МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
„КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
КАФЕДРА АВТОМАТИКИ ТА УПРАВЛІННЯ У ТЕХНІЧНИХ СИСТЕМАХ

Архітектура новітніх мікроконтролерів
Програмування мікропроцесорних систем на базі мікроконтролерів сімейства ARM

Надано гриф «Затверджено Вченою радою КПІ ім. Ігоря Сікорського як навчальний посібник для студентів, які навчаються за спеціальністю «Автоматизація та комп`ютерно–інтегровані технології»
(Протокол №1 від 16 січня 2017 року)

Київ КПІ ім. Ігоря Сікорського 2017

`

Архітектура новітніх мікроконтролерів: Програмування мікроконтролерів сімейства ARM: Навчальний посібник для студентів спеціальності 151 «Автоматизація та комп'ютерно–інтегровані технології» / Автор: А.О. Новацький–Київ: КПІ ім. Ігоря Сікорського, 2017–138с.

Відповідальний за випуск: к. т. н., доц. Л.Ю. Юрчук

Рецензенти: доктор фіз. мат. наук, професор А.Ю.Дорошенко,

к. т. н., доц. О.А.Чемерис

Посібнику надано гриф «Затверджено Вченою радою КПІ ім. Ігоря Сікорського як навчальний посібник для студентів, які навчаються за спеціальністю «Автоматизація та комп`ютерно–інтегровані технології»

(Протокол №1 від 16 січня 2017 року)

Навчальний посібник охоплює теоретичний матеріал та практичні завдання, які необхідні для вивчення базової частини дисципліни «Архітектура новітніх мікроконтролерів». Робота містить п`ять розділів, в яких розглядаються питання, що пов`язані із програмуванням мікропроцесорних систем на базі ARM–мікроконтролерів: загальна характеристика мікроконтролерів сімейства ARM; структура типового мікроконтролера ядра ARM7; характеристика ядра ARM–мікроконтролерів; способи адресації операндів та команди ARM–мікроконтролерів ядра ARM7.

Робота може бути корисною студентам відповідних спеціальностей при вивченні дисциплін, які пов'язані із проектуванням та використанням мікропроцесорних систем, а також при виконанні бакалаврських та магістерських робіт, курсових та дипломних проектів, в яких використовуються мікропроцесорні пристрої. Останнє було враховано при оформленні роботи, яку виконано згідно вимог до конструкторської документації.

`

ЗМІСТ

`

`

ВСТУП

Курс „Архітектура новітніх мікроконтролерів” належить до циклу дисцилін „Комп”ютерна електроніка та мікропроцесорна техніка” та є одним з основних при підготовці магістра в області комп'ютеризованих систем управління та автоматики. В ньому висвітлюються проблеми проектування, програмування і застосування мікропроцесорних систем на прикладі мікроконтролерів LPC2378 сімейства ARM7.

Архітектура ARM (Advanced RISC Machine) – 32–бітна RISC–архітектура процесорів. Розроблена компанією ARM Limited. Процесори ARM мають низьке енергоспоживання, тому завоювали сегмент масових мобільних продуктів (стільникові телефони, кишенькові комп'ютери) і знаходять широке застосування у вбудованих системах середньої і високої продуктивності (від мережевих маршрутизаторів і точок доступу до телевізорів). Станом на 2014 р. 95% смартфонів у світі, 80% усіх цифрових камер і 35% усіх електронних пристроїв використовують ARM–технології.

В результаті вивчення дисципліни студент повинен вміти проектувати мікропроцесорні системи на сучасних мікроконтролерах, розробляти алгоритми та керуючі програми на асемблері та мовах високого рівня.

У практичному плані студент повинен навчитися проектувати та програмувати мікропроцесорні системи за модульним принципом, користуватися науково–технічною і додатковою літературою, самостійно опановувати нові питання, які відносяться до даної дисципліни. Вивчення курсу базується на знаннях, набутих у процесі вивчення дисциплін: мікропроцесорні системи, проектування мікропроцесорних систем, архітектура комп'ютерних систем та мереж, комп'ютерна електроніка, алгоритмічні мови програмування, проектування мікропроцесорних систем та мереж.

`

СПИСОК СКОРОЧЕНЬ

АЦП – аналого–цифровий перетворювач

БО – безпосередній операнд

ГТІ – генератор тактових імпульсів

ДК – двійковий код

ЗПД – зовнішня пам’ять даних

ЗПП – зовнішня пам’ять програм

КОП – код операції

МК – мікроконтролер

МП – мікропроцесор

МПС – мікропроцесорна система

МЦ – машинний цикл

ОК – об’єкт керування

ПВВ – пристрій введення/виведення

ПВВ – пристрій введення/виведення

РГ – регістр

РЗП – регістр загального призначення

РКС – регістр керуючого слова

РПД – резидентна пам’ять даних

РПП – резидентна пам’ять програм

СБ – старший байт

СД – світлодіод

СМА – суматор адреси

СУР – схема узгодження рівнів

США – системна шина адреси

ЦАП – цифро–аналоговий перетворювач

ЦПП – центральний процесорний пристрій

Access Line – тактова частота

`

ACP – Accelerator Coherency Port

AHB – Advanced High–Performance Bus

AMP – Asymmetric Multi–Processing

APB (Advanced Peripheral Bus) – покращена периферійна шина

ARM – Advanced RISC Machine

Block System Functions – блок системних функцій

CISC – Complex Instruction Set Computing

CPSR (Current Program Status Register) – регістр поточного стану програми

DBX (Direct Bytecode eXecution) – пряме виконання байт-коду

EMC – контролер зовнішньої пам'яті

GIC – Generic Interrupt Controller

ICE (In–Circuit Emulator) – схемний емулятор

IoT – Internet of Things

Java ME – Java Mobile Edition

LLPP – Low Latency Peripheral Port

MIPS (Million Instructions Per Second) – мільйон інструкцій за секунду

MMU (Memory Management Unit) – блок управління пам'яттю

MPU (Memory Protection Unit) – модуль захисту пам’яті

MSB (Most Significant Bit) – старший біт

NVRAM(Non–Volatile Random–Access Memory) – енергонезалежна пам'ять

PLL – Phase Locked Loop

RISC – Restricted (reduced) Instruction Set Computer

SCU – Snoop Control Unit

SIMD (Single Instruction, Multiple Data) – одиночний потік команд, множинний потік даних

SMP – Symmetric Multi–Processing

`

SPSR – Stored Processor State Register

SPSR (Saved Program Status Register) – регістр збереженого стану програми

TCM (Tightly–Coupled Memory) – тісно зв’язана пам'ять

TDMI – Thumb + Debug + Multiplier + ICE

VIC (Vector Interrupt Controller) – векторний контролер переривань

`

1 ЗАГАЛЬНА ХАРАКТЕРИСТИКА МІКРОКОНТРОЛЕРІВ СІМЕЙСТВА ARM

1.1 RISC–архітектура мікроконтролерів

Усі x86–процесори компанії Intel, рішення компанії Motorola і переважна більшість випущених в 1980–і роки кристалів мали архітектуру CISC (complex instruction set computing). У ній набори команд виконували якомога більше роботи, аби полегшити ручне написання програм на мовах асемблерів чи прямо в машинних кодах, а також для спрощення реалізації компіляторів. Нерідко в набори включалися команди для прямої підтримки конструкцій мов високого рівня. Інша особливість цих наборів – більшість команд, як правило, допускали всі можливі методи адресації – наприклад, і операнди, і результат в арифметичних операціях доступні не тільки в регістрах, але і через безпосередню адресацію, і прямо в пам'яті.

В якийсь момент тогочасні чипи стали не тільки складними і дорогими у виробництві, але і досягли своєї межі продуктивності. Для подальшого збільшення швидкодії необхідно було нарощувати кількість транзисторів, однак освоєні технологічні норми не дозволяли створювати більш складні рішення. Для підвищення продуктивності треба було вносити зміни в архітектуру процесорів.

Було проведено дослідження і показано, що багато компіляторів просто не використовували всіх можливостей наборів команд CISC (наприклад, система Unix при компіляції використовувала лише 30% команд), а на складні методи адресації йшло багато часу через додаткові звернення до повільної пам'яті. Було показано, що такі функції краще виконувати послідовністю простіших команд, якщо при цьому процесор спрощується і в ньому залишається місце для більшого числа регістрів, завдяки яким можна скоротити кількість звернень до пам'яті.

`

Так у 1980 році, в університеті Берклі, був початий проект *RISC* (restricted (reduced) instruction set computer). Планувалося створити такий процесор, який би містив лише найнеобхідніші інструкції. Дослідження базувалися на використанні конвеєрної обробки і агресивного використання техніки регістрового вікна. У звичайному процесорі є невелика кількість регістрів, і програма може використовувати будь–який регістр в будь–який час. У процесорі, що використовує технологію регістрового вікна, дуже велика кількість регістрів (наприклад, 128), але програми можуть використовувати обмежену кількість (наприклад, тільки 8 в кожен момент часу).

Програма, обмежена лише вісьмома регістрами для кожної процедури, може виконувати дуже швидкі виклики процедур: «вікно» просто зрушується до 8–регістрового блоку потрібної процедури, а при поверненні з процедури зсувається назад до регістрів цієї процедури. У звичайному процесорі більшість процедур при виклику змушені зберігати значення деяких регістрів в стеку для того, щоб користуватися цими регістрами при виконанні процедури. При поверненні з процедури значення регістрів відновлюються зі стека.

У перших архітектурах, що зараховуються до RISC, більшість команд для спрощення декодування мають однакову довжину і схожу структуру, арифметичні операції працюють тільки з регістрами, а робота з пам'яттю йде через окремі команди завантаження (load) і збереження (store). Ці властивості і дозволили краще збалансувати етапи конвеєризації, зробивши конвеєри в RISC значно ефективнішими, дозволивши підвищити тактову частоту і ефективність суперскалярності (розпаралелювання команд між декількома виконавчими блоками).

Нерідко слова «скорочений набір команд» неправильно розуміється, як мінімізація кількості команд в системі команд. Насправді, команд у багатьох RISC–процесорів більше, ніж у CISC–процесорів. Термін «скорочений»

`

описує той факт, що скорочений обсяг і час роботи, що виконується кожною окремою командою – максимум один такт доступу до пам'яті – тоді як складні команди CISC–процесорів можуть вимагати для свого виконання сотень тактів доступу до пам'яті.

Характерні особливості RISC–процесорів:

– фіксована довжина машинних команд (наприклад, 32 біти) і простий формат команди;

– спеціалізовані команди для операцій з пам'яттю – читання або запису. Операції виду «прочитати–змінити–записати» відсутні. Будь–які операції «змінити» виконуються тільки над вмістом регістрів (т. зв. архітектура load–and–store);

– велика кількість регістрів загального призначення (32 і більше);

– відсутність підтримки операцій виду «змінити» над укороченими типами даних: байт або 16–бітове слово. Так, наприклад, система команд DEC Alpha містила лише операції над 64–бітними словами, і вимагала розробки і подальшого виклику процедур для виконання операцій над байтами, 16– і 32–бітними словами;

– відсутність мікропрограм всередині самого процесора. Те, що в CISC–процесорі виконується мікропрограмами, в RISC–процесорі виконується як звичайний (хоча і поміщений в спеціальне сховище) машинний код, який не відрізняється принципово від коду ядра ОС і додатків.

В даний час багато архітектур процесорів є RISC–подібними, наприклад, ARM, DEC, Alpha, SPARC, AVR, MIPS, POWER і PowerPC. Найбільш широко використовувані в настільних комп'ютерах процесори архітектури x86 раніше були CISC–процесорами, проте нові процесори, починаючи з Intel Pentium Pro, є CISC–процесорами з RISC–ядром. Вони безпосередньо перед виконанням перетворять CISC–інструкції x86–процесорів в більш простий набір внутрішніх інструкцій RISC.

`

Після того, як процесори архітектури x86 були переведені на суперскалярну RISC–архітектуру, можна сказати, що більшість існуючих нині процесорів засновані на архітектурі RISC.

1.2 Загальний огляд мікроконтролерів сімейства ARM

Архітектура ARM (Advanced RISC Machine) – 32–бітна RISC–архітектура процесорів. Розроблена компанією ARM Limited. Процесори ARM мають низьке енергоспоживання, тому завоювали сегмент масових мобільних продуктів (стільникові телефони, кишенькові комп'ютери) і знаходять широке застосування у вбудованих системах середньої і високої продуктивності (від мережевих маршрутизаторів і точок доступу до телевізорів). Станом на 2014 р., 95% смартфонів у світі, 80% усіх цифрових камер і 35% усіх електронних пристроїв використовують ARM технології.

Історія ARM почалася, коли компанія Acorn Computers після успіху з комп'ютером BBC Micro задумалася над переходом від відносно слабких процесорів MOS Technology 6502, яким не вистачало потужності для підтримки графічного інтерфейсу, до більш продуктивних рішень. Жодна з доступних 16–бітних архітектур не задовольняла їхніх вимог. Тому компанія вирішила розробити нову 32–бітну архітектуру на основі нової на той час архітектури RISC. Проект дістав назву Acorn RISC Machine (1983р.). Ключовим завданням було досягнення низьких затримок при обробці переривань, як у MOS Technology 6502. Архітектура доступу до пам'яті з 6502 дозволила розробникам досягнути хорошої продуктивності без використання дорогого модуля прямого доступу до пам'яті. Процесор вперше запрацював у 1985р. і був названий *ARM1*. Це був повністю функціональний 32–розрядний процесор з 26–розрядним (64 Мбайт) адресним простором. Він мав шістнадцять 32–бітних регістрів, ортогональний набір 32–бітних інструкцій, тобто при виконанні інструкції не було обмеження на використання тільки визначених для неї регістрів.

`

Лічильник команд був обмежений 26–ма бітами, тому що верхні 6 біт 32–розрядних регістрів були зарезервовані в якості статус–міток. Тому програмний код перебував всередині перших 64 Мбайт пам'яті.

Перші серійні процесори під назвою *ARM2* стали доступні в наступному році. У ньому додали інструкції множення, апаратну підтримку співпроцесорів математики з плаваючою комою. ARM2, можливо, є найпростішими із використовуваних 32–бітних мікропроцесорів в світі, робота якого забезпечена лише 30000 транзисторами (для порівняння, старіша на 6 років модель Motorola 68000 містила близько 70000 транзисторів). Така простота обумовлювалася відсутністю мікропрограми, яка, наприклад, у процесорі 68000 становила від 1/4 до 1/3 площі кристалу, і відсутністю кешу. Це привело до низьких витрат енергії, і той же час ARM був набагато продуктивніший, ніж Intel 80286 – до 4 MIPS (million instructions per second – мільйонів інструкцій за секунду).

Наступний процесор – *ARM3* – уже мав кеш (4 КБ). Тактова частота зросла до 25 МГц, продуктивність склала 13 MIPS.

В кінці 1980–х років компанія Apple почала працювати з Acorn Computers над новими версіями ядра ARM. В результаті співробітництва був створений *ARM6* (1992 р.). Адресну шину розширили до 32 біт, зберігаючи сумісність з попереднім 26–бітним режимом. Введені два нові режими процесора – для обробки помилок звернення до пам'яті та невизначених інструкцій. Додані два нові регістри: регістр поточного стану процесора (CPSR, Current Processor State Register) і регістр збереженого стану процесора (SPSR, Stored Processor State Register). ARM6 працював на частоті 20, 30 і 33 МГц і виконував в середньому 17, 26, і 28 MIPS відповідно. Це був також енергоефективний процесор, з низькою для того часу напругою живлення ядра 3.3В. Ядро ARM, попри всі зміни, залишилося фактично такого ж розміру: кількість транзисторів зросла всього до 35000. Це був початок для мобільних вбудованих систем: першим продуктом, який

`

використовував ARM6, став Apple Newton MessagePad – одна з перших серій кишенькових персональних комп'ютерів.

У *ARM7* (1993р.) збільшили розмір кеш–пам'яті до 8 КБ. Також був представлений розширений набір команд – Thumb. Thumb – це інший 16–бітний набір інструкцій, що дозволяє (теоретично), щоб програми займали половину від їх розміру в пам'яті. Використовуючи всього 8 регістрів, за відсутності підтримки умовного виконання, він працює повільніше, але був відповіддю на інші вбудовані 8–бітні і 16–бітні процесори. У ARM7 удосконалили множення – з'явилися як 32–розрядні, так і 64–бітні інструкції множення і множення/накопичення. Реалізація виявилася настільки успішною, що її використали в наступних версіях ядер – ARM8, ARM9 і StrongARM. В ARM7 вперше представили апаратне відлагодження (on–chip debugging).

ARM7TDMI є удосконаленням ядра ARM7. TDMI розшифровується, як «Thumb + Debug + Multiplier + ICE», де ICE (In–Circuit Emulator, схемний емулятор) – апаратний пристрій, який підключаються до системи через спеціальний порт, і дає можливість взяти контроль над вбудованою системою з метою відлагодження. З продуктивністю до 130 MIPS, це було одне з найбільш широко використовуваних ядер у вбудованих системах, Apple iPod, Nintendo Game Boy Advance і більшості популярних мобільних телефонів.

ARM8 (1996р.) включав модуль передбачення переходів (branch prediction) і пам'ять з подвійною пропускною здатністю. Тут з'явився п'ятиатапний конвеєр, у той час як в ARM7 тільки три. Завдяки цьому ARM8 вдвічі продуктивніший порівняно з ARM7.

У *ARM9* (1998р.) відбувся перехід з класичної архітектури фон Неймана на модифіковану Гарвардську, з відділенням шини команд від шини даних і кешу. Завдяки цьому вибірка команди і доступ до даних може відбуватися одночасно. ARM9 також використовував п'ятиетапний конвеєри, введений у ARM8.

`

ARM9TDMI став заміною для надзвичайно популярного ARM7TDMI. Додатки, призначені для ARM7TDMI, були приблизно на 30% швидше на ARM9TDMI.

Для *ARM10* задачею було подвоїти продуктивність ARM9. Для цього ARM працювали над двома основними аспектами: оптимізацією конвеєра і швидкістю виконання команд. Для підвищення швидкості конвеєра, етап декодування розділили на два, де на одному декодується частина інструкції, на другому читаються регістри з залишком декодованої послідовності. В ARM10 з'явилося нове ядро множення – швидкий 16 × 32 апаратний помножувач. Це дозволило виконувати одну 32–бітну операцію множення і накопичення за один такт, що є величезне поліпшення в порівнянні з 3–5 тактами на ARM9.

Архітектура, що використовується в ARM9 і ARM10, впровадила технологію Jazelle DBX (Direct Bytecode eXecution, пряме виконання байт–коду), яка дозволяє виконання Java байт–коду на апаратному рівні. Спрямована головним чином на ринок мобільних телефонів, Jazelle давала змогу Java ME (Java Mobile Edition) додаткам і іграм працювати швидше шляхом перетворення байт–коду в машинні інструкції ARM. ARM стверджує, що приблизно 95% байт–коду в типових програмах обробляється на апаратному рівні.

У 2002р. ARM випустила ядро *ARM11*. Його архітектура реалізовує технологію SIMD (Single Instruction, Multiple Data – одиночний потік команд, множинний потік даних) – принцип комп'ютерних обчислень, що дозволяє забезпечити паралелізм на рівні даних. Інструкції SIMD розроблені для виконання повторюваних інструкцій на багатьох наборах даних, що широко використовується в аудіо – і відеокодеках, зокрема, для MPEG–4 на ринку мобільних телефонів. Додавання інструкцій SIMD подвоїла швидкість обробки MPEG–4. П'ятиетапний конвеєр замінили на восьмиетапний, підвищивши очікувану частоту до 1 ГГц. ARM11 також підтримує обмежене

`

позачергове виконання команд, дозволяючи конвеєру продовжити виконання, якщо результат попередньої команди не потрібен наступним командам.

ARM11 був величезним успіхом, ARM–процесори стають все потужнішими, але й дорожчими. Тим не менш, деякі клієнти не були зацікавлені в дуже швидких процесорах та складній архітектурі і шукали більш «легкі» рішення, в той же час отримуючи всі переваги від останніх технологічних досліджень. ARM перелаштували свою лінію процесорів для задоволення потреб особливих сфер. Так у 2004р. було повідомлено про нове сімейство *Cortex* (рисунок 1.1).

До сімейства Cortex входить 3 класи процесорів: Cortex–A, Cortex–R і Cortex–M. Cortex–A («A» – application) призначено для пристроїв, що вимагають високої продуктивності (смартфони, планшети), на яких працює багатозадачна операційна система. Cortex–R («R» – real time) розроблено для додатків реального часу, де час реакції повинен бути мінімальний. Cortex–M («M» – microcontroller) призначено для мікроконтролерів і недорогих вбудованих пристроїв.

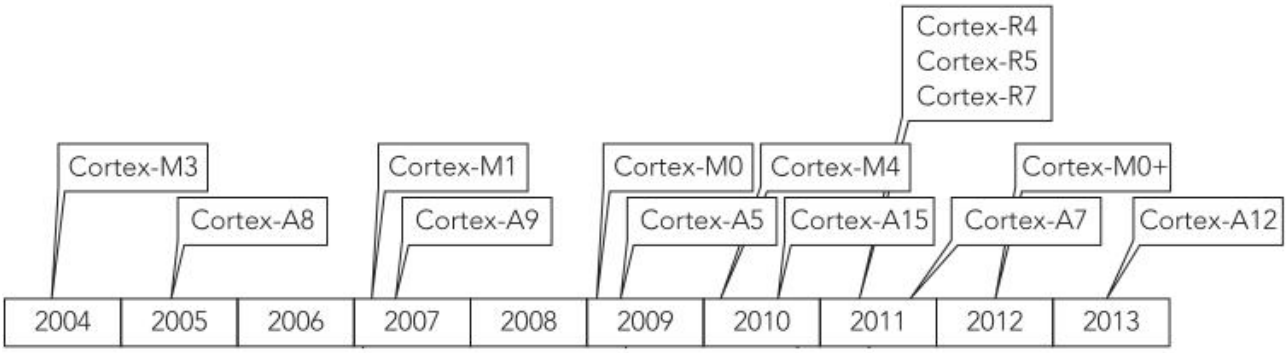


Рисунок 1.1 – Графік виходу ядер сімейства Cortex

Процесор Cortex–A призначено для прикладних систем. Це потужні чипи, які дозволяють запустити повноцінну операційну систему, високопродуктивні мультимедійні кодеки і вимогливі до ресурсів додатки. Серія Cortex–A оснащено MMU (memory management unit, блок управління

`

пам'яттю) – апаратний компонент, що відповідає за управління доступом до пам'яті, який запитується процесором. Додавши декілька зовнішніх компонентів, можна створювати передові платформи. На Cortex–A працюють смартфони, планшети, цифрові телевізори та інформаційно–розважальні системи, і навіть були розроблені ноутбуки на багатоядерних Cortex–A. В таблиці 1.1 наведено характеристики деяких процесорів сімейства Cortex.

Таблиця 1.1 – Характеристика процесорів Cortex–A

Cortex–A8 (2005р.)	Перший Cortex–A– процесор. Одноядерний, Thumb–2, NEON, TrustZone , динамічне передбачення переходів (заявлена точність 95%), оптимізований кеш 1 рівня, інтегрований налаштовуваний кеш 2 рівня (0КБ–1МБ). Частота до 1Гц. Бінарна сумісність з ARM926, ARM1136 і ARM1176
Cortex–A9 (2007р.)	Динамічна довжина конвеєра (8–11 етапів). До 4 ядер, Thumb–2, NEON, Jazelle DBX, оптимізований кеш 1 рівня до 64КБ, кеш 2 рівня до 8 КБ. Частота до 1.5 Гц (до 50% продуктивніший за Cortex–A8)
Cortex–A5 (2009р.)	Енергоефективний, багатоядерний (до 4 ядер), Thumb–2, SIMD, TrustZone, NEON, FPU (Floating–Point Unit), налаштовуваний кеш (4–64КБ). Частота до 1Гц. Бюджетний процесор для дешевих смартфонів і деяких мультимедійних пристроїв.

`

Продовження таблиці 1.1

Cortex–A15 (2010р.)	15–етапний конвеєр. Технологія big.LITTLE, Thumb–2, NEON, TrustZone, оптимізований кеш 1 рівня (32КБ + 32КБ), інтегрований налаштовуваний кеш 2 рівня до 4МБ, Large Physical Address Extensions (LPAE) до 1ТБ. Частота до 2–2.5Гц (до 40% продуктивніший за Cortex–A9)
Cortex–A7 (2011р.)	Має ті ж головні властивості і бінарну сумісність з Cortex–A15. Частота 1.2–1.6 Гц. Енергоефективний, для дешевих і середніх смартфонів і портативних пристроїв.
Cortex–A17 (2013р.)	15–етапний конвеєр. Технологія big.LITTLE, Thumb–2, NEON, TrustZone, оптимізований кеш 1 рівня (32–64КБ + 32КБ), інтегрований налаштовуваний кеш 2 рівня до 8МБ, LPAE до 1ТБ. Частота до 2.5Гц (до 60% продуктивніший за Cortex–A9)

Наприкінці 2011 р. була опублікована нова версія архітектури, ARMv8. У ній з'явилося визначення архітектури AArch64, в якій виконується 64–бітний набір команд A64. Підтримка 32–бітових команд отримала назву A32 і виконується на архітектурах AArch32. Інструкції Thumb підтримуються в режимі T32, тільки при використанні 32–бітних архітектур. Допускається виконання 32–бітних додатків в 64–бітній ОС і запуск віртуалізованої 32–бітної ОС за допомогою 64–бітного гіпервізора.

Особливості AArch64:

– новий набір команд A64;
– 31 регістр загального призначення, кожен довжиною 64 біт;
– окремі регістри SP і PC;

`

– інструкції мають розмір 32 біта, багато збігаються з командами A32;

– більшість інструкцій працюють як з 32–, так і з 64–бітними аргументами;

– адреси мають розмір 64 біта;

– підтримує обчислення з числами з плаваючою комою подвійної точності (64–біт double);

– нова система виняткових ситуацій;

– трансляція віртуальних адрес з 48–бітного формату працює за допомогою існуючих механізмів LPAE.

У 2012р. з'явилися перші процесори з новою архітектурою – *Cortex–A53* і *Cortex–A57*, які завдяки технології big.LITTLE можуть застосуватися разом. Підтримують до 16 ядер. Вони мають розширені можливості для управління кешем та інструкціями SIMD, що робить їх ідеальними процесорами для вимогливих мобільних додатків.

У 2014р. анонсовано процесор *Cortex–A72*. Заявлена частота 2.5Гц і в 3.5 рази вища енергоефективність порівняно з Cortex–A15. У широкий продаж він надійде у 2016р.

Процесор Cortex–R (таблиця 1.2) призначено для додатків реального часу, для критичних систем, де вирішальними є надійність і швидкість. Системи реального часу призначено для обробки даних, що швидко змінюються, і повинні бути достатньо чутливими для негайної обробки потоку даних без уповільнення. Тому процесори Cortex–R знаходять своє застосування в жорстких дисках, мережевому устаткуванні, у вбудованих критичних системах, як, наприклад, гальмівна система автомобіля. Існують як одноядерна, так і багатоядерна (до 4 ядер на одному чипі) реалізації. Для максимальної реактивності Cortex–R можуть мати тісно зв'язану пам'ять (tightly–coupled memory, TCM), а також модулі захисту пам'яті (memory protection unit, MPU) замість повного MMU.

`

Таблиця 1.2 – Характеристика процесорів Cortex–R

Cortex–R4 (2007р.)	8–етапний конвеєр з можливістю одночасного виконання двох інструкцій (dual issue) і предвибіркою коду (instruction pre–fetch), передбачення переходів. Thumb–2, DSP (Digital Signal Processing), FPU, TCM (8–12 регіонів), налаштовуваний кеш (4–64МБ), ECC (Error–correcting code). Частота до 1.4Гц
Cortex–R5 (2010р.)	TCM (12–16 регіонів), додано LLPP (Low Latency Peripheral Port) та ACP (Accelerator Coherency Port), удосконалена підтримка багатопроцесорної обробки для двоядерної конфігурації. Бінарна сумісність з Cortex–R4. Частота до 1.4Гц.
Cortex–R7 (2011р.)	11–етапний конвеєр. TCM (16 регіонів), додано GIC (Generic Interrupt Controller), SCU (Snoop Control Unit), повна підтримка SMP (Symmetric Multi–Processing) і AMP (Asymmetric Multi–Processing). Частота до 1.5Гц.

Процесор Cortex–M (таблиця 1.3) призначено для додатків до мікроконтролерів. Ці додатки не вимагають великої обчислювальної потужності, але потребують багато ліній введення і виведення, дуже малий форм–фактор, детерміновану реакцію на переривання і виключно низьке енергоспоживання. Cortex–M активно використовуються в Bluetooth–пристроях, контролерах сенсорного екрану, пристроях дистанційного керування.

`

Таблиця 1.3 – Характеристика процесорів Cortex–M

Cortex–M3 (2004р.)	3–етапний конвеєр з передбаченням переходів. Повний набір Thumb. Більший, ніж в Cortex–M0, набір Thumb–2 (включаючи команди ділення). Підтримка апаратного множення. MPU для 8 регіонів. Немасковане переривання, до 240 фізичних переривань, затримка переривання – 12 тактів.
Cortex–M1 (2007р.)	Оптимізоване ядро, спеціально для завантаження в чипи FPGA. Той самий набір інструкцій, що і в Cortex–M0. Незначне падіння продуктивності модуля множення (3–33 такти проти 1–32 у Cortex–M0).
Cortex–M0 (2009р.)	3–етапний конвеєр без передбачення переходів. Thumb (окрім команд CBZ, CBNZ, IT). Thumb–2 (тільки команди BL, DMB, DSB, ISB, MRS, MSR). Немасковане переривання, до 32 фізичних переривань, затримка переривання – 16 тактів.
Cortex–M4 (2010р.)	Додано DSP, підтримку додаткового FPU. Трохи швидший, ніж Cortex–M3, завдяки поліпшеному передбаченню переходів.
Cortex–M0+ (2012р.)	Той самий набір інструкцій, що і в Cortex–M0. Поєднує властивості Cortex–M3 і Cortex–M4. 2–етапний конвеєр для більшої енергоефективності. Додано Micro Trace Buffer для покращеного відлагодження та однотактний інтерфейс введення–виведення.

`

Продовження таблиці 1.3

Cortex–M7 (2014р.)	6–етапний суперскалярний конвеєр з передбаченням переходів. Додано ECC, кеш для інструкцій (до 64КБ), кеш для даних (до 64КБ), TCM для інструкцій (до 16МБ), TCM для даних (до 16МБ). Застосовується в автомобільній і промисловій автоматизації, медичних пристроях, обробці аудіо і відео, для розгортання IoT (Internet of Things)

1.3 Сімейство ядер ARM

Нумерація версій ARM–ядер не відповідає нумерації версій ARM–архітектури, тому часто виникає плутанина (таблиця 1.4). Версії архітектури записуються, як ARMv, а версії ядра – як ARM. Крім того, перше число у версії ядра не завжди однакове з першим числом у версії архітектури, до якої це ядро належить.

Наприклад, ядро ARM1 належить до архітектури ARMv1, а ядра ARM2 і ARM3 – до архітектури ARMv2. Хоча різниця між ARM2 і ARM3 більша, ніж між ARM2 і ARM1. Ядер ARM4 і ARM5 не існувало. Ядра ARM6 і ARM7 належать до архітектури ARMv3 і т.д. (таблиця 1.4). Після ARM11 всі ядра називають Cortex.

Таблиця 1.4 – Приналежність ядер ARM до архітектури

Архітектура	Ядро
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM8, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale

`

Продовження таблиці 1.4

Архітектура	Ядро
ARMv6	ARM11
ARMv6–M	Cortex–M0, Cortex–M0+, Cortex–M1
ARMv7–A	Cortex–A5, Cortex–A7, Cortex–A8, Cortex–A9, Cortex–A12, Cortex–A15
ARMv7–R	Cortex–R4, Cortex–R5, Cortex–R7
ARMv7–M	Cortex–M3
ARMv7E–M	Cortex–M4
ARMv8–A	Cortex–A53, Cortex–A57

`

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1) Назвіть характерні особливості CISC-архітектури.
2) Назвіть характерні особливості RISC-процесорів.
3) Які архітектури процесорів є RISC-подібними?
4) Як розуміти слова «скорочений набір команд» у RISC-процесорів?
5) Дайте визначення архітектурі ARM.
6) Де сьогодні широко використовуються ARM-контролери?
7) Коли відбувся перехід з класичної архітектури фон Неймана на модифіковану Гарвардську архітектуру ARM-контролерів?
8) Чим відрізняються Гарвардська архітектура та архітектури фон Неймана?
9) Які класи процесорів входять до сімейства Cortex ?
10) Охарактеризуйте процесор Cortex-A.
11) Охарактеризуйте процесор Cortex-R.
12) Охарактеризуйте процесор Cortex-M.
13) Чим відрізняються версії ARM–ядер від версій ARM– архітектури?
14) Назвіть відмінності між ARM7 та ARM7TDMI?

`

2 СТРУКТУРА ТИПОВОГО МІКРОКОНТРОЛЕРА З ARM–ЯДРОМ

2.1. Загальні відомості

Мікроконтролерне ядро ARM було розроблене однойменною англійською компанією, яка була організована в 1990 році. Назва ARM походить від "Advanced RISC Machines". Слід зауважити, що компанія спеціалізується суто на розробці мікропроцесорних ядер і периферійних блоків, при цьому не має виробничих потужностей з випуску мікроконтролерів. Компанія ARM постачає свої розробки в електронній формі, на основі якої клієнти конструюють свої власні мікроконтролери. Клієнтами компанії є понад 60 компаній–виробників напівпровідників, серед яких можна виділити таких популярних виробників на ринку напівпровідникових компонентів країн СНД, як Altera, Analog Devices, Atmel, Cirrus Logic, Fujitsu, MagnaChip (Hynix), Intel, Motorola, National Semiconductor, Philips , ST Microelectronics, Texas Instruments і т. ін.

В даний час архітектура ARM займає лідируючі позиції і охоплює значну частину ринку 32–розр. вбудованих RISC–мікропроцесорів. Поширеність даного ядра пояснюється його стандартністю, що надає можливість розробником більш гнучко використовувати, як свої, так і сторонні програмні напрацювання, як при переході на нове процесорний ARM–ядро, так і при міграціях між різними типами ARM–мікроконтролерів.

Деякі компанії використовують розроблені ARM–процесори для спеціальних застосувань, проте більшості вони потрібні для мобільних телефонів, систем управління автомобільними двигунами, лазерних принтерів PostScript та інших пристроїв масового застосування. Для всіх цих пристроїв необхідні такі якості, як висока швидкодія, помірна ціна і низьке енергоспоживання.

Фірмою розроблено цілий ряд 32–розрядних RISC–процесорів з різними можливостями і різною продуктивністю, а її процесор ARM, який розроблено ще в 1994 році, використовується до теперішнього часу.

`

Сама фірма визначає процесор ARM7 як універсальне ядро 32–розрядного RISC–мікропроцесора з малим енергоспоживанням, яке призначене для використання в різних замовних і спеціальних IC. Малі розміри RISC–ядра дозволяють успішно інтегрувати його в великі замовні схеми, які можуть містити RAM, ROM, DSP, додаткову логіку та інші елементи.

Основні характеристики ядра ARM7:

– 32–розрядний RISC–процесор (32–розрядні шини даних і адреси) з продуктивністю 17 MIPS при тактовій частоті 25 МГц (пікова продуктивність 25 MIPS);

– 32–розрядна адресація – лінійний адресний простір в 4 Гб – виключає потребу в сегментованій, розділеній на банки або оверлійній пам'яті;

– тридцять один 32–розрядний регістр загального призначення і шість регістрів стану;

– регістри адрес, запису і конвеєра;

– циклічний пристрій зі зсувом і перемножувач;

– трирівневий конвеєр (вибірка команди, її декодування і виконання);

– робочі режими Big Endian і Little Endian;

– напруга живлення 3,3 і 5 В;

– мале енергоспоживання 0,6 мА / МГц, при виготовленні за CMOS–технологією з топологічними нормами 0,8 мкм;

– повністю статична робота, що дозволяє додатково знижувати споживання за рахунок зменшення тактової частоти, що ідеально для критичних до споживання застосувань;

– швидке реагування на переривання застосувань реального масштабу часу;

– підтримка систем віртуальної пам'яті;

– проста, але потужна система команд.

`

В роботі розглядається ядро ARM7 та мікроконтролери LPC2378, STM32, які використовують це ядро.

2.2. Мікроконтролер LPC2378

LPC2378 – один із старших контролерів свого сімейства. Йому притаманні такі основні особливості:

- 32 бітне ядро ARM7TDMI–S;
- тактова частота до 72 МГц;
- 512 КБ Flash + 58 КБ SRAM;
- Ethernet 10/100 MAC + DMA;
- USB 2.0 в режимі Full–speed;
- двоканальний CAN 2.0B;
- контролер DMA;
- I2S, три I2C, три SPI / SSP, чотири UART;
- можлива робота з зовнішніми картами SD / MMC;
- АЦП (8 каналів, 10 біт), ЦАП (1 канал, 10 біт);
- і т. ін.

Більш повний перелік периферії наведено в документі «LPC2378 Preliminary datasheet».

Контролер зазвичай упаковується виробником в 144–вивідний корпус TQFP, зовнішній вигляд якого наведено на рисунку 2.1.

Зовні контролер нічим не примітний, окрім як великою кількістю виводів. Для того щоб визначити функцію того чи іншого виводу необхідно звернутися до відповідної документації. На рисунку 2.2 наведено рекомендоване електричне підключення контролера.

