
          //установлюємо високий рівень на виході переповнення
                  else // інакше
            begin
                  temp [index]=in; // запис слова в пам’ять
                  index=index+1;   // збільшення значення індексу
                  ind=index;  // вивід на порт кількості слів в буфері
                  full=0;       // обнуління сигналу переповнення
                  empty=0;
            end
```

```
            end
        else
begin
        if (index>0)                    // перевіряємо, чи є в буфері дані
            begin                       // якщо є
                out=temp[index]; // зчитуємо слово з пам'яті
                index=index-1;   // зменшуємо індекс
                ind=index;       // встановлюємо на вихідному
                                 // порту значення елементів
                                 буфера
                empty=0;  // обнуління сигналу «буфер порожній»
            end
        else if (index==0)       // якщо буфер порожній
            begin
                empty=1;  // установка сигналу «буфер порожній»
            end
        end
end
else                             // якщо сигнал дозволу має рівень 0
        begin
            index=0;             // обнуління значення індексу
            out=0;               // установка сигналу низького рівня на
            full=0;              // вихідних портах
            ind=0;
            empty=1;
        end
end
endmodule                        // кінець опису модуля
```

Як зазначалося вище, формування адреси в такому типі пам'яті відбувається внутрішнім лічильником. В описаному модулі цим лічильником є змінна index. Встановлення значення цієї змінної в 0 відбувається по фронту сигналу clk при умові того, що вхід enable має низький рівень. Взагалі, при низькому рівні сигналу на цьому вході, всі вихідні порти модуля, окрім сигналу empty, встановлюються в рівень логічного нуля. Якщо сигнал на вході enable має високий рівень, відбувається збільшення, або зменшення (в залежності від стану вхідного порту push_pop) змінної index, а потім і звернення до визначеної комірки пам'яті. Якщо змінна index приймає значення 7, то вихідний порт full переходить в стан логічної одиниці, що сигналізує про те, що всі комірки пам'яті заповнені. Якщо змінна index дорівнює 0, то високий рівень сигналу empty сигналізує про те, що буфер пустий. Часова діаграма, яка демонструє роботу модуля, приводитися на рис.2.56.

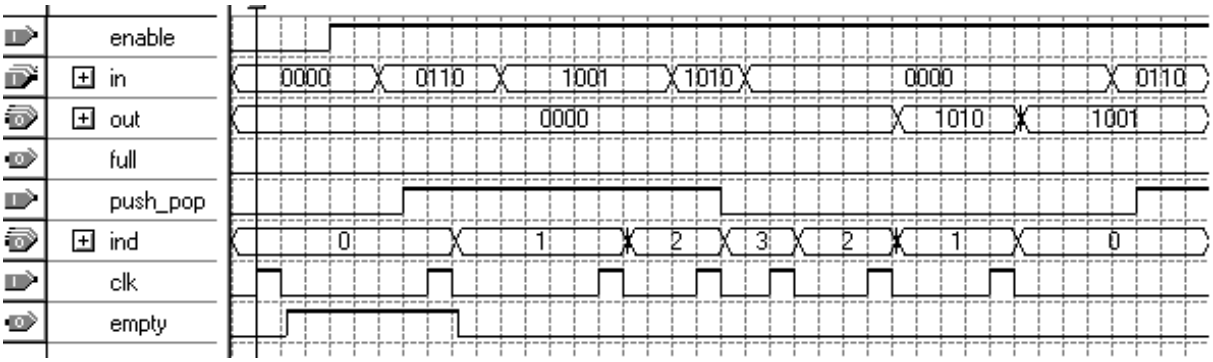

Рис.2.56. Результати моделювання роботи модуля LIFO

Як бачимо з часової діаграми, пам'ять LIFO видає дані у зворотній послідовності від тієї, у якій вони були записані. Пам'ять LIFO використовується в комп'ютерах (стек), де вона зберігає інформацію про параметри програм і підпрограм. Для пам'яті LIFO достатньо лише одного коду адреси.

Як зазначалося вище, система MAX+plus II, а точніше, підмножина мови Verilog, яка інтегрована в цю систему проектування, не підтримує тип даних пам'ять. Може виникнути питання: „чи можливий опис на мові Verilog різних модулів пам'яті, або для цього потрібно використовувати Графічний редактор системи?”. В системі MAX+plus II за допомогою Verilog можна описувати проекти, які містять елементи пам'яті. Разом із системою проектування поставляється бібліотека параметризованих модулів. Від розробника вимагається лише визначити кількість елементів пам'яті та розрядність цих елементів. Це робиться шляхом перевизначення параметрів за допомогою службового слова **defparam**. Розглянемо опис модуля елемента пам'яті.

```
module ram256x8 (data, address, we, inclock, outclock, q);
input   [7:0] data;                  // вхідна шина даних
input   [7:0] address;               // адресна шина
input   we, inclock, outclock;       // керуючі сигналі
output  [7:0] q;                     // вихідна шина даних
lpm_ram_dq inst_1 (.q (q), .data (data), .address (address), .we (we),
.inclock (inclock), .outclock (outclock));
// використовуємо модуль пам'яті з бібліотеки
//  параметризованих модулів
defparam inst_1.lpm_width = 8;
// визначаємо розрядність шини даних
defparam inst_1.lpm_widthad = 8;
// визначаємо розрядність адресної шини
endmodule                            // кінець опису модуля
```

Нічого нового з синтаксичних конструкцій в наведеному описі немає. Все дуже просто. Спочатку визначаємо тип та розрядність зовнішніх портів модуля, а потім підключаємо модуль пам'яті з бібліотеки параметризованих модулів. Далі встановлюємо розрядність елементів (8) та їх кількість (256) в описуваному модулі. В цьому прикладі параметри lpm_width та lpm_widthad використані для визначення розрядності вхідної шини даних, а також кількості слів даних, що будуть розміщені в пам'яті (розрядність адресної шини). Функції lpm_ram_dq потрібно передати всі параметри, які вказані в її прототипі. У випадку, коли не вказана розрядність та кількість елементів у модулі пам'яті, система MAX+plus II автоматично встановлює їх. Ці дані беруться з include-файлу потрібної функції. У випадку, коли файл з описом на Verilog є головним файлом проекту, компілятор використовує дані, які вказані в глобальних параметрах проекту (Global Project Parameters).

Розглянемо ще один приклад. Опишемо модуль пам'яті типу FIFO. Опис на мові Verilog приводитися нижче.

```
module fifo_buf (data, wrreq, rdreq, clock, aclr, q, full, empty);
input [7:0] data;          // вхідна 8-розрядна шина даних
input wrreq;               // сигнал запису
input rdreq;               // сигнал зчитування
input clock;               // тактовий сигнал
input aclr;                // сигнал обнуління
output [7:0] q;            // вихідна шина
output full;               // сигнал "буфер повний"
output empty;              // сигнал "буфер пустий"
wire  sub_wire0;           // додаткові ланцюги для роботи модуля
wire [7:0] sub_wire1;
wire  sub_wire2;
wire  empty = sub_wire0;
wire [7:0] q = sub_wire1[7:0];
wire  full = sub_wire2;
// використовуємо модуль з бібліотеки параметризованих модулів
      SCFIFO      SCFIFO_component (.rdreq (rdreq), .aclr (aclr), .clock
(clock), .wrreq (wrreq), .data (data), .empty (sub_wire0), .q (sub_wire1),
.full (sub_wire2));
      defparam
      SCFIFO_component.LPM_WIDTH = 8,      // розрядність слів
      SCFIFO_component.LPM_NUMWORDS = 256, // кількість слів
      SCFIFO_component.LPM_SHOWAHEAD = "OFF",
      SCFIFO_component.USE_EAB = "ON";
endmodule                          // кінець опису модуля
```

Функціональна схема пам'яті типу FIFO містить в собі два лічильника – лічильник запису та лічильник зчитування, вихідні коди яких мультиплексуються за допомогою 2-канального мультиплексора. Запис даних відбувається по сигналу запису wrreq, зчитування – по сигналу зчитування rdreq. Дані повинні виставлятися раніше появи сигналу запису, а закінчуватися після зняття сигналу на запис. Запис починається з нульової адреси пам'яті і відбувається послідовно по зростаючих адресах. Так само зчитування починається з нульової адреси пам'яті і відбувається по послідовно зростаючих адресах. Операції запису і зчитування можуть чергуватися. Часова діаграма, яка демонструє роботу модуля, приводитися на рис.2.57.

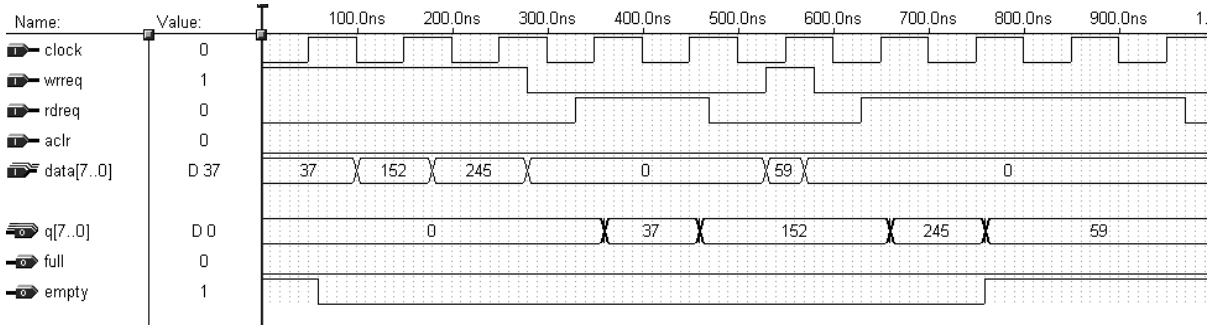

Рис.2.57. Діаграма роботи модуля FIFO

Пам'ять FIFO видає дані в тій самій послідовності, у якій вони були записані. Пам'ять FIFO можна порівняти з регістром зсуву, на виході якого дані з'являються в тій самій послідовності, у якій вони були записані в нього. Функціональна схема пам'яті типу LIFO простіша за структурою, ніж схема пам'яті FIFO, оскільки вона містить лише один лічильник і не потребує мультиплексування. Для обох типів пам'яті функціональні схеми досить складні, і в залежності від розрядності шини даних можуть суттєво змінюватися.

Важливе використання мікросхем оперативної пам'яті полягає в організації різноманітних інформаційних буферів, тобто буферної пам'яті для тимчасового зберігання даних, що передаються між двома пристроями або системами. Суть інформаційного буферу полягає в наступному: передаючий пристрій записує дані в буфер, а приймаючий пристрій зчитує дані з буферу. Зберігання масивів у пам'яті передбачає, що спочатку в пам'ять записується великий масив даних, а потім цей масив повністю зчитується. Таке проміжне зберігання дозволяє краще узгодити роботи пристроїв, які беруть участь в обміні даними, підвищують їх незалежність один від одного, узгоджують швидкості передачі та прийому даних. Це звичайний режим роботи процесора (швидкодіючого) з повільною периферією і називається режимом прямого доступу до пам'яті (ПДП), який забезпечується за допомогою контролерів і узгодженості асинхронного обміну.

Нехай, наприклад, в якості першого пристрою виступає комп'ютер, а в якості іншого – кабель локальної мережі. Комп'ютеру значно зручніше видавати дані зі швидкістю, що визначається його власною швидкодією, але в локальну мережу потрібно передавати дані з визначеною швидкістю, що задається стандартом на мережу. Крім того, комп'ютер, по можливості, не повинен відволікатися на спостереження за станом мережі. Тому буферна пам'ять в даному випадку необхідна. Так само вона потрібна при прийомі даних з локальної мережі в комп'ютер.

Головна відміна буферної пам'яті від пам'яті для тимчасового зберігання інформації полягає в тому, що до інформаційного буферу завжди мають доступ не один зовнішній пристрій, а два (і більше). Тому іноді суттєво ускладнюється як схема задання адреси мікросхеми пам'яті, так і схема розділення потоків даних.

2.10. Постійна пам'ять

Раніше були розглянуті приклади обчислення факторіала, множення двох чисел. Наприклад, для обчислення факторіала числа 5 в запропонованому варіанті модуля було потрібно 5 тактів генератора. Зі збільшенням числа пропорційно збільшується час, який затрачується на визначення значення факторіала. Крім того, істотний недолік полягає в тому, що затрачується велика кількість ресурсів мікросхеми. Тому для вирішення подібних завдань і не тільки вигідно використовувати постійний запам'ятовуючий пристрій. ПЗП може бути використаний для зберігання набору чисел, що відтворюють звук або мелодію, таблиці виправлень при лінеаризації, містити значення різних функцій. При цьому розрядність як адресної шини, так і внутрішніх комірок можна легко змінити і тим самим уникнути надлишкового використання ресурсів мікросхеми, як це може бути при використанні зовнішніх мікросхем пам'яті.

Системи проектування MAX+plus II і Quartus підтримують бібліотеку LPM-функцій. Компанія Altera рекомендує в проектах використовувати для реалізації пам'яті саме цю бібліотеку.

Розглянемо приклад реалізації ПЗП, описаного на Verilog. Завданням описуваного модуля є зведення у квадрат чисел. Опис модуля має вигляд:

```
module rom_test(address, outclock, q);      // початок модуля
input [3:0]  address;                       // адресна шина
input outclock;                             // тактовий сигнал
output [7:0] q;                             // вихідна шина даних
wire [7:0] sub_wire0;                       // 8-розрядний ланцюг
wire [7:0] q = sub_wire0[7:0];
        lpm_rom    lpm_rom_component (.outclock (outclock), .address
(address),
```

```
        .q (sub_wire0)); // підключення модуля ПЗП з бібліотеки САПР
        defparam          // встановка значень параметрів
        lpm_rom_component.lpm_width = 8,
        lpm_rom_component.lpm_widthad = 4,
        lpm_rom_component.lpm_address_control = "UNREGISTERED",
        lpm_rom_component.lpm_outdata = "REGISTERED",
        lpm_rom_component.lpm_file = "D:/max2work/ram_for_sc.mif";
endmodule
```

В наведеному описі використаний модуль постійного запам'ятовуючого пристрою «lpm_rom» із синхронним зчитуванням даних. У довідковій системі можна знайти повну інформацію, необхідну для використання параметризованих модулів в своїх проектах. Для прикладу розглянемо фрагмент із довідкової системи, що описує порядок роботи з описаним модулем. Все, що необхідно знати розробнику для використання цього типу пам'яті – це її інтерфейс (прототип функції, сигнатуру), а також необхідна інформація про параметри. Нижче приводиться опис прототипу функції пам’яті.

