# **Лабораторна робота №3**

**Тема:** Завантажувач основної програми. Обробка виключень. Вивід даних на відлагоджувальний порт або консоль.

**Мета:** Навчитися працювати з оперативною пам’яттю, використовувати інструкції спеціального призначення, використовувати виключення процесора Cortex-M4. Створення мінімального завантажувача системи. Навчитися користуватися виводом даних через відлагоджувальний порт (або консоль).

## **3.1 Скорочені теоретичні відомості**

### **3.1.1 Відлагодження**

Відлагоджувальний функціонал процесора дозволяє передавати інформацію через програматор на реальному пристрої на базі ARM-Cortex M, та в емуляторах, таких як Qemu. Qemu дозволяє використати системні визови ОС, на якій він виконується через системні (відлагоджувальні) визови пристрою, що емулюється.

Для відлагодження в цій лабораторній роботі використовується мікрокоманда **BKPT**, детальніше про яку сказано в розділі **5.1.5**. Відладка програми буде проходити в режимі **Semihosting**.

**Semihosting** - це механізм, який дає змогу коду, що працює у вбудованій системі (його також називають цільовим), обмінюватись даними та використовувати пристрої вводу / виводу хост-комп'ютера. В основному, цільовий процесор зупиняється, або шляхом запуску команди на зупинку, або через якусь іншу операцію, яка зупиняє виконання програми і ставить цільовий процесор під контроль відлагоджувального агента. Відлагоджувальний агент зчитує один або кілька регістрів, щоб визначити тип операції, яку повинен виконувати хост від імені цілі, а потім виконує дію, після чого перезапускає ціль.

У більшості систем важливою частиною є введення / виведення до терміналу або принаймні тільки вивід. Це можливість надсилати текст хосту, відображати його на консолі відлагодження або терміналі. В цій лабораторній роботі буде використана лише дана операція, для перевірки правильності роботи коду.

### **3.1.2 Виключення**

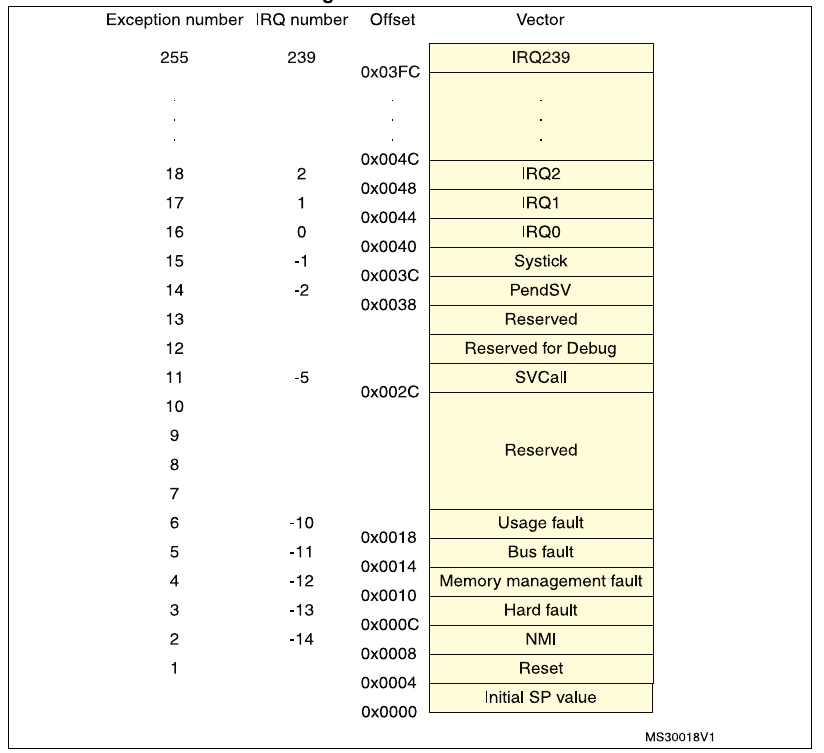
В даній лабораторній роботі використовується механізм виключень, на прикладі виключення **RESET**. Дане виключення відбувається при включенні живлення, або при сбросі системи. Найпершою під час старту системи запускається програма, що відповідає за обробку цього виключення.

Задати програму, що буде виконуватись відразу після старту системи, або оброблювати будь-яке інше виключення, можна за допомогою векторної таблиці переривань.

