Лабораторна робота №1  
Ваш перший проект для STM32

Ознайомившись з теоретичними положеннями і виконавши завдання цієї роботи, ви

***узнаєте***:

* 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

***навчиться:***

* 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

Умови виконання роботи

Комп’ютер із операційною системою Windows 7 і вище

До початку роботи на вашому комп'ютері потрібно встановити інтегроване середовище розробки IDE µVision5 (див. Додаток А Встановлення і налаштування Keil uVision5, файл [Getting started](http://www.keil.com/mdk5/install), або   
[фірмовий документ](http://www2.keil.com/docs/default-source/default-document-library/mdk5-getting-started.pdf?sfvrsn=0) ARM Keil).

Для налагодження програм цієї роботи необов’язково, але ж бажано мати лабораторний стенд – один з оціночних модулів , що вказані вище.

Чому ж асемблер, запитаєте ви?

У більшості випадків код програми пишуть в C/С++ або на інших мовах високого рівня, тому для більшості розробників програмного забезпечення не потрібно знати деталі набору інструкцій. Проте, корисно мати загальний огляд доступних інструкцій та уявлення про синтаксис мовлення. Наприклад, знання в цій області можуть бути дуже корисними для налагодження програм. До того ж застосування вбудованого (Inline-асемблеру), та технології роздільної компіляції asm-процедур, що викликаються з С-програми, із зв’язуванням їх у загальний проект, дозволяє оптимізувати результуючий код у деяких критичних випадках.

Навіть найпалкіші противники погодяться з тим, що програми на асемблері - найшвидші, найменші і можуть те, що не під силу програмам, які створено будь-якою мовою програмування.

Особисто мені подобається такий вислів Пирогова В.М. - автора популярної книги   
«ASSEMBLER Учебный курс»:

«Програмування на асемблері дає відчуття влади над комп'ютером,   
а жага влади ̶ один з найсильніших інстинктів людини!»

# ВАШ перший проект для STM32 у Keil µVision

# Запускаємо і налаштовуємо Keil μVision 5



Відкриється «робочий стіл» програми **µVision**, такий що відображено на рис.1.   
Для створення проекту на мові Assembler два наступних пункти можна не виконувати. Але для програмування мікроконтролера на мові С/C++, чим ми й будемо займатися у більшості лабораторних робіт, потрібна бібліотека функцій, які підтримують роботу з ресурсами цього мікроконтролера.

* + 1. Для встановлення відповідної бібліотеці запустимо утиліту Pack Installer (  )

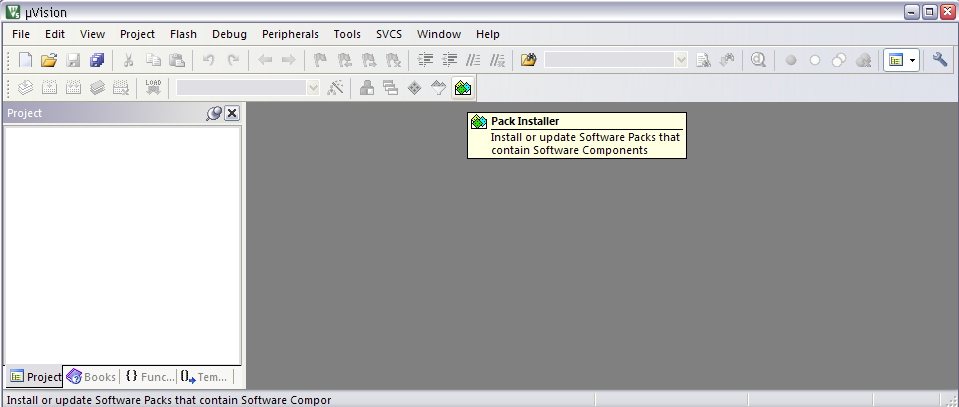


Рисунок 1 - ***Робочий стіл Keil μVision5. Виклик інсталятора Pack Installer***

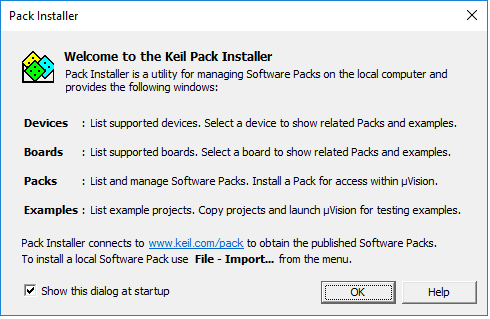
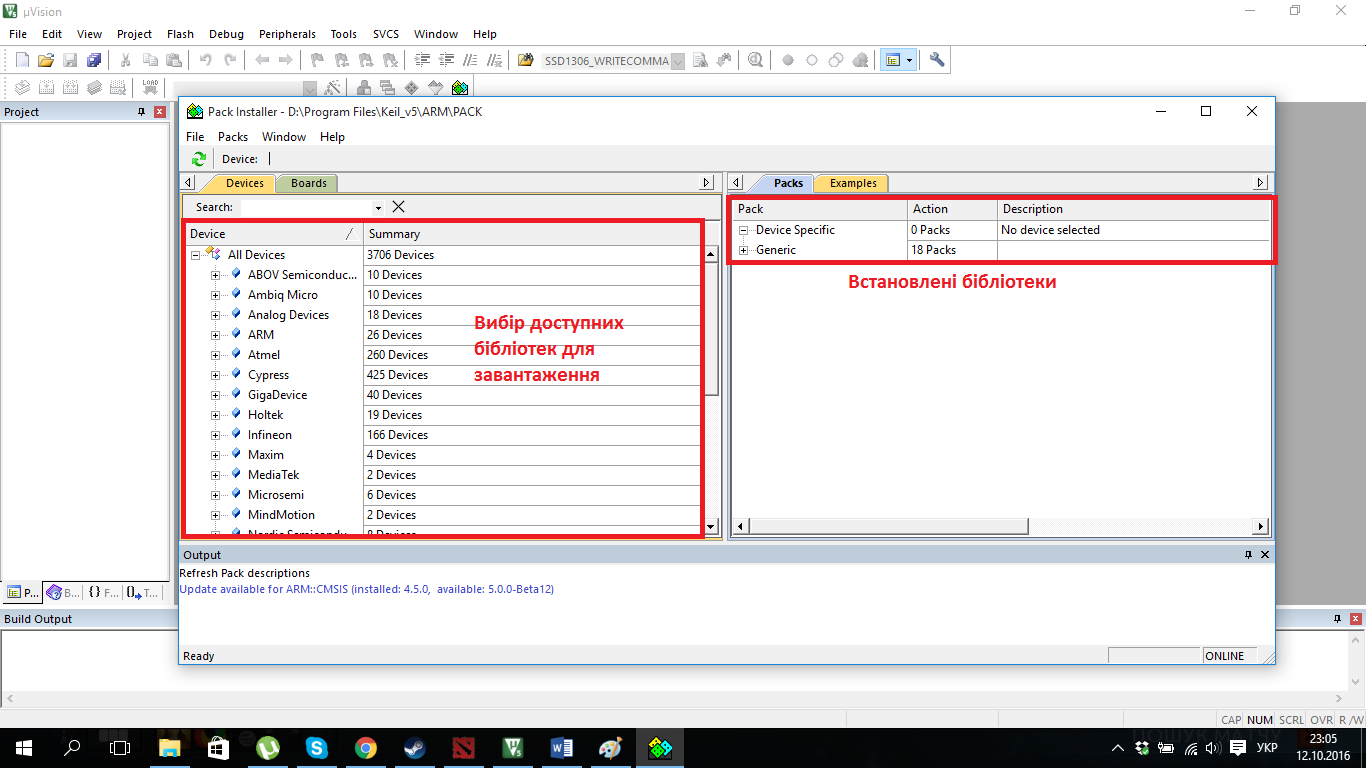
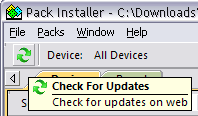
Pack Installer (рис.2) має два списки: з бібліотеками доступними для завантаження (ліве поле) та з відібраними і встановленими бібліотеками і прикладами готових проектів (праве поле)[[1]](#footnote-1). 

Рисунок2 − ***Вигляд вікна Pack Installer***

Якщо в полі вибору бібліотек пусто, або немає потрібної бібліотеки, слід зробити оновлення набору.

Для цього виконайте команду **Packs 🡪 Check For Updates** (рис. 3), або натисніть кнопку  :   
****

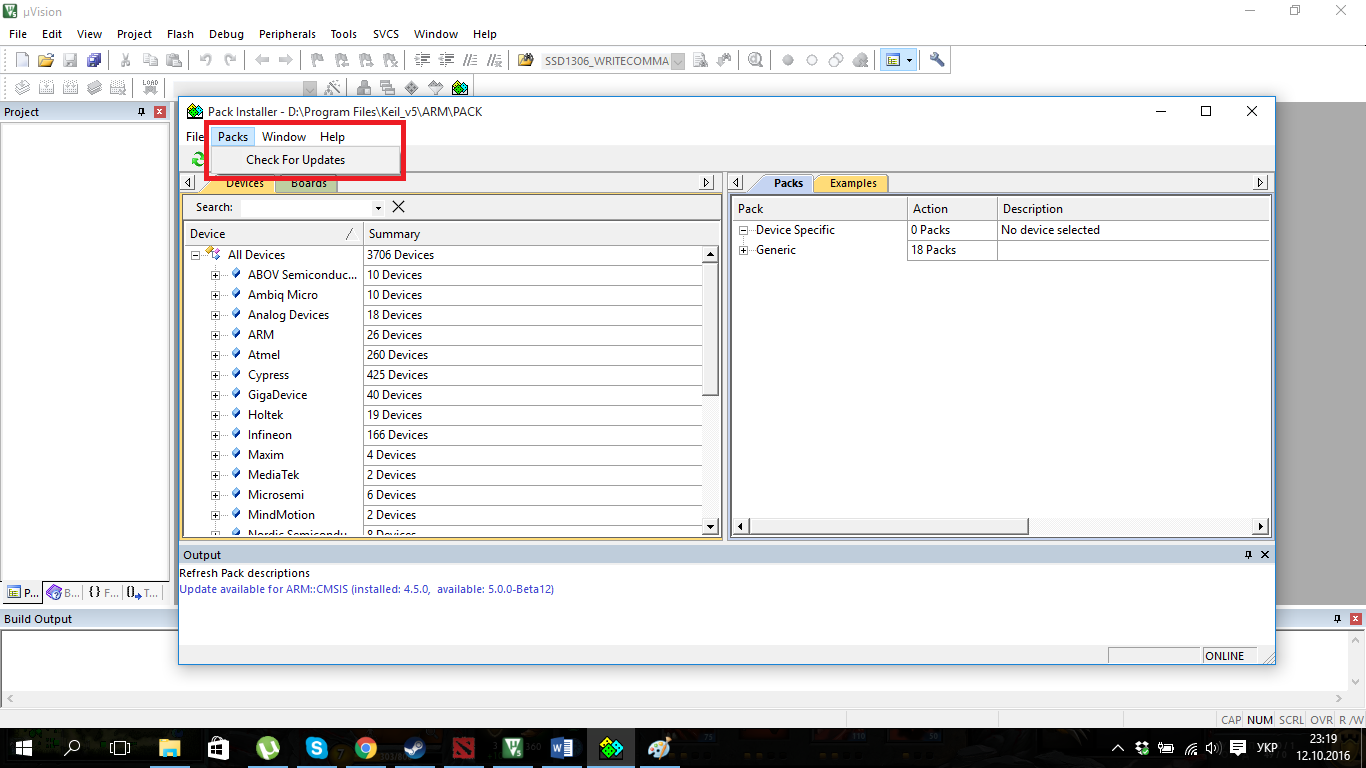
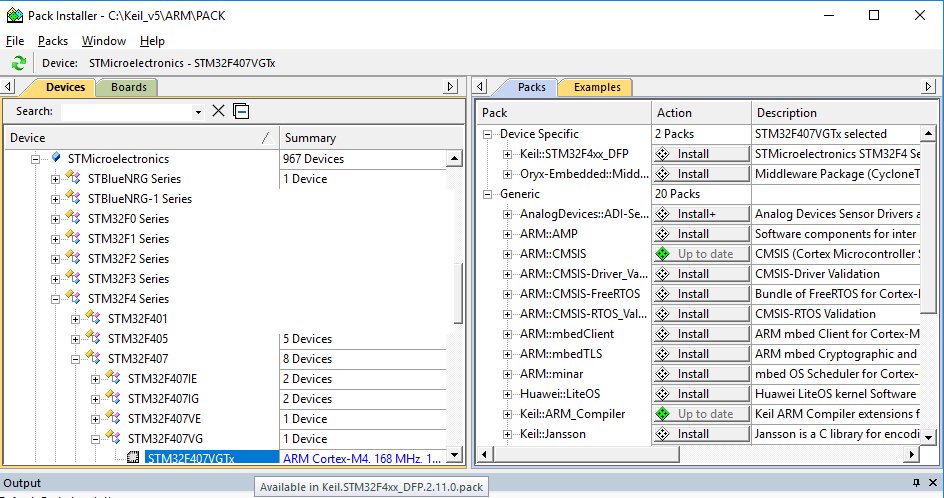


Рисунок 3 − ***Оновлення набору бібліотек***

* + 1. На вкладці Devices обираємо потрібний нам мікроконтролер: **STMicroelectronics 🡪 STM32F4Series 🡪 STM32F407 🡪 STM32F407VG 🡪 STM32F407VGTx[[2]](#footnote-2)** (рис. 4). Встановлюємо для нього бібліотеки (кнопка <Install>у правому полі). Чекаємо певний час поки буде йти завантаження та встановлення, потім закриваємо **Pack Installer**.



**Обираємо виробника  
(STMicroelectonics), серію   
та потрібний мікроконтролер**

**Бібліотека CMSIS підтримки ядра CORTEX обраного мікроконтролера і необхідні поширення компілятора  
будуть згенеровано і встановлені автоматично**

**Встановлюємо бібліотеки для  
обраного мікроконтролера**

Рисунок 4 − ***Встановлення бібліотек для певного мікроконтролера***

Подальші дії, які виконують при створенні проекту на мові С / C++, будуть розглянуті у наступних роботах. Зараз же розглянемо лише дії, що пов’язані із створенням проекту програм на мові Асемблер.