```
FUNCTION lpm_rom (address[LPM_WIDTHAD-1..0], inclock, outclock, memenab)
WITH (LPM_WIDTH, LPM_WIDTHAD, LPM_NUMWORDS, LPM_FILE, LPM_ADDRESS_CONTROL, LPM_OUTDATA)
RETURNS (q[LPM_WIDTH-1..0]);
```

Ця функція описана мовою AHDL, однак імена портів також можуть бути використані для мови Verilog HDL. Після назви функції, в дужках, міститься перелік вхідних сигналів. Після слова RETURNS зазначена назва та розрядність вихідного порту. Розрядності вхідних і вихідних портів вказані у вигляді параметрів. Назви параметрів, яким обов'язково необхідно присвоїти певне значення при роботі з модулем, перераховані в дужках після слова WITH. В табл.2.7 приводиться опис кожного з параметрів.

Таблиця 2.7. Опис параметрів модуля lpm_rom

Назва параметра	Опис
LPM_WIDTH	Розрядність порту q[ ]
LPM_WIDTHAD	Розрядність шини адресу address[]. Параметр повинен дорівнювати $\log_2$(LPM_NUMWORDS).Якщо розрядність шини менше, деякі з комірок пам'яті будуть недоступні. Якщо розрядність занадто велика, то

	при звертанні до незаповнених комірок значення на шині даних буде невизначеним.
LPM_NUMWORDS	Кількість слів, збережених у пам'яті. В загальному випадку, ця величина повинна бути дорівнює $2^{LPM\_WIDTHAD}$.
LPM_FILE	Шлях до файлу ініціалізації файлу (Memory Initialization File (.mif)).
LPM_ADDRESS_CONTROL	Може приймати значення "REGISTERED", "UNREGISTERED", і "UNUSED". Показує, чи зареєстрований порт адреси.
LPM_OUTDATA	Може приймати значення "REGISTERED", "UNREGISTERED", і "UNUSED". Показує, чи зареєстрований порт q[ ].

В розглянутому прикладі розрядність адресної шини дорівнює 4 (що дозволяє адресувати 16 комірок), розрядність збережених даних дорівнює 8. Розглянемо структуру файлу ініціалізації пам'яті. В коментарях приводиться опис кожного рядка файлу.

```
DEPTH=16;                    // кількість збережених слів
WIDTH=8;                     // розрядність збережених слів
ADDRESS_RADIX=DEC; // адреса комірки записується в десятковій
                             // системі счислення
DATA_RADIX=DEC;// дані записані в десятковій системі счислення.
CONTENT                      // опис вмісту пам'яті
BEGIN                        // початок
        0:0; 1:1;      // записані  «адреса комірки»:«вміст комірки»;
        2:4; 3:9; 4:16; 5:25; 6:36;
        7:49; 8:64; 9:81; 10:100;
        11:121; 12:144; 13:169;
        14:196; 15:225;
END                          // кінець опису
```

В даному файлі записана таблиця квадратів чотирьохрозрядних чисел. Після компіляції проекту і перевірки в Сигнальному редакторі, була отримана часова діаграма, представлена на рис.2.58.

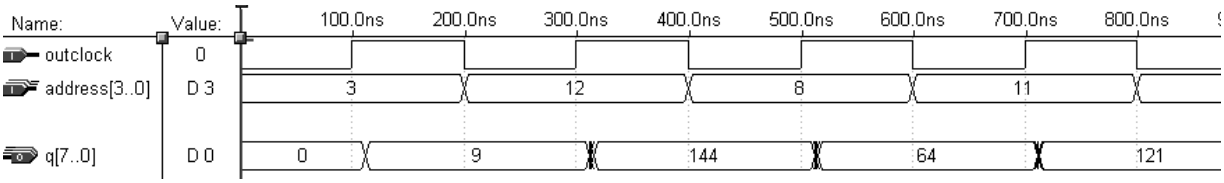

Рис.2.58. Перевірка роботи модуля ПЗП

Незважаючи на всю простоту та ефективність використання ПЗП для зберігання значень функцій, подібний підхід може бути застосований не у всіх можливих випадках. Головним чином це обумовлено вибором елементної бази. Наведений приклад може бути реалізований на мікросхемах, які підтримують пам'ять. До них відносяться FLEX10K, ACEX1K. Для мікросхем інших сімейств (FLEX8000, MAX7000) використати такі модулі не представляється можливим.

У задачах цифрової обробки сигналів часто використовуються перетворення Фур'є, перетворення Гільберта. Ці алгоритми вимагають виконання операції визначення квадратного кореню:

$$m = \sqrt{a^2 + b^2} \ .$$

Для виконання цієї операції необхідно використати алгоритм Координатної Цифрової Машини Обертання (CORDIC). Цей алгоритм обчислює тригонометричні функції, амплітуду і фазу використовуючи ітеративний процес. У пристроях Stratix™ виконання подібної операції можливо без використання реалізації алгоритму CORDIC, за допомогою операцій множення, додавання і зсуву над двійковими числами. Реалізувати подібну операцію можна з використанням елемента ПЗП. Опис модуля, що виконує зазначену операцію, має вигляд:

```
module magnitude(dataA, dataB, clk, q);     // початок опису модуля
input [2:0]  dataA;                          // вхідні шини даних
input [2:0]  dataB;
input clk;                                   // вхід тактового сигналу
output  [6:0] q;
lpm_rom     lpm_rom_component          (.outclock(clk),.address({dataA,
dataB}),.q (q));
defparam                                     // встановлення параметрів
lpm_rom_component.lpm_width = 7,
lpm_rom_component.lpm_widthad = 6,
lpm_rom_component.lpm_address_control = "UNREGISTERED",
lpm_rom_component.lpm_outdata = "REGISTERED",
lpm_rom_component.lpm_file = "D:/altera/quartus42/magnitude.mif";
endmodule
```

В якості елементу ПЗП використовується параметризований модуль lpm_rom. Максимальні значення вхідних даних рівні 7. Безумовно, практичне застосування цього модуля обмежене у зв'язку із значними похибками вихідного значення. Адресна 6-бітна шина елемента ПЗП умовно розділена на дві половини, кожна з яких є входом даних (data і data). Подібний поділ дозволив розглядати елемент пам'яті як двовимірну матрицю. Доступ до необхідної комірки пам'яті виконується шляхом об'єднання вхідних даних за допомогою фігурних дужок {} і передачі їх як адреси в елемент пам'яті. Проект створений в САПР Quartus. Карта прошивки ПЗП представлена на рис.2.59.

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	0	1	2	3	4	5	6	7
8	1	1	2	3	4	5	6	7
16	2	2	3	3	4	5	6	7
24	3	3	3	4	6	6	7	7
32	4	4	4	5	6	6	7	8
40	5	5	5	6	6	7	8	9
48	6	6	6	6	7	8	8	9
56	7	7	7	7	8	8	9	10

Рис.2.59. Карта прошивки ПЗП

Часова діаграма роботи модуля, отримана в результаті моделювання, представлена на рис.2.60.

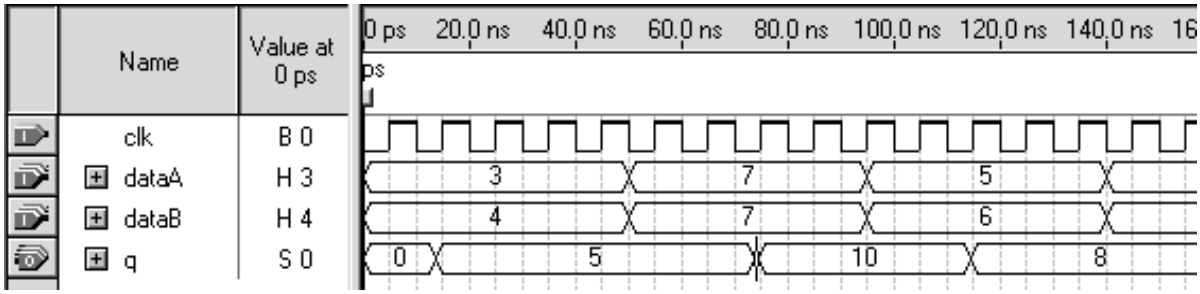

Рис.2.60. Діаграма роботи модуля обчислення квадратного кореня

САПР Quartus містить в бібліотеці модуль altsqrt, який виконує операцію визначення квадратного кореню із вхідного слова. Розглянемо приклад використання цього модулю.

```
module square(din, clk, en, clr, result, remaind); // заголовок модуля
parameter USER_WIDTH=16;        // розрядність вхідного слова
input [USER_WIDTH -1:0] din;    // вхідний порт
```

```
        input clk, en, clr;                    // входи керування
        output [USER_WIDTH -1:0] result;  // вихідний порт результату
        output [USER_WIDTH -1:0] remaind;      // вихідний порт залишку
        wire [USER_WIDTH -1:0] rswire;         // додаткові ланцюги
        wire [USER_WIDTH -1:0] rmwire;
        wire  [USER_WIDTH -1:0] result=rswire;
                                               // з'єднання з вихідними портами
        wire  [USER_WIDTH -1:0] remaind=rmwire;
        altsqrt
sqrt(.radical(din),.clk(clk),.ena(en),.aclr(clr),.q(rswire),.remainder(rmwire));
        defparam                       // встановлення значень параметрів
                msqrt.WIDTH= USER_WIDTH,
                msqrt.PIPELINE=0;
        endmodule
```

Перед описом інтерфейсу модуля (вхідних та вихідних портів) вказується параметр USER_WIDTH. Поява параметру цілком закономірна, адже модуль має кілька портів. За допомогою цього параметру легко змінити розрядність даних, з якими працюватиме модуль. Після моделювання була отримана часова діаграма, яка приводиться на рис.2.61.

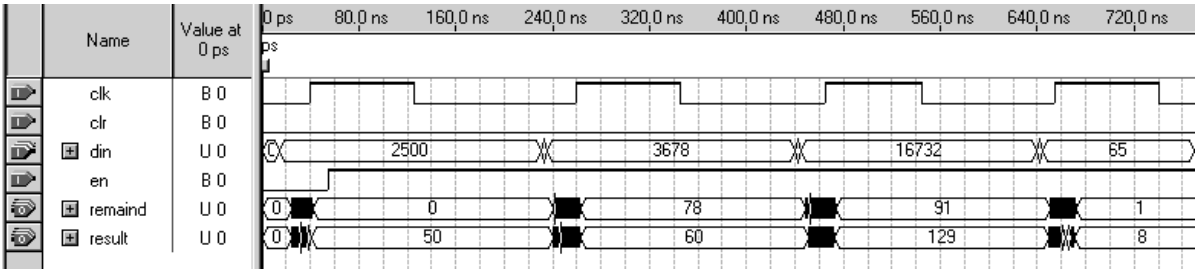

Рис.2.61. Діаграма роботи модуля altsqrt

Прототип параметризованого модуля обчислення квадратного кореня має вигляд:

```
FUNCTION altsqrt (radical[WIDTH - 1..0], clk, ena, aclr )
WITH (WIDTH, PIPELINE )
RETURNS (q[Q_PORT_WIDTH-1..0], remainder[R_PORT_WIDTH-1..0]);
```

Отже, для використання модуля необхідно вказати значення двох параметрів – розрядності даних і режиму конвеєра.

2.11. Оптимізація синтезу

Основними критеріями оптимізації при синтезі є складність (вартість, площа) схеми та її швидкодія. Бажано знайти схему, яка

характеризується найменшою складністю, найбільшою швидкодією (найменшою затримкою), або досягається певний компроміс між цими двома параметрами. При реалізації схеми на ПЛІС критерієм оптимізації є число елементарних кон'юнкцій у системі ДНФ, тобто використання ресурсів матриці. Оптимізація має на увазі під собою мінімізацію функцій. Завдання мінімізації булевої функції є класичною і полягає в знаходженні ДНФ, мінімальної по кількості літералів.

В процесі проектування об'єкту можуть бути запропоновані різні варіанти його функціональних схем з урахуванням різних критеріїв (мінімізації використаних ресурсів, часу спрацювання, регулярності структури, споживаної потужності). Розглянемо наступний опис модуля.

