

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи з дисципліни

«Методи синтезу енергоефективних систем»

за темою:

«Алгоритм Вітербі.

Створення згорткового декодера Вітербі»

для магістрів

спеціальності 123 "Комп'ютерна інженерія"

усіх форм навчання

Методичні вказівки до виконання лабораторної роботи з дисципліни «Методи синтезу енергоефективних систем» за темою «Алгоритм Вітербі. Створення згорткового декодера Вітербі» для магістрів спеціальності 123 "Комп'ютерна інженерія" усіх форм навчання / Укл. І.Я Зеленьова, Т.В. Голуб. – Запоріжжя: НУ«Запорізька політехніка», 2023. – 16 с.

Укладачі:

І.Я Зеленьова, доцент, к.т.н.
Т.В. Голуб, доцент, к.т.н.

Рецензент:

Грушко С.С.

Відповідальний за випуск:

Т.В. Голуб, доцент, к.т.н.

Затверджено
на засіданні кафедри КСМ
Протокол № 8 від 27.04.2023

Рекомендовано до видання
на засіданні НМК КНТ
Протокол № 10 від 24.05.2023

ЗМІСТ

1 Загальні відомості	4
2 Приклад згорткового декодування (2, 1, 4).....	4
2.1 Розробка загорткового кодування.....	4
2.2 Реалізація та тестування схеми декодера (2,1,4).	8
3 Завдання до лабораторної роботи.....	14
4 Зміст звіту по лабораторній роботі.....	15
5 Контрольні питання	15
Перелік рекомендованої літератури	16

Мета роботи: опанувати принципи організації та застосування алгоритму Вітербі, ознайомитися з декодуванням та створити згортковий декодер Вітербі мовою VHDL.

1 ЗАГАЛЬНІ ВІДОМОСТІ

Щоб декодувати згортковий код, можна скористатися алгоритмом Вітербі. З першої лабораторної роботи відомо, що вихідні біти згорткового коду (2,1,4) формувалися на основі попередніх біт. Завдяки цьому для кожної пари згорткового коду (2,1,4) зберігається залежність від попередніх пар біт, що дає змогу коректувати декодовані біти згорткового коду при декодуванні за допомогою метрик.

В загальному вигляді алгоритм декодування виглядає наступним чином.

1. Обираємо n біт згорткового коду (n - кількість вихідних кодованих біт кодера)
2. Розраховуємо відстані Хеммінга для кожного шляху графу станів.
3. Додаємо відстані Хеммінга до попередніх метрик шляхів.
4. Порівнюємо розраховані метрики шляхів які на графі ведуть до одного і того ж вузла.
5. Обираємо найменшу метрику (це і буде метрикою шляху яка за наступної ітерації буде знову складатися з новими відстанями Хеммінга).
6. Повторюємо процес для наступних n біт.
7. Проводимо трасування шляху з найменшою метрикою. Результатом трасування і буде декодована послідовність біт.

2 ПРИКЛАД ЗГОРТКОВОГО ДЕКОДУВАННЯ (2, 1, 4)

2.1 Розробка загорткового кодування

Для пояснення алгоритму декодування зручно одразу перейти до прикладу декодування згорткового коду.

На рисунку 2.1 зображений граф станів згорткового коду (2,1,4).

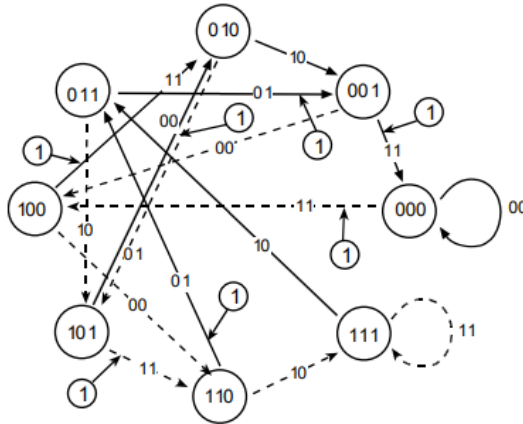


Рисунок 2.1 - Граф станів згорткового коду (2,1,4)

Представимо його, як зображено на рисунку 2.2.

