**УДК 681.3**

**Т.М. Микитів, Л.В. Мороз**
Національний університет „Львівська політехніка”
кафедра «Безпека інформаційних технологій»
79013, м. Львів, вул. С. Бандери, 12

# МІКРОКОНТРОЛЕРНИЙ СИНУСНО-КОСИНУСНИЙ ОБЧИСЛЮВАЧ ПІДВИЩЕНОЇ ШВИДКОДІЇ



**This paper presents results of various modifications on the microcontroller CORDIC algorithms for calculating trigonometric functions sine and cosine. We describe the general principles of this class of algorithms, and each of such modifications. Each of the methods as implemented in C and in assembler, which allows to evaluate the quality of compiler code generation. The results of numerical simulations, in particular on the number of elementary operations and the accuracy of calculation for the same input data.**

**Keywords - CORDIC, microcontroller, assembler, calculation, comparison, sine, cosine, analysis, methods**

**В роботі подано результати дослідження різних модифікацій CORDIC алгоритмів на мікроконтролері для обчислення тригонометричних функцій синуса та косинуса. Описано загальні принципи роботи алгоритмів цього класу, а також кожну з модифікацій зокрема. Кожен з методів реалізований як на мові С так і на асемблері, що дає можливість оцінити якість генерації коду компілятором. Наведені результати порівняння числового моделювання, зокрема по кількості елементарних операцій та точності обчислення для однакових вхідних даних.**

**Ключові слова – CORDIC, мікроконтролер, асемблер, обчислення, порівняння, синус, косинус, аналіз, методи.**

## Методи прискорення обчислень

Для прискорення програмної реалізації певного алгоритму використовують два підходи. Перший стосується самого алгоритму і полягає у його вдосконаленні чи заміні на більш швидкий, складність якого є меншою у порівнянні з базовим. Таким чином, ми отримаємо вищу швидкодію при однакових умовах програмної реалізації (при використанні однакового компілятора чи мікроконтролера).

Другий підхід базується на вдосконалені програмної реалізації і не стосується самого алгоритму. Його суть полягає в тому, що програміст, знаючи апаратні можливості конкретного процесора чи мікроконтролера пише код таким чином, щоб зменшити загальну кількість операцій і тим самим підвищити швидкодію. Реалізація цього методу, у більшості випадків, полягає у переписуванні найбільш критичних по часу ділянок коду на мову асемблера даного процесора, і вимагає не тільки високого рівня знань та кваліфікації у програміста, а й знання усіх тонкощів роботи алгоритму, що реалізується. Проте, іноді реалізація цього методу недоцільна, оскільки він вимагає значних часових затрат і не завжди приносить бажані результати.

У даній статті будуть наведені результати досліджень різних модифікацій CORDIC методу, які реалізовані на мікроконтролері ATmega16 без математичного препроцесора (FPU), тому доцільно розглядати не тільки принцип роботи алгоритмів, але й специфіку їх програмної реалізації, що має значний вплив на швидкодію.

**Апаратні особливості та програмна реалізація**

У мікроконтролері ATmega16 компанії Atmel, як і у більшості подібних, відсутній математичний препроцесор, і тому немає апаратної підтримки математичних функцій (таких як **sin, cos, log, exp**). Окрім того більшість мікроконтролерів не мають інструкцій для роботи з дійсними типами даних, тому ці операції необхідно реалізувати програмно, що має значний вплив на швидкодію. Враховуючи те, що метою статті є дослідження того, наскільки ефективно можна використати найпростіші мікроконтролери для обчислення математичних функцій синуса та косинуса, необхідно відмовитися від типів даних з плаваючою комою, оскільки більшість мікроконтролерів їх не підтримує. Враховуючи й те, що для обчислення значень функцій синуса та косинуса не потрібен широкий діапазон значень, доцільно використати тип даних з фіксованою комою. Таким чином спроститься програмна реалізація, особливо тоді, коли алгоритм планується написати на асемблері. Перевага полягає в тому, що операції з таким типом даних можна розглядати як операції з цілими типами даних, які підтримуються мікроконтролероми, хоч і побайтно.