# Створюємо новий проект на Асемблері

* + 1. Виберіть **Project 🡪 Nev** μ**Visson Project …** (рис.5)

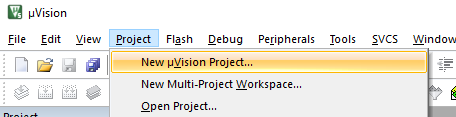


Рисунок 5 − ***Створення нового проекту***

* + 1. Тепер потрібно вказати назву проекту та шлях до директорії («папці»), в якій буде знаходитися проект. Взагалі-то Keil не потребує створювати для кожного проекту окрему папку, але я рекомендую це робити. Тому у вікні, що відкриється, створить () або оберіть попередньо створену папку, наприклад, **ASM\_1** і завдайте ім’я файлу проекту, наприклад, **Asm\_pro1** (рис. 6)[[3]](#footnote-3).

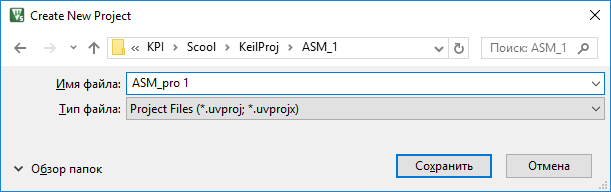
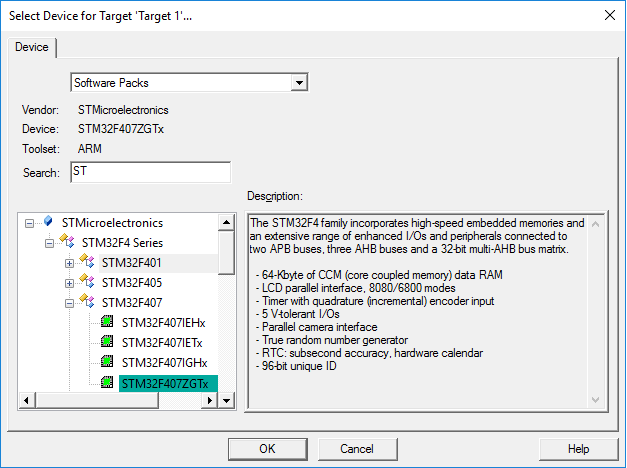


Рисунок 6 − ***Завдання назви і розміщення нового проекту***

* + 1. У наступному вікні (рис. 7) обираємо потрібний мікроконтролер: **STMicroelectronics 🡪 STM32F4Series -> STM32F407 🡪 STM32F407VG🡪 STM32F407VGTx** і, проглянувши опис (Description:) його параметрів, тиснемо <OK>.



Для спрощення пошуку почніть вводити назву мікроконтролера у полі «Search:»

Основні параметри мікроконтролера

Рисунок 7 − ***Обрання мікроконтролера***

Після цього зазвичай з’являється вікно для налаштування довкілля виконання проекту (рис. 8), але ж воно буде нам потрібно при створенні проекту на С. Тому пропустимо зараз цей крок, натиснувши кнопку “Cancel”.

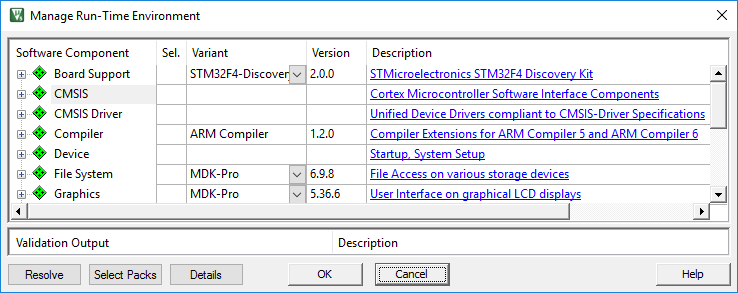
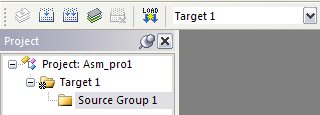
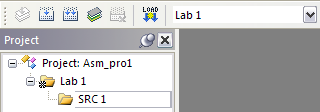


Рисунок 8 ***− Вікно обрання довкілля проекту***

* + 1. Наступним з’явиться вікно структури проекту, фрагмент якого показано на рис. 9, а. Але, при бажанні, назви елементів структури можна змінити, наприклад так, як показано на рис.9,б.   
       Для цього потрібно виділити елемент і натиснути клавішу редагування <F2>, або ще раз клацнути по ньому мишкою и ввести у текстове поле нове ім’я.

Заміна назви елементу структури проекту

Об’єкт

Група файлів

а) б)

Рисунок 9 ***− Заміна назви елементів структури проекту***

* + 1. Далі потрібно додати до проекту вихідний (*source*) файл програми. Ваша перша програма не буде виконувати ні яких дій, але її файл допоможе нам розібратися із загальною структурою програми на Асемблері і з основними директивами мові Асемблер для завдання елементів цієї структури. У подальшому цей файл можна буде застосовувати як файл-шаблон для створення на його основі інших програм[[4]](#footnote-4).
    2. Якщо вихідний файл програми вже створено[[5]](#footnote-5), то додавати його до проекту слід командою   
       **ADD Existing Files to Sourse Group 1** з контекстного меню[[6]](#footnote-6) елементу **Sourse Group 1** / **SRC 1**(рис 10):

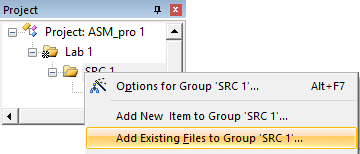


Рисунок 10 − ***Додавання до проекту існуючого файлу програми***

У діалоговому вікні, що з’явіться далі, оберіть тип файлу, знайдіть папку з вихідними файлами[[7]](#footnote-7) і оберіть необхідний файл, як показано у прикладі на рис. 11.

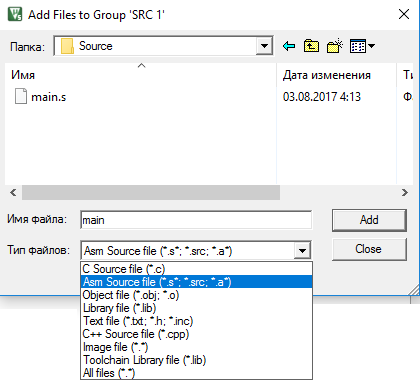


Рисунок 11 − ***Вибір існуючого вихідного файлу при його додаванні у проект***

Такі ж дії потрібно виконати, як що ви бажаєте застосувати файл з іншого проекту як основу вихідного файлу нового проекту.

* + 1. Як що ж, як у нашому випадку, файлу ще нема, то його можна створити безпосередньо у Keil. Для цього клацніть правою кнопкою мишки по елементу **Source Group** / SRC 1. З’явиться контекстне меню, у якому оберіть команду **Add New Item to Group ‘SRC 1’** (рис.12).
    2. У наступному вікні оберіть тип файлу **А** *(Asm File)* і вкажіть його ім’я, наприклад, **main**.   
       Зверніть увагу на розширення (тип) асемблерного файлу (**\*.s**).

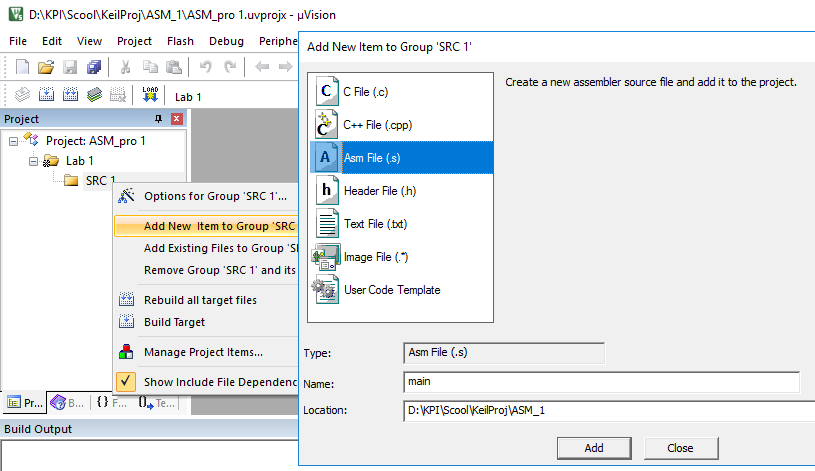
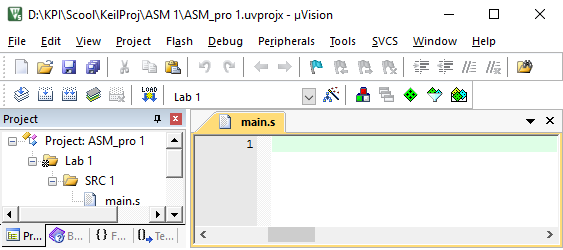


Рисунок 12 − ***Додавання до проекту нового файлу програми***

Поки що доданий до проекту файл порожній, але у правому полі відкриється вкладка для введення тексту програми (рис. 13) у файл **main.s**

****

Введіть суди текст програми

Рисунок 13 − ***Вікно для введення тексту програми вихідного файлу***

* + 1. Заглянемо у папку проекту **ASM1** на вашому диску (рис.14). Тут окрім вихідного (поки що порожнього) файлу **main.s**, а також основного файлу проекту **Asm\_pro1.uvprojx** і додаткового xml-файлу **Asm\_pro1.uvoptx** створені папки з файлами із налаштуванням налагоджувача (**DebugConfig**), і результатами компіляції (**Listings, Objects**). Тут же бажано самостійно створити вже згадану папку **Source** для розміщення в неї інших вихідних файлів даної роботи.

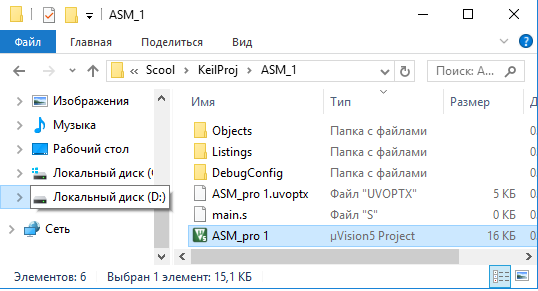


Рисунок 14 - ***Вигляд папці проекту після додавання вихідного файлу***

Зробимо паузу у створенні проекту і розглянемо деякі загальні речи, що до структури програми на асемблері, структури адресного простору пам’яті миікроконтролера (Memory map  *̶ карти пам’яті*) , а також можливі варіанти завантаження програми у пам’ять і розміщення елементів програми у певних її ділянках.

# Структура програми та особливості її розміщення і запуску

Перед тим як почати складати текст нашій першій програми, розглянемо загальні вимоги до її структури, особливості розміщення її елементів (*сегментів*) у пам’яті мікроконтролера і процедуру запуску програми.

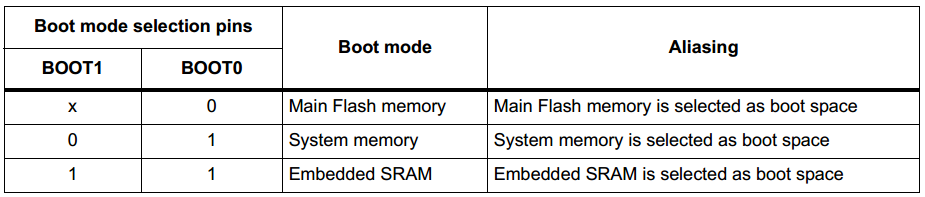
* + 1. Програми для мікроконтролерів будь якого виробника повинні задовольняти загальним вимогам архітектури їх процесорного ядра. У нашому випадку це ядро СortexTM-M3/CortexTM-M4[[8]](#footnote-8) компанії ARM®. Процесори Cortex-M мають фіксовану карту пам’яті, яка визначає, призначення окремих ділянок адресного простору і то, який шинний інтерфейс повинен бути застосовано для звернення до тої чи іншої ділянці пам’яті.
    2. Увесь 4GB адресний простір процесорів Cortex-M поділено на 8 ділянок (*regions*) по 512MB (0,5 GB) у кожній. Поділення пам’яті на певні ділянки ґрунтується на типовому їх застосуванні. В результаті виділяють області пам’яті:
* *Code region* − для розміщення і доступу до програмного коду (стартова адреса - 0x00000000);
* *SRAM-region* − длярозміщення і доступу до даних і стеку (стартова адреса - 0x20000000);
* *Peripherals region* − длядоступу до периферії (стартова адреса - 0x40000000);
* *Internal Peripherals* − длявнутріпроцесорних компонент управління і налагодження (0xE0000000)

В мікроконтролерах STM32F4xx області пам’яті реалізовані не в повному обсязі, але їх локалізація (розміщення) відповідає карті пам’яті процесорного ядра Cortex-M3/Cortex-M4 (рис. 15)[[9]](#footnote-9) і не може бути змінено.



Рисунок 15 - ***Карта пам’яті мікроконтролерів STM32F4xx***

* + 1. В мікроконтролерах STM32 реалізовано спеціальний механізм, який дає можливість загружати код програми у три різні ділянки пам’яті: **Flash**, **System memory** або **SRAM**. Область загрузки задається виводами мікроконтролера (BOOT1,BOOT0) [5, с.69].



* + 1. Як правило («за замовчанням»), загрузка програми здійснюється в область коду (0х0800 0000 ÷ 0x0801FFFF), реалізовану як Flash-пам’ять (див. Рисунок 15). При цьому стек и сегмент даних розміщуються у SRAM, що починається з адреси 0x2000 0000 і представлена у мікроконтролері STM32F4xx двома блоками SRAM1 (112KB) и SRAM2 (16KB). Регістри периферійних пристроїв відображені в область Peripherals (0x4000 0000).
    2. Після виходу процесора з стану скидання (*Reset*) він зчитує із пам’яті два 32-бітних значення (Рисунок 16):
* за адресою 0x08000000 − початкове значення регістру R13 (вказівника вершини стеку SP);
* за адресою 0x08000004 − «вектор скидання» (адреса, з якої починається виконання програми).