`

Рисунок 2.1 – Зовнішній вигляд мікроконтролера LPC2378 (вид зверху)

Для того, щоб контролер запустився, необхідно забезпечити його якісним живленням 3,3В і поставити необхідні фільтруючі конденсатори. Для внутрішніх потреб контролера (для живлення ядра) потрібно джерело напругою 1,8В, але контролери цього сімейства мають вбудований перетворювач напруги і, тому, потрібні лише конденсатори (С47…С49). Для тактування контролера потрібно як мінімум один кварц на входи X1, X2 (основний вхід тактового генератора) або на входи R1TCX1 + R1TCX2 (вхід тактового генератора для годинника реального часу). Кварц можна і не ставити, скориставшись внутрішнім (вбудованим в чіп) генератором, але через його низьку точність, використовувати деяку периферію не вдасться (так, наприклад USB працювати не буде, оскільки йому потрібні тактові сигнали з точними інтервалами часу, які не може забезпечити мініатюрний кварц). Великий 20–контактний роз'єм (JTAG) зліва вгорі (рисунок 2.2) призначено для налагодження та програмування контролера. При необхідності до нього підключається штекер з шлейфом, що йде від відлагоджувача JTAG. З використанням відповідного програмного забезпечення, наприклад, що входить в IDE Keil for ARM7, в контролер записується програма, яку програміст (користувач) набрав і відкомпілював. Програми створюються з використанням модифікації мови С, що включає розширення для програмування контролерів.

`

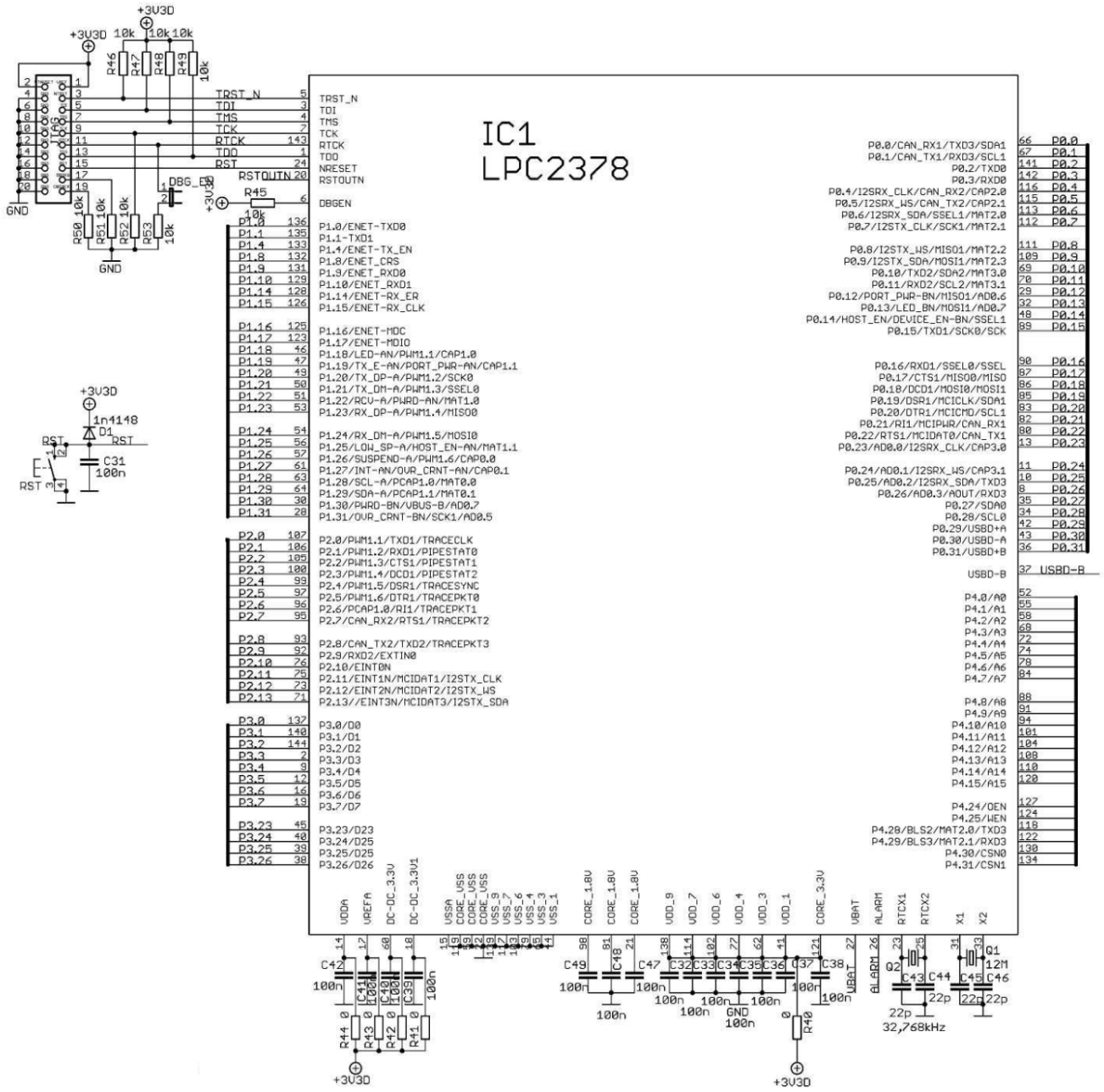


Рисунок 2.2 – Рекомендоване електричне підключення контролера

Нижче в таблиці 2.1 наведено порівняння мікроконтролера LPC2378 з іншими мікроконтролерами ряду LPC23xx.

`

Таблиця 2.1 – Характеристики мікроконтролерів LPC23xx

Table 2. LPC23xx features overview

Part	Local bus SRAM (kB)	Flash (kB)	EMC	USB/ GP SRAM (kB)	USB device	USB host/ OTG	Ethernet	Ethernet GP SRAM (kB)	CAN channels	SD/ MMC	ADC channels	GPIO pins
LPC2361	8	64	no	8	yes	yes	no	16	2	no	6	70
LPC2362	32	128	no	8	yes	yes	yes	16	2	no	6	70
LPC2364	8	128	no	8	yes	no	yes	16	2	no	6	70
LPC2365	32	256	no	8	no	no	yes	16	-	no	6	70
LPC2366	32	256	no	8	yes	no	yes	16	2	no	6	70
LPC2367	32	512	no	8	no	no	yes	16	-	yes	6	70
LPC2368	32	512	no	8	yes	no	yes	16	2	yes	6	70
LPC2377	32	512	Mini	8	no	no	yes	16	-	yes	8	104
LPC2378	32	512	Mini	8	yes	no	yes	16	2	yes	8	104
LPC2387	64	512	no	16	yes	yes	yes	16	2	yes	6	70
LPC2388	64	512	Mini	16	yes	yes	yes	16	2	yes	8	104

З таблиці видно, що контролер відрізняється наявністю USB–інтерфейсу, але не може виступати в якості хосту USB (тобто не може до себе підключати пристрої USB). Також у контролера є Ethernet–контролер, 8 аналогових входів (АЦП), і 104 виводи загального призначення. Кількості швидкої пам'яті (ОЗП) на локальній шині 32 Кб достатньо для виконання більшості завдань автоматики, так само як і постійної пам'яті Flash, об'ємом 512 Кб. При необхідності користувач може підключати зовнішні мікросхеми пам'яті і, таким чином, збільшити об`єм пам'яті.

Для того, щоб більш детально ознайомитися з контролером, розглянемо його структурну схему (рисунок 2.3).

Структура досить складна, тому її умовно розділено червоною смугою на важливі складові контролера: ядро і швидкі пристрої та іншу периферію (нижче червоної смуги). З першого погляду видно, що на контролері присутні кілька шин даних (AHB, APB, Local Bus – на рисунку без назви).

Найголовніший блок – ARM7TDMI–S – це ядро мікроконтролера, до якого відносяться всі основні функції по керуванню пам'яттю, обчисленням, обробці програми і т.ін.

`

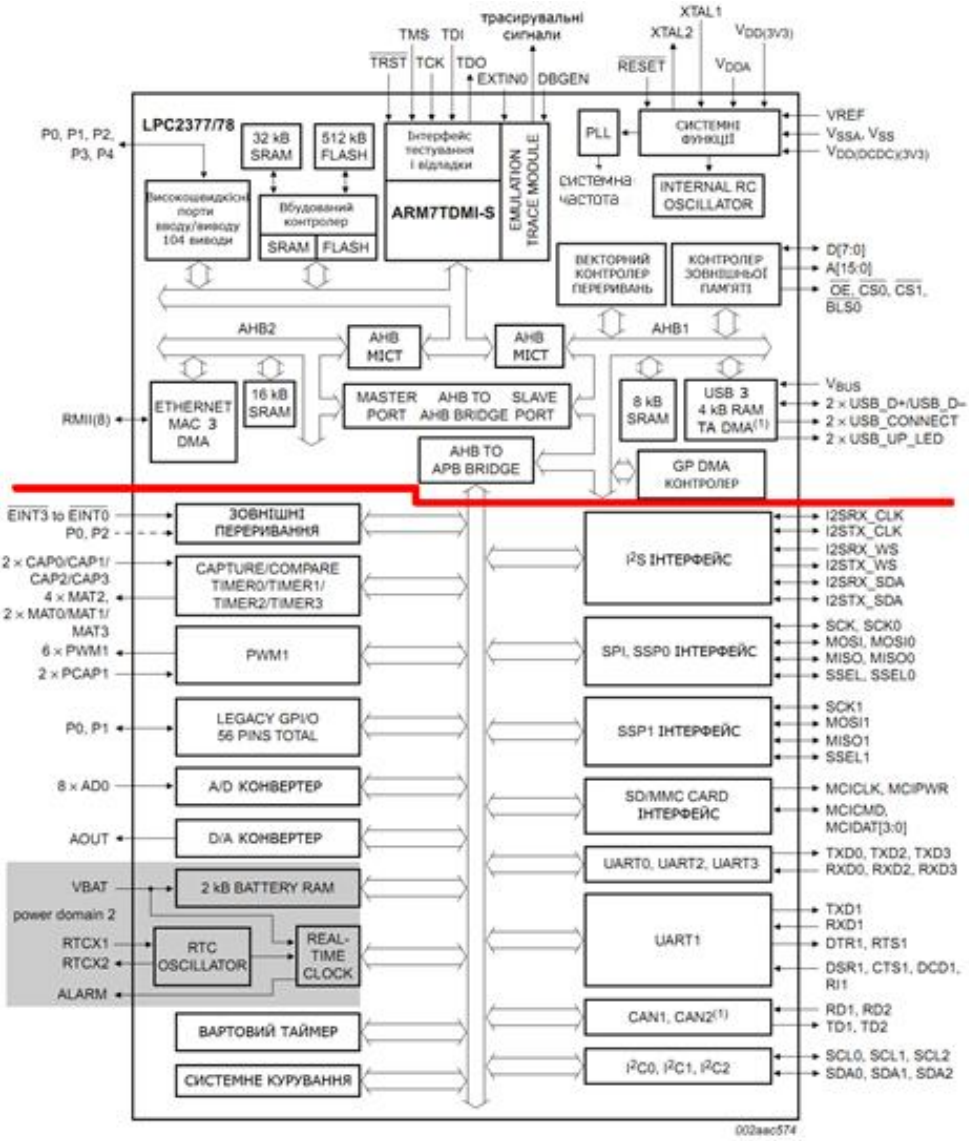


Рисунок 2.3 – Структурна схема мікроконтролера LPC2378

Блок System Functions (блок системних функцій) призначено для контролю роботи контролера, скидання (вхід RESET) та тактування (XTAL1, XTAL2). До блоку приєднано блоки вбудованого низькоточного RC–осцилятора (Internal RC Oscillator), з якого здійснюється запуск контролера, а також фазового автопідстроювання частоти (PLL = Phase Locked Loop), що відповідає за розподіл частоти і тактування контролера. Не дивлячись на відсутність зв'язку на структурній схемі від ядра до цих блоків, існує можливість налаштовувати їх роботу з програми через регістри (не обов'язково під час запуску контролера).

`

На рисунку 2.4 наведено ядро та швидкі пристрої з перекладом на українську мову деяких блоків.

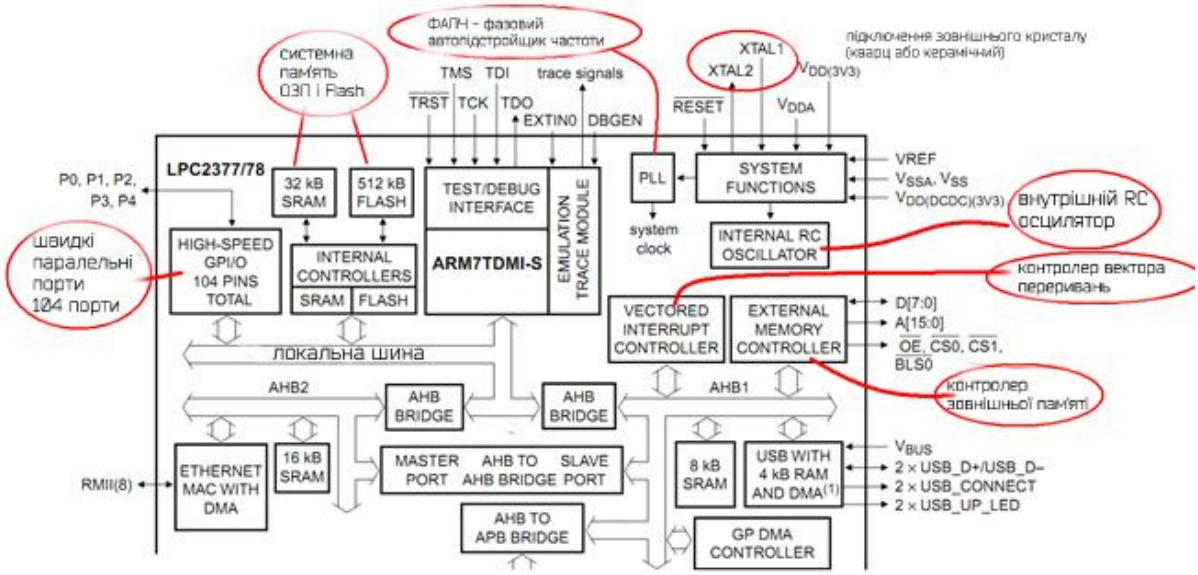


Рисунок 2.4 – Швидка периферія і ядро контролера LPC2378

Зв'язок з периферією проводиться за допомогою наступних спеціальних шин (рисунок 2.5):

– ARM7 local bus (локальна шина). Шина забезпечує швидкий доступ до пам'яті чіпа та швидкий обмін через паралельний порт (GPIO);

– AHB (Advanced High performance Bus – покращена високопродуктивна шина) – зв'язує ядро зі швидкістною периферією і зовнішнюю пам'яттю. На контролері є дві шини: AHB1 і AHB2;

– APB (Advanced Peripheral Bus–покращена периферійна шина) – зв'язує інші периферійні пристрої на чипі;

– Шина AHB1 – зв'язує ядро з:

– VIC (vector interrupt controller) – векторний контролер переривань (32 вектори) та контролер USB;

– GP DMA – контролер прямого доступу до пам'яті загального призначення (дозволяє скоротити кількість переривань при роботі з модулями, з

`

якими проводиться інтенсивний обмін даними через буфери, наприклад Ethernet, USB і т.ін.);

– EMC – контролер зовнішньої пам'яті. Забезпечує доступ до зовнішньої пам'яті (Flash) і різної зовнішньої периферії, яка є проектована на адресний простір пам'яті.

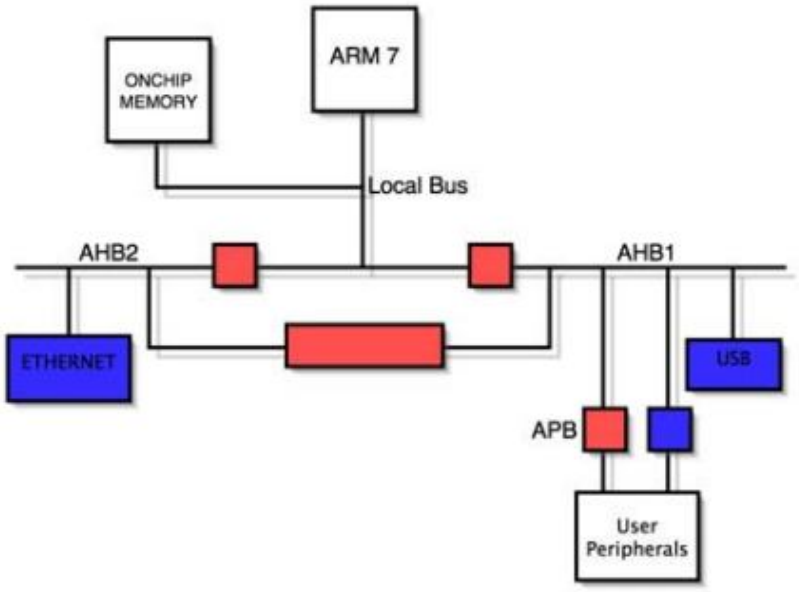


Рисунок 2.5 – Зв'язок шин даних контролера LPC2378

Шина AHB2 – зв'язує тільки ядро і Ethernet модуль (в тому числі і внутрішню пам'ять Ethernet = 16 КБ SRAM). AHB2 виділена в окрему шину, щоб не заважати роботі іншим пристроям, тому що обсяг даних, який передається через Ethernet може бути значним і повністю завантажити периферійну високопродуктивну шину даних.

Шини AHB незалежні, але можуть бути підключені в режим Master–Slave (рисунок 2.6).

У такому разі:

– AHB2 працює в режимі Master;

– AHB1 – в режимі Slave;

`

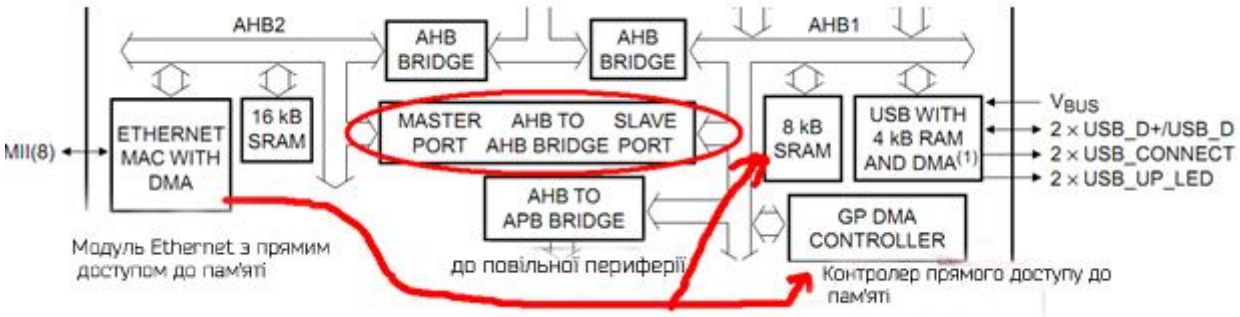


Рисунок 2.6 – З'єднання периферійних шин AHB1 і AHB2 через міст AHB–to–AHB

– AHB2 може запитувати з основної пам'яті контролера чи зовнішньої пам'яті додаткове місце під буфер Ethernet;

– адресація шиною AHB завжди 32–бітна, та прозора для програміста, тобто програміст напряму не працює з шиною;

– усім пристоям на шині AHB виділяється адресний простір нижче адреси 4.0 Гб (0xFFFF FFFF);

– кожен пристрій отримує блок 16 Кб (рисунок 2.7).

Також важливо зазначити наступні моменти, які викликані такою конфігурацією шин даних:

– пам’ять SRAM буфера Ethernet 16 КБ і пам'ять SRAM, яка використовується контролером GP DMA = 8 КБ, також може бути використана для зберігання даних або коду;

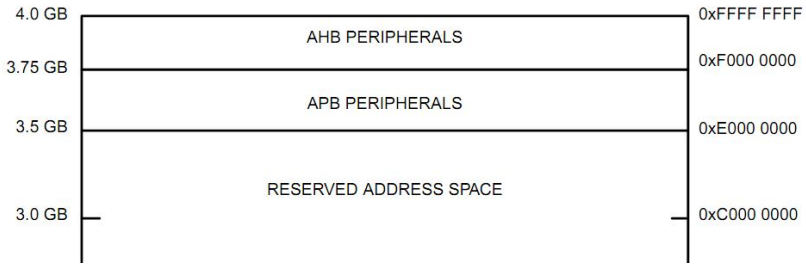


Рисунок 2.7 – Поділ на блоки адресного простору пристроїв шин AHB і APB

`

– пам'ять 2 Кб годинника реального часу (RTC), можна використовувати тільки для зберігання даних. Оскільки вона належить до повільної шини, зчитувати з неї код недоцільно, тому що це уповільнило б ядро контролера, який постійно очікує передачі нової порції коду;

– пам'ять годинника реального часу (RTC) можна використовувати в ролі NVRAM (Non–volatile random–access memory – енергонезалежної пам'яті), оскільки вона живиться від батарейки і її вміст зберігається у вимкненому стані.

– шина APB (Advanced Peripheral Bus – покращена периферійна шина) – призначена для розщеплення і з'єднання безлічі більш повільних периферійних пристроїв.

APB – ще більш повільна шина, ніж AHB. Зв'язок з APB здійснюється за допомогою мосту AHB–APB. Пристрої на шині APB також адресуються блоком, який лежить вище 3.5 Гб (рисунок 2.7). Блок займає 2 Мб. Кожен пристрій в блоці резервує за собою 16 Кб адресного простору із загального блоку (послідовно).

На рисунку 2.8 наведено пристрої, які з'єднуються шиною APB.

Кожний периферійний пристрій має по одній лінії, що сполучає його і VIC (векторний контролер переривань).

Будь–яка ніжка порту Port0 або Port2, незалежно від функцій, може бути запрограмована на генерацію переривань (сумарно 46 ніжок):

– за наростанням;

– за спаданням;

– за наростанням і спаданням вхідного сигналу.

Ці запити переривань будуть скомбіновані із запитом переривання EINT3.

`

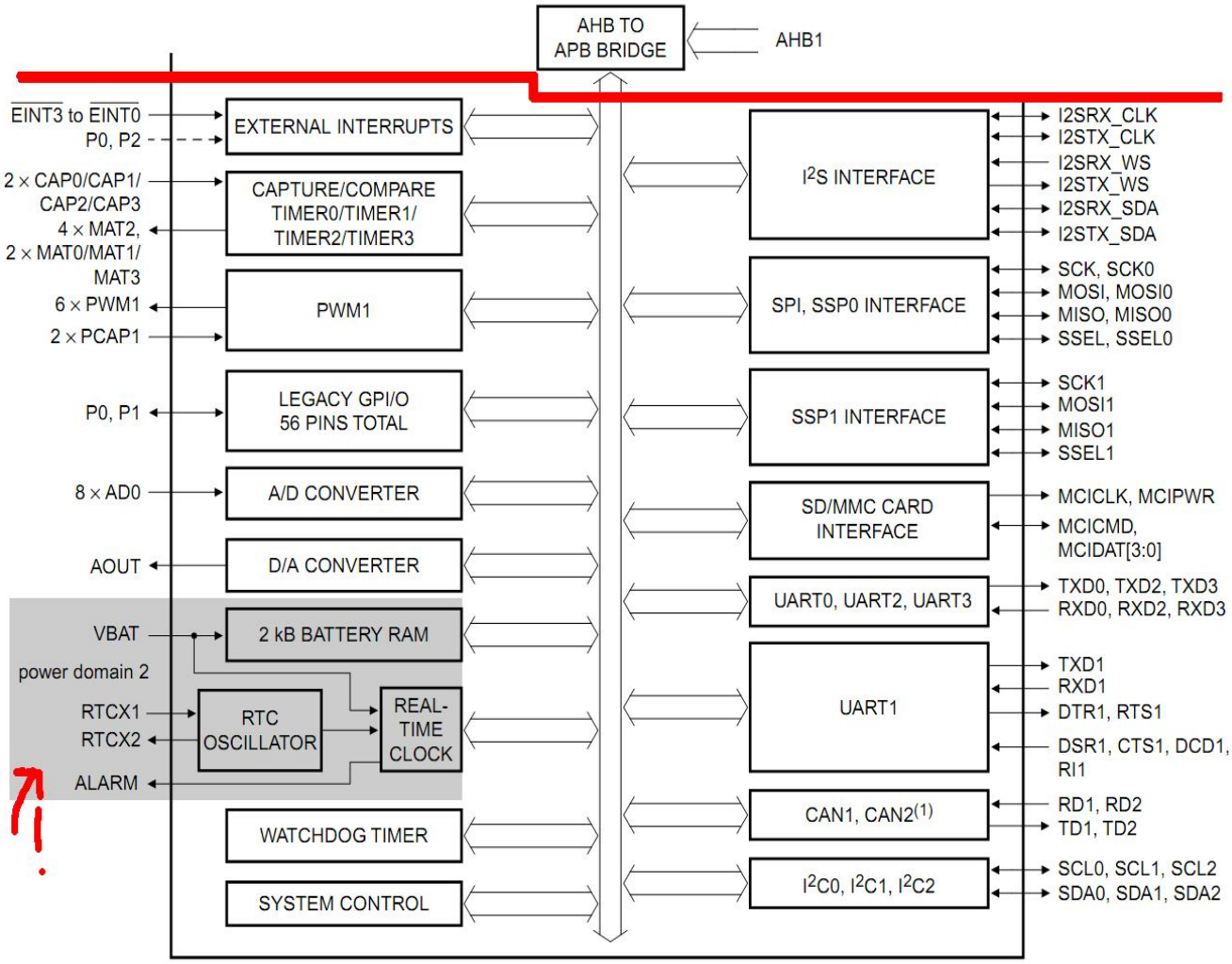


Рисунок 2.8 – Пристрої, які з’єднуються шиною APB

2.3. Мікроконтролер STM32

Враховуючи велику кількість компаній, що виробляють мікроконтролери архітектури ARM, велику кількість цих контролерів навіть у однієї компанії, коротко зупинимося на контролерах серії STM32 компанії STMicroelectronics.

STMicroelectronics – європейська мікроелектронна компанія, одна з найбільших, що займаються розробкою, виготовленням та продажем різних напівпровідникових електронних і мікроелектронних компонентів. Сьогодні штаб–квартира компанії знаходиться в Женеві, в той же час, її холдингова компанія STMicroelectronics NV зареєстрована в Амстердамі, однак компанія

`

історично пов'язана з Італією та Францією, де значно взаємодіє з ринком. Компанія має представництва в США, Китаї та Японії.

Мікроконтролери STM32 використовують ядро Cortex–M3. Сімейство ARM Cortex – нове покоління процесорів, які виконані за стандартною архітектурою і відповідають різним технологічним вимогам. На відміну від інших ЦПП (цифровий процесорний пристрій) ARM, сімейство Cortex є завершеним процесорним ядром, яке об'єднує стандартне ЦПП і системну архітектуру. Сімейство Cortex доступно в трьох основних профілях: профіль A для високопродуктивних застосувань, профіль R для застосувань у реальному часі і профіль M для чутливих до вартості мікроконтролерних застосувань.

Мікроконтролери STM32 виконано на основі профілю Cortex–M3, яке спеціально розроблене для застосувань, де необхідні розвинені системні ресурси і, при цьому, мале енергоспоживання. Вони характеризуються настільки низькою вартістю, що можуть конкурувати з традиційними 8 і 16–бітними мікроконтролерами. І хоча ЦПП ARM7 і ARM9 були з успіхом інтегровані в стандартні мікроконтролери, в них все ж таки простежується початкова орієнтованість на системи на кристалі (SoC). Це особливо помітно за способами обробки виняткових ситуацій та переривань, тому у різних виробників мікроконтролерів ці способи обробки реалізовано різним чином. Cortex–M3 є стандартизованим мікроконтролерним ядром, яке крім ЦПП, містить всі інші основні елементи мікроконтролера (в т.ч. система переривань, системний таймер SysTick, відлагоджувальна система та карта пам'яті). 4 гігабайтний адресний простір Cortex–M3 розділено на чітко розподілені області коду програми, статичного ОЗП, пристроїв введення–виведення і системних ресурсів.

На відміну від ядра ARM7, Cortex–M3 виконано за Гарвардською архітектурою і, тому, має декілька шин, що дозволяють виконувати операції паралельно. Сімейство Cortex має можливість оперувати з фрагментованими даними (unaligned data), що також відрізняє його від попередніх архітектур ARM. Цим гарантується максимальна ефективність використання внутріш-

`

нього статичного ОЗП. Сімейство Cortex також підтримує можливості встановлення і скидання біт в межах двох областей пам'яті розміром 1 Мбайт за методом bit banding. Цей метод надає ефективний доступ до регістрів і прапорців ПВВ, розташованих в області статичного ОЗП, і виключає необхідність інтеграції повнофункціонального бітового процесора.

Основою STM32 є процесор Cortex–M3. Він являє собою стандартизований мікроконтролер, який інтегрує 32–бітний ЦПП, шинну структуру, блок вкладених переривань, відлагоджувальну систему і відповідну організацію пам'яті.

До появи STM32 компанія ST вже мала у своєму асортименті 4 сімейства мікроконтролерів на основі ядер ARM7 і ARM9, однак саме у мікроконтролерів STM32 було досягнуто суттєве поліпшення співвідношення вартості і робочих характеристик. Мікроконтролери STM32, ціна яких за штуку при покупці великих кількостей становить трохи більше одного Євро, кидають серйозний виклик існуючим 8–бітним мікроконтролерам. Мікроконтролери STM32, які спочатку випускалися в 14 різних варіантах, розділено на дві групи: Performance Line, до якої увійшли мікроконтролери з тактовою частотою ЦПП до 72 МГц, і Access Line (тактова частота до 36 МГц). Обидві групи мікроконтролерів сумісні за розташуванням виводів і програмному забезпеченню. Обсяг їх вбудованої Flash–пам'яті досягає 128 Кбайт, а статичного ОЗП – 20 Кбайт. З моменту появи перших мікроконтролерів STM32 їх асортимент був істотно розширений новими представниками з підвищеними розмірами ОЗП і Flash–пам'яті, а також з більш складними ПВВ.

Сімейство STM32 складається з двох груп. Група Performance Line працює на тактових частотах до 72 МГц й оснащена повним набором ПВВ, а група Access Line працює на частотах до 36 МГц та інтегрує обмежений набір ПВВ.

Сімейство STM32 – це не тільки мікроконтролери на ядрі Cortex–M3. Архітектура Cortex–M включає в себе також ядра Cortex–M0 і Cortex–M4.

`

Cortex–M0 – це Cortex–M3 з скороченим набором команд, призначено для більш дешевих і менш вимогливих з точки зору продуктивності рішень. Cortex–M0 дозволить замінити 16–бітові мікроконтролери і, в меншій мірі, 8–бітові мікроконтролери. Cortex–M4 – це Cortex–M3, збагачено новими командами для обробки даних і призначено для застосувань, що вимагають більш високої продуктивності, з більш складною обробкою сигналу (операції з плаваючою комою на апаратному рівні). Cortex–M4 можна буде використовувати в нижньому сегменті DSP–додатків.

Програмний код, що працює на ядрі Cortex–M0, буде в повному обсязі працювати і на ядрі Cortex–M3, оскільки для Cortex–M3 діють всі інструкції Cortex–M0. Програмний код, що працює на ядрі Cortex–M3, також буде працювати на Cortex–M4, оскільки для Cortex–M4 залишаються чинними всі інструкції Cortex–M3. Тобто, зробивши виріб на Cortex–M3, можна буде далі зробити його більш дешеві і прості варіанти на Cortex–M0 або більш дорогі і складні вироби на Cortex–M4 з мінімальними витратами на переробку програмного коду. Оскільки Cortex–M3 вже став світовим стандартом, і оскільки Cortex–M0 і Cortex–M4 є натуральними продовженнями Cortex–M3, нікого не здивує, якщо вони також стануть стандартами найближчим часом. Інші виробники також активно працюють у цьому напрямку (Texas Instruments, Freescale, NXP і т.ін.)

У підсумку можна сказати, що обираючи STM32, розробник обирає досить популярний мікроконтролер на Cortex–M3, з перспективою переходу на інші ядра Cortex–M, але при цьому не закриває двері для продукції інших виробників.

`

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1) Ким було розроблено мікроконтролерне ядро ARM?
2) Від чого походить назва ARM?
3) На чому спеціалізується компанія "Advanced RISC Machines"?
4) Які компанії є клієнтами компанії ARM?
5) Які основні характеристики ядра ARM7?
6) Назвіть основні особливості мікроконтролера LPC2378.
7) Опишіть відмінності мікроконтролера LPC2378 від інших мікроконтролерів ряду LPC23xx.
8) Охарактеризуйте детальніше такі особливості мікроконтролера LPC2378 як:
   - структурна схема;
   - швидка периферія і ядро контролера;
   - зв'язок з периферією;
   - шина AHB1;
   - шина AHB2;
   - міст AHB-to-AHB.
9) Чим займається компанія STMicroelectronics?
10) На основі якого ядра виконані мікроконтролери STM32?
11) Охарактеризуйте склад та основні характеристики мікроконтролера STM32.
12) За якою архітектурою виконано мікроконтролер STM32?
13) Опишіть сімейство STM32.

3 ХАРАКТЕРИСТИКА ЯДРА ARM–МІКРОКОНТРОЛЕРІВ

3.1 Основні положення

Характеристику ядра ARM–мікроконтролерів розглянемо на прикладі сімейства LPC2300, побудованого на основі ЦПП ARM7. Щоб використовувати ці мікроконтролери, вам зовсім не потрібно бути експертом в області програмування процесора ARM7, оскільки турботу про більшість складних моментів бере на себе компілятор мови С. Тим не менш, щоб розробити надійний пристрій, ви повинні мати хоча б загальне уявлення про те, як працює ЦПП і які у нього є особливості.

У цьому розділі ми розглянемо основні характеристики ядра ARM7 разом з його моделлю програмування, а також коротко обговоримо набір команд, який використовується даним процесором. В результаті ви отримаєте всю необхідну інформацію про процесор, що є «серцем» сімейства LPC2300. Для більш поглибленого вивчення процесорів ARM рекомендуємо звернутися до книг, зазначених в списках літератури.

Головний принцип, що лежить в основі процесора ARM, це простота. Ядро ARM7 є RISC–машиною, яка передбачає використання невеликої кількості команд і відповідно складається з відносно невеликої кількості логічних елементів. Завдяки цьому процесор ARM7 ідеально підходить для використання у вбудованих системах. Він має високу продуктивність, низьке енергоспоживання і займає невелику частину загальної площі кристала.

3.2 Конвейер команд

Основний елемент ЦПП ARM7 це конвеєр команд, який використовується для обробки команд, які зчитуються з пам'яті програм. Конкретно, в ядрі ARM7 реалізовано триступеневий конвеєр (рисунок 3.1).

`

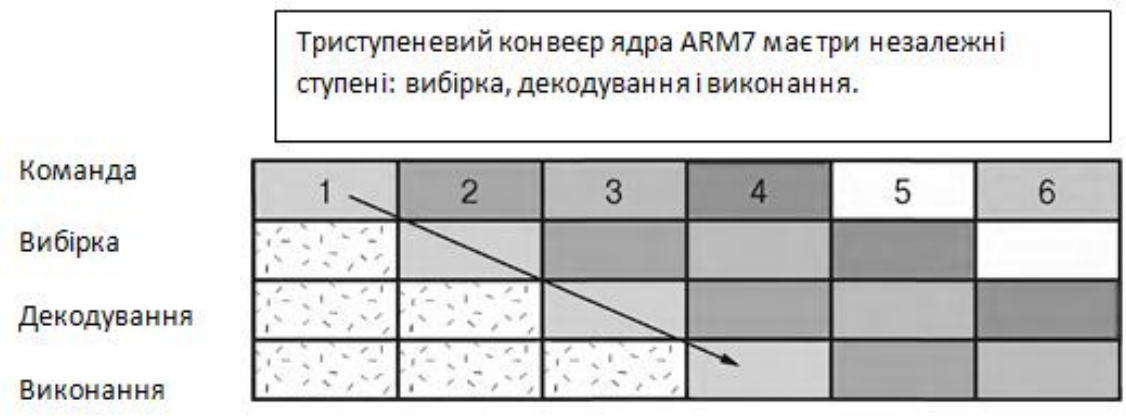


Рисунок 3.1 – Робота триступеневого конвеєра

Триступеневий конвеєр є найпростішою різновидністю конвеєрів і не схильний до виникнення різних небезпечних ситуацій, таких як «читання раніше запису», які зустрічаються в конвеєрах з великим числом ступенів.

Конвеєр має три апаратно–незалежні ступені, завдяки яким одночасно з виконанням однієї команди здійснюється декодування другої і вибірка третьої.

Він настільки ефективно прискорює проходження команд через ЦПП, що більшість команд ARM можуть виконуватися за один такт. Конвеєр найбільш ефективний при виконанні лінійного коду. При виявленні переходу конвеєр скидається, і для відновлення виконання програми з максимальною швидкістю він повинен спочатку заповнитися. Пізніше ми з вами побачимо, що набір команд процесора ARM має кілька цікавих особливостей, що дозволяють виключити з коду короткі переходи для поліпшення проходження коду по конвеєру. Оскільки конвеєр є складовою частиною ЦПП, він повністю прихований від програміста. Однак, важливо пам'ятати, що значення лічильника команд (Program Counter – PC) на 8 байт перевищує значення адреси поточної виконуваної команди. У зв'язку з цим необхідно обережно підходити до обчислення зсувів у разі відносної адресації з використанням лічильника команд.

