**Лабораторна робота 2**

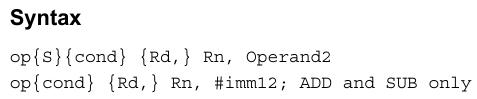
**Тема:** Основні інструкції 32-бітного ARM процесора для мікроконтролерів

**Мета:** Навчитися використовувати асемблерні інструкції ядра Cortex-M4, працювати з процедурами і базово зрозуміти архітектуру ядра.

**1 Скороченні теоретичні відомості**

**1.1 Основні мікрокоманди асемблера в ARM архітектурі:**

1. **ADD, ADC, SUB, SBC, and RSB**



cond -- умовний код

Rd -- регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn

Rn -- регістр, в якому знаходиться перший операнд

Operand2 -- константа або регістр

#imm12 -- значення з проміжку 0-4095

Мікрокоманда **ADD** додає значення operand2 або #imm12 до значення в Rn.

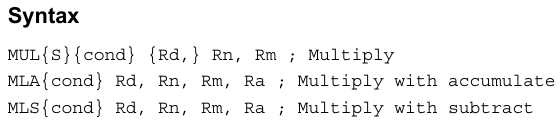
Мікрокоманда **ADC** додає значення operand2 до значення в Rn, враховуючи флаг переносу.

Мікрокоманда **SUB** віднімає значення operand2 або #imm12 від значення в Rn.

Мікрокоманда **SBC** віднімає значення operand2 або #imm12 від значення в Rn, враховуючи флаг переносу

Мікрокоманда **RSB** віднімає значення Rn від значення в operand2.

1. **MUL, MLA, and MLS**

cond -- умовний код

Rd -- регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn

Rn, Rm -- регістри, в яких знаходяться операнди

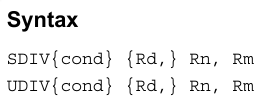
Ra -- регістр, що зберігає значення, до якого додається або від якого віднімається результат

Мікрокоманда **MUL** множить значення в Rn та Rm, молодші 32 біти результату зберігаються в Rd.

Мікрокоманда **MLA** множить значення в Rn та Rm, а потім додає то результату значення з Rd, молодші 32 біти результату зберігаються в Rd.

Мікрокоманда **MLS** множить значення в Rn та Rm, а потім віднімає результат від значення з Rd, молодші 32 біти результату зберігаються в Rd.

1. **SDIV and UDIV**

cond -- умовний код

Rd -- регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn

Rn -- регістр, в якому знаходиться ділене

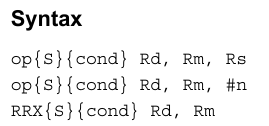
Rm -- регістр, в якому знаходиться дільник

Мікрокоманда **SDIV** виконує ділення зі знаком значення в Rn на Rm, результат зберігається в Rd.

Мікрокоманда **UDIV** виконує ділення без знаку значення в Rn на Rm, результат зберігається в Rd.

Для обох мікрокоманд виконується наступне правило:

Якщо значення в Rn не ділиться на Rm, результат округлюється до 0.

1. **ASR, LSL, LSR, ROR and RRX**

cond -- умовний код

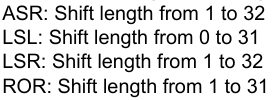
Rd -- регістр призначення (куди зберігається результат). (ОБОВ’ЯЗКОВИЙ!)

Rm -- регістр, що містить значення, яке зсувається

Rs -- регістр, що містить значення довжини зсуву регістра Rm

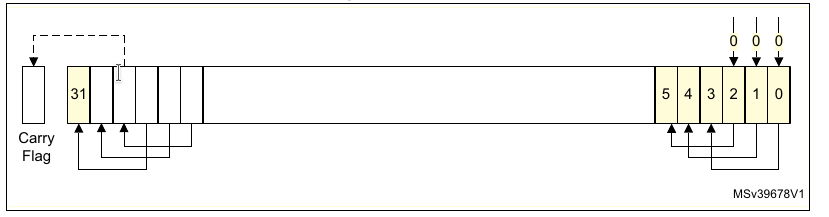
n -- значення довжини зсуву регістра Rm

Допустимі значення n:

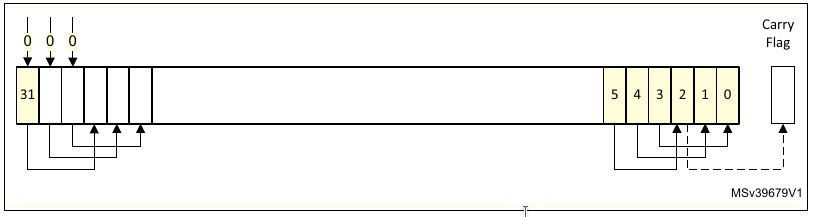




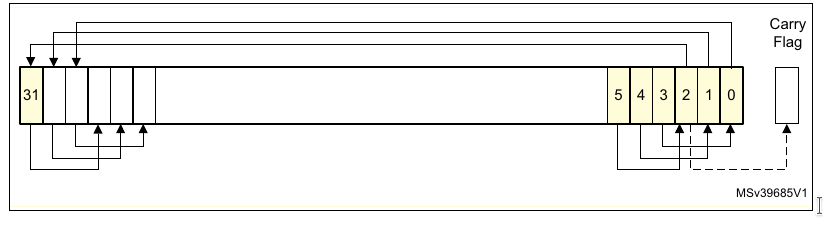
Мікрокоманда **ASR** виконує арифметичний зсув праворуч значення в Rm на Rs, результат зберігається в Rd.



Мікрокоманда **LSL** виконує логічний зсув ліворуч значення в Rm на Rs, результат зберігається в Rd.



Мікрокоманда **LSR** виконує логічний зсув праворуч значення в Rm на Rs, результат зберігається в Rd.



Мікрокоманда **ROR** виконує поворот праворуч значення в Rm на Rs, результат зберігається в Rd.

1. **AND, ORR and EOR**

cond -- умовний код

Rd -- регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn

Rn -- регістр, в якому знаходиться перший операнд

Operand2 -- константа або регістр

Мікрокоманда **AND** виконує побітову операцію І на операндах Rn та Operand2.

Мікрокоманда **ORR** виконує побітову операцію АБО на операндах Rn та Operand2.

Мікрокоманда **EOR** виконує побітову операцію ВИКЛЮЧНЕ АБО на операндах Rn та Operand2.

1. **PUSH and POP**

Push – «заштовхує» регістри в стек

Pop – «виштовхує» регістри із стека

Команди приймають список регістрів. Наприклад:

push {r0, r1, r2}

pop {r0, r1, r2}

**1.2 Основні директиви ассемблера в ARM архітектурі**

.syntax -- Директива **.syntax** дозволяє вибрати синтаксис мови *thumb2*(новий або застарілий). Ми використовуємо останню версію за замовчуванням (*unified*). Директива є архітектурно – залежною(директива має бути використана в кожному асемблерному файлі).

.size -- Вказує скільки місця займають дані, на які вказує певний символ.

Наприклад .size \_\_hard\_reset\_\_, .-\_\_hard\_reset\_\_ обчислить розмір функціі \_\_hard\_reset\_\_ в байтах.

.word -- (.word \_\_hard\_reset\_\_+1) Встановлює комірку пам’яті 32 -бітним числом, що є адресою процедури(переривання RESET)+1. Біт 0 в даному випадку – флажок переходу в режим виконання thumb інструкцій, якби він був поставлений в 0, процесор видав би помилку, оскільки у нас є тільки 1 режим інструкцій(перевірте це).

.thumb --Дає зрозуміти асемблеру, що треба генерувати виключно thumb інструкції процесора.

.global -- За замовчуванням асемблер створює невидимі ззовні файлу символи коли ми створюємо нові процедури, змінні, будь-які інші об’єкти. Щоб процедура могла звернутись з іншого файлу в процедуру в цьому файлі, її потрібно створити видимою за допомогою директиви .global

**1.3 Типи ARM інструкцій**

Існує 2 типа ARM інструкцій:

1. Стандартний ARM набір, що дозволяє швидко виконувати операції.
2. Thumb, що дозволяє значно зменшити розміри виконуваної програми.