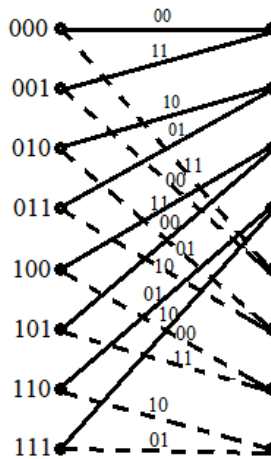


Рисунок 2.2 – Перетворений граф станів згорткового коду (2,1,4)

Почнемо декодування послідовності 11 11 01 11. Спочатку обчислюються відстані Хеммінга для кожної пари біт згорткового коду, як це можна побачити на рисунку 2.3. Розраховуються вони як кількість відмінних біт між вхідною парою біт та парою біт шляху.

Наприклад, для вхідної пари 11: $11 \text{ з } 00 \Rightarrow 2$, $11 \text{ з } 11 \Rightarrow 0$, $11 \text{ з } 10 \Rightarrow 1$, $11 \text{ з } 01 \Rightarrow 1$.

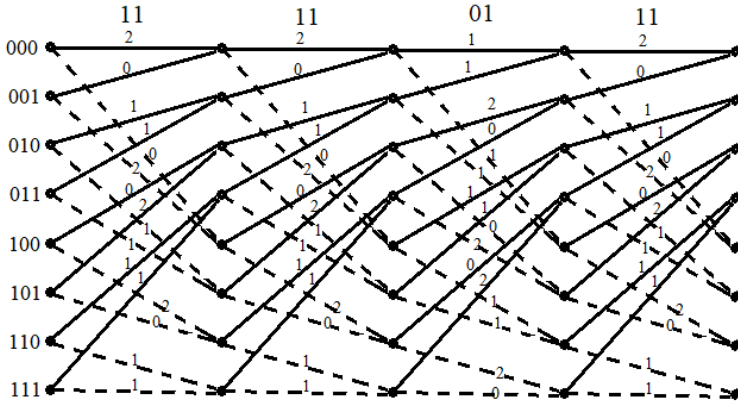


Рисунок 2.3 – Розрахунок відстаней Хеммінга

Далі почергово для кожної пари біт викреслюємо шляхи з більшою мертикою та зберігаємо метрику шляху, що залишився, для подальшого додання до неї обчисленої відстані Хеммінга наступної пари біт згорткового коду.

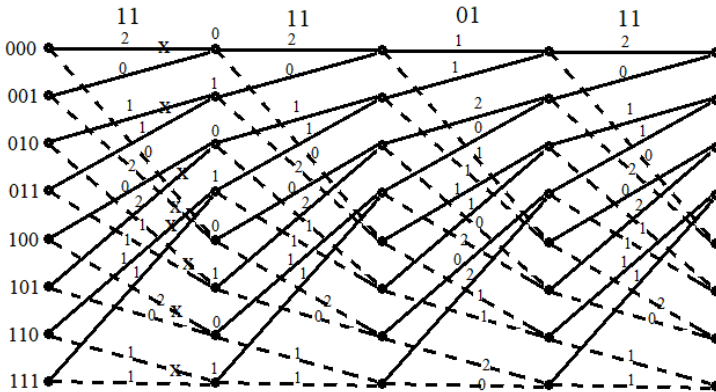


Рисунок 2.4 – Викреслення шляхів для першого етапу

Продовжуємо для кожної пари біт.

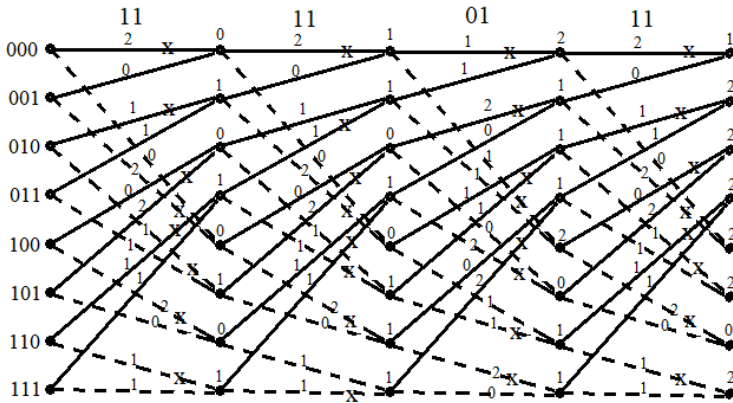


Рисунок 2.5 – Викреслення шляхів для всіх етапів

На рисунку 2.6 зображено фінальний крок, а саме – трасування шляху найменшої метрики. Пам'ятаємо, що пунктирна лінія – це 1, суцільна – 0.

