

# Лабораторна робота №1

## Синхронні по фронту елементи пам'яті

<b>1.1 Практичне застосування досліджуваних схем</b>	<b>1</b>
<b>1.2 Теоретична частина</b>	<b>1</b>
1.2.1 Синхронний по фронту D-тригер	1
1.2.2 Опис схем синхронних по фронту на мові Verilog	4
1.2.11 Паралельний регістр	9
1.2.12 Регістр зсуву, оператор конкатенації мови Verilog	11
1.2.13 Регістр зсуву з входом дозволу зсуву	14
1.2.14 Регістр зсуву з лінійним зворотнім зв'язком (LFSR)	15
1.2.15 Синхронізація даних з входів FPGA	18
1.2.16 Детектор фронту	21
1.2.17 Інтерфейс SPI	23
1.2.18 SPI приймач	27
1.2.19 Програмний код SPI передавача для Nucleo F401RE	32
1.2.20 Програмний код SPI передавача для Arduino	34
<b>1.3 Практична частина</b>	<b>36</b>
1.3.1 Передача даних по SPI з використанням MBED та Nucleo F401RE	36
1.3.2 Передача даних по SPI з використанням Arduino	38
1.3.3 Використання логічного аналізатора SignalTap в Quartus Prime	39
1.3.4 Завдання на лабораторну роботу	44
<b>1.4 Контрольні запитання</b>	<b>45</b>
<b>1.5 Перелік посилань</b>	<b>46</b>

## 1.1 Практичне застосування досліджуваних схем

Синхронні по фронту D-тригери присутні у будь яких відносно складних цифрових мікросхемах. Ці компоненти зберігають проміжні значення цифрових сигналів, забезпечуючи боротьбу з паразитними імпульсами, що виникають під час гонок в комбінаційній логіці. На базі синхронних по фронту D-тригерів будують паралельні регістри та регістри зсуву. У паралельних регістрах зберігають проміжні значення обчислень (наприклад, регістри мікропроцесору). Регістри зсуву застосовують для передачі даних у послідовній формі (наприклад, інтерфейс SPI) та для побудови високочастотних лічильників в унітарному коді (one-hot counters). Також регістри зсуву застосовують у ланцюжках граничного сканування (boundary scan chain) для тестування мікросхем та відлагодження процесорів через JTAG. Регістр зсуву з лінійним зворотнім зв'язком (LFSR) використовують для генерації псевдовипадкових послідовностей чисел. LFSR застосовують у криптографії (потоківі шифри), для розрахунку циклічних контрольних сум CRC з метою виявлення помилок в прийнятих даних, для генерації тестових послідовностей під час тестування і самотестування цифрових мікросхем (built-in self-test, BIST), і в скремблерах телекомунікаційних систем.

## 1.2 Теоретична частина

### 1.2.1 Синхронний по фронту D-тригер

Синхронний по фронту D-тригер є елементом пам'яті, що зберігає 1 біт даних. Умовне графічне позначення такого тригера наведено на рис.1.1.

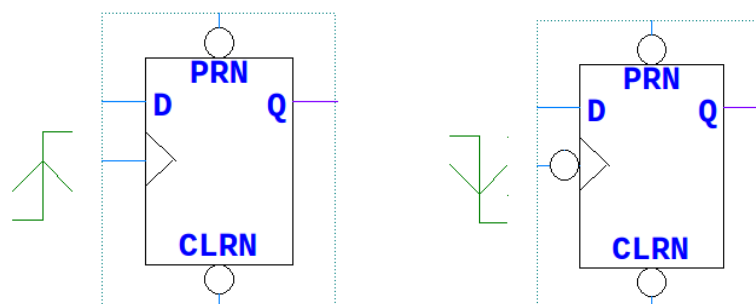


Рис.1.1 - Умовне графічне позначення D-тригерів синхронних по передньому фронту (зліва) та по задньому фронту (зправа)

Синхронний по фронту D-тригер має вхід даних D та вихід даних Q. В деяких тригерах також присутній інверсний вихід даних  $\overline{Q}$  значення якого інверсне (протилежне) до значення виходу Q (якщо  $Q=0$ , то  $\overline{Q}=1$  і навпаки). Кожен такий тригер має вхід синхронізації (його ще іноді називають тактовий вхід), який позначено трикутником для тригера синхронного по передньому фронту. Для тригера синхронного по задньому фронту зліва біля трикутника малюють круг. Часто вхід синхронізації називається CLK. Деякі синхронні по фронту тригери мають входи асинхронного скидання CLRN та асинхронного встановлення PRN.

**Логіка роботи синхронного по фронту D-тригера.** Після подачі напруги живлення D-тригер встановлюється у випадкове значення (0 або 1). Якщо подати на вхід даних D нове значення (0 або 1), а після цього на вхід синхронізації подати активний фронт (передній фронт для тригера синхронного по передньому фронту і задній фронт для тригера синхронного по задньому фронту), це нове значення з входу D запишеться в тригер і з'явиться на його виході Q.

Якщо є необхідність записати в тригер 0 в будь який момент часу, а не лише в момент активного фронту, можна скористатися асинхронним входом скидання CLRN, подавши на нього активний рівень. У більшості сучасних тригерів активний логічний рівень для операцій асинхронного встановлення і скидання -- логічний 0. Як тільки на вхід CLRN подається активний логічний рівень, тригер скидається в 0 і на виході Q з'являється 0. Аналогічно після подачі активного логічного рівня на вхід асинхронного встановлення PRN, тригер встановлюється в 1 і на його виході Q з'являється 1.

Асинхронні входи встановлення і скидання бувають корисні для приведення станів тригерів у задані значення після подачі напруги живлення. Як зазначено вище, після подачі напруги живлення значення тригерів встановлюються випадковим чином. Для приведення станів тригерів до заданих значень асинхронні входи встановлення (для тригерів які після подачі живлення повинні містити 1) та скидання (для тригерів які після подачі живлення повинні містити 0) підключаються до глобального входу RESET. Одразу після подачі напруги живлення на глобальному вході RESET формується активний рівень, що приводить до встановлення станів тригерів у задані значення.

Важливо розуміти, що запис нового значення в D-тригер, його асинхронне встановлення та зкидання відбуваються не миттєво, а з певною затримкою. Наприклад, затримка запису  $t_{cq}$  (time from clock to q) показана на рис.1.2.

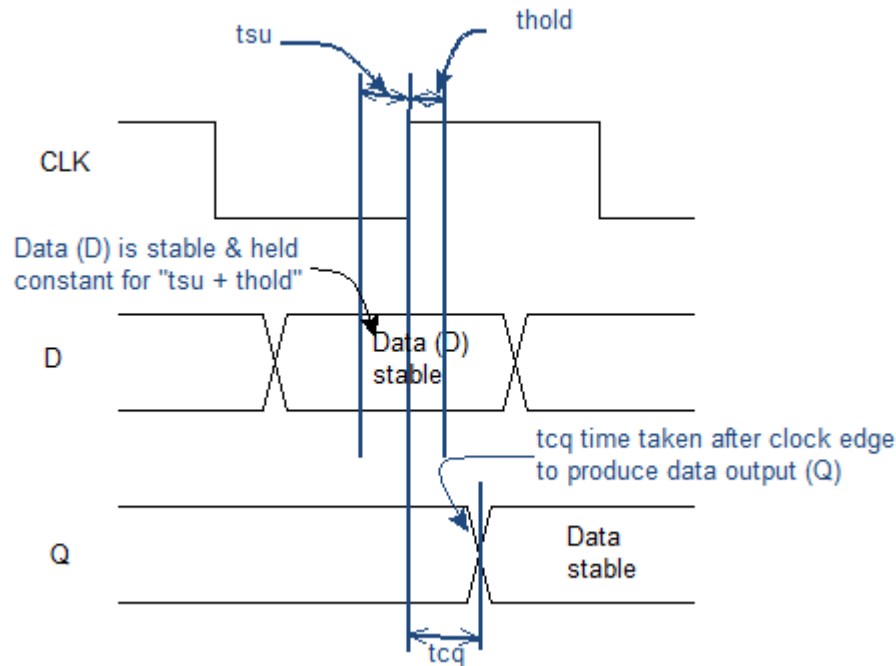


Рис.1.2 - Таймінги (затримки) D-тригера синхронного по фронту

У D-тригера синхронного по фронту є дуже важливе обмеження. Значення на вході D не повинно змінюватись одночасно з активним фронтом CLK. Якщо дані на вході D будуть змінюватись під час активного фронту CLK, тригер може перейти в метастабільний стан. Коли D-тригер знаходиться в метастабільному стані на його виході Q присутня напруга, що дорівнює половині напруги живлення ( $V_{dd}/2$ ). Це не лог. 0 і не лог. 1, а напруга із забороненого діапазону значень. Тригер знаходиться у метастабільному стані невеликий випадковий проміжок часу, а потім виходить в стан лог.0, або в стан лог.1 випадковим чином. Вхідження тригерів в метастабільний стан призводить до помилок в цифрових мікросхемах і їх нестабільної роботи.

Щоб тригер не входив у метастабільний стан, дані на вході D не повинні змінюватись протягом проміжку часу  $t_{su}$  (time setup) перед приходом активного фронту та протягом проміжку часу  $t_{hold}$  (time hold) після приходу активного фронту (див. рис.1.9). Проміжки часу  $t_{su}$  та  $t_{hold}$  називають таймінгами D-тригера. Таймінги тригерів наведені в документації на цифрову мікросхему. Ми повернемося до таймінгів

D-тригерів, коли будемо розглядати питання максимально можливої частоти сигналу синхронізації цифрових схем.

Розглянутий вище D-тригер записує дані з входу D після кожного активного фронту на вході синхронізації. Однак в сучасних цифрових системах сигнал синхронізації - це періодична послідовність імпульсів з високою частотою. В такому випадку активний фронт буде приходити на вхід синхронізації регулярно і дуже часто. В тригер постійно будуть записуватися нові значення з входу D. Що робити, якщо хочеться використовувати D-тригер як елемент пам'яті і зберігати в ньому значення не лише протягом періоду сигналу синхронізації, а довільний проміжок часу? Тут стане у нагоді синхронний по фронту D-тригер з входом дозволу запису (рис.1.3).

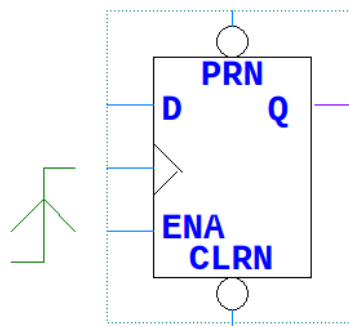


Рис.1.3 - Умовне графічне позначення синхронного по передньому фронту D-тригера з входом дозволу запису ENA

Запис в подібний тригер нового значення з входу D відбувається по активному фронту і лише, якщо на вході дозволу запису ENA присутній активний логічний рівень (зазвичай це лог.1). Якщо ж під час активного фронту на вході дозволу запису ENA присутній не активний логічний рівень (зазвичай це лог.0), нове значення в тригер не записується і тригер зберігає попереднє значення. Часто вхід ENA ще називають WE (Write Enable).

### 1.2.2 Опис схем синхронних по фронту на мові Verilog

Згадаємо, що процес створення цифрової схеми з опису на Verilog називається **синтезом**. Зазвичай синтез - шаблонізована процедура. Для різних компонентів цифрових схем існують шаблони їх опису на Verilog і необхідно дотримуватись цих

шаблонів для правильного синтезу. Коли САПР зустрічає в коді Verilog певний шаблон коду, цей шаблон замінюється на відповідну цифрову схему (наприклад, оператор + замінюється на суматор і т.д.). Звісно, це спрощений опис синтезу і в реальності є нюанси. Однак в загальному вигляді процедура синтезу приблизно так і відбувається.

В цьому розділі розглянемо шаблони коду для опису на Verilog схем синхронних по фронту. Прикладами подібних схем є D-тригери, паралельні регістри, регістри зсуву.

Шаблон коду для опису D-тригера синхронного по передньому фронту без асинхронних входів і дозволу запису наведено в лістингу 1.1.

Як видно з лістингу для опису D-тригера застосовано процедурний **always** блок.

**Лістинг 1.1** - Шаблон опису D-тригера синхронного по передньому фронту без асинхронних входів і дозволу запису

```
module flip_flop(i_clk, i_data, o_data);

    input      i_clk;
    input      i_data;
    output reg o_data;

    always @(posedge i_clk) begin
        o_data <= i_data;
    end

endmodule
```

Процедурний **always** блок - важлива конструкція мови Verilog, тому розглянемо його детальніше. По суті, **always** блок є нескінченним циклом. Його вміст ітеративно виконується раз за разом. Ось приклад такого нескінченного циклу:

**Лістинг 1.2** - Приклад нескінченного циклу на базі **always** блоку (поганий приклад)

```
reg [7:0] i;
always i = i + 1;
```

Після ключого слова **always** повинна міститися або одна інструкція, як в прикладі зверху, або блок коду обмежений ключовими словами **begin** / **end**. Між ключовими словами **begin** / **end** може міститися довільна кількість інструкцій. Інструкції блокуючого присвоювання (оператор =) в середині блоку коду обмеженого **begin** / **end** виконуються послідовно одна за одною.

Дуже важливий момент! В операціях присвоювання всередині **always** блоку, в лівій частині оператора присвоювання повинні бути лише змінні типу **reg**.

В лістингу 1.3 дві операції блокуючого присвоювання всередині блоку виконуються раз за разом у нескінченному циклі **always** блоку.

Однак подібне безперервне виконання коду всередині **always** блоку є грубою помилкою, яка призводить до зависання симулятора під час моделювання.

**Лістинг 1.3** - Приклад нескінченного ітеративного виконання блоку коду з двох операцій блокуючого присвоювання

```
input      [7:0] i_op1, i_op2;
output reg [7:0] o_sum, o_sub;

always begin
    o_sum = i_op1 + i_op2;
    o_sub = i_op1 - i_op2;
end
```

Правильна робота **always** блоку полягає у тому, щоб очікувати на певну подію (наприклад, зміну одного з аргументів, або на передній фронт сигналу синхронізації), після настання події виконати блок коду і знову перейти в режим очікування на подію.

Очікування на подію в мові Verilog забезпечується оператором **@(...)**. Оператор **@** блокує виконання наступного за ним коду до виконання події/подій, що перераховані всередині круглих скобок. Якщо є кілька подій на які необхідно очікувати, їх перераховують через кому і виникнення будь якої з цих подій приводить до продовження виконання коду, що слідує після оператора **@**. Подією може бути зміна сигналу, передній, або задній фронт сигналу. Якщо необхідно очікувати на будь яку зміну сигналу (зміну рівня), тоді в круглих скобках пишуть просто ім'я сигналу. Якщо необхідно очікувати на передній фронт сигналу, перед ім'ям сигналу пишуть ключове слово **posedge**. Якщо необхідно очікувати на задній фронт сигналу, перед ім'ям сигналу пишуть ключове слово **negedge**. Сигнали у круглих скобках оператору **@** називають **списком чутливості**.