`

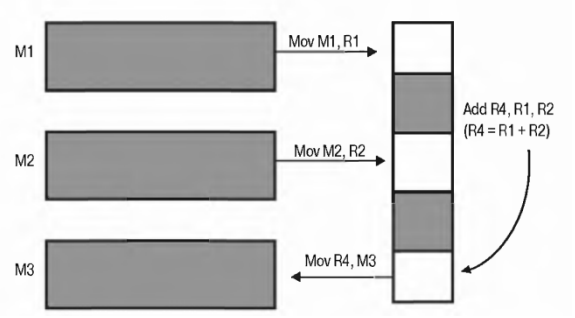
Наприклад, команда:

```
0x4000          LDR          PC, [PC, # 4]
```

завантажить в лічильник команд PC вміст, що знаходиться за адресою PC + 4. Оскільки PC випереджає поточну команду на 8 байт, в нього буде додано вміст за адресою 0x400C, а не 0x4004.

3.3 Регістри регістрового файлу

Процесор ARM7 має архітектуру «load–and–store» (завантаження – збереження), тому для виконання будь–якої команди обробки даних необхідно спочатку перенести ці дані з пам'яті в певні регістри, виконати команду обробки даних і потім записати отримані значення назад в пам’ять (рисунок 3.2).



ЦПП ARM7 має архітектуру «Load-and-store». Усі команди обробки даних можуть працювати тільки з основним регістровим файлом.

Рисунок 3.2 – Обробка даних

Основний регістровий файл складається з 16 призначених регістрів: R0 ... R15 (рисунок 3.3).

Кожен з цих регістрів є 32–бітовим. У вітчизняній літературі прийнято користуватися поняттями «розряд», «розрядний». Двійковий розряд – це біт. У даних лекціях ми будемо дотримуватися зарубіжної термінології («біт», «бітний»), що більш відповідає сучасній тенденції в цифровій техніці.

`

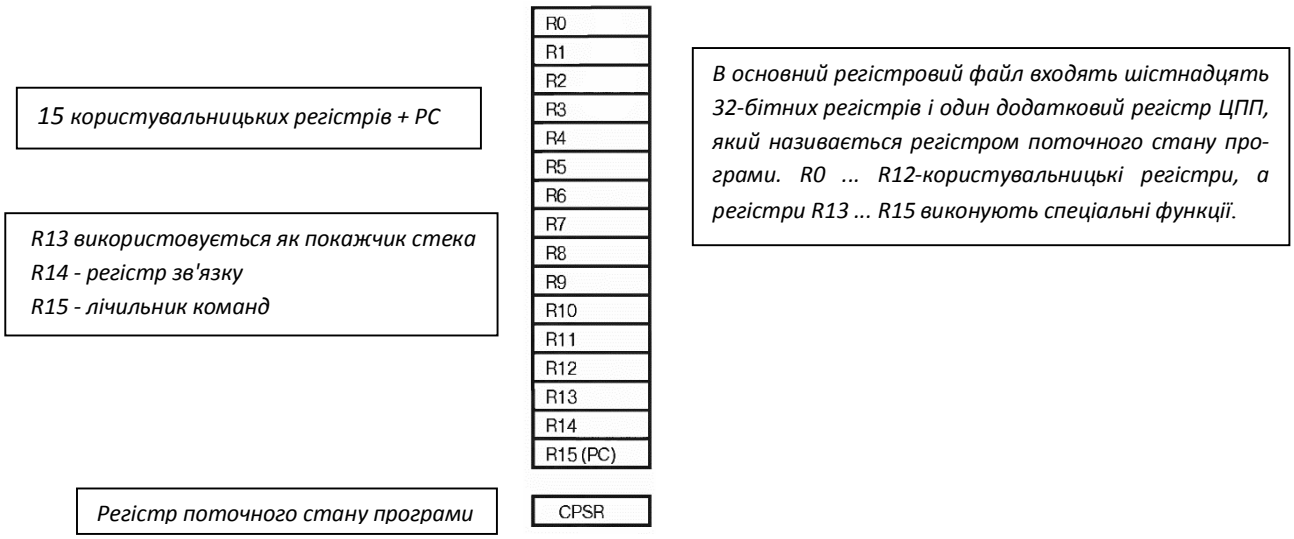


Рисунок 3.3 – Структура основного регістрового файлу

Регістри R0 ... R12 призначено виключно для потреб користувача і не виконують ніяких інших функцій, в той час як регістри R13 ... R15 мають додаткові функції. Регістр R13 використовується як покажчик стека (Stack Pointer – SP). Регістр R14 називається регістром зв'язку (Link Register – LR). При виклику підпрограми адреса повернення автоматично запам'ятовується в регістрі зв'язку, звідки потім зчитується при поверненні. Таке рішення дозволяє швидко переходити до «кінцевих» функцій (функції, які не викликають інших функцій) і повертатися з них. Якщо функція входить до складу «гілки», тобто викликає інші функції, вміст регістра зв'язку необхідно зберігати в стеку (R13). Нарешті, регістр R15 виконує функції лічильника команд (PC). Що цікаво, багато команд можуть працювати з регістрами R13 ... R15, як із звичайними користувацькими регістрами.

3.4 Регістр поточного стану програми

Поряд з банком регістрів у ЦПП є додатковий 32–бітний регістр, який називається регістром поточного стану програми (Current Program Status Register – CPSR). Регістр CPSR містить набір прапорців, які керують функціонуванням ЦПП ARM7 і відображають його стан (рисунок 3.4).

`

Рисунок 3.4 – Регістр поточного стану програми

У старших чотирьох бітах регістра CPSR зберігаються прапорці умов, значення яких визначаються процесором. Ці прапорці відображають результат виконання чергової команди обробки даних. Завдяки їм ви можете дізнатися, чи не було отримано в результаті виконання команди від'ємне або нульове значення, а також чи не відбулося перенесення або переповнення. Молодші вісім бітів регістра CPSR містять прапорці, значення яких прикладна програма може змінювати. Біти 7 і 8 є прапорцями I і F відповідно. Ці прапорці використовуються для дозволу і заборони двох ліній переривань, які є зовнішніми по відношенню до ЦПП ARM7.

Як ми з вами побачимо пізніше, усі периферійні модулі мікроконтролерів LPC2300 підключені до цих двох ліній переривань. При роботі з даними бітами потрібно дотримуватися обережності, оскільки для заборони будь–якого з джерел переривань у відповідний біт необхідно записати 1, а не 0, як можна було б припустити. П'ятий біт регістра є прапорцем THUMB.

ЦПП ARM7 підтримує два набори команд – 32–бітний набір команд ARM і 16–бітний набір команд THUMB. Відповідно прапорець T показує, який з наборів команд використовується. Врахуйте, що програма не повинна

`

намагатися безпосередньо встановлювати або скидати цей прапорець для перемикання між наборами команд. Коректний механізм зміни поточного набору команд ми розглянемо трохи пізніше. Останні п'ять молодших бітів регістра CPSR є прапорцями режиму. Процесор ARM7 підтримує 7 режимів роботи (рисунок 3.5).

ЦПП ARM7 має 6 різних робочих режимів, які використовуються для обробки виняткових ситуацій. Затінені регістри представляють собою дублюючі регістри, які «включаються» при зміні режиму роботи. Регістр SPSR використовується для збереження вмісту регістра CPSR при перемиканні режимів.

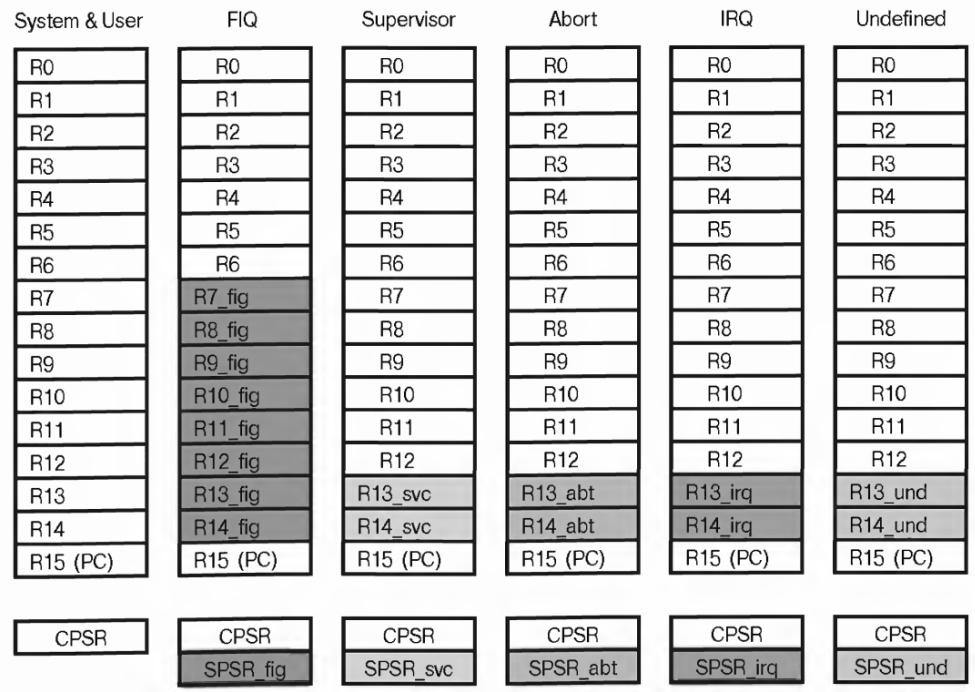


Рисунок 3.5 – Режими роботи процесора

Прикладні програми, як правило, виконуються в режимі User. У цьому режимі програма може звертатися до регістрів R0 ... R15 і CPSR. Однак при виникненні виняткових ситуацій (таких як переривання, помилка пам'яті, або виконання команди генерації програмного переривання) режим роботи процесора змінюється. При цьому регістри R0 ... R12 і R15 залишаються тими ж самими, а регістри R13 (SP) і R14 (LR) замінюються новою парою регістрів, унікальної для кожного режиму. Таким чином, кожен режим має власні регістр зв'язку та покажчик стека. Більш того, в режимі швидких переривань

`

(FIQ) дублюються і регістри R7 ... R12. Це дозволяє відразу приступити до обробки переривання FIQ, не витрачаючи час на збереження регістрів в стеку.

У кожному з режимів, за винятком режиму User, є додатковий регістр, який називається регістром збереженого стану програми (Saved Program Status Register – SPSR). Якщо в момент виникнення виняткової ситуації програма знаходилася в режимі User, відбувається зміна режиму і поточний зміст регістра CPSR зберігається в регістрі SPSR.

Після обробки виняткової ситуації (при поверненні з обробника) вміст регістра CPSR відновлюється з SPSR, забезпечуючи відновлення виконання прикладної програми.

3.5 Режими обробки виняткових ситуацій

При виникненні виняткової ситуації змінюється режим роботи ЦПП, і в PC завантажується адреса відповідного вектора переривання (таблиця 3.1). Таблиця векторів починається з нульової адреси. Першим в таблиці розташовано вектор скидання, а за ним інші вектори (по 4 байти на кожен).

При одночасному виникненні декількох виняткових ситуацій використовується метод пріоритетів. Пріоритети переривань наведено в таблиці 3.2.

Коли виникає виняткова ситуація, наприклад, переривання IRQ, процесор виконує наступні дії (рисунок 3.6). По–перше, адреса наступної виконуваної команди (PC + 4) зберігається в регістрі зв’язку.

Таблиця 3.1 – Адреси векторів переривань

Виняткова ситуація	**Режим**	**Адреса вектора**
Reset (скидання)	Supervisor	0x00000000
Undefined instruction (невизначена команда)	Undefined	0x00000004

Кожному режиму роботи відповідає свій вектор переривання. При зміні процесором режиму проводиться перехід з цього вектора. Зверніть увагу! Вектор за адресою 0x00000014 відсутній!

`

Продовження таблиці 3.1

Виняткова ситуація	**Режим**	**Адреса вектора**
SWI (програмне переривання)	Supervisor	0x00000008
Prefetch Abort (помилка звернення до пам'яті при вибірці команди)	Abort	0x0000000C
Data Abort (помилка звернення до пам'яті при доступі до даних)	Abort	0x00000010
IRQ (переривання)	IRQ	0x00000018
FIQ (швидке переривання)	FIQ	0x0000001C

Зауваження. *У таблиці векторів є «дірка», оскільки вектор з адресою 0x00000014 відсутній. Ця адреса використовувався в попередніх версіях процесорів ARM, а в процесорі ARM7 він збережений, щоб забезпечити програмну сумісність між різними архітектурами ARM. Однак, як ми побачимо пізніше, в мікроконтролерах сімейства LPC2300, ці чотири байти грають дуже важливу роль.*

Таблиця 3.2 – Пріоритети переривань

Пріоритет		**Виняткова ситуація**
Найвищий	1	Reset
	2	Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
Найнижчий	6	Undefined instruction / SWI

Кожне джерело виняткової ситуації має фіксований пріоритет. Вбудовані периферійні пристрої обслуговуються перериваннями FIQ і IRQ. Пріоритети переривань від периферійних пристроїв можна призначати всередині цих груп.

Потім регістр CPSR копіюється в регістр SPSR кінцевого режиму (в нашому випадку SPSR irq). Після цього в PC заноситься адреса вектора переривання режиму виняткової ситуації. Для режиму IRQ це адреса – 0x00000018. У той самий час режим роботи процесора змінюється на IRQ, в результаті чого регістри R13 і R14 замінюються відповідними регістрами цього режиму.

`

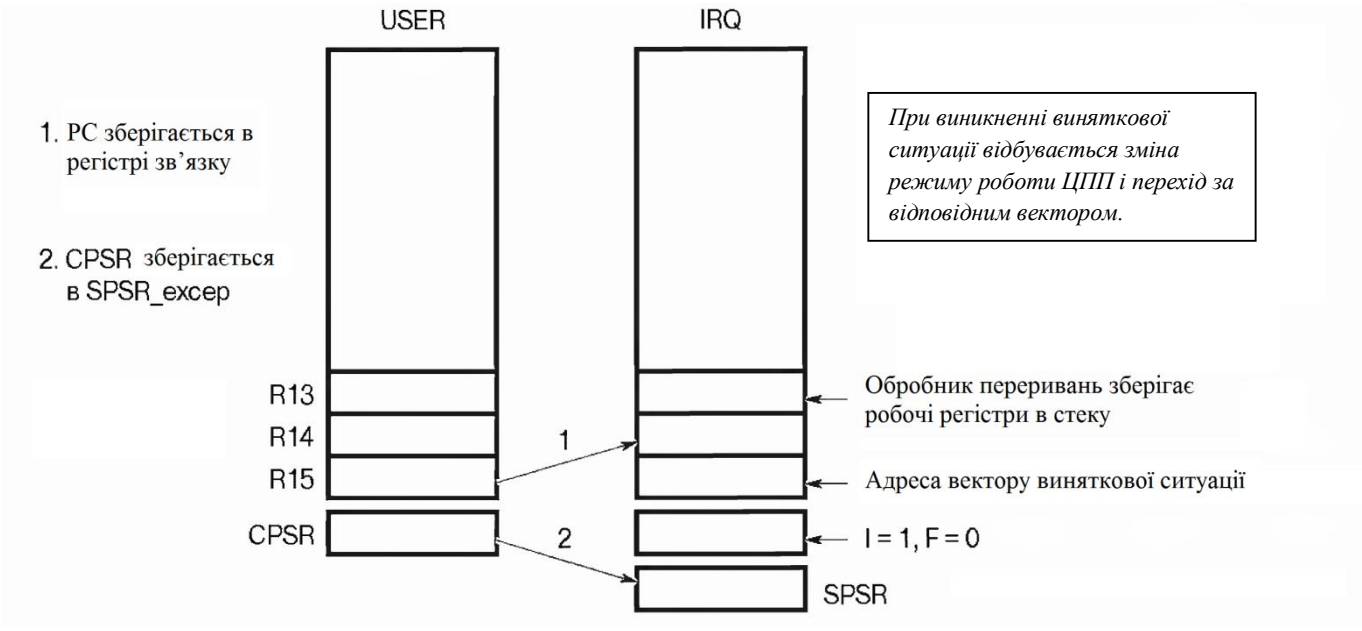


Рисунок 3.6 – Обробка виняткових ситуацій

При вході в режим IRQ встановлюється прапорець I регістра CPSR, що призводить до відключення лінії IRQ. Якщо потрібно використовувати вкладені переривання, то треба вручну дозволити переривання IRQ в програмі і занести вміст регістра зв'язку в стек, щоб зберегти вихідну адресу повернення.

З вектора переривання програма перейде до виконання підпрограми обробки переривань. Перше, що необхідно зробити в цій підпрограмі, – зберегти у стеку IRQ всі регістри з діапазону R0 ... R12, які будуть в ній використовуватися. Потім можна приступати до обробки виняткової ситуації.

Після завершення обробки виняткової ситуації необхідно повернутися в режим User і продовжити виконання програми з перерваного місця. Проте в наборі команд ARM відсутні команди типу «повернення» або «повернення з підпрограми», тому маніпуляції з лічильником команд PC необхідно здійснювати, використовуючи звичайні команди. Ситуація ускладнюється тим, що існує декілька різних варіантів повернення.

Для початку глянемо на команду SWI. При виконанні цієї команди адреса наступної виконуваної команди зберігається в регістрі зв'язку, після чо-

`

го проводиться обробка виняткової ситуації. Все, що потрібно зробити для повернення з виняткової ситуації, – це завантажити вміст регістра зв'язку назад в PC, і програма продовжить своє виконання.

Однак щоб ЦПП при цьому переключився назад у режим User, необхідно використовувати спеціальну команду пересилання MOVS (більш докладно ми розглянемо її трохи пізніше). Таким чином, команда повернення з програмного переривання буде наступною:

```
MOVS   R15,   R14              ; Скопіювати регістр зв'язку в PC і переключити режими.
```

А при виникненні виняткової ситуації за перериваннями FIQ і IRQ, поточна виконувана команда скидається і виконується перехід до обробника виняткової ситуації. При поверненні з виняткової ситуації в регістрі зв'язку знаходиться адреса відкинутої команди плюс 4. Щоб відновити виконання програми з потрібного місця, треба зменшити значення, що зберігається в регістрі зв'язку, на 4. В даному випадку для зменшення вмісту регістра зв'язку і збереження результату в PC ми використовуємо спеціальну команду віднімання, яка також відновлює і режим роботи ЦПП. Команда повернення з режимів FIQ, IRQ і Abort виглядає наступним чином:

```
SUBS   R15,   R14,   # 4              ; R15 <– R14 – #4.
```

У випадку, якщо відбулася помилка звернення до пам'яті, виняткова ситуація виникне через одну команду після тієї, виконання якої стало її причиною. В ідеалі, в цьому випадку треба перейти до підпрограми обробки переривання Data Abort, з'ясувати і усунути причину труднощів і знову спробувати виконати команду, що викликала виняткову ситуацію. Відповідно, треба «відмотати» PC назад на дві команди – відкинуту і ту, яка викликала виникнення виняткової ситуації. Іншими словами, потрібно відняти від регістра зв'язку число вісім і зберегти результат в PC. Таким чином, команда повернення з переривання Data Abort має вигляд:

`

SUBS R15, R14, # 8 ; R15 <– R14 – #8.

При виконанні команди повернення модифікований вміст регістра зв'язку завантажується в лічильник команд, ЦПП перемикається назад у режим User, а вміст регістра SPSR переписується назад в CPSR. У разі виникнення виняткових ситуацій FIQ або IRQ додатково дозволяються відповідні переривання. В результаті всіх цих дій процесор виходить з привілейованого режиму і повертається до виконання користувальницької програми (рисунок 3.7).

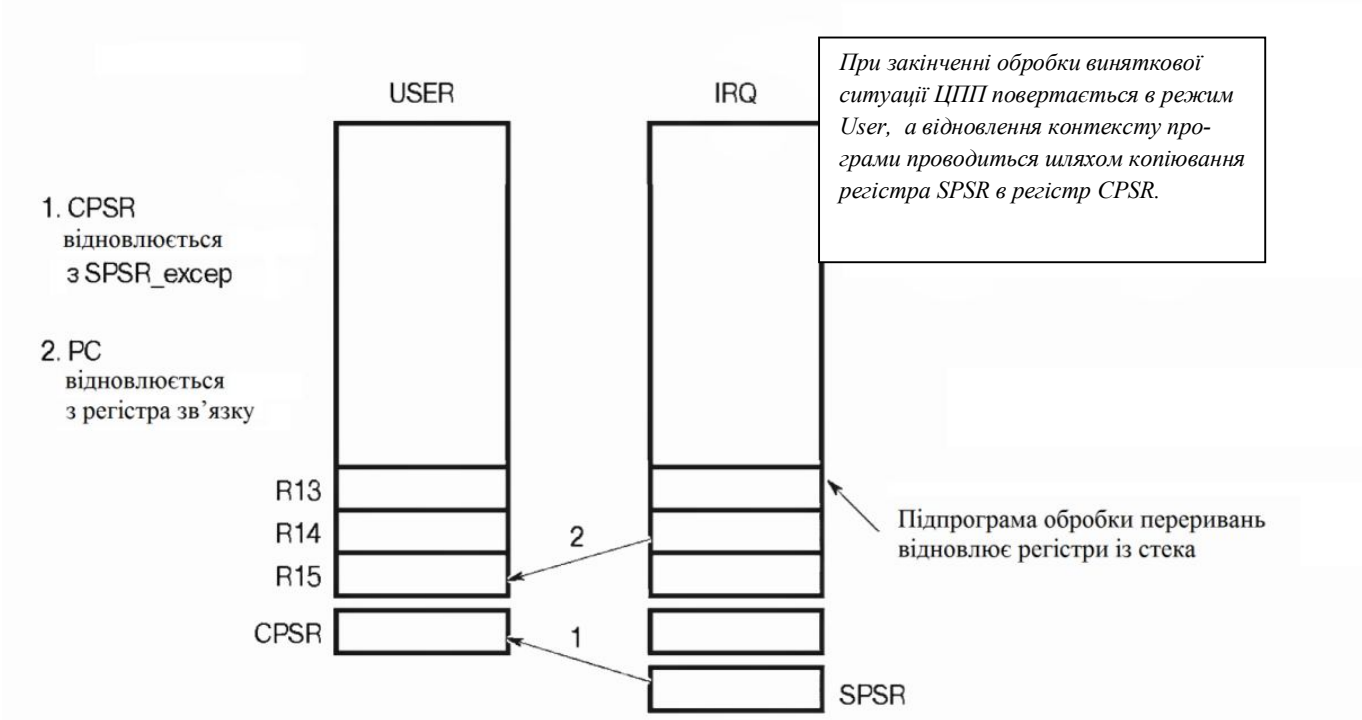


Рисунок 3.7 – Завершення обробки виняткової ситуації

3.6 Набір команд ARM7

Тепер, коли отримано загальне уявлення про ядро ARM, його моделі програмування і режими роботи, настав час познайомитися з його набором, або, якщо точніше, наборами команд. Оскільки сьогодні для програмування МК–в широко використовується мова Сі, немає необхідності бути експертом в області програмування на асемблері ARM7.

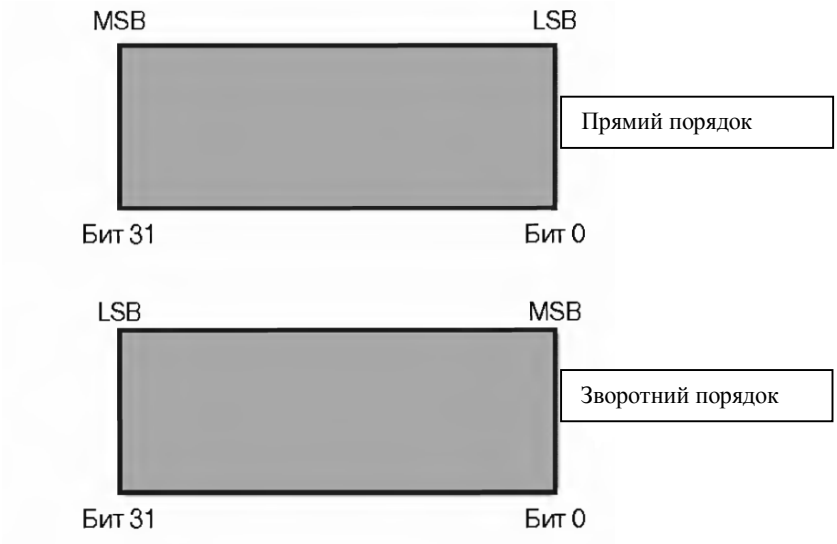
`

Однак, щоб розробляти дійсно ефективні програми, дуже важливо розуміти машинний код, який переховується за рядками програми на мові високого рівня. Перш ніж приступити до вивчення команд ARM7, необхідно зазначити, що насправді ЦПП ARM7 підтримує два набори команд: набір команд ARM з 32–бітними командами і набір команд THUMB з 16–бітними командами. Далі в посібнику слово ARM означатиме 32–бітний набір команд, а слово ARM7 – власне ЦПП.

Ядро ARM7 було розроблено таким чином, щоб його можна було використовувати як у якості процесора із зворотним порядком байтів (big–endian processor), так і в якості процесора з прямим порядком байтів (little–endian processor). У першому випадку старший біт (Most Significant Bit – MSB) 32–бітного слова розташовується на початку слова, а в другому випадку – в кінці (рисунок 3.8). В сімействі LPC2300 використовується тільки прямий порядок байтів (тобто MSB є бітом з самою старшою адресою), що значно полегшує роботу з процесором. Проте використовуваний вами компілятор для ARM7 повинен вміти компілювати код в обох форматах. У зв'язку з цим необхідно упевнитися, що формат слів задано правильно, інакше отриманий код буде «вивернуто навиворіт».

Одна з найбільш цікавих особливостей набору команд ARM полягає в тому, що кожна команда підтримує умовне виконання. У традиційних мікроконтролерах єдиними умовними командами є команди умовних переходів, і, можливо, ряд інших, таких як команди перевірки або зміни стану окремих бітів.

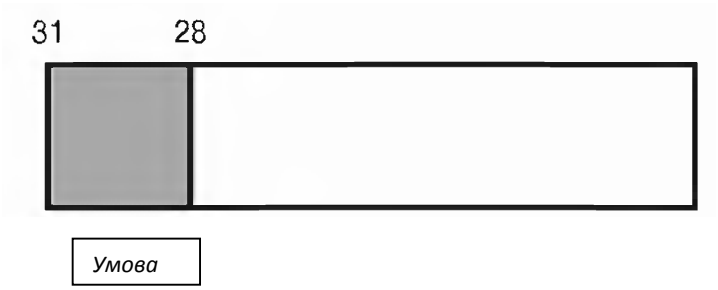
`



ЦПП ARM7 підтримує як прямий, так і зворотний порядок байтів. Тим не менш, в мікроконтролерах NXP використовується тільки прямий порядок байтів (little endian).

Рисунок 3.8 – Прямий і зворотний порядок байтів

В наборі команд ARM старші 4 біти коду команди завжди порівнюються з прапорцями умов в регістрі CPSR (рисунок 3.9). Якщо їх значення не співпадають, команда не виконується і проходить через конвеєр як команда NOP (немає операції).



Кожна команда ARM (32-бітна) є умовно виконуваної. Між 4 старшими бітами коду команди і прапорцями умов регістра CPSR проводиться операція «Логічне І». Якщо значення не співпадають, виконується команда NOP

Рисунок 3.9 – Положення бітів порівняння в команді ARM

Таким чином, можна виконати будь–яку команду обробки даних, що змінює прапорці умов в регістрі CPSR. Потім, залежно від результату, наступна команда може бути виконана, а може і ні. До базових мнемонічних позначень команд асемблера, таким як MOV або ADD, можна додати будь–який з шістнадцяти префіксів, що визначають тестований стан прапорців умов (таблиця 3.3).

`

Таблиця 3.3 – Префікси команд

КОД	Префікс	Прапорці	Значення
0000	EQ	Z встановлений	Дорівнює
0001	NE	Z скинутий	Не дорівнює
0010	CS	C встановлений	Більше або дорівнює (беззнакове)
0011	CC	C скинутий	Менше (беззнакове)
0100	MI	N встановлений	Від'ємний результат
0101	PL	N скинутий	Додатний результат
0110	VS	V встановлений	Переповнення
0111	VC	V скинутий	Немає переповнення
1000	HI	C встановлений, Z скинутий	Більше (беззнакове)
1001	LS	C скинутий, Z встановлений	Менше або дорівнює (беззнакове)
1010	GE	N дорівнює V	Більше або дорівнює (беззнакове)
1011	LT	N не дорівнює V	Менше (знакове)
1100	GT	Z скинутий I (N = V)	Більше (знакове)
1101	LE	Z встановлений АБО (N < > V)	Менше або дорівнює (знакове)
1110	AL	(ігнорується)	Безумовне виконання

Наприклад, команда:

```
EQMOV   R1, # 0x00800000
```

виконує завантаження числа 0x00800000 в регістр R1 тільки в тому випадку, якщо результат виконання останньої команди обробки даних був «дорівнює» і відповідно встановлено прапорець Z регістра CPSR. Метою такого умовного виконання команд є забезпечення безперервності потоку команд через конвеєр, тому що при кожному виконанні команд переходу конвеєр скидається і на його повторне заповнення потрібен час, що різко знижує загальну продуктивність. На практиці існує певний поріг, при якому примусове «проштовхування» команд NOP через конвеєр виявляється ефективніше ви-

`

конання традиційних команд умовного переходу і пов'язаного з цим повторним заповненням буферу.

Зазначений поріг дорівнює трьом командам, тому короткий перехід, такий як:

```
        if ( x < 100 )
{               x++;            }
```

при використанні умовно–виконуваних команд ARM буде реалізовано більш ефективно.

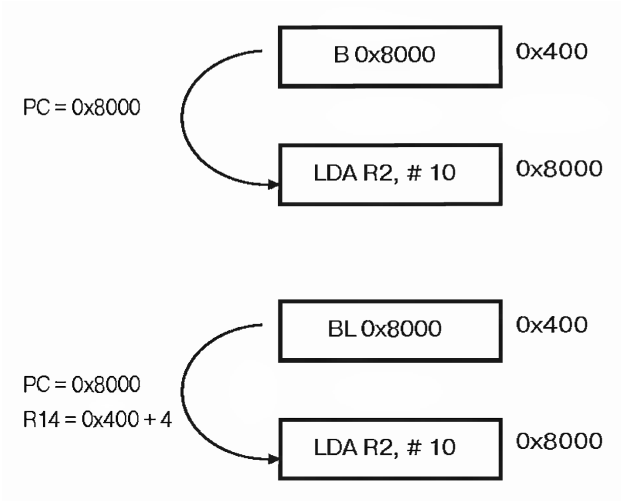
Всі команди ARM можна розбити на 6 основних груп: команди розгалуження, команди обробки даних, команди передачі даних, команди передачі блоків даних, команди множення і команда програмного переривання.

3.6.1 Команди розгалуження

Базова команда переходу (B), як випливає з її назви, дозволяє виконувати перехід в діапазоні до 32 Мбайт як вперед, так і назад. Модифікована версія команди, команда переходу зі збереженням адреси (BL), виконує ту ж операцію, проте при цьому зберігає в регістрі зв'язку поточне значення PC, яке збільшене на чотири (рисунок 3.10).

Таким чином, команда переходу зі збереженням адреси використовується як команди виклику підпрограм, що зберігає адресу повернення в регістрі зв'язку. Для повернення з підпрограм можна використовувати команду звичайного переходу, що виконує перехід за адресою, яка знаходиться в регістрі зв'язку.

`

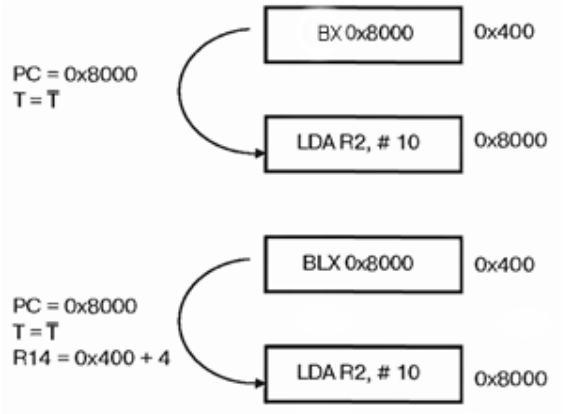


Команда переходу має кілька різновидів. Звичайна команда переходу (B) здійснює перехід за заданою адресою. Команда переходу зі збереженням адресу (BL) здійснює перехід за заданою адресою, зберігаючи при цьому адресу повернення в регістрі R14.

Рисунок 3.10 – Команди переходу B і BL

Використовуючи префікси умов, виконуються умовні переходи і умовні виклики підпрограм. Існує ще два різновиди команди переходу: «перехід зі зміною стану» (BX) і «перехід зі зміною стану та збереженням адреси» (BLX). Ці команди виконують ті ж операції, що і попередні команди, але при цьому ще й виконують переключення з набору команд ARM на THUMB і назад (рисунок 3.11).

Це єдиний спосіб, який трнба застосовувати для зміни використовуваного набору команд, так як безпосередні маніпуляції з прапорцем T регістра CPSR можуть привести до непередбачуваних результатів.



Команди переходу зі зміною стану (BX) і переходу зі зміною стану та збереженням адреси (BLX) виконують такі ж операції, як і звичайні команди переходу та переходу зі збереженням адреси (B і BL), додатково здійснюючи перемикання між наборами команд ARM і THUMB.

Рисунок 3.11 – Команди переходу BX і BLX

`

3.6.2 Команди обробки даних

Команди обробки даних наведено в таблиці 3.4, а узагальнений формат всіх команд обробки даних наведено на рисунку 3.12.

Таблиця 3.4 – Команди обробки даних

Мнемокод	Опис команди
AND	Логічне побітове « І »
EOR	Логічне побітове « виключне АБО »
SUB	Віднімання
RSB	Зворотне віднімання
ADD	Додавання
ADC	Додавання з урахуванням перенесення
SBC	Віднімання з запозиченням
RSC	Зворотне віднімання з запозиченням

Мнемокод	Опис команди
TST	Перевірка бітів
TEQ	Побітове порівняння
CMP	Порівняння
CMN	Порівняння з запереченням
ORR	Логічне побітове « АБО »
MOV	Пересилання
BIC	Скидання бітів (маскування)
MVN	Пересилання з інверсією

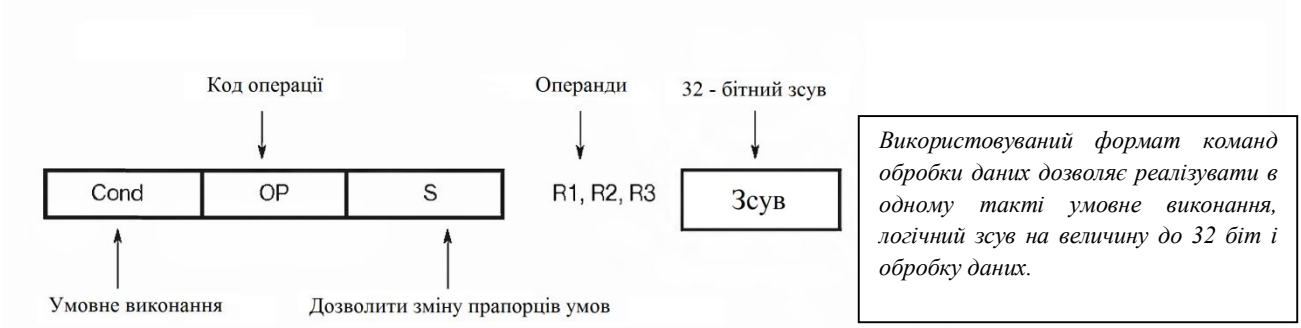


Рисунок 3.12 – Формат команд обробки даних

У кожній команді є регістр результату і два операнди. Перший операнд обов'язково повинен бути регістром,тоді як другий може бути як регістром, так і константою.

`

Крім цього, в ЦПП ARM7 є багаторегістровий пристрій циклічного зсуву (barrel shifter), що дозволяє при виконанні команди зсувати значення 2–го операнда на величину до 32 біт. Біт S використовується для керування прапорцями умов. Якщо цей біт встановлено, прапорці умов змінюються відповідно до результату виконання команди.

Якщо цей біт скинуто, стан прапорців умов не змінюється. Однак якщо при встановленому біті S в якості регістра результату вказаний лічильник команд (R15), проводиться копіювання вмісту регістра SPSR поточного режиму в регістр CPSR. Ця можливість використовується для відновлення PC і перемикання в початковий режим в кінці обробки виняткових ситуацій. Не намагайтеся виконати таку команду в режимі User, оскільки в цьому режимі відсутній регістр SPSR і відповідно результат виконання цієї команди неможливо передбачити.

Ці особливості надають нам багатий набір команд обробки даних (таблиця 3.4), який, з одного боку, дозволяє створювати дуже ефективні програми, а з іншого – є джерелом «нічних кошмарів» для розробників компіляторів.

Наприклад, в результаті компіляції виразу

```
if (Z == 1)
R1 = R2 + (R3 x 4);
```

може бути згенерована наступна команда:

```
EQADDS R1, R2, R3, LSL # 2.
```

3.6.3 Команди передачі даних

3.6.3.1 Команди завантаження/збереження

Наступну групу складають команди передачі даних (таблиця 3.5). ЦПП ARM7 підтримує команди завантаження / збереження, які дозволяють пере-

`

силати знакові і беззнакові числа різного розміру (слово, напівслово, байт) в / із заданого регістра.