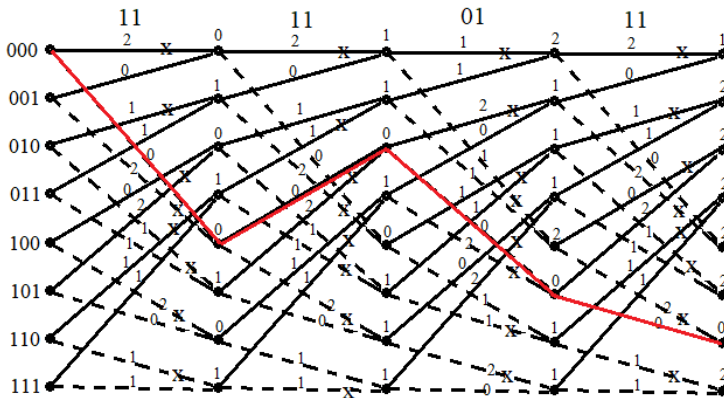


Рисунок 2.6 – Трасування шляху найменшої метрики

Отже декодована послідовність для коду 11 11 01 11 буде 1 0 1 1.

2.2 Реалізація та тестування схеми декодера (2,1,4).

Для програмної реалізації вихідний алгоритм буде дещо відрізнятись. А саме – не буде виконуватися фінальне трасування, адже набори бітів для шляхів, що залишилися, будуть зберігатися в процесі декодування. Фінальним кроком потрібно буде лише обрати з сформованих наборів бітів той набір, який відповідає найменшій метриці. Приклад, зміни наборів декодованих біт для одного етапу можна побачити на рисунку 2.7 (символ & на рисунку означає конкатенцію).

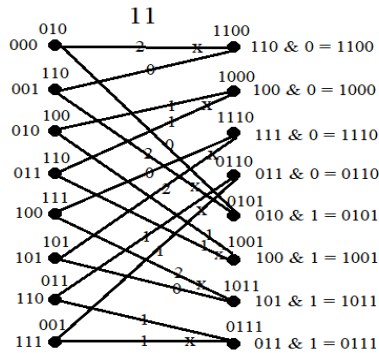


Рисунок 2.7 – Приклад зміни наборів декодованих біт для однієї пари біт згорткового коду

Метрики та набори декодованих біт шляхів будуть зберігатися у масивах з восьми елементів і змінюватися в процесі декодування. VHDL реалізація Декодера наведена у лістингу 2.1.

Лістинг 2.1 – VHDL реалізація Декодера Вітербі

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_unsigned.all;

entity Decoder is
  Port (
    clk : in std_logic;
```

```

        inEncBits    :    in    std_logic_vector(1
downto 0);
        outDecBits   :    out    std_logic_vector(8
downto 0));
    end Decoder;
    architecture Behavioral of Decoder is
        --створення типів масивів для метрик та
        декодованих бітів
        type decBitsArr is array (7 downto 0) of
std_logic_vector(8 downto 0);
        type branchValArr is array (15 downto 0) of
std_logic_vector(1 downto 0);
        type metricsArr is array (7 downto 0) of
std_logic_vector(3 downto 0);
        constant branchVal    :    branchValArr    :=
("10","01","11","00","01","10","00","11","01","10"
,"00","11","10","01","11","00");                --можливі
закодовані біти. Зцих біт обчислюються відстані
Хеммінга
        function                                hammDist(inpEncVal,
branchVal:std_logic_vector(1    downto 0))    return
std_logic_vector is --Функція обчислення відстаней
Хеммінга
            variable hemmDistance : std_logic_vector(1
downto 0);
            begin
                hemmDistance := (NOT(inpEncVal(0) XOR (NOT
branchVal(0)))) & (NOT(inpEncVal(1)    XOR    (NOT
branchVal(1))));
                hemmDistance    :=    ((hemmDistance(0)    AND
hemmDistance(1)) & (NOT(hemmDistance(0) XOR (NOT
hemmDistance(1))));
                return "00" & hemmDistance;
            end hammDist;
            begin
                process(clk)
                    --створення масивів для зберігання метрик
                    та декодованих бітів шляхів

```