Для того, щоб забезпечити 16 біт точності дійсних значень та сумісність з типами даних компілятора мови С, використовується наступний 4 байтовий тип даних:

*Таблиця 1.*

**Структура типу даних з фіксованою комою**

| Знак числа | Мантиса | $2^{-const}$ експонента |
|---|---|---|
| 1 біт | 7 біт | 24 біти |

Таким чином ціла частина може бути в межах $[-128,127]$, а під дійсну частину виділяється 24 біти, що більш ніж достатньо для поставленої задачі.

ATmega16 – це 8 бітний мікроконтролер, що містить 32 регістри (R0 – R31) загального призначення, які напряму підєднані до арифметико-логічного пристрою, тому більшість інструкцій виконується за один такт, що значно підвищує його продуктивність. При реалізації програм на асемблері, для зберігання локальних змінних стек не використовується, що також підвищує швидкодію, оскільки основний обмін даними відбувається між регістрами мікроконтролера.

При написанні програм на мові С під мікроконтролер використовувався компілятор Code VisionAVR 1.25.94. Недолік даного компілятора полягає в недостатньому рівні оптимізації програмного коду. Тому, коли швидкодія програмного забезпечення стоїть на першому місці, доцільно використовувати асемблер.

В нашому випадку, для того щоб проаналізувати, наскільки доцільним буде вдосконалення програмної реалізації алгоритму, необхідно поглянути на отриману скомпільовану програму, а точніше на згенерований асемблерний код, який на пряму перетвориться в машинні інструкції. Тому, на вхід компілятора Code VisionAVR, подається текст програми на мові С, а сам компілятор налаштовуються на максимальний рівень оптимізації по швидкодії, оскільки в даному випадку більш критичною є швидкодія, ніж розмір отриманої програми. Результат роботи компілятора – асемблерний лістинг програми, та бінарний файл програми який можна прошивати у мікроконтролер.

Першим, що кидається в очі при аналізі згенерованого асемблерного лістинга, це дуже часті операції роботи з стеком навіть тоді, коли змінна оголошена зі специфікатором **register**. Для компілятора це виправдано, оскільки по замовчуванню при виклику підпрограм та функцій параметри їм передаються через стек, за рахунок чого є можливими рекурсивні виклики. Також пам'ять під усі локальні змінні по замовчуванню виділяється з стеку. Проте це призводить до значних втрат у швидкодії. У чому ж полягає суть проблеми? Нехай необхідно виконати наступний оператор мови С:

```
x = x + y;          /* тут x та y 16-ти бітні змінні, цілі числа */
```

Порівняємо реалізацію цього фрагменту програми на асемблері з використанням стеку і без нього (регістрова пара R29:R28 – Y – вказує на вершину стеку):

*Таблиця 2.*

**Порівняння ділянок коду з використанням стеку і без нього**

| Використовуючи стек під локальні змінні | | Зберігаючи змінні в регістрах | |
|---|---|---|---|
| Інструкції | Кількість тактів | Інструкції | Кількість тактів |
| LD R30,Y ; завантаження з стеку | 2 | ADD R30,R26 | 1 |
| LDD R31,Y+1 | 2 | ADC R31,R27 | 1 |
| LDD R26,Y+2 | 2 | | |
| LDD R27,Y+2+1 | 2 | | |
| ADD R30,R26 ; знаходження суми | 1 | | |
| ADC R31,R27 | 1 | | |
| STD Y+2,R30 ; збереження результату | 2 | | |
| STD Y+2+1,R31 | 2 | | |
| Всього тактів | 14 | | 2 |

Використовуючи стек, необхідно спочатку завантажити відповідні значення, знайти їх суму та зберегти результат. Як бачимо, на такому простому прикладі, кількість тактів, необхідних для виконання даного фрагменту можна зменшити у 7 разів, якщо обидві змінні зберігати у регістрах загального призначення мікроконтролера. Проте, на такий код накладається ряд обмежень, зокрема, при виконанні рекурсії.

