Міністерство освіти і науки України

Національний технічний університет України

«Київський Політехнічний Інститут імені Ігоря Сікорського»

Кафедра конструювання електронно-обчислювальної апаратури

Звіт

З лабораторної роботи №1

по курсу «Обчислювальні та МП засоби в РЕА-2»

Виконав:

cтудент III курсу

групи ДК-81

Шунь П.О.

Перевірив:

доц. Корнєв В.П.

cт. в. Бондаренко Н.О.

Київ – 2021

**Завдання (варіант 10)**

Створити проект у IDE Keil μVision5 та програму, що виконує:

Арифметичний розрахунок виразу:

1. (X + Y) + (Y - Z) - (X - Y) = Q   ;X = 20h, Y = 05h, Z= 04h;

Обчислення логічного виразу:

2. Q = !A⋅B⋅!C + (!B⋅C⋅D ⊕ A⋅C) , де A=50h, B=01h, C=33h, D=E2h.

**Хід роботи**

Мета:

Дізнатися:

Структуру програми для мікроконтролера STM32. Карту пам’яті мікроконтролера STM32 і розміщення частин програми. Основні директиви і команди мови Асемблер для STM32 (Сortex-M4). Призначення і основні можливості IDE Keil µVision5 ̶ інтегрованого середовища для створення програм для STM32 з комплекту інструментальних засобів розробки програм KeilTM Microcontroller Development Kit for ARM (MDK-ARM). Структуру і призначення складових проекту у IDE µVision5. Технологію створення і налагодження програми на асемблері і С для мікроконтролера STM32.

Навчитися:

Налаштовувати IDE Keil µVision5. Створювати проект у Keil µVision5 для мікроконтролера STM32. Створювати програму на асемблері для мікроконтролера STM32 і налагоджувати її у режимі емуляції і безпосередньо на мікроконтролері у складі оціночних модулів STM32F401 NUCLEO та (або) STM32F407 DISCOVERY, що є у складі лабораторного стенду Globallogic Starter Kit. Оцінювати часові параметри програми за даними IDE µVision5.

**Структура програми**

Розглянемо структуру написаної програми із одночасними поясненнями коду та теоретичних відомостей. На рис. 1 зображена блок-схема програми. Розроблена програма має послідовну структуру з «пустим» безкінечним циклом наприкінці.

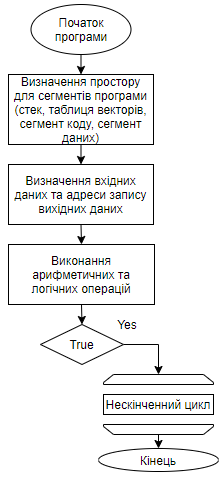


Рис. 1. Блок-схема програми

Розглянемо першу частину програми або так звану «преамбулу», яка подібна у кожній програмі на асемблері для STM32:

1. ; Initialize Stack Size
2. ; Equ 400 hex (1024) bytes
3. Stack\_Size EQU 0x00000400
4. ; Area STACK (NOINIT and READ/WRITE)
5. AREA STACK**,** NOINIT**,** READWRITE**,** ALIGN**=**3
6. ; Initialize memory equals Stack\_Size
7. Stack\_Mem SPACE Stack\_Size
8. \_\_initial\_sp
9. ; Vector Table Mapped to Address 0 at Reset
10. AREA RESET**,** DATA**,** **READONLY**
11. EXPORT \_\_Vectors
12. EXPORT Reset\_Handler **[**WEAK**]**
13. \_\_Vectors
14. DCD \_\_initial\_sp ; Top of Stack
15. DCD Reset\_Handler ; Reset Handler
16. \_\_Vectors\_End
17. AREA MYDATA **,** DATA**,** READWRITE**,** ALIGN**=**3
18. QA DCD 0
19. QL DCD 0
21. AREA MYCODE **,** CODE**,** **READONLY**
22. ; X = 20h, Y = 05h, Z= 04h
23. X EQU 0x04
24. Y EQU 0x35
25. Z EQU 0x08
27. ; Define Constant Data in Code Area
28. X\_\_ DCD 0x04
29. Y\_\_ DCD 0x35
30. Z\_\_ DCD 0x08
32. A\_\_ DCD 0xBF
33. B\_\_ DCD 0xA1
34. C\_\_ DCD 0x35
35. D\_\_ DCD 0x31
36. Reset\_Handler PROC
37. LDR R0**,** **=**\_\_main
38. **BX** R0
39. ENDP