```

variable decBits : decBitsArr;
variable metrics : metricsArr :=
("0000","0000","0000","0000","0000","0000","0000",
"0000");
variable tempDecBits : decbitsarr;
variable tempMetrics : metricsArr :=
("0000","0000","0000","0000","0000","0000","0000",
"0000");
variable newDecBit : std_logic;
variable metricIndex,
decBitsAndMetrArrsIndex, branchesValues, i :
integer := 0;
variable bitIndex : integer := 8;
begin
if(clk'event and clk='1')then--по clk
newDecBit := '0';
metricIndex := 0;
decBitsAndMetrArrsIndex := 0;
i := 0;
decodingProcess: while i < 8 loop
-- порівнюємо попередню метрику шляху+
відстань Хеммінга двох шляхів ведучих до поточного
вузла (i вузла)
if((metrics(decBitsAndMetrArrsIndex)
+hammDist(inEncBits,branchVal(branchesValues))) <
(metrics(decBitsAndMetrArrsIndex+1)
+hammDist(inEncBits,branchVal(branchesValues+1))))
then --для першого шляху
tempMetrics(i) :=
metrics(decBitsAndMetrArrsIndex)+hammDist(inEncBits,branchVal(branchesValues)); --формуємо нову
метрику
tempDecBits(i) :=
decBits(decBitsAndMetrArrsIndex);--переносимо у
поточний стан набір декодованих біт обраного шляху
tempDecBits(i)(bitIndex) := newDecBit;--
додаємо біт

```