```
module func(A1, A2, B1, B2);          // заголовок модуля
input A1, A2;                         // вхідні сигнали
output B1, B2;                        // вихідні сигнали
assign B1=(({A1,A2})==2'b11)?1:0; // установка сигналів на виходах
assign B2=(({A1,A2})==2'b11)?0:1;
endmodule                             // кінець опису модуля
```

На рис.2.62 представлені варіанти схем, за допомогою яких може бути реалізована описана конструкція.

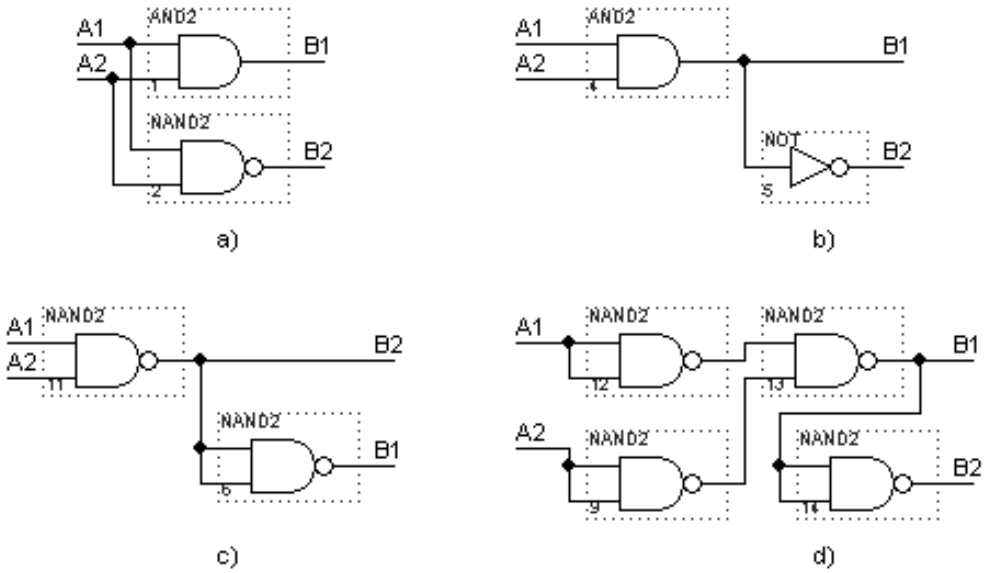

Рис.2.62. Варіанти апаратної реалізації модуля

Схема, що представлена на рис.2.62,а, при інших рівних умовах, швидша за схему, що представлена на рис.2.62,с. Опис архітектури об'єкта проекту, яка відповідає варіанту схеми на рис.2.62,а, може бути таким:

```
module funcA(A1, A2, B1, B2);      // початок опису модуля
input A1, A2;                      // вхідні сигнали
output B1, B2;                     // вихідні сигнали
      assign B1=A1 & A2;           // використання оператора I
      assign B2=!(A1 & A2);        // використання операторів НІ, І
endmodule
```

В цьому варіанті опису кожному елементу схеми співпоставлений процес, який представляє послідовність перетворення вхідної інформації і передачі її на вихід. Процес представлений у формі оператора безперервного призначення **assign**. Цей оператор спрацьовує при зміні хоча б одного з сигналів у правій частині.

Можна бачити, що у всіх приведених варіантах описи модулів мають однаковий перелік портів:

```
input A1, A2;          // вхідні порти
output B1, B2;         // вихідні порти
```

Цей фрагмент можна виділити в окремий файл (наприклад ports.v) і підключити його директивою **‘include**. Варіанту схеми, яка представлена на рис.2.62,b, відповідає наступний опис модуля.

```
module funcC(A1, A2, B1, B2);      // заголовок модуля
`include "ports.v"                 // підключення файлу з описом портів
wire W;                            // додатковий ланцюг
      assign B2=!(W);              // операція НІ
      assign W=A1 & A2;            // логічний оператор І
      assign B1=W;                 // передача сигналу на вихідний порт
endmodule                          // кінець опису модуля
```

Сигнал B2, як і B1, виробляється тільки після зміни сигналу W. Додатковий ланцюг W можна не використовувати, що демонструє наступний приклад.

```
module decoder(A1, A2, B1, B2);    // заголовок модуля
input A1, A2;                      // вхідні сигнали
output B1, B2;                     // вихідні сигнали
      assign B2=!(B1);             // операція інверсії
      assign B1=A1 & A2;           // оператор І
endmodule                          // кінець опису модуля
```

Цей опис також відповідає схемі на рис.2.62,b.

Механізм інерційної затримки дозволяє відфільтрувати вхідні сигнали, які змінюються занадто швидко. Даний механізм, власне кажучи, імітує роботу реальної схеми. Логічний сигнал представлений на схемному рівні напругою вузла. Через наявність електричних ємностей напруги вузлів не можуть змінюватися миттєво. Необхідно, щоб певна кількість енергії подавалася впродовж певного відрізку часу, – тільки в цьому випадку напруга вузла зміниться настільки, щоб викликати перемикання схеми. Тому, при моделюванні реальних логічних схем використовується інерційна затримка. На етапі алгоритмічного проектування використовується транспортна затримка.

Сполучене використання апаратних ресурсів зменшує затрати. Наступний приклад при синтезі призведе до створення двох суматорів та одного мультиплексора.

```
module sample1(a, b, c, d, sel, sum); // заголовок модуля
input a,b,c,d,sel;                    // вхідні сигнали
output sum;                           // вихідний сигнал
reg sum;                              // вихід має регістровий тип
always @(a or b or c or d or sel)
      // при зміні будь-якого з вхідних сигналів
      if(sel)                         // якщо sel=1
            sum<=a+b;                 // на виході сума значень входів a та b
      else                            // інакше
            sum<=c+d;                 // на виході сума значень c і d
endmodule                             // кінець опису модуля
```

На рис.2.63 представлено вікно ієрархії проекту, в якому видні використані модулі суматорів.

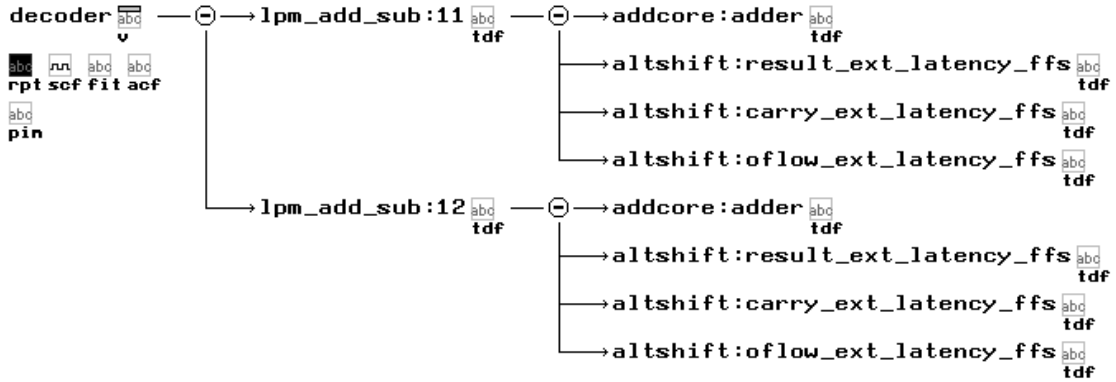

Рис.2.63. Вікно ієрархії проекту

Часові параметри схеми представлені на рис.2.64.

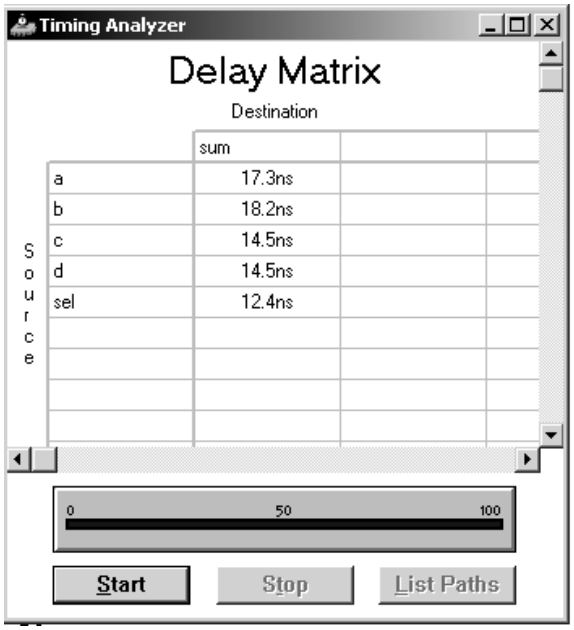

Рис.2.64. Часові затримки в суматорі

Розглянемо оптимізований варіант модуля. При синтезі буде створений один суматор та два мультиплексори.

```
module sample2(a, b, c, d, sel, sum); // заголовок модуля
input a,b,c,d,sel;                    // вхідні сигнали
output sum;                           // вихідний порт
reg tmp1, tmp2;                       // додаткові змінні
always @(a or b or c or d or sel)     // при зміні сигналів на вході
if(sel)                               // якщо sel=1
        begin
                tmp1=a;               // записуємо в додаткові змінні
                tmp2=b;          // значення сигналів на входах a та b
        end
else                                  // якщо sel=0
        begin
                tmp1=c;               // записуємо в додаткові змінні
                tmp2=d;               // значення сигналів на входах c і d
        end
        assign sum=tmp1+tmp2;         // на виході результат суми
значень
endmodule                             // кінець опису модуля
```

На рис.2.65 представлене вікно ієрархії проекту, в якому видно один суматор.

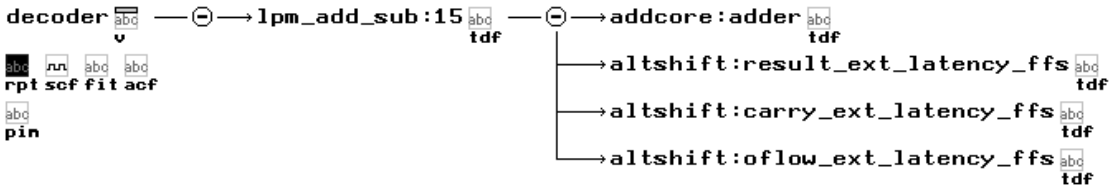

Рис.2.65. Вікно ієрархії проекту

Часові характеристики схеми представлені на рис.2.66.

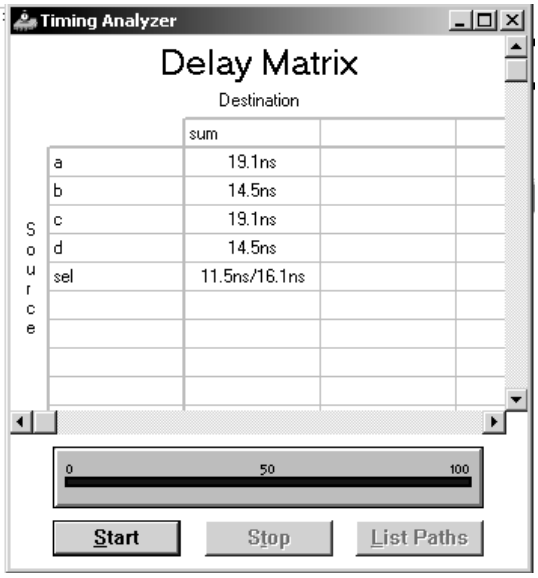

Рис.2.66. Часові затримки, що виникають в схемі

Проект був відкомпільований для мікросхеми EPF8282ALC84-4 сімейства FLEX8000. Отже, як бачимо, другий варіант реалізації модуля потребує менших затрат ресурсів мікросхеми. Що стосується часових затримок, то в першому варіанті реалізації вони трохи кращі. Виявити неефективність схеми можна після аналізу даних про кількість використаних вентилів і аналізі за допомогою графічного редактора синтезованої схеми (RTL Viewer в САПР Quartus).

При описанні лічильників найпростіший шлях – це використання арифметичних операторів додавання та віднімання. Але в деяких випадках такий підхід не завжди призводить до синтезу лічильника з найкращими часовими характеристиками. Іноді вдіється отримати більш швидкодіючий лічильник якщо використовувати послідовністну (sequencer) методологію. Наступний приклад демонструє подібний підхід.