Розглянемо по черзі команди:

Stack\_Size EQU 0x00000400 Присвоєння константі Stack\_Size значення 400h (1 кБ)

AREA STACK**,** NOINIT**,** READWRITE**,** ALIGN**=**3

Визначимо сегмент з ім’ям STACK. Будь-яка програма для мікроконтролера STM32F401RE складається з декількох ділянок пам’яті трьох типів:  
 - стеку STACK  
 - даних DATA  
 - коду CODE,   
які відрізняються їх іменами і властивостями. Властивості задаються параметрами READWRITE (параметри доступу: читання, запис, читання-запис) та ALIGN=3 – вирівнювання даних на границі 4 байти. NOINIT – вміст стеку не буде ініціалізовано при запуску.

Далі командою Stack\_Mem SPACE Stack\_Size виділяється простір в 1 кБ під стек.

\_\_initial\_sp - завдаємо мітку наприкінці стеку.

Наступний сегмент коду:

1. AREA RESET**,** DATA**,** **READONLY**
2. EXPORT \_\_Vectors
3. EXPORT Reset\_Handler **[**WEAK**]**
4. \_\_Vectors
5. DCD \_\_initial\_sp ; Top of Stack
6. DCD Reset\_Handler ; Reset Handler
7. \_\_Vectors\_End

Визначаємо сегмент типу DATA**,** **READONLY** з зарезервованим ім’ям RESET, що розташовується з самого початку пам’яті програм (з адреси 0x08000000).

У цьому сегменті розміщені 4-байтні елементи так званої таблиці векторів переривань (Vector table), початок і кінець якої позначається зарезервованими іменами \_\_Vectors і \_\_Vectors\_End (вони починаються з подвійного підкреслення).

Будь-яка програма повинна мати принаймі хоча б два елементи цієї таблиці: перший повинен містити адресу верхівки стеку із зарезервованим ім’ям \_\_initial\_sp, а другий ̶ адресу початкової підпрограми Reset\_Handler, з якої управління передається основній процедурі, що зазвичай зветься **main**.   
Ці елементи завдаються директивами асемблера DCD (Define Constant Data).

Команди EXPORT \_\_Vectors та EXPORT Reset\_Handler **[**WEAK**]**

визначають зовнішні імена, такі, що будуть доступні іншим програмам. **[**WEAK**]** вказує на те, що це ім’я можна перевизначити в іншому місці.

Розглянемо код, що залишився:

1. AREA MYDATA **,** DATA**,** READWRITE**,** ALIGN**=**3
2. QA DCD 0
3. QL DCD 0
5. AREA MYCODE **,** CODE**,** **READONLY**
6. ; X = 20h, Y = 05h, Z= 04h
7. X EQU 0x04
8. Y EQU 0x35
9. Z EQU 0x08
11. ; Define Constant Data in Code Area
12. X\_\_ DCD 0x04
13. Y\_\_ DCD 0x35
14. Z\_\_ DCD 0x08
16. A\_\_ DCD 0xBF
17. B\_\_ DCD 0xA1
18. C\_\_ DCD 0x35
19. D\_\_ DCD 0x31
20. Reset\_Handler PROC
21. LDR R0**,** **=**\_\_main
22. **BX** R0
23. ENDP

У строці 20 визначаємо сегмент даних MYDATA. Далі визначаємо дві 32-х бітні змінні QA і QL, ініціалізовані нулем. У них буде зберігатися результат виконання першого та другого завдання відповідно.

У 24 строці визначаємо сегмент коду MYCODE. Далі визначаються константи так, аби показати варіативність їх задання та доступу до них.

Директивою DCD можна записати подвійне слово (32 біти) в пам’ять і завдати цієї конструкції з чотирьох 1-байтових комірок ім’я, яке буде вказувати на комірку з меншою адресою (це відповідатиме способу розміщення багаторозрядних даних little endian, який прийнято у контролерах на основі Cortex-M за замовчуванням).