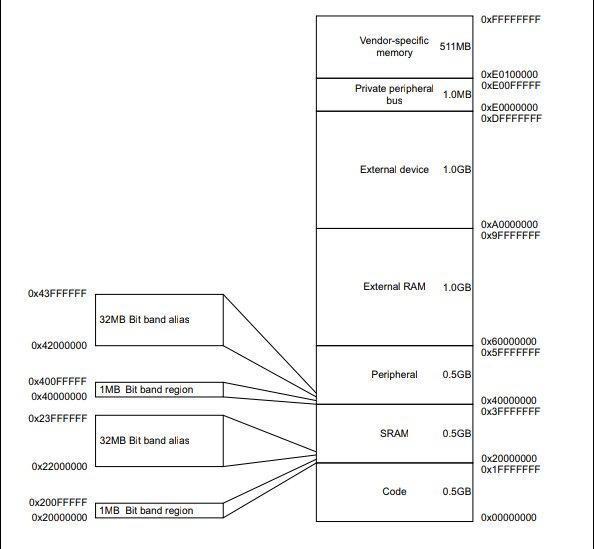
Стандартний ARM набір складається з інструкцій, що мають різну довжину (від 16 біт до 64 і навіть більше), що робить конвеєр в ядрі досить складним, але швидкодія більша ніж у процесорів, що мають тільки thumb набір інструкцій.

Thumb набір – підмножина стандарту, що складається з 16-ти та 32-бітних існтрукцій.

Наприклад, сімейство Cortex-M0 реалізує в основному тільки 32 бітні інструкції.

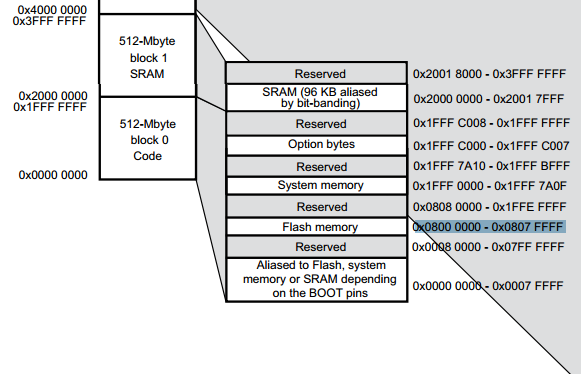
Сімейство Cortex-M4 реалізує повний набір thumb інструкцій, але має тільки thumb режим. Сімейства Cortex-A4 можуть працювати в 2х режимах: thumb та ARM.

**1.4 Адресний простір та робота з пам'яттю**



Це фон-Неймановська архітектура, інструкції можна виконувати і з розділу code і з розділу SRAM ( STATIC RAM), і з розділу External RAM, якщо підключено зовнішню пам’ять. Зверніть увагу на регіон 0x0000 0000 – 0x2000 0000, зазвичай там є доступ до постійної (FLASH) пам’яті.

При чому в STM32 девайсах це адресний простір постійної пам’яті.



0x0000 0000 - 0x0007 FFFF- це *Віртуальна пам’ять*, в залежності від початкової конфігурації може виступати і SRAM(0x2000 0000), і флеш пам’ять (0x0800 0000), і системна пам’ять(ROM) 0x1FFF 0000.

Процесор вважає , що в стартовій адресі 0x0000 0000 знаходиться таблиця векторів переривань, при чому кожний вектор є 32-бітним, 1-ий вектор – початкове значення вказівника стеку SP, друге значення – вказівник на початкове значення PC.

Рядок:

.word \_\_stack\_start

Встановлює вказівник стеку у вказану адресу, в даному випадку:

{

RAM ( rxw ) : ORIGIN = 0x20000000, LENGTH = 128K

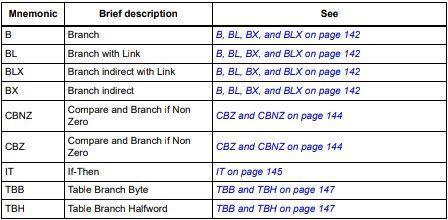
}

\_\_stack\_start = ORIGIN(RAM) + LENGTH(RAM);

Вона встановлює його в кінець оперативної пам’яті.