```
module seq(count, enable, clk, reset);        // заголовок модуля
output [1:0] count;                           // вихідний порт
input enable, clk, reset;                     // входи керування
reg [1:0] count;                              // вихід має регістровий тип
reg [1:0] next_count;                         // для зберігання значення
        always @(count)                       // процедурний блок
        begin
        case (count)                  // в залежності від поточного значення
        2'h0: next_count = 2'h1;  // на виході, присвоюється наступне
        2'h1: next_count = 2'h2;  // значення змінній next_count
        2'h2: next_count = 2'h3;
        2'h3: next_count = 2'h0;
        endcase                       // кінець конструкції вибору
        end
        always @ (posedge clk or posedge reset)
        // очікування фронту сигналів
        begin
                if (reset)                            // якщо сброс
                        count = 0;                    // на виході 0
                else if (enable)      // якщо встановлений сигнал дозволу
                        count = next_count;           // на виході нове значення
        end
endmodule
```

Описаний лічильник має синхронний вхід reset, фронт сигналу на якому призводить до встановлення значення 0 на виході count лічильника. При зміні значення count виконується перший процедурний блок, де значення виходу порівнюється з набором констант. При рівності одній із констант, змінній next_count присвоюється наступне значення, яке буде встановлено на виході при появі фронту тактового сигналу clk.

Раніше було розглянуто дешифратор двійкового коду в двійково-десятковий. До недоліків наведеного опису можна віднести тривалий перехідний процес, а також значне використання ресурсів мікросхеми. Дещо покращити характеристики дешифратора вдалося шляхом невеликої зміни алгоритму його роботи: введенням тактуючого сигналу та додаткових сигналів керування. Опис модуля дешифратора має вигляд:

```
module bintobcdclk(dout0, dout1, dout2, dout3, ready, din, clk, start);
output [3:0] dout0, dout1, dout2, dout3;      // вихідні порти
input [13:0] din;                             // вхід даних
input clk, start;     // тактовий вхід та сигнал початку перетворення
output ready;                         // вихід готовності даних
reg [3:0] dout0, dout1, dout2, dout3;  // виходи мають регістровий тип
```