Увага! Очікування зміни рівня і зміни фронту в одному операторі **@** приводить до одержання коду, який моделюється, але не синтезується. Зазвичай при описі комбінаційної логіки (наприклад, суматора, чи перемножувача) за допомогою **always** блоку, в операторі **@** очікують на зміну рівня будь-якого з вхідних аргументів. При

описі синхронних по фронту схем в операторі @ очікують на активний фронт сигналу синхронізації або на фронт сигналу асинхронного зкидання/встановлення, що призводить до активного рівня входів зкидання/встановлення (далі будуть наведені приклади).

Повернемося до лістингу 1.1. Як видно, в **always** блоці за допомогою оператора @ відбувається очікування переднього фронту сигналу синхронізації i\_clk і одразу після переднього фронту цього сигналу (грубо кажучи, в той же момент часу, коли було зафіксовано передній фронт), відбувається запис значення входу i\_data у змінну o\_data, що має тип **reg** і є виходом модуля. Так відбувається моделювання зазначеного блоку коду в симуляторі.

Логіка синтезу коду з лістингу 1.1 наступна. При аналізі цього коду САПР Quartus Prime бачить, що після активного фронту сигналу синхронізації відбувається запис в змінну типу **reg**. Це відповідає логіці роботи і шаблону опису синхронного по фронту D-тригера, тож з такого коду синтезується синхронний по фронту D-тригер.

Дуже важливе уточнення! При описі синхронних по фронту схем обов'язково необхідно використовувати неблокуюче присвоювання (оператор <=). Використання блокуючого присвоювання (оператор =) для опису синхронних по фронту схем призводить до помилок симуляції та іноді призводить до неправильного синтезу.

Блокуюче ж присвоювання (оператор =) використовують для опису комбінаційної логіки з використанням **always** блоку. Це питання ми розглянемо в одній з наступних лабораторних робіт.

Тепер розглянемо шаблон коду для опису D-тригера синхронного по передньому фронту з входом асинхронного зкидання по низькому рівню, без входу дозволу запису (лістинг 1.4).

В даному випадку, вхід зкидання асинхронний, а значить зкидання повинно відбуватися без прив'язки до сигналу синхронізації. Тож необхідно додати у список чутливості оператора @ також сигнал i\_arst\_n. Раз зкидання повинне відбуватися по низькому рівню i\_arst\_n, будемо очікувати на задній фронт цього сигналу. Всередині ж **always** блоку будемо перевіряти стан i\_arst\_n. Якщо ~i\_arst\_n (не i\_arst\_n) приймає значення 1, значить сигнал i\_arst\_n дорівнює нулю, вхід зкидання активний, тому записуємо 0 в змінну, що зберігає стан тригера. Якщо ж i\_arst\_n приймає значення 1, значить **always** блок спрацював по передньому фронту сигналу синхронізації і раз вхід



зкидання не активний, необхідно записати в змінну, що зберігає стан тригера нове значення з входу даних.

**Лістинг 1.4** - Шаблон опису D-тригера синхронного по передньому фронту з входом асинхронного скидання по низькому рівню, без входу дозволу запису

```
module flip_flop(i_clk, i_arst_n, i_data, o_data);

input      i_clk;
input      i_arst_n;
input      i_data;
output reg o_data;

always @(posedge i_clk, negedge i_arst_n) begin
    if (~i_arst_n) begin
        o_data <= 1'b0;
    end else begin
        o_data <= i_data;
    end
end

endmodule
```

Наостанок, розглянемо шаблон коду для опису D-тригера синхронного по передньому фронту з входом асинхронного скидання по низькому рівню і входом дозволу запису по високому рівню (лістинг 1.5).

**Лістинг 1.5** - Шаблон опису D-тригера синхронного по передньому фронту з входом асинхронного скидання по низькому рівню і входом дозволу запису

```
module flip_flop(i_clk, i_arst_n, i_we, i_data, o_data);

input      i_clk;
input      i_arst_n;
input      i_we;
input      i_data;
output reg o_data;

always @(posedge i_clk, negedge i_arst_n) begin
    if (~i_arst_n) begin
        o_data <= 1'b0;
    end else begin
        if (i_we)
            o_data <= i_data;
    end
end

endmodule
```

Вхід дозволу запису  $i\_we$  в даному випадку синхронний, тож в список чутливості оператору @ ми його не додаємо. Якщо ж відбувся передній фронт, сигнал  $i\_arst\_n$  не активний, а сигнал  $i\_we$  дорівнює 1, в змінну, що зберігає стан тригеру записуємо нове значення з входу.

### 1.2.11 Паралельний регістр

Синхронний по фронту паралельний регістр дуже схожий на синхронний по фронту D-тригер. З тією різницею, що тригер зберігає 1 біт даних, а паралельний регістр зберігає багаторозрядне число. Відповідно, паралельні регістри використовують для збереження багаторозрядних даних (наприклад, регістри мікропроцесору).

Як видно з рис.1.4 синхронний по фронту паралельний регістр складається з кількох синхронних по фронту D-тригерів, у яких входи синхронізації з'єднані між собою і підключені до спільного джерела сигналу синхронізації.

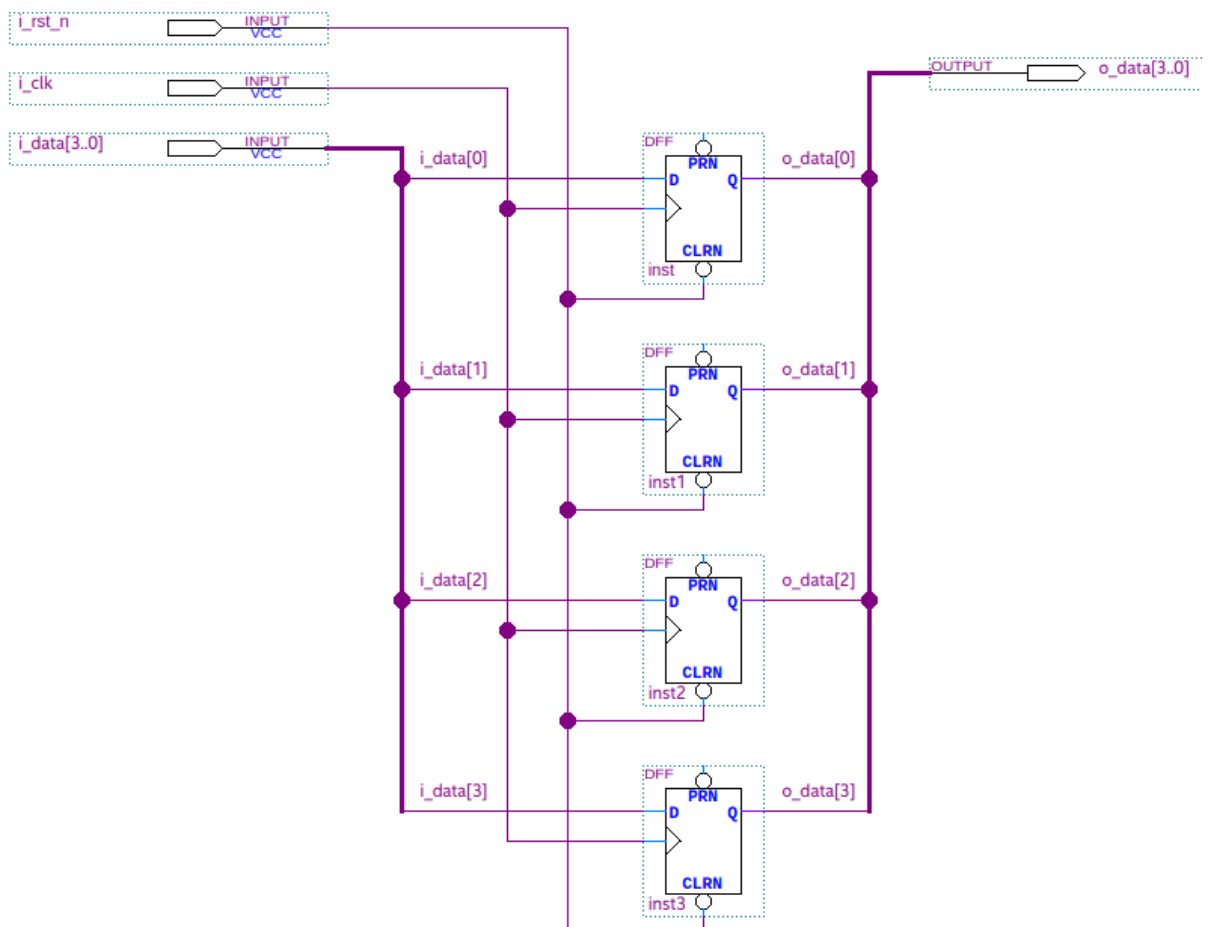


Рис.1.4 - Схема паралельного регістру синхронного по передньому фронту

Входи асинхронного скидання D-тригерів, що утворюють паралельний регістр, теж з'єднанні між собою і підключені до спільного входу асинхронного скидання регістру. Аналогічно можна реалізувати вхід асинхронного встановлення розрядів регістру в одиниці, або взагалі в будь які довільні значення. Наприклад, якщо ми хочемо, щоб по сигналу асинхронного ресету 4-х розрядний регістр встановлювався у значення b1100 (в двійковій системі звісно), необхідно підключити вхід асинхронного ресету регістру до входів асинхронного скидання двох молодших розрядів та до входів асинхронного встановлення двох старших розрядів.

Логіка роботи синхронного по фронту паралельного регістру така ж сама, як і у синхронного по фронту D-тригера. У момент активного фронту дані з багаторозрядного входу даних i\_data[3..0] записуються в регістр і з невеликою затримкою з'являються на багаторозрядному виході даних o\_data[3..0] (у тригерів, що входять до складу FPGA мікросхем Intel, активний фронт D-тригерів передній).

Не забувайте, що для D-тригерів, що утворюють паралельний регістр, обов'язково повинні витримуватись затримки tsetup і thold про які йшла мова в розділі 1.2.1.

В лістингу 1.6 наведений опис паралельного регістру синхронного по передньому фронту на мові Verilog. Для зміни розрядності такого регістру достатньо змінити розрядність i\_data, o\_data та константи 4'b0000. Розрядність регістру можна зробити конфігурованою за допомогою механізму параметрів Verilog, який ми вивчимо пізніше.

**Лістинг 1.6** - Опис паралельного регістру синхронного по передньому фронту на мові Verilog

```
module paral_reg(i_clk, i_rst_n, i_data, o_data);

    input          i_clk;
    input          i_rst_n;
    input [3:0]    i_data;
    output reg [3:0] o_data;

    always @(posedge i_clk, negedge i_rst_n) begin
        if(~i_rst_n) begin
            o_data <= 4'b0000; // you may use decimal consts, e.g. 4'd0
        end else begin
            o_data <= i_data;
        end
    end

endmodule
```

Результат синтезу Verilog коду з лістингу 1.6 в RTL Viewer наведено на рис.1.5.

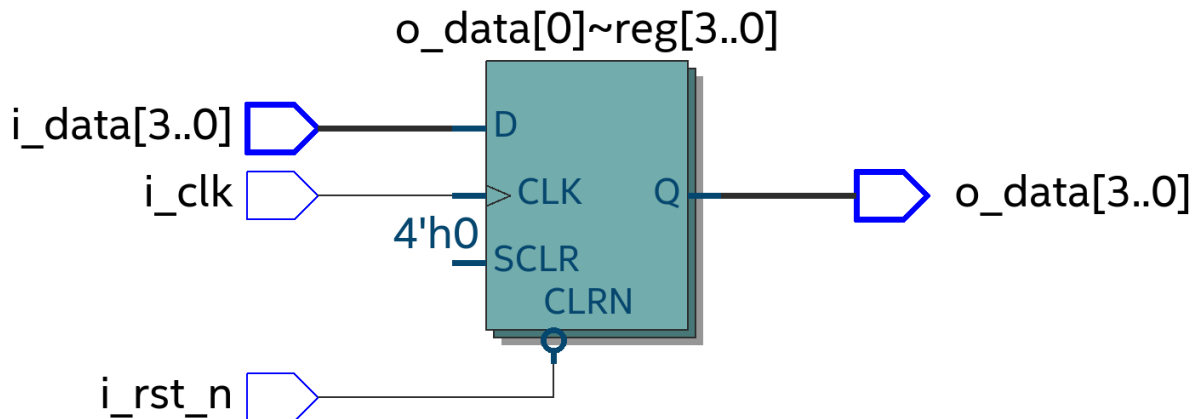


Рис.1.5 - Результат синтезу Verilog коду з лістингу 1.6 в RTL Viewer

На рис.1.4.1 показано паралельний регістр без входу дозволу запису. Але такий вхід не складно додати, якщо будувати регістр на базі синхронних по фронту D-тригерів з входом дозволу запису. Входи дозволу запису тригерів з'єднуються і підключаються до входу дозволу запису регістру. Відповідно, дані будуть записуватись в такий регістр лише за умови, що під час надходження активного фронту на вхід синхронізації, на вході дозволу запису буде присутній активний логічний рівень (у тригерів, що входять до складу FPGA мікросхем Intel, високий активний рівень входів дозволу запису).

### 1.2.12 Регістр зсуву, оператор конкатенації мови Verilog

Регістри зсуву використовують для послідовної передачі багаторозрядних даних (біт за бітом). Яскравий приклад - інтерфейс SPI (розд. 1.2.17). На базі регістру зсуву можна побудувати генератор псевдовипадкової послідовності (розд. 1.2.14). Регістри зсуву використовують для побудови високочастотних one-hot лічильників.

Схему 4-розрядного регістру зсуву наведено на рис.1.6. Стан бітів регістрів виведено на шину `o_data[3..0]`. В даному прикладі досліджуємо 4-розрядний регістр, але загалом розрядність може бути довільна. Регістр має однорозрядний вхід `i_data`, вхід синхронізації `i_clk` на який поступають періодичні імпульси сигналу синхронізації та вхід `i_rst_n` для асинхронного скидання розрядів регістру в нулі.

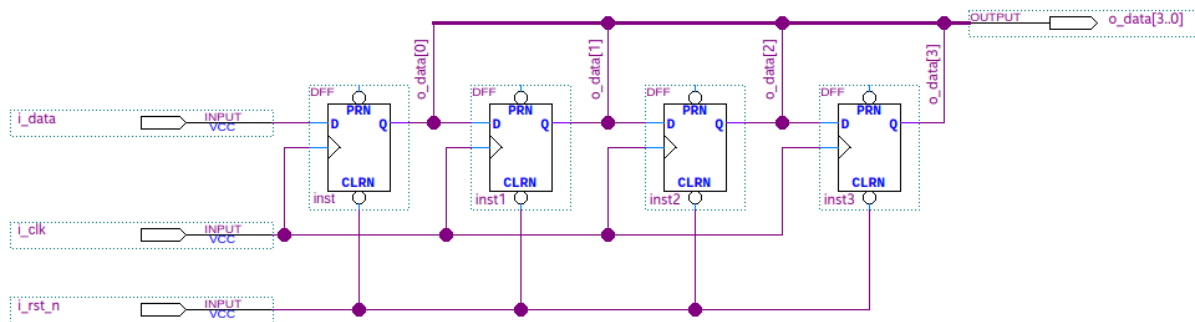


Рис.1.6 - Схема регістру зсуву

Логіка роботи регістру зсуву наступна. По кожному активному (передньому) фронту на вході синхронізації `i_clk` значення з виходу кожного попереднього D-тригера переписується в наступний D-тригер, а у наймолодший D-тригер `o_data[0]` записуються дані з входу `i_data`.