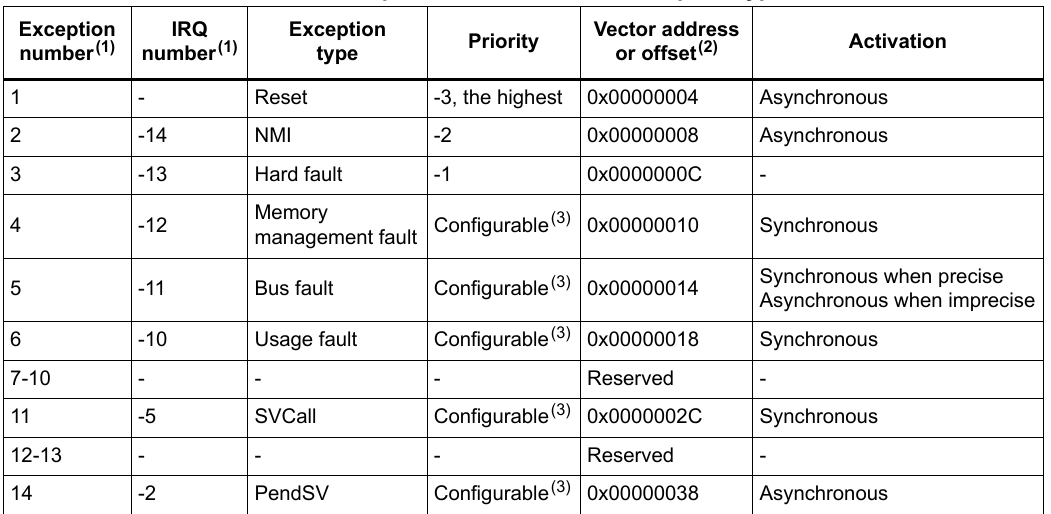
Векторна таблиця складається з: початкового значення **Stack Pointer**, та адрес, які називаються векторами переривань (виключень). 1-ий біт кожного вектора має дорівнювати 1, щоб вказати процесору на використання **thumb** інструкцій.

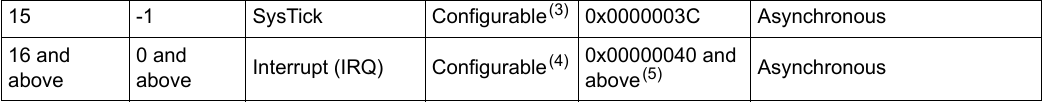
Кожний вектор переривання (виключення) є по суті адресою програми, що буде виконуватись якщо дане виключення з'явиться.

Векторна таблиця виглядає наступним чином:



**Виключення Cortex - M4 процесора:**





Детальніше з механізмом виключень та з їх видами можна в розділі 2.3 PM0214

### **3.1.3 Основні команди для лабораторної роботи**

Основними командами, що використовуються в даній лабораторній роботі, є команди доступу до пам’яті **STR** (store register) та **LDR** (load register).

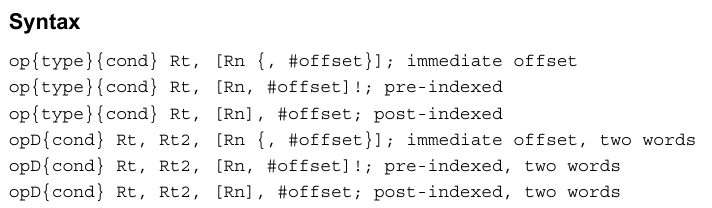
**LDR** – завантажує дані з пам’яті до регістрів.

**STR** – зберігає значення регістрів у пам’ять.

Існує декілька варіантів цих команд, в залежності від виду зсуву, адресації, типу даних та кількості регістрів. Також, як уже було сказано в **5.1.1**, в даній лабораторній роботі використовується відлагодження програми в режимі **Semihosting**, що виконується за допомогою мікрокоманди **BKPT**.

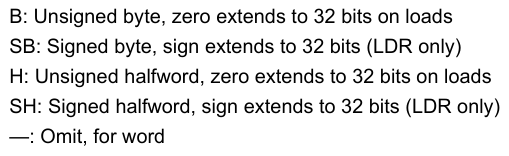
### **LDR & STR**

#### **З числовим зсувом:**



**op** -- **LDR** або **STR**

**type** -- Один з допустимих типів:



При чому слово - 32 біта.