В флеш пам’яті зазвичай знаходиться наша програма і стартує звідти за замовчуванням.

**1.5 Основні інструкції переходу**



Всі інструкції переходу відносні.

Особливу увагу слід звернути на інструкцію **BL** **<відносна адреса>** та її відмінність від **B <відносна адреса>**. Вона не тільки переходить (змінює **PC**), а ще й зберігає стару адресу **PC** в регістр **LR** та збільшує її на 1 (при переході за адресою в регістрі перевіряється тип інструкцій за першим бітом(флагом)). Визвана процедура має зберігати **LR**(або значення з нього) щоб мати можливість завершитись і передати виконання процедурі, що її визвала. Якщо значення адреси зберігалося в **LR** можна визвати **BX LR**.

**BX** переходить за адресою у регістрі. Є і інший більш використовуваний спосіб. Це - використання стеку. Викликана процедура зберігає у стек разом з регістрами **змінних** (див AAPCS (Procedure Call Standard for the ARM Architecture)), що вона буде використовувати і LR.

push { r4, r6, lr }. Зверніть Увагу, що лише 1 інструкція може зберегти декілька регістрів у стек. pop { r4, r6, pc } одразу витаскує значення всіх регістрів, у тому числі виходить з процедури.

Проведемо наступний дослід - спробуємо визвати коректно процедуру без **BL**.

В коді до 1 лабораторної знайдіть:

\_\_hard\_reset\_\_:

// initialize stack here

// if not initialized yet

bl lab1

\_loop: b \_loop

.size \_\_hard\_reset\_\_, .-\_\_hard\_reset\_\_

Замініть bl lab1 на:

mov lr, \_loop

add lr, 1

b lab1

Отже 3 інструкції замість одної правильно її замінюють. mov lr, \_loop записує адресу повернення, add lr, 1 дає процесору зрозуміти при виході з процедури, що використовується набір thumb інструкцій, b lab1 просто переходить до виконання процедури.

**1.6 GDB**

GDB – відлагоджувач програм. З його допомогою можна покроково виконати програму та слідкувати за станами регістрів.

Основні команди:

layout regs -- Відображає усі регістри разом з вихідним кодом.

help -- Відображає список класів команд

kill -- Примусово зупиняє запущену програму

step -- Виконати наступний рядок коду

stepi -- Виконати наступну машинну інструкцію

continue -- Продовжити виконання програми

quit -- Вийти з GDB

**2 Література**

Основний манул по роботі в архітектурі STM32 Cortex-M4 - <https://www.st.com/content/ccc/resource/technical/document/programming_manual/6c/3a/cb/e7/e4/ea/44/9b/DM00046982.pdf/files/DM00046982.pdf/jcr:content/translations/en.DM00046982.pdf>

Також, невеликі за обсягом статті, cпецифічні для Cortex директиви:

<https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/useful-assembler-directives-and-macros-for-the-gnu-assembler>

Усі GAS директиви:

<https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html>

Вони дозволять вам розуміти синтаксис мови асемблера GAS (GNU Assembly), що є частиною стандартного пакету тулчейну GCC (GNU Compiler Collection) для арм (arm-none-eabi-). А також ви зможете вже працювати з GDB відлагоджувальником самостійно.

Слід звернути увагу на те, що препроцесор С, що є частиною пакету GCC, може бути використаний(і так слід робити) в GAS. Тобто Сшні макроси, коментарі, і т.і там підтримуються. В GAS є свій препроцесор, але він вмикається після роботи Сішного препроцесора. Деякий його функціонал ми також використовуємо.

Повний набір thumb інструкцій можна знайти за посиланням:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html>

Але рекомендується використовувати мануал від ST, оскільки ми процюємо з їх реалізацію Cortex мікропроцесорів(див розділ Література):

PM0214 Programming manual STM32 Cortex®-M4 MCUs and MPUs programming manual.