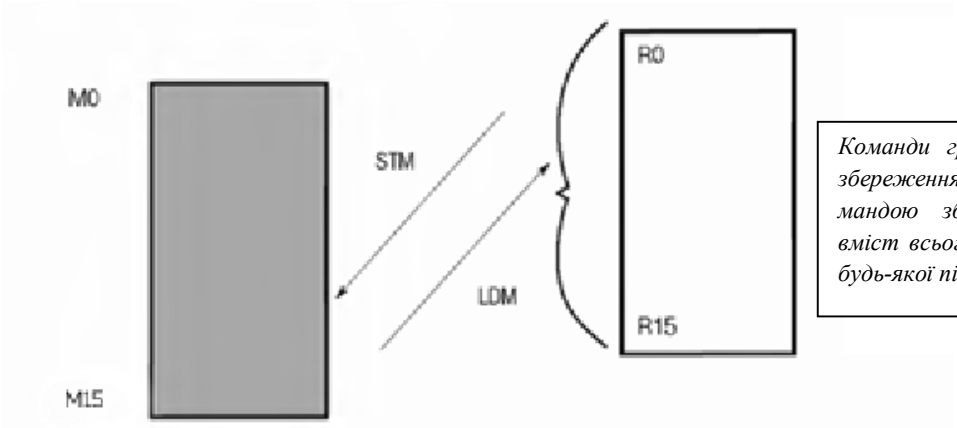
Таблиця 3.5 – Команди передачі даних

Мнемокод	Опис команди	Мнемокод	Опис команди
LDR	Завантажити слово	STR	Зберегти слово
LDRH	Завантажити напівслово	STRH	Зберегти напівслово
LDRSH	Завантажити напівслово зі знаком	STRSH	Зберегти напівслово зі знаком
LDRB	Завантажити байт	STRB	Зберегти байт
LDRSB	Завантажити байт зі знаком	STRSB	Зберегти байт зі знаком

Оскільки набір регістрів повністю незалежний, можна завантажувати 32–бітове значення безпосередньо в PC, здійснюючи, таким чином, перехід в межах всього адресного простору процесора. Якщо кінцева адреса лежить поза діапазоном команди переходу, можна просто завантажити збережену константу в лічильник команд.

3.6.3.2 Групове копіювання регістрів

Крім команд завантаження / збереження вмісту окремих регістрів, в наборі команд ARM є команди для завантаження (LDM) і збереження (STM) груп регістрів (рисунок 3.13). Таким чином, за допомогою однієї команди можна скопіювати в пам'ять весь блок регістрів або його частину, а за допомогою іншої – відновити його вміст.



Команди групового завантаження / збереження дозволяють однією командою зберігати / відновлювати вміст всього регістрового файлу або будь-якої підмножини регістрів.

Рисунок 3.13 – Команди LDM і STM

`

3.6.4 Команда обміну

У наборі команд ARM є також команда обміну (SWP), завдяки якій забезпечується підтримка семафорів реального часу. Ця неперервна команда здійснює одночасний обмін вмісту двох регістрів (рисунок 3.14). Завдяки такому рішенню запобігається переривання процесу обміну критичними даними при виникненні виняткової ситуації. У мові Сі ця команда безпосередньо недоступна та підтримується вбудованими функціями бібліотек компіляторів.

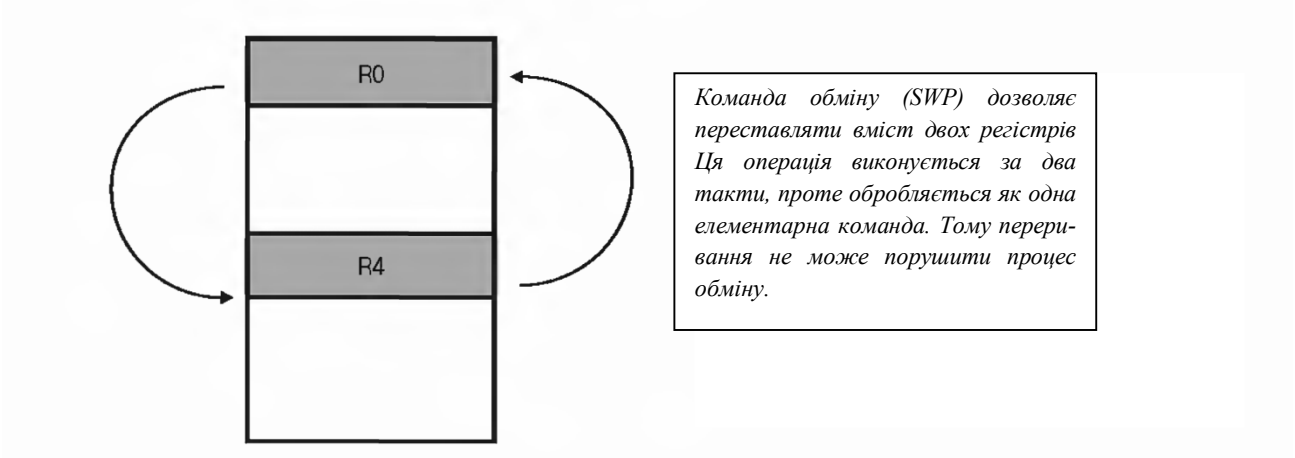


Рисунок 3.14 – Команди обміну

3.6.5 Команди зміни регістрів стану

Як вже було зазначено вище регістри CPSR і SPSR є регістрами ЦПП, однак не входять до складу основного банку регістрів. Безпосередньо звертатися до цих регістрів можуть тільки дві команди ARM – MSR і MRS. Зазначені команди забезпечують пересилання вмісту регістра CPSR або SPSR в / із заданого регістра (рисунок 3.15). Наприклад, щоб заборонити переривання IRQ, необхідно скопіювати вміст регістра CPSR в робочий регістр, встановити прапорець I (шляхом виконання операції «I» між цим регістром і числом 0x00000080) і завантажити отримане значення назад в регістр CPSR. Команди MSR і MRS доступні у всіх режимах процесора, за винятком режиму User.

`

Таким чином, тільки перебуваючи в привілейованому режимі, можна змінювати робочий режим процесора і дозволяти / забороняти переривання,.

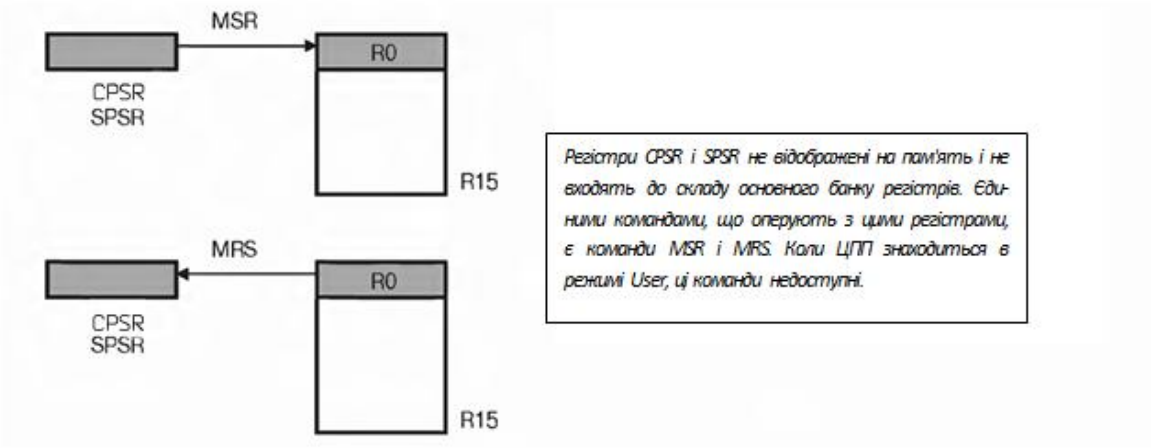


Рисунок 3.15 – Зміна регістрів стану

Після входу в режим User ви можете з нього вийти тільки при виникненні виняткової ситуації, скиданні, генерації переривань FIQ і IRQ або ж в результаті виконання команди SWI.

3.6.6 Команда програмного переривання

Команда програмного переривання SWI генерує виняткову ситуацію, в результаті чого процесор перемикається в режим Supervisor, а в лічильник команд заноситься значення 0x00000008. Як і всі інші команди ARM, команда SWI містить в чотирьох старших бітах прапорці умовного виконання, за якими розташовується код операції (рисунок 3.16). Інша частина слова команди залишається вільною. Однак у цих невикористовуваних бітах може зберігатися число. Дані біти можна перевіряти на початку підпрограми обробки переривання, щоб визначити, яку саме частину підпрограми слід виконувати. Таким чином, за допомогою команди SWI можна перемикатися в захищений режим для виконання привілейованих ділянок програми або обробки системних викликів.

`

Команда програмного переривання (SWI) перемикає ЦПП в режим Supervisor і переходить за адресою вектора SWI. Біти О...23 не задіяні, проте в них можна передавати значення, визначені користувачем.

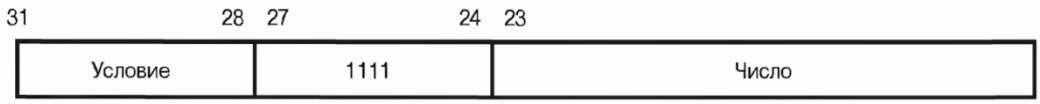


Рисунок 3.16 – Команда SWI

Після компіляції команди

SWI # 3

в невикористовуваних бітах слова команди буде записано число 3. У підпрограмі обробки переривання SWI ми можемо перевірити значення слова команди наступним чином (текст написаний на псевдокоді):

```
switch ( * (R14 – 4) & OxOOFFFFFF)          / / Повернутися на 4 байти назад
 {                                          / / Замаскувати старші 8 бітів і
                                            / / Виконати перехід
                                            / / відповідно до результату
case (SWI–1)
            ...
```

Залежно від використовуваного компілятора доводиться реалізовувати таку перевірку самостійно або ж компілятор автоматично вставить необхідні команди в код програми.

3.6.7 Команди множення чисел

Поряд з багаторегістровим пристроєм циклічного зсуву в ядрі ARM7 є вбудований модуль помножувача / суматора (MAC). Модуль MAC підтримує множення чисел типу integer і long integer. Команди множення чисел типу integer виконують множення двох 32–бітних регістрів і поміщають результат в третій 32–бітний регістр. Команда множення з накопиченням виконує мно-

`

ження і додає результат до проміжної суми. Команди множення чисел типу long integer перемножують вміст двох 32–бітних регістрів і поміщають 64–бітний результат у два регістри. Аналогічно, є команда довгого множення з накопиченням (таблиця 3.6).

Таблиця 3.6 – Команди множення

Мнемокод	**Опис команди**	**Роздільна здатність**
MUL	Множення	32–бітний результат
MULA	Множення з накопиченням	32–бітний результат
UMULL	Беззнакове множення	32–бітний результат
UMLAL	Беззнакове множення з накопиченням	32–бітний результат
SMULL	Знакове множення	32–бітний результат
SMLAL	Знакове множення з накопиченням	32–бітний результат

3.7 Набір команд Thumb

Незважаючи на те, що ARM7 є 32–бітовим процесором, він підтримує ще один набір команд (16–бітний), який називається THUMB (рисунок 3.17). Насправді, цей набір команд є стислою формою набору команд ARM. За рахунок цього команди, збережені в 16–бітному форматі, розпаковуються в команди ARM, а потім виконуються. Хоча команди THUMB забезпечують меншу продуктивність в порівнянні з командами ARM, завдяки їм досягається більш висока щільність коду. Таким чином, для створення компактних програм, що розміщуються в невеликих однокристальних мікроконтролерах, код програм необхідно компілювати у вигляді сукупності функцій THUMB і ARM.

`

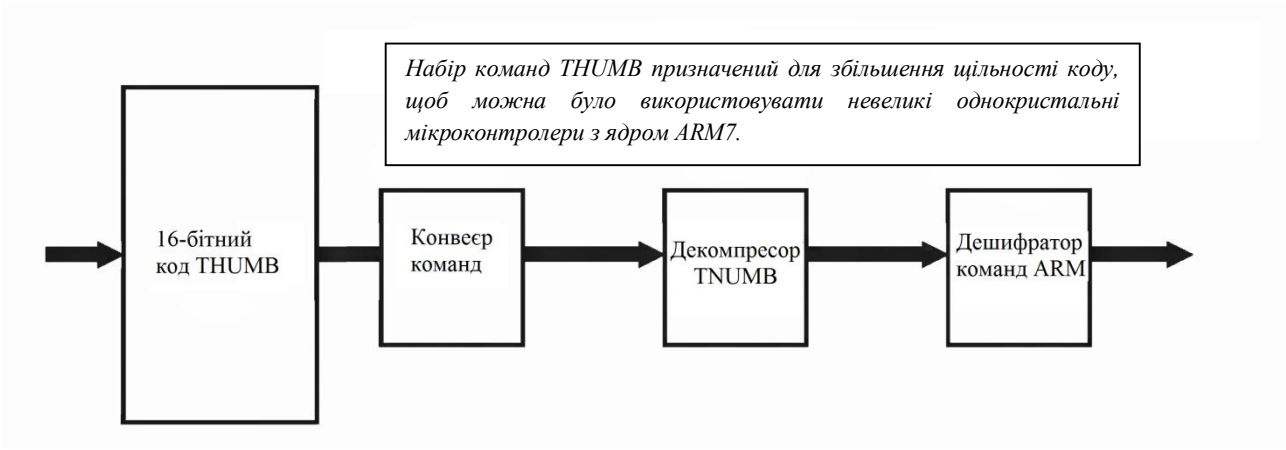


Рисунок 3.17 – Набір команд THUMB

Цей процес називається interworking і елементарно підтримується всіма компіляторами. При компіляції з використанням набору команд THUMB ви отримаєте 30–відсоткову економію пам'яті програм, у той час як цей же код, скомпільований з використанням команд ARM, буде виконуватися на 40% швидше.

Набір команд THUMB набагато більше схожий на набори команд традиційних мікроконтролерів. На відміну від команд ARM, команди THUMB не підтримують умовного виконання (за винятком команд умовних переходів). Команди обробки даних мають двоадресний формат, причому як регістр результату використовується один з регістрів–джерел:

Команда ARM	Команда THUMB	Дія
ADD R0, R0, R1	ADD R0, R1	R0 = R0 + R1

Команди THUMB не мають повного доступу до всіх регістрів регістрового файлу (рисунок 3.18). З регістрами R0 ... R7 (так званими молодшими регістрами) можуть працювати всі команди обробки даних. А до регістрів R8 .... R12 (старші регістри) можуть звертатися тільки 3 з них:

MOV, ADD, CMP.

`

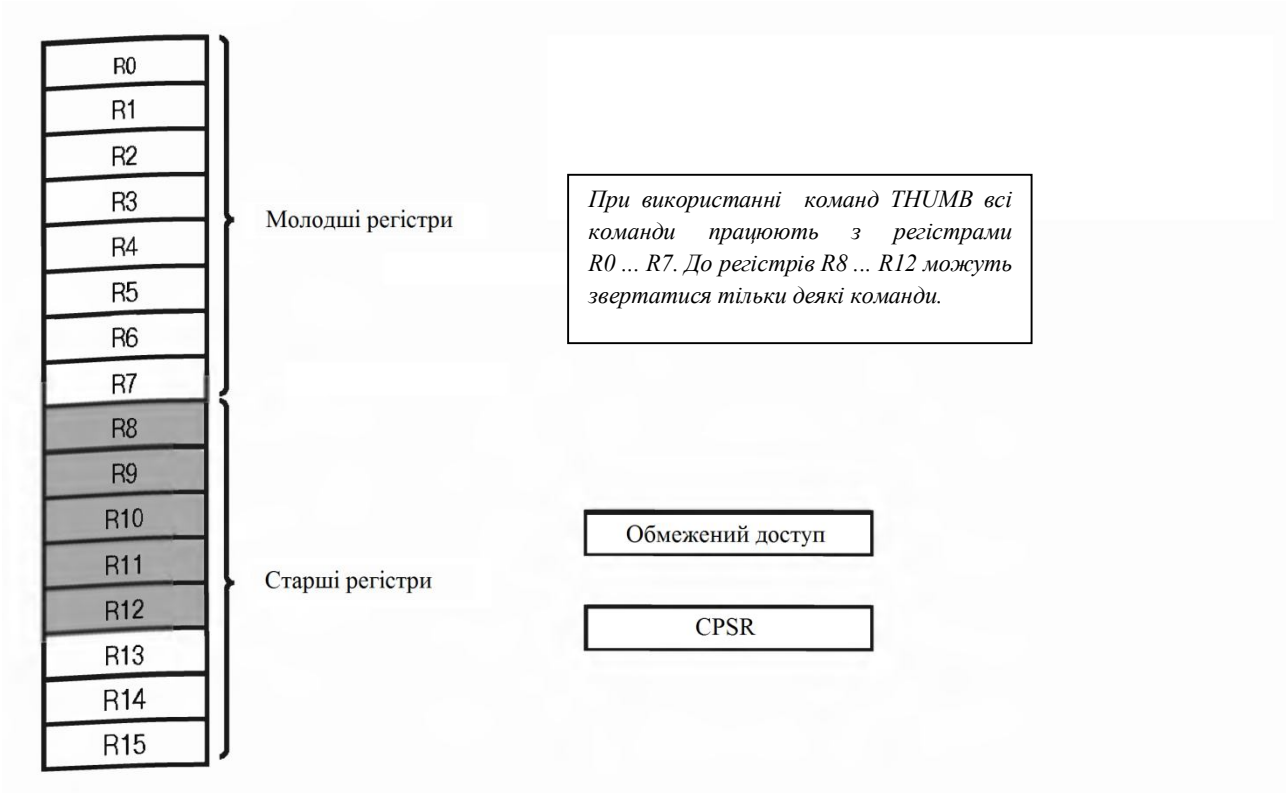


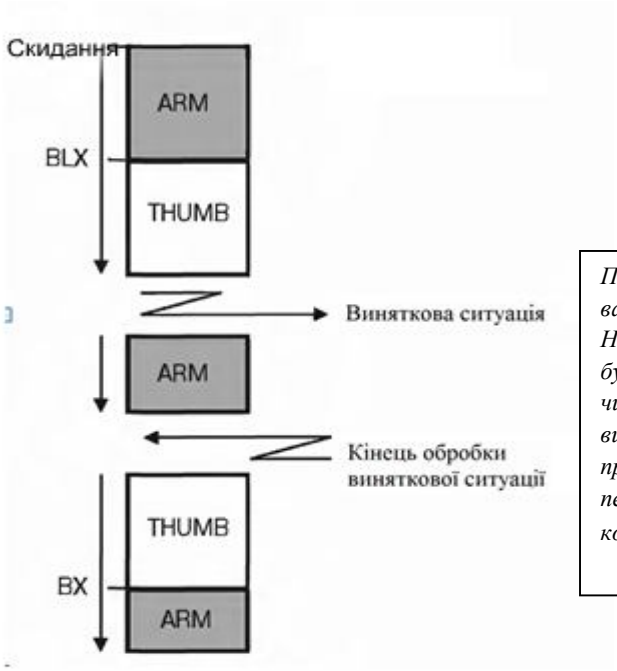
Рисунок 3.18 – Доступ до регістрового файлу

У наборі команд THUMB відсутні команди MSR і MRS, тому змінювати регістри CPSR і SPSR можна тільки за допомогою непрямої адресації. Якщо треба змінити будь–які користувацькі біти в регістрі CPSR, необхідно переключитися в режим ARM. Змінювати режими можна за допомогою команд BX і BLX (рисунок 3.19). Крім того, автоматичне перемикання в режим ARM відбувається при скиданні, або при переході до обробки виняткової ситуації.

У складі набору команд THUMB є більш звичні команди роботи зі стеком PUSH і POP (рисунок 3.20). За допомогою цих команд реалізується спадаючий стек, який жорстко прив'язаний до регістра R13.

І, нарешті, в наборі команд THUMB є команда SWI, що працює так само, як і аналогічна команда набору ARM, але містить тільки 8 невикори–стовуваних бітів, що обмежує максимальну кількість викликів SWI до 255.

`



Після скидання ARM7 буде виконувати команди ARM (32-бітні). Набір команд можна змінити в будь-який момент, використовуючи команду BX, BLX. При виникненні виняткової ситуації процесор автоматично перемикається на 32-бітний набір команд ARM.

Рисунок 3.19 – Перемикання в режим ARM

У наборі команд THUMB є спеціальні команди PUSH і POP, які реалізують спадаючий стек, використовуючи в якості покажчика стека регістр R13.

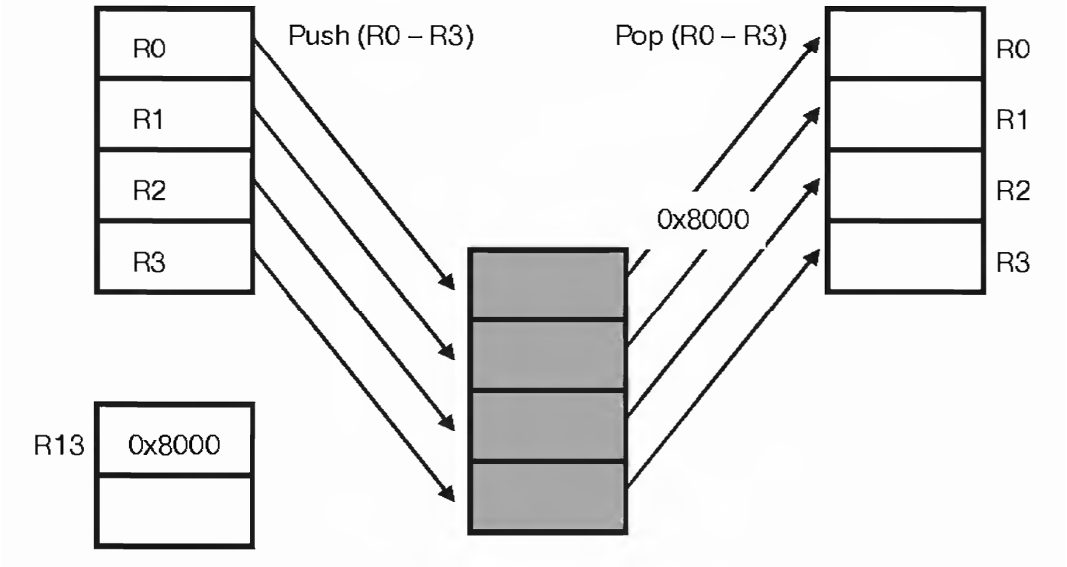


Рисунок 3.20 – Команди PUSH і POP

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1) Охарактеризуйте конвеєр команд ЦПП ARM7.
2) Яку має архітектуру процесор ARM7?
3) Перелічіть регістри ARM7 та опишіть структуру основного регістрового файлу.
4) Опишіть призначення та формат регістра поточного стану програми.
5) Які режими роботи підтримує процесор ARM7?
6) Опишіть відмінність режимів обробки виняткових ситуацій від основних режимів роботи процесора.
7) Чим відрізняються процесори із зворотним порядком байтів (big–endian processor) від процесорів з прямим порядком байтів (little–endian processor)?
8) Як команди процесорів ядра ARM7 підтримують умовне виконання?
9) Дайте пояснення префіксам команд (їх прапорці і значення):
   EQ; NE; CS; CC; VC; VS; MI; PL; HI; LS; GE; LT; GT; LE; AL.
10) Опишіть особливості виконання наступних груп команд: переходів; обробки даних; копіювання регістрів; копіювання груп регістрів; обміну; зміни регістрів стану; програмного переривання та множення..

`

4 СПОСОБИ АДРЕСАЦІЇ ОПЕРАНДІВ ТА ФОРМАТИ КОМАНД

В командах ARM використовуються наступні види адресацій:

– неявна;

– регістрова;

– непряма;

– безпосередня;

– преіндексна з регістровим зміщенням без оновлення бази;

– преіндексна з масштабованим регістровим зміщенням без оновлення бази;

– преіндексна з безпосереднім зміщенням без оновлення бази;

– преіндексна з регістровим зміщенням та оновленням бази;

– преіндексна з масштабованим регістровим зміщенням та оновленням бази;

– преіндексна з безпосереднім зміщенням та оновленням бази;

– постіндексна з регістровим зміщенням;

– постіндексна з масштабованим регістровим зміщенням;

– постіндексна з безпосереднім зміщенням.

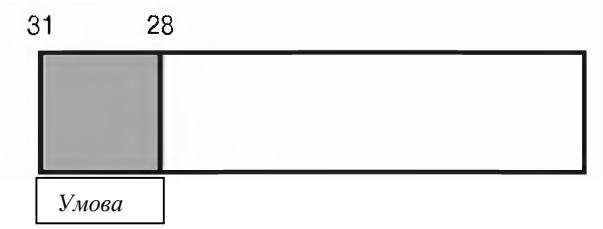
Всі команди ARM можна розбити на 6 основних груп: команди обробки даних, команди розгалуження, команди передачі даних, команди передачі блоків даних, команди множення, команда програмного переривання і команди роботи з співпроцесами.

4.1 Префікси команд

Одна з найбільш цікавих особливостей набору команд ARM полягає в тому, що кожна команда підтримує умовне виконання. У традиційних мікроконтролерах єдиними умовними командами є команди умовних переходів, і, можливо, ряд інших, таких як команди перевірки або зміни стану окремих бітів.

`

А в наборі команд ARM старші 4 біти коду команди завжди порівнюються з прапорцями умов в регістрі CPSR (рисунок 4.1).



Кожна команда ARM (32-бітна) є умовно виконуваної. Між 4 старшими бітами коду команди і прапорцями умов регістра CPSR проводиться операція «Логічне І». Якщо значення не співпадають, виконується команда NOP.

Рисунок 4.1 – Положення бітів порівняння в команді ARM

Якщо їх значення не співпадають, команда не виконується і проходить через конвеєр як команда NOP (немає операції).

Таким чином, можна виконати будь–яку команду обробки даних, що змінює прапорці умов в регістрі CPSR. Потім, залежно від результату, наступна команда може бути виконана, а може і ні. До базових мнемонічних позначень команд асемблера, таким як M0V або ADD, можна додати будь–який з шістнадцяти префіксів, що визначають тестовий стан прапорців умов (таблиця 4.1).

Таблиця 4.1 – Префікси команд

КОД	**Префікс**	**Прапорці**	**Значення**
0000	EQ	Z встановлений	Дорівнює
0001	NE	Z скинутий	Не дорівнює
0010	CS	C встановлений	Більше або дорівнює (беззнакове)
0011	CC	C скинутий	Менше (беззнакове)
0100	MI	N встановлений	Від’ємний результат
0101	PL	N скинутий	Додатний результат
0110	VS	V встановлений	Переповнення

`

Продовження таблиці 4.1

КОД	Префікс	Прапорці	Значення
0111	VC	V скинутий	Немає переповнення
1000	HI	C встановлений, Z скинутий	Більше (беззнакове)
1001	LS	C скинутий, Z встановлений	Менше або дорівнює (беззнакове)
1010	GE	N дорівнює V	Більше або дорівнює (беззнакове)
1011	LT	N не дорівнює V	Менше (знакове)
1100	GT	Z скинутий I (N = V)	Більше (знакове)
1101	LE	Z встановлений АБО (N < > V)	Менше або дорівнює (знакове)
1110	AL	(ігнорується)	Безумовне виконання

Наприклад, команда:

```
EQMOV    R1,  # 0x00800000
```

виконає завантаження числа 0x00800000 в регістр R1 тільки в тому випадку, якщо результат виконання останньої команди обробки даних був «дорівнює» і відповідно встановлено прапорець Z регістра CPSR. Метою такого умовного виконання команд є забезпечення безперервності потоку команд через конвеєр, тому що при кожному виконанні команд переходу конвеєр скидається і на його повторне заповнення потрібен час, що різко знижує загальну продуктивність. На практиці існує певний поріг, при якому примусове «проштовхування» команд NOP через конвеєр виявляється ефективніше виконання традиційних команд умовного переходу і пов'язаного з цим повторним заповненням буфера.

Встановлення прапорців CPSR не є обов'язковою, і управляється бітом S інструкції.

`

4.2 Формати команд обробки операндів

На рисунку 4.2 наведено машинний код (формат) команд обробки операндів.

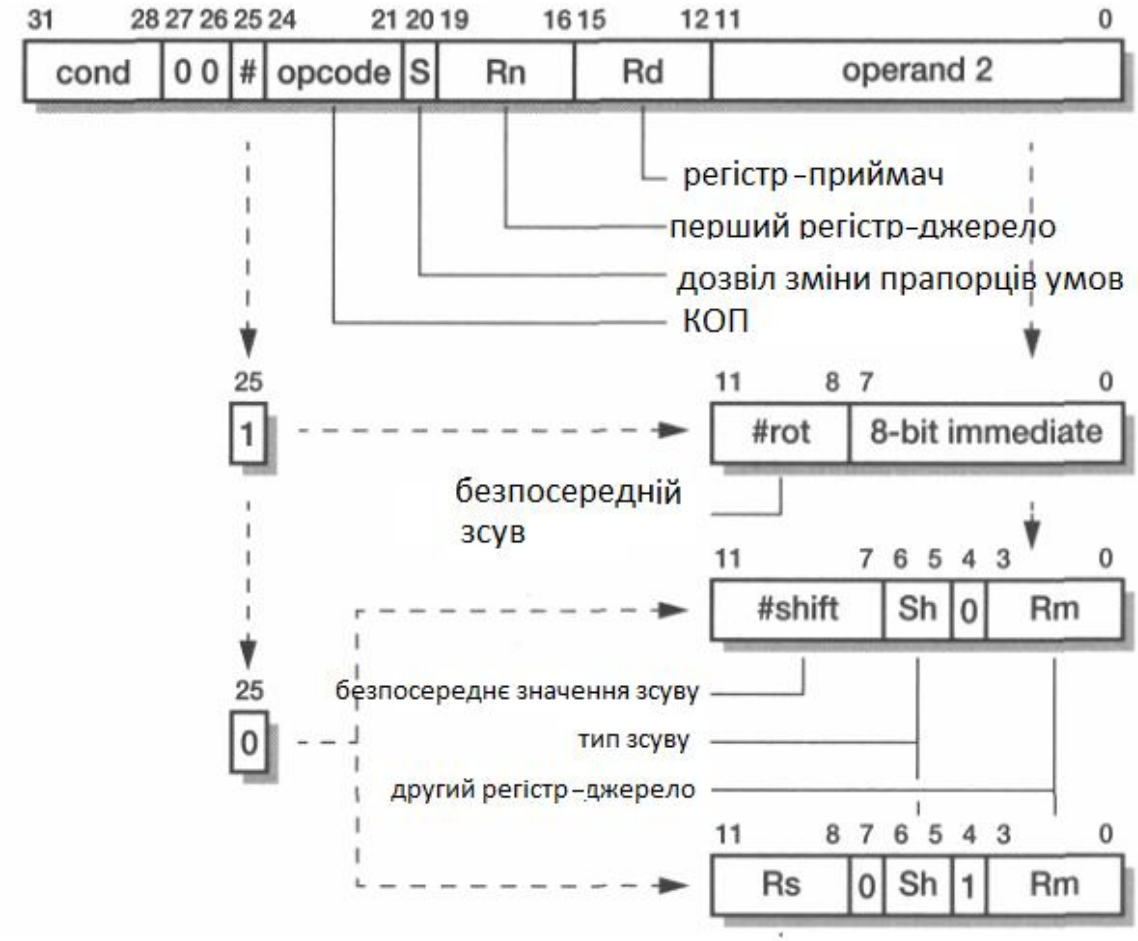


Рисунок 4.2 – Формат команд обробки операндів

Нижче наведено пояснення окремих полів формату команд, наведеного на рисунку 4.2:

– cond (біти 31…28) формує компілятор в залежності від префіксу команди (таблиця 3.3), якщо останній використовується;

– розряди 27, 26 дорівнюють нулю;

– # (25 р) залежить від того, де знаходиться операнд, який може зсуватися: якщо 25 р = 1, зсувається безпосередній 8–розрядний операнд (8–bit immediate) на величину зсуву, яка визначається полем #rot

`

(11 р…8 р); якщо 25 р = 0, зсувається вміст регістра Rm на величину зсуву, яка визначається полем #shift (11 р…7 р), або полем Rs (11 р…8 р), яке визначає один з регістрів, в яких знаходиться величина зсуву;

– opcode (24 р…21 р) являє собою код операції команди (КОП), який визначається мнемокодом команди;

– S (20 р) дозволяє/забороняє зміну прапорців умов за результатом виконання команди, наприклад:

AND R1, #0x0 ; не змінюватиме прапорці;

ANDS R1, #0x0; встановить прапорець Z в одиницю.

В інструкціях CMP, CMN, TST та TEQ цей розряд завжди = 1;

– Rn (19 р…16 р) визначає 1–й операнд–джерело (один з регістрів основного регістрового файлу);

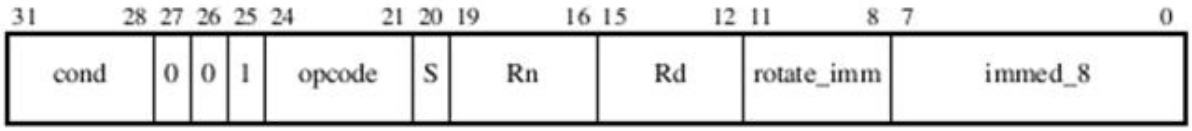
– Rd (15 р…12 р) визначає регістр–приймач (один з регістрів основного регістрового файлу);

– operand 2 (11 р…0 р) визначає 2–й операнд, вміст якого може зсуватися: #rot – число: 0…15, яке визначає величину зсуву 2–го 8–бітного безпосереднього операнда (8–bit immediate); #shift – визначає величину зсуву: (0…31), а поле Sh (6 р, 5 р) – визначає тип зсуву: Sh = 00 – арифметичний/логічний вліво; 01 логічний вправо; 10– арифметичний вправо; 11 – циклічний вправо; 4 р = 0, що визначає, що зсувається в залежності від поля Sh операнд Rm на величину #shift; Rm визначає другий операнд (один з регістрів основного регістрового файлу); якщо 4 р = 1, то поле Rs (11 р…8 р) визначає один з регістрів основного регістрового файлу, в якому записано величину зсуву, при цьому 7 р = 0, а 6 р, 5 р виконують функцію поля Sh, яка описана вище.

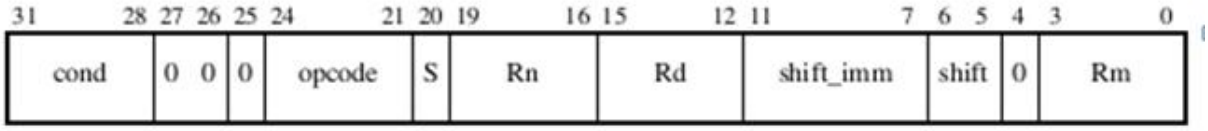
На рисунку 4.3 наведено приклади трьох форматів команд, які відповідають рисунку 4.2.

`

32-bit immediate 32-біти (безпосередній операнд)



Безпосередній зсув



Регістровий зсув

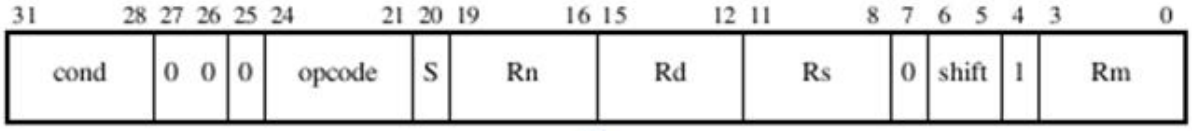
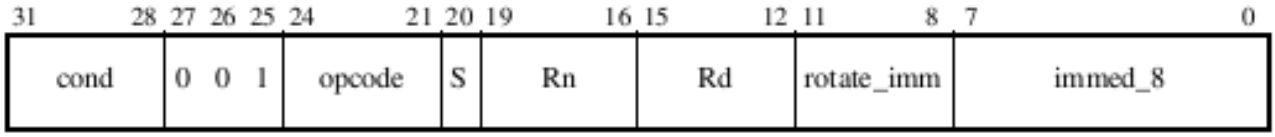


Рисунок 4.3 – Види форматів команд при обробці операндів

Нижче наведено приклади мнемокодів та форматів команд, які відповідають рисунку 4.3.