Вибірка Вибірка Вибірка Продовження

початкового вектора першої виконання

значення SP скидання у PC команди програми

**. . .**

Адреса =   
RESET VECTOR

Адреса =   
0x08000000

Адреса =   
0x08000004  
(Reset Vector)

RESET

Таблиця векторів виключень і переривань (*Vectors Table*)

Час

Рисунок 16 − ***Процес запуску програми (цикл скидання- RESET Cycle)***

Взагалі-то вказані тут два 4-байтні елементи, а саме - початкове значення вказівника стеку (SP) і вектор скидання (Reset Vector), є лише першими елементами так званої таблиці векторів (*Vectors Table*), яка зазвичай передує основній програмі. *Таблиця векторів* це масив з 256 елементів, які є 32-бітними значеннями початкових адрес програм-обробників так званих *виключень (Exceptions)* та *переривань (Interrupts).* При появі події, що генерує виключення/переривання (як що це дозволено), запускається відповідний *обробник виключення/переривання*.

**Увага**: *Молодші біти значень векторів (і взагалі будь яких покажчиків адрес у Code-сегменті) повинні містити «1» («одиницю»), що вказує процесорові на застосування набору команд* ***Thumb/Thumb-2****. Як що ж цей біт окажеться скинутим у 0, то процесор зробить спробу переключитися до виконання команд з набору* ***ARM.*** *Це* *цілком природно для ранніх моделей ARM-процесорів, в яких застосовувалися різні набори команд. Але ж у процесорів Cortex-M, які виконують лише команди* ***Thumb/Thumb-2****, це недопустимо і визве генерацію виключення відмови (Hard Fault).*

Більш детально із структурою таблиці векторів ми ознайомимося пізніше, при вивченні теми переривань. Тут лише відмітимо, що подія скидання (RESET) генерує виключення з номером 1, тому адреса вектору скидання має зміщення 1×4 = 0×00000004 від адреси початку таблиці. Оскільки ж область загрузки коду у STM32 починається з адреси 0x0800 0000, то вектор скидання розміщено за адресою 0x0800 0004.

# Файл-шаблон програми на асемблері

**Увага:** *Синтаксис мови асемблеру залежить від інструментарію, що застосовується для створення проекту. У нашому випадку середовище IDE Keil µVision5 орієнтовано на синтаксис асемблера ARM (armasm). Вивчення синтаксису інших асемблерів краще за все починати із розгляду прикладів програм, які поставляються разом із засобами розробки.*

* + 1. Щоб створити програму безпосередньо у вікні Keil μVision перейдіть у вікно введення тексту програми “**main.s**” і почніть вводити текст, який надано на рис.17.

## 

Вікно редактора тексту програми

Рисунок 17 − ***Вікно із текстом вихідного файлу програми***

Редактор автоматично виконує нумерацію рядків програми та підфарбовує певні елементи синтаксису.

* + 1. Програма починається з визначення сегменту стека (рядки 3÷7). Потім об’являється сегмент запуску програми із зарезервованим ім’ям RESET, у якому завдається таблиця векторів (\_\_Vectors ÷ \_\_Vectors\_END) із не менш ніж двома елементами (32-бітними словами). Останні завдають початкове значення вершини стеку (\_\_initial\_sp) і вектор скидання з адресою обробника цієї події (*RESET\_Handler*).
    2. Процедура-обробник скидання (RESET\_Handler) передає управління основній процедурі програми (\_\_main), яка зазвичай складається з двох секцій:  
        ̶ початкова секція, яка виконується одноразово. В неї, наприклад, здійснюється початкова ініціалізація змінних;  
        ̶ ̶ основна секція, яка зазвичай уявляє собою безкінечний цикл.

Розберемо цей файл по директивам[[10]](#footnote-10). Їх більш докладніший опис надано у файлі «Основні директиви Асемблеру», що знаходиться у папці «Додатки».

**Stack\_Size EQU 0x00000400**   
; присвоїмо константі Stack\_Size значення 0х400 (десятковий еквівалент 1024d).

**AREA STACK,NOINIT,READWRITE,ALIGN=3**   
; Визначимо сегмент з ім’ям **STACK,** вміст якого не буде ініціалізовано при запуску (параметр **NOINIT**), вирівняний на кордоні по 4 байти (**ALIGN=3)**, і який має дозвіл як читання так і запису (**READWRITE)**.  
У цьому сегменті буде розміщено стек, але ім’я може бути обране інше. Важливо, щоб на комірку, яка лежить зразу поза цим сегментом, вказував перший елемент таблиці векторів. У нашій програмі адреса поза стеком помічено як **\_\_initial\_sp** і значення цієї мітки містить перша комірка таблиці векторів. Саме це і робить цю ділянку пам’яті стеком, оскільки, як було роз’яснено у п.1.3 (див. рис.16), значення цього елементу завантажується у регістр-вказівник стеку SP (*Stack Pointer*) при запускі.

**Stack\_Mem SPACE Stack\_Size** ; виділяємо під стек 0x400 байт (1Kbyte).

**\_\_initial\_sp** ; завдаємо мітку наприкінці стеку. Так як стек «росте вниз» ̶ то по факту це є вершина стеку.

**AREA** **RESET**, **DATA**, **READONLY** - початковий сегмент, що розміщується на початку пам'яті даних, із зарезервованим ім’ям RESET, допускає тільки читання (**READONLY)**, ініціалізується при запуску або скиданні мікроконтролера, і у своєму початку містить таблицю векторів виключень (переривань). Кожний вектор – це 32-бітна адреса процедури обробки зовнішніх переривань (*Interrupts handlers*), або внутрішніх виключень і похибок (*Exeptions*, *Faults*).

**EXPORT \_\_Vectors** - визначення зовнішніх імен, таких, що будуть доступні іншим програмам,

**EXPORT RESET\_Handler**  наприклад, лінковщіку Keil (Linker).

**\_\_Vectors** - мітка початку сегменту (таблиці векторів), з якої буде розміщено адреси (*вектори*) переривань. (Лінковщіком їй буде завдано адресу 0x08000000)

Оскільки у ваших перших програмах переривання не будуть застосовані, то таблиця векторів може містити лише два елементи, які завдаються директивами **DCD** (*Define Constant Data*), а саме:

**DCD \_\_initial\_sp** - Vector 0 – 32-бітне слово із зміщенням 0, яке буде завантажене у регістр SP(MSP), і тому в цьому слові повинно бути розміщене початкове значення вершини стеку, завдане в нашій програмі міткою \_\_initial\_sp.

**DCD Reset\_Handler** - Vector 1. Слово в якому розміщено адреса мітки Reset\_Handler. Лінковщиком вказано мітка Reset\_Handler як точка входу в програму, тобто виконання програми починається з цієї процедури.

**\_\_Vectors\_End** - мітка кінця таблиці векторів.

**AREA |.text| , CODE, READONLY** ; об’явлення сегменту коду із ім’ям **.text**.

**[WEAK]** вказує на те, що це ім’я можна перевизначити в іншому місці.

**\_\_main** – ім’я процедури в якій ми будемо писати свою програму, тому ми передаємо управління на процедуру \_\_main командою **BX**, без збереження адреси в регістрі LR (Тобто без можливості повернення до процедури Reset\_Handler).

Програма містить 3 команди – **LDR**, **BX**, і **В**.

Команда **LDR** (*Load Regi*ster)(рядок 21) «Завантажити регістр»   
 **LDR R0, =\_\_main** ; Занести у регістр R0 значення адреси початку процедури main

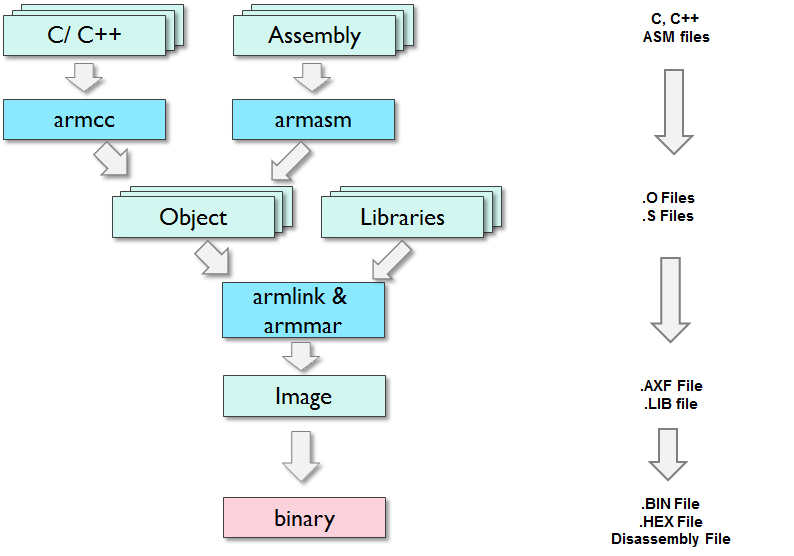
Команда **BX reg** (*Branch indirect*) – перехід по адресі, що визначено регістром.

Команда **B label**(*Branch*) безумовний перехід до адреси, що визначено міткою.

Повернемось до побудови проекту.

# Етапи Побудови проекту. Налаштування IDE Keil µVision

Процес побудови програми містить декілька етапів, а саме (рис.18):   
Compile (Компіляція) 🡪 Assemble (Асемблювання) 🡪 Link (Зв’язування «Лінковка») 🡪  
 🡪Download (Створення виконавчих файлів програми і завантаження у пам’ять контролера):



*Трансляція:****armasm*** *{options} inputfile*

*Зв’язування «Лінковка»):****armlink*** *{options} object-file-list*

**Образи програм  
(Об’єктні файли)**

***.bin*** *та* ***.hex*** *- файли створюються на заключній стадії побудови проекту*

**Вихідні файли**

**Виконавчі файли програм**

**Об’єктні файли**

Рисунок 18 − ***Етапи процесу побудови проекту***

Інтегроване середовище для розробки програм **IDE** (*Integrated Development Environment*) **Keil µVision** з пакету розробки для ARM компанії Keil (MDK-ARM) підтримує всі вказані етапи. У основному всі компоненти IDE вже налаштовані, але при побудові першого проекту для певного мікроконтролера слід задати деякі їх параметри, а саме: вказати тактову частоту мікроконтролера, обрати спосіб налагодження виконавчого файлу програми (емуляція або реальний контролер), вибрати програму-загружчик і налаштувати її параметри.

* + 1. Оберіть команду Option for Target “Target 1” … з контекстного меню елементу «Lab 1», або натисніть відповідну кнопку (рис.19):

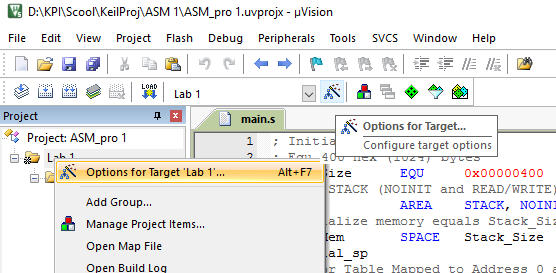
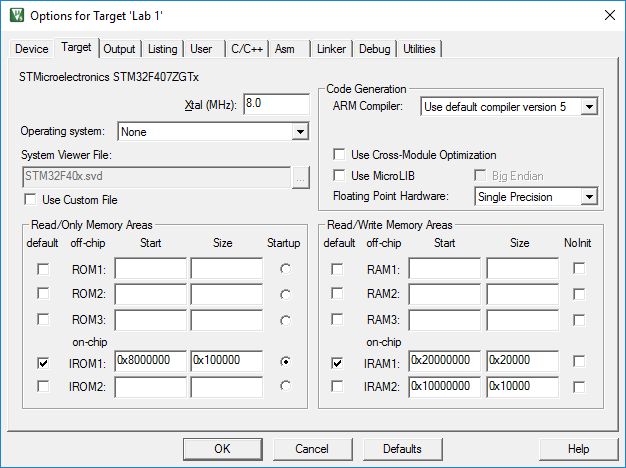


Рисунок 19 − ***Виклик операцій налаштування інструментів побудови проекту***

* + 1. На вкладці Target встановіть частоту мікроконтролера і перегляньте інші параметри проекту (рис.20).



С Початкова адреса і розмір двох ділянок SRAM

С Початкова адреса і розмір Flash

Рисунок 20 − ***Налаштування параметрів проекту***

* + 1. На рис.21 показано вміст вкладки Linker

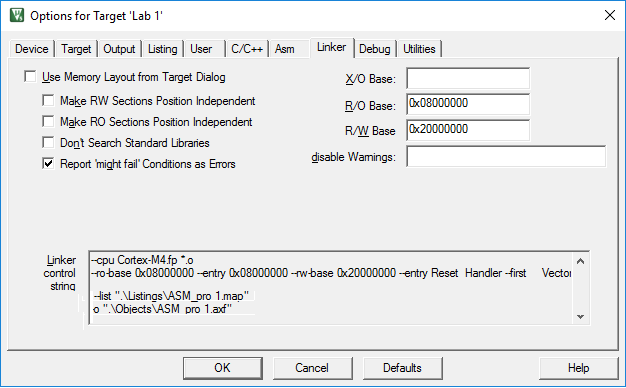


Рисунок 21 − ***Опції лінкеру***

Тут у полі стану вказано тип процесорного ядра ( --cpu Cortex-M4.fp) для якого буде створено об’єктний файл (\*.o), початкові адреси (ro - *read only*)-сегменту коду (ro-base 0x08000000) і сегменту даних (-rw-base 0x20000000), адреса (--entry 0x08000000) і зарезервоване ім’я (**Reset\_Handler**) точці входу (entry) в основну програму, що є вмістом першого вектору (first \_\_Vectors ) таблиці векторів, а також типи, місця розміщення і назви додаткових файлів, які будуть створені при побудові проекту, а саме - карта проекту \Listings\ASM\_pro 1.map і файл образу програми \Objects\ASM\_pro 1.axf )

* + 1. Наш перший проект, оскільки у ньому ми не виходитиме за межи можливостей процесорного ядра контролера, будемо налагоджувати у режимі програмної емуляції (**«Use Simulator»**), тобто не на реальному контролері. Для цього на вкладці Debug оберіть відповідну опцію (рис.22).