**3 Завдання:** Написати програму на GAS, яка рахує функцію за варіантом та запустити її в відлагоджувачі gdb, як показано в лабораторній роботі 1. Показати регістри з вірним результатом.

Варіант визначається остачею від ділення на 5 номера залікової книжки: ХХХХ%5 = № варіанта.

**4 Варіанти**

|  |  |
| --- | --- |
| **№** | **Функція** |
| 0 | (a+b)/2 + c! |
| 1 | (a-b)\*3 + |
| 2 | (a&b)>> + c! |
| 3 |  |
| 4 |  |

**\*>> -** логічний зсув вправо

**5 Порядок виконання**

1. Створити файли start.S та lscript.ld як в лабораторній роботі №1, але зміст файлу start.S наступний:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | .syntax unified | |  | .cpu cortex-m4 | |  | //.fpu softvfp | |  | .thumb | |  |  | |  | // Global memory locations. | |  | .global vtable | |  | .global \_\_hard\_reset\_\_ | |  |  | |  | /\* | |  | \* vector table | |  | \*/ | |  | .type vtable, %object | |  | .type \_\_hard\_reset\_\_, %function | |  | vtable: | |  | .word \_\_stack\_start | |  | .word \_\_hard\_reset\_\_+1 | |  | .size vtable, .-vtable | |  | \_\_hard\_reset\_\_: | |  | // initialize stack here | |  | // if not initialized yet | |  | bl lab1 | |  | \_loop: b \_loop | |  | .size \_\_hard\_reset\_\_, .-\_\_hard\_reset\_\_ | |

1. Створіть файл lab1.S, в якому і буде програма за варіантом. Приклад:

lab1.S

|  |
| --- |
| .global lab1  .syntax unified  #define A #4  lab1:  push {lr}  // calculate  mov r0, A  mov r1, #0  bl test\_var  pop {pc}  test\_var:  push { r0, r1, lr }  cmp r0, r1  ITE GE  movGE r3, r0  movLT r2, r0  pop { r0, r1, r2, pc } |

1. Створіть Makefile для автоматичної збірки проекту:

|  |
| --- |
| SDK\_PREFIX?=arm-none-eabi-  CC = $(SDK\_PREFIX)gcc  LD = $(SDK\_PREFIX)ld  SIZE = $(SDK\_PREFIX)size  OBJCOPY = $(SDK\_PREFIX)objcopy  QEMU = qemu-system-gnuarmeclipse  BOARD ?= STM32F4-Discovery  MCU=STM32F407VG  TARGET=firmware  CPU\_CC=cortex-m4  TCP\_ADDR=1234  deps = \  start.S \  lscript.ld  all: target  target:  $(CC) -x assembler-with-cpp -c -O0 -g3 -mcpu=$(CPU\_CC) -Wall start.S -o start.o  $(CC) -x assembler-with-cpp -c -O0 -g3 -mcpu=$(CPU\_CC) -Wall lab1.S -o lab1.o  $(CC) start.o lab1.o -mcpu=$(CPU\_CC) -Wall --specs=nosys.specs -nostdlib -lgcc -T./lscript.ld -o $(TARGET).elf  $(OBJCOPY) -O binary -F elf32-littlearm $(TARGET).elf $(TARGET).bin  qemu:  $(QEMU) --verbose --verbose --board $(BOARD) --mcu $(MCU) -d unimp,guest\_errors --image $(TARGET).bin --semihosting-config enable=on,target=native -gdb tcp::$(TCP\_ADDR) -S  clean:  -rm \*.o  -rm \*.elf  -rm \*.bin  flash:  st-flash write $(TARGET).bin 0x08000000 |

1. Виконайте збірку проекту за допомогою make:

>>> make

1. Запустіть емулятор qemu за допомогою make qemu:

>>> make qemu

1. В іншому терміналі запустіть відлагоджувач gdb командою *arm-none-eabi-gdb firmware.elf****.*** Та виконайте програму крок за кроком. Продемонструйте значення регістрів.