Ще одним недоліком згенерованого лістинга є часті виклики службових підпрограм. Ними можуть бути ділянки коду які, часто виконуються, і щоб зменшити розмір коду, та не дублювати їх кожен раз, ці інструкції виділяють в окрему підпрограму. Хоча такі підпрограми не мають параметрів, все ж на їх виклик та повернення необхідні деякі часові затрати (приблизно 7-8 тактів). Тому необхідно шукати певний компроміс між розмірами програмного коду та його швидкодією.

Також, при написанні програми на мові С з використанням операторів зсуву на довільну кількість розрядів, необхідно пам'ятати, що як і у більшості мікроконтролерів, ATmega16 не має спеціальної інструкції, і насправді ця операція реалізовується програмно у циклі, за одну ітерацію – один двійковий розряд. Тому, при необхідності виконати зсув на більш ніж 8 розрядів (за умови 8-ми бітних регістрів), доцільно спочатку виконати копіювання відразу у цільові регістри, а вже потім застосувати операцію зсуву на кількість розрядів, що залишилися.

**Метод одностороннього повороту**

Метод одностороннього повороту складається з двох етапів [1]. Перші $k = \lfloor m/2 + 0{,}5 \rfloor$ ітерацій виконуються класичним методом за формулами:

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\ z_{i+1} &= z_i - \sigma_i \alpha_i \end{aligned} \qquad (1)$$

де, оператор $\sigma = sign(z_i)$ визначає напрямок повороту вектора, $\alpha_i = \arctan 2^{-i}, i = 0,1,2...k$ - таблиця кутів елементарних поворотів, значення яких є наперед порахованими та збереженими в пам'яті мікроконтролера.

При цьому вхідні дані задаються наступним чином:

$$x_0 = 1/K_m,\ \ y_0 = 0,\ \ z_0 = \varphi \qquad (2)$$

тут $K_m \approx 1{,}647$ коефіцієнт деформації вектора після виконання $m$ ітерацій, $\varphi$ - вхідний кут. Таким чином отриманий результат не потребує масштабування оскільки коефіцієнт деформації вектора врахований у вхідних даних.

Після виконання першого етапу, отримаємо наближене значення (перші $k = \lfloor m/2 + 0{,}5 \rfloor$ біт) результату.

Особливість другого етапу в тому, що після виконання половини ітерації, прирости значення коефіцієнта деформації вектора виходять за межі розрядної сітки і тому їх можна не враховувати. Таким чином можна пропустити частину поворотів, головне, щоб в результаті вектор опинився на

потрібному місці, і це не вплине на результати обчислень. Тобто, можна один раз вибрати напрямок повороту, а ітерації виконувати так, щоб наближатися до заданого кута тільки з однієї сторони, не змінюючи напрямок повороту.

Наведемо таблицю кутів елементарних поворотів $\alpha_i$, взяту з [5]:

*Таблиця 3.*

**Таблиця кутів елементарних поворотів**

| $i$ | $\alpha_i$ | $\tan\alpha_i$ | $\alpha_i$, 22 бітний бінарний вигляд | $\tan\alpha_i$, 22 бітний бінарний вигляд |
|---|---|---|---|---|
| 0 | 0,500000 | 0,546302 | 0100000000000000000000 | 0100010111101101001111 |
| 1 | 0,250000 | 0,255342 | 0010000000000000000000 | 0010000010101111000010 |
| 2 | 0,125000 | 0,125655 | 0001000000000000000000 | 0001000000010101011101 |
| 3 | 0,062500 | 0,062582 | 0000100000000000000000 | 0000100000000010101010 |
| 4 | 0,031250 | 0,031260 | 0000010000000000000000 | 0000010000000000010101 |
| 5 | 0,015625 | 0,015626 | 0000001000000000000000 | 0000001000000000000010 |
| 6 | 0,007812 | 0,007813 | 0000000100000000000000 | 0000000100000000000000 |
| 7 | 0,003806 | 0,003906 | 0000000010000000000000 | 0000000010000000000000 |
| 8 | 0,001953 | 0,001953 | 0000000001000000000000 | 0000000001000000000000 |

Як бачимо, починаючи з кроку $i = 6$ можна відмовитися від табличних значень арктангенса, оскільки в цьому випадку:

$$\arctan 2^{-i} \approx 2^{-i} \tag{3}$$

Відмова від табличних значень зекономить не тільки інструкції, необхідні для доступу до них, а й пам'ять для зберігання цих значень.