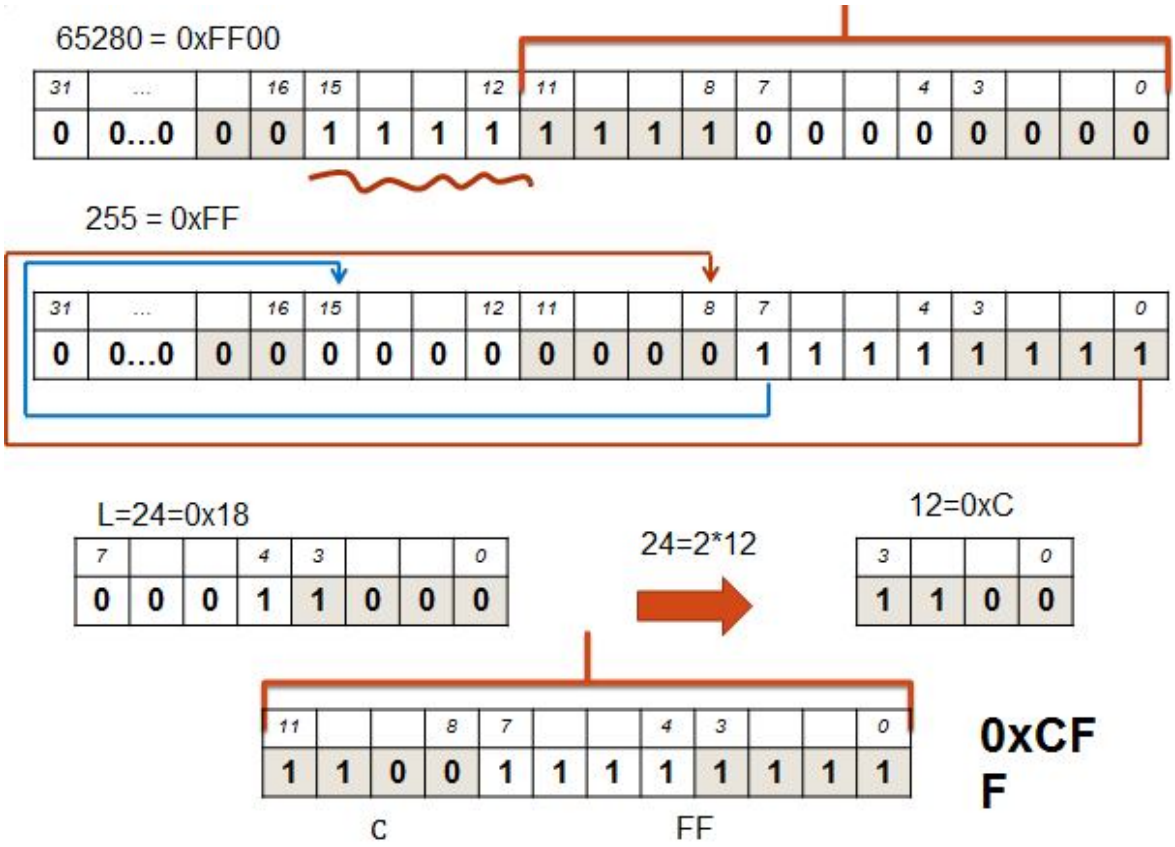
4.2.1 Безпосередній операнд

Операндом в даному випадку являється 32–бітне число. При цьому, дозволені лише ті числа. які можуть бути утворені циклічним зсувом вправо 8–бітного значення (immed – 8) на парну кількість позицій.

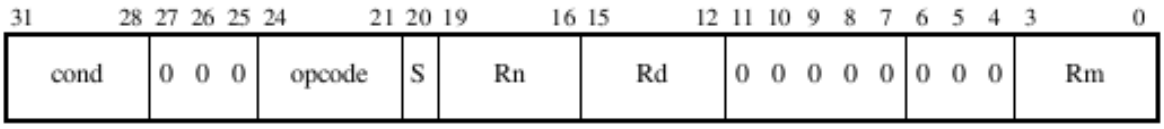


ADD R1, R2, #0xFF00 ; R1=R2+ 65280
E2821CF

1110 0010 1000 0010 0001 1100 1111 1111



4.2.2 Регістровий операнд



MOV R2, R0 ; *R2=R0*

MOV R2, R1 ; *R2=R1*

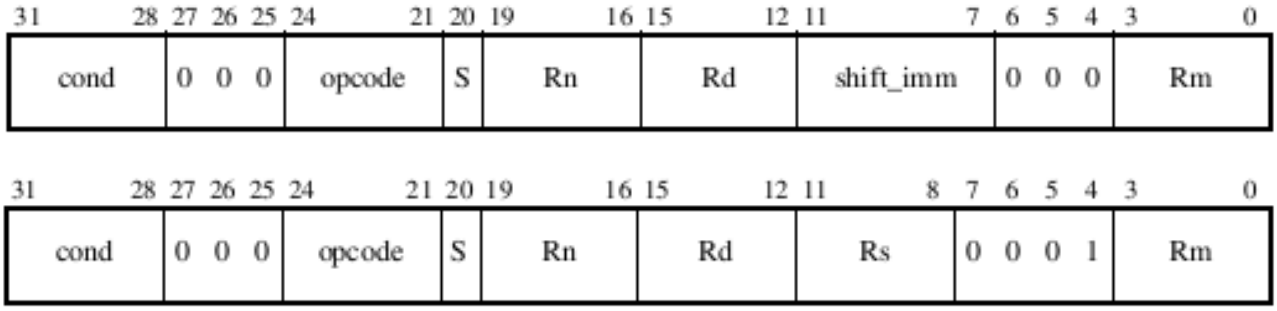
ADD R4, R3, R2 ; *R4=R3+R2*

E1A02000 = 1110 0001 1010 0000 0010 0000 0000 0000

E1A02001 = 1110 0001 1010 0000 0010 0000 0000 0001

E0834002 = 1110 0000 1000 0011 0100 0000 0000 0010

4.2.3 Логічний зсув регістра вліво



<Rm>, LSL #<shift_imm>;

<Rm>, LSL <Rs>;

<Rm> вказує регістр, чиє значення буде зсунуто;

LSL вказує логічний зсув вліво;

<shift_imm> вказує зсув. Значення у діапазоні 0..31;

<Rs> – регістр, що містить значення зсуву;

ADD R9, R4, R5, LSL #3 ; R9 = R4 + R5 x 8

ADD R9, R4, R5, LSL R1 ; R9 = R4 + R5 x R1

4.2.4 Логічний зсув регістра вправо

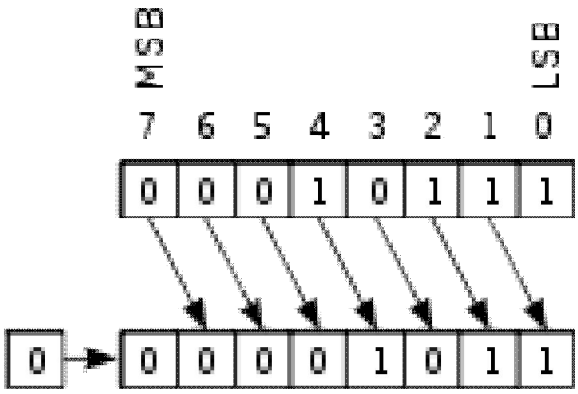
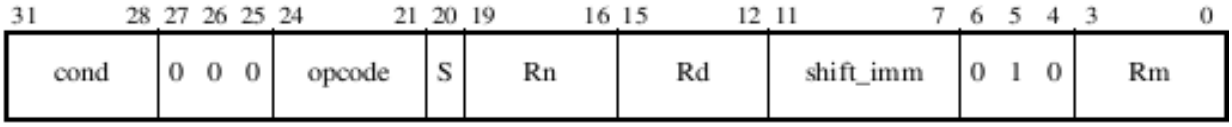
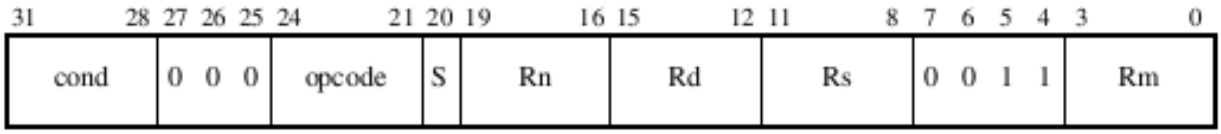


Рисунок 4.3 – Логічний зсув вправо



`



<*Rm*>, LSR #<*shift_imm*>

<*Rm*>, LSR <*Rs*>

<*Rm*> вказує регістр, чиє значення буде зсунуто

<*shift_imm* > вказує зсув (1..32, при значенні 32 : shift_imm = 0.)

<*Rs*> регістр, що містить значення зсуву

ADD R9, R3, R5, LSR #3; R9=R3+R5/8

ADD R9, R3, R5, LSR R1; R9=R3+R5/R1

4.2.5 Арифметичний зсув регістра вправо

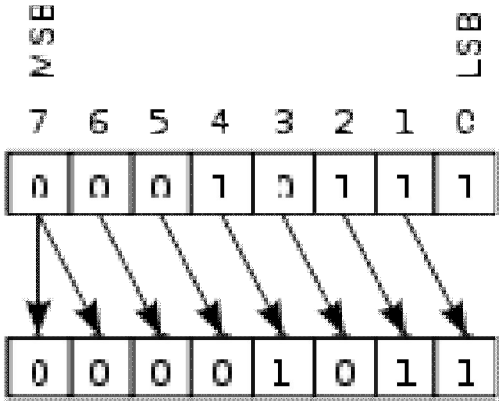
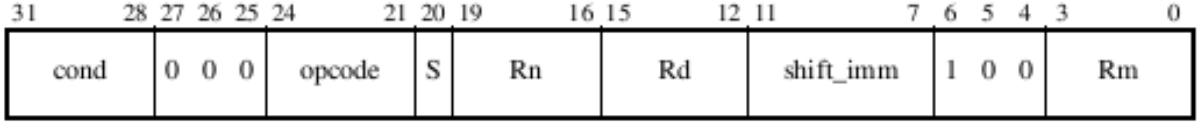


Рисунок 4.4 – Арифметичний зсув вправо



`

<Rm>, ASR #<shift_imm>

<Rm>, ASR <Rs>

<Rm> вказує регістр, чиє значення буде зсунуто

<shift_imm > вказує зсув (1..32, при значенні 32 : shift_imm = 0.)

<Rs> регістр, що містить значення зсуву

ADD R9, R2, R5, ASR #3; R9=R2+R5/8

ADD R9, R2, R5, ASR R1; R9=R2+R5/R1

4.2.6 Циклічний зсув вправо

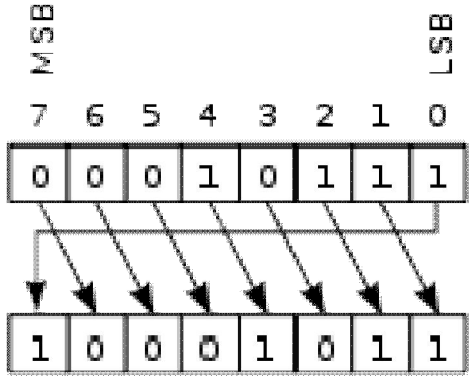
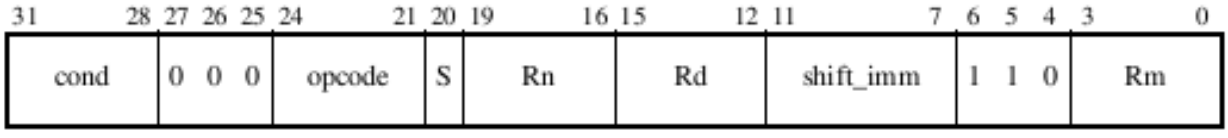
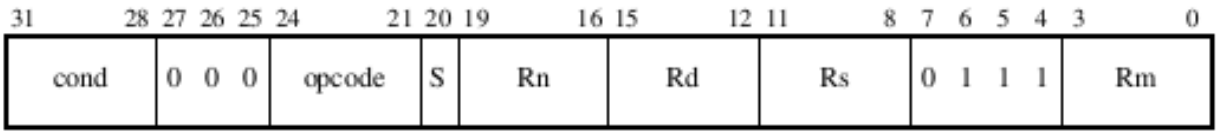


Рисунок 4.5 – Циклічний зсув вправо





<Rm>, ROR #<shift_imm>

<Rm>, ROR <Rs>

<Rm> вказує регістр, чиє значення буде зсунуто

<shift_imm > вказує зсув (1..32, при значенні 0: RRX.)

<Rs> регістр, що містить значення зсуву

MOV R2, R2, ROR #5 ; R2 = R2 ROR #5

MOV R2, R2, ROR R1 ; R2 = R2 ROR R1

4.2.7 Циклічний зсув вправо (розширений)

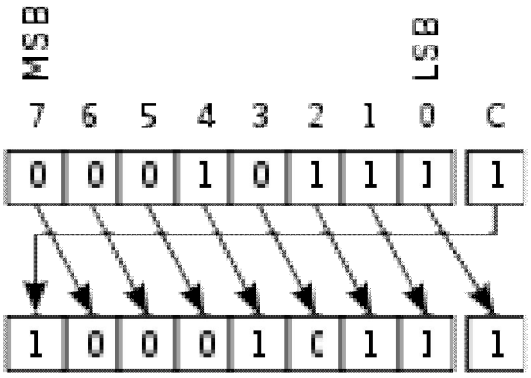
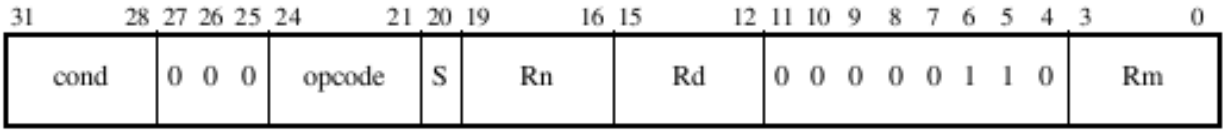


Рисунок 4.6 – Циклічний зсув вправо (розширений)



<Rm>, RRX

<Rm> вказує регістр, чиє значення буде зсунуто

MOVS R2,R2,RRX ; змінює папорці CPSR

MOV R2,R2,RRX ; не змінює прапорці CPSR

Приклади команд зі зсувом:

ADD R9, R4, R5, LSL #3 ;

E0849185 = 1110 0000 1000 0100 1001 0001 1000 0101

ADD R9, R3, R5, LSR R1;

E0839135 = 1110 0000 1000 0011 1001 0001 0011 0101

ADD R9, R2, R5, ASR #3;

E08291C5 = 1110 0000 1000 0010 1001 0001 1100 0101

MOV R2, R2, ROR R1

E1A02172 = 1110 0001 1010 0000 0010 0001 0111 0010

MOVS R2,R2, RRX

E1B02062 = 1110 0001 1011 0000 0010 0000 0110 0010

4.3 Формати команд завантаження/збереження

Команди завантаження (load) та зберігання (store) мають декілька форматів.

4.3.1 Команди завантаження/ збереження слова або байта без знаку

На рисунку 4.7 наведено формати команд завантаження та зберігання слова або байта без знаку.

Нижче наведено пояснення окремих полів формату команд, наведених на рисунку 4.7:

– cond (біти 31…28) формує компілятор в залежності від префіксу команди (таблиця 4.1), якщо останній використовується;

– 27 p = 0; 26 p = 1;

– # (25 p) залежить від того, де знаходиться зміщення (offset_12), яке може використовуватись командою: якщо 25 p = 0, то в якості зміщення використовується 12–розрядне число; якщо 25 p = 1, то зміщення знаходиться в одному з регістрів основного регістрового файлу – Rm;

`

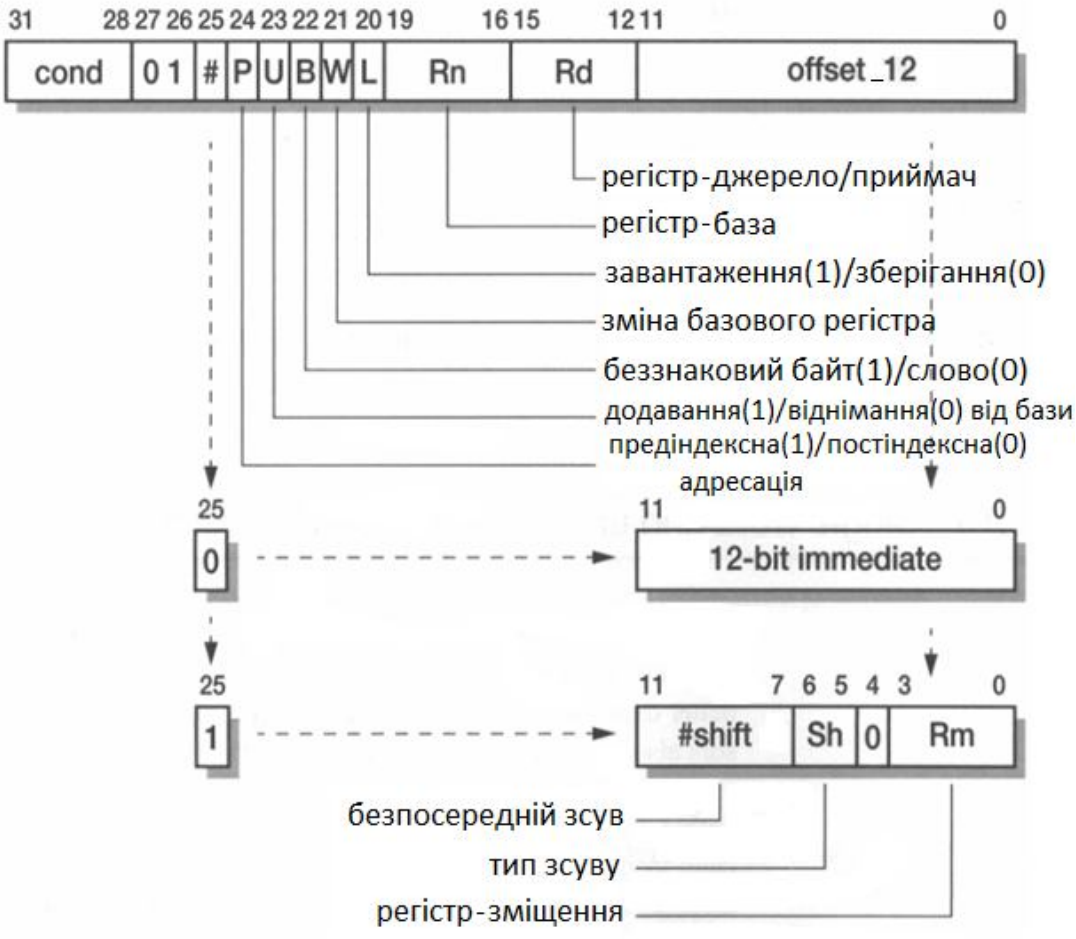


Рисунок 4.7 – Формат команд завантаження/збереження слова або байта без знаку

– 24 р (P) залежить від виду адресації, яка використовується: преіндексна (P = 1), або постіндексна (P = 0);

– 23 р (U) визначає, яка операція виконується над базою: додавання (U = 1), або віднімання (U = 0);

– 22 р (B) визначає вид операнда: байт без знаку (B=1), або слово (B=0);

– 21 р (W) приймає значення в залежності від значення 24 розряду (P): P=0, W=0, якщо виконуються команди LDR, LDRB, STR або STRB та здійснюється нормальний доступ до пам’яті; P=0, W=1, якщо виконуються інструкції LDRBT, LDRT, STRBT, STRT та здійснюється непривіле-

`

йований доступ до пам’яті (режим користувача); P=1, W=0, якщо базовий регістр не оновлюється (адресація зі зміщенням); P=1, W=1, якщо розрахована адреса пам’яті записується назад у базовий регістр (преіндексна адресація);

– 20 р (L) визначає вид операції: завантаження (L=1), або зберігання (L=0);

– 19 р…16 р (Rn) визначає один з регістрів основного регістрового файлу, який виступає в якості бази;

– 15 р … 12 р (Rd) визначає один із регістрів основного регістрового файлу в якості джерела (при збереженні) або приймача (при завантаженні);

– 11 р … 0 р (offset_12) визначає, де знаходиться зміщення: якщо 25р (#)=0, то в якості зміщення використовується 12–розрядне число; якщо 25р (#)=1, то в якості зміщення використовується один із регістрів основного регістрового файлу – R_m. Регістр R_m називають індексом, значення R_m також може зсувається на величину, яка визначається полем #shift. Вид зсуву визначається бітами 6, 5 (Sh): Sh=00 –логічний (арифметичний) вліво; 01 – логічний вправо; 10 – арифметичний вправо; 11 – циклічний вправо; 4 р = 0; 3 р … 0 р (R_m) визначає один із регістрів основного регістрового файлу – індекс.

На рисунку 4.8 наведено приклади трьох форматів команд, які відповідають рисунку 4.7.

У наведених нижче прикладах при виконанні деяких команд змінюється вміст регістра, який називають базою. Таку зміну називають індексуванням. Значення, на яке змінюється база, називають зміщенням. Останнє може зберігатися у регістрі (регістрове зміщення), або задаватися безпосереднім значенням (безпосереднє зміщення). Регістрове зміщення може масштабуватися (зсуватися вліво або вправо на відповідну величину). Адресацію, коли база змінюється до виконання команди, називають преіндексною, а якщо база

`

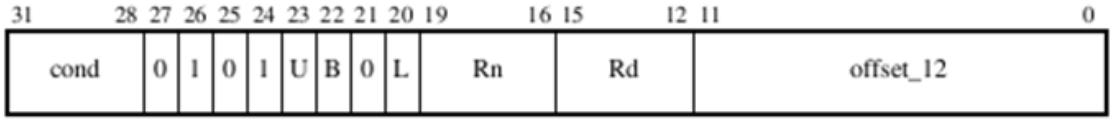
змінюється після виконання команди, називають постіндексною. Преіндексна адресація буває двох типів: без зберігання зміненої бази та зі зберіганням змінної бази.

Рисунок 4.8 – Види форматів команд завантаження/збереження слова, або байта без знаку

При преіндексній адресації без зберігання зміненої бази до виконання команди до бази додається зміщення, потім виконується команда, але після цього у базі зберігається значення, яке було до виконання команди. При преіндексній адресації зі зберіганням змінної бази до виконання команди до бази додається зміщення, потім виконується команда, після цього у базі зберігається нове значення. При постіндексній адресації база змінюється після виконання команди.

Нижче наведено приклади деяких команд розглянутого формату.

4.3.1.1 Безпосереднє зміщення (зсув) з преіндексною адресацією без зміни бази



Rn – база;

offset_12 – 12–ти розрядне зміщення;

`

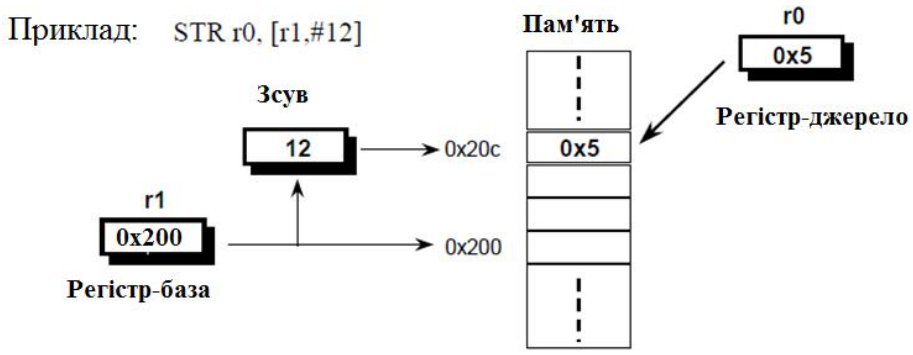


Рисунок 4.9 – Безпосереднє зміщення (зсув) з преіндексною адресацією без зміни бази

E581000C = 1110 0101 1000 0001 0000 0000 0000 1100

4.3.1.2 Безпосереднє зміщення (зсув) з преіндексною адресацією зі зміною бази

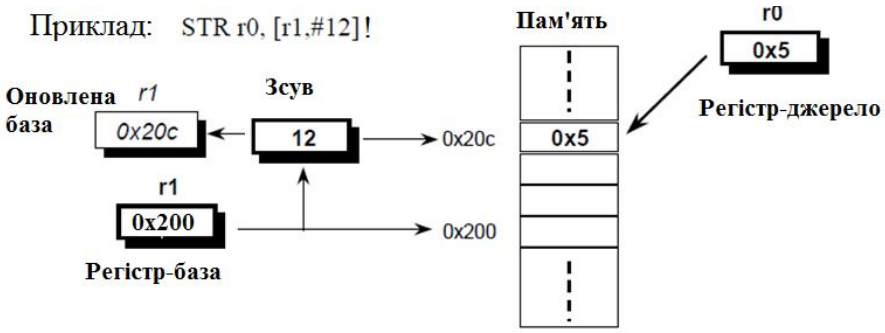


Рисунок 4.10 – Безпосередня преіндексна

E5A1000C = 1110 0101 1010 0001 0000 0000 0000 1100

4.3.1.3 Безпосередня постіндексна адресація

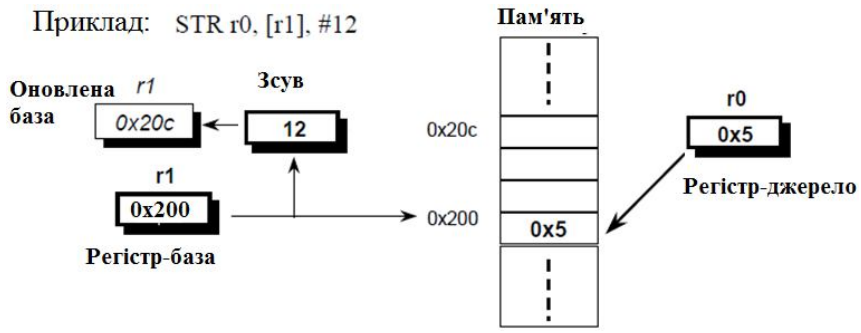
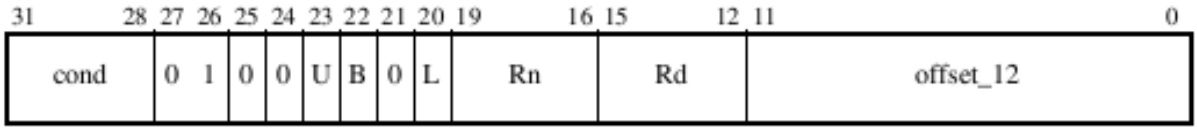


Рисунок 4.11 – Безпосередня постіндексна



E481000C = 1110 0100 1000 0001 0000 0000 0000 1100

4.3.1.4 Регістрове зміщення (зсув)з преіндексною адресацією без зміни бази

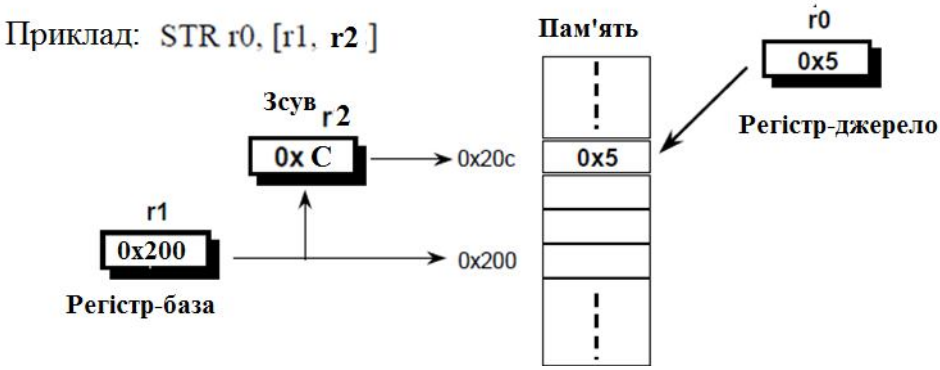
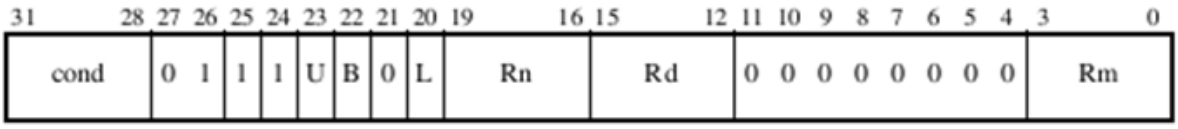


Рисунок 4.12 – Регістрове зміщення (зсув) з преіндексною адресацією без зміни бази

STR R0, [R1,R2]



4.3.1.5 Регістрове зміщення (зсув) з преіндексною адресацією та зміною бази

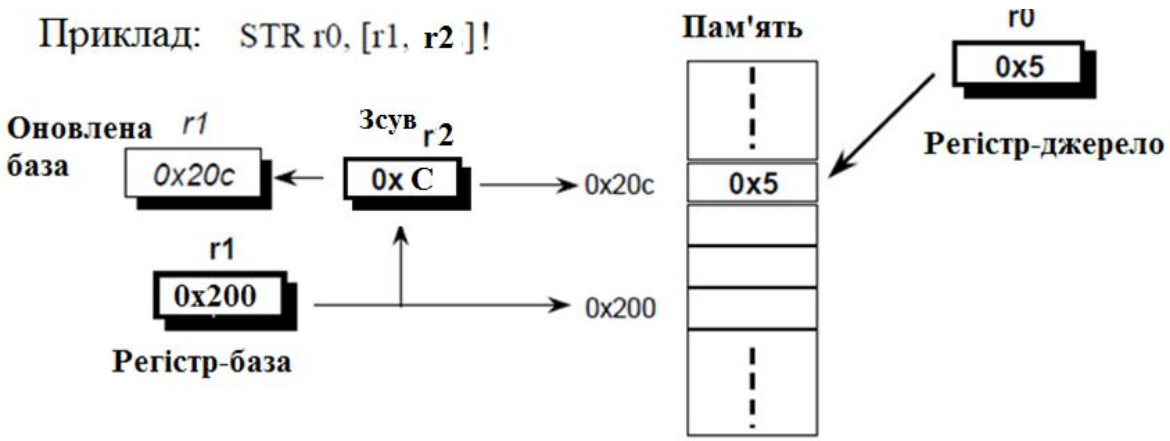
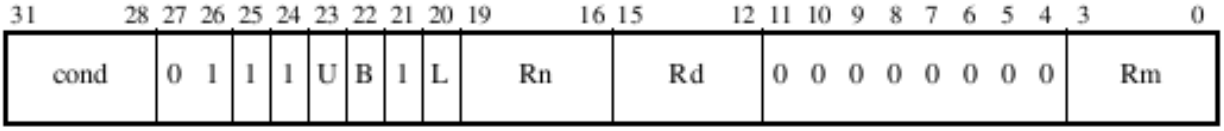


Рисунок 4.13 – Преіндексна регістрова адресація

STR R0, [R1,R2]!



4.3.1.6 Регістрове зміщення (зсув) з постіндексною адресацією

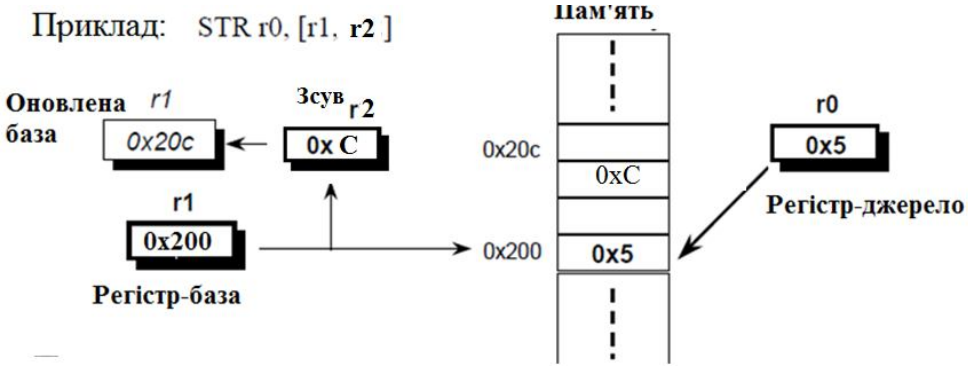
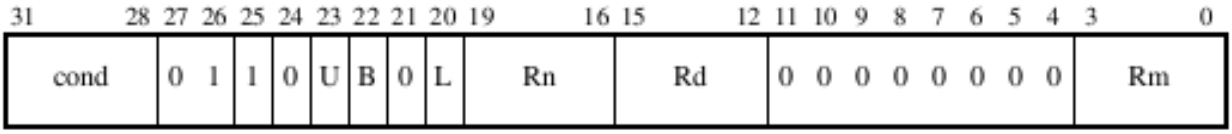


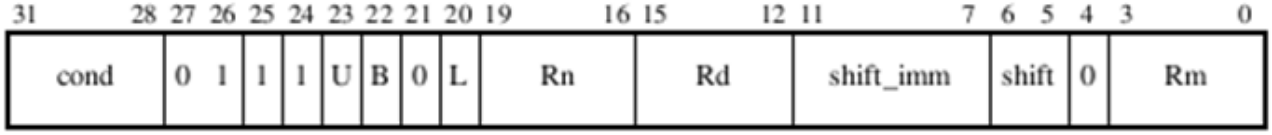
Рисунок 4.14 – Регістрове зміщення (зсув) з постіндексною адресацією

`

STR R0, [R1], R2



4.3.1.7 Регістрове зміщення із зсувом та прединдексною адресацією з масштабованим регістровим зміщенням без оновлення бази



Варіанти адресації:

[<Rn>, +/–<Rm>, LSL #<shift_imm>];

[<Rn>, +/–<Rm>, LSR #<shift_imm>];

[<Rn>, +/–<Rm>, ASR #<shift_imm>];

[<Rn>, +/–<Rm>, ROR #<shift_imm>];

[<Rn>, +/–<Rm>, RRX];

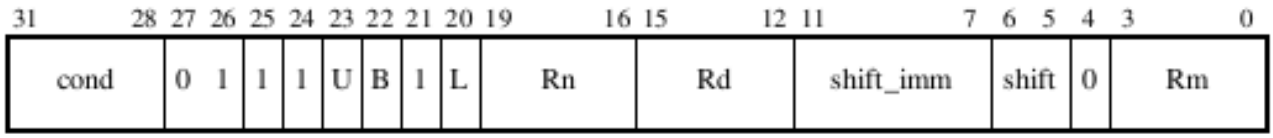
<Rn> – база;

<Rm> – індекс;

<shift_imm> – зсув;

STR R0, [R1, R2, LSL #5]; M(R1 + R2x32) = R0,

4.3.1.8 Регістрове зміщення зі зсувом та преіндексною адресацією з масштабованим регістровим зміщенням та оновленням бази

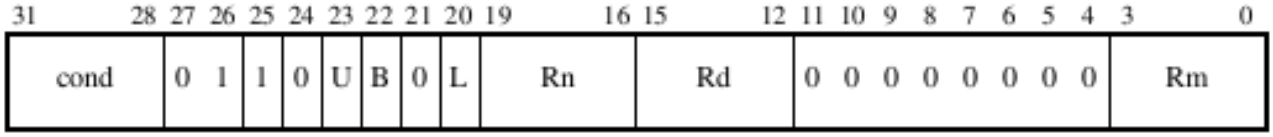


LDR R0, [R1, R2, LSL #2]! ; R0 = M(R1 + R2x4); R1 = R1 + R2x4.

`

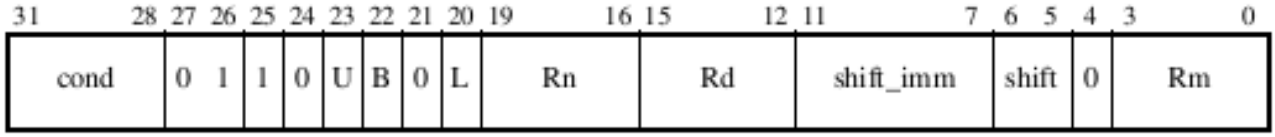
Може використовуватись, наприклад, для доступу до елементів масиву із розміром значення більшим ніж один елемент з попереднім оновленням покажчика елемента.

4.3.1.9 Регістрове зміщення з постіндексною адресацією



LDR R1, [R2], R3; R1 = M(R2) ; R2= R2 + R3.

4.3.1.10 Регістрове зміщення зі зсувом та постіндексною адресацією



LDR R0, [R1], R2, LSL #2 ; R0 = M(R1); R1 =R1+ R2x4.

Може використовуватись, наприклад, для доступу до елементів масиву із розміром більшим ніж один елемент з наступним оновленням покажчика елемента.

4.3.2 Команди завантаження/збереження половини слова, або подвійного слова та завантаження байта зі знаком

На рисунку 4.15 наведено формат команд завантаження/збереження половини слова, або подвійного слова та завантаження байта зі знаком.

`

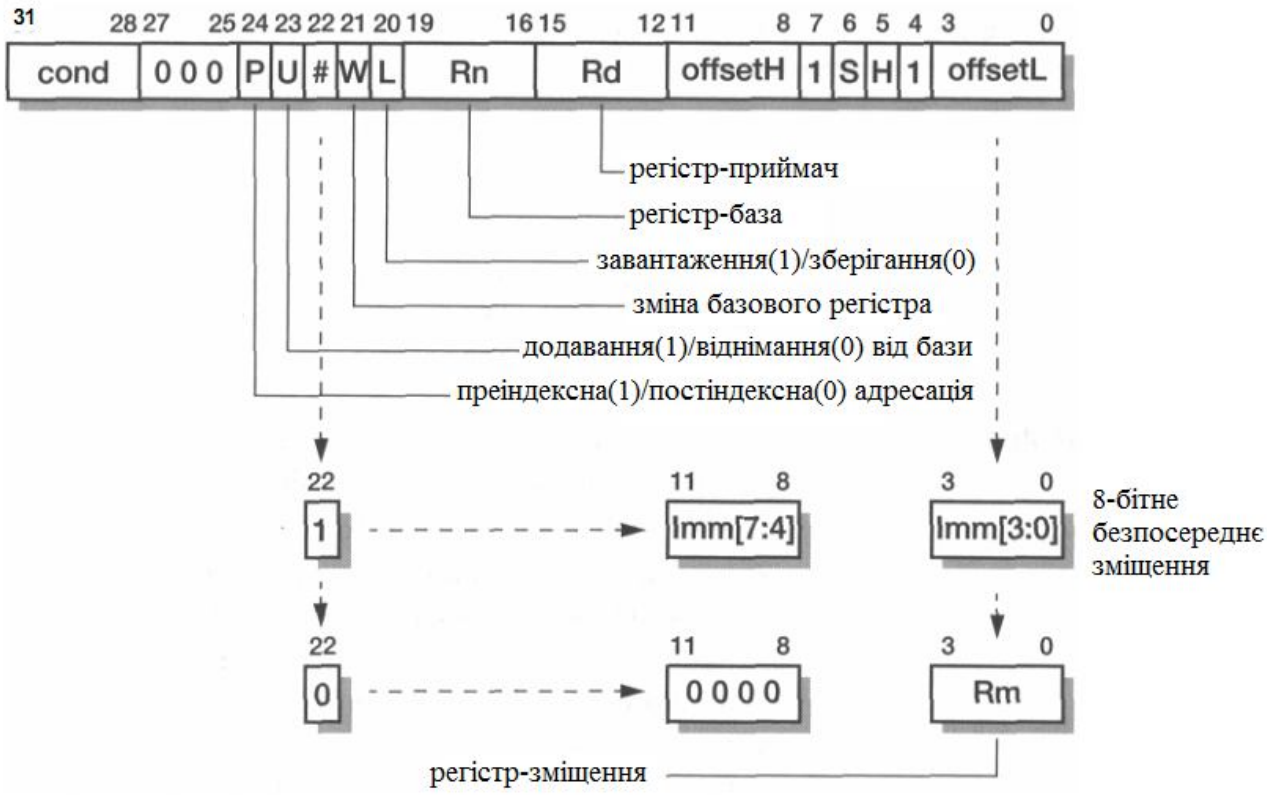


Рисунок 4.15 – Завантаження/збереження половини слова, або подвійного слова та завантаження байта зі знаком

Пояснення більшості бітів співпадає з рисунком 4.7. Нижче наведено тільки відмінності:

– 22р (#) визначає, де знаходиться зміщення відносно бази при формуванні адреси пам’яті: якщо 22р=1, то в якості зміщення використовується 8–розрядне число (розряди 11 … 8 (imm[7:4]) та 3 … 0(imm[3:0]); якщо 22р=0, то зміщення знаходиться в одному з регістрів основного регістрового файлу ;

– 21р (W) приймає значення в залежності від значення 24–ого розряду (P): P=0, W=1 при преіндексній адресації (базовий регістр оновлюється); W=0 при адресації зі зміщенням (базовий регістр не оновлюється);

– 6р (S) та 7р (H) змінюється в залежності від виконуваної команди (таблиця 4.2).