```
reg [29:0] shreg;                    // регістр для збереження даних
reg [3:0] temp0, temp1, temp2, temp3;  // додаткові регістри
reg ready;
reg [3:0] cnt;
always @(posedge clk)  // процедурний блок
        begin
        if (start==1)        // якщо є сигнал початку перетворення
        begin
        ready=0;             // обнуління сигналу готовності
        shreg=0;             // обнуління додаткового регістра
        shreg[13:0]=din; // запис вхідних даних даних у молодші розряди
        cnt=0;
        end
        else if (ready==0)         // якщо на виході готовності даних 0
        begin
        cnt=cnt+1;                 // збільшуємо значення лічильника
        shreg=shreg<<1;            // зсув вмісту регістра на один розряд
        {temp3,temp2,temp1,temp0}=shreg[29:14];
        if(cnt<14)                 // якщо значення лічильника менше 14
        begin
        if (temp0>=5) temp0=temp0+3;       // виконання корекції
        if (temp1>=5) temp1=temp1+3;       // даних у додаткових
        if (temp2>=5) temp2=temp2+3;       // змінних при виконанні
        if (temp3>=5) temp3=temp3+3;       // умови
        shreg[29:14]={temp3,temp2,temp1,temp0};
        end
        else
        begin
        {dout3,dout2,dout1,dout0}<=shreg[29:14];
                                   // установка значень на виходах
        ready=1;                   // установка сигналу готовності даних
        end
        end
        end
endmodule                          // кінець опису модуля
```

Операції, які виконувались в циклі **for**, в даному випадку виконуються по фронту сигналу clk. Часова діаграма, що представлена на рис.2.67, демонструє роботу дешифратора.

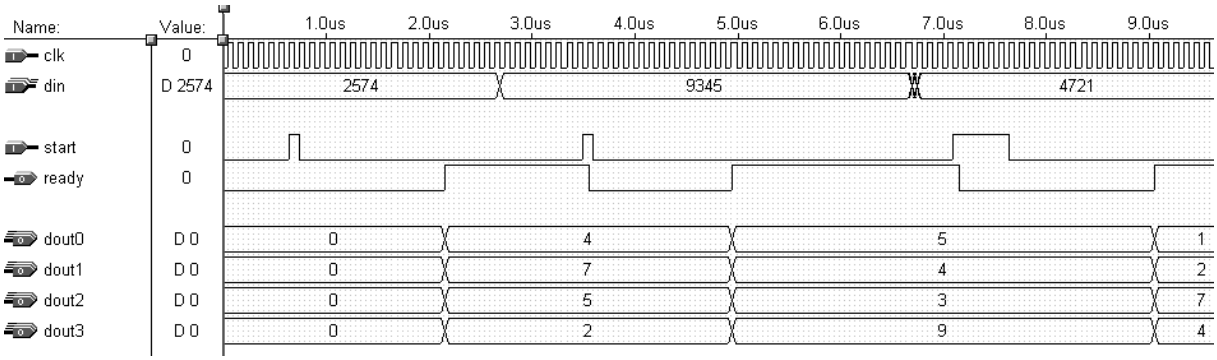

Рис.2.67. Діаграма роботи дешифратора

Початок перетворення двійкового коду в двійково-десятковий починається з появою високого рівня на вході start. Після 14 тактів зовнішнього генератора (для виконання перетворення потрібна кількість операцій зсуву, що дорівнює розрядності вхідного слова) на відповідних виходах встановлюються значення одиниць, десятків, сотень і тисяч. Значення кодів на виходах з'являються одночасно. Разом з цим, сигнал на лінії ready переходить у високий рівень. Кількість використаних логічних комірок для цього варіанту дешифратора вдалося зменшити з 69% до 44%. Щодо недоліків подібного принципу побудови можна віднести необхідність у додатковому генераторі. Однак, як було розглянуто в першому розділі, синтезований генератор можна побудувати на основі інвертора та модуля LCELL, тому цей недолік немає принципового значення.

2.12. Задачі до розділу

1. Привести структурний опис схеми, що представлена на рис.2.68.

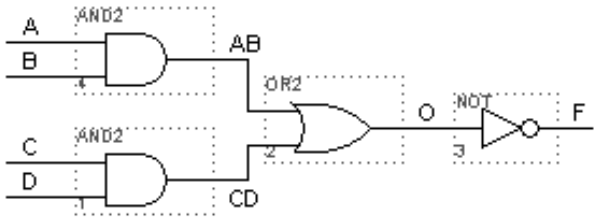

Рис.2.68. Схема реалізації логічної функції

2. Розробити модуль, який виконує функції схеми, що представлена на рис.2.68, використовуючи логічні оператори.

3. Привести структурний та поведінковий опис схеми, що представлена на рис.2.69.

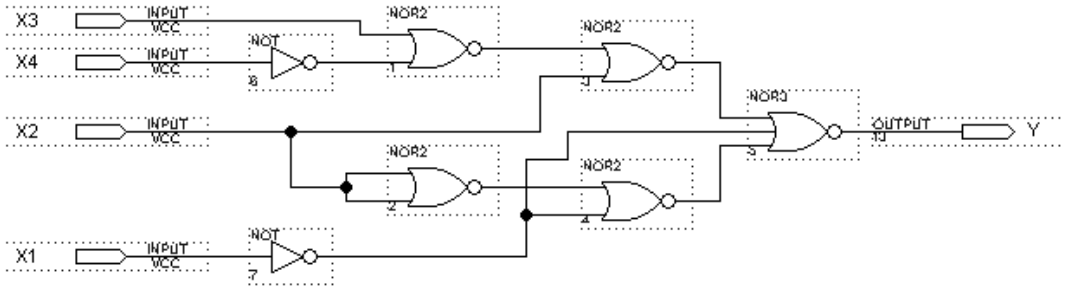

Рис.2.69. Схема реалізації логічної функції

4. Використовуючи примітиви Verilog (and, nand, or) розробити модуль, який реалізує логічну функцію:

$$y = x_1 \cdot \overline{x_2} + (\overline{x_2} + \overline{x_3}) \cdot \overline{x_1}.$$

5. Розробити модуль, який реалізує логічну функцію:

$$y = x_1 \cdot (\overline{x_1} + \overline{x_2}) \cdot (x_2 + \overline{x_3} \cdot x_4).$$

6. Привести Verilog-опис примітиву користувача, який реалізує логічну функцію відповідно до таблиці станів:

Таблиця 2.8.

N	X_2	x_1	x_0	y
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

7. Привести опис модуля, який реалізує логічну функцію у відповідності до рівняння:

$$y = \overline{\overline{x_0 \cdot x_1 \cdot x_2} + \overline{x_3 \cdot x_4 \cdot x_5} + \overline{x_6 \cdot x_7 \cdot x_8}}.$$

8. Розробити модуль мультиплексора, який має 5 інформаційних входів. При появі кодів 5, 6, 7 на адресних входах, вихід мультиплексора повинен переходити в стан високого імпедансу. Також модуль мультиплексора повинен мати вхід дозволу.

9. Функцію п’яти змінних:

$$y = x_0 \cdot x_1 \cdot \bar{x}_3 \cdot \bar{x}_4 + \bar{x}_0 \cdot \bar{x}_3 \cdot x_4 + x_0 \cdot \bar{x}_2 \cdot x_3 + x_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_2$$

реалізувати з використанням мультиплексора “з 4-х в 1”.

10. Привести структурний опис схеми (рис.2.70), яка реалізує логічну функцію, з використанням мультиплексора з бібліотеки САПР.

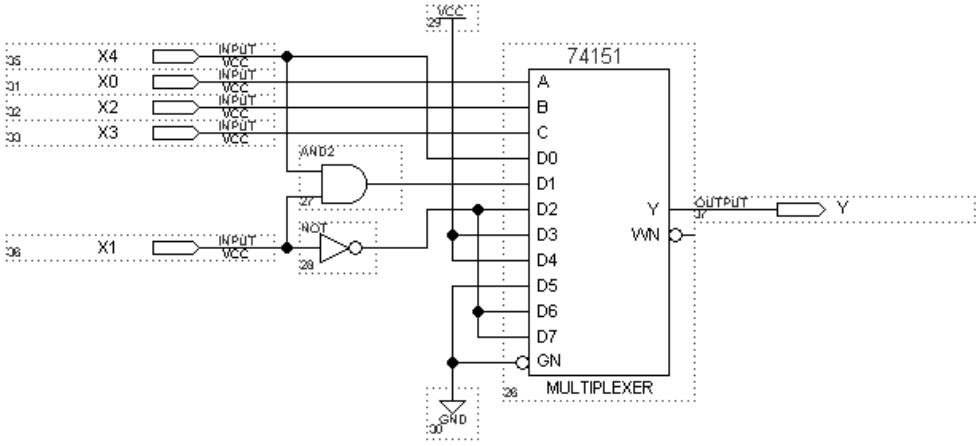

Рис.2.70. Реалізація логічної функції на базі мультиплексора

11. Використовуючи мультиплексор 8:1 з бібліотеки MAX+plus II, реалізувати логічну функцію: y=V 0,2,3,5,7.

12. Розробити модуль, на вхід якого подаються дані по лініях d_3 d_2 d_1 d_0, котрі необхідно передавати на вихідні лінії y_3 y_2 y_1 y_0 зі зсувом від 0 до 3 розрядів у залежності від значення адресного двійкового коду.

13. Реалізувати пристрій з попередньої задачі за допомогою мультиплексорів, використовуючи структурний опис модуля.

14. З двох каналів даних з частотою 1 кГц зчитуються однобайтові слова, задані в паралельному форматі. Розробити модуль, за допомогою якого інформація перетворювалася б у послідовний формат з паузою між словами в 2 біти. Визначити частоту зміни адресних сигналів для рівномірної передачі інформації у послідовному форматі.

15. Розробити модуль перетворювача прямого двійкового коду в доповнюючий.

16. Використовуючи дешифратор 74154, розробити модуль дешифратора для перетворення п’ятирозрядного коду $x_0 \div x_4$ з напругою низького рівня на одному з 32 виходів.

17. Використовуючи дешифратор “з 4 в 16” 74154 і допоміжну логіку, розробити модуль для реалізації системи логічних функцій:

$$y_1 = \bigvee_{0}^{15} 2, 4, 6, 14;$$

$$y_2 = \bigvee_{0}^{15} 0, 1, 2, 3, 5, 7, 11, 13;$$

$$y_3 = \bigvee_{0}^{15} 1, 3, 4, 5, 12, 14, 15.$$

18. Розробити схему дешифратора “з 16 на 4” для перетворення гексадецимального коду у двійковий.

19. Спроектувати дешифратор, алгоритм функціонування якого описується приведеною таблицею відповідності.

Таблиця 2.9

N	S_2	S_1	S_0	Виходи
0	0	0	0	***A***
1	0	0	1	***B***
2	0	1	0	***A***
3	0	1	1	***C***
4	1	0	0	***A***
5	1	0	1	***D***
6	1	1	0	***A***
7	1	1	1	***E***

20. Розробити модулі (шифратор та дешифратор) криптографічного захисту інформації для однобайтових даних. Модуль повинен мати вхід вибору алгоритму шифрування даних (4 варіанти). Дані завантажуються в модуль в паралельному форматі по фронту тактового сигналу, а передаються в послідовному форматі з паузою в три біти між переданими байтами.

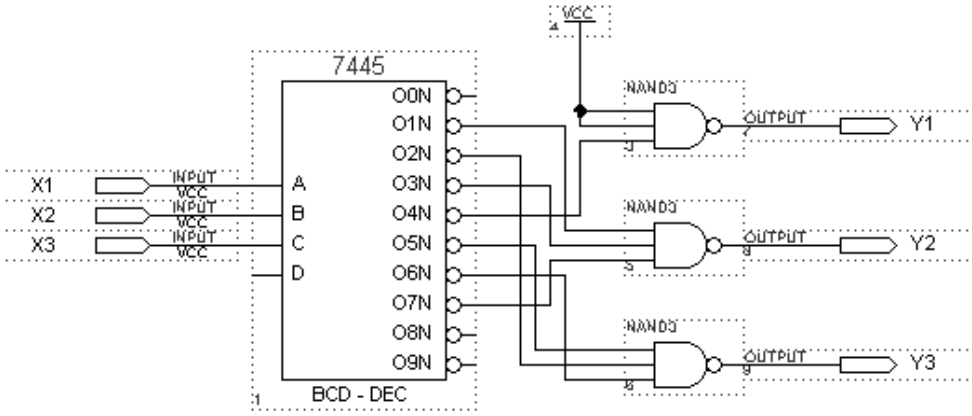

Рис.2.71. Схема реалізації логічної функції на дешифраторі

21. Привести структурний опис схеми реалізації логічних функцій, що представлена на рис.2.71, використовуючи дешифратор з бібліотеки САПР. За результатами моделювання розробити аналогічний модуль, використовуючи поведінковий стиль опису. Порівняти швидкодію обох модулів.

22. Код “2 з 5” використовується для безпомилкової передачі цифрової інформації. Розробити модуль перетворювача чотирьохрозрядного двійкового коду в код “2 з 5” та коду “2 з 5” в двійковий код (табл.3).

Таблиця 2.10.

Десят-кова цифра	4-розрядний двійковий код				Код “2 з 5”				
	x_3	x_2	x_1	x_0	a_4	a_3	a_2	a_1	a_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	1
2	0	0	1	0	0	0	1	0	1
3	0	0	1	1	0	0	1	1	0
4	0	1	0	0	0	1	0	0	1
5	0	1	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1	0	0
7	0	1	1	1	1	0	0	0	1
8	1	0	0	0	1	0	0	1	0
9	1	0	0	1	1	0	1	0	0

23. Привести поведінковий опис модуля дешифратора 5×32 без використання оператора вибору **case**. Використовуючи дешифратори 3×8 з бібліотеки САПР, розробити аналогічний модуль дешифратора. Який з

модулів дешифраторів більш ефективний з точки зору використання ресурсів та часових затримок?

24. Розробити параметризований модуль для виконання арифметичних операцій додавання та віднімання. Модуль повинен мати два виходи. На першому виході дані з'являються в двійковому коді, на другому – в двійково-десятковому. Також повинен бути вихід переповнення.

25. Розробити модуль двійкового множення на суматорах та логічних елементах І, що дозволяє множити чотирьохрозрядне число А на трьохрозрядне число В.

26. Розробити модуль множення двох однобайтових слів використовуючи:
а) оператор множення „$*$";
б) параметризований модуль з бібліотеки САПР.
Проаналізувати часові затримки в кожному з реалізованих варіантів.

27. Розробити модуль АЛП, призначений для роботи з двома чотирьохрозрядними двійковими словами (А та Е). Конкретний вид операції задається п'ятирозрядним керуючим кодом, що подається на вхідний чотирьохрозрядний порт М. АЛП повинен працювати як з сигналами позитивної логіки, так і з сигналами негативної логіки у відповідності до таблиці станів:

Таблиця 2.11

Входи вибору функції				Вхід-вихід (негативна логіка)		Вхід-вихід (позитивна логіка)	
M_3	M_2	M_1	M_0	Логічні функції M_4 = H	Арифметичні дії M_4 = L; P_i = L	Логічні функції M_4 = H	Арифметичні дії M_4 = L P_i = H
L	*L*	*L*	*L*	$\overline{A}$	$A-1$	$\overline{A}$	A
L	*L*	*L*	*H*	$\overline{A}\vee\overline{E}$	$AE-1$	$\overline{A}\overline{E}$	$A\vee E$
L	*L*	*H*	*L*	$\overline{A}\vee E$	$A\overline{E}-1$	$\overline{A}E$	$A\vee\overline{E}$
L	*H*	*L*	*L*	$\overline{A}\overline{E}$	$A+(A\vee\overline{E})$	$\overline{A}E$	$A+A\overline{E}$
L	*H*	*L*	*H*	$\overline{E}$	$A+(A\vee\overline{E})$	$\overline{E}$	$(A\vee E)+A\overline{E}$
H	*H*	*L*	*H*	$A\overline{E}$	$AE+A$	$A\vee\overline{E}$	$(A\vee E)+A$
H	*H*	*H*	*L*	AE	$A\overline{E}+A$	$A\vee\overline{E}$	$(A\vee\overline{E})+A$

Передбачити можливість роботи модуля в режимі компаратора.

28. Використовуючи багатозначний алфавіт мови Verilog (0, 1, x), розробити модуль компаратора, який виділяє з потоку цифрової інформації шини A слова, які задовольняють контрольному коду (наприклад 8'b10xx0x11) і передає їх на шину B.

29. Розробити модуль тригера, який працює у відповідності до діаграми, що представлена на рис.2.72.

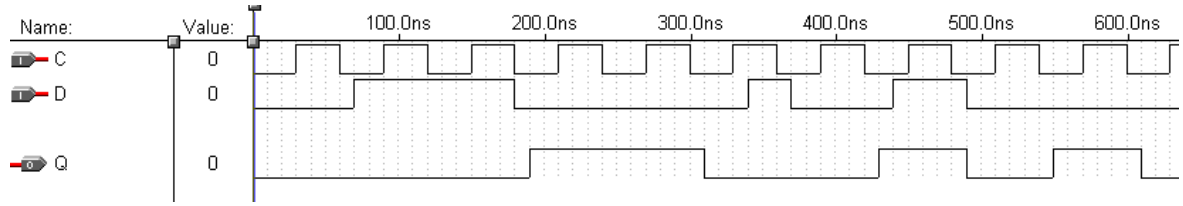

Рис.2.72. Часова діаграма роботи тригера

30. Розробити модуль тригера, схема якого приводиться на рис.2.73.

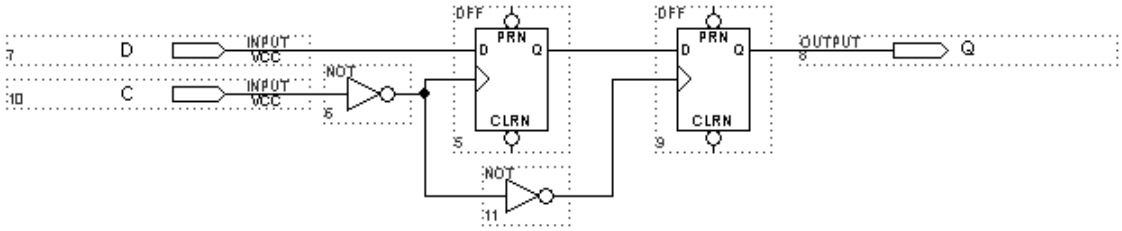

Рис.2.73. Схема тригера

31. Розробити модуль тактованого фронтом двовходового тригера, що функціонує відповідно до табл.2.12.

Табл.2.12

x_1	x_0	Q_{n+1}
0	0	0
0	1	Q_n
1	0	$\overline{Q_n}$
1	1	1

32. Привести опис примітиву (**primitive**) тригера, що функціонує відповідно до табл.4.

33. Розробити синхронний двовходовий тригер, що функціонує відповідно до табл.2.13, на базі універсального JK-тригера.

Табл.2.13.

x_1	x_0	Qn+1
0	0	0
0	1	Q_n
1	0	$\overline{Q_n}$
1	1	0

34. На рис.2.74 приведена схема тригера. Привести структурний опис схеми і виконати моделювання роботи тригера. Розробити модуль тригера, який працює по заданому алгоритму, використовуючи поведінковий стиль опису проекту.

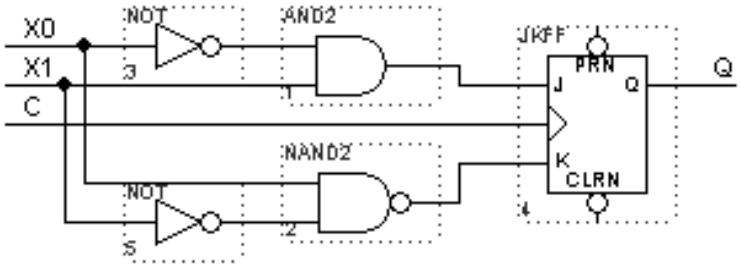

Рис.2.74. Схема тригера

35. Використовуючи D-тригер і допоміжну комбінаційну логіку, розробити модуль Т-тригера з допоміжним дозволяючим входом. Використовуючи Т-тригер і допоміжну комбінаційну логіку, розробити модуль D-тригера з допоміжним дозволяючим входом.

36. Спроектувати тактований синхронний автомат на основі таблиці переходів (табл.2.14).

Табл. 2.14.

Q \ X	x		y
	0	1	
Q_0	Q_1	Q_3	0
Q_1	Q_2	Q_1	0
Q_2	Q_1	Q_0	1
Q_3	Q_1	Q_2	0

37. Привести опис модуля синхронного автомата, схема якого представлена на рис.2.75. Отримати часові діаграми роботи автомата.

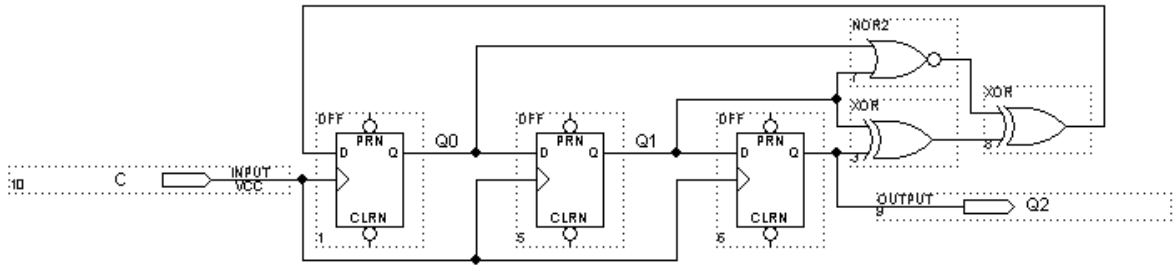

Рис.2.75. Схема синхронного автомата

38. Розробити модуль скінченного автомата, що описується таблицею переходів (табл.2.15).

Табл.2.15.

Q \ X	$x_1\,x_0$				Y
	00	01	11	10	
Q_0	Q_1	Q_1	Q_2	Q_2	0
Q_1	Q_3	Q_3	Q_2	Q_2	0
Q_2	Q_1	Q_1	Q_4	Q_4	0
Q_3	Q_3	Q_3	Q_4	Q_2	1
Q_4	Q_1	Q_3	Q_4	Q_4	1

39. Розробити модуль кодового замка (асинхронного автомата), який спрацює після правильного послідовного натискання чотирьох цифр. На вхід автомату подається код натиснутої кнопки.

5. Спроектувати синхронний тактований автомат Мура, який виконував би перевірку лінії даних, що передаються, на парність. Автомат повинен мати два входи – вхід синхронізації *C*– і вхід даних *D*–, а також вхід тактового генератора, і один вихід *y*, високий рівень якого встановлює дев’ятий біт в 1, якщо кількість одиниць в однобайтовому слові непарна.

40. Привести структурний опис синхронного автомату схема якого представлена на рис.2.76.

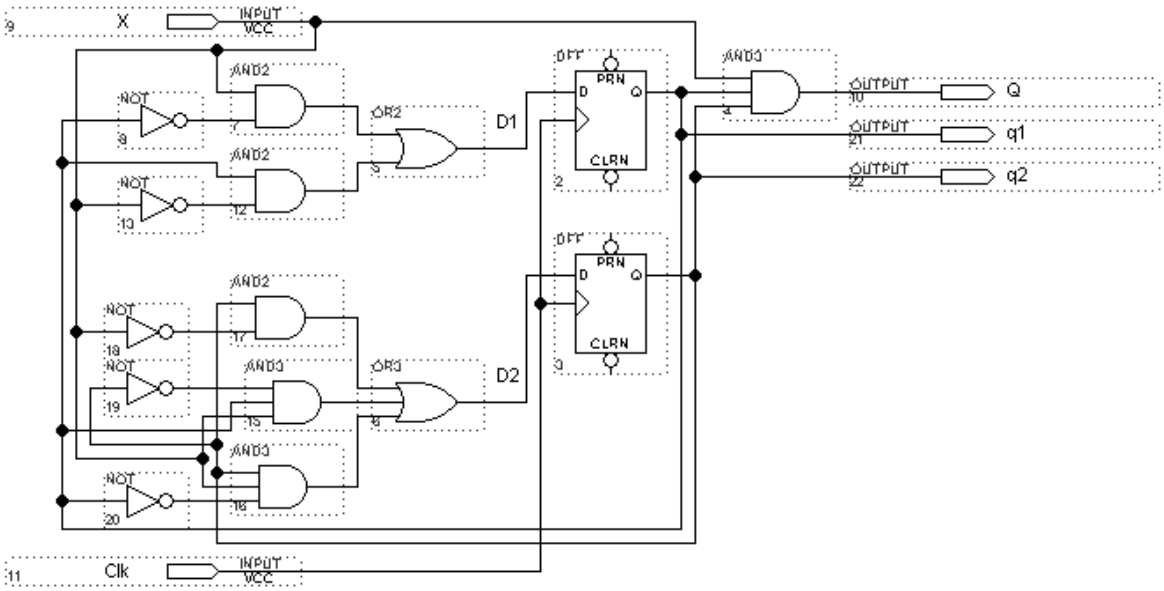

Рис.2.76. Схема синхронного автомату

Промоделювати роботу автомату та за отриманими часовими діаграмами розробити модуль автомату, використовуючи поведінковий опис. Чи впливає на результати синтезу стиль опису проекту (структурний, поведінковий)?

41. Розробити модуль скінченного автомату, граф-схема якого представлена на рис.2.77.

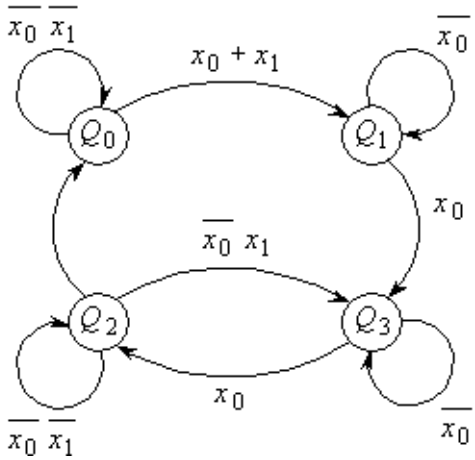

Рис.2.77. Діаграма станів автомата

42. Розробити параметризований модуль двійкового лічильника з синхронним завантаженням даних.

43. Розробити модуль 8-розрядного двійкового лічильника з синхронним обнуліннням та завантаженням даних, з входом дозволу рахування.

44. Розробити параметризований модуль двійкового реверсивного лічильника з входом дозволу та синхронним завантаженням даних.

45. Привести опис параметризованого модуля двійкового реверсивного лічильника, який би при подачі синхросигналів міг би автоматично забезпечувати реверс при заповненні (обнулінні).

46. Розробити модуль лічильника з модулем перерахунку 147.

47. Розробити циклічний генератор послідовності, що задана у табл.2.16.

Табл.2.16.

N	Q_2	Q_1	Q_0
1	0	0	1
2	1	0	0
3	0	1	0
4	1	0	1
5	1	1	0
6	0	1	1

48. Розробити модуль лічильника з бінарним шестирозрядним керуючим входом і одним виходом. Лічильник повинен видавати в циклі 64 вхідних синхроімпульсів таку кількість вихідних, яка дорівнює десятковому значенню вхідного двійкового коду.

49. Привести опис модуля 10-бітного кільцевого лічильника, в якому циркулює три одиниці та сім нулів в ряд.

50. Розробити модуль вимірювання частоти мережі з відображенням двох значущих цифр на семисегментних індикаторах. На вхід модуля подається сигнал у вигляді меандру, частота якого пропорційна частоті мережі.

51. Розробити параметризований модуль універсального регістра. Модуль регістра повинен мати вхід керування і в залежності від коду на цьому вході виконувати операції зсуву вліво або вправо, а також працювати як паралельний регістр.

52. Розробити параметризований модуль, який би автоматично і циклічно генерував сигнал, вид якого задається вхідним N-розрядним словом (наприклад 101101110110).

53. Привести опис модулів наступних регістрів:

а) асинхронний регістр пам’яті з дозволяючим входом запису і виходом на чотири розряди;

б) синхронний регістр пам’яті з одним входом дозволу запису та дозволу зчитування на чотири розряди;

в) синхронний регістр пам’яті з дозволом запису та дозволом зчитування у прямому або інверсному кодах;

г) синхронний регістр пам’яті з буферизованим виходом та Z-станом.

54. Розробити параметризований модуль регістра в якому запис та зчитування даних відбувається по одній шині.

55. Привести опис модуля регістра, який виконує бітові логічні операції над словом Q, що зберігається в ньому, та словом D, що поступає на його вхід і видає результат на вихідний порт. Тип логічної операції визначається двійковим кодом на керуючому вхідному порту. Оцінити швидкодію отриманого регістра.

56. Розробити модуль регістра який працює у відповідності до табл.2.17.

Таблиця 2.17.

OE	CLRN	CLKENN	CLK	D	Q
L	X	X	X	X	Z
H	L	X	X	X	L
H	H	H	X	X	Q_0
H	H	L	P	L	L
H	H	L	P	H	H

H – високий рівень сигналу, L – низький рівень сигналу, X – байдужий стан, P – фронт сигналу, Z – високоімпедансний стан.

60. Розробити модуль асоціативного запам’ятовуючого пристрою з загальним асоціативним накопичувачем, який має 4-бітову адресну шину комірок пам’яті (A), однобайтову шину даних (D), шину маски (MK), входи сигналів запису, зчитування та пошуку даних (WR, RD, FN), входи сигналів пошуку за адресою та даними (FA, FD), а також входи сигналів режиму запису та адресації (MO, MOA), сигнал стану (SA). Модуль виконує операції у відповідності до табл.2.18.

Таблиця 2.18.

FA	FD	FN	MOA	MO	RD	WR	Виконувана операція
1	X	X	1	X	1	0	Запис за адресою
1	X	X	1	X	0	1	Зчитування за адресою
X	X	0	X	X	1	1	Пошук за ознакою
X	0	X	0	1	0	1	Зчитування за ознакою
X	0	X	0	1	1	0	Запис за ознакою

61. Розробити модуль керування багатосегментним знакосинтезуючим індикатором (рис.2.78).

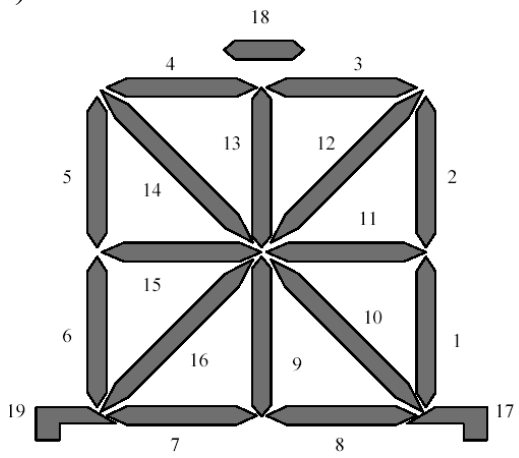

Рис.2.78. Багатосегментний індикатор

Вхідними даними модуля є ASCII-коди символів.

62. Розробити модуль оперативного запам’ятовуючого пристрою на 256 байт, в якому є можливість виконання операції сортування даних по збільшенню або зменшенню. Реалізувати модуль з використанням сортування методом „бульбашок” та вставками. Порівняти апаратні витрати на реалізацію кожного з алгоритмів. Який з алгоритмів сортування більш ефективний з точки зору швидкодії?

3. ПРИКЛАДИ ОПИСУ ПРИСТРОЇВ НА VERILOG

3.1. Процес проектування. Загальні відомості

Проектування – комплекс робіт, метою якого є одержання технічної документації, що дозволяє реалізувати і виготовити новий об'єкт із заданими властивостями і функціонуванням в заданих умовах. Стратегія проектування – функціональна декомпозиція. При декомпозиції складний пристрій розбивається на більш прості модулі, між якими повинні бути встановлені певні зв'язки. В результаті розбивки виходить деяка структура. Декомпозиція функцій блоків виконується доти, доки не будуть отримані типові функції, кожна з яких може бути реалізована на елементах обраного рівня ієрархії.

Процес проектування – багаторівневий, багатокроковий та ітераційний, з поверненням назад і переглядом прийнятих рішень. Комплекс робіт, як правило, містить у собі теоретичні та експериментальні дослідження, розрахунки і конструювання. Зміни починаються з перших кроків проектування.

Перший етап роботи – створення вихідних файлів. Створення вихідного файлу незалежно від мови програмування підтримується сучасними редакторами. В їх число входить текстовий редактор систем MAX+plus II та Quartus. Редактори істотно спрощують процедуру набору і попереднього контролю програм. Для цього редактор підтримує фарбування кольором всіх синтаксичних конструкцій мови. Безупинно перевіряється синтаксична коректність програми. При сумнівах у синтаксичних конструкціях можна викликати не тільки підказку, але і включити в текст програми шаблон потрібної конструкції, а потім його відредагувати. Компоненти майбутньої системи розробляються окремо і зберігаються у вигляді файлів проектних модулів. Потім вони поєднуються в єдиний проект. Кожен модуль може бути використаний повторно у вигляді бібліотечного елементу. Ряд бібліотечних функціональних модулів утворюють клас мегафункцій. Проектована система складається в основному із стандартних компонентів більш низького рівня інтеграції – макрофункції, мегафункції, базові логічні елементи. Бібліотека елементів САПР досить різноманітна, тому в побудові складної системи "з нуля", а точніше лише на логічних елементах, немає необхідності. В основному потреба розробника повністю задовольняється елементами стандартної бібліотеки компонентів, з яких можна створювати нові підсистеми оригінального типу.

Черговий етап роботи – компіляція програми. Програміст має можливість налаштувати компілятор на роботу з різним ступенем локальної або глобальної оптимізації, що створює програмний код, оптимізований по швидкодії та розмірам.

Один із самих відповідальних етапів – налагодження програми. Одержали поширення і використовуються різні підходи до налагодження програмного забезпечення. Неправильна робота може бути пов'язана з помилками програми і апаратури. Тому, щоб забезпечити швидкість і якість налагодження, бажано користуватися різними методами і апаратними засобами. Налагодження може відбуватися на кінцевій продукції. Бажано на попередніх етапах використовувати моделюючі системи. Позитивний ефект від цього пов'язаний з тим, що працездатність подібної системи не викликає сумнівів у відмінності від апаратури розробленої системи.

На наступному етапі робіт потрібні засоби, які допоможуть виявити програмні помилки. Програмні помилки, в свою чергу, поділяються на програмні помилки і змістовні. Змістовні помилки пов'язані із помилками програмної інтерпретації необхідних системних дій і у більшості випадків виявляються на етапі комплексного апаратно-програмного налагодження. Помилки програмування, в свою чергу, поділяються на статичні та динамічні. Синтаксичними є помилки синтаксису, невідповідність типів, неоголошені ідентифікатори. Більша частина цих помилок виявляється на етапі трансляції. Динамічні помилки проявляються на етапі роботи програми. До таких помилок можна віднести:

- вихід за межі стекової пам'яті;
- звертання до недійсного елемента масиву;
- неправильна робота з вказівником та ін.

Виявлення помилок цього типу при налагодженні ускладнене.

Проектована система піддається структурній і алгоритмічній декомпозиції, в результаті чого проект розбивається на окремі модулі. Над кожним з них може бути виконано функціональне та часове моделювання. У такий спосіб процедура повториться заново, і ми одержимо ієрархічну структуру проекту. Кожен рівень ієрархії і кожен її вузол представлений окремим файлом проекту, або Design File. У САПР ведення опису модуля або проекту в цілому можна робити різними засобами.

Апаратне налагодження, так само як налагодження програмного забезпечення, може виконуватися на всіх трьох етапах реалізації системи. Наявність у розпорядженні розробника модулів окремих фрагментів проектованої системи дозволяє легко створювати бажані конфігурації системи і моделювати її поводження. Всі види помилок проектування при моделюванні можна виявити в рідких випадках, тому тільки експерименти з реальними зразками можуть дати впевненість у досягнутих характеристиках системи. Далеко не всі проектовані системи містять таку кількість власних ресурсів, що дозволяє легко і просто організувати проведення експериментальних робіт, здійснити швидку корекцію апаратних і програмних рішень.

Результатом виконання етапу високорівневого синтезу є функціонально-структурна схема. Елементами такої схеми є типові пристрої (генеруємі модулі – суматори, перемножувачі, мультиплексори і т.д.), тригери і двовходові елементи. Високорівневий синтез досить складний – потрібно проводити складний аналіз реалізованого Verilog-опису і замінити оператори та інші конструкції мови відповідними підсхемами. Одній і тій же схемі можуть відповідати різні архітектурні тіла. Різні за формою, але однакові по суті представлення одного алгоритму можуть задавати те саме поводження. Автоматичний синтез у системах MAX+plus II і Quartus при різних представленнях того самого алгоритму мовою Verilog може давати різні логічні схеми.

3.2. Розробка асинхронного приймача і передавача

При розробці різних пристроїв на ПЛІС часто виникають завдання обміну інформацією між цими пристроями або пристроєм і комп'ютером. Задача організації обміну між двома пристроями може бути успішно вирішена з використанням послідовного синхронного інтерфейсу SPI. Можна організувати обмін між пристроями за допомогою паралельного каналу зв'язку. У цьому параграфі будуть показані приклади розробки асинхронного приймача і передавача, які виконують передачу та прийом даних в послідовному форматі. Але перш ніж приступити безпосередньо до розробки, необхідно знати алгоритм роботи цих пристроїв, формат переданого слова даних, принцип послідовної передачі.

Асинхронна передача даних широко використовується для обміну інформацією в комп'ютерах та інших пристроях. При такому способі передачі синхронізуючі події (початок передачі, закінчення передачі, помилки і т.п.), а також дані передаються по загальній лінії. Синхронізуюча інформація міститься в часових параметрах передачі, тому немає необхідності у використанні додаткових сигналів. В передавачі дані перетворяться з паралельного коду в послідовність бітів.

Принцип асинхронної послідовної передачі показаний на рис.3.1.

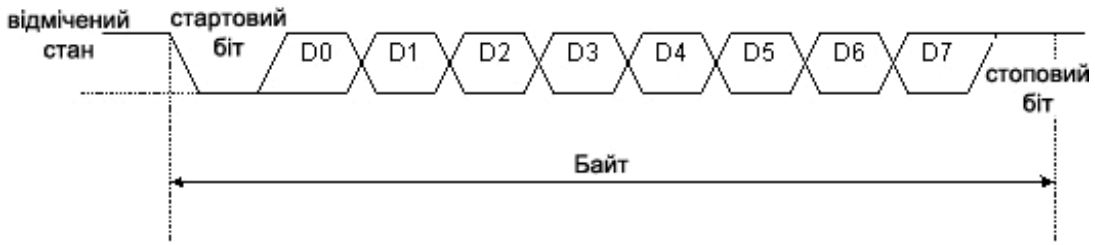

Рис.3.1. Принцип асинхронної послідовної передачі

Передача слова даних при асинхронному зв'язку починається з переходу лінії з високого рівня в низький. Це ініціює схему приймача на прийом слова даних. В стані лог.0 лінія перебуває протягом періоду

передачі одного біта. Перший біт, що передається від передавача до приймача завжди нульовий, і називається стартовим битому (Start bit). Потім послідовно передаються біти даних, починаючи з молодшого біта. Останнім передається стоповий біт (Stop bit), який має високий рівень. В загальному випадку, посилка даних може містити довільну кількість біт даних (використовуються 5, 6, 7 або 8 біт), один стартовий біт. Після бітів даних може передаватися біт парності. Завершують передачу стопові біти. Формат посилки даних обов'язково узгоджується між приймачем і передавачем. Якщо слова передаються один за іншим, то мінімальний інтервал між словами даних становить тривалість передачі стопових біт.

Функціональна схема послідовного порту представлена на рис.3.2. Працює схема в такий спосіб. Якщо в регістр передавача RgDataTx записуються дані, то вони відразу ж передаються в регістр зсуву передавача SRgTx. В момент запису даних схема ініціює початок передачі. Регістр зсуву тактується від дільника частоти Div. Частота проходження тактів нормується. Таким чином, швидкість передачі бітів чітко фіксована. Паралельний код перетворюється в послідовність бітів. Якщо по входу Rx фіксується перехід з “1” в “0”, то запускається регістр зсуву приймача SRgRx. Він також тактується опорною частотою. Коли прийом завершується, дані із регістра зсуву передаються в регістр даних приймача RgDataRx. Схеми приймача і передавача працюють незалежно, тобто можливо одночасно і передавати, і приймати дані.

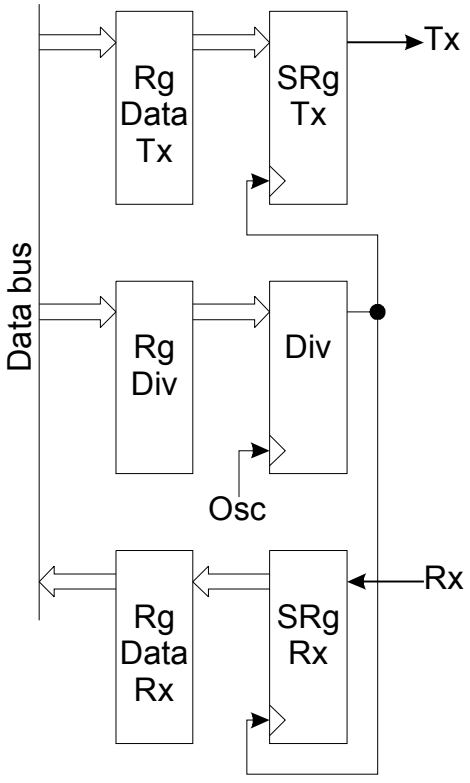

Рис.3.2. Функціональна схема послідовного порту

Швидкість послідовної передачі виміряється в бітах у секунду (б/с, b/s), або кілобитах у секунду (кб/с, kb/s). Для послідовного обміну даних прийнятий наступний ряд швидкостей: 50, 110, 150, 300, 600, 1200, 2400, 3600, 4800, 7200, 9600, 19200, 38400, 115200 б/с.

Наведені нижче приклади приймача і передавача не мають повний набір можливостей. При необхідності, можна буде змінити ці пристрої таким чином, щоб їх функції задовольняли конкретному завданню.

Для розробки асинхронного приймача виконаємо структурну декомпозицію проекту для того, щоб розділити реалізацію окремих модулів проекту. Кожен модуль може бути промодельований. Окремі модулі проекту поєднуються в єдину бібліотеку проекту. В бібліотеці перебувають символи кожного модуля, які далі можна використати або в графічному редакторі, або як окремі модулі у текстовому редакторі. В розглянутому нижче прикладі розробляється асинхронний приймач із використанням різних засобів опису (Графічний і Текстовий редактор), створений ряд файлів з описом окремих модулів, а також використані елементи із бібліотеки САПР. Проект створюється в системі MAX+plus II.

Схема розробленого асинхронного приймача в системі MAX+plus II приводиться на рис.3.3.

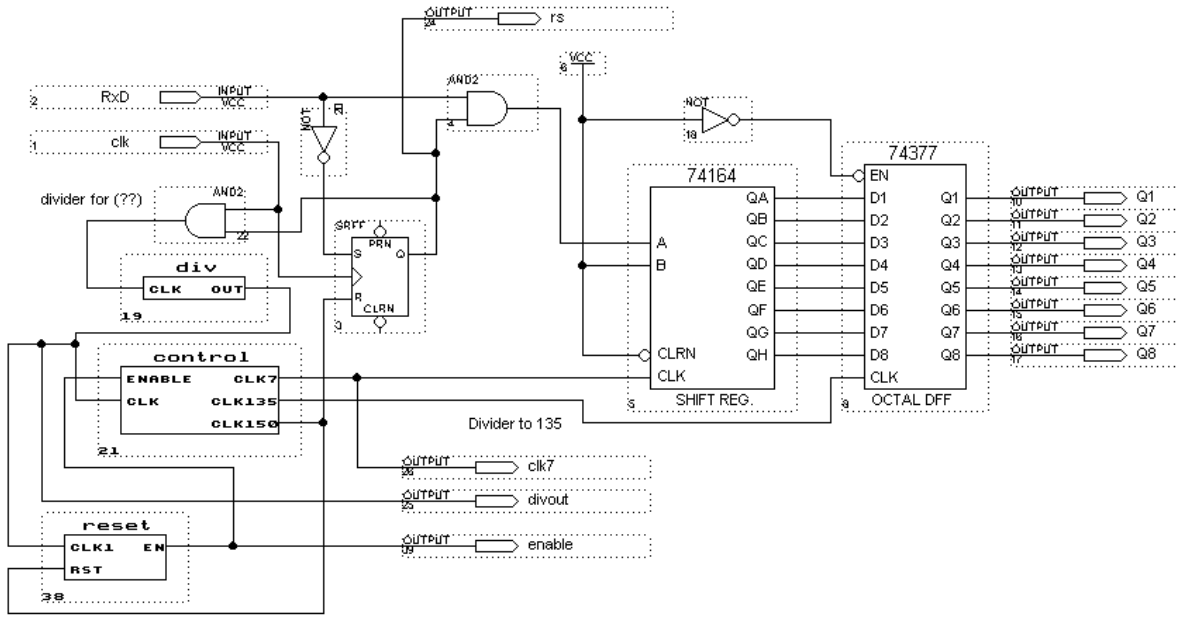

Рис.3.3. Схема асинхронного приймача

При відсутності даних, на лінії RxD встановлений сигнал високого рівня. В момент початку передачі чергового байту даних безпосередньо перед самими даними передається стартовий біт. Цей стартовий біт завжди має низький рівень. Таким чином, перехід сигналу на лінії з високого рівня в низький сигналізує про початок передачі даних. Якщо сигнал на лінії має високий рівень, на вході s тригера присутній сигнал низького рівня (це забезпечує інвертор). При переході сигналу на лінії зі стану з високим

рівнем сигналу в стан з низьким рівнем, на вході тригера з'являється лог.1. На виході тригера встановлюється сигнал високого рівня, який дозволяє роботу тактового генератора і прийом даних шляхом переведу логічних елементів 2I в режим повторювача, і на вході A регістра зсуву будуть з'являтися біти, що відповідають прийнятому слову. Елемент Control, описаний мовою Verilog, виконує наступну функцію. На 7 такті сигналу генератора clk, на виході clk7 елемента з'являється імпульс, тривалістю 1 період тактового генератора. Цей імпульс з'являється в момент часу, що відповідає середині прийнятого біта. В цей момент відбувається зсув даних в регістрі 74164. Після того, як буде прийнятий 8 біт даних, на виході clk130 елемента control з'являється одиночний імпульс, що подається на вхід запису паралельного регістра 74377. Таким чином, на виході паралельного регістра з'являється прийнятий байт даних. Після цього з'являється імпульс на виході clk150, який переводить схему в режим очікування прийому чергового байта даних. Відбувається обнуління тригера. На його виході з'являється сигнал низького рівня. Цей сигнал блокує роботу регістра зсуву, забороняє тактовий сигнал. Модуль reset забезпечує обнуління внутрішніх змінних в елементі control. Шляхом зміни коефіцієнту ділення частоти в модулі div, приймач налаштовується на різні швидкості прийому даних.

Розглянемо опис використаних модулів. Програма, що описує дільник частоти div має вигляд:

```
module div(out,clk);      // початок опису модуля
output out;               // оголошення вхідного сигналу
input clk;                // оголошення вихідного сигналу
reg out;
reg [5:0] temp;     // тимчасовий регістр для підрахунку імпульсів
генератора
always @(posedge clk)  // по фронту тактового сигналу
begin
      temp=temp+1;      // збільшуємо значення додаткового регістра
      if (temp==13)     // кожні 26 тактів генератора
            begin
            temp=0;     // обнуління значення змінної
            out=~out;   // інвертуємо значення на вихідному порту
            end
end
endmodule               // кінець опису модуля
```

Коефіцієнт ділення частоти змінюється шляхом встановлення іншого значення в дужках умовного оператора **if**. При великих значеннях дільника частоти можливо доведеться збільшити розрядність регістра temp.

Після того, як виконано опис модуля дільника частоти, з нього потрібно зробити символ. Для цього з меню File слід вибрати команду Create Default Symbol.
Розглянемо опис модуля control:

```
module control(clk7,clk135,clk150,enable,clk); // початок модуля
output clk7, clk135, clk150;    // оголошення вихідних портів
input clk, enable;              // оголошення вхідних портів
reg clk7, clk135, clk150;       // виходи мають реєстровий тип
reg [3:0] count15;              // розрядність лічильника імпульсів
дорівнює 4
reg [7:0] count150;
always @ (posedge clk)          // по фронту тактового сигналу
begin
    if (enable)                 // якщо високий рівень на вході en
        begin
        clk7<=0;    // обнуління значення внутрішніх змінних
        clk135<=0;
        clk150<=0;
        count15<=0;         // на виходах встановлено 0
        count150<=0;
        end
    else
        begin
        count15<=count15+1;    // збільшуємо лічильники на 1
        count150<=count150+1;
    if (count15==7)    // перевіряємо, якщо середина прийнятого
    біта
        begin
        clk7=1;       // на виході одиночний імпульс
        end
    if (count15==8)
        begin
        clk7<=0;     // на виході встановлено 0
        end
    if (count15==15)
        begin
        count15<=0;
         // для наступного біта починаємо  рахунок заново
        end
    if (count150==135) // якщо прийнято 8 бітів
        begin
        clk135<=1;  // передаємо прийнятий байт на вихід
```

```
            end          // сигналом запису в паралельний регістр
        if (count150==136)
            begin
            clk135<=0;
            end
        if (count150==179) // переводимо схему в режим очікування
            begin
            clk150<=1;
            end
        if (count150==180)
            begin        // обнуління значення
            clk150<=0;  // всіх змінних
            count15<=0; // на виходах встановлено низький рівень
            count150<=0;
            clk7<=0;
            clk135<=0;
            end
            end
end
endmodule               // кінець опису модуля
```

Умовний оператор **if** в залежності від заданих умов виконує визначені оператори. В описаному модулі змінна count служить для підрахунку імпульсів, що пройшли з моменту зміни даних на лінії. Саме за допомогою цієї змінної формується імпульс на середині інтервалу часу, впродовж якого на лінії втримується прийнятий біт даних. Фронт вихідного сигналу clk7 забезпечує зсув даних у регістрі зсуву 74164 на один розряд. При цьому в регістр буде записаний черговий прийнятий біт. Після того, як прийняті 9 біт даних, на виході clk135 формується короткий імпульс, по фронту якого відбувається запис прийнятого байта даних в паралельний регістр 74377. Імпульс на виході clk150 переводить вихід тригера в низький логічний рівень, тим самим очікуючи наступного стартового біта на лінії. При цьому відбувається обнуління всіх змінних і установка сигналів на вихідних портах у низький стан. З цього опису також потрібно створити графічний символ.
Розглянемо опис модуля reset:

```
module reset(en,clk1,rst);      // початок модуля
output en;                      // оголошення вихідного порту
input clk1,rst;                 // вхідні порти
reg en;                         // вихід має регістровий тип
reg flag;                       // додаткова змінна для відстеження
стану
```