**cond** -- умовний код

**Rt** -- регістр, в який завантажуються, чи з якого вивантажуються дані.

**Rn** -- регістр, в якому знаходиться адреса пам’яті.

**offset** -- зсув (відступ) в байтах від адреси, що знаходиться в **Rn**. Якщо не вказаний - адресою вважається вміст **Rn**.

**Rt2** -- додатковий регістр для 2-слівних операцій.

1. **Зсувна адресація:**

Значення зсуву (**offset**) додається (чи віднімається) до адреси, що знаходиться в регістрі **Rn**. По адресі, що утворилась в результаті, відбувається доступ до пам’яті. При цьому значення в регістрі **Rn** не змінюється.

Синтаксис для запису цього виду адресації наступний: **[Rn, #offset]**

1. **Пре-індексна адресація:**

Значення зсуву (**offset**) додається (чи віднімається) до адреси, що знаходиться в регістрі **Rn**. Адреса, що утворилась в результаті, використовується для доступу до пам’яті та записується в регістр **Rn**.

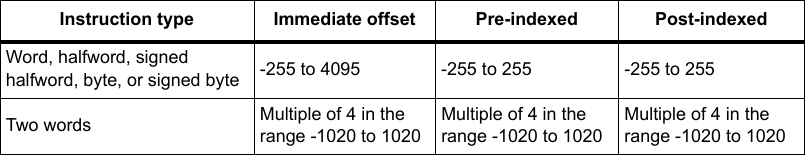
Синтаксис для запису цього виду адресації наступний: **[Rn, #offset]**!

1. **Пост-індексна адресація:**

Адреса, що знаходиться в регістрі **Rn**, використовується для доступу до пам’яті. Значення зсуву (**offset**) додається (чи віднімається) до адреси, та записується назад в регістр **Rn**.

Синтаксис для запису цього виду адресації наступний: **[Rn], #offset**

При даних видах адресації, значення зсуву (**offset**) може приймати наступні значення:



**Приклади використання:**

**LDR R8, [R10]** ; Завантажує в R8 слово за адресою в R10.

**STRH R3, [R4], #4** ; Вивантажує півслова з R3 в пам’ять за адресою в R4

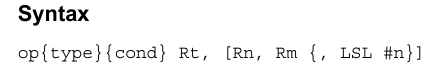
; після чого додає до R4 4.

**STRD R0, R1, [R8], #-16** ; Вивантажує слово з R0 за адресою в R8, та

; вивантажує слово з R1 за адресою на 4 байти

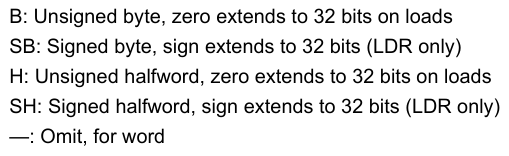
; більшою від R8, після чого декрементує R8 на 16.

#### **З регістровим зсувом:**

****

**op** -- **LDR** або **STR**

**type** -- Один з допустимих типів:



При чому слово - 32 біта.

**cond** -- умовний код

**Rt** -- регістр, в який завантажуються, чи з якого вивантажуються дані.

**Rn** -- регістр, в якому знаходиться адреса пам’яті.

**Rm** -- регістр, що містить значення зсуву.

**LSL #n** -- необов’язковий логічний зсув зі значенням **n =**

**LDR** та **STR** з регістровим зсувом працюють наступним чином:

Адреса в пам'яті, за якою відбувається завантаження (вивантаження), отримується зі зсуву від адреси в регістрі **Rn**. Зсув визначається значенням в регістрі **Rm**, яке може бути додатково зсунуте до 3 бітів вліво, використовуючи **LSL**.

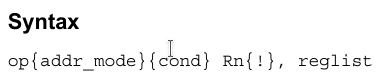
**Приклади використання:**

**STR R0, [R5, R1]** ; Вивантажує значення в R0 в пам’ять

; за адресою, що дорівнює сумі R5 та R1.

**STR R0, [R1, R2, LSL #2]** ; Вивантажує значення R0 за адресою, що ; дорівнює сумі R1 та 4 \* R2.

#### **Операції доступу до пам’яті з декількома регістрами:**



**op** -- **LDM** або **STM**

**addr\_mode** -- Один з наступних:

**IA** -- Інкрементувати адресу після кожного доступу (за замовчуванням)

**DB** -- Декрементувати адресу перед кожним доступом.

**cond** -- умовний код.

**Rn** -- регістр, в якому знаходиться адреса пам’яті.

**!** -- Необов’язковий суфікс. Якщо він вказаний, то результуюча адреса, за якою завантажуються (чи вивантажуються) дані, буде записана до **Rn**.