`

Таблиця 4.2 – Зміни бітів L,S та H в залежності від виконуваної команди

L	S	H	Результат
0	0	1	Зберегти половину слова
0	1	0	Завантажити подвійне слово
0	1	1	Зберегти подвійне слово
1	0	1	Завантажити половину слова без знаку
1	1	0	Завантажити байт зі знаком
1	1	1	Завантажити половину слова зі знаком

LDR|STR{<cond>}H|SH|SB|D <Rd>, <addressing_mode>

SH – Signed Halfword – половина слова зі знаком (лише LDR)

H – unsigned Halfword – половина слова без знаку

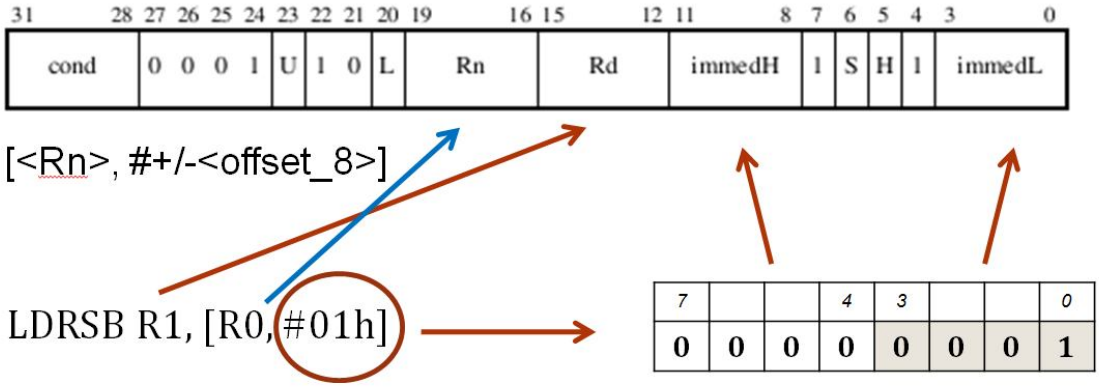
SB – Signed Byte – байт зі знаком (лише LDR)

D – Doubleword – подвійне слово.

Нижче наведено приклади деяких команд розглянутого формату.

`

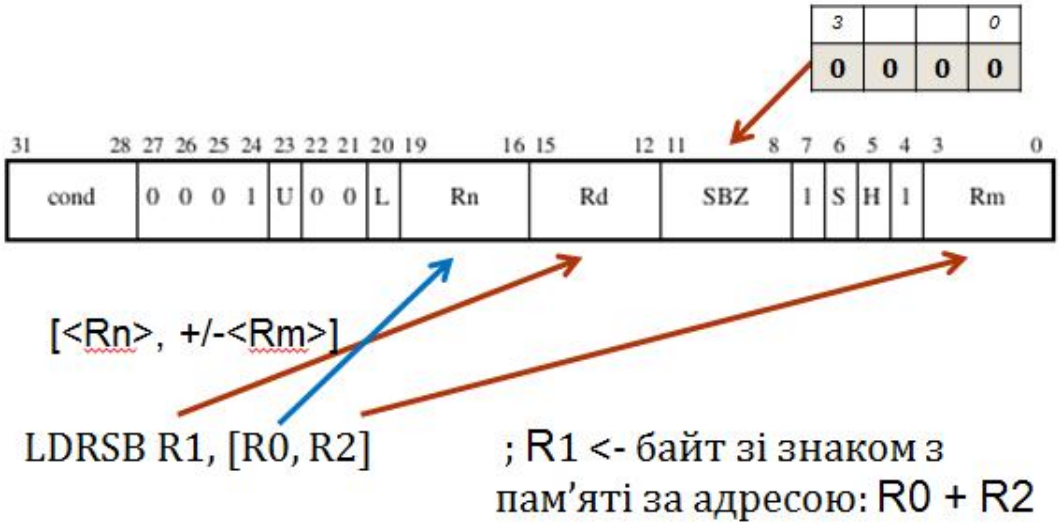
4.3.2.1 Преіндексна адресація з безпосереднім зміщенням без оновлення бази



; R1 <- байт зі знаком з пам'яті за адресою: R0 + 01h

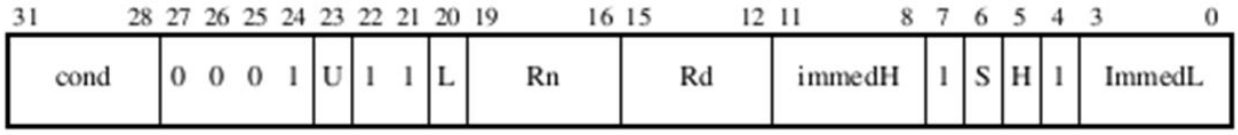
E1D010D1 = 1110 0001 1101 0000 0001 0000 1101 0001

4.3.2.2 Преіндексна адресація з регістровим зміщенням без оновлення бази



`

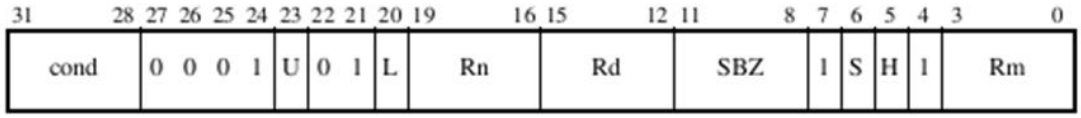
4.3.2.3 Преіндексна адресація з безпосереднім зміщенням та оновленням бази




```
LDRSB R0, [R1, #4]!        ;  R1 <- R1 + 4,
                           ; R0 <- байт зі знаком з
                           ; пам'яті за адресою R1 + 4
```

E1F100D4 = 1110 0001 1111 0001 0000 0000 1101 0100

4.3.2.4 Преіндексна адресація з регістровим зміщенням та оновленням бази

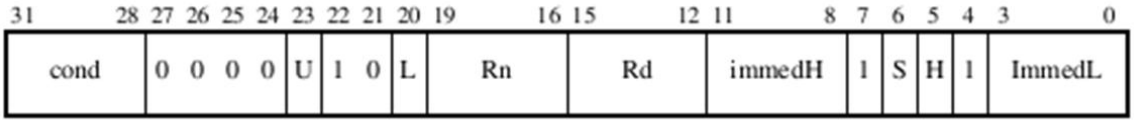



```
LDRSB R0, [R1, R2]!        ; R1 <- R1 + R2
                           ; R0 <- байт зі знаком з
                          ;пам'яті з адресою R1 + R2;
```

E1B100D2 = 1110 0001 1011 0001 0000 0000 1101 0010

`

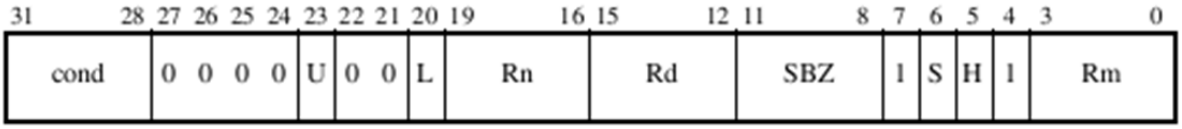
4.3.2.5 Постіндексна адресація з безпосереднім зміщенням




```
LDRSB R0, [R1], #2          ; R0 <- байт зі знаком з
                            ;пам'яті з адресою в R1
                            ; R1 <- R1 + 2
```

E0D100D2 = 1110 0000 1101 0001 0000 0000 1101 0010

4.3.2.6 Постіндексна адресація з регістровим зміщенням




```
LDRSB R0, [R1], R2          ; R0 <- байт зі знаком з
                            ;пам'яті з адресою в R1
                            ; R1 <- R1 + R2
```

E09100D2 = 1110 0000 1001 0001 0000 0000 1101 0010

4.3.3 Команди множинного запису/читання

На рисунку 4.16 наведено формат команд множинного запису/читання.

LDM|STM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}

`

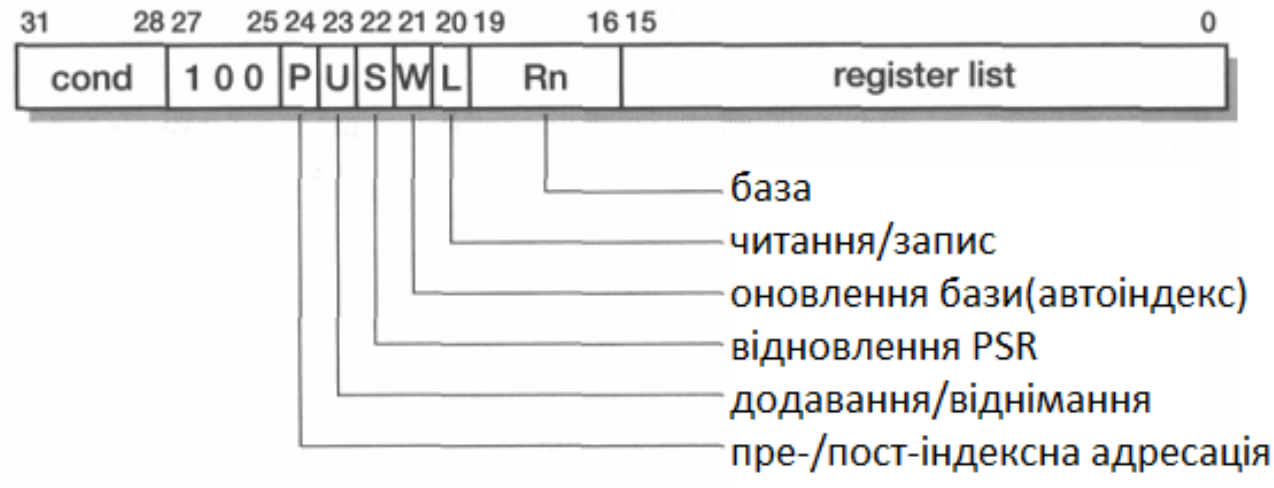


Рисунок 4.16 – Адресація множинного запису/читання

Нижче наведено пояснення окремих полів формату команд, наведеного на рисунку 4.16:

– 31 р..28 р (cond) формує компілятор в залежності від префіксу команди (таблиця 4.1), якщо останній використовується;

– 27 р дорівнює одиниці;

– 25 та 26 р дорівнюють нулю;

– 24 р (P) визначає, чи включене слово, яке адресоване базою, до діапазону комірок пам’яті (0 – так, 1– ні);

– 23 р (U) визначає, яка операція виконується над базою: додавання (U = 1), або віднімання (U = 0);

– 22 р (S) встановлюється в 1, коли користувацькі регістри копіюються під час роботи в привілейованому режимі;

– 21 р (W) визначає, чи база оновлюється після пересилки;

– 20 р (L) визначає вид операції: завантаження (L=1), або зберігання (L=0);

– 19 р..16 р (Rn) визначає один з регістрів основного регістрового файлу, який виступає в якості бази;

`

– 15 p..0 p (register list) визначає, які регістри будуть використовуватись.

Зміну бітів L, P, U в залежності від використаних команд відображає таблиця 4.3.

Таблиця 4.3 – Зміна бітів L, P, U в залежності від використаних команд

	L	P	U
LDMDA (Decrement After)	1	0	0
LDMIA (Increment After)	1	0	1
LDMDB (Decrement Before)	1	1	0
LDMIB (Increment Before)	1	1	1
STMDA (Decrement After)	0	0	0
STMIA (Increment After)	0	0	1
STMDB (Decrement Before)	0	1	0
STMIB (Increment Before)	0	1	1

На рисунку 4.17 наведено пояснення виконання команд множинного запису/читання

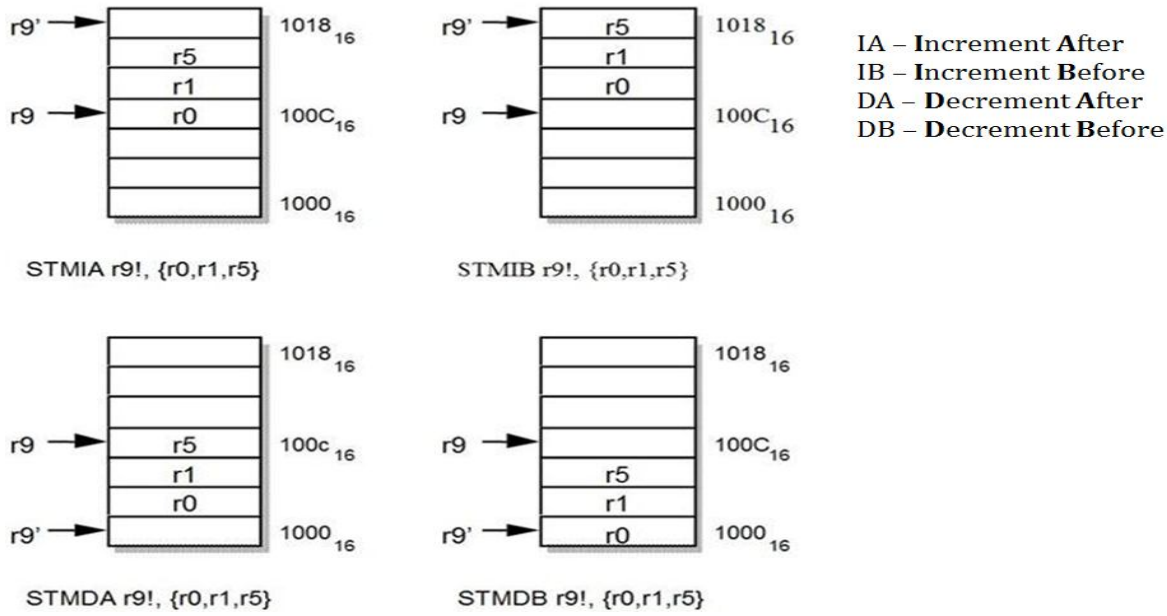
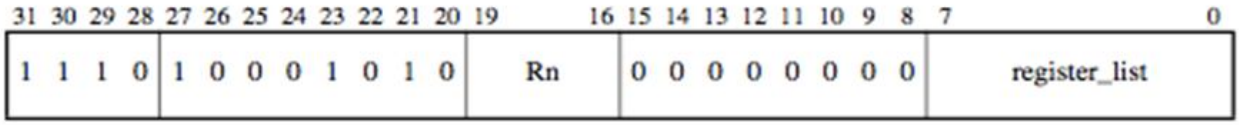


Рисунок 4.17 – Пояснення виконання команд множинного запису/читання

`

4.3.3.1 Приклад виконання команди STMIA

```
STMIA  <Rn>!, <registers>
```




```
MOV R0,#0xFFFFFFAA    ; R0 = 0xFFFFFFAA
MOV R1,#0xFFFFFFBB    ; R1 = 0xFFFFFFBB
MOV R2,#0xFFFFFFCC    ; R2 = 0xFFFFFFCC
STMIA R8,{R0,R1,R2} ; M(R8)=0xFFFFFFAA, M(R8+4)=0xFFFFFFBB,
M(R8+8)=0xFFFFFFCC; R8 = R8+12
```

E8A80007= 1110 1000 1010 1000 0000 0000 0000 0111

4.3.3.2 Приклад виконання команди STMIB

```
MOV R0,#0xFFFFFFAA    ; R0 = 0xFFFFFFAA
MOV R1,#0xFFFFFFBB    ; R1 = 0xFFFFFFBB
MOV R2,#0xFFFFFFCC    ; R2 = 0xFFFFFFCC
STMIB R8,{R0,R1,R2} ; M(R8+4)=0xFFFFFFAA, M(R8+8)=0xFFFFFFBB,
M(R8+12)=0xFFFFFFCC; R8 = R8+12
```

E9A80007= 1110 1001 1010 1000 0000 0000 0000 0111

4.3.3.3 Приклад виконання команди STMDA

```
MOV R0,#0xFFFFFFAA    ; R0 = 0xFFFFFFAA
MOV R1,#0xFFFFFFBB    ; R1 = 0xFFFFFFBB
MOV R2,#0xFFFFFFCC    ; R2 = 0xFFFFFFCC
STMDA R8,{R0,R1,R2} ; M(R8)=0xFFFFFFAA, M(R8-4)=0xFFFFFFBB, M(R8-
8)=0xFFFFFFCC; R8 = R8-12
```

E8280007= 1110 1000 0010 1000 0000 0000 0000 0111

`

4.3.3.4 Приклад виконання команди STMDB

STMDB SP!, <registers>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	0
1	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1	0	R	0	0	0	0	0	0	register_list	

```
MOV R0,#0xFFFFFFAA    ; R0 = 0xFFFFFFAA
MOV R1,#0xFFFFFFBB    ; R1 = 0xFFFFFFBB
MOV R2,#0xFFFFFFCC    ; R2 = 0xFFFFFFCC
STMDB R8,{R0,R1,R2}; M(R8-4)=0xFFFFFFAA, M(R8-8)=0xFFFFFFBB, M(R8-
12)=0xFFFFFFCC; R8 = R8-12;
```

E9280007= 1110 1001 0010 1000 0000 0000 0000 0111

4.4 Адресація команд роботи з співпроцесорами

На рисунку 4.18 наведено формат команд роботи зі співпроцесорами. Команди цього класу використовуються для читання (LDC) або запису (STC) регістрів співпроцесорів безпосередньо з / в пам'ять.

<LDC|STC>{<cond>}{L} <coproc>,<CRd>,<addressing_mode>

Рисунок 4.18 –Формат команд роботи з співпроцесорами

Нижче наведено пояснення окремих полів формату команд, наведених на рисунку 4.18:

– 31 р..28 р (cond) формує компілятор в залежності від префіксу команди (таблиця 4.1), якщо останній використовується;

– 27 р та 26 р дорівнюють одиниці;

`

– 25 р дорівнює нулю;

– 24 р (P) визначає тип адресації: P=0 – постіндексна (біт W визначає, яка саме), P=1 – преіндексна (біт W визначає, яка саме);

– 23р (U) визначає яка операція виконується над базою: додавання (U = 1), або віднімання (U = 0);

– 22 р (N) значення довжини передачі задається співпроцесором: 0–1 слово; 1 – більше;

– 21 р (W) визначає, чи база оновлюється після пересилки: 0 – ні; 1 – так;

– 20 р (L) визначає вид операції: завантаження (L=1), або зберігання (L=0);

– 19 р..16 р (Rn) визначає один з регістрів основного регістрового файлу, який виступає в якості бази;

– 15 р..12 р (CRd) визначає регістр співпроцесора;

– 11 р..8 р (CP#) визначає номер співпроцесора;

– 7 р..0 р (8–bit offset) –8–бітне зміщення.

Поле CP # використовується для ідентифікації того співпроцесора, який повинен передати або прийняти дані, при цьому команду виконає тільки той співпроцесор, номер якого збігається з номером у полі CP #.

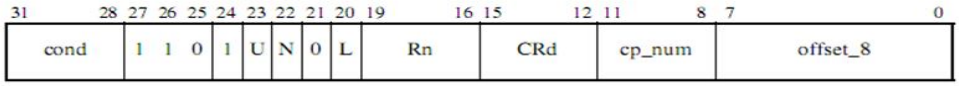
Регістр CRd завжди виконує роль регістра, який має бути переданий (або перший регістр зі списку при блоковій передачі), а біт N визначає довжину передачі: при N = 0 обмін проводитися тільки з одним регістром (CRd), при N = 1 – зі всіма регістрами (для перемикання контексту завдань).

Нижче наведено приклади деяких команд розглянутого формату:

`

4.4.1 Команда з безпосереднім зміщенням з преіндексною адресацією без зміни бази

[<Rn>, #+/-<offset_8>*4]



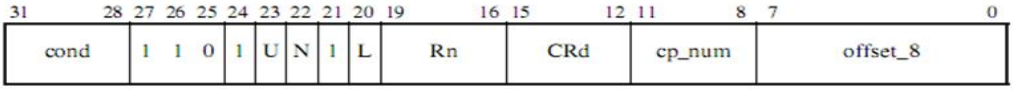
STC p2, c10, [R2, #24]

ED82A206 = 1110 1101 1000 0010 1010 0010 0000 0110

R2 c10 p2 24/4=6

4.4.2 Команда з безпосереднім зміщенням з преіндексною адресацією зі зміною бази

[<Rn>, #+/-<offset_8>*4]!



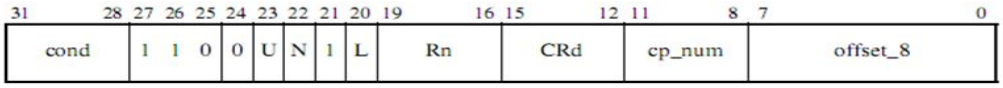
STC p2, c10, [R2, #24] !

EDA2A206 = 1110 1101 1010 0010 1010 0010 0000 0110

R2 c10 p2 24/4=6

4.4.3 Команда з безпосереднім зміщенням з постіндексною адресацією

[<Rn>], #+/-<offset_8>*4



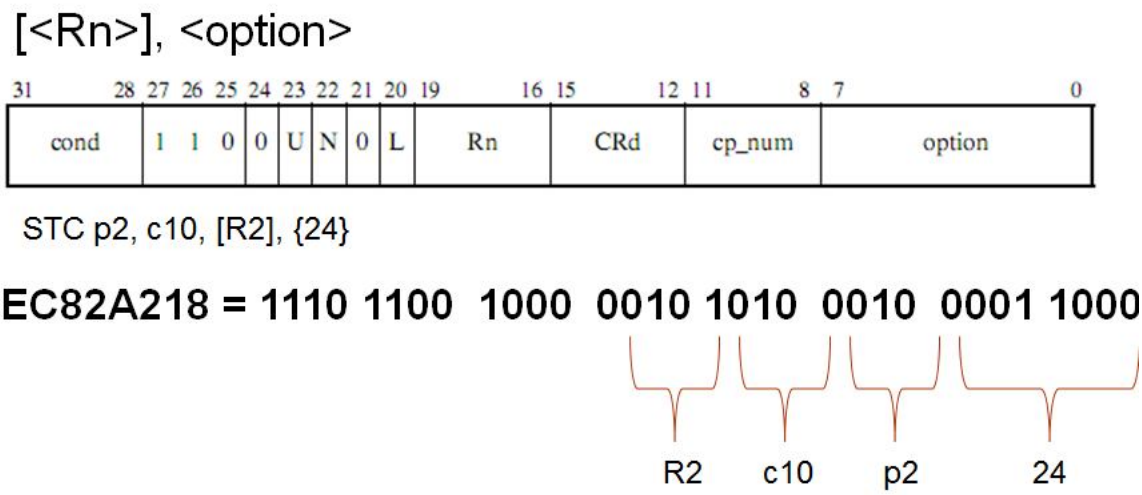
STC p2, c10, [R2], #24

ECA2A206 = 1110 1100 1010 0010 1010 0010 0000 0110

R2 c10 p2 24/4=6

`

4.4.4 Команда з безіндексною адресацією



4.5 Приклади способів адресацій

1 Команда:ADDS R1, R2, #0xFF00.

Коментар до команди: R1←R2+0xFF00, запис до регістра R1 результату додавання R2 та 0xFF00.

Способи адресації операндів: лівий – регістрова, правий – безпосередня, отримувач результату –регістрова.

2 Команда: LDRSB R0, [R1, R2].

Коментар до команди: завантаження байта зі знаком у регістр R0 з пам’яті за адресою: (R1 + R2).

Способи адресації операндів: джерело – преіндексна з регістровим зміщенням без оновлення бази, отримувач – регістрова.

3 Команда: LDR R0, [R1], R2, LSR#2.

Коментар до команди: завантаження слова: R0 ← M (R1); R1← R1 + (R2·4).

Способи адресації операндів: джерело – постіндексна з масштабованим регістровим зміщенням, отримувач –регістрова.

4 Команда: ADD R1, R2, #0xFF00.

`

Коментар до команди: R1←R2+0xFF00, запис до регістра R1 результату додавання R2 та 0xFF00.

Способи адресації операндів: перший операнд–регістрова–R2, другий операнд–безпосередня–0xFF00, отримувач–регістрова– R1.

5 Команда: ADDS R9, R4, R5, LSR R1.

Коментар до команди: R9 ← R4 + $R5/2^{R1}$, запис до регістра R9 результату додавання R4 та $R5/2^{R1}$.

Способи адресації операндів: перший операнд – регістрова, другий – регістрова з масштабуванням, отримувач – регістрова.

6 Команда: STR R0, [R2, #12].

Коментар до команди: зберігання слова: M (R1 + 12) ← R0.

Способи адресації операндів: джерело – регістрова, приймач – преіндексна з безпосереднім зміщенням без оновлення бази.

7 Команда: STR R0, [R1, #0xAB]!.

Коментар до команди: R1 ← R1 + 0xAB; M (R1) ← R0 (зберігання слова).

Способи адресації операндів: джерело – регістрова, отримувач – преіндексна з безпосереднім зміщенням та оновленням бази.

8 Команда: STR R5, [R2], #25.

Коментар до команди: M (R2) ← R5; R2 ← R2 + 25 (зберігання слова).

Способи адресації операндів: джерело – регістрова, отримувач – постіндексна з безпосереднім зміщенням.

9 Команда: STR R0, [R1, R2].

Коментар до команди: M (R1+R2) ← R0 (зберігання слова).

Способи адресації операндів: джерело – регістрова, приймач – преіндексна з регістровим зміщенням без оновлення бази.

10 Команда: STR R1, [R2].

`

Коментар до команди: M (R2) ← R1 (зберігання слова)

Способи адресації операндів: джерело – регістрова, отримувач –

непряма.

11 Команда: STR R0, [R1], R2.

Коментар до команди: M (R1) ← R0; R1 ← R1 + R2 (зберігання слова)

Способи адресації операндів: джерело – регістрова, отримувач – постіндексна з регістровим зміщенням.

12 Команда: STR R0, [R1, R2, LSL#2].

Коментар до команди: M (R1+R2*2^2) ← R0 (зберігання слова).

Способи адресації операндів: джерело – регістрова, отримувач – преіндексна з масштабованим регістровим зміщенням без оновлення бази.

13 Команда: STR R0, [R1, R2, LSL#2]!.

Коментар до команди: R1 ← R1 + R2*2^2; M (R1) ← R0 (зберігання слова).

Способи адресації операндів: джерело – регістрова, отримувач – преіндексна з масштабованим регістровим зміщенням та оновленням бази.

14 Команда: STR R0, [R1], R2 LSL#2.

Коментар до команди: M (R1) ← R0; R1← R1 + R2*2^2 (зберігання слова).

Способи адресації операндів: джерело – регістрова, отримувач – постіндексна з масштабованим регістровим зміщенням.

15 Команда: STR R2, [R1].

Коментар до команди: M (R1) ← R2 (зберігання слова).

Способи адресації операндів: джерело – регістрова, отримувач – непряма.

16 Команда: ANDS R1, R0, #0x0FA00000.

`

Коментар до команди: R1 ← R0 & 0x0FA00000 , запис до регістра R1 результату логічного множення R0 та 0x0FA00000.

Способи адресації операндів: перший операнд – регістрова, другий операнд – безпосередня, отримувач – регістрова.

17 Команда: ORRS R3, R0, #0xAB.

Коментар до команди: R3 ← R0 V 0xAB, запис до регістра R3 результату логічного додавання R0 та 0xAB.

Способи адресації операндів: перший операнд – регістрова, другий операнд – безпосередня, отримувач –регістрова.

18 Команда: EORS R3, R0, #0x35.

Коментар до команди: R3 ← R0 $\oplus$ 0x35, запис до регістра R3 результату логічного додавання за модулем 2: R0 та 0x35.

Способи адресації операндів:перший операнд – регістрова, другий – безпосередня, отримувач – регістрова.

19 Команда: SBCS R3, R2, #0xDE.

Коментар до команди: R3 ← R2 – 0xDE – $\overline{C}_F$, запис до регістра R3 результату віднімання від регістра R2 безпосереднього операнда 0xDE та інверсії C_F ($\overline{C}_F$).

Способи адресації операндів: перший операнд – регістрова, другий – безпосередня, $\overline{C}_F$ – неявна, отримувач – регістрова

20 Команда: LDRSB R0, [R1, #35].

Коментар до команди: завантаження байта зі знаком –

R0 ← M (R1 + 35).

Способи адресації операндів: джерело – преіндексна з безпосереднім зміщенням без оновлення бази, отримувач – регістрова.

21 Команда: LDRB R0, [R1, 0x35]!.

Коментар до команди: завантаження байта –

R1 ← R1 ,+ 0x35; R0 ← M (R1).

`

Способи адресації операндів: джерело – преіндексна з безпосереднім зміщенням та оновленням бази, отримувач – регістрова.

22 Команда: LDR R0, [R1], #0x78.

Коментар до команди: завантаження слова – R0 ← M (R1); R1← R1 + 0x78.

Способи адресації операндів: джерело – постіндексна з безпосереднім зміщенням, отримувач – регістрова.

23 Команда: LDR R0, [R1, R2].

Коментар до команди: завантаження слова – R0 ← M (R1 + R2).

Способи адресації операндів: джерело – преіндексна з регістровим зміщенням без оновлення бази, отримувач – регістрова.

24 Команда: LDR R0, [R1, R2]!.

Коментар до команди: завантаження слова – R1 ← R1 + R2; R0 ← M(R1).

Способи адресації операндів: джерело – преіндексна з регістровим зміщенням та оновленням бази, отримувач – регістрова.

25 Команда: LDR R0, [R1], R2.

Коментар до команди: завантаження слова – R0 ← M (R1); R1← R1 + R2.

Способи адресації операндів: джерело – постіндексна с регістровим зміщенням, отримувач –регістрова.

26 Команда: LDR R0, [R1, R2, LSL#2].

Коментар до команди: завантаження слова – R0 ← M (R1 + R2*2^2).

Способи адресації операндів: джерело – преіндексна з масштабованим регістровим зміщенням без оновлення бази, отримувач – регістрова.

27 Команда: LDR R0, [R1, R2, LSL#2]!.

Коментар до команди: завантаження слова – R1 ← R1 + R2*2^2;

`

R0 ← M(R1).

Способи адресації операндів: джерело – преіндексна з масштабованим регістровим зміщенням та оновленням бази, отримувач – регістрова.

28 Команда: LDR R0, [R1], R2, LSL#2.

Коментар до команди: завантаження слова – R0 ← M (R1);

R1← R1 + R2*2^2.

Способи адресації операндів: джерело – постіндексна з масштабованим регістровим зміщенням, отримувач – регістрова.

29 Команда: LDR R2, [R1].

Коментар до команди: завантаження слова – R2 ← M (R1).

Способи адресації операндів: джерело – непряма, отримувач – регістрова.

30 Команда: RSCS R3, R1, R2.

Коментар до команди: R3 ← R2 - R1 - $\overline{C_F}$, запис до регістра R3 результату зворотного віднімання від регістра R2 регістра R1та інверсії прапорця C_F ($\overline{C_F}$).

Способи адресації операндів: перший операнд – регістрова, другий – регістрова, прапорець $\overline{C}_F$ – неявна, отримувач –регістрова.

`

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1) Які види адресацій використовуються в командах ARM?

2) На які основні групи можна розбити всі команди ARM?

3) За якої умови команда проводиться через конвеєр як команда NOR (немає операції)?

4) Дати пояснення кожного з полів формату команд обробки операндів.

5) Як в командах ARM задається умовне виконання?

6) Як в командах ARM задається встановлення прапорців регістра CPSR?

7) Пояснити як відбувається виконання команд з преіндексною та постіндексною адресаціями?

8) Пояснити виконання команд множинного запису/читання.

9) Пояснити виконання команд логічного, арифметичного та циклічного зсувів.

10) Дати пояснення кожного з полів формату команд завантаження/збереження.

11) Дати пояснення кожного з полів формату команд роботи зі співпроцесорами.

`

5 ОПИС КОМАНД ЯДРА ARM7TDMI

5.1 Набір команд ядра ARM7

Скорочений набір інструкцій ARM7 ARM7 представлений у таблиці 5.1.

Таблиця 5.1 – Скорочений список інструкцій ARM7

Операції	Коментар	Синтаксис Асемблера
Пересилання	Пересилання	**MOV** {cond} {S} Rd, <Oprnd2>
	Пересилання з інверсією	**MVN** {cond} {S} Rd, <Oprnd2>
	Пересилання регістра SPSR в регістр	**MRS** {cond} Rd, SPSR
	Пересилання регістра CPSR в регістр	**MRS** {cond} Rd, CPSR
	Пересилання регістра в регістр SPSR	**MSR** {cond} SPSR {field}, Rm
	Пересилання регістра в регістр CPSR	**MSR** {cond} CPSR {field}, Rm
	Пересилання константи у прапорці SPSR	**MSR** {cond} SPSR_f, # 32bit_Imm
	Пересилання константи у прапорці CPSR	**MSR** {cond} CPSR_f, # 32bit_Imm
Арифметичні	Додавання	**ADD** {cond} {S} Rd, Rn, <Oprnd2>
	Додавання з перенесенням	**ADC** {cond} {S} Rd, Rn, <Oprnd2>

`

Продовження таблиці 5.1

Операції	Коментар	Синтаксис Асемблера
	Віднімання	**SUB** {cond} {S} Rd, Rn, <Oprnd2>
	Віднімання з перенесенням	**SBC** {cond} {S} Rd, Rn, <Oprnd2>
	Зворотнє віднімання	**RSB** {cond} {S} Rd, Rn, <Oprnd2>
	Зворотнє віднімання з перенесенням	**RSC** {cond} {S} Rd, Rn, <Oprnd2>
	Множення	**MUL** {cond} {S} Rd, Rm, Rs
	Множення з накопиченням	**MLA** {cond} {S} Rd, Rm, Rs, Rn
	Множення довгих беззнакових чисел	**UMULL** {cond} {S} RdLo, RdHi, Rm, Rs
	Множення беззнакових чисел з накопиченням	**UMLAL** {cond} {S} RdLo, RdHi, Rm, Rs
	Множення довгих чисел зі знаком	**SMULL** {cond} {S} RdLo, RdHi, Rm, Rs
	Множення довгих чисел зі знаком з накопиченням	**SMLAL** {cond} {S} RdLo, RdHi, Rm, Rs
	Порівняння	**CMP** {cond} Rd, <Oprnd2>
	Порівняння від’ємне	**CMN** {cond} Rd, <Oprnd2>
Логічні	Перевірка на І	**TST** {cond} Rn, <Oprnd2>

`

Продовження таблиці 5.1

Операції	Коментар	Синтаксис Асемблера
	Перевірка на еквівалентність	**TEQ** {cond} Rn, <Oprnd2>
	Логічне І	**AND** {cond} {S} Rd, Rn, <Oprnd2>
	Виключне АБО	**EOR** {cond} {S} Rd, Rn, <Oprnd2>
	Логічне АБО	**ORR** {cond} {S} Rd, Rn, <Oprnd2>
	Скидання заданих бітів	**BIC** {cond} {S} Rd, Rn, <Oprnd2>>
Перехід	Перехід	**B** {cond} label
	Перехід зі збереженням адреси	**BL** {cond} label
	Перехід та зміна набору інструкцій	**BX** {cond} label
	Перехід зі зміною набору інструкцій та збереженням адреси	**BLX** {cond} label
Завантаження (читання)	слова	**LDR** {cond} Rd, <a_mode2>
	слова з перевагою режиму користувача	**LDR** {cond} T Rd, <a_mode2P>
	байта	**LDR** {cond} B Rd, <a_mode2>
	байти з перевагою режиму користувача	**LDR** {cond} BT Rd, <a_mode2P>

`

Продовження таблиці 5.1

Операції	Коментар	Синтаксис Асемблера
	байта зі знаком	**LDR** {cond} SB Rd, <a_mode3>
	напівслова	**LDR** {cond} H Rd, <a_mode3>
	напівслова зі знаком	**LDR** {cond} SH Rd, <a_mode3>
Завантаження (читання) групи регістрів	з попередніми інкрементом	**LDM** {cond} IB Rd {!}, <reglist> {^}
	з наступним інкрементом	**LDM** {cond} IA Rd {!}, <reglist> {^}
	з попередніми декрементом	**LDM** {cond} DB Rd {!}, <reglist> {^}
	з наступним декрементом	**LDM** {cond} DA Rd {!}, <reglist> {^}
	операція над стеком	**LDM** {cond} <a_mode4L> Rd {!}, <reglist>
	операція над стеком і відновлення CPSR	**LDM** {cond} <a_mode4L> Rd {!}, <reglist+pc> ^
	операція над стеком з регістрами користувача	**LDM** {cond} <a_mode4L> Rd {!}, <reglist> ^
Зберігання (запис)	слова	**STR** {cond} Rd, <a_mode2>
	слова з перевагою режиму користувача	**STR** {cond} T Rd, <a_mode2P>

`

Продовження таблиці 5.1

Операції	Коментар	Синтаксис Асемблера
	байта	**STR** {cond} B Rd, <a_mode2>
	байти з перевагою режиму користувача	**STR** {cond} BT Rd, <a_mode2P>
	напівслова	**STR** {cond} H Rd, <a_mode3>
Зберігання (запис) групи регістрів	з попередніми інкрементом	**STM** {cond} IB Rd {!}, <reglist> {^}
	з наступним інкрементом	**STM** {cond} IA Rd {!}, <reglist> {^}
	з попередніми декрементом	**STM** {cond} DB Rd {!}, <reglist> {^}
	з наступним декрементом	**STM** {cond} DA Rd {!}, <reglist> {^}
	операція над стеком	**STM** {cond} <a_mode4S> Rd {!}, <reglist>
	операція над стеком з регістрами користувача	**STM** {cond} <a_mode4S> Rd {!}, <reglist> ^
Обмін між регістром і пам’яттю	слів	**SWP** {cond} Rd, Rm, [Rn]
	байт	**SWP** {cond} B Rd, Rm, [Rn]
Команди роботи з співпроцесорами	Операція над даними	**CDP** {cond} p <cpnum>, <op1>, CRd, CRn, CRm, <op2>

`

Продовження таблиці 5.1

Операції	Коментар	Синтаксис Асемблера
	Пересилання в ARM–регістр з співпроцесора	**MRC** {cond} p <cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Пересилання в співпроцесор з ARM–регістра	**MCR** {cond} p <cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Завантаження (читання)	**LDC** {cond} p <cpnum>, CRd, <a_mode5>
	Зберігання (запис)	**STC** {cond} p <cpnum>, CRd, <a_mode5>
Програмне переривання		**SWI** 24bit_Imm

5.2 Дослідження виконання команд у налагоджувачі µVision Keil 4

5.2.1 Інсталяція

Завантажити µVision Keil 4 можна з сайту або з локального диску W. Після запуску файлу інсталятора з’явиться вікно привітання (рисунок 5.1). Далі необхідно виконати приведену нижче на рисунках послідовність дій.