При цьому ми можемо визначити:

− або константу, як що задамо її ім’я у сегменті коду чи у ділянці пам’яті даних типу READONLY;

− або змінну, якщо зробимо це у пам’яті даних типу READWRITE

У 41 строці описано початкову процедуру Reset\_Handler, з якої починається виконання коду після скидання або увімкненні мікроконтролера. Командою LDR (Load register) в регістр R0 завантажується адреса основної процедури. Після чого відбувається перехід по адресі, яка визначена регістром R0 командою **BX**.

**Визначення констант і змінних та способи доступу до них**

RISC-мікроконтролер виконує всю обробку даних лише у регістрах. Дані потрапляють у регістри як константи, задані безпосередньо (вказані у команді оператором #imm), або завантажуються у регістр, як вміст комірок пам’яті, до яких звертається спеціальна команда завантаження регістру ldr (load register).

Якщо значення задане безпосередньо, то завантаження константи у регістр здійснюється командами переміщення mov (move) , або mvn (move not). Також можливо спочатку за допомогою директиви EQU присвоїти числовій константі деяке ім’я, а потім завантажити її значення у регістри командою mov.

У наступному уривку коду описаний початок основної процедури. У регістри r0-r3 завантажуються три рази одні і ті ж дані різними способами.

1. \_\_main PROC
2. ; load immediate data
3. **mov** r0**,** #0x04
4. **mov** r1**,** #0x35
5. **mov** r2**,** #0x08
6. ; load immediate (synonym Constant)
7. **mov** r0**,** #X
8. **mov** r1**,** #Y
9. **mov** r2**,** #Z
10. ; load Constant of code memory
11. ldr r3**,** **=**X\_\_
12. ldr r0**,[**r3**]** ; r0 = X
13. ldr r3**,** **=**Y\_\_
14. ldr r1**,[**r3**]** ; r1 = Y
15. ldr r3**,** **=**Z\_\_
16. ldr r2**,[**r3**]** ; r2 = Z
18. ldr r3**,** **=**A\_\_
19. ldr r5**,** **[**r3**]** ;r5 = A
20. ldr r3**,** **=**B\_\_
21. ldr r6**,** **[**r3**]** ;r6 = B
22. ldr r3**,** **=**C\_\_
23. ldr r7**,** **[**r3**]** ;r7 = C
24. ldr r3**,** **=**D\_\_
25. ldr **r8,** **[**r3**]** ;r8 = D

Команди рядків 49-51 завантажують в регістри значення констант безпосередньо. Команди рядків 53-55 завантажують в регістри значення констант, звертаючись до них по імені. Команди рядків 57-71 завантажують в регістри значення констант, зчитуючи їх з пам’яті. А саме: ldr r3**,** **=**X\_\_ завантажує в регістр r3 адресу константи X\_\_. Тоді команда ldr r0**,[**r3**]** завантажує в регістр r0 значення, яке лежить по адресу, який знаходиться в регістрі r3. Аналогічно для наступних команд.

**Перевірка виконання завдань та пояснення коду програми**

1. ; (Y + Z) + X - (Y - X) + Z = Q ;X = 04h, Y = 35h, Z = 08h;
2. ;and r3, 0
3. **add** r3**,** r1**,** r2 ;r3 = (Y + Z)
4. **add** r3**,** r3**,** r0 ;r3 = (Y + Z) + X
5. **sub** r4**,** r1**,** r0 ;r4 = (Y - X)
6. **sub** r3**,** r3**,** r4 ;r3 = (Y + Z) + X - (Y - X)
7. **add** r3**,** r3**,** r2 ;r3 = (Y + Z) + X - (Y - X) + Z
8. ; Q = !(A and !B and C) + D and !C and B xor !(A and C and D), A = BFh, B = A1, C = 35h, D = 31h
9. mvn **r9,** r6 ;r9 = !B
10. **and** **r10,** r5**,** **r9** ;r10 = r5 and r9 = A and !B
11. **and** **r10,** **r10,** r7; r10 = r10 and r7 = A and !B and C
12. mvn **r10,** **r10**; r10 = !r10 = !(A and !B and C)
13. mvn **r9,** r7; r9 = !r7 = !C
14. **and** **r9,** **r8,** **r9** ;r9 = r8 and r9 = D and !C
15. **and** **r9,** **r9,** r6; r9 = D and !C and B
16. orr **r9,** **r9,** **r10** ;r9 = !(A and !B and C) + D and !C and B
17. **and** **r10,** r5**,** r7 ;r10 = A and C
18. **and** **r10,** **r10,** **r8** ;r10 = A and C and D
19. mvn **r10,** **r10** ;r10 = !r10 = !(A and C and D)
20. eor **r10,** **r10,** **r9** ;r10 = !(A and !B and C) + D and !C and B xor !(A and C and D)
22. ;Save results
23. LDR r0**,** **=**QA
24. **STR** r3**,** **[**r0**]**
25. LDR r0**,** **=**QL
26. **STR** **r10,** **[**r0**]**
27. \_\_mainloop
28. B \_\_mainloop
29. ENDP
30. END