```
always @ (clk1 or rst)          // при зміні вхідних сигналів
begin
      if (!rst)                 // якщо умова виконується
begin
            en=1;               // установлюємо на виходах 1
            flag=1;
end
      if (flag==1)              // якщо не було сигналу rst
            en=0;               // установлюємо на виході нуль
      if (rst==1)               // в протилежному випадку
            flag=0;             // присвоєння змінній значення 0
end
endmodule                       // кінець опису модуля
```

В принципі, схема буде працювати і без модуля reset. Основним його призначенням є видача одиночного імпульсу при включенні схеми. Цей імпульс скине лічильники в модулі control, оскільки важко передбачити їх початкові значення безпосередньо після процедури програмування або конфігурування мікросхеми.

Після того, як створені всі модулі. А також набрана схема в Графічному редакторі, можна виконувати моделювання. Часова діаграма, що демонструє роботу асинхронного приймача, представлена на рис.3.4.

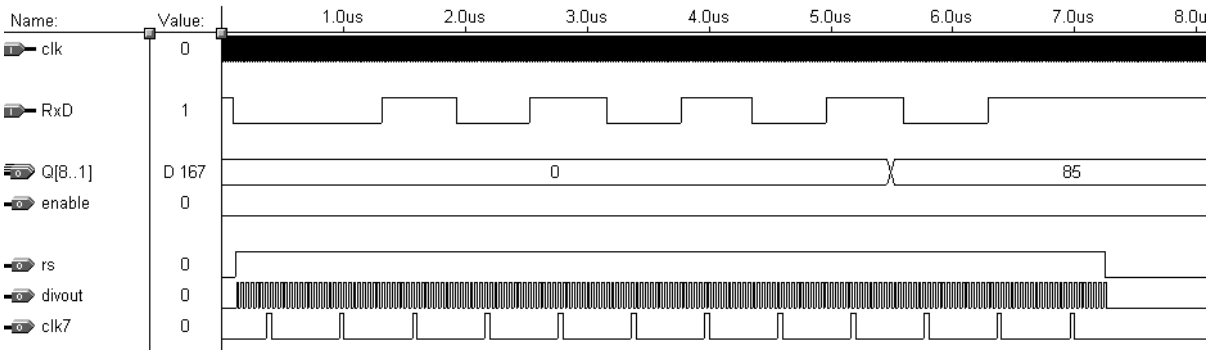

Рис.3.4. Часова діаграма роботи асинхронного приймача

Як видно з діаграми, імпульси на виході clk7 з’являються на середині прийнятого біту. Після 9 біт даних (1 стартовий плюс 8 біт даних), на виході з'являється прийнятий байт даних. Розроблена схема не перевіряє наявність стопового біту, біту парності. Читачеві пропонується самостійно доробити схему так, щоб вона мала ці можливості. В попередньому розділі приводиться приклад опису модуля перевірки на парність.

В розглянутому прикладі був розроблений асинхронний приймач. При цьому ми використали не тільки текстовий опис модулів на Verilog, але і користувалися Графічним редактором системи MAX+plus II.

Розробимо асинхронний передавач послідовного коду. Однак при цьому ми будемо користуватися винятково засобами Текстового редактора і можливостями мови Verilog. Схема передавача представлена на рис.3.5. Передавач складається з наступних модулів: дільник частоти divider, регістр даних datareg, регістр зсуву trans, RS-тригер, логічні елементи І, АБО, НІ. Установивши дані на вході data[7:0] регістра даних datareg, подаємо одиницю на вхід load. Ці дані з'являться на виході регістра. Крім того, на виході тригера з'явиться високий рівень, що дозволить роботу регістра зсуву trans і забезпечить на вході clk модуля присутність тактового сигналу. Вихід регістра зсуву через елемент АБО з'єднаний з вихідним портом. Інший вхід елемента АБО з'єднаний через інвертор з виходом тригера. Це потрібно для того, щоб при очікуванні початку передачі байта даних тримати лінію у стані лог.1.

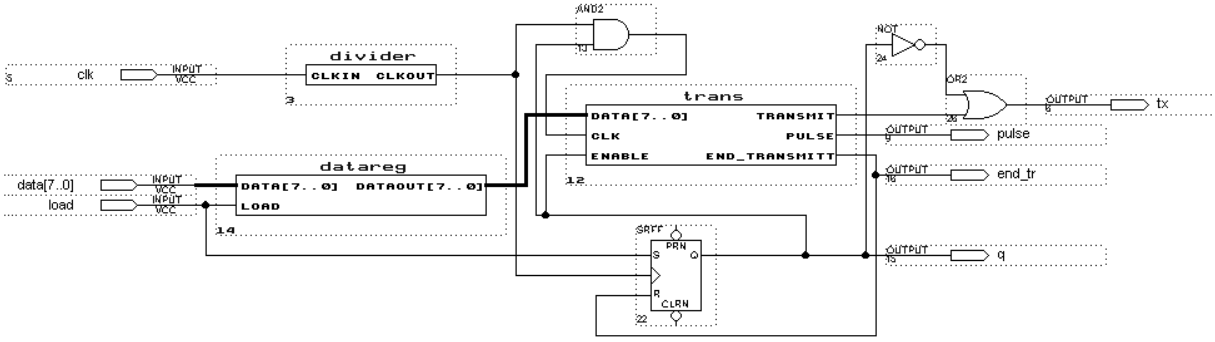

Рис.3.5. Схема асинхронного передавача

Структурний опис передавача на Verilog має вигляд:

```
module transmitter_rs(tx, end_tr, pulse, q, data, load, clock); // початок
output tx;                      // вихід
input [7:0] data;               // вхідний порт
input load;                     // сигнал запису даних
output end_tr;                  // сигнал завершення передачі
output pulse;                   // імпульси кожні пів тактового
інтервалу
output q;
reg pulse;
input clock;                    // вхід тактового сигналу
wire w2, w3, clk, clk1;         // додаткові ланцюги для організації
з'єднань
reg end_tr;
wire tr;
reg q;
reg [7:0] data_T;
        datareg dr(data_T, data, load);          // регістр даних
```

```
    divider div1(clk, clock);                  // дільник частоти
    and(clk1, clk, q);                         // логічний елемент 2І
    trans tr1(tr, pulse, end_tr, data_T, clk1, q);  // модуль trans
    or(tx, tr, ~q);                            // логічний елемент 2АБО
    trig_rs trig(q, clock, load, end_tr);      // тригер
endmodule                                      // кінець опису модуля
```

Даний модуль є головним модулем проекту. По суті, в ньому описані тільки зв'язки між іншими модулями і логічними елементами. Модуль також містить вхідні і вихідні порти, за допомогою яких буде відбуватися обмін даними. Схема на рис.3.5 показує з'єднання між всіма модулями, які входять в проект.

Розглянемо опис RS-тригеру.

```
module trig_rs(q, clock, s, r);        // початок модуля
output q;                              // оголошення виходу
input clock;                           // вхідний тактовий сигнал
input s, r;                            // входи установки і сбросу
wire w1, w2;                           // додаткові ланцюги
srff sr1(s, r, clock, w1, w2, q);      // підключаємо модуль із
бібліотеки
endmodule                              // кінець модуля
```

В описі тригера був використаний RS-тригер з бібліотеки системи MAX+plus II. Цей приклад показує, що поряд з описаними власними модулями можна використовувати також елементи з бібліотеки системи.

Опис дільника частоти приводиться нижче.

```
module divider(clkout, clkin);         // початок опису модуля
output clkout;                         // вихідний порт
input clkin;                           // вхідний порт
reg [5:0] count;                       // внутрішній лічильник
reg clkout;                            // вихід має тип регістр
always @(posedge clkin)                // по фронту на вході clkin
begin
    count=count+1;                     // збільшуємо значення лічильника
    if (count==13)                     // якщо виконується умова
    begin
        count=0;                       // обнуління лічильника
        clkout=~clkout;                // інвертуємо сигнал на виході
    end
end
endmodule                              // кінець опису модуля
```

Як бачимо, опис цього модуля нічим не відрізняється від описаного раніше дільника. Наступний модуль представляє з себе найпростіший 8-розрядний паралельний регістр. По фронту сигналу load відбувається запис даних у регістр і установка їх на вихідному порту.

```
module datareg(dataout, data, load); // заголовок модуля
output [7:0] dataout;                // оголошення вихідного порту
input [7:0] data;                    // вхідний порт даних
input load;                          // вхід завантаження даних
reg [7:0] dataout;                   // вихід визначений як регістр
always @ (posedge load)        // по фронту сигналу завантаження
      dataout=data;                  // записуємо дані в регістр
endmodule                            // кінець опису модуля
```

Основним елементом розробленого передавача є модуль trans. Саме цей модуль формує на лінії біти переданого слова в потрібні моменти часу. Розглянемо його опис мовою Verilog.

```
module trans(transmit, pulse, end_transmitt, data, clk, enable);
output transmit;              // визначення вихідних портів
output end_transmitt;         // сигнал завершення передачі
output pulse;
input [7:0] data;             // вхідний порт даних
input clk;                    // вхід тактового сигналу
input enable;                 // вхід дозволу
reg transmit;                 // вихідні порти мають регістровий
тип
reg pulse;
reg end_transmitt;
reg [7:0] temp2;              // оголошення додаткових змінних
reg [3:0] temp1;
always @ (posedge clk)        // по фронту тактового сигналу
      begin
      if (enable)   // якщо встановлено сигнал дозволу передачі
      begin
      if (temp2==0) transmit=0;
      // сигнал завершення передачі має рівень 0
      temp2=temp2+1;          // збільшуємо на 1 значення змінних
      temp1=temp1+1;
      if (temp1==7) pulse=1;
      //на середині інтервалу формуємо імпульс
      if (temp1==9) pulse=0;
      if (temp1==15) pulse=1;  // при посилці наступного біту
```

```
        //даних також формуємо імпульс для демонстрації
        //моментів переходу на часовій діаграмі
        if (temp1==0) pulse=0;
        if (temp2==16) transmit=data[0];
        // в залежності від кількості тактів
        if (temp2==32) transmit=data[1];      // на лінії встановлюється
        if (temp2==48) transmit=data[2];      // відповідний біт даних
        if (temp2==64) transmit=data[3];
        if (temp2==80) transmit=data[4];
        if (temp2==96) transmit=data[5];
        if (temp2==112) transmit=data[6];
        if (temp2==128) transmit=data[7];
        if (temp2==144) transmit=1;
        if (temp2==159)
        begin
        end_transmitt=1;   // сигналізуємо про завершення передачі
        pulse=0;
        end
        if (temp2==160)
        begin
        end_transmitt=0;// сигнал завершення передачі встановлено в 0
        temp2=0;
        end
        end
        else          // якщо сигнал дозволу передачі не встановлений
        begin
                end_transmitt=0;
                temp2=0;            // обнуління додаткових змінних
                temp1=0;
        end
        end
endmodule                           // кінець опису модуля
```

Часова діаграма, отримана при моделюванні роботи схеми, представлена на рис.3.6.

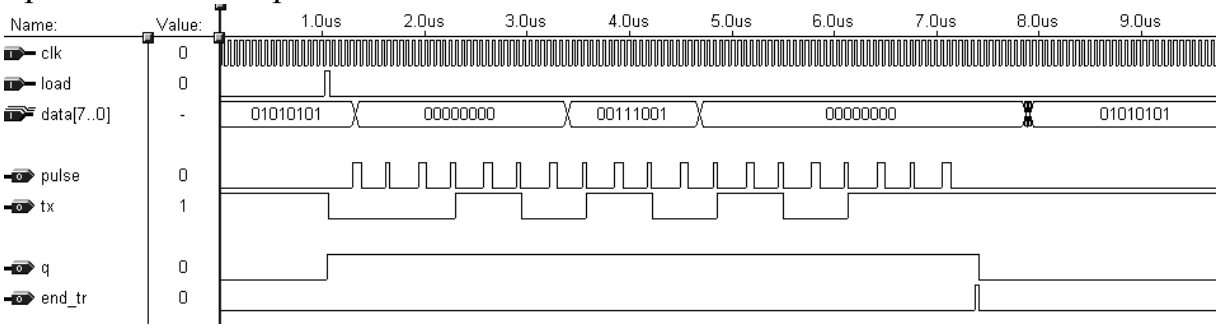

Рис.3.6. Моделювання роботи передавача

На часовій діаграмі широкий імпульс на виводі pulse відповідає середині інтервалу біта. В цей момент відбувається запис біта в регістр зсуву. Період вузького імпульсу дорівнює 16 періодам тактового генератора і демонструє момент зміни сигналу на лінії. Після завершення передачі на виході tnd_tr формується короткий імпульс.

3.3. Приклад розробки годинника

На рис.3.7 приводиться принципова схема одного з можливих варіантів годинка.

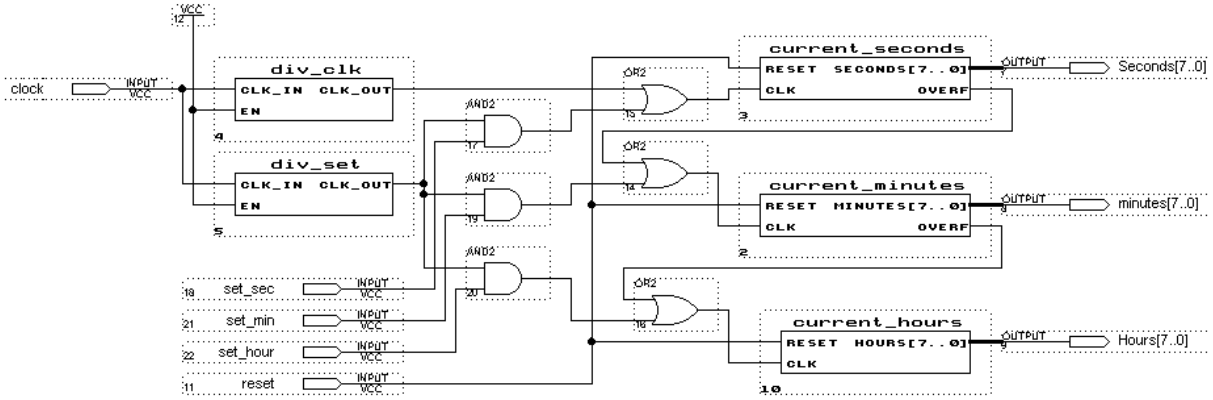

Рис.3.7. Схема годинника