```

-- якщо другий шлях все ж менша або
дорівнює першій обираємо її
else
    tempMetrics(i) :=
(metrics(decBitsAndMetrArrsIndex+1)+hamDist(inEnc
Bits,branchVal(branchesValues+1)));
    tempDecBits(i) :=
decBits(decBitsAndMetrArrsIndex+1);
    tempDecBits(i)(bitIndex) := newDecBit;
end if;
decBitsAndMetrArrsIndex :=
decBitsAndMetrArrsIndex+2;
if(i=3) then
    decBitsAndMetrArrsIndex := 0;--
обнуляємо для роботи з масивами з п'ятого узла
    newDecBit := '1';--починаючи з п'ятого
стану буде додаватися '1' а не '0'
end if;
i := i+1;
branchesValues := branchesValues+2;
end loop decodingProcess;
bitIndex := bitIndex-1;
decBits := tempDecBits;--з часових масивів
у ті з якими будемо працювати при наступному clk
metrics := tempMetrics;
rightPathDecBitsArrIndexFinder: for i in 0
to 6 loop -- знаходимо індекс найменшої метрики
    if(metrics(i) < metrics(i+1)) then
        metricIndex := i;
    end if;
end loop rightPathDecBitsArrIndexFinder;
--індекс найменшої метрики також є індексом
outDecBits <= decBits(metricIndex); --отже
передаємо його на вихід мікросхеми
end if;
end process;
end Behavioral;

```

Тепер протестуємо створений код. Для цього налаштуємо входні сигнали clk та inpBits на часовій діаграмі (як це робити, пояснювалось у попередній лабораторній роботі) як на рисунку 2.8. Декодуємо послідовність біт без помилок. На рисунку 2.8 наведено результати декодування.

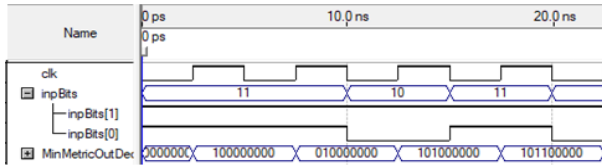


Рисунок 2.8 – Часова діаграма роботи декодеру

На рисунку 2.9 наведена діаграма з всіма декодованими наборами біт та метриками на кожному етапі, завдяки чому можна прослідкувати процес декодування.

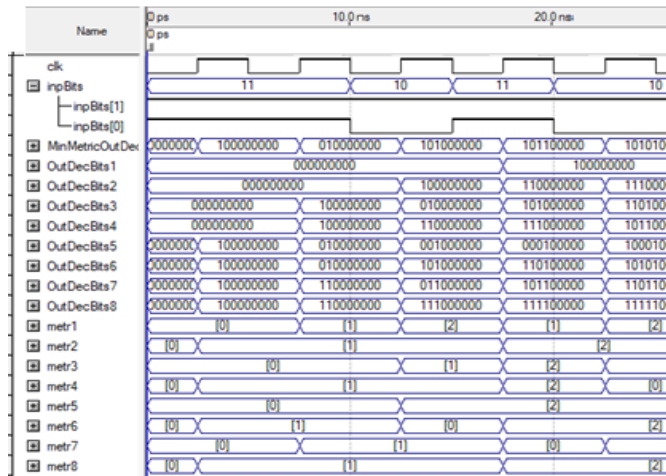


Рисунок 2.9 – Часова діаграма роботи декодеру з метриками та всіма бітами

На рисунку 2.10 показано, що робота часової діаграми декодеру відповідає таблиці дійсності.

Input Bits		Output Bit
O ₁	O ₂	I ₁
0	0	0
1	1	1
1	1	0
0	0	1
1	0	0
0	1	1
0	1	0
1	0	1
0	0	1
0	0	0
1	1	1
0	1	0
1	0	1
0	1	0

Рисунок 2.10 – Відповідність часової діаграми декодера його таблиці дійсності

Тепер декодуємо послідовність біт із помилками. Необхідно згадати, що при кодуванні послідовності 1011 на вхід кодера ще деякий час подаються нулі, тобто 1011000... При цьому вміст регістра все ще не пустий і тому вихідні біти кодера теж будуть не пустими, тобто вихідна послідовність кодера буде 11 11 01 11 01 01 11 00 00 00... Цей “хвіст” потрібен на той випадок, якщо декодер буде працювати більше, ніж чотири такти, і декодер зміг виправити хибні біти у згортковому коді.

Спробуємо декодувати послідовність з помилкою - 11 10 01 11 01 01 11 00 00 00... На рисунку 2.11 наведено часову діаграму роботи декодера з помилкою.

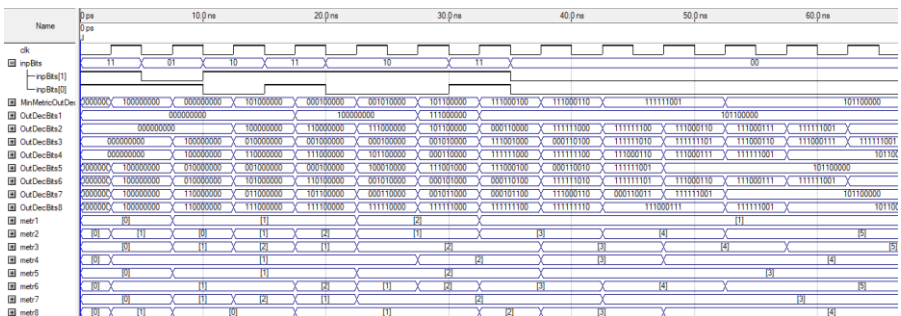


Рисунок 2.11 – Часова діаграма декодера з однією бітовою помилкою

Як можна побачити, коли процес розрахунку метрик закінчується і вони стають постійними, отримуємо декодовану послідовність – 10110000.