Зміна сигналів схеми з рис.1.6 показана на рис.1.7.

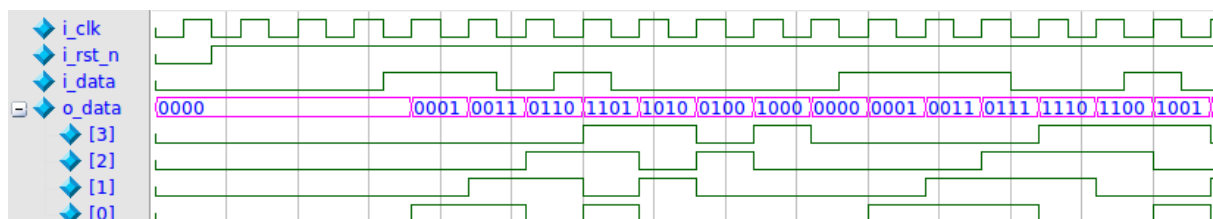


Рис.1.7 - Часові діаграми регістру зсуву

Опис регістру зсуву на мові Verilog наведено в лістингу 1.7. Зверніть увагу, що єдина глобальна відмінність від паралельного регістру з лістингу 1.6 полягає у строчці Verilog коду, що визначає запис нових даних в змінну `shift_reg` типу `reg`:

```
shift_reg <= {shift_reg[2:0], i_data};
```

В даному випадку використовуються оператор конкатенації `{ }` мови Verilog. За допомогою оператора конкатенації можна створити нову шину сигналів з існуючих шин і однорозрядних сигналів типів `wire` і `reg`. Сигнали, що об'єднуються в шину перераховуються через кому. При цьому крайній правий сигнал представляє молодший біт шини, що створюється. А крайній лівий сигнал представляє старший біт шини, що створюється.

Отже зазначена вище строчка Verilog коду створює нову шину, молодшим бітом якої є сигнал `i_data`, а далі йдуть три біти `shift_reg[2:0]`. По активному фронту сигналу синхронізації, за відсутності асинхронного зкидання, створена шина записується у змінну `shift_reg`. Тобто запис відбувається наступним чином (справа від знаку `<--` показані старі значення `shift_reg`):

```
shift_reg[0] <-- i_data
shift_reg[1] <-- shift_reg[0]
shift_reg[2] <-- shift_reg[1]
shift_reg[3] <-- shift_reg[2]
```

Така логіка роботи, коли по активному фронту сигналу синхронізації значення попереднього розряду переписується в наступний розряд, відповідає логіці роботи регістру зсуву. Тому САПР Quartus Prime і синтезує регістр зсуву з шаблону коду, що наведений в лістингу 1.7.

**Лістинг 1.7** - Опис регістру зсуву на мові Verilog

```
module shift_reg (i_clk, i_rst, i_data, o_data);

input          i_clk;
input          i_rst_n;
input          i_data;
output [3:0]   o_data;

reg [3:0] shift_reg;

assign o_data = shift_reg;

always @(posedge i_clk, negedge i_rst_n) begin
    if (~i_rst_n) begin
        shift_reg <= 4'd0;
    end else begin
        shift_reg <= {shift_reg[2:0], i_data};
    end
end

endmodule
```

### 1.2.13 Регістр зсуву з входом дозволу зсуву

В реєстрі зсуву з попереднього розділу зсув відбувається після кожного активного фронту сигналу синхронізації. А в цифрових мікросхемах сигнал синхронізації представляє собою періодичні імпульси з високою частотою. Щоб мати можливість робити зсув лише в задані моменти часу знадобиться реєстр з входом дозволу зсуву схема якого зображена на рис.1.8.

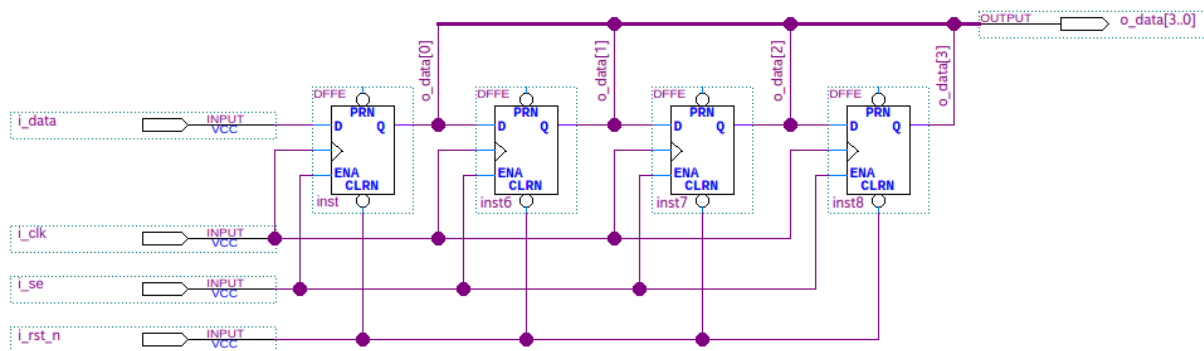


Рис.1.8 - Схема реєстру зсуву з входом дозволу зсуву

Схема на рис.1.8 побудована на базі синхронних по фронту D-тригерів з входом дозволу запису. Входи дозволу запису з'єднані між собою і підключені до входу дозволу зсуву регістру i\_se (Shift Enable). Зсув відбувається лише якщо в момент приходу активного фронту сигналу синхронізації на вході дозволу зсуву i\_se присутній високий рівень (згадаємо, що у тригерів середині мікросхем Intel FPGA активний високий рівень дозволу запису).

Зміна сигналів у схемі з рис.1.8 наведена на рис.1.9.

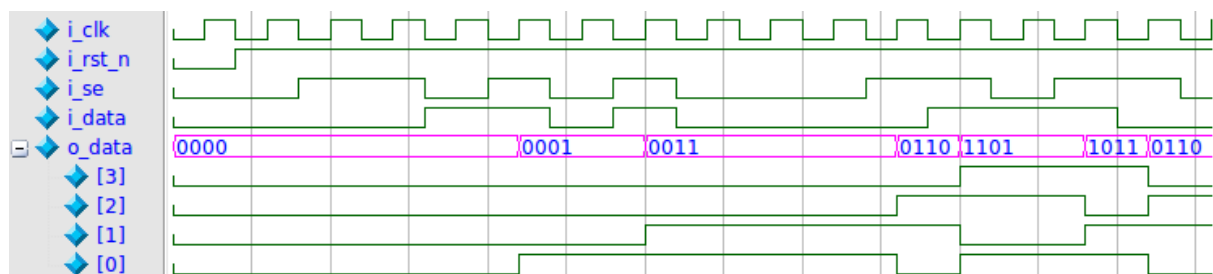


Рис.1.9 - Часові діаграми реєстру зсуву з входом дозволу зсуву

Опис регістру зсуву з входом дозволу зсуву на мові Verilog наведений в ліст. 1.8.

### Лістинг 1.8 - Опис регістру зсуву з входом дозволу зсуву на мові Verilog

```
module shift_reg_we (i_clk, i_rst_n, i_se, i_data, o_data);

    input          i_clk;
    input          i_se;
    input          i_rst_n;
    input          i_data;
    output [3:0]   o_data;

    reg [3:0]      shift_reg;

    assign o_data = shift_reg;

    always @(posedge i_clk, negedge i_rst_n) begin
        if (~i_rst_n) begin
            shift_reg <= 4'd0;
        end else begin
            if(i_se)
                shift_reg <= {shift_reg[2:0], i_data};
        end
    end

endmodule
```

#### 1.2.14 Регістр зсуву з лінійним зворотнім зв'язком (LFSR)

На базі регістру зсуву можна побудувати схему для генерації псевдовипадкових послідовностей бітів. Послідовність бітів називається псевдовипадковою, оскільки з точки зору статистичних тестів на випадковість значення бітів, що створюються, поведуть себе, як випадкова величина з рівномірно випадковим законом розподілення.

Знаючи схему генератора можна спрогнозувати значення кожного наступного біту, що створюється. Після зкидання стану такого генератора в 0 послідовність починається спочатку.

Схема генерації псевдовипадкових послідовностей бітів на базі регістру зсуву називається LFSR (Linear Feedback Shift Register).

Схема 4-розрядного LFSR наведена на рис.1.10. Можна створити LFSR генератор з довільною розрядністю.

LFSR генератор створює  $2^N - 1$  випадкових N-розрядних числа, де N - розрядність LFSR.



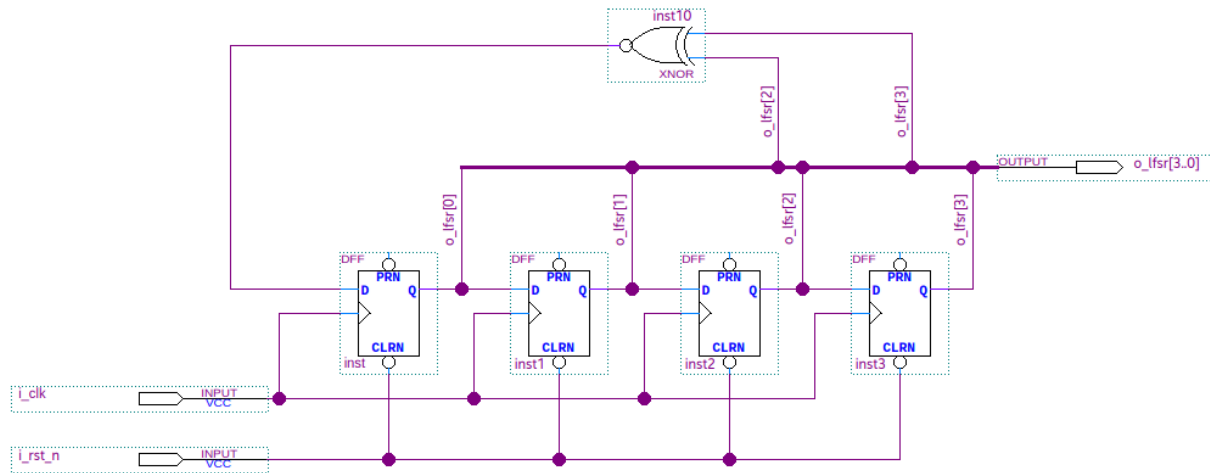


Рис.1.10 - Схема 4-розрядного LFSR генератора псевдовипадкової послідовності бітів

Як видно з рис.1.10 схема LFSR генератора побудована на базі регістр зсуву. Значення кількох розрядів регістру зсуву подаються на входи логічного елементу XNOR, а значення з виходу цього логічного елементу є псевдовипадковим бітом, що записується в молодший розряд регістру LFSR після кожного наступного активного фронту сигналу синхронізації.

Зміна сигналів для LFSR генератора з рис.1.10 наведено на рис.1.11.

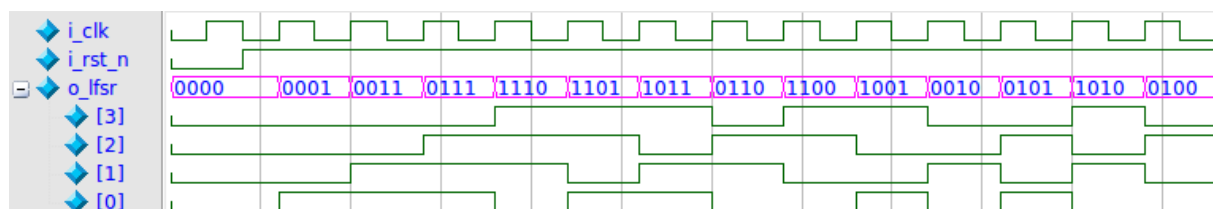


Рис.1.11 - Часові діаграми 4-розрядного LFSR генератора псевдовипадкової послідовності бітів

У разі використання логічного елементу XNOR стан регістру, коли всі його розряди приймають значення лог.1. є забороненим, оскільки в такому випадку на виході XNOR завжди буде лог. 1 і LFSR зациклиться (зависне) в цьому стані.

За необхідності замість лог. елементу XNOR можна використовувати лог. елемент XOR, однак в цьому випадку забороненим станом для LFSR будуть нулі у всіх розрядах.

Дуже важливо правильно вибрати значення розрядів регістру LFSR, які подаються на входи логічного елементу XNOR, щоб отримати максимальну довжину псевдовипадкової послідовності рівну  $2^N - 1$ . Для різних розрядностей LFSR генератора значення розрядів зворотнього зв'язку можна взяти з рис.1.12, або із документу за посиланням [1.1]. Зверніть увагу, що на рис.1.12 та в таблиці за посиланням [1.1] розряди LFSR нумеруються від 1 до N, де 1 молодший розряд, а N - старший розряд. У мові ж Verilog зазвичай нумерують розряди від 0 до N-1, де 0 молодший розряд, а (N - 1) старший розряд.

В лістингу 1.9 наведено опис 4-розрядного LFSR генератора псевдовипадкової послідовності бітів на мові Verilog.

**Лістинг 1.9** - Опис 4-розрядного LFSR генератора псевдовипадкової послідовності бітів на мові Verilog

```

module lfsr4bit(i_clk, i_rst_n, o_lfsr);

input          i_clk;
input          i_rst_n;
output [3:0]   o_lfsr;

reg [3:0]      lfsr;

assign o_lfsr = lfsr;

wire lfsr_lsb = ~(lfsr[3] ^ lfsr[2]);

always @(posedge i_clk, negedge i_rst_n) begin
    if (~i_rst_n) begin
        lfsr <= 4'd0;
    end else begin
        lfsr <= {lfsr[2:0], lfsr_lsb};
    end
end

endmodule

```

Регістр зсуву з лінійним зворотнім зв'язком (LFSR) застосовують у криптографії (потоккові шифри), для розрахунку циклічних контрольних сум CRC з метою виявлення помилок в прийнятих даних, для генерації тестових послідовностей під час тестування / самотестування цифрових мікросхем (built-in self-test, BIST), і в скремблерах телекомунікаційних систем.

## Linear Feedback Shift Register Taps

Table 1 lists the appropriate taps for maximum-length LFSR counters of up to 168 bits. The outputs are designated as 1 through  $n$  with 1 as the first stage.

Table 1: Taps for Maximum-Length LFSR Counters

$n$	XNOR from	$n$	XNOR from	$n$	XNOR from	$n$	XNOR from
3	3,2	45	45,44,42,41	87	87,74	129	129,124
4	4,3	46	46,45,26,25	88	88,87,17,16	130	130,127
5	5,3	47	47,42	89	89,51	131	131,130,84,83
6	6,5	48	48,47,21,20	90	90,89,72,71	132	132,103
7	7,6	49	49,40	91	91,90,8,7	133	133,132,82,81
8	8,6,5,4	50	50,49,24,23	92	92,91,80,79	134	134,77
9	9,5	51	51,50,36,35	93	93,91	135	135,124
10	10,7	52	52,49	94	94,73	136	136,135,11,10
11	11,9	53	53,52,38,37	95	95,84	137	137,116
12	12,6,4,1	54	54,53,18,17	96	96,94,49,47	138	138,137,131,130
13	13,4,3,1	55	55,31	97	97,91	139	139,136,134,131
14	14,5,3,1	56	56,55,35,34	98	98,87	140	140,111
15	15,14	57	57,50	99	99,97,54,52	141	141,140,110,109
16	16,15,13,4	58	58,39	100	100,63	142	142,121
17	17,14	59	59,58,38,37	101	101,100,95,94	143	143,142,123,122

Рис.1.12 - Номери розрядів зворотнього зв'язку для різних розрядностей LFSR генератора псевдовипадкової послідовності бітів

### 1.2.15 Синхронізація даних з входів FPGA

Як було зазначено в розділі 1.2.1, сигнал на вході даних D-тригера не повинен змінюватись одночасно з активним фронтом сигналу синхронізації CLK. Сигнал на вході даних повинен бути незмінним протягом часу  $t_{su}$  перед приходом активного фронту та протягом часу  $t_{hold}$  після приходу активного фронту сигналу синхронізації CLK (рис.1.2). Якщо ця умова порушується, тригер з високою ймовірністю може перейти в метастабільний стан.