**reglist** -- Список з 1-го чи більше регістрів, в які завантажуються (чи з яких вивантажуються) дані, що задані в дужках **{}**. Може бути заданий в якості ряду регістрів. Якщо задається більше, ніж 1 регістр, то вони мають бути відокремлені комами (див. Приклад використання).

**LDM** -- завантажує слова в регістри, вказані в **reglist**, за адресою в **Rn**.

**STM** -- вивантажує слова з регістрів, вказаних в **reglist**, за адресою в **Rn**.

Для команд **LDM**, **LDMIA**, **STM**, **STMIA** адресація відбувається наступним чином:

Адреси, що використовуються для доступу до пам’яті, поділяються на 4-байтні інтервали від **Rn** до **(Rn + 4 \* (n - 1))**, де **n** - кількість регістрів в **reglist**. Доступ відбувається в порядку нумерації регістрів, тобто регістр з найнижчим порядковим номером у **reglist** використовує найнижчу адресу, а регістр з найвищим порядковим номером - найвищу. При цьому, якщо заданий суфікс **!**, то значення **(Rn + 4 \* n)** буде записано назад в **Rn**.

**Приклади використання:**

**LDM R8,{R0,R2,R9}** ; Завантажує значення наступним чином:

; R0 := [R8], R2 := [R8 + 4], R9 := [R8 + 8]

; Значення в регістрі R8 лишається без змін.

**STMDB R1!,{R3-R6,R11,R12}** ; Вивантажує значення наступним чином:

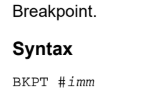
; [R1 - 4] := R3, [R1 - 8] := R4,

; [R1 - 12] := R5, [R1 - 16] := R6,

; [R1 - 20] := R11, [R1 - 24] := R12.

Новим значенням адреси в регістрі R1 стає: R1 := R1 + 24.

### **BKPT**



Де **imm** - вираз, що оцінює ціле число в діапазоні 0-255.

Інструкція **BKPT** приводить процесор до стану налагодження. Інструменти для налагодження можуть використовувати це, щоб дослідити стан системи, коли буде досягнута інструкція за певною адресою. Процесор ігнорується. Якщо потрібно, відладчик може використовувати його для зберігання додаткової інформація про точку розриву.

Значення **0xAB** - означає [Semihosting](https://wiki.segger.com/Semihosting), тобто використання можливостей зовнішнього пристрою для відладки.

Також **BKPT** має регістрові параметри, для **0xAB**:

перший параметр - тип операції, що потрібно зробити хосту.

2-й - данні для операції.

Наприклад, операція **0x4** (згідно з [Semihosting](https://wiki.segger.com/Semihosting)) – вивід рядку, параметр 2 - строка символів з 0 в кінці.

Приклад hello world:

#define SEMIHOSTING\_SYS\_WRITE0 #0x04

data: .asciz "Hello World"

mov r0, SEMIHOSTING\_SYS\_WRITE0

ldr r1, =data

bkpt #0xAB

Цей фрагмент коду виведе рядок у відлагоджувальний порт (або в термінал з якого запущено Qemu).

Зверніть увагу на те, що інструкція **bkpt** затримує процесор на виконання, поки відлагоджувальний пристрій не відповість. Тобто, якщо відлагоджувача нема, код, що містить цю інструкцію, не виконується. Тому, бажано зробити програму таким чином, щоб було легко відключити всі включення цієї інструкції в коді, оскільки виконання програми буде перевірятись на реальному мікроконтролері, наприклад, таким чином:

#ifdef DEBUG

bkpt #0xAB

#endif

Для простоти можна сказати, що це все потрібно, щоб виводити в відлагоджувальний порт або консоль значення регістрів, рядки і тд.