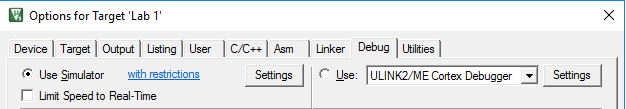


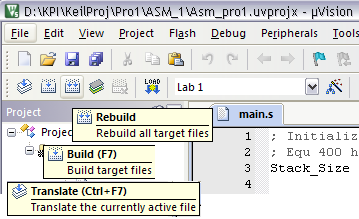
Рисунок 22 − ***Вибір режиму програмної емуляції***

Для завершення процедури встановлення параметрів IDE натисніть у останньому вікні кнопку <OK>.

Тепер все готово до побудови проекту.

# Побудова проекту (Трансляція + Лінковка + Налагодження)

* + 1. Запуск окремих етапів або всього процесу побудови проекту виконується відповідними командами (рис. 23):



Створення проекту: Трансляція + Лінковка

Видалення усіх компонент проекту і створення його знову

Трансляція поточного файлу

Рисунок 23 − ***Інструменти запуску етапів побудови проекту***

* + 1. Виконаємо побудову проекту ̶  (<**F7**>). Внизу з’явиться вікно стану <Build Output> (Рисунок 24)

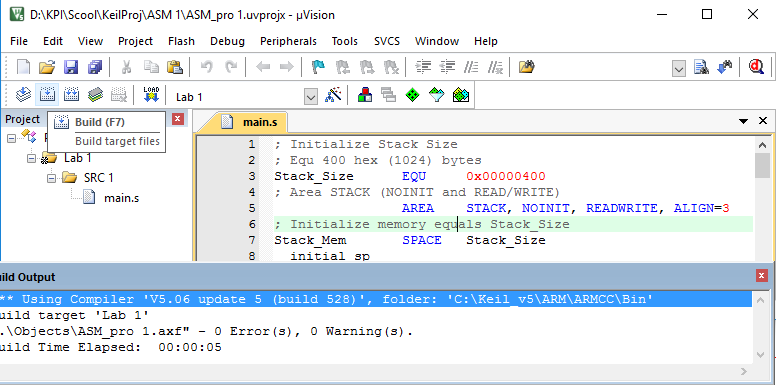
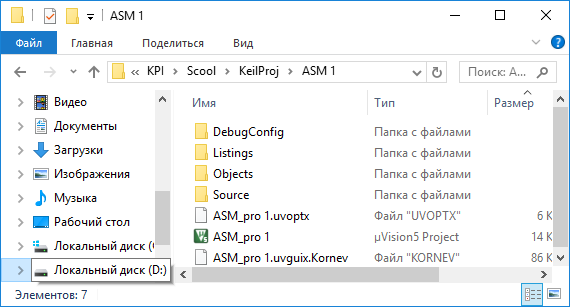


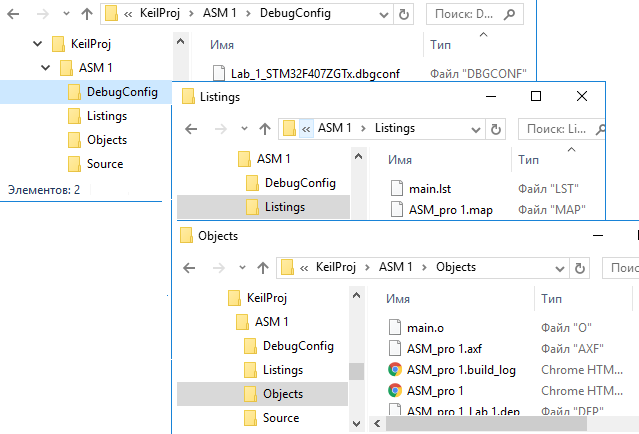
Рисунок 24 − ***Побудова проекту. Вікно «Build Output”***

* + 1. Тепер знову заглянемо в основну (рис.1.25, а) і додаткові папці проекту (рис. 1.25, б).



Папку з ім’ям Source створить самостійно і зберігайте в неї вихідні файли проекту

а)



Об’єктні файли («Образи» проекту)

Лістинг і карта пам’яті проекту

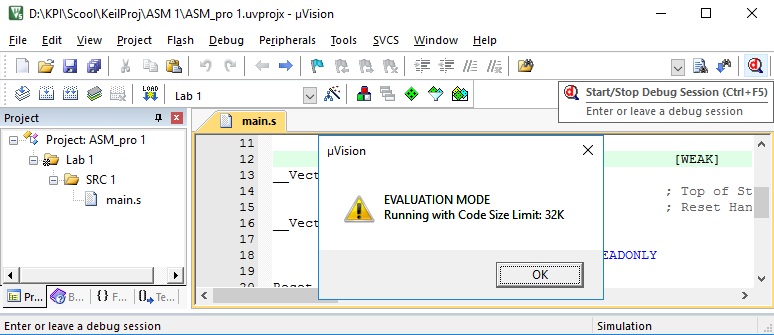
Файл конфігурації проекту

б)

Рисунок 25 − ***Файли проекту***

***а) основна папка, б) додаткові папки Listing та Objects***

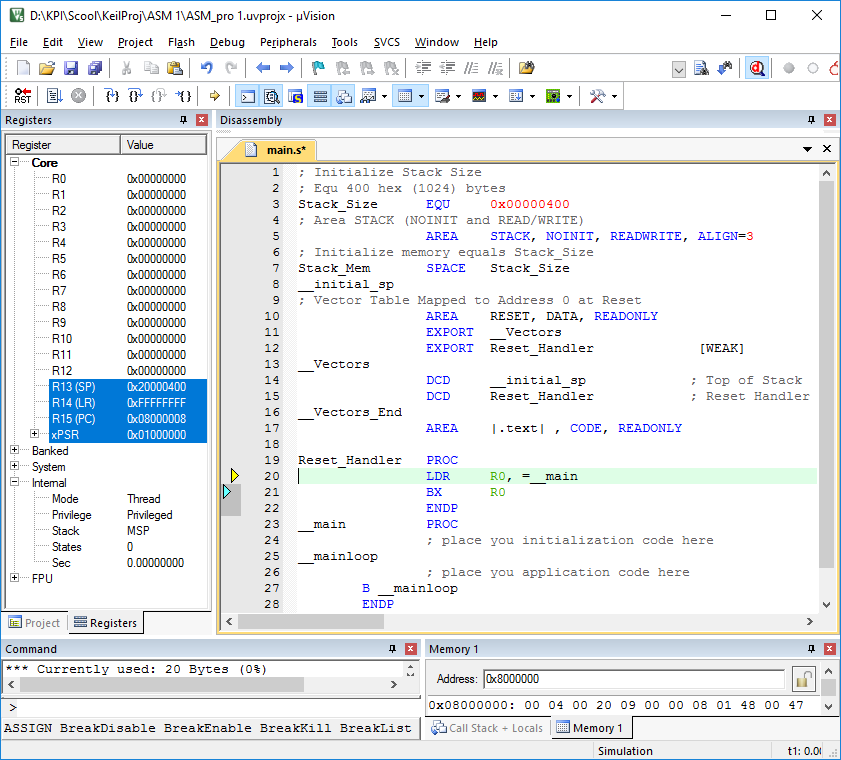
* + 1. Перейдемо до налагодження («відладки») програми проекту. Для цього тиснемо кнопку [[11]](#footnote-11), та після ознайомлення із попередженням про обмеження навчальної версії програми Keil Light (Evaluation Mode), яку ми встановили, тиснемо кнопку <OK> (рис.26).



Попередження про обмеження навчальної версії Keil

Рисунок 26 − ***Запуск Debugger***

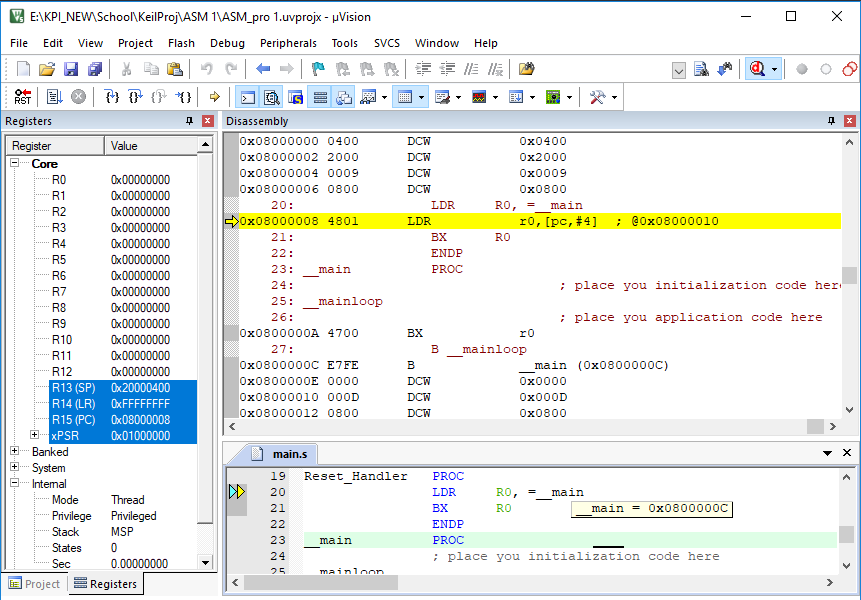
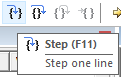
З’явиться вікно Debugger (рис.27)



Потягніть за цей елемент управління і встановіть необхідний розмір поля “Disassembly”

Рисунок 27 − ***Вікно Debugger***

Якщо вікно Debugger відкрилося у такому вигляді, тобто поле «Disassembly», яке нам потрібне, повністю закрито вкладкою із текстом вихідного файлу «main.s», то показати це поле можна різними способами:  
 ̶̶̶̶ наведіть курсор мишки на білу смужку, що розділяє заголовки поля «Disassembly» і вкладки «main.s», і потягніть вниз за «інструмент», такий як показано на рис.27, і встановіть потрібний розмір вікон (рис. 28);  
 ̶ або двічі клацніть по заголовку поля і воно відкриється поверх вкладки «main.s».



Vector Reset

Stack Top

Біт Vector Reset.0 =1   
признак Thumb

Наведіть курсор на елемент синтаксису, щоб побачити його значення

Рисунок 28 − ***Поля «Registers», «main» і «Disassembly» у вікні Debugger***

Що ми тут бачимо?

|  |  |
| --- | --- |
| **На вкладці «Register»**:  - 13 регістрів загального призначення R0 ÷ R12;  - R13 (SP) регістр-вказівник стеку (*Stack Pointer*);  - R14 (LR) регістр зв’язку (*Link Register*) ̶ в якому зберігається адреса повернення, при виклику процедури або функції, наприклад, командою переходу із посиланням (BL *– Branch Link*);  - R15 (PC) регістр-вказівник команд (*Program Counter*);  - xPSR – регістр стану програми (Program Status Register)[[12]](#footnote-12)  У групі «Internal» вказано: **Mode** – режим роботи процесору (Tread /«*Потік*» режим програми споживача ); **Privilege** – рівень привілеї (Privileged/Non Privileged); **Stack** – активний у даний момент вказівник стеку –«Основний» (MSP – *Main-SP*) або «Вказівник стеку процесу» (PSP – *Process -SP*) **States** – крок налагодження програми; **Sec** – кількість секунд виконання фрагменту програми у режимі реального часу. | **У полі «Disassembly»**  - фрагмент лістингу програми, з адресами, кодами і мнемоніками команд і директив.  Тут, у перших чотирьох рядках вказано (виділено синім) вміст векторів  Vector 0🡪 Stack Top = 0x20000400 Vector 1🡪Vector RESET=0x08000009, що завантажуються після скидання (RESET) у відповідні регістри SP і РС (див. вкладку Registers).  Зверніть увагу, що стартова адреса  РС = 0х8000008, а Vector RESET = 0х8000009, де біт <0> =1 указує на систему команд Thumb.  Жовтим у вікні “Disassembly” виділяється поточна команда.  На вкладці вихідного тексту «**main.s**» знаком  також виділено рядок поточної команди. |

* + 1. Розглянемо синтаксис і виконання трьох команд, що є у 20-му, 21-му та 27-му рядках тексту програми.

**Команда у рядку 20** має мнемоніку **LDR**(*Load Register - Завантажити регістр* )*.*

Згідно з документацією [2, 4], команда **LDR**  має дві синтаксичні форми «нормальну» і «альтернативну»:

**LDR** <Rt>, <Label> ; Normal form

**LDR** <Rt>, [Rn,#+/- imm] ; Alternative form

і, обидві дозволяють завантажити у регістр **вміст комірки** **пам’яті** адресу якої вказано двома способами:  
‒ прямою адресацією (*direct addressing*) – тут другий операнд є ім’я комірки (Label - мітка);  
‒ непрямою регістровою адресацією (*indirect registers addressing*) – тут другий операнд завдає адресу комірки як суму вмісту **Rn** і безпосереднього значення **imm** (*immediate*).

Зокрема, як що у якості регістру Rn обрати регістр PC, то отримаємо так звану «відносну» адресацію, тобто адресацію відносно адреси поточної команди ‒ тут адреса комірки буде задаватися сумою **РС +** **imm**, де **imm** дорівнює відстані від адреси поточної команди до адреси комірки-джерела.

Однак у 20-му рядку вихідного тексту програми застосовано інша синтаксична конструкція, а саме:

**LDR R0, =\_\_main,**

яка має означати дію ‒ «Загрузити у регістр R0 адресу комірки, помічену міткою **\_\_main**».

Про це має свідчити і підказка «**\_\_main = 0х0800000С**», яку ми побачимо, як що наведемо вказівник мишки на операнд <**=\_\_main>** в цьому запису (див. примітку у нижній частині малюнку 28), а також і порівняння приведеного там значення з дійсною адресою команди, поміченої цією міткою. Щоб дознатися цієї адреси, загляньте у вікно «Disassembly», і «прокрутить» його до певного рядка.