`

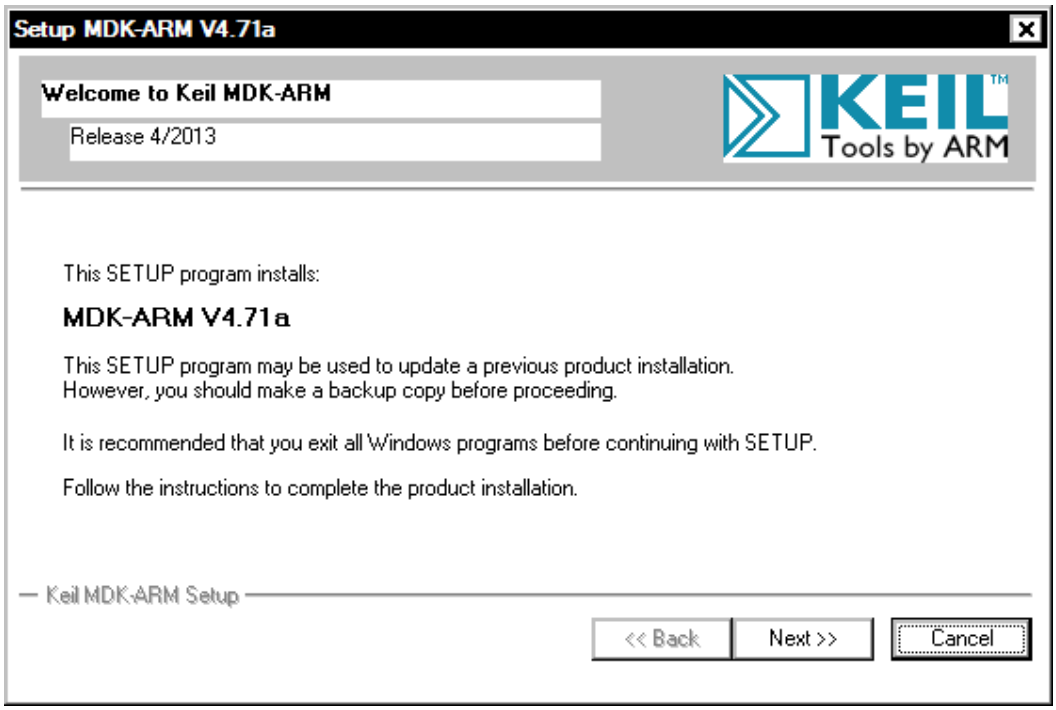


Рисунок 5.1 – Вікно привітання

Тиснемо «Next», з'являється вікно ліцензійної угоди, ставимо відмітку, як зображено на рисунку 5.2, і знову натискаємо «Next».

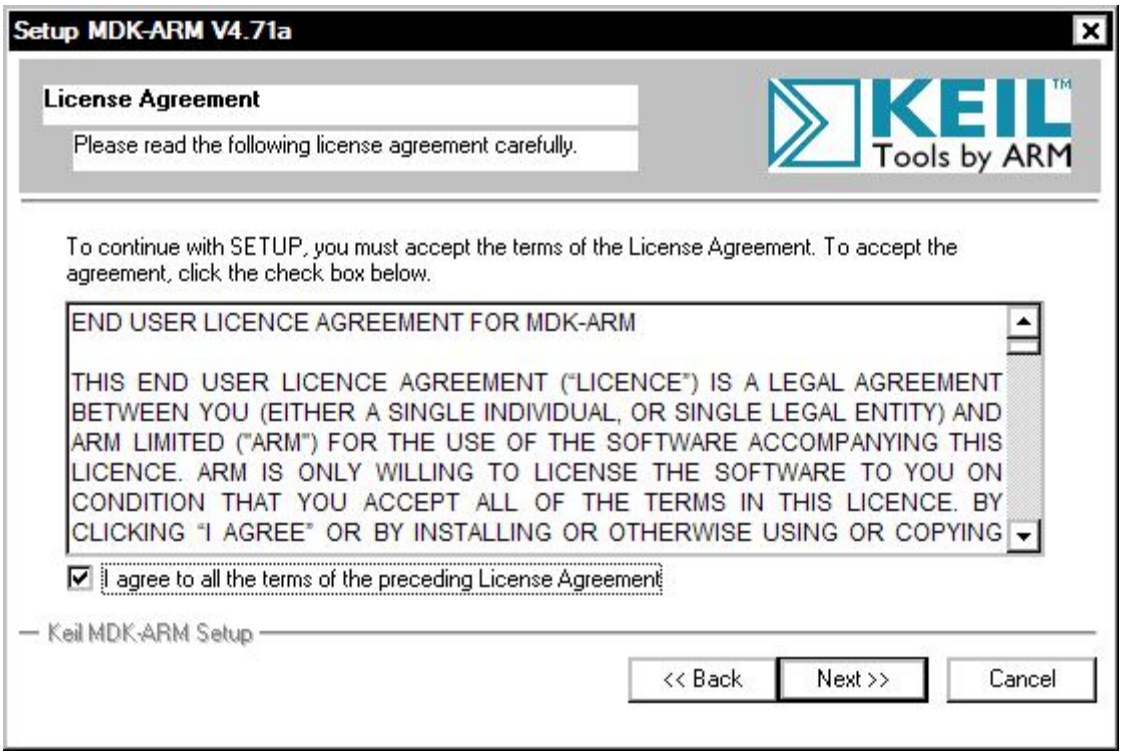


Рисунок 5.2 – Ліцензійна угода

`

У вікні вибору місця встановлення вказуємо шлях (рисунок 5.3) і натискаємо «Next».

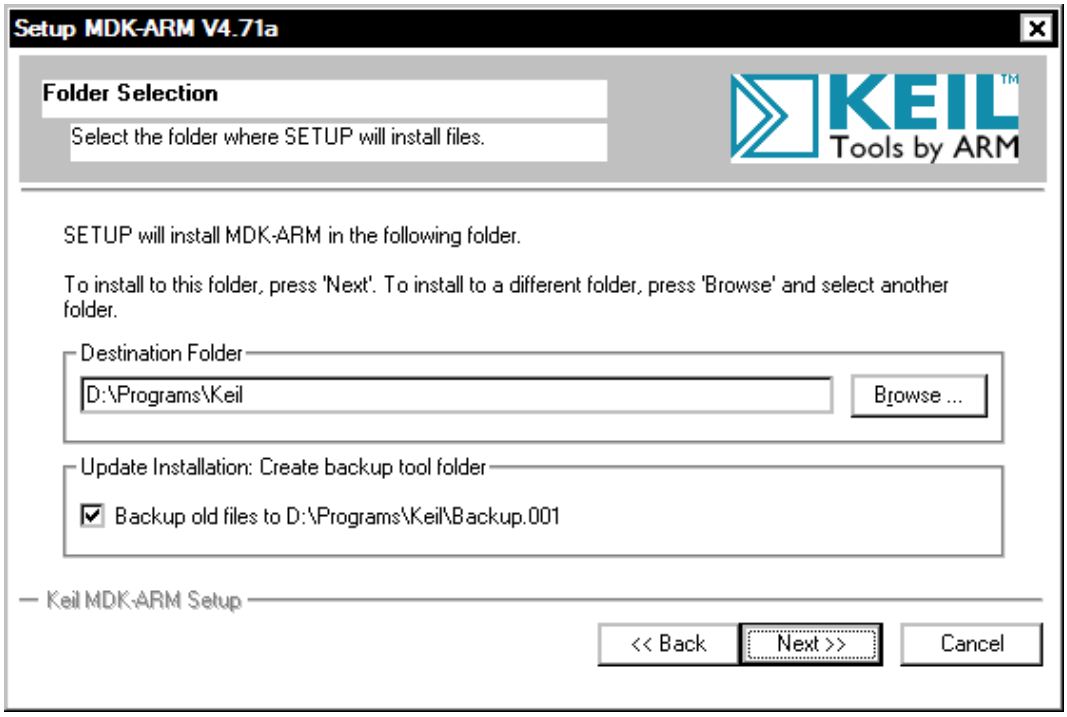


Рисунок 5.3 – Вибір місця встановлення

У наступному вікні вказуємо дані користувача (рисунок 5.4) і тиснемо «Next».

В останньому вікні вибираємо встановлення драйверу і тиснемо «Finish» (рисунок 5.5), після чого інсталяція завершиться і програма буде готова до використання.

`

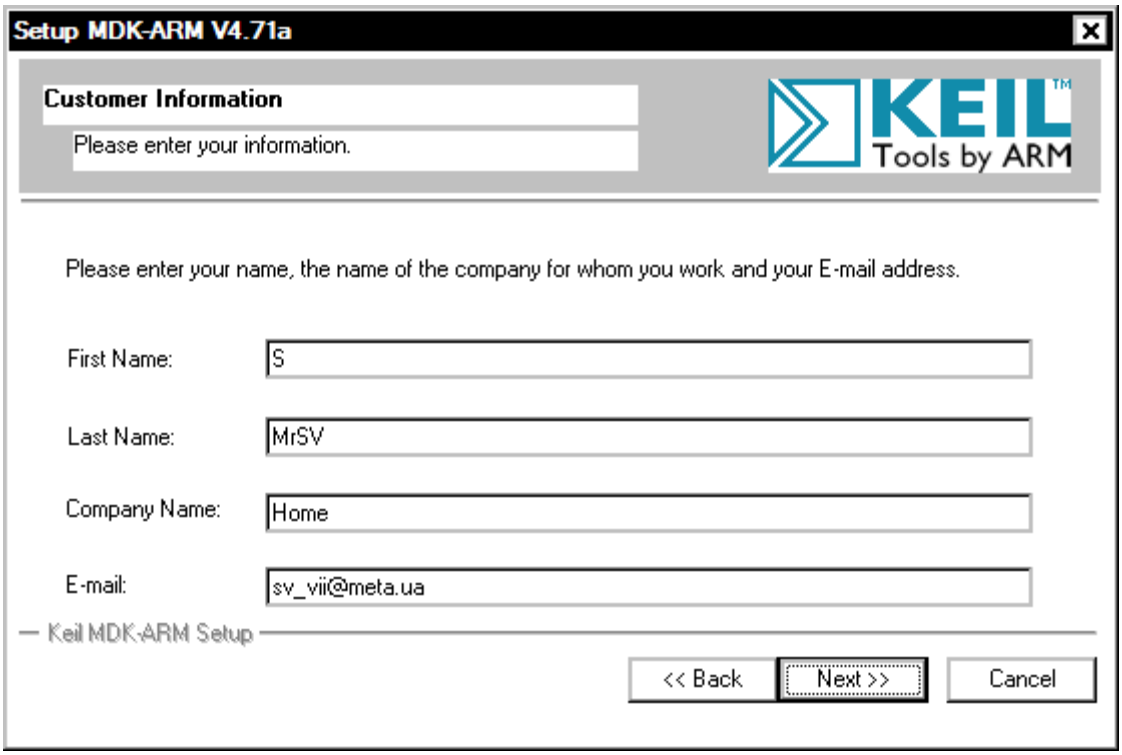


Рисунок 5.4 – Введення даних користувача

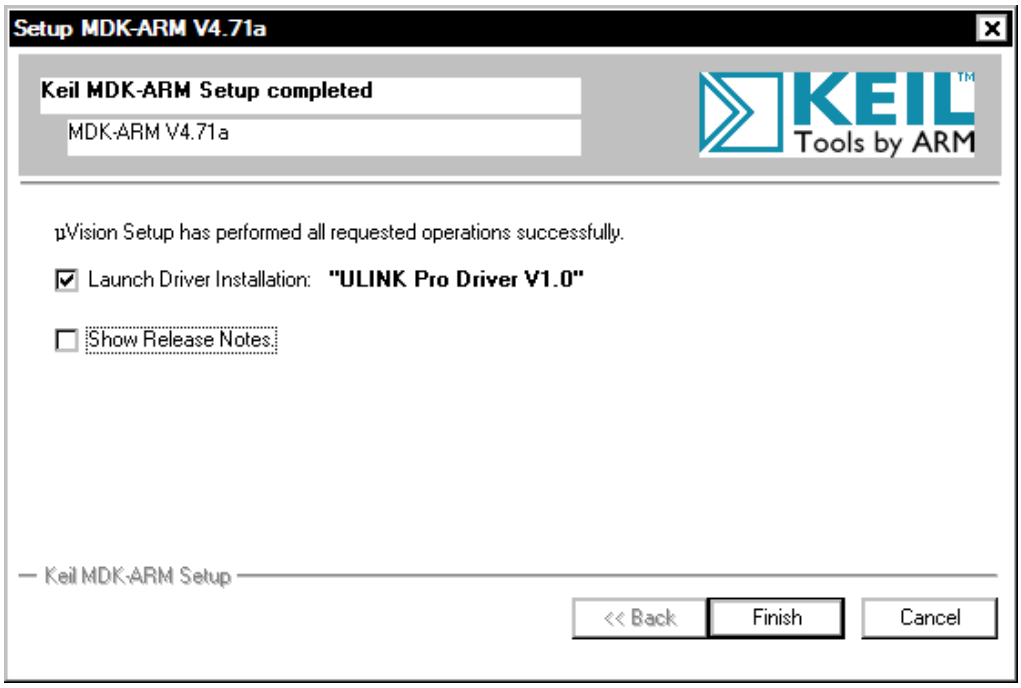


Рисунок 5.5 – Завершення інсталяції

5.2.2 Перший запуск середовища

Після того, як було встановлено µVision Keil 4, можна здійснити його перший запуск. Для запуску середовища в операційній системі Windows можна скористатися ярликом на робочому столі, в підменю програм головного меню «Пуск», або з папки, в яку було проведено встановлення.

Після запуску середовища можна побачити вікно, яке зображено на рисунку 5.6.

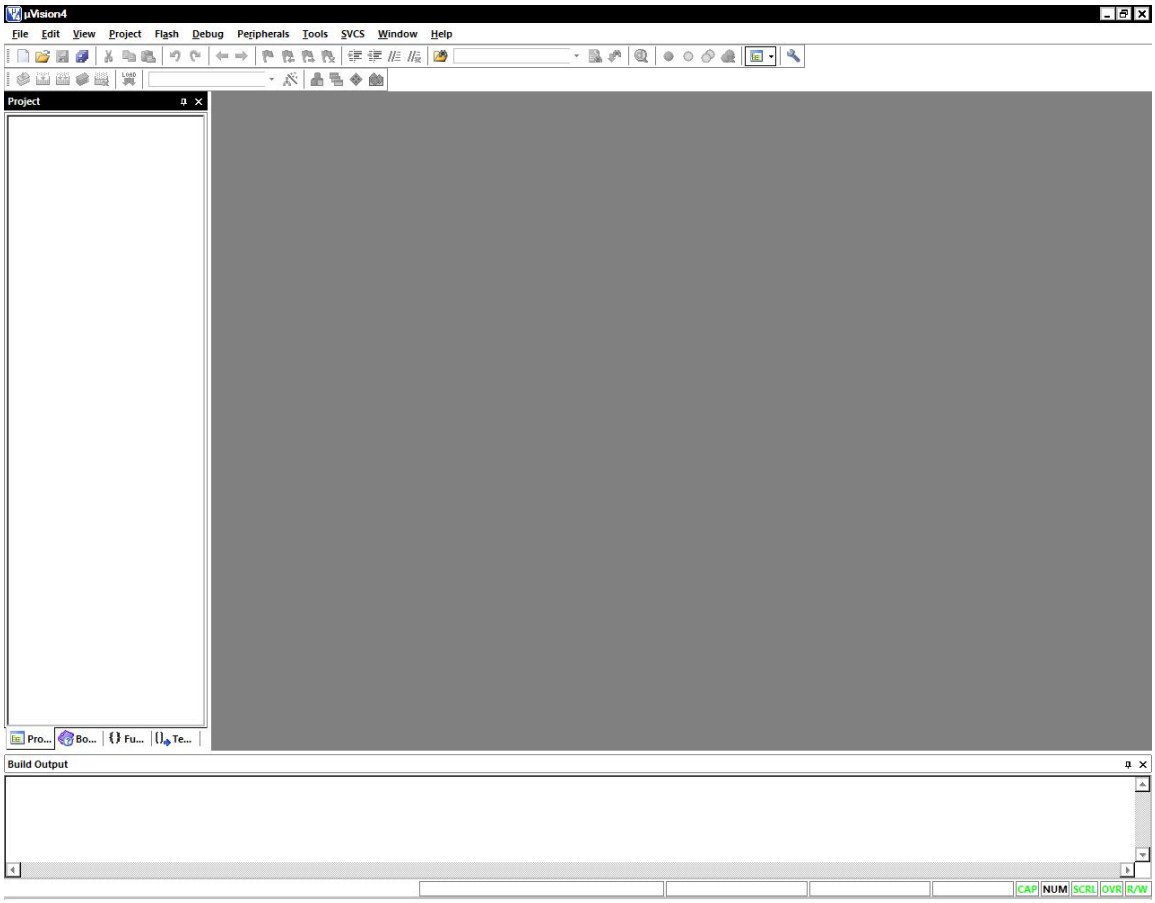


Рисунок 5.6 – Загальний вигляд головного вікна середовища

5.2.3 Створення нового проекту

Коли середовище успішно запущено, для створення нового проекту необхідно в головному меню обрати «Project → New µVision Project», після чого відкриється вікно, що зображено на рисунку 5.7.

`

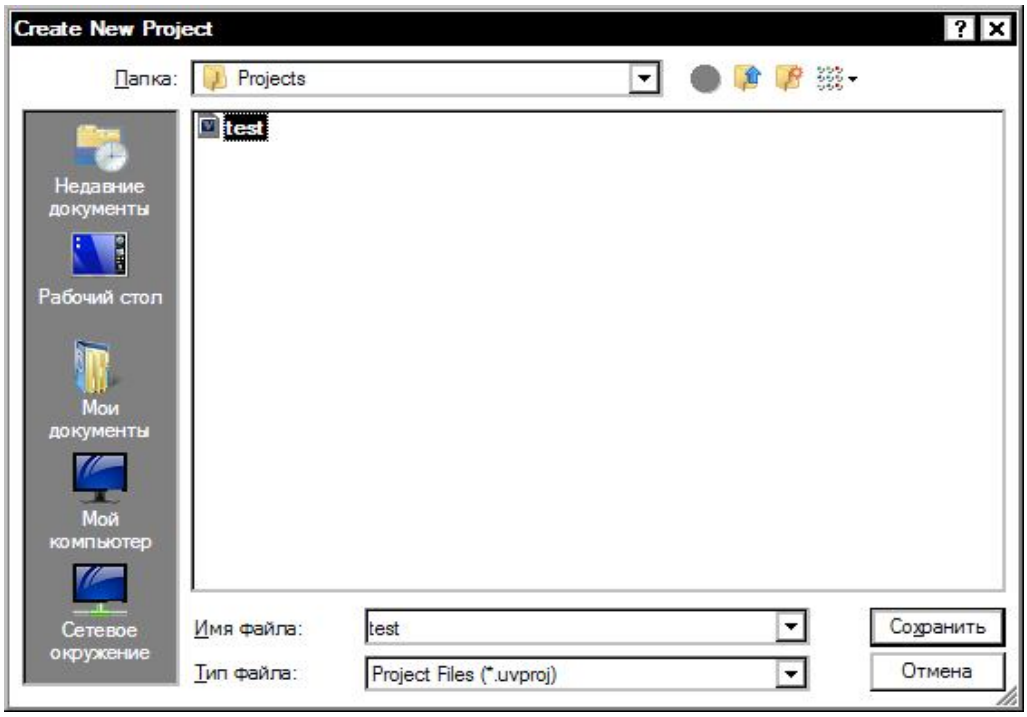


Рисунок 5.7 – Створення нового проекту

У заздалегідь створену папку (або за бажанням створюємо нову) створюємо проект, ввівши його назву, тиснемо «Сохранить» і переходимо до вікна вибору мікроконтролера (рисунок 5.8). Вибираємо ARM>ARM7(Little Endian) (прямий порядок байтів) або ARM>ARM7(Big Endian) (зворотний порядок) і натискаємо «ОК». Варто пам'ятати, що при написанні коду слід правильно задавати формат слів, інакше отриманий код буде «вивернутий навиворіт».

`

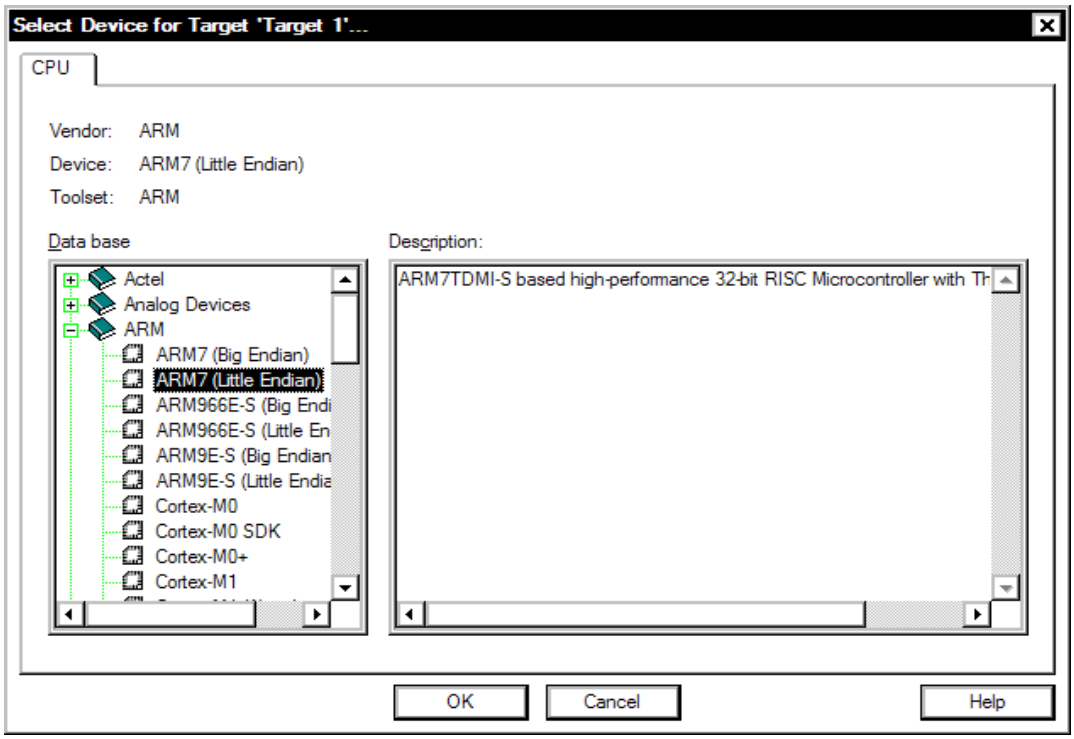


Рисунок 5.8 – Вибір мікроконтролера зі списку

5.2.3.1 Створення нового файлу

В головному меню налагоджувача натискаємо «File>New…» або комбінацією Ctrl+N створюємо новий текстовий файл. В цей файл пишемо або вставляємо скопійований код програми (рисунок 5.9). Зверніть увагу, що для правильної компіляції проекту в програмі слід писати рядки «AREA ProgramName, CODE, READWRITE», «ENTRY», «END». Для збереження файлу в головному меню «File>Save As…», з'явиться вікно, як на рисунку 5.10. Зберігаємо в форматі:

«Ім'я_файлу.s».

`

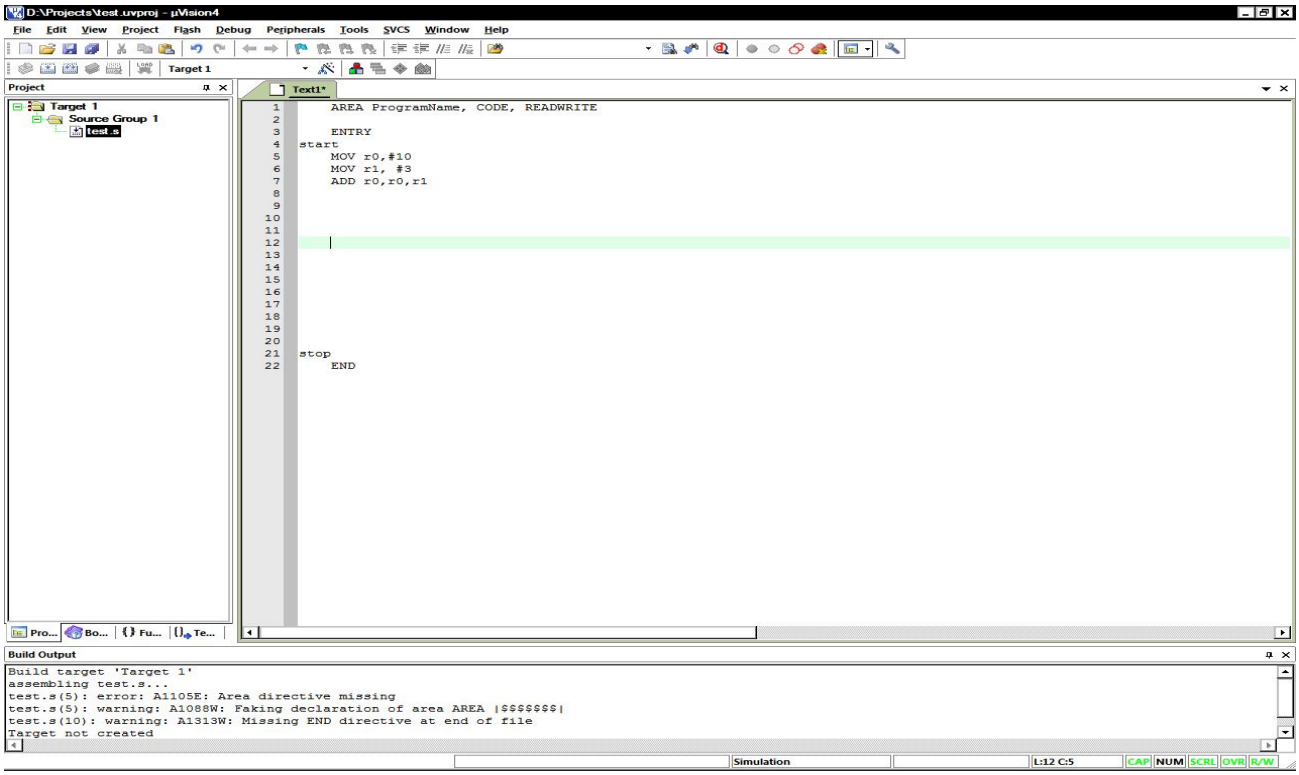


Рисунок 5.9 – Створення файлу та додавання коду

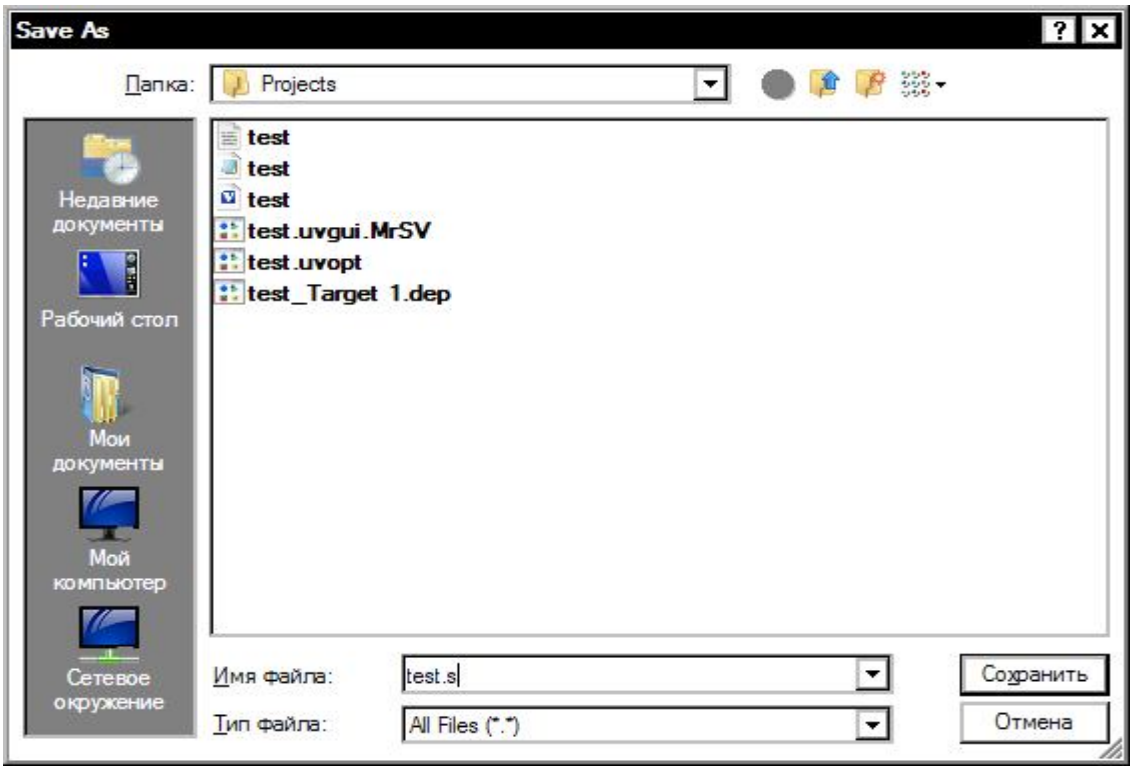


Рисунок 5.10 – Збереження файлу

`

5.2.3.2 Додавання файлів у проект

Для додавання файлів до проекту слід у вікні Projects (рисунок 5.9, вікно ліворуч) лівою клавішею натиснути: + та правою клавішею миші натиснути на Source Group 1 і вибрати Add Files to Group ‘Source Group 1’, відкриється вікно типу (рисунок 5.11):

Обираємо збережений раніше файл «Ім’я_файлу.s», тиснемо Add, Close. Файл додано.

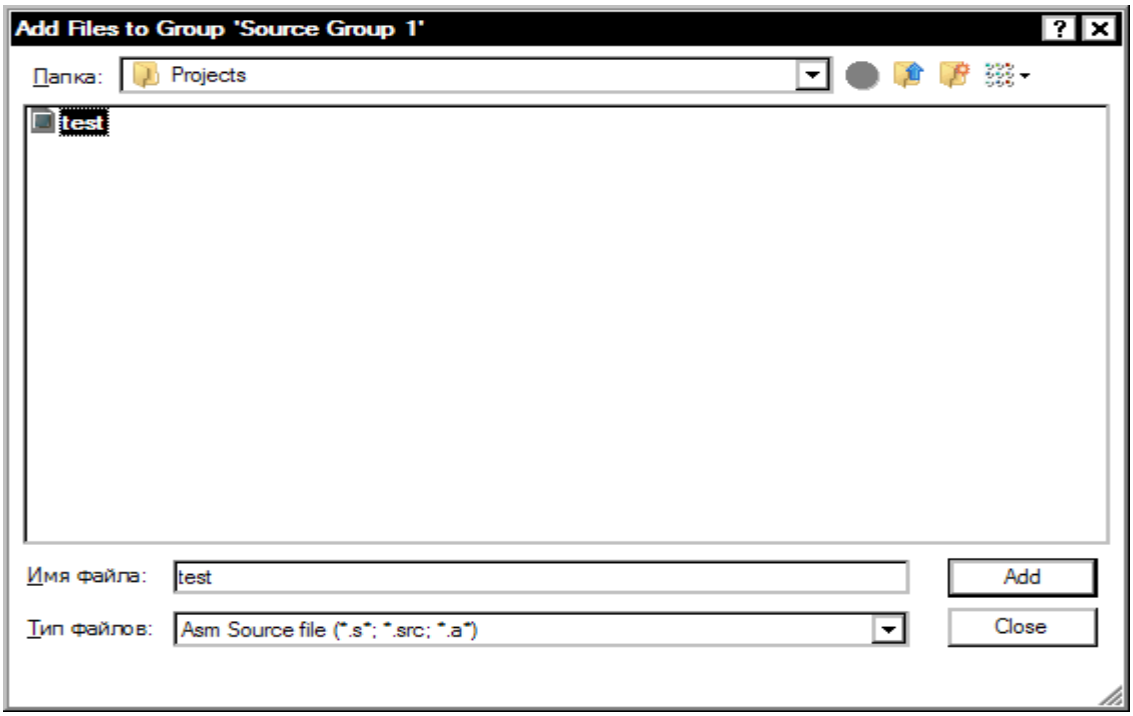


Рисунок 5.11 – Додавання файлу до групи

5.2.3.3 Компіляція проекту

Для компіляції проекту обираємо у головному меню «Project>Build Target» або натискаємо F7. У вікні Build Output можна побачити інформацію про перебіг компіляції. Якщо отримано повідомленяя: «0 Error(s), 0 Warning(s)», це свідчить про успішну компіляцію (рисунок 5.12).