### **3.1.4 Завантажувач**

Метою даної лабораторної роботи є розробка примітивного завантажувача програми, що була написана в попередній лабораторній роботі, тобто побудова такої програми, що зможе побайтно перенести іншу програму в оперативну пам’ять та запустити її звідти. Розглянемо цю програму детальніше на прикладі:

kernel.S:

|  |
| --- |
| .syntax unified  .cpu cortex-m4  .thumb  #define A #4  // Global memory locations.  .global vtable\_kernel  .global \_\_kernel\_reset\_\_  .type vtable\_kernel, %object  .type \_\_kernel\_reset\_\_, %function  .section .interrupt\_vector  vtable\_kernel:  .word \_\_stack\_start  .word \_\_kernel\_reset\_\_+1  .size vtable\_kernel, .-vtable\_kernel  .section .rodata  data: .asciz "kernel started!\n"  final: .asciz "Value in register #3: "  .section .text  \_\_kernel\_reset\_\_:  ldr r0, =data  bl dbgput\_line    // calculate  mov r0, A  mov r1, #0  cmp r0, r1  ITE GE  movGE r3, r0  movLT r2, r0    ldr r0, =final  bl dbgput  mov r0, r3  bl dbgput\_num    end:  b end |

Ця програма буде відігравати роль завантажуваної програми. Зверніть увагу на те, як код з попередньої лабораторної роботи розміщений в даному файлі. Розберемо її трохи детальніше.

За допомогою директив **.section** **.interrupt\_vector** ми задаємо векторну таблицю переривань. Першим рядком таблиці ми визначаємо початковий стан **Stack Pointer**, тобто адресу, з якої починається стек. Далі ми вказуємо для оброблення виключення **RESET** адресу програми **\_\_kernel\_reset\_\_**, вказуючи при цьому використання thumb інструкцій.

Дана програма виводить в консоль відлагодження рядок **data**, після чого виконує приклад варіанту з лабораторної роботи №2, виводить вміст **r3** в консоль відлагодження та крутиться у нескінченному циклі. Для виводу результату виконання програми за варіантом використовується процедура **dbgput\_num**, з принципом роботи якої можна ознайомитись в **print.s**. Для виводу вмісту певного регістра потрібно скопіювати його значення в **r0** і лише після цього викликати процедуру.

Тепер розглянемо приклад роботи завантажувача (**bootloader.S**):

|  |
| --- |
| .syntax unified  .cpu cortex-m4  //.fpu softvfp  .thumb  .global bootload  .section .rodata  image: .incbin "kernel.bin"  end\_of\_image:  str\_boot\_start: .asciz "bootloader started"  str\_boot\_end: .asciz "bootloader end"  str\_boot\_indicate: .asciz "#" |

Початок файлу такий самий, як і в **kernel.S**. Нижче ми вказуємо мітку, що буде відповідати за завантаження **kernel.S** в оперативну пам’ять, як глобальну, щоб цю мітку було видно зі **start.S**. Далі, в секції **rodata** ми створюємо рядки з ASCII символів, для перевірки правильності роботи програми, та мітки **image** та **end\_of\_image**, що містять адресу пам’яті, в якій знаходиться початок та кінець програми (**end\_of\_image** вказує на наступне слово після останнього слова програми).

Для прикладу ми будемо завантажувати програму послівно, за допомогою інструкцій **ldr** та **str**:

|  |
| --- |
| .section .text  bootload:  ldr r0, =str\_boot\_start  bl dbgput\_line  ldr r0, =end\_of\_image  ldr r1, =image  ldr r2, =\_ram\_start |

Спочатку потрібно завантажити в регістри адреси початку та кінця програми, а також адресу початку оперативної пам’яті. Також за допомогою процедури **dbgput\_line** виводиться в консоль відлагодження рядок **str\_boot\_start**. Ознайомитись з принципом роботи процедур можна у файлі **print.S**.

Завантаживши у відповідні регістри адреси початку та кінця програми, будемо послівно завантажувати її в оперативну пам’ять:

|  |
| --- |
| loop:  ldr r3, [r1], #4  str r3, [r2], #4  cmp r0, r1  bhi loop |

Спочатку за допомогою інструкції **ldr** ми завантажуємо слово програми, що знаходиться за адресою в **r1**, після чого вивантажуємо його в оперативну пам’ять за адресою в **r2**, та переходимо до наступного слова. Цикл закінчується як тільки буде завантажене останнє слово програми.