Насправді запис у вихідному тексті не є командою асемблера, а є однією з його так званих «*псевдокоманд*». Її призначення – створити регістр R0 *показчиком* на комірку з ім’ям **\_\_main**, завантаживши в нього **адресу** поміченої комірки.

***Тут треба сказати*** *як взагалі в асемблері* ***armasm****, що підтримується IDE Keil µVision5, організовано робота з покажчиками (pointers).*

*Адреси всіх помічених комірок, доступ до яких у програмі здійснюється через покажчики, розміщуються у сегменти коду, зразу за коміркою з останньою командою програми. Указані вище псевдокоманди типу* **ldr Rt, = <label>** *перетворюються асемблером у дійсну команду* **ldr** *«альтернативної форми» з відносною адресацією* **ldr Rt, [pc, #imm]***, де зміщення* **imm**  *розраховується як різниця між адресою комірки, в яку записане значення мітки, і текучим значенням регістру PC.*

Дійсно, як що у вікні Disassembly розглянути рядок з командою, що відповідає псевдокоманді   
 **LDR R0, =\_\_main** (виділено жовтим у верхньої частині рис. 28), то побачимо, що асемблер перетворив її у команду **LDR R0, [pc, #4]**. При чому, запис у полі коментарю **@0х08000010** указує, що у регістр **R0** буде завантажено **вміст** (про це говорить ознака **@**) комірки з адресою **0х08000010**. Точніше, оскільки регістр **R0** 32-бітний, то у нього буде завантажено 4-байтне слово, яке розміщене у комірках з адресами **0х08000010 ÷ 0х08000013.** Ці комірки показано у двох нижніх рядках вікна Disassembly (виділені синім). Саме сюди двома директивами DCW (*Define Constant Word*) асемблер помістив 4-байтний покажчик на мітку **\_\_main**. Із урахуванням прийнятого у архітектурі ARM способу розміщення багаторозрядних даних, який називається ***little endian****[[13]](#footnote-13)****,*** вміст цих комірок означає **0х0800000D.**

**Але, чому це відрізняється від очікуваного значення 0х0800000С (?)**, замислитеся ви! Причина тут та ж, що і у різниці між значенням стартової адреси програми **0x8000008,** яка завантажується у програмний лічильник PC (R15), і значенням вектору скидання Vector RESET = **0x8000009**, який записано у комірках з адресом **0x8000004÷0x8000007** (див. 3 і 4 рядки у вікні Disassembly, рис.28).   
У обох випадках значення адреси у сегменті коду і покажчика на цю адресу відрізняються молодшим бітом, а саме, у покажчику на адресу він дорівнює «1», а адреси команд завжди парні, тобто мають у молодшому біті «0». Про це правило ми вже згадували на сторінці 9, але повторимо його тут ще раз.

***Молодші біти*** *значень векторів (і взагалі будь яких покажчиків адрес у Code-сегменті) повинні містити «1» («одиницю»), що вказує процесорові на застосування набору команд* ***Thumb/Thumb-2****. Якщо ж цей біт окажеться скинутим у «0», то процесор зробить спробу переключитися до виконання команд з набору* ***ARM.*** *Це* *цілком природно для ранніх моделей ARM-процесорів, в яких застосовувалися різні набори команд. Але ж у процесорів Cortex-M, які виконують лише команди* ***Thumb/Thumb-2****, це недопустимо і визве генерацію виключення відмови (Hard Fault).*

До іншого непорозуміння може призвести спроба самостійно розрахувати адресу комірки, до якої звертається команда, що розглядається:



Дійсно, як що вважати, що регістр PC(R15) при виконанні поточної команди має значення її адреси, тобто   
**PC = 0х8000008**, то, додавши зміщення **imm = 4**, отримаємо адресу (**0х8000008+4= 0х800000С**).

Але ж ми щойно показали, що команда звертається до комірки **0х8000010**. У чому річь**?**

Тут важливо нагадати, що MCU Cortex-M3/M4 мікроконтролерів STM32 мають в своєму складі 3-х ступінчастий конвеєр: **fetch**🡪**decode**🡪**execution** (рис. 29). Тобто, під час дешифровки однієї команди, виконується вибірка наступної.

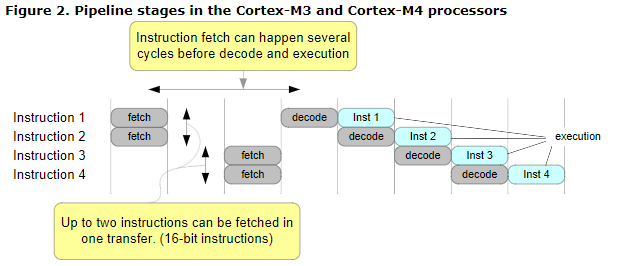


Рисунок 29 − ***3-х ступінчастий конвеєр в архітектурі Cortex-M3 та Cortex-M4***

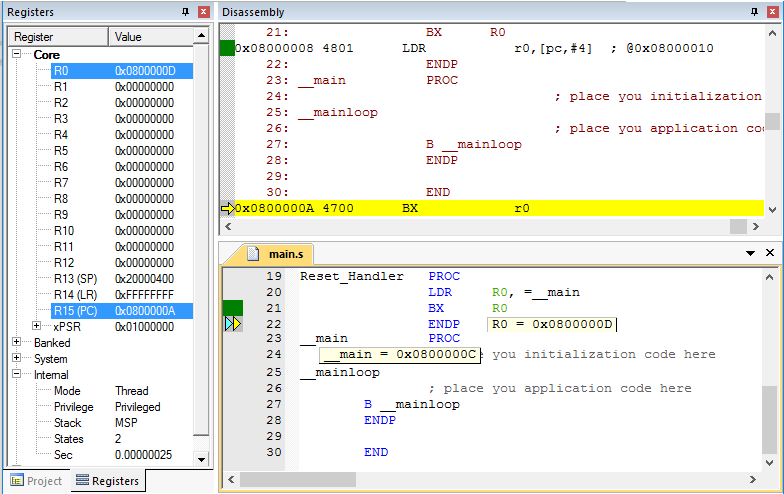
У фазі ***fetch*** за одну транзакцію з пам’яті витягуються (одночасно) одну 32-бітову, або дві 16-бітові інструкції.

Програмний лічильник при цьому збільшується на 4 і, таким чином, адреса, до якої звертається команда,   
буде визначатися так:

**РС+4+imm = 0х8000008+4 +imm = 0х8000010**

* + 1. Натискаючи кнопку , або «гарячу клавішу» <**F11**> (див. рис.28), крок за кроком (**Step one Line ̶** «по командно» ) виконуйте програму, відслідковуючи змінення значень регістрів, текучих адрес команд (РС) та інше.

Після виконання першої команди у вікні будемо мати таке (рис. 29)



? Поясніть розліч

Рисунок 29 − ***Вікно Debugger після виконання першої команди***

Бачимо, що вміст програмного лічильника R15(PC) збільшився на довжину виконаної команди PC=PC+2 = **0х0800000A** і указує на наступну команду, якою є команда переходу:

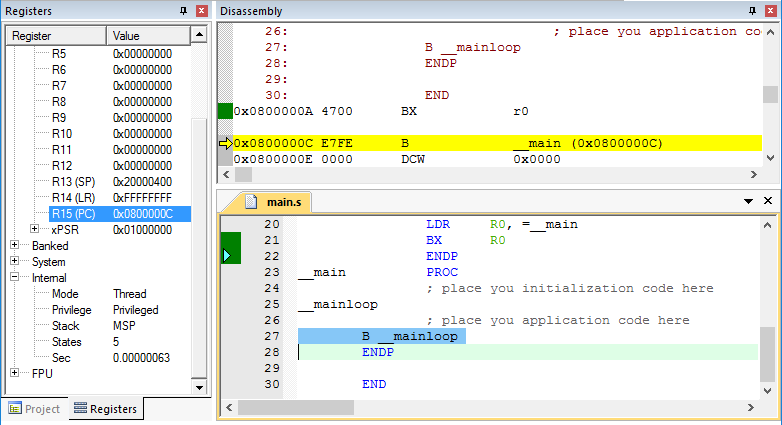
**BX R0 ;** «Перейти до адреси, на яку вказує регістр R0»

Згідно документації [4, c. 141] ̶ це є команда типу Branch indirect and Exchange, загальний синтаксис якої такий :

**BX{cond} Rm** ,   
 де cond(condition) – можливий суфікс, який завдає умову виконання команди;  
 Rm ̶ синтаксичне позначення регістра-покажчика адреси при непрямій адресації пам’яті  
 (*indirect addressing* ).

У вікні Registers бачимо (див. рис.29), що вміст регістру **R0** дорівнює **0x0800000D**, тому з огляду на зазначену раніше особливість значення покажчиків, команда передасть управління на адресу **0x0800000C**, тобто на мітку **\_\_main**. Після виконання команди програмний лічильник повинен мати значення 0x0800000C.

* + 1. Виконайте цю команду (/ <**F11**>). Стан програми буде такий, як на рис.30:



Тривалість виконання двох команд

Рисунок 30 − ***Debugger після виконання другої команди***

* + 1. Наступною стане команда безумовного переходу

**B \_\_mainloop ;** «Повернення на мітку \_\_mainloop»

Це команда типу Branch immediate – перейти по безпосередньому значенню. Її загальний синтаксис такий (див. [4, c.141]):

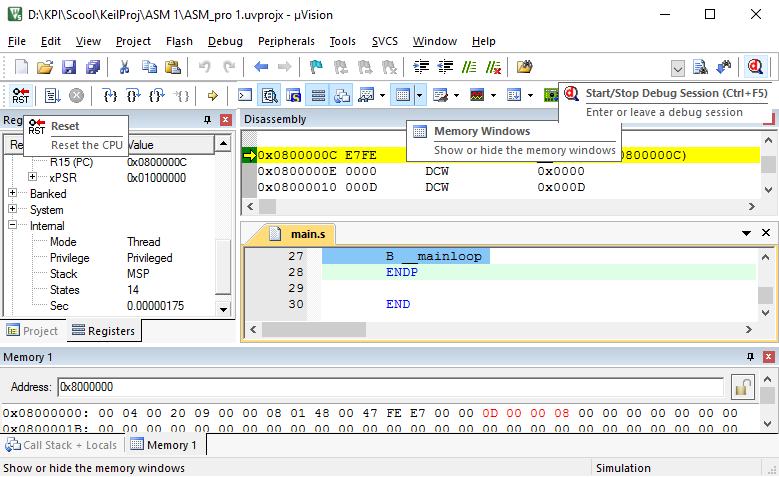
**B{cond} <label> ,**

тобто при наявності суфіксу **cond** , команда може завдавати умовний перехід.

Але у даному випадку перехід безумовний. Оскільки ж між мітками **\_\_main**  і **\_\_mainloop** поки що немає ні однієї команди, то асемблер побудував команду, яка передає управління на мітку **\_\_main**. При цьому поточна адреса команди (0х0800000С) збігається з адресою призначення (0х0800000С), тобто команда «зациклюється» сама на себе.

* + 1. Подальше натиснення клавіши <**F11**> не призведе ні до яких змін, за виключенням збільшення кількості командних циклів (**States**) та загальної тривалості виконання програми (**Sec**).

На рис. 31 показано стан програми після трьох повторень виконання останньої команди.



Pointer to \_\_main

B \_main

Stack Top

Vector Reset

BX RO

LDR R0

Захист доступу до пам’яті «Знято»

Рисунок 31 − ***Заключний стан програми***

Як бачимо значення вмісту PC, тобто адреса команди не змінюється, а кожний крок лише додає до загальної тривалості виконання програми (14 – 5)/3 = 3 такти. При встановленої частоті Fclk=8MHz тривалість одного такту складає 1/8×10-6 s, тобто 0,125 µs, що збігається із розрахунком на основі даних Debugger (1,75×10-6 / 14 = 0,125×10-6). Значення показників **States** і **Sec** дозволяють проводити необхідні розрахунки. Наприклад, тривалість циклу команди B (*Branch immediate*) займає 3 такти (14 – 5)/3 = 3, що становіть 0,125\*3 ≅ 0,375 мікросекунди і збігається із розрахунками на основі показів Debugger:  
 (175 – 63)×10-2/3 ≅ 37,33\*10-2 ≅ 0,373µs.

На рисунку, у нижньої частині вікна µVision/Debug показано вікно “**Memory 1**”, яке можна встановити за допомогою команди **View**🡪**Memory Windows**🡪**Memory 1**, або натиснувши відповідну кнопку у центрі лінійки інструментів. Щоб відобразити тут певну ділянку пам’яті треба, знявши попередньо захист ( ), ввести у поле «**Address:**» початкову адресу цієї ділянці. На рисунку виділено 14 байт з адреси 0х08000000, (тобто вся програма: Vector Table + 3 команди).

Виконання програми і її трасування (налагодження) можна відновити з початку. Для цього достатньо натиснути кнопку **RST**(Reset the CPU) у лівому куті лінійки інструментів.

* + 1. Завершить роботу Debugger, натиснувши кнопку  у правому куті лінійки інструментів, і закрійте проект командою Project🡪Clouse Project (рис. 32)

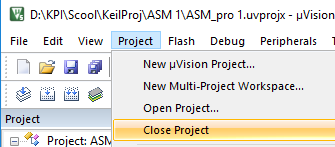


Рисунок 32 − ***Закриття проекту***

На цьому робота з вашим першим проектом-шаблоном для STM32 на асемблері завершено.

* + 1. Збережить файл main.s у папці Source

# Дослідження системи команд STM32F4xx (cortex-m4)

У цієї частині роботи ви зробите декілька вправ з редагування і налагодження програми безпосередньо на мікроконтролері оціночної плати. Виконуючи їх, ви ознайомитися із директивами визначення і розміщення даних у різних розділах пам’яті та синтаксисом і дією команд з їх обробки. Для закріплення матеріалу виконаєте індивідуальні завдання ([Додаток 1](file:///C:\Users\Kornev\AppData\Roaming\Microsoft\Word\Додаток%201.%20ПРАКТИЧНІ%20ЗАВДАНHЯ.docx)).