Із 75-ї строки по 92-гу відбувається обрахунок виразів 1-го та 2-го завдання. Строки 95-98 виконують зберігання результату в пам’ять. Тут LDR r0**,** **=**QA – завантаження адреси змінної QA в регістр r0. **STR** r3**,** **[**r0**]** – завантажує вміст регістру r3 (із результатом виконання 1-го завдання) по адресу, який записаний в регістрі r0 (store register). Аналогічно для двох наступних команд.

Далі бачимо нескінченний цикл. Програма переходить на одну і ту ж мітку за допомогою команди B – безумовний перехід до адреси, що визначена міткою.

Визначимо, які повинні бути результати виконання обох завдань:

Завдання 1:

(Y + Z) + X - (Y - X) + Z = Q;

X = 04h = 0000\_0100b,

Y = 35h = 0011\_0101b,

Z = 08h = 0000\_1000b.

Y + Z =

0011\_0101b +

0000\_1000b =

0011\_1101b.

Y – X =

0011\_0101b -

0000\_0100b =

0011\_0001b.

(Y + Z) + X =

0011\_1101b +

0000\_0100b =

0100\_0001b.

(Y + Z) + X - (Y - X) =

0100\_0001b –

0011\_0001b =

0001\_0000b.

(Y + Z) + X - (Y - X) + Z =

0001\_0000b +

0000\_1000b =

0001\_1000b = 18h = Q

Завдання 2:

Q = !(A and !B and C) + D and !C and B xor !(A and C and D);

A = BFh = 1011\_1111b,

B = A1 = 1010\_0001b,

C = 35h = 0011\_0101b,

D = 31h = 0011\_0001b.

A and !B and C =

1011\_1111b &

0101\_1110b &

0011\_0101b =

0001\_0100b.

!(A and !B and C) = 1110\_1011b.

A and C and D =

1011\_1111b &

0011\_0101b &

0011\_0001b =

0011\_0001b.

!(A and C and D) = 1100\_1110b.

D and !C and B =

0011\_0001b &

1100\_1010b &

1010\_0001b =

0000\_0000b.

D and !C and B = 0000\_0000b.

!(A and !B and C) + D and !C and B =

1110\_1011b +

0000\_0000b =

1110\_1011b.

!(A and !B and C) + D and !C and B xor !(A and C and D) =

1110\_1011b xor

1100\_1110b =

0010\_0101b = 25h = Q

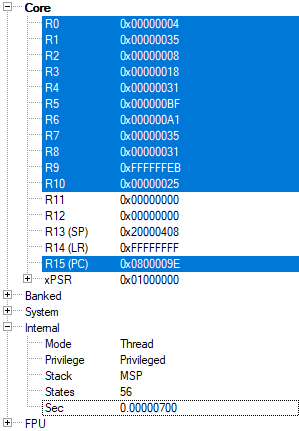


Рис. 2. Стан прапорців у результаті виконання програми

У результаті виконання програми бачимо, що обрахунки виконано вірно (результати було збережено у регістри R3 та R10), рис. 2.

**Дослідження виконання програми у Keil μVision5**

На рис. 3 зображений стан регістрів на початку викнонання програми.