Нагадаємо, що коли тригер знаходиться в метастабільному стані на його виході Q присутня напруга, що дорівнює половині напруги живлення ( $V_{dd}/2$ ). Це не лог. 0 і не лог. 1, а напруга із забороненого діапазону значень. Тригер знаходиться у метастабільному стані невеликий випадковий проміжок часу, а потім виходить в стан лог.0, або в стан лог.1 випадковим чином.

Дотримання таймінгів  $t_{su}$  і  $t_{hold}$  не є проблемою всередині мікросхеми, де всі D-тригери підключені до одного джерела сигналу синхронізації. Але що робити, якщо на вхід даних D-тригеру сигнал приходить ззовні мікросхеми і теоретично може змінюватись одночасно з активним фронтом сигналу синхронізації, що формується всередині мікросхеми?

Розглянемо ситуацію зображену на рис.1.13. Маємо D-тригери всередині певної мікросхеми, що тактуються від сигналу синхронізації CLK-B. Активний фронт тригерів передній. Вхід одного з таких тригерів виведений на зовні мікросхеми через контакт Din. Сигнал на вхід Din подається з виходу D-тригеру, який міститься всередині іншої мікросхеми і тактується від сигналу синхронізації CLK-A.

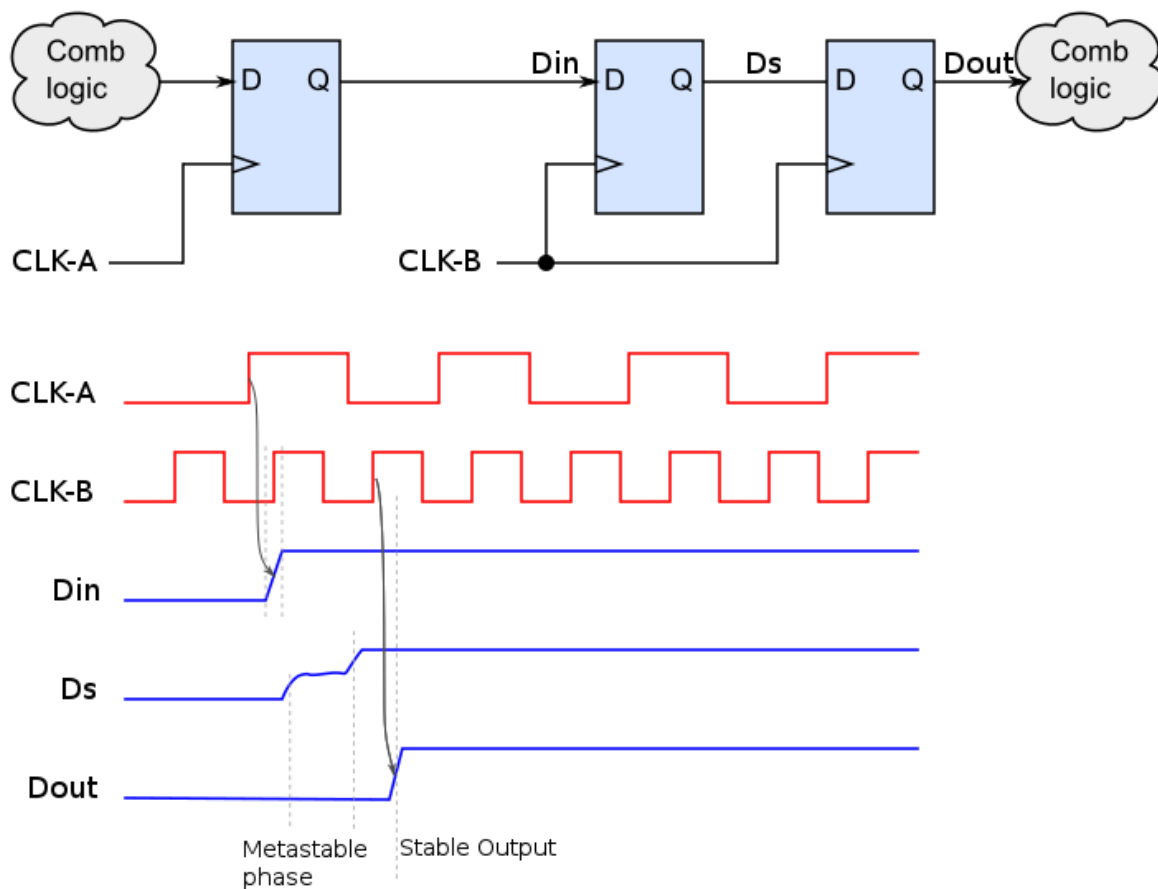


Рис.1.13 - Спосіб синхронізації зовнішніх сигналів, асинхронних до тактового сигналу CLK-B

Взаємозв'язок між тактовими сигналами CLK-A і CLK-B часто невідомий, тому цілком можлива ситуація, коли сигнал на вході Din буде змінюватись одночасно з

переднім фронтом CLK-B. Сигнал на вході Din може формуватися не обов'язково якимось іншим D-тригером. Це може бути сигнал, наприклад, від кнопки, яку натиснув користувач. Але основна проблема залишається незмінною - сигнал на вході Din може змінюватись одночасно з переднім фронтом CLK-B і це може призвести до входження тригера в метастабільний стан.

Проблема метастабільного стану полягає не лише у тому, що з метастабільного стану тригер виходить випадковим чином. Оскільки D-тригер виходить з метастабільного стану з затримкою, значить нове стійке значення на виході D-тригера з'являється теж з затримкою і з затримкою подається на вхід комбінаційної логіки, до якої підключений цей вихід D-тригера. Відповідно на виході комбінаційної логіки та на вході даних наступного D-тригера сигнал з'являється пізніше ніж планувалося, що може привести (і часто приводить!) до порушення setup/hold таймінгів наступного D-тригера. Таким чином метастабільний стан може дуже швидко розповсюдитись по великій кількості D-тригерів мікросхеми, привести до неправильних значень сигналів керування та до краху роботи системи. Проблемність подібних помилок полягає у тому, що їх дуже складно відслідкувати, оскільки для виникнення метастабільності необхідна складна комбінація факторів. Тому більшу частину часу система може працювати коректно і збоїти у випадкові непередбачувані моменти часу.

Хороша новина полягає в тому, що для усунення метастабільності на однорозрядному вході достатньо включити послідовно два D-тригери, як це зроблено на рис.1.13. Якщо перший тригер увійде в метастабільний стан, у нього буде цілий період тактового сигналу щоб вийти з метастабільного стану у стійке значення 0, або 1. По наступному активному фронту це стабільне значення запишеться в наступний тригер. Якщо тригер вийде з метастабільного стану не в те значення, яке подавалося на вхід Din в момент активного фронту CLK-B, обидва тригери ланцюжка синхронізації просто залишаться в старих значеннях, а нове значення на вході Din коректно зчитується по наступному активному фронту і в найгіршому випадку з'явиться на виході другого тригера Dout з затримкою в 1 такт.

Такий підхід називається double-flopping і широко використовується для синхронізації однорозрядних даних зчитаних з входів цифрових мікросхем. Для більшої надійності іноді послідовно включають не 2, а 3 тригери.

Double-flopping не застосовують до синхронізації багаторозрядних входів, оскільки частина розрядів може вийти з метастабільності у правильні значення, а частина розрядів може вийти з метастабільності у хибні значення, що приведе до одержання неправильних даних. У випадку 1-розрядного сигналу такої проблеми не виникає і ми у будь-якому випадку зчитуємо правильне значення, лише іноді з затримкою в 1 такт. Тому при передачі багаторозрядних даних саму шину даних не синхронізують, а синхронізують лише 1-бітний сигнал дозволу запису, що передається разом з шиною даних. Інший спосіб - використання асинхронної черги FIFO, в яку дані записуються по одному сигналу синхронізації, а зчитуються по іншому сигналу синхронізації. Схему асинхронного FIFO розглянемо в одній із наступних робіт.

Детальніше про боротьбу з метастабільністю можна почитати в [\[1.2\]](#).

### 1.2.16 Детектор фронту

Схема детектора переднього фронту наведена на рис.1.14.

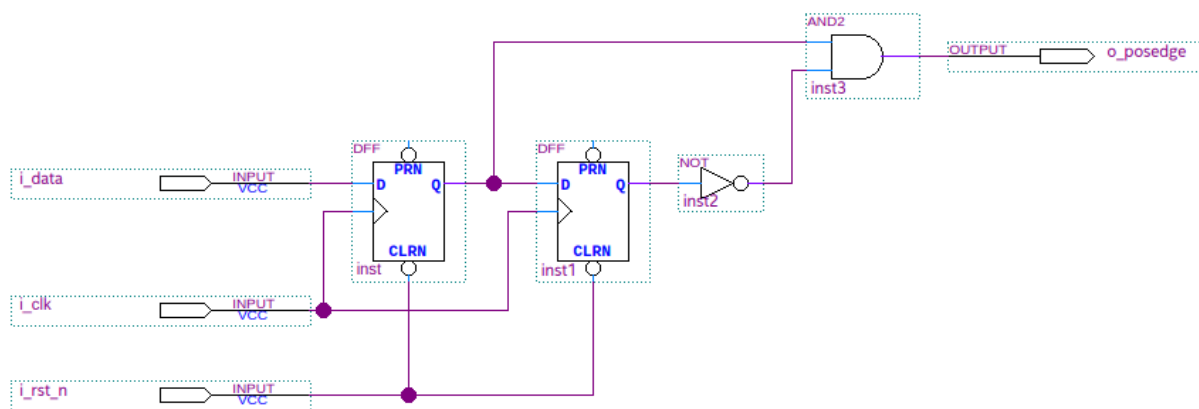


Рис.1.14 - Схема детектора переднього фронту

Часові діаграми зміни сигналів схеми з рис.1.14 наведені на рис.1.15.

Як видно з рис.1.14, детектор фронту містить вхід `i_data` на який подається сигнал передній фронт якого необхідно визначити. Також детектор фронту містить вхід `i_rst_n` для асинхронного зкидання обох тригерів та вхід `i_clk` на який подається сигнал синхронізації. Частота сигналу на вході `i_clk` повинна бути набагато більша (бажано, разів в 10 і вище) ніж частота зміни сигналу на вході `i_data`.

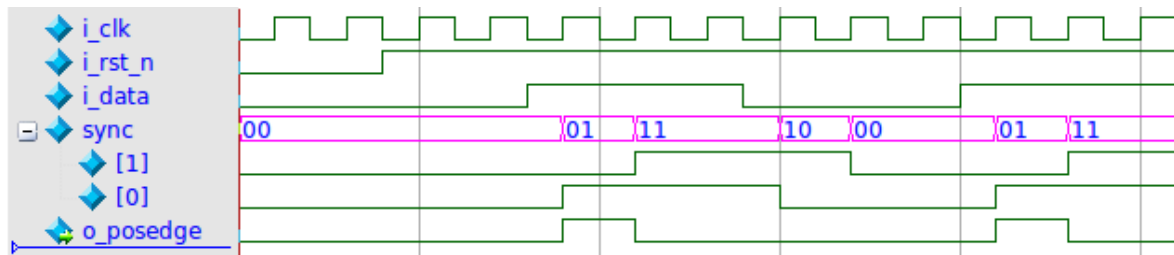


Рис.1.15 - Часові діаграми сигналів у схемі детектору переднього фронту

Після надходження переднього фронту на вхід `i_data`, на виході `o_posedge` з'являється імпульс тривалістю один період сигналу `i_clk`. Розглянемо чому так відбувається. Якщо на вході `i_data` тривалий час присутній лог.0, обидва тригери будуть в лог.0 і на виході елементу AND буде 0. Припустимо на вхід `i_data` надійшов сигнал лог. 1 і відповідне значення записалося в перший тригер, а в другому тригері поки що зберігається 0. В такій ситуації на виході елементу AND буде присутній сигнал лог.1. По наступному активному фронту лог. 1 запишеться і в другий тригер. Тепер обидва тригери містять лог.1 і на виході елементу AND буде 0.

Опис детектору переднього фронту на мові Verilog наведено в лістингу 1.10. З аналогічних міркувань не складно створити детектор заднього фронту. Якщо ж виходи тригерів подати на входи елементу XOR, отримаємо детектор, що буде формувати одиночний імпульс після надходження як переднього, так і заднього фронтів.

**Лістинг 1.10** - Опис детектору переднього фронту на мові Verilog

```

module edge_detector (i_clk, i_rst_n, i_data, o_posedge);

input      i_clk;
input      i_rst_n;
input      i_data;
output     o_posedge;

reg       [1:0]  sync;

assign o_posedge = ~sync[1] & sync[0];

always @(posedge i_clk, negedge i_rst_n) begin
    if(~i_rst_n) begin
        sync <= 2'b00;
    end else begin
        sync <= {sync[0], i_data};
    end
end

endmodule

```

### 1.2.17 Інтерфейс SPI

Абревіатура SPI розшифровується як Serial Peripheral Interface. Це широко розповсюджений інтерфейс за допомогою якого до мікроконтролерів зазвичай підключають різноманітні сенсори, невеликі дисплеї, модулі пам'яті та інші периферійні пристрої, що не потребують високої швидкості передачі даних.

Інтерфейс SPI зазвичай використовують для підключення одного головного пристрою (master device) до одного або кількох периферійних пристроїв (slave devices). Головним пристроєм виступає найчастіше мікроконтролер.

Сигнали інтерфейсу SPI наведено на рис.1.16. MOSI служить для передачі даних від головного пристрою до периферійного пристрою (MOSI - Master Output Slave Input). MISO служить для передачі даних від периферійного пристрою до головного пристрою (MISO - Master Input Slave Output). SCK є тактовим сигналом синхронізації передачі, який створює головний пристрій (мікроконтролер). Відповідно SCK направлений від мікроконтролеру до периферійного пристрою/пристроїв.