# Язик асемблера: основи синтаксису

У даному пункті ви ознайомитесь із базовими синтаксичними конструкціями язика асемблера ARM[[14]](#footnote-14). Більш детально це надано у документах з архітектури ARM процесорів [3] та їх програмуванню [4]. Тут у цілому будемо притримуватися порядку і стилю викладення цього матеріалу, що прийнято у книгах Джозефа Ю. (Joseph Yiu) [2, розділ 4] та [6, chapter 5]. Стислу, але ж вичерпну довідкову карту ([Quick Reference Card](file:///C:\Users\Kornev\AppData\Roaming\Microsoft\Word\Додаток%202.%20Quick-Reference_Instruction%20Set%20Card%20(QRC0001_UAL).pdf) ) з системи команд надано у [Додатку 2](file:///C:\Users\Kornev\AppData\Roaming\Microsoft\Word\Додаток%202.%20Quick-Reference_Instruction%20Set%20Card%20(QRC0001_UAL).pdf). Ознайомтеся із цими матеріалами до виконання вправ цього розділу.

* + 1. Ось загальний формат рядку програми на асемблері, що містить   
       **команду (***instruction***)**:

<label>  
 <**op-mnemonic**> <s><cond> operand1, operand2, … ; comment

**директиву** (*directive*):  
< label >  
 <**mnemonic**> {operands} ; Сomment

**УВАГА!** ARM-процесори ̶ типчині RISC-процесори із так званою «*Load-Store архітектурою*», тому тільки інструкції загрузки **LDR** (*Load Register*) та збереження **STR**(*Store Register*) мають доступ до пам’яті. Інструкції ж з обробки даних виконують операції виключно із вмістом одного або декілька регістрів.

**Операндами команд** зазвичай є ARM-регістри[[15]](#footnote-15), але ж ними (принаймні, у якості operand 2) також можуть бути константи (*const*) та (або) позначення специфічних логічних операцій (*shifters*) , якими модифікують operand2, отримуючи, так званий  *Flexible second operand* («Гнучкий другий операнд»), перед його застосуванням у основній операції команди, яку завдає <op-mnemonic>.

**Результат операцій** обробки даних, або читання (Load) пам’яті, (за винятком так званих *операцій множинного завантаження* - *multiple load instructions*), записується у регістр-призначення <Rd>(*destinations register*), позначення якого ставлять попереду інших операндів, тобто зразу після <op-mnemonic>.  
Але ж у командах запису у пам’ять (Store), (за винятком команд множинного збереження - *multiple store instructions*), першім операндом є регістр, вміст якого записується у пам’ять, тобто регістр-джерело (Source register).

**Таким чином**, порядок, кількість і спосіб запису операндів залежить від типу операції, яку завдає інструкція

**Необов’язкові** (*options*) елементи в структурі команди, так звані - **суфікси (***Suffixes*) <S>|<cond>- завдають *модифікації* команд, що направлені на придання командам деяких особливих властивостей, як-то: вплив команди на стан регістру прапорців APRS (суфікс S), умова виконання команди (cond – *condition) –* *EQ* 🡪«дорівнює»*|NE* 🡪«не дорівнює»*|LT* 🡪«менше»*|GT* 🡪«більшее» та ін., а також змінення розрядності команди ( *N* 🡪 Narrow - «вузька, 16-бітна, Tumb» , *W* 🡪 Wide – «широка, 32-бітна, Tumb-2).

* + 1. Архітектура набору інструкцій (**ISA** - *Instruction Set Architecture*) у процесорах Cortex-M називається "Thumb ISA", яка заснована на технології Thumb-2®, що підтримує суміш 16-розрядних та 32-розрядних інструкцій[[16]](#footnote-16). Для підтримки набора команд Thum-2, компанія ARM розробила уніфіковану мову асемблера (*Unified Assembler Language* - **UAL**). ЇЇ застосування полегшує перехід між кодами ARM и Тumb, оскільки тут команди з обох наборів мають однаковий синтаксис:

**АDD RO, R1 ;** RO = RO + R1, це традиційний синтаксис Thumb

**АDD RO, RO, R1** ; Еквівалентна команда, з синтаксисом UAL

На застосування синтаксису UAL вкаже директива **THUMB** на початку заголовку тексту програми асемблеру.

* + 1. Слід відзначити, що деякі традиційні 16-бітні Тumb інструкції впливають на прапорці у регістрі стану програми xPSP(**APSR**) незалежно від наявності суфіксу **S**. Але ж у синтаксисі UAL вплив команди на прапорці стану завжди визначається цим суфіксом. Наприклад:

**AND**  **R0, R1 ;** RO = RO&R1, це традиційний синтаксис Thumb

**ANDS**  **R0, R1 ;** Еквівалентний синтаксис UAL

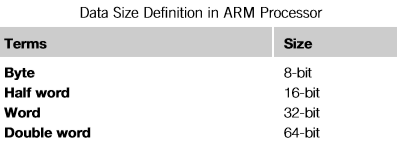
* + 1. Щоб явно вказати розрядність команди в UAL є спеціальні суфікси:

**ANDS**  **R0, #1** ;16-бітний синтаксис Thumb за замовчанням

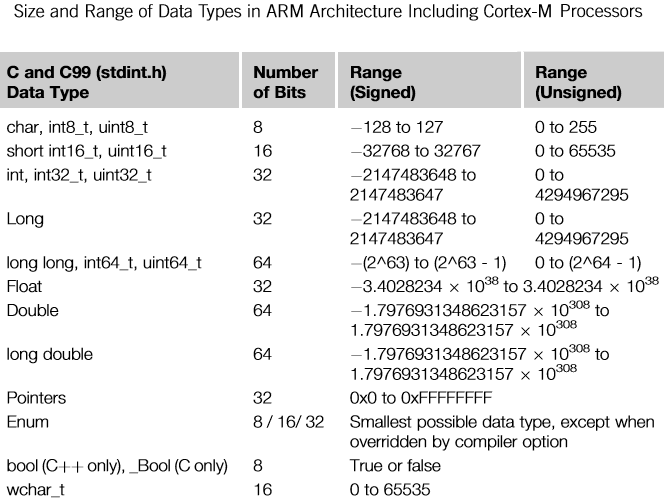
**ANDS.N**  **R0, #1** ;16-бітна Narrow (вузька)Thumb-команда

**ANDS.W**  **R0, #1** ;32-бітна Wide (широка) Thumb-2 команда

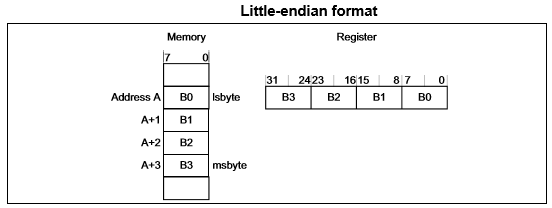
* + 1. Процесор Cortex-M підтримує данні таких розмірів:



Нижче вказано типи даних в С-програмах, розмір і діапазони їх значень:



* + 1. Порядок байт при розміщенні багато-байтних операндів у пам’яті і у регістрах за замовчанням прийнято *little-endian –«молодші розряди попереду»*:



* + 1. Більшість 16-бітних інструкцій можуть звертатися лише до так званих молодших регістрів R0 – R7, тоді як 32 бітні команди не мають такого обмеження. Однак для деяких команд не допускається застосування регістра лічильника команд R15(PC). Докладніше про це розкрито у [2] (розділ А.4.6).
    2. Для визначення у коді програми однобайтних констант, наприклад, символів, застосовують директиву **DCB** (*Define Constant Byte*), а для завдання 32-бітних констант ̶ директиву **DCD** (*Define Constant Data*,   
       або *Define Constant Double Word*):

**LDR R3, =My\_Number** ; Отримати адресу константи **My\_Number**

**LDR R4, [R3]** ; Считати значення 0х12345678 у R4

...

**LDR Ro, =HELLO\_TXT** ; Отримати початкову адресу рядку  
 ; **Hello\_TXT**

**BL PrintText** ; Визвати функцію **PrintText**; для відображення рядку

**My\_Number  
 DCD 0x12345678**

**HELLO\_TXT**

**DCB “Hello\n”,0** ;ASCIIZ-рядок, що завершується «нулем»

Двохбайтні (16-бітні) числа завдаються директивами **DCW** (*Define Constant Word*)[[17]](#footnote-17):

**DCW 0x5678  
 DCW 0x1234**

Як що розглядати два останні числа як єдине 32-бітне число, то отримаємо **0x12345678.**

* + 1. На завершенні відмітимо, що процесори Cortex-M підтримують звернення як до так званих «вирівняних даних» так і до «невирівняних даних» (див. ілюстрацію нижче). У останньому випадку вдається більш компактне розміщення даних, але звернення до невирівняних даних більш тривале, оскільки у деяких випадках потребує двох транзакцій.

32-бітні данні типу **long**, **int**, **uint32\_t** 8-бітні данні типу **char**, **int8\_t**, **uint8\_t**

16-бітні данні типу **uint16\_t**  комірки пам’яті, що не застосовуються



0x100C

0x1008

0x1004

0x1000

Byte

Half Word

Half Word

Byte 4

Byte 3

Byte 2

Byte 1

Byte 2

Byte 1

Half Word

Half Word

Byte

Word

Half Word

**Вирівняні данні** **Невирівняні данні**

# способи завдання даних і доступу до них

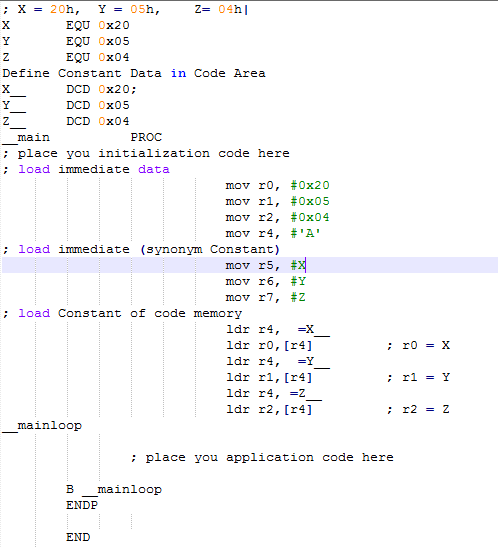
Приклади цього і наступного підрозділів побудовані як пояснення до виконання відповідних вправ з індивідуальних завдань до цієї лабораторної роботи (див. Додаток 1)[[18]](#footnote-18).

Припустимо нам потрібно обчислити такий арифметичний вираз (без його спрощення):

**Q = (X + Y) + (Y - Z) – (X - Y)** ; де X = 20h, Y = 05h, Z= 04h

Розглянемо спочатку різні способи завдання *даних* і *констант* і доступу до них у командах.

* + 1. Відкриємо вихідний файл main.s (наприклад, у редакторі Notepad++) і добавим у нього директиви завдання *констант* (тобто даних, які не повинні змінюватися при виконанні програми) і команди доступу до них (рис. 33).



Ці константи визначені у програмної **Flash-пам’яті,**  потрапляють до неї при завантаженні програми і зчитуються командою LDR

Ці константи створюються при трансляції програми і доступні командам як безпосередні (***immediate)*** операнди

Рисунок 33 − ***Способи завдання констант і доступ до них***

З метою навчання тут розглянуто декілька способів завдання і доступу до констант, а саме:  
 -- безпосереднє (*immediate*) завдання константи у команді **mov Rd, #const** ;  
 -- завдання константи у Flash memory і завантаження її у регістр командою **LDR Rn, [Rs]**

* + 1. Збережіть файл у папці Source під іншою назвою, наприклад, **Lab1\_1.s** і додайте його до проекту. Для цього :  
        - відкрийте вже існуючий проект (Project🡪Open Project🡪ASM\_1/ASM\_pro1);  
        - видалите з нього «старий» вихідний файл **main.s** командою **Remove File ‘main.s’** із контекстного меню (рис.34)   
        - додайте до проекту (див. рис. 10 і рис.11) файл **Lab1\_1.s** (рис. 35)

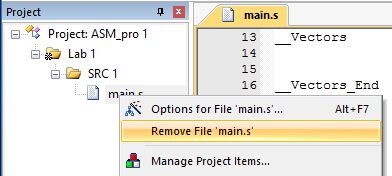


Рисунок 34 − ***Видалення вихідного файлу с проекту***

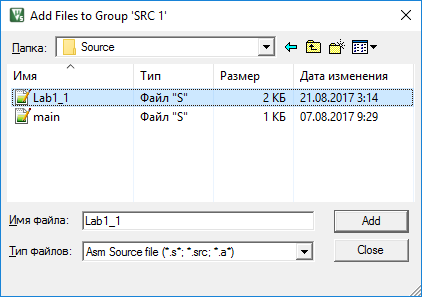


Рисунок 35 − ***Додавання нового вихідного файлу до проекту***

Отримаємо наступне (рис.36):

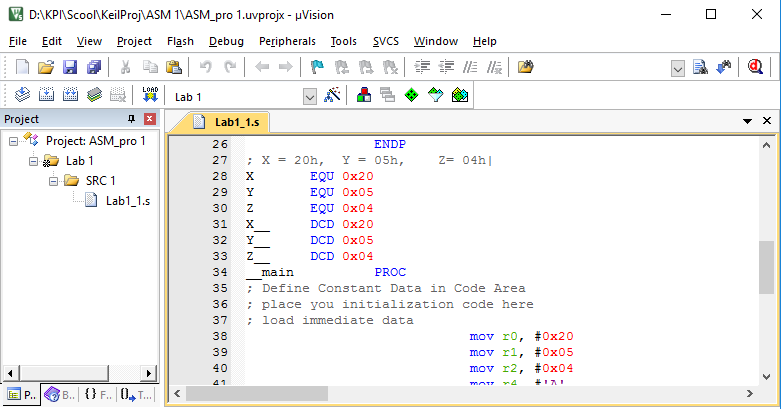


Рисунок 36 −***Оновлений проект.Налаштування***

На цей раз випробування дії програми будемо проводити на реальному мікроконтролері, тому підключить до комп’ютера лабораторний стенд - оціночний модуль (STM32Discovery/NUCLEO) і здійсните налаштування програми, що показані далі.