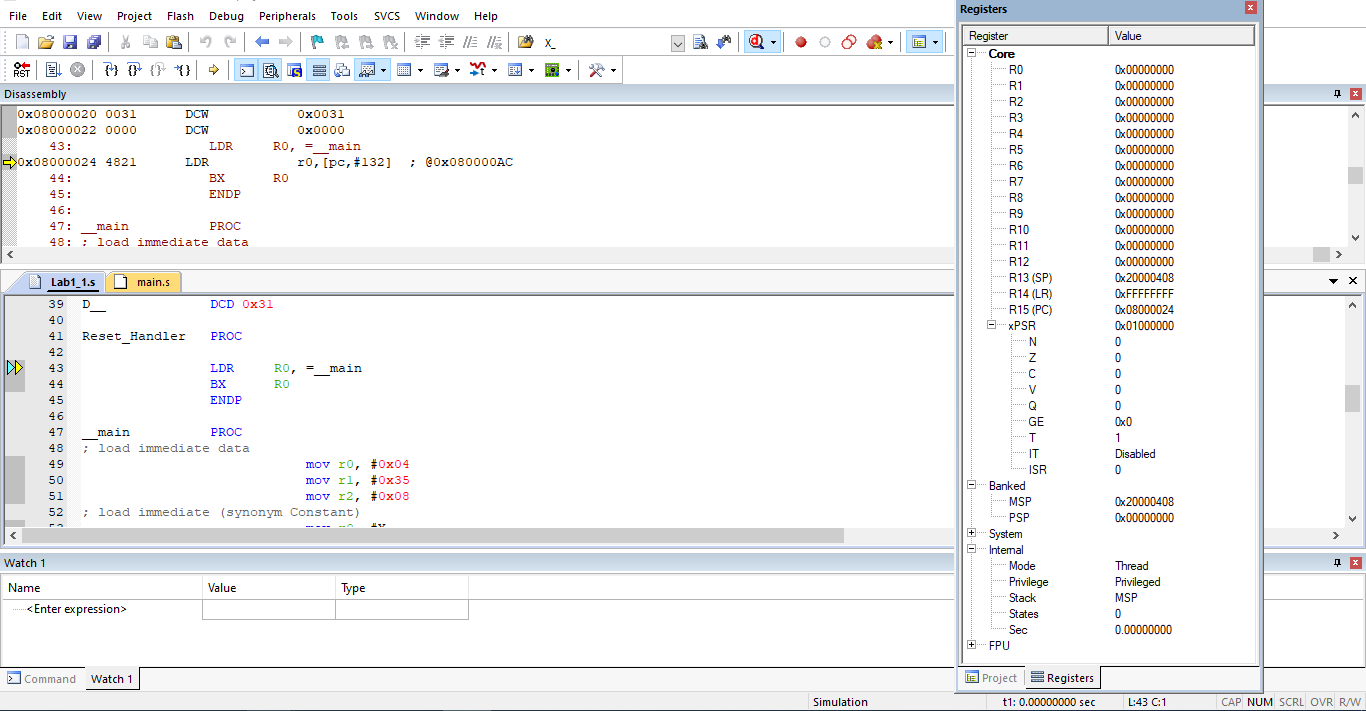
****

Рис. 3. Стан програми після скидання (RESET) мікроконтролера

Регістри R0-R12 – регістри загального призначення, де R0-R7 – молодші, звертатися до цих регістрів можуть всі команди як 16-ти бітноrо набору Thumb, так і 32-ти бітноrо набору Thumb2. R8-R12 – старші, звертатися до цих регістрів можуть всі команди набору Thumb 2 і деякі команди набору Thumb.

Регістр процесора R13 використовується в якості покажчика стека, причому в процесорі CortexM3 є два таких покажчика. Наявність двох покажчиків дозволяє реалізувати два незалежних стека. Основний покажчик стека MSP: це покажчик, який використовується за умовчанням. Він використовується ядром операційної системи, обробниками виняткових ситуацій, а також всіма програмними модулями, які вимагають привілейованого доступу до пам'яті. Покажчик стека процесу PSP: цей покажчик використовується прикладною програмою (якщо тільки не виконується оброблювач виняткової ситуації).

Реrістр R14 використовується в якості регістру зв'язку (Link Register LR). У acсемблерній програмі можна використовувати будь-яке ім’я реrістра R14 або LR. Регістр зв'язку призначений для збереження адреси повернення при виклику процедури або функції.