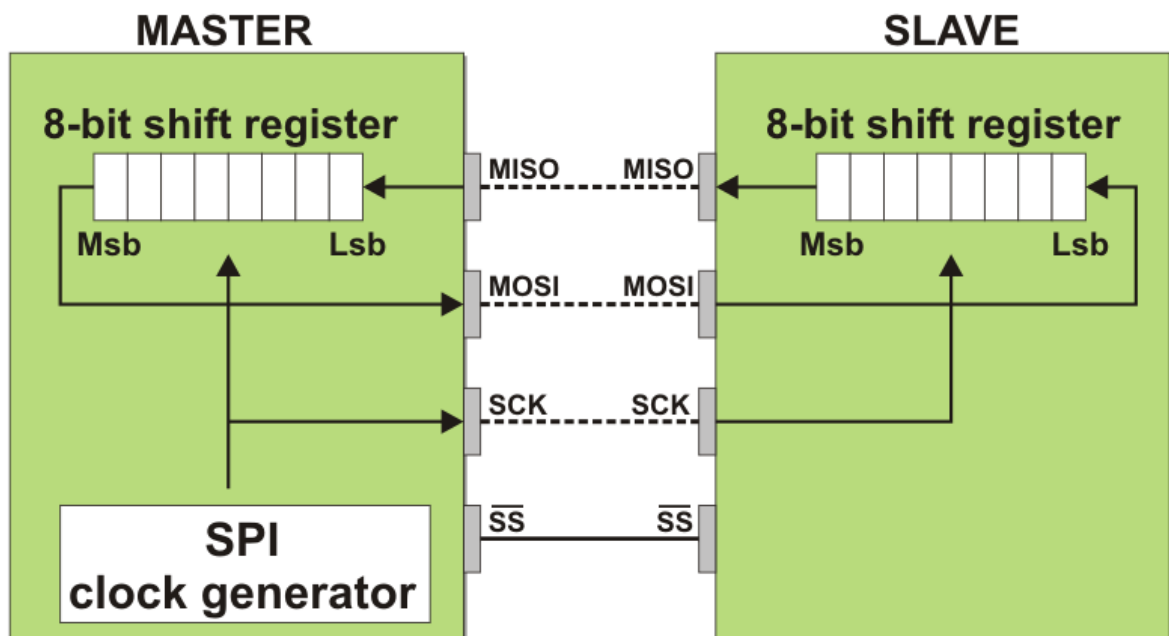


Рис.1.16 - Спрощена схема передачі по інтерфейсу SPI

Сигнал SS розширюється як Slave Select і служить для активації периферійного пристрою мікроконтролером. Якщо до головного пристрою (мікроконтролеру)



підключено кілька периферійних пристроїв, до кожного периферійного пристрою від мікроконтролера йде сигнал Slave Select: SS1, SS2, SS3 і т.д. (рис.1.17). При цьому сигнали SCK, MISO і MOSI будуть спільними для мікроконтролера і периферійних пристроїв (іноді сигнал SCK позначають як SCLK, а SS позначають як CS).

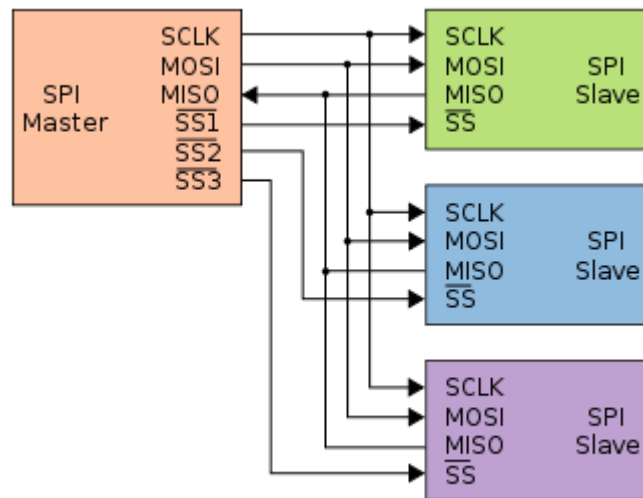


Рис.1.17 - Підключення кількох SPI приймачів до SPI передавача

Сигнали SS зазвичай мають активний низький логічний рівень. Це означає, що мікроконтролер для передачі даних по SPI певному периферійному пристрою виставляє логічний 0 на сигналі SS, що підключений для цього пристрою.

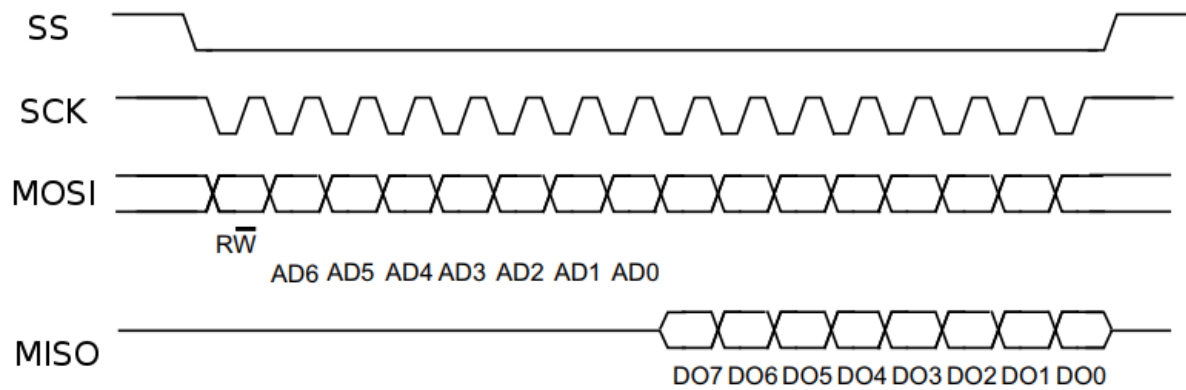
В кожен момент часу може бути активним лише один сигнал SS (в кожен момент часу мікроконтролер працює лише з одним периферійним пристроєм). Така умова впливає з обмеження, що у провідника MISO повинен бути лише один драйвер сигналу. Якщо у провідника буде кілька драйверів сигналу (наприклад, провідник підключений одночасно до виходів двох логічних елементів, це може призвести до короткого замикання, коли один вихід виставить лог.1, а інший вихід виставить лог.0).

Особливістю SPI є той факт, що передача і прийом даних відбуваються одночасно. Всі транзакції (прийом/передача) ініціює головний пристрій (master device). Якщо повернутися до рис.1.16 видно, що для передачі даних в SPI використовують регістри зсуву. Зсув здійснюється по сигналу SCK. Як головний, так і периферійні пристрої зчитують дані по активному фронту сигналу SCK, який формує головний пристрій (мікроконтролер). По неактивному фронту SCK головний пристрій видає на MOSI

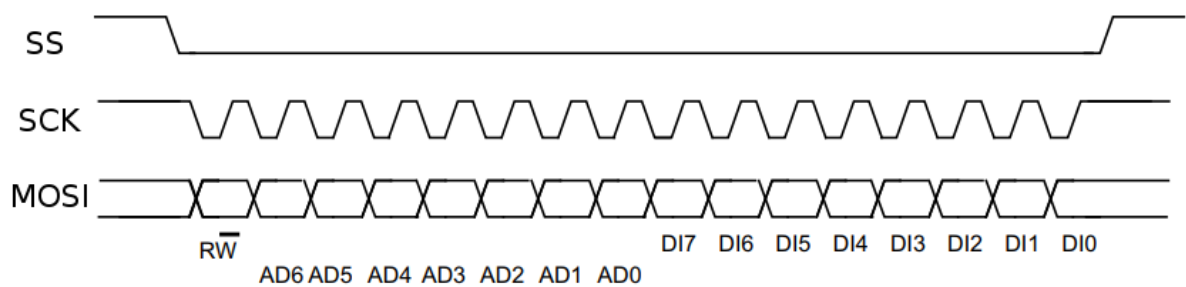
новий біт даних, а периферійний пристрій видає новий біт даних на MISO. Імпульси на SCK формуються лише під час передачі даних. Якщо дані по MOSI/MISO не передаються, імпульси на SCK відсутні. Дані зазвичай передаються по 8 біт підряд (хоча іноді використовують і 16 бітну передачу). Кожному біту даних, відповідає 1 імпульс на SCK. Дані зазвичай передаються старшим бітом вперед (старший біт часто позначають як MSB - Most Significant Bit, а молодший біт позначають як LSB - Least Significant Bit). Хоча в деяких периферійних пристроях буває так, що дані передаються молодшим бітом вперед. Мікроконтролер можна налаштувати на передачу по SPI в обох режимах (як старшим, так і молодшим бітом вперед).

Відповідно до рис.1.16 у найпростішому випадку SPI MASTER і SPI SLAVE містять регістри зсуву, що тактуються від SCK. Дані в цих регістрах зсуву зсуваються від молодшого біту до старшого. Спершу в регістри зсуву завантажують необхідні дані. Далі головний пристрій (мікроконтролер) формує 8 імпульсів тактової частоти на лінії SCK і регістри зсуву обмінюються між собою даними.

В лабораторній роботі ми поки що створимо SPI приймач, який лише приймає дані. Але в реальних периферійних пристроях які повинні і приймати і передавати дані, використовують наступний підхід. Спершу мікроконтролер виставляє активний низький рівень (лог.0) на лінії SS периферійного пристрою. Далі мікроконтролер передає в периферійний пристрій 8 біт керування (дані прийняті на MISO на цьому етапі мікроконтролер ігнорує). Переданий байт містить 1 конфігураційний біт R/W (старший біт) та 7 молодших біт адреси, що задають номер регістру периферійного пристрою з яким необхідно обмінюватись даними. Конфігураційний біт R/W задає напрямок передачі наступного байту. Якщо біт напрямку передачі R/W приймає значення 0, наступний байт буде записуватись в регістр периферійного пристрою за переданою щойно адресою. Якщо біт напрямку передачі R/W приймає значення 1, наступний байт буде зчитуватися з регістру периферійного пристрою, що має передану щойно адресу. Після передачі другого байту мікроконтролер знову робить неактивним сигнал SS для даного периферійного пристрою. Приклади запису/зчитування по інтерфейсу SPI з використанням подібного підходу показано на рис.1.18.



a)



b)

Рис.1.18 - Приклади зчитування (а) та запису (b) по інтерфейсу SPI для акселерометра LIS3DSH (для зчитування біт R/W повинен приймати значення 1, для запису - 0)

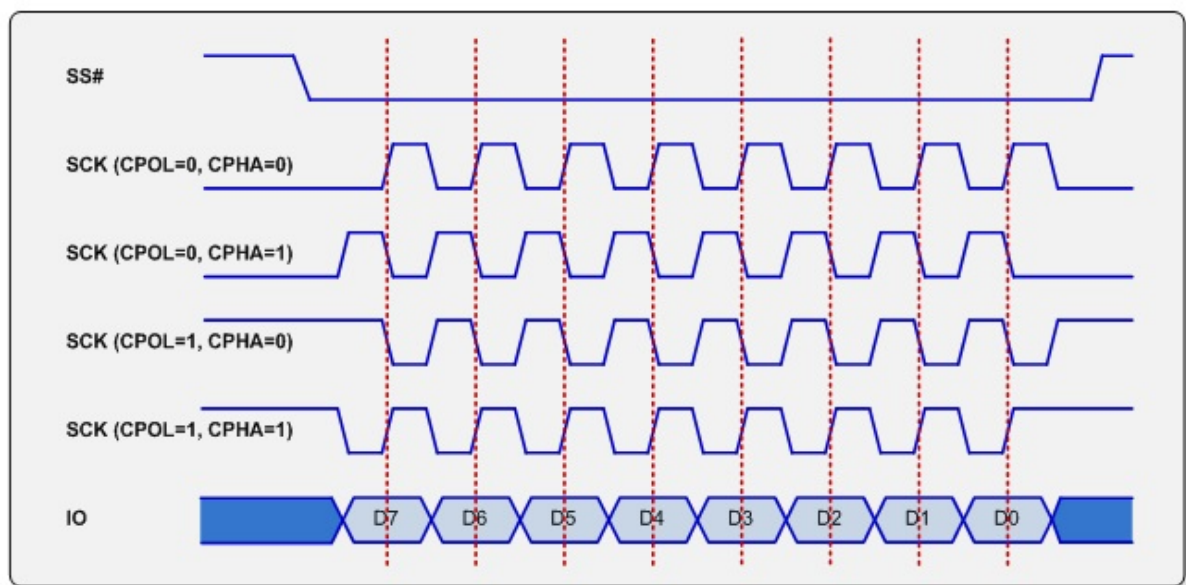


Рис.1.19 - Режими роботи SPI

Інтерфейс SPI може працювати в 4-х режимах, що визначаються параметрами CPOL (скорочення від Clock Polarity) та CPHA (скорочення від Clock Phase). Зазвичай для кожного периферійного пристрою з документації відомі значення CPOL та CPHA і мікроконтролер можна налаштувати на роботу в режимі SPI, що визначається цими відомими значеннями. Значення параметрів CPOL та CPHA пояснені на рис.1.19.

Якщо CPOL=0, це означає, що при відсутній передачі даних по SPI сигнал SCK приймає значення 0. Якщо CPOL=1, то при відсутній передачі даних по SPI сигнал SCK приймає значення 1.

Якщо CPHA=0 це означає, що активним фронтом є перший фронт кожного тактового імпульса на SCK. У випадку коли CPHA=1 активним фронтом є другий фронт кожного тактового імпульса на SCK.

### 1.2.18 SPI приймач

У цьому розділі розглянемо апаратний SPI приймач у якого CPOL=0, CPHA=0. Це означає, що при відсутності передачі даних по SPI на SCK буде присутній 0, а активним фронтом буде передній фронт імпульсу SCK. Дані будуть зчитуватися SPI приймачем із MOSI по передньому фронту SCK і виставлятися SPI передавачем (наприклад, мікроконтролером) на MOSI по задньому фронту SCK.

Схема апаратного SPI приймача наведена за посиланням [\[1.3\]](#). Для зручності перегляду краще завантажити файл на комп'ютер.

Опис SPI приймача, зображеного на схемі [\[1.3\]](#), на мові Verilog наведено в лістингу 1.11.

Зі схеми [\[1.3\]](#) та лістингу 1.11 видно, що всі D-тригери SPI приймача підключені до входу синхронізації i\_clk.

Схема SPI приймача включає два детектори фронту (що детектують активний передній фронт сигналу i\_spi\_sck та передній фронт по якому відбувається деактивація i\_spi\_cs), схему синхронізації сигналу i\_spi\_mosi побудовану на двох послідовно ввімкнених D-тригерах, регістр зсуву з входом дозволу зсуву та паралельний регістр з входом дозволу запису. Зверніть увагу, що на входах детекторів фронтів присутній додатковий D-тригер для їх синхронізації.