Після завершення завантаження, залишилось тільки перейти на початок оперативної пам’яті, щоб почати виконання завантаженої програми:

|  |
| --- |
| bl newline  ldr r0, =str\_boot\_end  bl dbgput\_line |

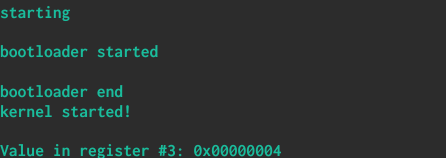
За допомогою процедур **newline** та **dbgput\_line** виводиться текст до відлагоджувальної консолі, після чого командою **ldr** ми завантажуємо адресу початку оперативної пам’яті:

|  |
| --- |
| ldr lr, =bootload\_end  add lr, #1  ldr r2, =\_ram\_start |

Оскільки на початку програми знаходиться векторна таблиця, то нам потрібно перейти на наступне слово, яке містить адресу підпрограми, що відповідає за обробку виключення **RESET** (див. **5.1.2**). В даному випадку це адреса підпрограми **\_\_reset\_lernel\_\_**:

|  |
| --- |
| add r2, #4 // go to \_\_reset\_kernel\_\_  ldr r0, [r2]  bx r0  bootload\_end:  b bootload\_end |

Після того як ми отримали з векторної таблиці адресу **\_\_kernel\_reset\_\_**, можна нарешті перейти до виконання завантаженої програми, що відбувається за допомогою команди **bx r0**. Також, в **lr** була збережена адреса мітки **bootload\_end**, на випадок повернення з завантаженої програми.

Приклад результату роботи завантажувача:

### **3.1.5 Основні директиви для лабораторної роботи**

**.incbin [some\_file]** - дозволяє зберегти в пам’ять файл в тому вигляді, в якому він є

**.section** – вказує на певний блок коду. Може використовуватися с директивами **.text**, **.data**, **.rodata**.

**.interrupt\_vector** - вказує використання об’єкта в якості векторної таблиці переривань.

**.text** – вказує на блок виконуваного коду

**.rodata** – read-only data, тобто вказує на блок даних тільки для читання.

**.asciz** “Some string” – кодує рядок в ASCII та повертає адресу його початку.

## **3.2 Література**

<https://vivonomicon.com/2018/04/02/bare-metal-stm32-programming-part-1-hello-arm/> - курс для розуміння програм на асемблері в архітектурі STM32 Cortex-M4.

## **3.3 Завдання**

1. «Зпулити» шаблон лабораторної роботи №2 з репозиторію <https://github.com/Igor1101/CompArch2STM32/tree/master/lab2>
2. Розібратись, як мають працювати завантажувач, обробка виключення та Semihosting.
3. Скопіювати код розрахунку виразу з 2-ої роботи та вставити його в kernel.s.
   1. Або модифікувати файл обчислення виразу з 2-ої роботи, тобто вставити в нього векторну таблицю, та модифікувати Makefile, щоб <your\_file\_name>.bin файл автоматично створювався. Це не є обов’язковим, але такий підхід є заохочувальним.
   2. Результат розрахунку має виводитись в консоль!
4. Створити завантажувач за варіантом, який буде виконувати програму (наприклад, kernel.bin).
5. Запустити програму в gdb, продемонструвати запуск завантажуваної програми та виведення результату в консоль.

Варіант визначається остачею від ділення номера залікової книжки на 16.

Зверніть увагу на те, що використання вашого завантажувача є обов’язковим в лабораторній 4.

## 

## **3.4 Варіанти**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Команди для роботи з пам’яттю** | **Інкремент/Декремент регістру адреси** | **Вид зсуву** | **Кількість байт для зсуву** |
| 0 | LDR, STR | інкремент | числовий | 4 |
| 1 | LDR, STR | інкремент | регістровий | 4 |
| 2 | LDR, STR | декремент\* | числовий | 4 |
| 3 | LDR, STR | декремент\* | регістровий | 4 |
| 4 | LDRB, STRB | інкремент | числовий | 1 |
| 5 | LDRB, STRB | інкремент | регістровий | 1 |
| 6 | LDRB, STRB | декремент\* | числовий | 1 |
| 7 | LDRB, STRB | декремент\* | регістровий | 1 |
| 8 | LDRH, STRH | інкремент | числовий | 2 |
| 9 | LDRH, STRH | інкремент | регістровий | 2 |
| 10 | LDRH, STRH | декремент\* | числовий | 2 |
| 11 | LDRH, STRH | декремент\* | регістровий | 2 |
| 12 | LDM, STM | інкремент | регістровий | 8 ( 2 регістра по 4) |
| 13 | LDM, STM | інкремент | регістровий | 12 (3 регістра по 4) |
| 14 | LDM, STM | декремент\* | регістровий | 8 (2 регістра по 4) |
| 15 | LDM, STM | декремент\* | регістровий | 12 (3 регістра по 4) |

\* - *Підказка: для варіантів з декрементною адресацією перед початком роботи завантажувача потрібно обчислити різницю адрес початку та кінця програми, після чого додати її до адреси початку оперативної пам’яті.*