Регістр R15 служить в якості лічильника команд РС. Через наявність в процесорі CortexM3 конвеєра, значення, зчитане з цього регістра, буде відрізнятися від адреси виконуваної в даний момент команди (як правило, на 4).

Решта регістрів – регістри спеціального призначення: регістри стану проrрами (xPSR); регістри маскування переривань (PRIMASK, FAULTMASK і BASEPRI); регістр управління (CONTROL).

Із рис. 3. бачимо, що Елементи таблиці векторів (\_\_initial\_sp = 0x20000408) і Reset\_Handler (0x080000009) завантажено у відповідні регістри (SP) і (PC) процесора. В регістрі стану програми xPSR всі прапорці скинуто у стан «0», за винятком прапорця <T>, що вказує на застосування процесором системи команд Thumb/Thumb2.

Варто відзначити, як відбувається доступ до констант і змінних, розміщених у пам’яті:

Доступ до таких констант і змінних виконується конструкцією із двох команд,

; load Constant of code memory

ldr r3**,** **=**X\_\_

ldr r0**,[**r3**]** ; r0 = X

перша ініціює регістр вказівник адресою комірки пам’яті, а друга здійснює непряме читання вмісту цієї комірки у певний регістр, або запис вмісту регістру у комірку, на яку вказує регістр-вказівник.

При цьому команда ldr r3**,** **=**X\_\_ перетворюється асемблером у команду з відносною адресацією: ldr r3**,** **[pc, #offset]**

Для підтримки роботи цієї конструкції у сегменті коду, зразу за останньою командою програми, асемблер розміщує показники на константи і змінні, тобто записує в них адреси комірок пам’яті, у яких лежать константи та змінні. При цьому зміщення #offset в команді дорівнює відстані до комірки-показника.

pc + 4 + #offset

Важливим є те, що MCU Cortex-M3/M4 мікроконтролерів STM32 мають в своєму складі 3-х ступінчастий конвеєр: fetch -> decode -> execution.

Тобто, під час дешифровки однієї команди, виконується вибірка наступної.

Наслідки цього факту можемо бачити в наступному прикладі - дизасемблер вказує на адресу в пам’яті на 4 байти більше очікуваної:

57**:** ldr r3**,** **=**X\_\_

0x08000040 4B1B LDR r3**,[**pc**,**#108**]** ; @0x080000B0

58**:** ldr r0**,[**r3**]** ; r0 = X

0x08000042 6818 LDR r0**,[**r3**,**#0x00**]**

Вміст регістра РС перед початком команди дорівнює 0x08000040. Тоді в R3 мало би записатись число, що лежить за адресою 0x08000040+0x0800006C = 0x080000AC (6Ch=108), але мітка у коментарі вказує, що запис в R0 буде здійснюватися з адреси 0x080000B0.

Це пояснюється тим, що у фазі fetch за одну транзакцію з пам’яті одночасно витягуються 4 байти (одна 32-бітова, або дві 16-бітові інструкції) і після їх вибірки РС збільшиться на 4 (РС=0x08000040+4=0x08000044). Тому у фазі дешифрування (decode) і фазі виконання (execution) першої команди адреса комірки пам’яті, вміст якої буде зчитано у регістр r0, визначатиметься як PC + 0x0800006C= 0x08000044+ 0x0800006C= 0x080000B0, на що і вказує коментар @0x080000B0.