Отож, на другому етапі напрямок повороту вектора $\sigma_k = sign(z_k)$ визначається всього один раз, а ітераційні формули набувають вигляду:

$$\begin{aligned} x_{i+1} &= x_i - \sigma_k \cdot y_i 2^{-i} \\ y_{i+1} &= y_i + \sigma_k \cdot x_i 2^{-i} \\ z_{i+1} &= z_i - \sigma_k \cdot 2^{-i} \end{aligned} \tag{4}$$

при чому ітерації проводяться тільки тоді, коли виконується умова:

$$\left|z_i\right| \le 2^{-i} \tag{5}$$

Ця умова гарантує здійснення повороту в одному напрямку.

**Метод залишкового множення**

Як і в попередньому випадку метод складається з двох етапів [1]. На першому етапі проводяться $k = \lfloor m/2 + 0{,}5 \rfloor$ ітерацій, використовуючи класичний алгоритм (1) та вхідні дані (2).

Результат роботи другого етапу зводиться до виконання однієї ітерації за формулою:

$$\begin{aligned} x_{k+1} &= x_k - \sigma_k y_k \cdot z_k \\ y_{k+1} &= y_k + \sigma_k x_k \cdot z_k \end{aligned} \tag{6}$$

де кут $z_k = a_k 2^{-k} + a_{k+1} 2^{-(k+1)} + \ldots + a_m 2^{-m}$, $a_i = \{0,1\}$ – відповідний біт $z_k$.

Тут $x_k, y_k, z_k$ – результати роботи першого етапу, $\sigma_k$ – напрямок повороту вектора на другому етапі, який визначається тільки один раз – на початку етапу. Формула (6) справедлива лише в тому випадку, коли виконується умова (5), для $i > k$.

Слід зазначити, що тут, на другому етапі, як і у методі одностороннього повороту, значення приростів коефіцієнту деформації вектора виходять за межі розрядної сітки, а тому не впливають на результат обчислень.

У формулі (6) присутня операція множення, тому під час реалізації даного методу можливі два варіанти. У першому необхідно застосувати апаратну операцію множення. За умови використання типу даних представленого у *табл. 1*, достатньо операції множення цілих чисел.

Альтернативним варіантом реалізації є розклад операції множення на серію операцій зсуву та додавання [1], хоча це призведе до зменшення швидкодії, але дасть можливість використовувати алгоритм на будь-якому мікроконтролері.

**Порівняння на мікроконтролері**

При реалізації алгоритмів за апаратну частину взято мікроконтролер ATmega16 компанії Atmel, в якому відсутній математичний препроцесор (FPU). Він містить 16 кБайт флеш пам'яті програм, що цілком достатньо для реалізації кожного з методів, як на асемблері так і на мові С (компілятор Code VisionAVR 1.25.94), та створення тестової програми яка керується з персонального комп'ютера через інтерфейс RS232 (COM порт).

Тестова програма, яка працює на мікроконтролері, прослуховує COM порт та відповідає на запити керуючої програми з комп'ютера. Такими запитами можуть бути: видача списку функцій що підтримуються, задання параметрів табуляції та отримання результату обчислення функції. Таким чином, процес дослідження повністю керується з комп'ютера за допомогою спеціального програмного забезпечення.

Для отримання кількості тактів виконання певного алгоритму, використовується шістнадцятирозрядний лічильник, який запускається перед проведенням обчислення. Після відпрацювання підпрограми одержане значення лічильника та прапорець переповнення повертається керуючій програмі на ПК.