* + 1. У правому полі вкладки Debug (рис. 37) оберіть програму загружчик-налагоджувач ST-LINK, не забутьте зняти прапорець “Run to maim” і натисніть кнопку Settings щоб продовжити налаштування (рис.38,39).

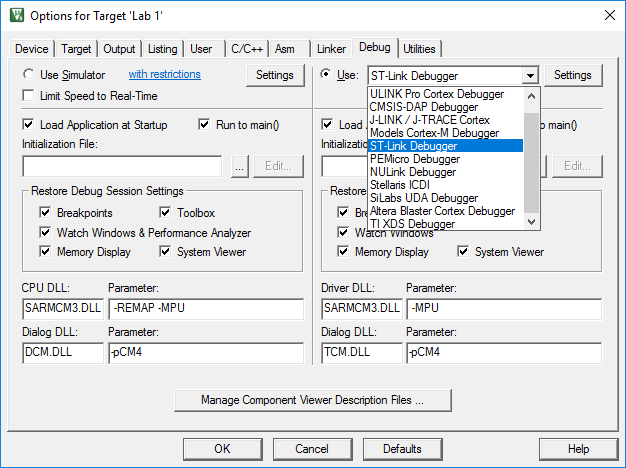
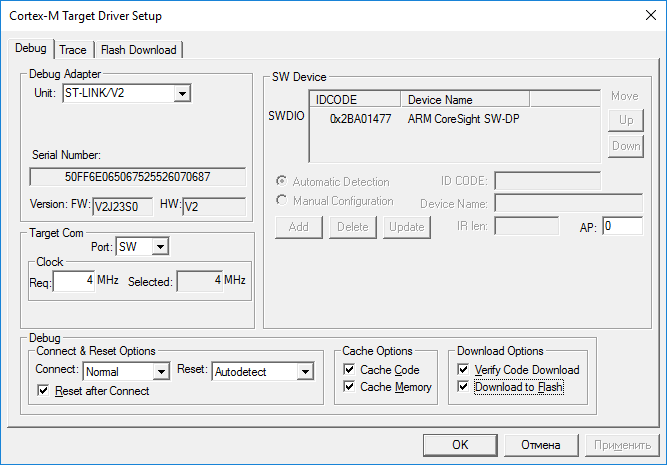


Рисунок 37 − ***Встановлення параметрів Debug (крок 1)***



Як що з’єднання здійснилося правильно, то з’явиться інформація про атрибути підключеного пристрою, а на платі засвітяться два червоні світлодіоди.   
Оберіть порт загрузки («прошивки») SW і   
параметри завантаження: - у Flash та із здійсненням перевірки коду при загрузці.

Рисунок 38 − ***Налаштування Debug (крок 2)***

Не забутьте натиснути кнопку <OK> у останньому вікні, шо б зберегти параметри налаштування (рис.38).

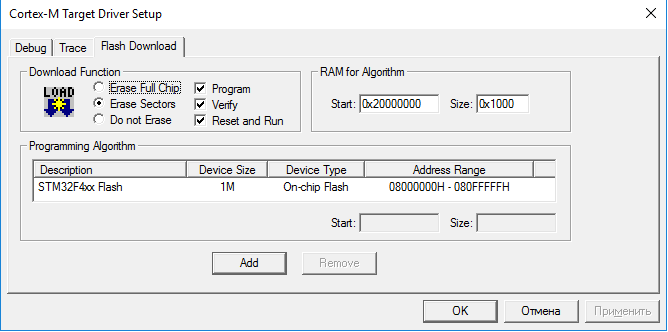
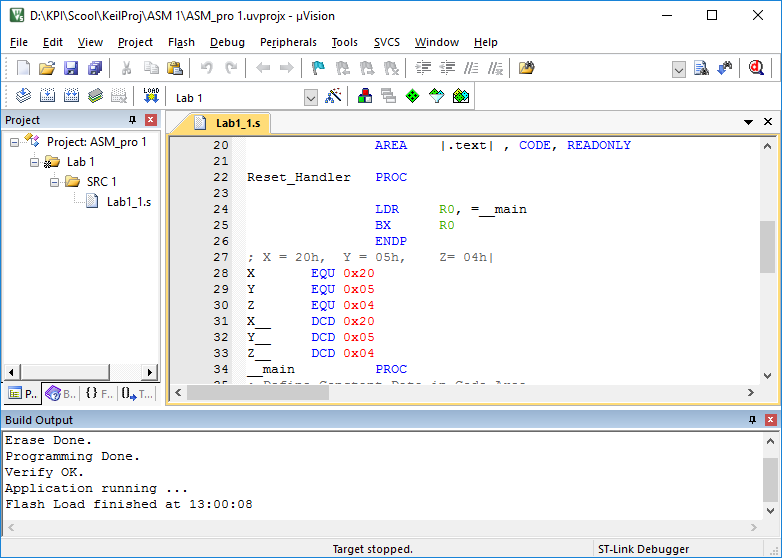


Рисунок 39 − ***Налаштування Debug (крок 3)***

* + 1. Запустіть процедуру трансляції і лінкування проекту (). Як що помилок не зроблено, то у вікні статусу Build Output отримаєте такий результат (рис.40).
    2. Натисніть кнопку загрузки програми у мікропроцесор  і медитіруйте. Після успішного завантаження, в ході якого у вікні стану будуть з’являтися оповіщення (див. рис.40), на платі світлодіод змінить колір з червоного на зелений.



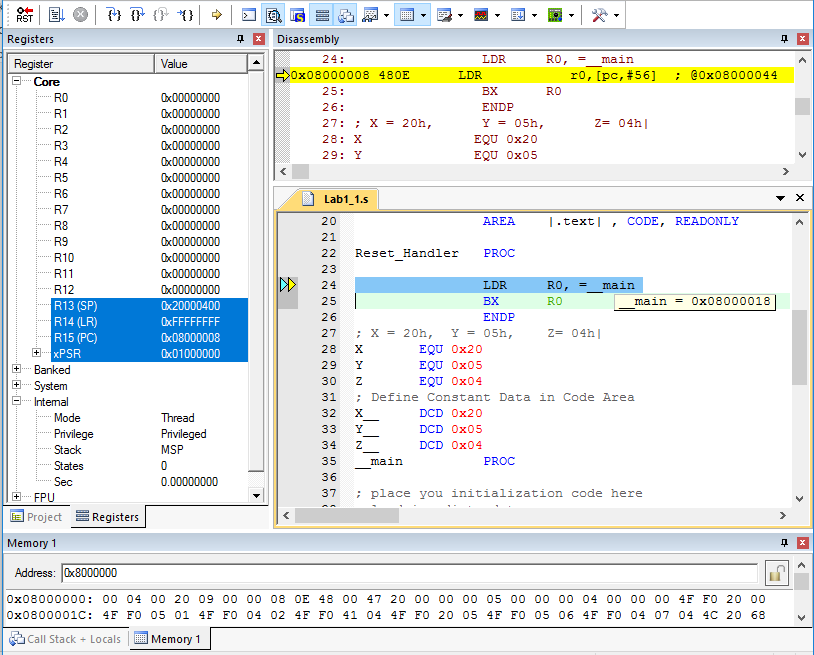
Попередню програму стерто. Нову – завантажено і

перевірено.  
Загрузку завершено у 13:00:08

Рисунок 40 − ***Проект побудовано і завантажено успішно***

***Вітаю! Ви тільки що вперше успішно запрограмували мікроконтролер STM32!***

* + 1. Натисніть кнопку запуску Debugger та продовжте дослід структури програми і її виконання безпосередньо мікроконтролером (рис.41).



Z\_\_

Reset Handler

X\_\_

Y\_\_

Vector Table

Рисунок 41 − ***Старт Debug***

Перше на що звертаю вашу увагу, це те що стартовий стан регістрів не змінився (програма і стек розміщено там же), однак адреса процедури \_\_main збільшилася на 12 (08000018h – 0800000Ch). Чому? – поясніть.

* + 1. Натискаючи кнопку (<F11>) крок за кроком виконуйте («трасуйте») програму, відслідковуючи дію команд безпосередньої загрузки констант **mov rd, #const** у регістри R0 ÷ R2, R4 ÷ R7 і потім команд непрямої регістровій адресації **LDR rd, [reg]** у регістри R0 ÷ R2 із застосуванням R4 у якості регістра-вказівника, як це показано на рис.42.

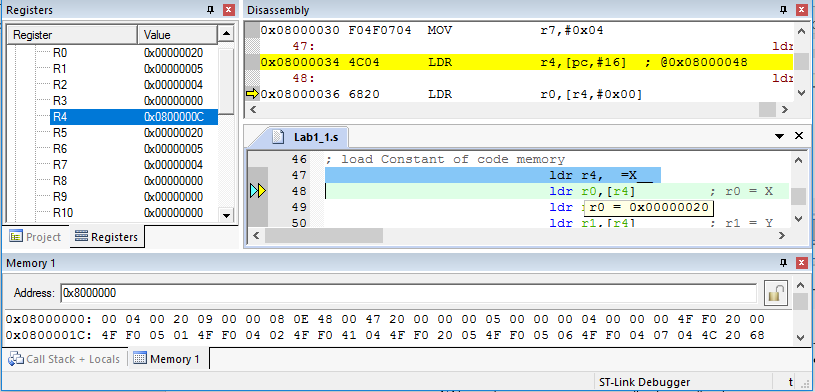


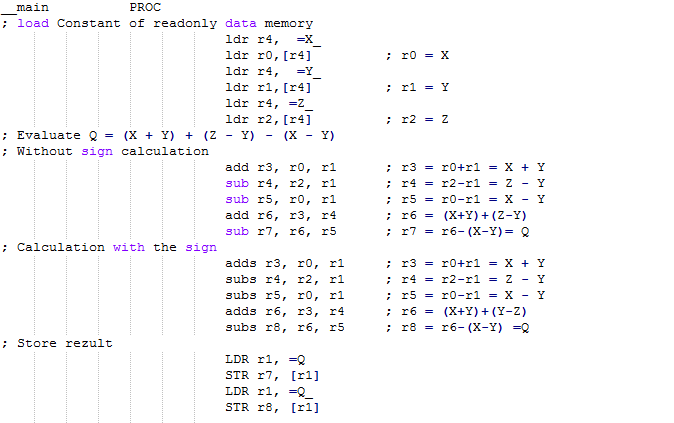
Рисунок 42 − ***Читання константи із сегменту коду командою непрямої адресації***

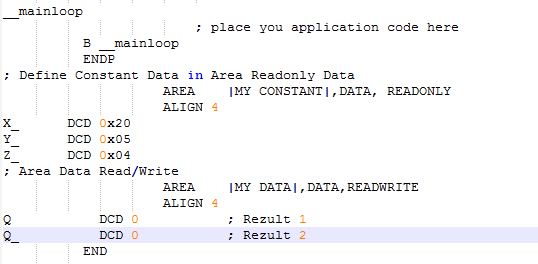
# Дослідження команд обробки даних

Додаймо до файлу команди для обчислення виразу **Q = (X + Y) + (Y - Z) – (X - Y)**

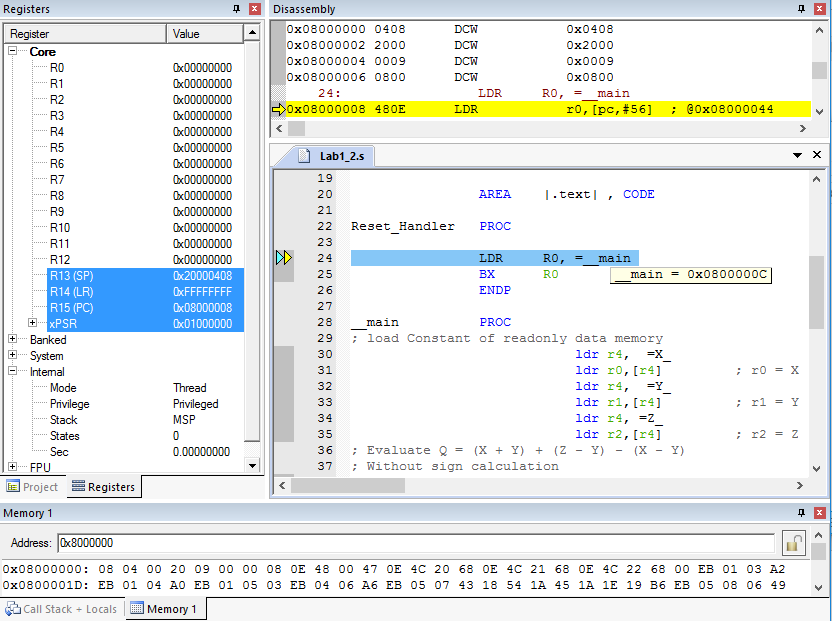
При цьому з навчальних цілей проведемо порівняння команд знаковій та беззнаковій обробки, а також розглянемо способи визначення даних у **ro (*readonly*)** та **rw (*read-write*)** пам’яті SRAM.