**Текст програми**

1. ; Initialize Stack Size
2. ; Equ 400 hex (1024) bytes
3. Stack\_Size EQU 0x00000400
4. ; Area STACK (NOINIT and READ/WRITE)
5. AREA STACK**,** NOINIT**,** READWRITE**,** ALIGN**=**3
6. ; Initialize memory equals Stack\_Size
7. Stack\_Mem SPACE Stack\_Size
8. \_\_initial\_sp
9. ; Vector Table Mapped to Address 0 at Reset
10. AREA RESET**,** DATA**,** **READONLY**
11. EXPORT \_\_Vectors
12. EXPORT Reset\_Handler **[**WEAK**]**
13. \_\_Vectors
14. DCD \_\_initial\_sp ; Top of Stack
15. DCD Reset\_Handler ; Reset Handler
16. \_\_Vectors\_End
17. AREA MYDATA **,** DATA**,** READWRITE**,** ALIGN**=**3
18. QA DCD 0
19. QL DCD 0
21. AREA MYCODE **,** CODE**,** **READONLY**
22. ; X = 20h, Y = 05h, Z= 04h
23. X EQU 0x04
24. Y EQU 0x35
25. Z EQU 0x08
27. ; Define Constant Data in Code Area
28. X\_\_ DCD 0x04
29. Y\_\_ DCD 0x35
30. Z\_\_ DCD 0x08
32. A\_\_ DCD 0xBF
33. B\_\_ DCD 0xA1
34. C\_\_ DCD 0x35
35. D\_\_ DCD 0x31
36. Reset\_Handler PROC
37. LDR R0**,** **=**\_\_main
38. **BX** R0
39. ENDP
41. \_\_main PROC
42. ; load immediate data
43. **mov** r0**,** #0x04
44. **mov** r1**,** #0x35
45. **mov** r2**,** #0x08
46. ; load immediate (synonym Constant)
47. **mov** r0**,** #X
48. **mov** r1**,** #Y
49. **mov** r2**,** #Z
50. ; load Constant of code memory
51. ldr r3**,** **=**X\_\_
52. ldr r0**,[**r3**]** ; r0 = X
53. ldr r3**,** **=**Y\_\_
54. ldr r1**,[**r3**]** ; r1 = Y
55. ldr r3**,** **=**Z\_\_
56. ldr r2**,[**r3**]** ; r2 = Z
58. ldr r3**,** **=**A\_\_
59. ldr r5**,** **[**r3**]** ;r5 = A
60. ldr r3**,** **=**B\_\_
61. ldr r6**,** **[**r3**]** ;r6 = B
62. ldr r3**,** **=**C\_\_
63. ldr r7**,** **[**r3**]** ;r7 = C
64. ldr r3**,** **=**D\_\_
65. ldr **r8,** **[**r3**]** ;r8 = D
66. ; (Y + Z) + X - (Y - X) + Z = Q ;X = 04h, Y = 35h, Z = 08h;
67. ;and r3, 0
68. **add** r3**,** r1**,** r2 ;r3 = (Y + Z)
69. **add** r3**,** r3**,** r0 ;r3 = (Y + Z) + X
70. **sub** r4**,** r1**,** r0 ;r4 = (Y - X)
71. **sub** r3**,** r3**,** r4 ;r3 = (Y + Z) + X - (Y - X)
72. **add** r3**,** r3**,** r2 ;r3 = (Y + Z) + X - (Y - X) + Z
73. ; Q = !(A and !B and C) + D and !C and B xor !(A and C and D), A = BFh, B = A1, C = 35h, D = 31h
74. mvn **r9,** r6 ;r9 = !B
75. **and** **r10,** r5**,** **r9** ;r10 = r5 and r9 = A and !B
76. **and** **r10,** **r10,** r7; r10 = r10 and r7 = A and !B and C
77. mvn **r10,** **r10**; r10 = !r10 = !(A and !B and C)
78. mvn **r9,** r7; r9 = !r7 = !C
79. **and** **r9,** **r8,** **r9** ;r9 = r8 and r9 = D and !C
80. **and** **r9,** **r9,** r6; r9 = D and !C and B
81. orr **r9,** **r9,** **r10** ;r9 = !(A and !B and C) + D and !C and B
82. **and** **r10,** r5**,** r7 ;r10 = A and C
83. **and** **r10,** **r10,** **r8** ;r10 = A and C and D
84. mvn **r10,** **r10** ;r10 = !r10 = !(A and C and D)
85. eor **r10,** **r10,** **r9** ;r10 = !(A and !B and C) + D and !C and B xor !(A and C and D)
87. ;Save results
88. LDR r0**,** **=**QA
89. **STR** r3**,** **[**r0**]**
90. LDR r0**,** **=**QL
91. **STR** **r10,** **[**r0**]**
92. \_\_mainloop
93. B \_\_mainloop
94. ENDP
95. END

**Висновок**

В результаті виконання даної роботи я набув основ роботи з мовою асемблеру STM32F4. Розглянув роботу 3-х ступінчастого конвеєра та структуру програми для STM32F4.