`

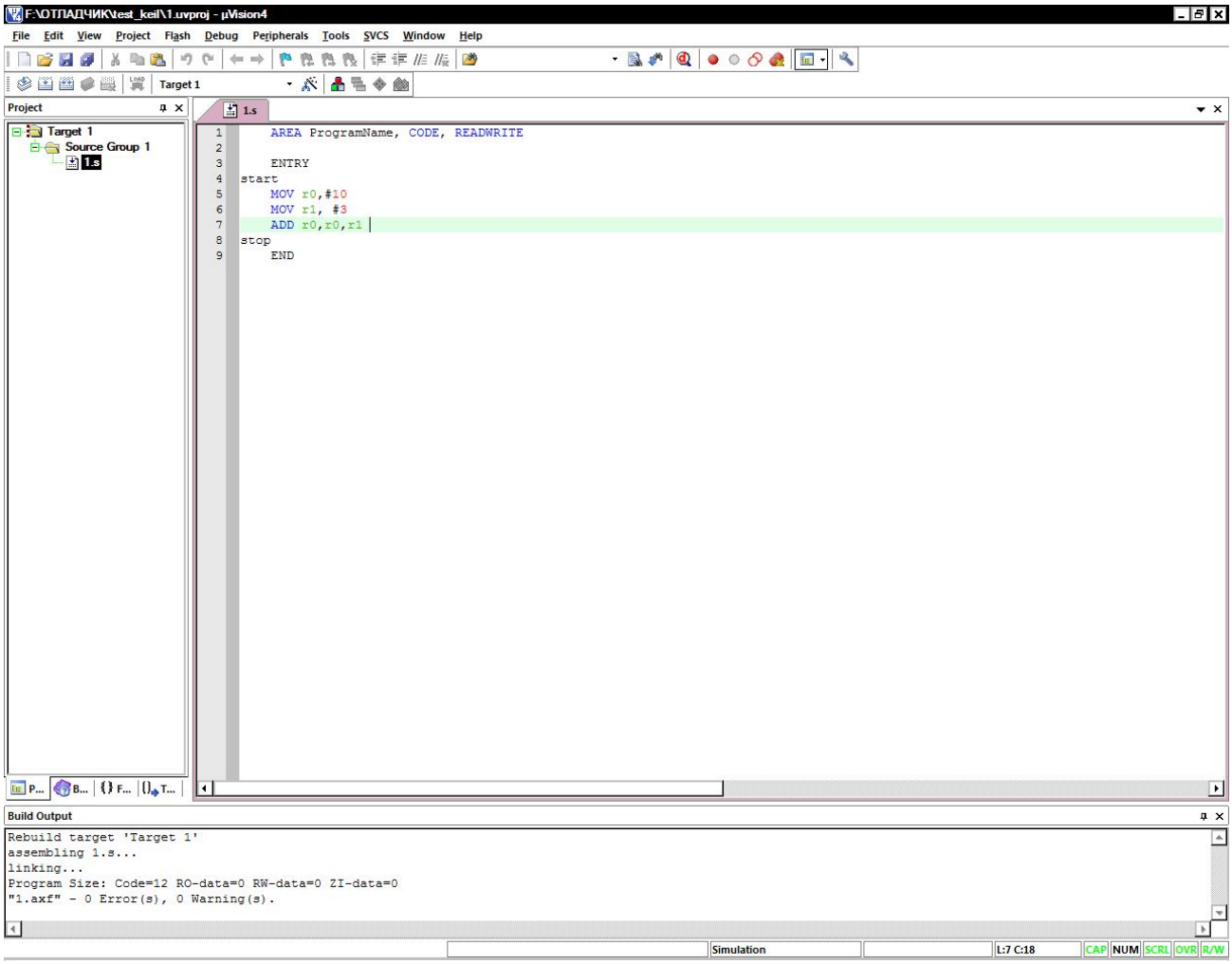


Рисунок 5.12 – Компіляція проекту

5.2.4 Налагодження проекту

Для налагодження проекту обираємо у головному меню «Debug>Start/Stop Debug Session» або натискаємо Ctrl+F5. З`являється попередження: EVALUATION MODE Running with Code Size Limit: 32 K (об`єм коду до 32 К). Тиснемо ОК. З`являється вікно програми, яке виглядатиме таким чином, як зображено на рисунку 5.13. Вікно регістрів розширюємо лівою кнопкою миші.

`

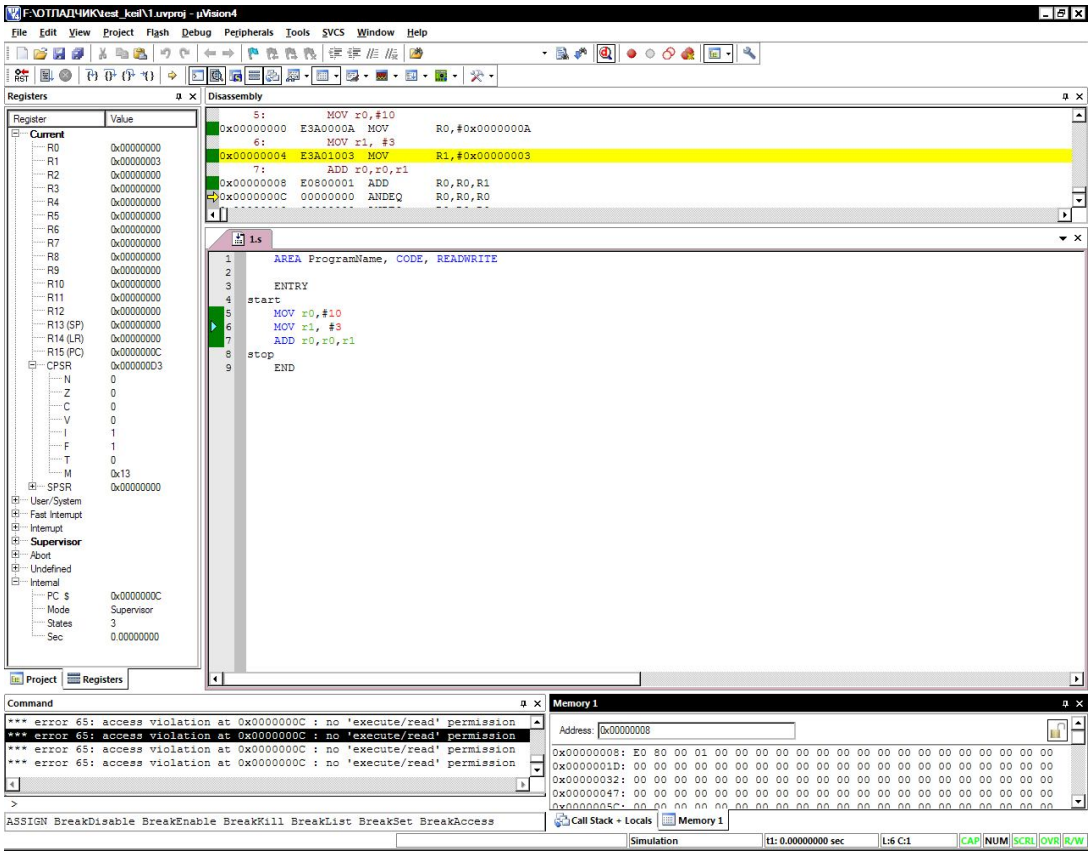


Рисунок 5.13 – Налагодження проекту

5.2.4.1 Покрокове виконання команд

Після запуску налагодження покрокове виконання команд забезпечується натисканням клавіші F10.

5.2.4.2 Зміна регістрів та прапорців

Вміст регістрів змінюємо подвійним кліком лівої клавіші миші на значенні потрібного регістра у вікні Registers. Для зміни прапорців слід розкрити CPSR і аналогічно до регістрів змінити значення потрібних прапорців.

5.2.4.3 Робота з пам’яттю

Задання діапазону адрес пам’яті, під час налагодження можна зробити у меню Debug>Memory Map.

`

Для зміни пам’яті в нижньому правому куті вікна слід обрати вкладку Memory 1, у полі Adress ввести необхідну адресу комірки. У полі значень навпроти введеної адреси (рисунок 5.14) вносимо потрібні зміни.

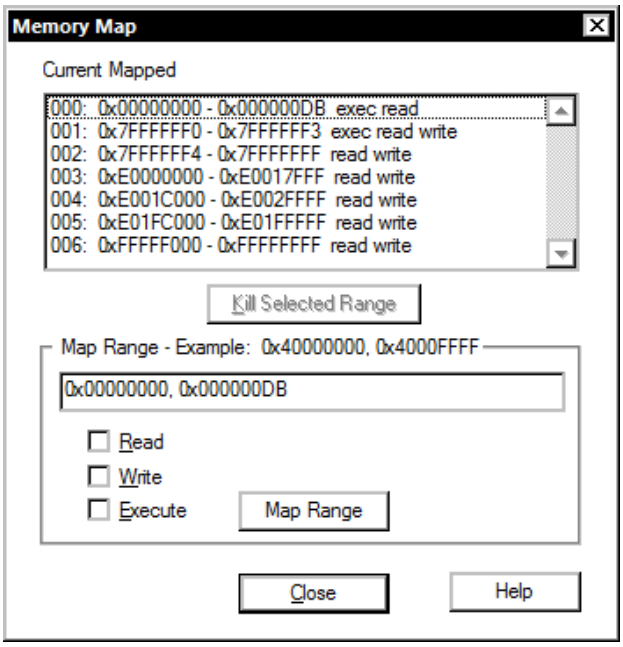


Рисунок 5.14 – Задання діапазону адрес пам’яті

5.2.4.4 Перегляд машинних кодів паралельно з мнемокодом

Для перегляду машинних кодів та мнемокоду достатньо перемкнутись на вікно Disassembly під час налагодження проекту (рисунок 5.13).

5.2.4.5 Зміни в коді програми

Для того, щоб внести зміни до коду слід зупинити налагодження, натискаючи у головному меню «Debug>Start/Stop Debug Session». Вікно повертається до вигляду як на рисунку 5.12. У файлі програми вносимо необхідні зміни або додаємо нові команди, зберігаємо внесені поправки та компілюємо проект: «Project>Build Target», або натискаємо F7.

`

5.2.5 Приклади окремих проектів для виконання у налагоджувачі

Нижче наведено три файли (проекти) для виконання окремих команд у налагоджувачі:

– data_processing.s;

– lab2_load_store.s;

– JMP_MUL.s.

5.2.5.1 Data_processing.s

```
AREA ProgramName, Code, ReadWrite

ENTRY

start
MOV R0, #0x01                ; R0 = 0x01
MOV R1, #0x03                ; R1 = 0x03
MOV R2, #0x08                ; R2 = 0x08
MOV R0,#0                    ; R0 = 0x00
MOVS R0,#0;                  ; R0 = 0x00, Z=1
MOVEQ R3,#0xFF               ; if (Z == 1) R3=0xFF
MOVNE R3,#0xAA               ; if (Z == 0) R3=0xAA

AND R3,R1,R0                 ; R3 = R1 AND R0
EOR R4,R2,R1                 ; R4 = R2 EOR R1
SUB R5,R2,R1                 ; R5 = R2 – R1
RSB R6,R2,R1                 ; R6 = R1 – R2
;
ADD R1,R1                  ; R1 = R1 + R1
ADD R7,R1,R0                 ; R7 = R1 + R0
```


`

```
        ADD R7,R1                       ; R7 = R7 + R1
        ADD R8,R1,R1, LSL #2    ; R8=R1+R1*(2^2)        R8=0x06+(0x06*4)=0x1E
        MOV R4,#2                       ; R4 = 0x02
        ADD R8,R1,R1, LSL R4    ; R8=R1+R1*(2^2)        R8=0x06+(0x06*4)=0x1E
        ADD R9,R1,R2, LSR #1    ; R9=R1+(R2 << 1)       R9=0x06+(0x08/2)=0x0A
        MOV R4,#1                       ; R4=0x01
        ADD R9,R1,R2, LSR R4    ; R9=R1+(R2 << 1)       R9=0x06+(0x08/2)=0x0A
        ADD R10,R1,R1, ROR #1   ; R10=R1+(R1 ROR 1)             R10=0x06+(0x06
ROR 1)==0x80000007
        ADD R10,R1,R1, ROR R4   ; R10=R1+(R1 ROR 1)             R10=0x06+(0x06
ROR 1)= 0x80000007
        ADD R11,R1,R1, RRX      ; R11=R1+(R1.0 –> C, R1 >> 1)
        ADDLO R11,R1,R1                 ; if (C==0) R11=0x06+0x06=0x0C
        MOV R3, #–5                     ; R3 <– –5
        ADD R12,R0,R3, ASR #1   ; 5 >> 1 = 2 ; –2+1=–1 => 0xFFFFFFFE
        ADDMI R12,R0,R3, ASR R4         ; if (N == 1) {5 >> 1 = 2 ; –2+1=–1 =>
0xFFFFFFFE }
        ;
        ADC R8,R1,R0                    ; R8 = R1 + R0+Carry
        SBC R9,R2,R1                    ; R9 = R2 – R1–NOT(Carry)
        RSC R10,R2,R1                   ; R10 = R1 – R2–NOT(Carry)
        TST R0, #0x09                   ; R0 AND R11, affects flags
        MOV R0,#6                       ; R0 = 0x06
        TEQ R0,#5                       ; R0 EOR 0x1, affects flags
        ;
        CMP R7, #101                    ; R7 – 101, affects flags
        ;
        MOV R7,#8                       ; R7 = 0x08
        MOV R8,#8                       ; R8 = 0x08
        CMN R8, R7                      ; R8 + R7, N–>0,Z–>0
```


`

```
;
MOV R0,#7                       ; R0 = 0x07
MOV R1,#8                       ; R1 = 0x08
CMN R0, R1                      ; R0 + R1, N–>0,Z–>0
;
MOV R0,#–8                      ; R0 = 0xFFFFFFF8
MOV R1,#–8                      ; R1 = 0xFFFFFFF8
CMN R0, R1                      ; R0 + R1, N–>1,Z–>0,C–>1
;
MOV R0,#–8                      ; R0 = 0xFFFFFFF8
MOV R1,#–8                      ; R1 = 0xFFFFFFF8
CMP R0, R1                      ; R0 – R1, N–>0,Z–>1,C–>1
;
MOV R0,#–5                      ; R0 = 0xFFFFFFFB
MOV R1,#–8                      ; R1 = 0xFFFFFFF8
CMP R0, R1                      ; R0 – R1, N–>0,Z–>0,C–>1
;
MOV R0,#–8                      ; R0 = 0xFFFFFFF8
MOV R1,#–5                      ; R1 = 0xFFFFFFFB
CMP R0, R1                      ; R0 – R1, N–>1,Z–>0,C–>0
;
ORR R12, R5, R2                 ; R12 = R5 OR R2
BIC R13, R6, #0xFF00        ; R13 = R6 AND NOT 0xFF00
MVN R14, R6                             ; R14 = NOT R6
stop
END
```

5.2.5.2 Lab2_load_store.s

```
AREA ProgramName, Code, ReadWrite
```


`

```
ENTRY

start

;;; load commands ;;;

MOV R0, #0x01;       R0 = 0x01
MOV R1, #0x03;       R1 = 0x03
MOV R2, #0xFFFFFF3C;     R2 = 0xFFFFFF3C
MOV R4, #0x05; R4 = 0x05
MOV R5, #0xFFFFFF5C; R5 = 0xFFFFFF5C
MOV R6, #0xFFFFFF01; R6 = 0xFFFFFF01
MOV R7, #0x00000007; R7 = 0x00000007
MOV R8, #0xFFFFFF30; R8 = 0xFFFFFF30

LDR R3, [R2] ; R2 = 0xFFFFFF3C; M(R2)=12 34 56 78; R3 = 0x78563412
LDRH R3, [R2]; R2 = 0xFFFFFF3C; M(R2)=12 34 56 78; R3 = 0x00003412, where R3[31:16] := 0x0000, R3[15:0] := 0x3412
LDRB R3, [R2]; R2 = 0xFFFFFF3C; M(R2)=12 34 56 78; R3 = 0x00000012, where R3[31:8] := 0x000000, R3[7:0] := 0x12
LDRT R3, [R2], #44; R2 = 0xFFFFFF3C; M(R2)=12 34 56 78; R3 = 0x78563412; R2 := ((R2)2+44)16 = 0xFFFFFF68
LDRT R3, [R2], #–44; R2 = 0xFFFFFF68; M(R2)=23 45 67 89; R3 = 0x89674523; R2: = ((R2)2–44)16 = 0xFFFFFF3C
LDRT R3, [R2], R1; R2 = 0xFFFFFF3C; M(R2)=12 34 56 78; R3 = 0x78563412; R2 := R2+R1=FFFFFF3C+00000003 = 0xFFFFFF3F

MOV R2, #0xFFFFFF3C;     R2 = 0xFFFFFF3C
LDRT R3, [R2], –R1; R2 = 0xFFFFFF3C;  M(R2)=78 90 12 34; R3 = 0x34129078; R2 := R2–R1=FFFFFF3C–00000003 = 0xFFFFFF39
SUB R2, R2, R0; R2 = 0xFFFFFF39–0x01 = 0xFFFFFF38, stands for address align-
ment
```


`

LDRT R3,[R2], R4, LSL #2 ; R2 = 0xFFFFFF38; M(R2) = 78 90 12 34; R3 = 0x34129078; R2 = R2+ (R4 lsl #2) = 0xFFFFFF4C

LDRBT R3, [R2], #44; R2 = 0xFFFFFF4C; M(R2)=12 34 56 78; R3 = 0x00000012; R2 := ((R2)2+44)16 = 0xFFFFFF78

LDRBT R3, [R2], #–44; R2 = 0xFFFFFF78; M(R2)=12 34 56 78; R3 = 0x00000012; R2: = ((R2)2–44)16 = 0xFFFFFF4C

LDRBT R3, [R2], R1; R2 = 0xFFFFFF4C; M(R2)=12 34 56 78; R3 = 0x00000012; R2 := R2+R1=FFFFFF4C+00000003 = 0xFFFFFF4F

ADD R2, R2, R0; R2 = 0xFFFFFF4F+0x01 = 0xFFFFFF50, stands for address alignment

LDRBT R3,[R2], R4, LSL #2 ; R2 = 0xFFFFFF50; M(R2) = 12 34 56 78; R3 = 0x00000012; R2 = R2+ (R4 lsl #2) = 0xFFFFFF64

LDRSB R3, [R5]; R5 = 0xFFFFFF5C; M(R5)=11 12 13 14; R3 = 0x00000011; S := 0, where R3[31:8] := 0xSSSSSS, R3[7:0] := 0x11

LDRSB R9, [R8]; R8 = 0xFFFFFF30; M(R8)= E7 FF FF FF; R8[7:0]= E7; S := F, R9 = 0xFFFFFFE7, where R9[31:8] := 0xSSSSSS, R9[7:0] := 0xE7

LDRSH R3, [R5]; R5 = 0xFFFFFF5C; M(R5)=11 12 13 14; R3 = 0x00001211; S := 0, where R3[31:16] := 0xSSSS, R3[15:0] := 0x1211

LDRSH R10, [R8]; R8 = 0xFFFFFF30; M(R8)= E7 FF FF FF; R8[7:0]= E7; S := F; R10 = 0xFFFFFFE7; where R10[31:8] := 0xSSSS, R10[15:0] := 0xFFE7

LDR R3, [R2, #0x04]; R2=0xFFFFFF64; R2+0x04 = 0xFFFFFF68; M(R2+0x04) = 11 12 13 14; R3 = 0x14131211

LDR R3, [R5, #–0xA4]; R5 = FFFFFF5C; R5–0xA4=0xFFFFFEB8; M(R5–0xA4) = 11 12 13 14; R3 = 0x14131211

LDR R3, [R0, #0xFFFFFF37]!; R0=0x00000001; R0+0xFFFFFF37 = 0xFFFFFF38; M(R0+0xFFFFFF37) = 11 12 13 14; R3 = 0x14131211; R0 = 0xFFFFFF38

LDR R3, [R5, #–0x30]!; R5=0xFFFFFF5C; R5–0x30 = 0xFFFFFF2C; M(R5–0x30) = 11 12 13 14; R3 = 14 13 12 11; R5 = 0xFFFFFF2C

LDR R3,[R5,–R2]; R5=0xFFFFFF2C; R5–R2= 0xFFFFFF2C – 0xFFFFFF64 = 0xFFFFFFC8; M(R5–R2)=23 45 67 89; R3 = 0x89674523

LDR R3,[R6,R7]; R6 = 0xFFFFFF01; R6+R7 = 0xFFFFFF08; M(R6+R7) = 11 12 13 14; R3 = 0x14131211

`

LDR R3,[R5, –R2]!; R5=0xFFFFFF2C; R5–R2= 0xFFFFFF2C – 0xFFFFFF64 = 0xFFFFFFC8; M(R5–R2)=23 45 67 89; R3 = 0x89674523; R5 = 0xFFFFFFC8

LDR R3,[R6,R7]!; R6 = 0xFFFFFF01; R6+R7 = 0xFFFFFF08; M(R6+R7) = 11 12 13 14; R3 = 0x14131211; R6 = 0xFFFFFF08

MOV R2, #0xFFFFFF63; R2 = 0xFFFFFF63, stands for address alignment

LDR R3,[R2, –R5, lsr #25]; R5 = 0xFFFFFFC8; R5 lsr #25 = 0x0000007F; R2 – R5 lsr #25 = 0xFFFFFEE4; M(R2 – R5 lsr #25) = 11 12 13 14; R3 = 0x14131211

SUB R0,R0,R1; R0 = R0–R1 = 0xFFFFFF38–0x03=0xFFFFFF35, stands for address alignment

LDR R3,[R0, R5, lsr #26]; R5 = 0xFFFFFF35; R5 lsr #26 = 0x0000003F; R0 + R5 lsr #26 = 0xFFFFFF74; M(R0 + R5 lsr #26) = 11 12 13 14; R3 = 0x14131211

LDR R3,[R2, –R5, lsr #25]!; R5 = 0xFFFFFFC8; R5 lsr #25 = 0x0000007F; R2 – R5 lsr #25 = 0xFFFFFEE4; M(R2 – R5 lsr #25) = 11 12 13 14; R3 = 0x14131211; R2 = 0xFFFFFEE4

LDR R3,[R0, R5, lsr #26]!; R5 = 0xFFFFFF35; R5 lsr #26 = 0x0000003F; R0 + R5 lsr #26 = 0xFFFFFF74; M(R0 + R5 lsr #26) = 11 12 13 14; R3 = 0x14131211; R0 = 0xFFFFFF74

LDR R3,[R2], #44; R2 = 0xFFFFFEE4; M(R2)= 11 12 13 14; R3 = 0x14131211; R2 = 0xFFFFFEE4+44 = 0xFFFFFF10

LDR R3,[R5], #–44; R5 = 0xFFFFFF74; M(R5) = 11 12 13 14; R3 = 0x14131211; R5 = 0xFFFFFF74–44 = 0xFFFFFF9C

LDR R3,[R0], –R1; R3 = 0xFFFFFF74; M(R0) = 11 12 13 14; R3 = 0x14131211; R0 = 0xFFFFFF74–0x03 = 0xFFFFFF71

LDR R3,[R6], R0; R6 = 0xFFFFFF08; M(R6) = 11 12 13 14; R3 = 0x14131211; R6 = 0xFFFFFF08+0xFFFFFF71 = 0xFFFFFE79

ADD R2, R2, #0x04; R2 = 0xFFFFFF10+0x04 = 0xFFFFFF14, stands for address alignment

`

LDR R3,[R2], R4, ror #30; R2 = 0xFFFFFF14; R4 ror #30 = 0x00000014; R2+R4 ror #30 = 0xFFFFFF14+0x00000014 = 0xFFFFFF28; M(R2)= 11 12 13 14; R3 = 0x14131211

LDR R3,[R2], –R4, ror #30;R2 = 0xFFFFFF28; R4 ror #30 = 0x00000014; R2–R4 ror #30 = 0xFFFFFF14–0x00000014 = 0xFFFFFF14; M(R2)= 11 12 13 14; R3 = 0x14131211

;;; store commands ;;;

MOV R0, #0x01; R0 = 0x01

MOV R1, #0x03; R1 = 0x03

MOV R2, #0xFFFFFF3C; R2 = 0xFFFFFF3C

MOV R3, #0xFFFFFF12; R3 = 0xFFFFFF12

MOV R4, #0x05; R4 = 0x05

MOV R5, #0xFFFFFF5C; R5 = 0xFFFFFF5C

MOV R6, #0xFFFFFF01; R6 = 0xFFFFFF01

MOV R7, #0x00000007; R7 = 0x00000007

STR R3, [R2] ; R2 = 0xFFFFFF3C; R3 = 0xFFFFFF12; M(R2)= 12 FF FF FF

STRH R3, [R3]; R3 = 0xFFFFFF12; M(R3)=12 FF 00 99; where R3[31:16] := 0x0000, R3[15:0] := 0x12FF

STR R9,[R3]; clear M(R3)

STR R9, [R2]; clear M(R2)

STRB R3, [R2]; R3 = 0xFFFFFF12, R2 = 0xFFFFFF3C; M(R2)=12 00 00 00; where R3[31:8] := 0x000000, R3[7:0] := 0x12

STRT R3, [R2], #44; R3 = 0xFFFFFF12; R2 = 0xFFFFFF3C; M(R2)=12 FF FF FF; R2 := ((R2)2+44)16 = 0xFFFFFF68

STRT R3, [R2], #–44; R3 = 0xFFFFFF12; R2 = 0xFFFFFF68; M(R2)=12 FF FF FF; R2: = ((R2)2–44)16 = 0xFFFFFF3C

STRT R3, [R2], R1; R3 = 0xFFFFFF12; R2 = 0xFFFFFF3C; M(R2)=12 FF FF FF; R2 := R2+R1=FFFFFF3C+00000003 = 0xFFFFFF3F

`

MOV R2, #0xFFFFFF3C; R2 = 0xFFFFFF3C, stands for address alignment

STRT R3, [R2], –R1; R3 = 0xFFFFFF12; R2 = 0xFFFFFF3C; M(R2) = 12 FF FF FF; R2 := R2–R1=FFFFFF3C–00000003 = 0xFFFFFF39

SUB R2, R2, R0; R2 = 0xFFFFFF39–0x01 = 0xFFFFFF38, stands for address alignment

STRT R3,[R2], R4, LSL #2 ; R2 = 0xFFFFFF38; R3 = 0xFFFFFF12; M(R2) = 12 FF FF FF; R2 = R2+ (R4 lsl #2) = 0xFFFFFF4C

STRBT R3, [R2], #44; R2 = 0xFFFFFF4C; R3 = 0xFFFFFF12; M(R2)= 12 00 00 00; R2 := ((R2)2+44)16 = 0xFFFFFF78

STRBT R3, [R2], #–44; R2 = 0xFFFFFF78; R3 = 0xFFFFFF12; M(R2)=12 00 00 00; R2: = ((R2)2–44)16 = 0xFFFFFF4C

STRBT R6, [R2], R1; R2 = 0xFFFFFF4C; R6 = 0xFFFFFF01; M(R2)=01 00 00 00; R2 := R2+R1=FFFFFF4C+00000003 = 0xFFFFFF4F

ADD R2, R2, R0; R2 = 0xFFFFFF4F+0x01 = 0xFFFFFF50, stands for address alignment

STRBT R3,[R2], R4, LSL #2 ; R2 = 0xFFFFFF50; R3 = 0xFFFFFF12; M(R2) = 12 00 00 00; R2 = R2+ (R4 lsl #2) = 0xFFFFFF64

STR R3, [R2, #0x04]; R2=0xFFFFFF64; R3 = 0xFFFFFF12; R2+0x04 = 0xFFFFFF68; M(R2+0x04) = 12 FF FF FF

STR R3, [R5, #–0xA4]; R5 = FFFFFF5C; R3 = 0xFFFFFF12; R5–0xA4=0xFFFFFEB8; M(R5–0xA4) = 12 FF FF FF

STR R6, [R0, #0xFFFFFF37]!; R0=0x00000001; R0+0xFFFFFF37 = 0xFFFFFF38; R6 = 0xFFFFFF01; M(R0+0xFFFFFF37) = 01 FF FF FF; R0 = 0xFFFFFF38

STR R3, [R5, #–0x30]!; R5=0xFFFFFF5C; R5–0x30 = 0xFFFFFF2C; R3 = 0xFFFFFF12; M(R5–0x30) = 12 FF FF FF; R5 = 0xFFFFFF2C

STR R3,[R5,–R2]; R5=0xFFFFFF2C; R5–R2= 0xFFFFFF2C – 0xFFFFFF64 = 0xFFFFFFC8; R3 = 0xFFFFFF12; M(R5–R2)= 12 FF FF FF

STR R3,[R6,R7]; R6 = 0xFFFFFF01; R6+R7 = 0xFFFFFF08; R3 = 0xFFFFFF12; M(R6+R7) = 12 FF FF FF

`

STR R6,[R5, –R2]!; R5=0xFFFFFF2C; R5–R2= 0xFFFFFF2C – 0xFFFFFF64 = 0xFFFFFFC8; R6 = 0xFFFFFF01; M(R5–R2)= 01 FF FF FF; R5 = 0xFFFFFFC8

STR R0,[R6,R7]!; R6 = 0xFFFFFF01; R6+R7 = 0xFFFFFF08; R0 = 0xFFFFFF38; M(R6+R7) = 38 FF FF FF; R6 = 0xFFFFFF08

MOV R2, #0xFFFFFF63; R2 = 0xFFFFFF63, stands for address alignment

STR R3,[R2, –R5, lsr #25]; R5 = 0xFFFFFFC8; R5 lsr #25 = 0x0000007F; R2 – R5 lsr #25 = 0xFFFFFEE4; R3 = 0xFFFFFF12; M(R2 – R5 lsr #25) = 12 FF FF FF

SUB R0,R0,R1; R0 = R0–R1 = 0xFFFFFF38–0x03=0xFFFFFF35, stands for address alignment

STR R3,[R0, R5, lsr #26]; R5 = 0xFFFFFF35; R5 lsr #26 = 0x0000003F; R0 + R5 lsr #26 = 0xFFFFFF74; R3 = 0xFFFFFF12; M(R0 + R5 lsr #26) = 12 FF FF FF

STR R6,[R2, –R5, lsr #25]!; R5 = 0xFFFFFFC8; R5 lsr #25 = 0x0000007F; R2 – R5 lsr #25 = 0xFFFFFEE4; R6 = 0xFFFFFF08; M(R2 – R5 lsr #25) = 08 FF FF FF; R2 = 0xFFFFFEE4

STR R6,[R0, R5, lsr #26]!; R5 = 0xFFFFFF35; R5 lsr #26 = 0x0000003F; R0 + R5 lsr #26 = 0xFFFFFF74; R6 = 0xFFFFFF08; M(R0 + R5 lsr #26) = 08 FF FF FF; R0 = 0xFFFFFF74

STR R3,[R2], #44; R2 = 0xFFFFFEE4; R3 = 0xFFFFFF12; M(R2)= 12 FF FF FF; R2 = 0xFFFFFEE4+44 = 0xFFFFFF10

STR R3,[R5], #–44; R5 = 0xFFFFFFC8; R3 = 0xFFFFFF12; M(R5) = 12 FF FF FF; R5 = 0xFFFFFFC8–44 = 0xFFFFFF9C

STR R3,[R0], –R1; R0 = 0xFFFFFF74; R6 = 0xFFFFFF12; M(R0) = 12 FF FF FF; R0 = 0xFFFFFF74–0x03 = 0xFFFFFF71

STR R5,[R6], R0; R6 = 0xFFFFFF08; R5 = 0xFFFFFF9C; M(R6) = 9C FF FF FF; R6 = 0xFFFFFF08+0xFFFFFF71 = 0xFFFFFE79

ADD R2, R2, #0x04; R2 = 0xFFFFFF10+0x04 = 0xFFFFFF14, stands for address alignment

STR R3,[R2], R4, ror #30; R2 = 0xFFFFFF14; R4 ror #30 = 0x00000014; R2+R4 ror #30 = 0xFFFFFF14+0x00000014 = 0xFFFFFF28; R3 = 0xFFFFFF12; M(R2)= 12 FF FF FF

`

```
        STR R3,[R2], –R4, ror #30;R2 = 0xFFFFFF28; R4 ror #30 = 0x00000014; R2–R4 ror
#30 = 0xFFFFFF14–0x00000014 = 0xFFFFFF14; R3 = 0xFFFFFF12; M(R2)= 12 FF FF FF


        ;;;; MULTIPLE REGISTERS


        MOV R8, #0xFFFFFF3C                 ; R8 = 0xFFFFFF3C

        STR R8,[R8]                         ; M(R8) = 0xFFFFFF3C

        STR R8,[R8,#4]                      ; M(R8+4) = 0xFFFFFF3C

        STR R8,[R8,#8]                      ; M(R8+8) = 0xFFFFFF3C

        LDM   R8,{R0,R2,R9}                 ; R0=M(R8)=0xFFFFFF3C,
R1=M(R8+4)=0xFFFFFF3C, R2=M(R8+8)=0xFFFFFF3C

        LDMIA R8!,{R0,R1,R2}                ; R0=M(R8)=0xFFFFFF3C,
R1=M(R8+4)=0xFFFFFF3C, R2=M(R8+8)=0xFFFFFF3C

        LDMIB R8!,{R0,R1,R2}                ; R0=M(R8)=0xFFFFFF3C,
R1=M(R8+4)=0xFFFFFF3C, R2=M(R8+8)=0x00000000

        LDMDA R8!,{R0,R1,R2}                ; R0=M(R8)=0x00000000,
R1=M(R8+4)=0x00000000, R2=M(R8+8)=0xFFFFFF3C

        LDMDB R8!,{R0,R1,R2}                ; R0=M(R8)=0x00000000,
R1=M(R8+4)=0x00000000, R2=M(R8+8)=0x00000000

        ;

        MOV R0,#0xFFFFFFAA                  ; R0 = 0xFFFFFFAA

        MOV R1,#0xFFFFFFAA                  ; R1 = 0xFFFFFFAA

        MOV R2,#0xFFFFFFAA                  ; R2 = 0xFFFFFFAA

        STMIA R8!,{R0,R1,R2}                ; M(R8)=0xFFFFFFAA,
M(R8+4)=0xFFFFFFAA, M(R8+8)=0xFFFFFFAA

        STMIB R8!,{R0,R1,R2}                ; M(R8+4)=0xFFFFFFAA,
M(R8+8)=0xFFFFFFAA, M(R8+12)=0xFFFFFFAA

        STMDA R8!,{R0,R1,R2}                ; M(R8)=0xFFFFFFAA, M(R8–
4)=0xFFFFFFAA, M(R8–8)=0xFFFFFFAA

        STMDB R8!,{R0,R1,R2}                ; M(R8–4)=0xFFFFFFAA, M(R8–
8)=0xFFFFFFAA, M(R8–12)=0xFFFFFFAA


        stop
```


`

```
    END
```

5.2.5.3 JMP_MUL.s

```
    AREA ProgramName, Code, ReadWrite

    ENTRY

start

    MOV R0, #0x01;                  R0 = 0x01
    MOV R2, #0xFFFFFFCE;    R2 = 0xFFFFFFCE
    MOV   R3, #0x04;            R3 = 0x04
    MOV R4, #0x05;              R4 = 0x05
    MOV R5, #0x06;              R5 = 0x06
    MOV R6, #0xFFFFFFCD;    R6 = 0xFFFFFFCD
    MOV R7, #0x08;                  R7 = 0x08
    MOV R8, #0x09;                  R8 = 0x09
    MOV R9, #0x10;                  R9 = 0x010

    B label                         ; Go to "label"
    MOV R10,#–5
label
    BL label2                       ; R14 =0x00000030 Go to "label2"
    MOV R10,#–7
label2
    MOV R10,#0x40
    BX R10

    MUL R4, R3, R5              ; R4 = R3 x R5 = 0x18
```


`

```
    MULS R4, R2, R0      ; R4 = R2 x R0 = FFFFFFCE, N–>1

    MULS R4, R2, R1      ; R4 = R2 x R1 = 0, Z–>1

    MLA R4, R7, R8, R3           ; R4 = R7 x R8 + R3 = 0x4C

    UMULL R7, R8, R2, R6 ; R7, R8 = R2 x R6

    MOV R5, #0x05                ; R5 = 0x05

    UMLAL R5, R9, R2, R6 ; R2 x R6 = FFFFFF9B 000009F6, R5 = 0x9F6 + R5 =
0x9FB, R9 = FFFFFF9B + R9 = FFFFFFAB

    SMLAL R0, R3, R2, R7 ; R2 x R7 = FFFFFFFFF FFFE0DF4, R0 = R0 +
0xFFFE0DF4 = 0xFFFE0DF5, R3 = R3 + 0xFFFFFFFF = 0x03


stop

    END
```


`

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1) Поясніть синтаксис команд:

– пересилання;

– арифметичних;

– логічних;

– завантаження/збереження;

– групового завантаження/збереження;

– обміну;

– роботи зі співпроцесорами;

– програмного переривання.

2) Які дії необхідно виконати щоб створити новий проект у середовищі μVision Keil 4?

3) Як додати файли по проекту?

4) Яку комбінацію клавіш потрібно натиснути для компілювання проекту?

5) Як виконати перегляд машинних кодів та мнемокоду?

6) Як задається діапазон адрес памяті?

7) Як виконати покрокове виконання команд?

8) Як виконати зміну регістрів та прапорців?

`

ПРЕДМЕТНИЙ ПОКАЖЧИК

`

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Тревор Мартин Микроконтроллеры ARM7 семейств LPC2300/2400. Вводный курс разработчика / пер. с англ. Евстифеева А. В. – М. : Додэка-XXI, 2010. – 336 с.: ил.

2. Редькин П.П. 32/16-битные микроконтроллеры ARM7 семейства AT91SAM7 фирмы Atmel. Руководство пользователя (+ CD). – М. : Издательский дом «Додэка-XXI», 2008. – 704 с.: ил.

3. ARM® IAR Embedded Workbench™ IDE. User Guide for Advanced RISC Machines Ltd's ARM Cores. 2005, IAR Systems.

4. ARM® IAR C/C++ Compiler. Reference Guide for Advanced RISC Machines Ltd's ARM Cores. 2005, IAR Systems.

5. IAR Linker and Library Tools. Reference Guide. Version 4.59. 2005, IAR Systems.

6. Редькин П.П. Микроконтроллеры ARM7 семейства LPC2000. Руководство пользователя (+ CD). – М. : Издательский дом «Додэка-XXI», 2007.

7. Мартин Т. Микроконтроллеры ARM7. Семейство LPC2000 компании Philips. Вводный курс (+ CD).- М.: Издательский дом «Додэка-XXI», 2006.

8. Отладочная плата AS-sam7X. Руководство пользователя. ARGUSSOFT. www.argussoft.ru.

9. Отладочная плата AS-sam7S64. Руководство пользователя. ARGUSSOFT. www.argussoft.ru.

10. ARM-JTAG Wiggler COMPATIBLE DONGLE FOR PROGRAMMING AND DEBUGGING. 2004, OLIMEX Ltd., www.olimex.com/dev.

11. Microchip 24AA64/24LC64 64K I2C ™ CMOS Serial EEPROM. 1999 Microchip Technology Inc. DS21189C.

`

12. ARM 7TDMI Data Sheet. Document Number: ARM DDI 0029E. Issued: August 1995. Advanced RISC Machines Ltd (ARM) 1995.

13. ARM7TDMI-S Technical Reference Manual (Rev 4) ARM Limited. ARM DDI 0234A.

14. AT91 ARM Thumb-based Microcontrollers AT91SAM7X256/AT91SAM7X128. Preliminary. 6120E-ATARM-04-Apr-06.

`