* + 1. Вийдіть з режиму налагодження (Stop Debugg), клацнув кнопку .   
       Далі можна діяти двома способами.
    2. Як що необхідні зміни у вихідному файлі невеликі, то його можна відредагувати прямо у Keil µVision. Потім перебудувати ціль Target (Lab 1), натиснувши кнопку 
    3. Як що ж файл потребує великих змін, то краще діяти так же, як і у пп. 2.2.1 і 2.2.2, тобто видалити попередній файл (Lab1\_1.s) з проекту, відредагувати його у Notepad, зберегти (бажано під іншим ім’ям, наприклад , Lab1\_2.s, щоб оставити попередню версію), додати новий файл до проекту і побудувати його заново (). Саме так ми і поступимо зараз.
    4. Нижче показано фрагмент із змінами файлу Lab 1\_2.s , а саме:   
        - команди звернення (читання) констант (X\_ , Y\_ , Z\_) з *readonly data memory*;  
        -два варіанта обчислення виразу із застосуванням команд «арифметиці без знаку» **Q**(**R7**) та арифметиці із знаком **Q\_**(**R8)**; - приклад визначенням сегменту констант Area Readonly Data (**MY\_CONSTANT**) і сегменту даних Area Data Read/Write (**MY\_DATA**):





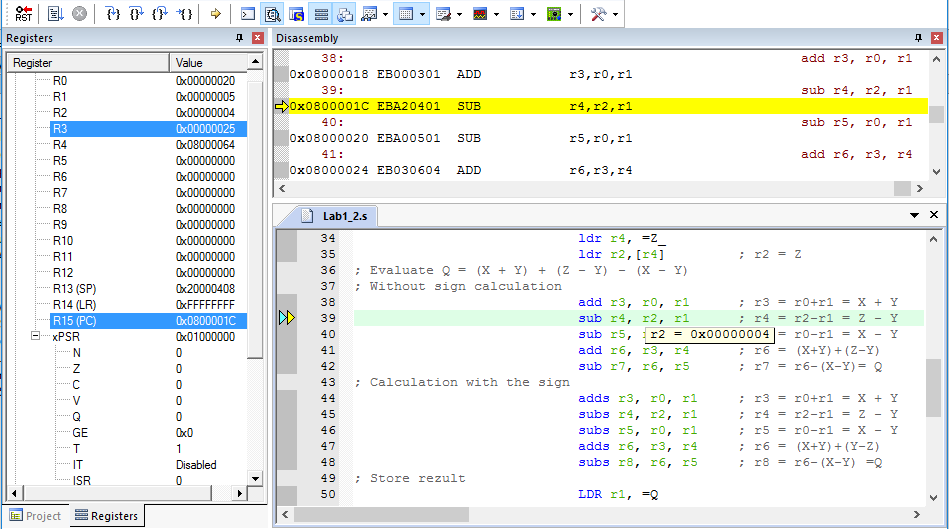
* + 1. Виконайте дії, що описані у п.2.3.4. Загрузіть програму у стенд  і запустіть Debugger. Проаналізуйте стартовий стан регістрів MCU (рис.43).



Вершина стеку піднялася на 8 байтів. Поясніть це

Рисунок 43 − ***Стан регістрів при старті програми Lab1\_2***

* + 1. Виконайте всі команди до початку команд арифметичних обчислень (з 24 по 38 рядки). Для цього установіть курсор на тому рядку до якого потрібно виконати програму (у нашому випадку це рядок 39) і натисніть кнопку  .В результаті отримаєте таке (рис.44)



Прапорці регістру стану програми PSR.APSR

Рисунок 44 − ***Автоматичне виконання програми «до курсору» (Ctrl+F10)***

Оскільки у рядках 38 – 42 варіанти команд «арифметики без знаку», то вони не впливають на стан прапорців APSR. Навіть коли у рядку 39 при відніманні більшого числа (r1=5) з меньшого (r2=4) отримано число 0xFFFFFFFF (-1 у додатньому коді) стан прапорців не зміниться (рис. 45).

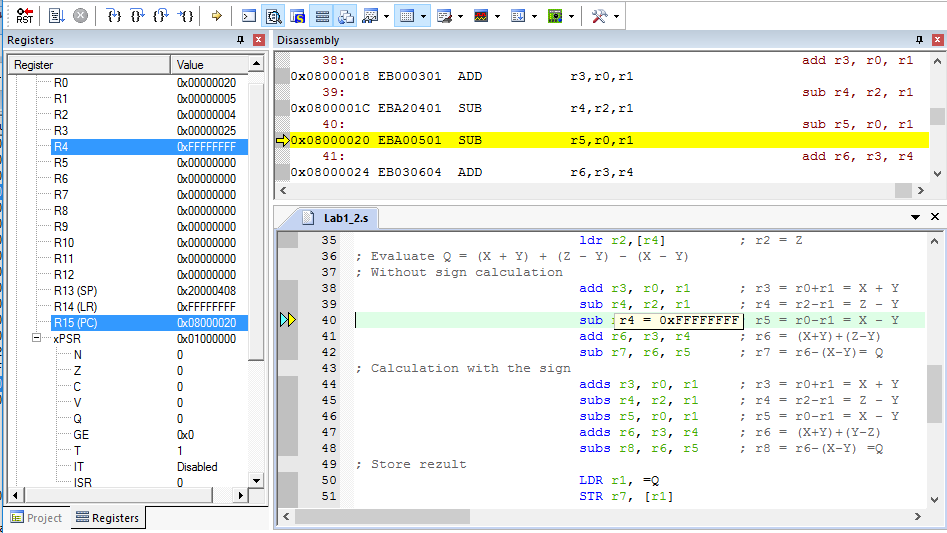


Рисунок 45 − ***Команди, що не впливають на стан прапорців APSR***

* + 1. Трасуючи ж команди у рядках 44 ÷ 48 переконайтесь у тому, що вони впливають на прапорці (рис. 46).

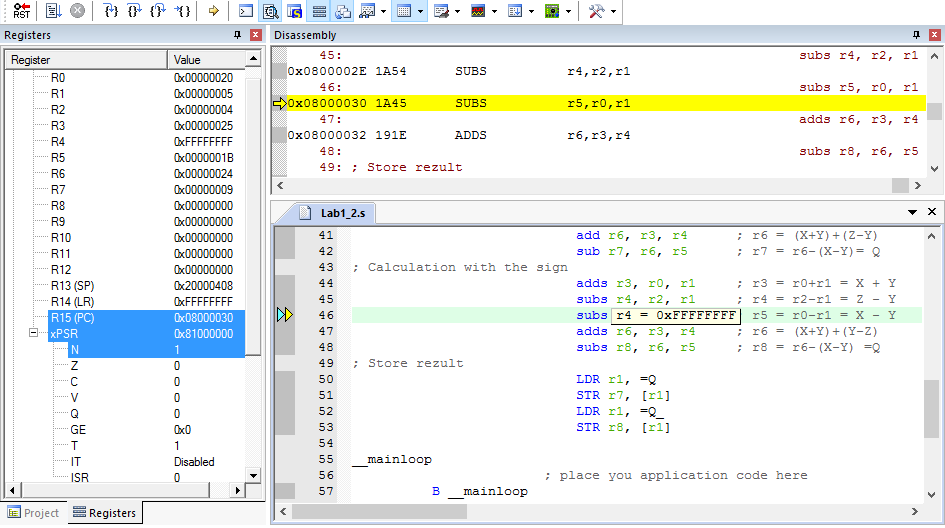
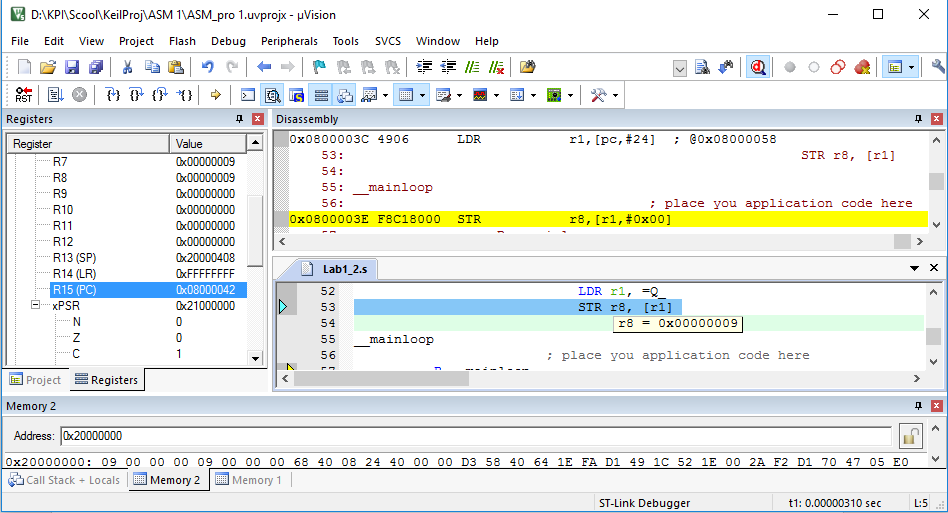


Рисунок 46 − ***Команди, що впливають на стан прапорців APSR***

* + 1. Наприкінці виконайте команди запису результатів обчислень (R7 та R8) у комірки пам’яті даних із іменами Q та Q\_ (рядки 50÷53). Для перегляду пам’яті даних додайте вікно Memory 2 (рис. 47).

На цьому завершимо дослідження програми. Остановіть Debugger і закрійте проект.



Q\_

Q

Рисунок 47 − ***Команди запису у пам’ять***

* + 1. Для закріплення матеріалу виконайте свій варіант індивідуальних завдань 1 і 2 з дослідом команд арифметичної і логічної обробки. З синтаксисом і роботою команд логічної обробки ознайомитися у   
       [2, 4].
    2. Складіть звіт (протокол) з описом програми і етапів її дослідження (відладки).

У роботі 2 ви ознайомитеся із технологією створення проекту, де основна програма написана на Си, а окремі вставки (функції) на асемблері.

1. Призначення вкладок «Devices», «Boards», «Packs» і «Examples» у лівому і правому полі інтуїтивно ясно, до того ж це роз’яснюється у діалозі, що з’являється при запуску Pack Installer поверх його вікна (див. рис. 2).

   [↑](#footnote-ref-1)
2. Лабораторні роботи проводитимуться із застосуванням оціночних модулів (*Evolution board*) STM32F4 Discovery (на основі МК STM32F407VGTx) та STM32F401 Nucleo (на основі STM32F401RET6). Якщо у інших проектах ви застосовуватиме інші мікроконтролери, то ви завжди зможете додати до Keil відповідні бібліотеки, повторюючи п. 1.1.1. і 1.1.2. [↑](#footnote-ref-2)
3. В іншої версії Keil діалогові вікна можуть відрізнятися від показаних тут, але їх призначення і опції однакові [↑](#footnote-ref-3)
4. Як що все це вам вже відомо, то ви можете пропустити дії і матеріал цього пункту, а також трьох наступних підрозділів і зразу ж приступити до виконання вправ другого розділу цієї інструкції.  
   Але ж для тих, хто робить перші крокі у програмуванні для STM або подібних мікроконтролерів із ARM архітектурою, наполягаю ознайомитися із матеріалом і ретельно виконати дії наступних пунктів. [↑](#footnote-ref-4)
5. Для створення вихідних файлів можна застосовувати будь який текстовий редактор (наприклад, [Notepad++](https://notepad-plus-plus.org/download/v7.5.8.html) для відповідної версії Windows). [↑](#footnote-ref-5)
6. Нагадуємо, що контекстне меню до будь якого елементу інтерфейсу Windows-додатку викликається правою кнопкою мишки. [↑](#footnote-ref-6)
7. . Приклад вихідного файлу-шаблону **main.s** ви можете знайти у папці Source поруч із файлом цієї інструкції. Створить і ви у папці проекту ASM\_1 таку ж папку Source - ..\\ASM\_1\Source для розміщення в неї вихідних файлів. [↑](#footnote-ref-7)
8. У випадках, коли різниця між процесорами Cortex-M3 та Cortex-M4 відсутня, або не є суттєвою, будемо говорити просто Cortex-M. [↑](#footnote-ref-8)
9. Більш детально карта пам’яті розглянуто у відповідній лекції [1, Less3], у книзі [2,c.93-98] та у фірмовій документації на процесори Cortex-M [3,Chapter B3-2] та мікроконтролери STM32F4xx [4,c.27-28],  
   [5, 64-67], або у інтернет ресурсах ARM (<http://www.arm.com>) та STMicroelectronics (<http://www.st.com>) . [↑](#footnote-ref-9)
10. Директиви і команди виділено коліром. [↑](#footnote-ref-10)
11. Як що ця кнопка не активна, то можливо ви не перемкнули Debugger у режим «програмної емуляції» (Use *Simulator*) (див. рис.22). Тоді виконайте дії пп. 1.5.4, або перейдіть до режиму загрузки і налагодження програми на реальному пристрої (див. розділ 2) [↑](#footnote-ref-11)
12. Натисніть на знак <+>, щоб розгорнути групу, яку згорнуто. Наприклад, у групі xPSR буде показано складові регістри узагальненого регістру стану програми, а саме прапорці N, Z, C,V, Q та інше. [↑](#footnote-ref-12)
13. Більш детально про це дивись у наступному розділі [↑](#footnote-ref-13)
14. Нагадуємо, що деякі конструкції асемблерного коду залежать від фірми-розробника інструментарію (vendor toolchain), який ви застосовуєте для створення програм. Приклади асемблерних програм, що розглянуті у цьому курсі написані в **ARM ® assembler** (***armasm***), який до того ж застосовується у комплекті інструментальних засобів розробки програм KeilTM Microcontroller Development Kit for ARM (MDK-ARM). Асемблери інших компаній-розробників, зроблені на **GNU toolchain** (як, наприклад, Sourse G++ від CodeSourcery), мають (окрім мнемонік інструкцій) інший синтаксис, принаймні що стосується *directives* та інших елементів (deﬁnitions, labeling and comment). [↑](#footnote-ref-14)
15. ARM-процесори ̶ типчині RISC-процесори із так званою «*Load-Store архітектурою*», тому тільки інструкції загрузки **LDR** (*Load Register*) та збереження **STR**(*Store Register*) мають доступ до пам’яті. Інструкції ж з обробки даних виконують операції виключно із вмістом одного або декілька регістрів. [↑](#footnote-ref-15)
16. На відміну від попередніх моделей ARM-процесорів, які для переключення з 16-бітних до 32-бітних ARM-команд потребували окремої процедури, для процесорів Cortex-M така суміш є «природньою» і не потребує такої процедури, тому справляється з цім «з великого пальця»  . [↑](#footnote-ref-16)
17. Тут, проте, позначається розбіжність позначень, прийнятих в асемблері ARM і описі ARM архітектури. [↑](#footnote-ref-17)
18. Раджу виконати ці приклади перш ніж братися за вправи індивідуальних завдань. [↑](#footnote-ref-18)