Лістинг 1.11 - Опис SPI slave-приймача на мові Verilog

```
module simple_spi(i_clk,i_rst_n,i_spi_mosi,i_spi_sck,i_spi_cs,o_data);

input          i_clk;
input          i_rst_n;
input          i_spi_mosi;
input          i_spi_sck;
input          i_spi_cs;
output [7:0]   o_data;

wire           sck_rs_edg;
wire           tr_cmplt;
reg [7:0]      user_reg_ff;
reg [7:0]      spi_dat_ff;
reg [2:0]      sck_sync_ff;
reg [2:0]      cs_sync_ff;
reg [1:0]      mosi_sync_ff;

assign o_data      = user_reg_ff;
assign sck_rs_edg  = ~sck_sync_ff[2] & sck_sync_ff[1];
assign tr_cmplt    = ~cs_sync_ff[2] & cs_sync_ff[1];

always @(posedge i_clk, negedge i_rst_n)
    if (~i_rst_n) begin
        sck_sync_ff <= 3'b000;
    end else begin
        sck_sync_ff <= {sck_sync_ff[1:0], i_spi_sck};
    end

always @(posedge i_clk, negedge i_rst_n)
    if (~i_rst_n) begin
        cs_sync_ff <= 3'b111;
    end else begin
        cs_sync_ff <= {cs_sync_ff[1:0], i_spi_cs};
    end

always @(posedge i_clk)
    mosi_sync_ff <= {mosi_sync_ff[0], i_spi_mosi};

always @(posedge i_clk)
    if (sck_rs_edg)
        spi_dat_ff <= {spi_dat_ff[6:0], mosi_sync_ff[1]};

always @(posedge i_clk)
    if (tr_cmplt)
        user_reg_ff <= spi_dat_ff;

endmodule
```

Схема обох детекторів фронту зображена на рис.1.20.

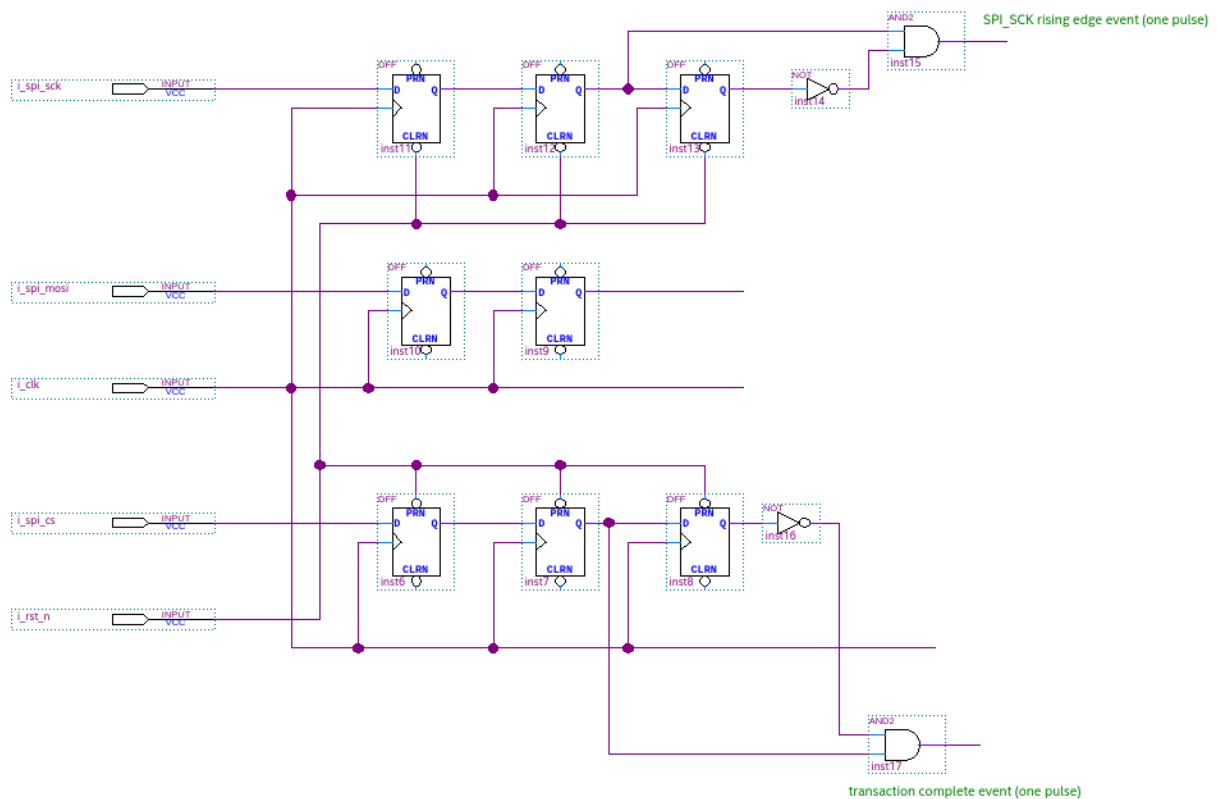


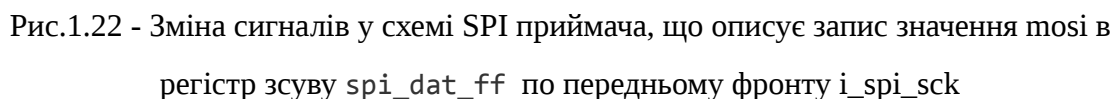
Рис.1.20 - Схеми детекторів переднього фронту сигналів `i_spi_sck`, `i_spi_cs` та схема синхронізації входу `i_spi_mosi` а проекті SPI приймача

Зі схеми [1.3] та лістингу 1.11 видно, що асинхронне зкидання відбувається лише для тригерів детекторів фронтів. Тригери регістрів зсуву та паралельного регістру не підключені до входу асинхронного зкидання `i_rst_n`, як і тригери ланцюжка синхронізації сигналу `mosi`. При цьому тригери детекторів фронтів зкидаються у стан, що відповідає неактивному сигналу на їх входах (0 для `spi_sck` та 1 для `spi_cs`).

Насправді, це нормальна практика, коли зкидання після подачі напруги живлення відбувається лише для тригерів, що впливають на сигнали керування (в нашому випадку це виходи детекторів фронтів). Оскільки під час роботи пристрою регістри даних все одно будуть перезаписані правильними значеннями. Менша ж кількість провідників зкидання у великих проектах може полегшити трасування з'єднань між елементами всередині FPGA мікросхеми.

В лістингу 1.11 ланцюжок синхронізації `mosi` описаний 2-розрядною змінною `mosi_sync_ff` типу `reg`. Тригери детектору переднього фронту сигналу `i_spi_sck`

Зміна сигналів у схемі SPI приймача показана на рис.1.21.



Після надходження переднього фронту на вхід `i_spi_sck`, після чергового активного фронту сигналу `i_clk`, на виході детектору фронту `sck_rs_edg` з'являється лог.1. Оскільки `sck_rs_edg` підключений на вхід дозволу зсуву, по наступному активному фронту `i_clk`, відбувається зсув даних в регістрі `spi_dat_ff` в сторону старших розрядів. Розряд регістру зсуву з номером 1 записується в розряд з номером 2, розряд з номером 0 записується в розряд з номером 1 і т.д. А значення `mosi_sync_ff[1]` з виходу ланцюжка синхронізації `mosi` записується в розряд `spi_dat_ff` з номером 0. По тому ж активному фронту `i_clk`, по якому відбувається зсув, значення виходу детектору фронту `sck_rs_edg` переходить в 0 і по наступному активному фронту `i_clk` зсув уже відбуватися не буде. Наступний зсув відбудеться лише після появи чергового значення 1 на `sck_rs_edg`, після наступного переднього фронту `i_spi_sck`.

Після переходу сигналу `i_spi_cs` в неактивне значення (з 0 в 1), по черговому активному фронту сигналу `i_clk` на виході детектору фронту `tr_cmplt` з'являється 1. Оскільки `tr_cmplt` підключений на вхід дозволу запису регістру даних, по наступному активному фронту `i_clk`, відбувається запис вмісту регістру `spi_dat_ff` в регістр даних `user_reg_ff`. Зміна сигналів, що відповідає описаному випадку позначена на рис.1.23. Варто зауважити, що вміст регістру `user_reg_ff` виводиться на вихідний порт `o_data`.

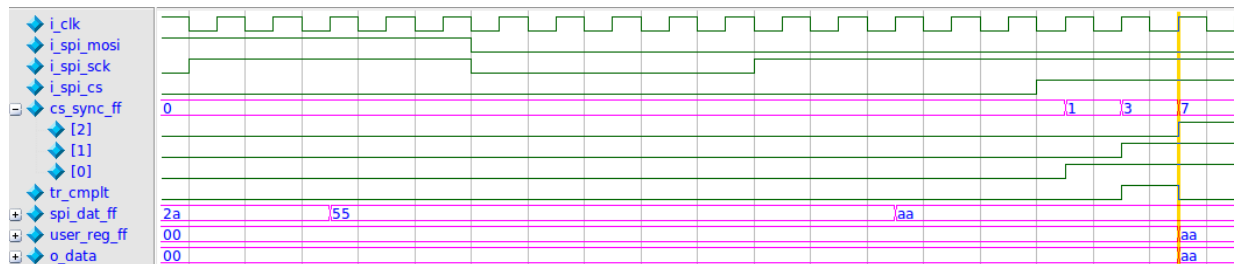


Рис.1.23 - Зміна сигналів у схемі SPI приймача, що описує запис даних з регістру зсуву в регістр користувача по переходу `i_spi_cs` в неактивний стан

В даному найпростішому випадку можна було б не використовувати регістр даних `user_reg_ff` і підключити вихідний порт `o_data` одразу до регістру зсуву `spi_dat_ff`. Однак в реальному більш складному SPI приймачі, який ми побудуємо пізніше, буде кілька регістрів даних доступ до яких буде задаватися конфігураційним байтом (див. розд.1.2.17). І в такому випадку доведеться переписувати вміст `spi_dat_ff` в обраний регістр даних. Тому є сенс розглянути логіку такого переписування вже зараз.



### 1.2.19 Програмний код SPI передавача для Nucleo F401RE

У попередньому розділі був описаний SPI приймач, який можна реалізувати на мікросхемі FPGA. Для передачі даних в описаний SPI приймач можна використати будь який мікроконтролер. Нижче наведені вихідні коди для реалізації SPI передачі на налагоджувальних платах NUCLEO-F401RE і Arduino Leonardo.

Налагоджувальна плата NUCLEO-F401RE побудована на базі мікроконтролеру STM32F401RET6 (процесорне ядро ARM Cortex-M4 з FPU), а її опис можна знайти за посиланням [\[1.4\]](#). Для програмування NUCLEO-F401RE будемо використовувати MBED - хмарне середовище розробки і бібліотеку класів з функціоналом операційної системи реального часу.

Робота в хмарному середовищі розробки описана в розділі 1.3.1, а в даному розділі розглянемо програмний код для передачі даних по SPI на платі NUCLEO-F401RE з використанням бібліотеки класів MBED. Зазначений програмний код наведено в лістингу 1.12.

**Лістинг 1.12** - Програмний код SPI передавача для налагоджувальної плати NUCLEO-F401RE на базі мікроконтролеру STM32 з використанням бібліотеки MBED

```
#include "mbed.h"

SPI spi(D11, D12, D13); // spi_mosi, spi_miso, spi_sck
DigitalOut cs(D10);     // spi_cs

int main()
{
    cs = 1;
    uint8_t val = 0;

    spi.format(8,0);
    spi.frequency(1000000);

    while (1) {
        wait(0.5);
        cs = 0;
        spi.write(val++);
        cs = 1;
    }
}
```

В лістингу 1.12 створюється глобальний об'єкт класу SPI з назвою `spi`. В конструктор об'єкта передаємо номера контактів налагоджувальної плати NUCLEO-F401RE, що будуть використані у якості `spi_sck`, `spi_miso`, `spi_mosi`. Перший аргумент конструктора дозволяє налаштувати номер контакту для `spi_mosi`. Другий аргумент конструктора дозволяє налаштувати номер контакту для `spi_miso`. Третій аргумент конструктора дозволяє налаштувати номер контакту для `spi_sck`. З лістингу 1.12 видно, що у якості `spi_mosi` використовується контакт D11, у якості `spi_miso` використовується контакт D12, у якості `spi_sck` використовується контакт D13.

Також у лістингу 1.12 створюється глобальний об'єкт класу DigitalOut з назвою `cs` для програмного керуванням логичними рівнями на контакті мікроконтролеру. У якості аргументу конструктора передається номер контакту плати NUCLEO-F401RE, яким будемо керувати за допомогою об'єкту `cs`. В даному випадку це контакт D10. Присвоєння об'єкту `cs` значення 1 призводить до появи лог.1 на контакті D10. Присвоєння об'єкту `cs` значення 0 призводить до появи лог.0 на контакті D10. Даний контакт будемо використовувати як `spi_cs` для SPI приймача.

Щоб задати налаштування параметрів передачі по SPI використовуємо методи `format` та `frequency` об'єкту `spi`:

```
spi.format(8,0);  
spi.frequency(1000000);
```

Метод `frequency` дозволяє задати тактову частоту передачі по SPI (частоту сигналу `spi_sck`) в герцах. В даному випадку обрана частота 1 000 000 Гц (1 МГц).

Перший аргумент методу `format` задає кількість біт даних, що передаються по SPI (в нашому випадку 8 біт). Другий аргумент методу `format` задає режим роботи SPI (режим роботи SPI визначає параметри CPOL, CPHA). Відповідність параметрів CPOL, CPHA номеру режиму SPI наведена на рис.1.24. Оскільки наш SPI приймач використовує значення CPOL=0, CPHA=0, обираємо режим 0.

Для передачі байту даних по SPI використовують метод `spi.write(...)`, де аргументом метода є байт, який необхідно передати.

Інші методи класу SPI бібліотеки MBED описані за посиланням [\[1.5\]](#).

Mode	Polarity	Phase
0	0	0
1	0	1
2	1	0
3	1	1

Рис.1.24 - Відповідність параметрів CPOL, CPHA номеру режиму SPI

Функція `wait(...)` затримує виконання програми на кількість секунд, що передається у якості аргументу.

Після подачі живлення на налагоджувальну плату NUCLEO-F401RE створюються зазначені вище глобальні об'єкти і запускається на виконання функція `main`. В функції `main` створюється 8-розрядна змінна `val`, ініціалізується нулем і на початку роботи програми подається неактивний рівень на вихід `cs` (`cs = 1`). Далі викликаються методи налаштування SPI. Після цього у нескінченному циклі чекаємо 0.5 секунди, виставляємо активний рівень на `cs` (`cs = 0`), передаємо значення змінної `val` по SPI і інкрементуємо `val` (`spi.write(val++)`), виставляємо неактивний рівень на `cs` (`cs = 1`), переходимо до наступної ітерації нескінченного циклу. Оскільки змінна `val` 8-розрядна, після досягнення значення 255, наступний інкремент обнулить змінну. Таким чином значення змінної змінюватимуться від 0 (всі розряди байту нулі) до 255 (всі розряди байту одиниці).

### 1.2.20 Програмний код SPI передавача для Arduino

Налагоджувальна плата Arduino Leonardo описана за посиланням [\[1.6\]](#).

Вихідний код SPI передавача для Arduino Leonardo наведено в лістингу 1.13.

В лістингу 1.13 глобальна змінна `spi_cs` задає номер контакту Arduino Leonardo, який буде керувати сигналом `chip select` для SPI. В даному випадку це цифровий контакт Arduino №8.

**Лістинг 1.13** - Програмний код SPI передавача для налагоджувальної плати Arduino Leonardo

```
#include "SPI.h"

int spi_cs = 8;

void setup()
{
    pinMode(spi_cs, OUTPUT);
    digitalWrite(spi_cs, HIGH);
    SPI.begin();
    SPI.beginTransaction(SPISettings(1000000,MSBFIRST,SPI_MODE0));
}

void loop()
{
    for(uint8_t val = 0; val < 256; val++) {
        delay(500);
        digitalWrite(spi_cs, LOW);
        SPI.transfer(val);
        digitalWrite(spi_cs, HIGH);
    }
}
```

Функція `setup()` викликається один раз після подачі напруги живлення на плату Arduino Leonardo, або після перезавантаження кнопкою RESET. Застосовується для початкових налаштувань. В нашому випадку в функції `setup()` ми налаштуємо контакт з номером `spi_cs` на вихід, а також налаштуємо параметри SPI.