Отож, для оцінки швидкодії та порівняння, було реалізовано наступні методи обчислення синуса та косинуса, як на мові С так і на асемблері:

а) класичний, за формулою (1);

б) метод одностороннього повороту;

в) метод залишкового множення.

Слід зазначити, що кожен з методів використовує тільки прості операції, такі як додавання, віднімання, арифметичний та логічний зсув, що дає можливість їх використання на будь-якому мікроконтролері.

При табуляції функції $\sin(x)$ на відрізку $[-\pi/2;\pi/2]$ було отримано наступні результати:

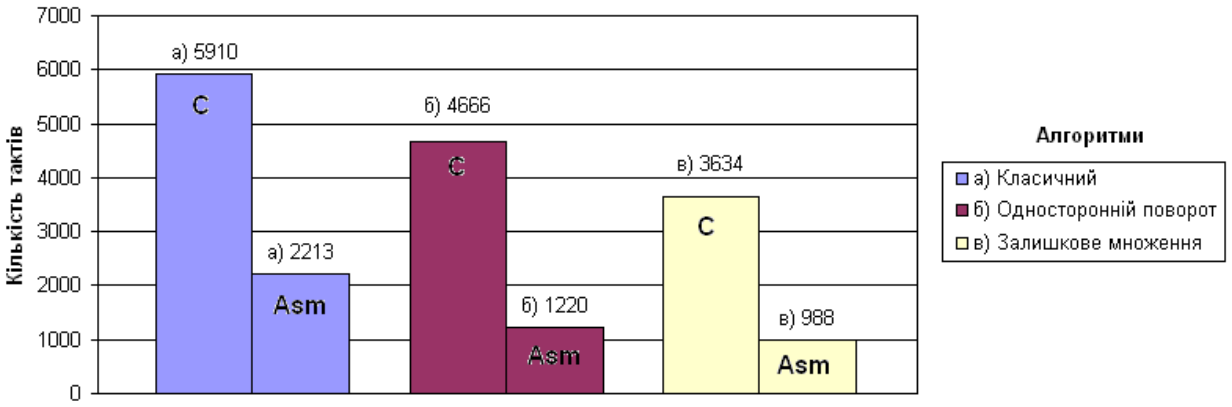


*Рис. 1. Результати порівняння реалізації різних методів обчислення sin(x) на мові С (з ліва) та асемблері (з права) по кількості тактів*

При цьому модулі абсолютних похибок, як алгоритмів на мові С так і на асемблері, складають:


*Рис.2. Модуль похибки обчислення функції sin(x)*

**Висновок**

Написання критичних по часу ділянок коду, під мікро контролер, необхідно виконувати на асемблері, оскільки він надає повний контроль над програмною реалізацією на найнижчому рівні, та за умілого використання досягається значний виграш у швидкодії. Як бачимо, асемблерні реалізації методів, виграють по швидкодії у 2-4 рази, порівняно з тими, що написані на мові С з використанням даного компілятора.

**Список використаних джерел**


*1. Мороз Л.В., Микитів Т.М. Дослідження модифікацій синусно-косинусних CORDIC алгоритмів на мікроконтролері AVR // Матеріали IV Міжнародної конференції молодих вчених CSE-2010, 2010 - ст. 218-219*

*2. Andraka R. A survey of CORDIC algorithms for FPGA based computers // ACM/SIGDA 6th International Symposium on FPGAs, 1998, c. 1981–2000.*

*3. Lachowicz S., Pfleiderer H.-J. Fast evaluation of nonlinear functions using FPGAs // Advances in Radio Science 6, 2008,pp. 233–237.*

*4. Marino M. Implementazione di funzioni trigonometriche su microcontrollori a 8 bit mediante CORDIC. Giugno 2009. http://www.mmetft.it.*

*5. Madisetti A., Kwentus A.Y. A 100 MHz, 16-b, Direct Digital Frequency Synthesier with 100-dBc Supurious-Free Dynamic Range // IEEE Journal, 1999*