Спробуємо декодувати послідовність з двома помилками - 11 10 01 11 01 01 10 00 00 00... На рисунку 2.12 наведено часову діаграму роботи декодера з двома помилками.

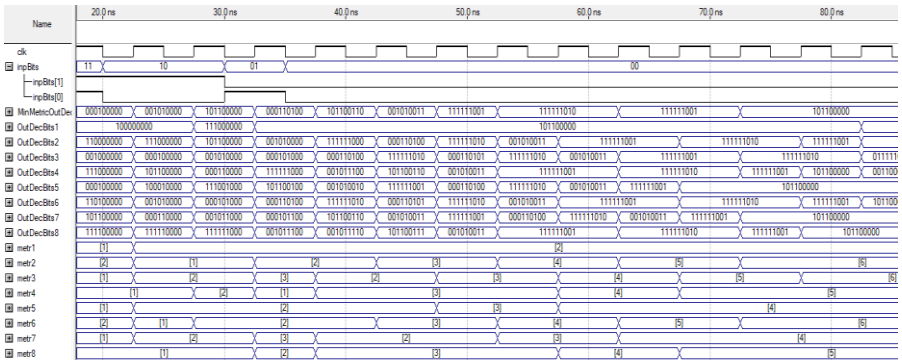


Рисунок 2.12 – Часова діаграма декодера з двома бітовими помилками

Очевидно, що процес декодування подовжився, але зрештою, коли метрики стають постійними, отримаємо ту ж саму декодовану послідовність.

Отже, виходячи з приведених прикладів декодування, можна зробити висновок, що спроектований декодер декодує згортковий код та виправляє помилки у ньому, що доводить коректність його схеми.

3 ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

3.1 Відповідно до обраного варіанта з попередньої лабораторної роботи щодо синтезу згорткового кодера, описати та обґрунтувати в звіті параметри згорткового кодера (n , k , m):

n – кількість вихідних кодованих бітів.

k – кількість вхідних бітів що кодуються.

m – кількість регістрів (тригерів) пам'яті для зберігання попередніх вхідних бітів.

3.2 Виконати синтез та тестування схеми декодера, зробити висновок про коректність роботи схеми. Навести в звіті всі етапи синтезу.

3.3 Проаналізувати звіт з імплементації та навести в звіті скриншот топології схеми декодера на кристалі мікросхеми.

3.4 Дати відповіді на два контрольних питання (на вибір).

4 ЗМІСТ ЗВІТУ ПО ЛАБОРАТОРНІЙ РОБОТІ

Зміст звіту має містити:

- опис всіх етапів розробки декодера;
- фрагмент розробленої схеми декодера;
- результати моделювання програмного опису розробленої схеми на VHDL;
- результати тестування на стенді;
- відповіді на контрольні питання.

Результати тестування отриманої схеми привести в звіті у вигляді часових діаграм, таблиці розпінування та фото стенду з результатами. Зробити висновки по роботі.

5 КОНТРОЛЬНІ ПИТАННЯ

1. На якому принципі заснований алгоритм декодування Вітербі?

2. Які ще алгоритми використовуються для декодування згорткового коду? В чому переваги алгоритму Вітербі?

3. Від чого залежить формування оптимального шляху на діаграмі?

4. Що таке глибина декодування?

5. Чому згортковий код дозволяє коректувати помилки?

ПЕРЕЛІК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Signal processing & simulation newsletter [Електронний ресурс]. : complextoreal.com, 2018. - Режим доступу: <http://complextoreal.com/wp-content/uploads/2018/09/convolutional-codes.pdf>.

2. Проектування комп'ютерних систем на основі мікросхем програмованої логіки [Текст]: монографія. / С. А. Іванець, Ю. О. Зубань, В. В. Казимир, В. В. Литвинов.; Сумський державний університет, Чернігівський державний технологічний університет. – Суми, Чернігів: СДУ, ЧДТУ, 2013. – 313 с.

3. Viterbi algorithm [Електронний ресурс] : wikipedia.org, 2022. - Режим доступу: [https://en.wikipedia.org/wiki/Viterbi_algorithm#:~:text=The%20Viterbi%20algorithm%20is%20a,hidden%20Markov%20models%20\(HMM\)](https://en.wikipedia.org/wiki/Viterbi_algorithm#:~:text=The%20Viterbi%20algorithm%20is%20a,hidden%20Markov%20models%20(HMM)).