Після запуску плати Arduino Leonardo функція `loop()` викликається ітеративно раз за разом. Тобто після виконання коду функції `loop()` і звершення функції ця функція викликається знову. Схоже на нескінченний цикл.

Функція `pinMode(spi_cs, OUTPUT)` викликається, щоб сконфігурувати цифровий контакт Arduino з номером `spi_cs` на вихід. Функція `digitalWrite(spi_cs, HIGH)` виставляє лог.1 на цифровому контакті Arduino з номером `spi_cs`.

Для початкової конфігурації SPI використовуються функції:

```
SPI.begin();
SPI.beginTransaction(SPISettings(1000000,MSBFIRST,SPI_MODE0));
```

Аргумент **1000000** задає тактову частоту передачі по SPI (частоту сигналу spi\_sck) в герцах. В даному випадку обрана частота 1 000 000 Гц (1 МГц). Аргумент **MSBFIRST** визначає, що дані на лініях miso/mosi передаються старшим бітом вперед. Аргумент **SPI\_MODE0** задає режим роботи SPI (CPOL=0, CPHA=0).

Для передачі значення 8-розрядної змінної `val` по SPI використовується функція `SPI.transfer(val)`.

Інші функції для роботи з SPI Arduino описано в [1.7].

Всередині функції `loop()` в циклі `for()` відбувається передача по SPI значень змінної `val` в інтервалі 0 ... 255 з періодом 500 мс. По завершенню цикл функція завершується. Після цього функція `loop()` викликається знову і цикл починається спочатку.

## 1.3 Практична частина

### 1.3.1 Передача даних по SPI з використанням MBED та Nucleo F401RE

Для компіляції коду SPI передавача для налагоджувальної плати NUCLEO F401RE з лістингу 1.12 необхідно виконати наступні кроки:


1. Перейдіть на сайт хмарного середовища розробки MBED за адресою <https://os.mbed.com/compiler>
2. Зареєструйтеся та авторизуйтеся в MBED

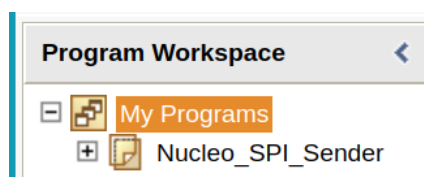
3. Перейдіть за посиланням [1.4] і натисніть кнопку



4. Перейдіть за посиланням:

[https://os.mbed.com/users/korotkiy\\_eugene/code/Nucleo\\_SPI\\_Sender/](https://os.mbed.com/users/korotkiy_eugene/code/Nucleo_SPI_Sender/)

5. Натисніть на кнопку  у правій частині вікна. Проект SPI передавача буде додано до вашого середовища розробки:



6. Натисніть на кнопку вибору платформи у правому верхньому куті середовища розробки MBED (кнопка знаходиться під номером версії MBED) та оберіть налагоджувальну плату NUCLEO-F401RE



7. Після вибору налагоджувальної плати натисніть



8. Для компіляції проекту натисніть



Після завершення компіляції вам запропонують зберегти файл з розширенням .bin, що містить прошивку для налагоджувальної плати NUCLEO-F401RE.

Для програмування NUCLEO-F401RE підключіть плату до комп'ютера. Комп'ютер розпізнає налагоджувальну плату, як USB накопичувач. Зкопіюйте \*.bin файл з прошивкою на цей USB накопичувач. Після того, як файл зникне з накопичувача, програмування NUCLEO-F401RE буде завершене і почнеться виконання створеної вами програми.

Штирьовий роз'єм CN10 з цифровими виводами налагоджувальної плати NUCLEO-F401RE зображено на рис.1.25.

Нагадаємо, що у якості контактів SPI передавача використані D11 (MOSI), D13 (SCK), D10 (CS). Розміщення цих контактів на роз'ємі CN10 плати NUCLEO-F401RE зображено на рис.1.25.

Для обміну даними з SPI приймачем підключіть згадані вище контакти SPI передавача до контактів SPI приймача на налагоджувальній платі з FPGA. Обов'язково переконайтеся, що контакти GND обох налагоджувальних плат з'єднані між собою.

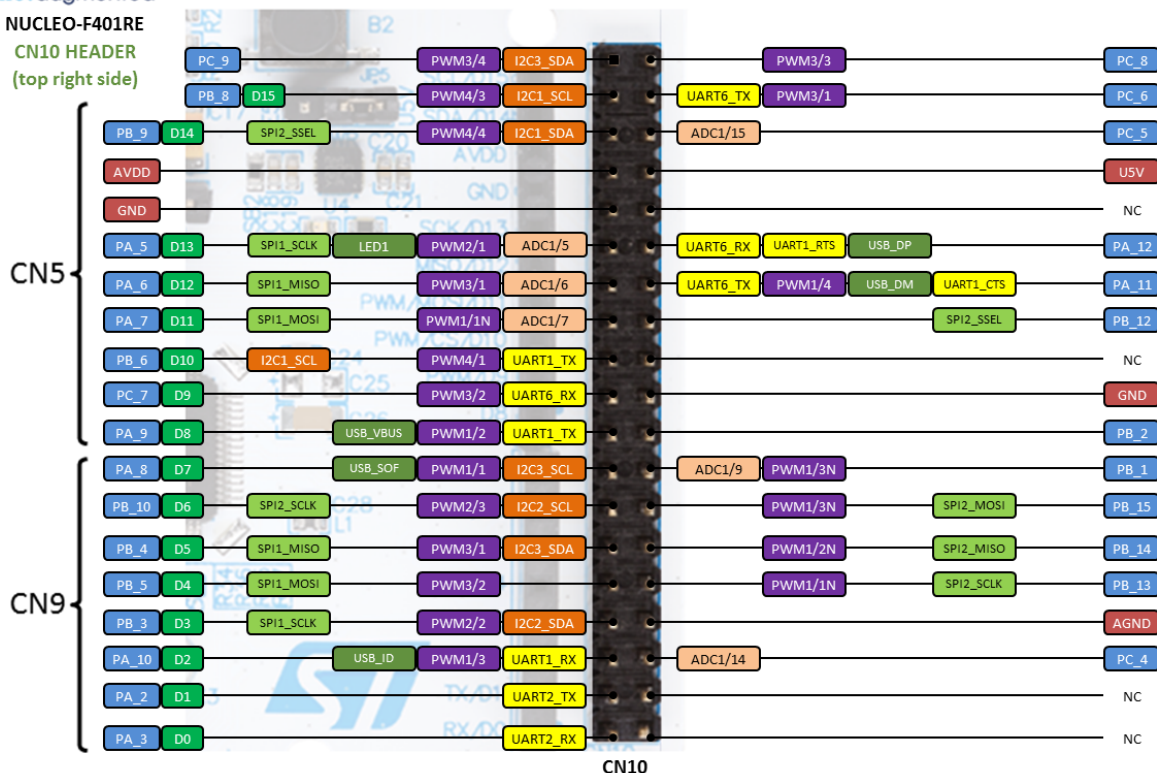


Рис.1.25 -Штир'ювий роз'єм CN10 з цифровими виводами налагоджувальної плати  
NUCLEO-F401RE

### 1.3.2 Передача даних по SPI з використанням Arduino

Для компіляції коду з лістингу 1.13 завантажте і встановіть середовище розробки Arduino [1.8].

Підключіть налагоджувальну плату Arduino Leonardo до ПК через інтерфейс USB.

Запустіть середовище розробки Arduino і відкрийте у ньому скетч `arduino_spi_sender.ino` який можна завантажити за посиланням: [https://github.com/KorotkiyEugene/digital\\_lab/tree/master/Lab1/sw/spi/arduino\\_spi\\_sender](https://github.com/KorotkiyEugene/digital_lab/tree/master/Lab1/sw/spi/arduino_spi_sender)

В меню Tools -> Board оберіть "Arduino Leonardo".

В меню Port оберіть підключену плату.

Для компіляції проекту оберіть пункт меню Sketch -> Verify/Compile

Для завантаження зкомпільованого файлу прошивки проекту в Arduino оберіть пункт меню Sketch -> Upload.

В Arduino Leonardo контакти SPI виведено на окремий роз'єм, зображення якого наведено на рис.1.26



Рис.1.26 - ICSP роз'єм Arduino Leonardo з контактами інтерфейсу SPI

Підключіть контакти MOSI, SCK, GND до відповідних контактів налагоджувальної FPGA плати з SPI приймачем. Також підключіть цифровий контакт №8 Arduino Leonardo (CS) до входу CS SPI приймача.

### 1.3.3 Використання логічного аналізатора SignalTap в Quartus Prime

Логічним аналізатором називають пристрій для спостереження за зміною цифрових сигналів. У мікросхемах компанії Intel FPGA можна реалізувати вбудований логічний аналізатор SignalTap Logic Analyzer, який дозволяє аналізувати зміну цифрових сигналів як всередині FPGA, так і сигналів, що надходять ззовні FPGA мікросхеми.

Принцип роботи логічного аналізатора дуже простий. Значення цифрового сигналу зчитується з високою частотою дискретизації ( $F_s$  - sampling frequency). Частота дискретизації може лежати у межах від кількох мегагерц до кількох гігагерц. Зчитані значення послідовно записуються в пам'ять логічного аналізатора. Важливим моментом є умова початку запису, яку у закордонній літературі називають trigger. Це може бути фронт сигналу (передній, задній, будь яка зміна), або значення, яке приймає багаторозрядна шина. Логічний аналізатор постійно перевіряє чи не відбулася подія trigger. Як тільки ця подія відбудеться, починається зчитування цифрового сигналу і запис зчитаних значень послідовно в пам'ять. Коли тільки пам'ять виділена для певного сигналу повністю заповниться, зчитування цифрового сигналу завершується і вміст пам'яті відображається графічно у вигляді зміни цифрового сигналу у часі.



В SignalTap Logic Analyzer, реалізованому на FPGA мікросхемах сімейств Cyclone V і MAX 10 частота дискретизації може бути до 200 МГц, а в деяких випадках навіть вище.

У логічного аналізатора може бути кілька каналів, що дозволяє спостерігати за зміною кількох сигналів. Для кожного каналу виділяється окремий обсяг пам'яті, який можна задати.

Очевидно, частота дискретизації і обсяг пам'яті каналу логічного аналізатора визначають тривалість цифрового сигналу, яку можна записати за допомогою логічного аналізатора. Ця тривалість розраховується, як кількість комірок пам'яті каналу аналізатора помножена на період частоти дискретизації.

Для активації логічного аналізатора в Quartus Prime перейдіть у вікно налаштувань (Assignments -> Settings) і на вкладці Signal Tap Logic Analyzer поставте галочку “Enable Signal Tap Logic Analyzer”.

Для запуску вікна логічного аналізатора виконайте пункт меню Tools -> Signal Tap Logic Analyzer.

Спершу зробимо деякі налаштування у вкладці Signal Configuration (рис.1.27).

Signal Configuration:

Clock: MAX10\_CLK1\_50

**Data**

Sample depth: 8 K RAM type: Auto

☐ Segmented: 2 4 K sample segments

Nodes Allocated: ☒ Auto ☐ Manual: 0

Pipeline Factor: 0

**Storage qualifier:**

Type: Continuous

Input port:

Nodes Allocated: ☒ Auto ☐ Manual: 0


☒ Record data discontinuities

☐ Disable storage qualifier

Рис.1.27 - Частина налаштувань вкладки Signal Configuration вікна Signal Tap Logic Analyzer

Обсяг пам'яті для кожного сигналу задається у полі Sample Depth. Для початку можна обрати значення порядку кількох тисяч і збільшувати це значення у разі необхідності перегляду тривалих цифрових сигналів.

У полі Clock необхідно вибрати тактовий сигнал дискретизації по передньому фронту якого буде відбуватися зчитування значень цифрових сигналів. Це повинен бути один із тактових сигналів проекту. Якщо брати у якості прикладу проект SPI для налагоджувальної плати DE10\_Lite, то у цьому проекті тактовий сигнал синхронізації представлений входом MAX10\_CLK1\_50 (цей вхідний порт у налаштуваннях пов'язаний з контактом FPGA на який подається тактовий сигнал з частотою 50 МГц). Тому у якості тактового сигналу дискретизації логічного аналізатору для проекту SPI можна обрати MAX10\_CLK1\_50. Для інших налагоджувальних плат назва тактового сигналу дискретизації буде інша (наприклад, CLOCK\_50 для плати DE0-CV). Конкретне ім'я тактового сигналу вхідної частоти можна знайти в документації на налагоджувальну плату і в імпортованих налаштуваннях портів вводу-виводу для обраної плати (див. опис попередньої лабораторної роботи).

Для вибору тактового сигналу дискретизації натисніть кнопку  біля поля Clock на вкладці Signal Configuration. З'явиться вікно вибору сигналів з проекту (рис.1.28).

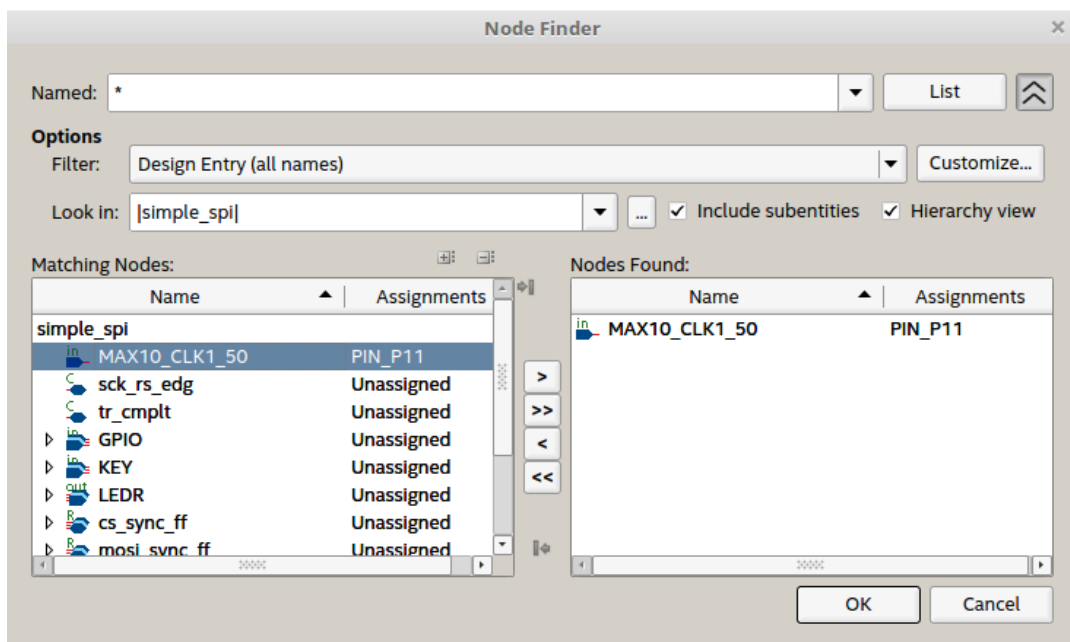



Рис.1.28 - Налаштування імпорту тактового сигналу дискретизації лог. аналізатора  
Signal Tap

В полі Named вікна на рис.1.28 залиште \*, щоб відображалися всі знайдені сигнали. В полі Filter оберіть “Design Entry (all names)”. Модуль всередині якого необхідно шукати сигнали обирається в полі “Look in”. Зазвичай це Top Level Entity. Для пошуку сигналів проекту із заданими налаштуваннями натисніть кнопку List. оберіть сигнал тактової частоти, натисніть кнопку , а потім кнопку OK.

Інші налаштування вкладки Signal Configuration можна залишити за замовчуванням.

Тепер у вкладці Setup необхідно додати сигнали логічного аналізатора (рис.1.29). Спершу ця вкладка прожня. Для додавання нових сигналів двічі клікніть на вільному місці і з'явиться діалог імпорту сигналів з проекту (рис.1.30). Додайте в логічний аналізатор сигнали позначені на рис.29 і рис.30.

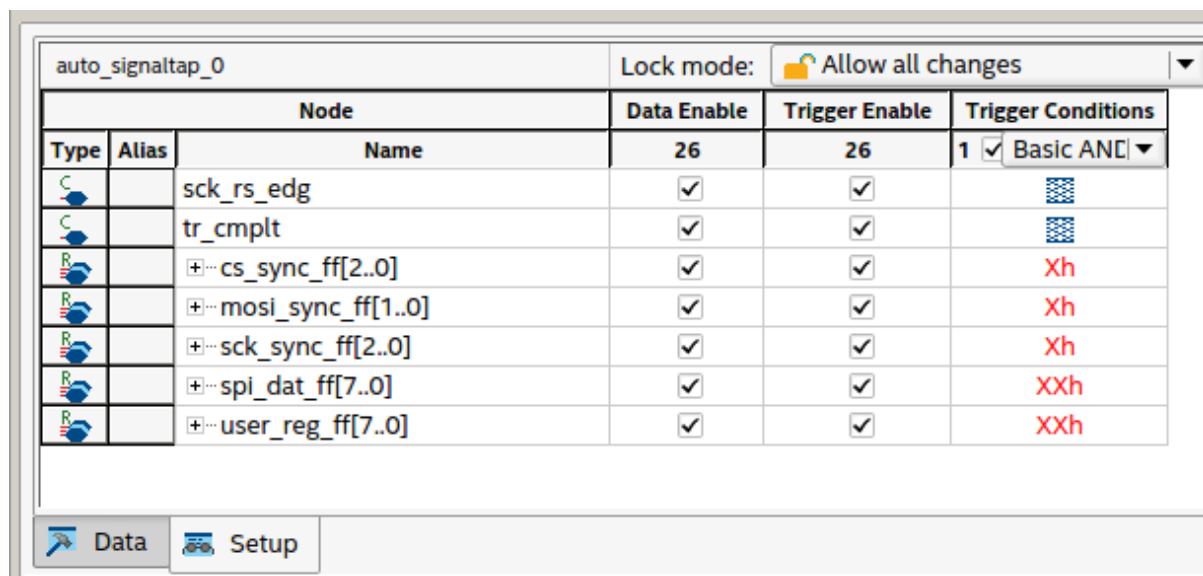


Рис.1.29 - Вкладка вибору сигналів логічного аналізатора SignalTap

Далі необхідно зберегти налаштування логічного аналізатора обравши пункт меню File -> Save. Залиште за замовчуванням ім'я файлу налаштувань лог. аналізатора “stp1.stp”.

Після активації і налаштування логічного аналізатора SignalTap, необхідно перекомпілювати проект, щоб Quartus Prime додав схему логічного аналізатора обраних сигналів в конфігураційний файл FPGA.

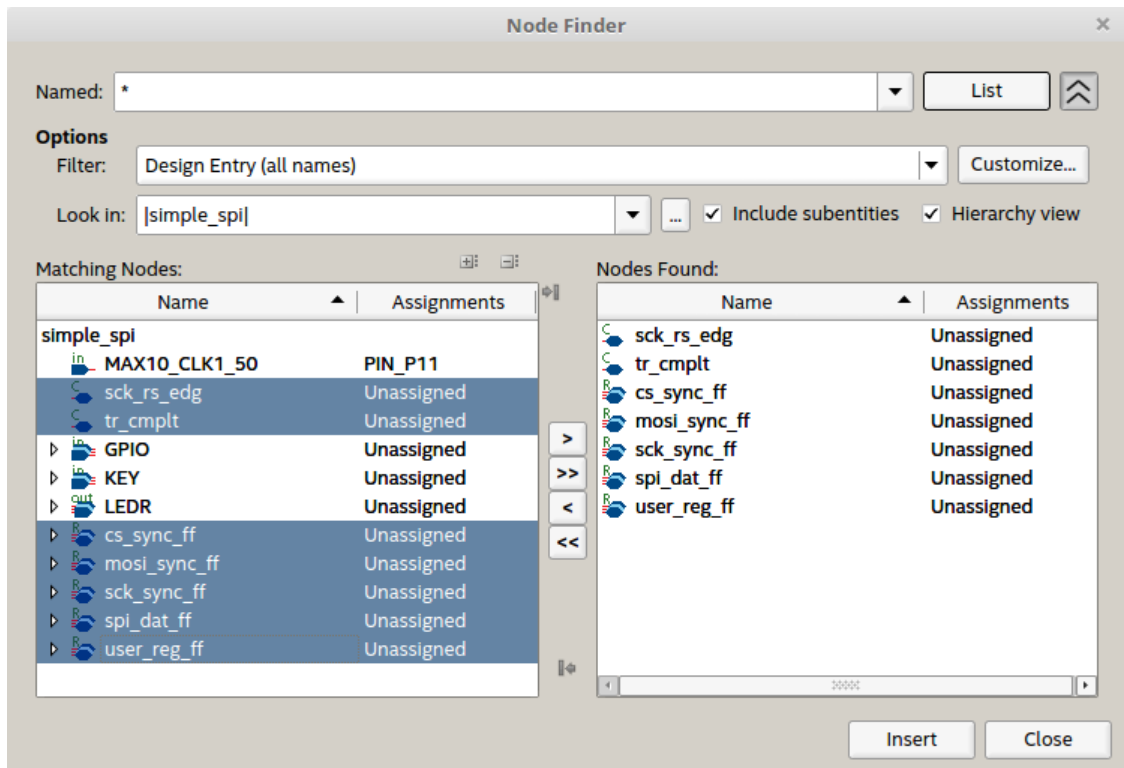



Рис.1.30 - Вікно імпорту сигналів в логічний аналізатор

Після перекомпіляції підключіть налагоджувальну плату з FPGA до комп'ютера за допомогою інтерфейсу USB та завантажте у плату новий конфігураційний файл. У вікні Signal Tap Logic Analyzer на вкладці Setup оберіть програматор USB Blaster та натисніть кнопку Scan Chain.

У вікні сигналів проекту додайте тригер (подію), по якому почнеться зчитування значень обраних сигналів. У якості тригеру підійде передній фронт сигналу sck\_rs\_edg, що є виходом детектору фронту сигналу spi\_sck. Для додавання тригеру клацніть правою клавішею миші в полі "Trigger Conditions" сигналу та оберіть "Rising Edge". Після зміни тригеру немає необхідності заново компілювати проект.

Для початку роботи логічного аналізатора натисніть кнопку  (Run Analysis). Після натиснення кнопки логічний аналізатор буде очікувати на обрану подію (тригер), після настання події по кожному передньому фронту сигналу дискретизації будуть зчитуватися значення обраних сигналів у пам'ять всередині FPGA. Після заповнення пам'яті зібрані дані будуть передані на комп'ютер і відображені у вигляді зміни сигналів у часі (рис.1.31).

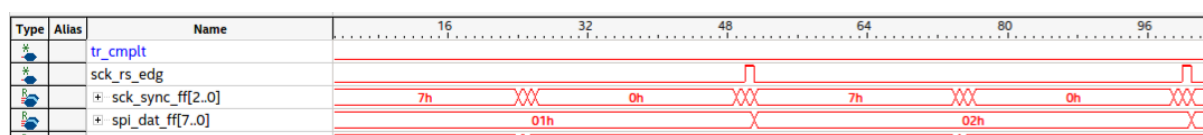


Рис.1.31 - Відображення зміни обраних сигналів в логічному аналізаторі Signal Tap

### 1.3.4 Завдання на лабораторну роботу

Приклади виконання частини завдань можна переглянути за посиланням [\[1.9\]](#).

1. В Quartus Prime створіть проект 8-розрядного регістру зсуву. У якості сигналу тактової частоти використовуйте кнопку KEY[0]. У якості сигналу асинхронного зкидання тригерів регістру використовуйте кнопку KEY[1]. Вміст регістру виведіть на світлодіоди плати. Значення для молодшого розряду регістру зсуву зчитуйте з перемикача SW[0]. Перевірте роботу створеного регістру на одній з налагоджувальних плат.
2. В Quartus Prime створіть проект 8-розрядного регістру зсуву на мові Verilog. Налаштування вхідних/вихідних портів проекту такі ж самі, як для попереднього завдання. Перевірте роботу створеного регістру на одній з налагоджувальних плат.
3. В Quartus Prime створіть проект 8-розрядного регістру зсуву з лінійним зворотнім зв'язком (LFSR). У якості сигналу тактової частоти використовуйте кнопку KEY[0]. У якості сигналу асинхронного зкидання тригерів регістру використовуйте кнопку KEY[1]. Вміст регістру виведіть на світлодіоди плати. Перевірте роботу створеного регістру на одній з налагоджувальних плат.
4. В Quartus Prime створіть проект 8-розрядного регістру зсуву з лінійним зворотнім зв'язком (LFSR) на мові Verilog. Налаштування вхідних/вихідних портів проекту такі ж самі, як для попереднього завдання. Перевірте роботу створеного регістру на одній з налагоджувальних плат.
5. В Quartus Prime створіть проект SPI приймача на мові Verilog (лістинг 1.11). У якості сигналу системної тактової частоти оберіть високочастотний вхідний порт зпоміж імпортованих налаштувань. Зазвичай такий сигнал має частоту

порядку 50 МГц - 100 МГц і ім'я на зразок MAX10\_CLK1\_50 чи CLOCK\_50 (для різних налагоджувальних плат можуть бути різні імена). У якості сигналу асинхронного зкидання використайте кнопку KEY[1]. Вміст регістру даних SPI приймача виведіть на світлодіоди налагоджувальної плати. Для сигналу SPI\_SCK використайте контакт GPIO[0] роз'єму GPIO. Для сигналу SPI\_CS використайте контакт GPIO[1] роз'єму GPIO. Для сигналу SPI\_MOSI використайте контакт GPIO[2] роз'єму GPIO. З використанням налагоджувальних плат Nucleo F401RE або Arduino Leonardo перевірте роботу SPI приймача.

6. Перегляньте зміну внутрішніх сигналів SPI приймача в логічному аналізаторі SignalTap.

#### **1.4 Контрольні запитання**

1. Намалюйте умовне графічне позначення синхронного по фронту D-тригера, опишіть входи/виходи і поясніть логіку його роботи;
2. Поясніть, що таке метастабільний стан D-тригера та які умови входження D-тригера в метастабільний стан;
3. Поясніть недоліки входження D-тригера в метастабільний стан;
4. Поясніть, що таке затримка запису в синхронний по фронту D-тригер. Відповідь проілюструйте часовою діаграмою;
5. Поясніть, що таке затримки  $t_{setup}$  і  $t_{hold}$  для синхронного по фронту D-тригера. Чому важливо витримувати ці затримки? Відповідь проілюструйте часовою діаграмою;
6. Наведіть шаблон опису синхронного по фронту D-тригера на мові Verilog;
7. Намалюйте схему паралельного регістру на базі синхронних по фронту D-тригерів;
8. Напишіть Verilog код для опису паралельного регістру на базі синхронних по фронту D-тригерів;
9. Намалюйте схему регістру зсуву з входом дозволу зсуву на базі синхронних по фронту D-тригерів з входом дозволу запису. Поясніть логіку роботи такої схеми;

10. Напишіть Verilog код для опису регістру зсуву з входом дозволу зсуву;
11. Поясніть логіку моделювання always блоку на мові Verilog;
12. Поясніть логіку моделювання оператора @ на мові Verilog;
13. Поясніть функцію оператора {} на мові Verilog;
14. Намалюйте схему і поясніть призначення регістру зсуву з лінійним зворотнім зв'язком (LFSR);
15. Напишіть Verilog код для опису регістру зсуву з лінійним зворотнім зв'язком (LFSR);
16. Намалюйте схему детектора переднього фронту;
17. Намалюйте схему детектора заднього фронту;
18. Намалюйте схему детектора обох фронтів;
19. Поясніть навіщо необхідна синхронізація асинхронних сигналів на входах цифрової мікросхеми. Намалюйте схему синхронізації однорозрядних сигналів з використанням двох синхронних по фронту D-тригерів;
20. Намалюйте схему підключення SPI Master до SPI slave. Опишіть сигнали SPI інтерфейсу;
21. Намалюйте часову діаграму передачі байту по інтерфейсу SPI;
22. Опишіть принцип роботи SPI приймача.

### 1.5 Перелік посилань

[1.1] “Linear Feedback Shift Registers in Virtex Devices. Application Note XAPP-210”, 2007. [Online]. Available:

[https://www.xilinx.com/support/documentation/application\\_notes/xapp210.pdf](https://www.xilinx.com/support/documentation/application_notes/xapp210.pdf)

[1.2] “Метастабільність тригера і міжтактова синхронізація”, 2015 [Online]. Режим доступу: <https://habr.com/post/254869/>

[1.3] “Схема простого SPI приймача”, 2018 [Online]. Режим доступу: [https://github.com/KorotkiyEugene/digital\\_lab/blob/master/Lab1/hw/spi/simple\\_spi\\_schematic.png](https://github.com/KorotkiyEugene/digital_lab/blob/master/Lab1/hw/spi/simple_spi_schematic.png)

[1.4] “Опис налагоджувальної плати NUCLEO-F401RE”, 2018. [Online]. Available: <https://os.mbed.com/platforms/ST-Nucleo-F401RE/>

[1.5] “Опис класу SPI бібліотеки класів MBED”, 2018. [Online]. Available:

<https://os.mbed.com/handbook/SPI>

[1.6] “Опис налагоджувальної плати Arduino Leonardo”, 2018. [Online]. Available:

[https://www.arduino.cc/en/Main/Arduino\\_BoardLeonardo](https://www.arduino.cc/en/Main/Arduino_BoardLeonardo)

[1.7] “Робота з SPI в Arduino”, 2018. [Online]. Available:

<https://www.arduino.cc/en/Reference/SPI>

[1.8] “Arduino IDE Download Page”, 2018. [Online]. Available:

<https://www.arduino.cc/en/Main/Software>

[1.9] Вихідні коди прикладів до лабораторних робіт, 2018 [Online]. Режим доступу: [https://github.com/KorotkiyEugene/digital\\_lab](https://github.com/KorotkiyEugene/digital_lab)