Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

АРХІТЕКТУРА КОМП'ЮТЕРНИХ СИСТЕМ: МОВА АСЕМБЛЕРА

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

Укладачі: Л.Ю, Гальчинський, О. В. Козленко

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент Тимошенко Ю.О., к.т.н., доцент, ІПСА НТУУ «КПІ» ім. Ігоря Сікорського

Відповідальний редактор Смірнов С.А, к.ф-м.н, с.н.с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 2 від 30.09.2022 р.)
за поданням Вченої ради Навчально-наукового фізико-технічного інституту
(*протокол № 11від 01.09.2022 р.)*

В навчальному посібнику «АРХІТЕКТУРА КОМП'ЮТЕРНИХ СИСТЕМ: МОВА АСЕМБЛЕРА» викладено основи архітектури сучасних мікропроцесорів двох основних напрямків: ті, що базуються на ідеї набору складних команд(CISC) та тих, хто базуються на набори спрощених команд(RISC). Посібник містить детальний опис архітектур сучасних мікропроцесорів, на базі архітектур x86 та ARM, які відповідно є реалізаціями архітектур CISC та RISC. Викладені особливості систем команд та їх виконання шляхом використання регістрів процесорів. Приділено увагу особливостям взаємодії процесорів з пам'яттю. Паралельно показано можливості програмування мовою асемблера з врахування тісного взаємозв'язку конструкцій мови з архітектурою процесора та особливостями операційних систем, зокрема для організації системних викликів. Для засвоєння практичних навичків здобувачами наведені варіанти використання декількох асемблерних середовищ разом з прикладами програм в них. Запропоновані контрольні запитання для поглиблення засвоєння матеріалу навчального посібника.

Навчальний посібник «АРХІТЕКТУРА КОМП'ЮТЕРНИХ СИСТЕМ: МОВА АСЕМБЛЕРА» призначений для здобувачів ступеня бакалавр за спеціальністю 125 «Кібербезпека», буде також корисним для здобувачів галузі знань Інформаційні технології та спеціальності 113 «Прикладна математика».

Реєстр. № НП 22/23-114. Обсяг 6,5 авт. арк.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

ВСТУП

Мова ассемблера, не є винаходом ери персональних обчислень та Інтернету. Вона з'явилася як засіб полегшити каторжну роботу перших програмістів, які писали код в машинних командах і абсолютних адресах для комп'ютерів першого покоління ще у 40-х роках минулого століття. Але скільки би мов високого рівня не виникло після неї та скільки б ще не з'явилося в майбутньому, вони застарівають, але тільки не мова асемблера. Через своє призначення вона не зникне доти, доки існуватимуть комп'ютери. Найголовнішим аргументом переконання в цьому є те, що будь-яка програма, яка написана на мові високого рівня, має виконуваний файл. Проте усі виконувані файли для свого створення були отримані шляхом перетворення команд з мови високого рівня в машинні команди (коди). Асемблер за свої задумом має можливості символічного представлення команд і розміщення коду та даних у пам'яті. І тому, якщо в якійсь мові високого рівня з'являються нові можливості щодо обробки даних, то такі можливості спершу мають бути закладені в асемблері. Живучість такої мови як С великою мірою пояснюється тим, що вона наближена до асемблерів для різних процесорів. Відтак мова асемблера постійно удосконалюється і доповнюється новими командами. Програміст, який вивчав мову асемблера десятилітньої давнини, вже не може вважатися експертом по асемблеру, тому що за цей час з'явилося так багато нових наборів команд, які суттєво змінили знання про асемблер.

Не треба забувати, що асемблер найближче знаходиться до рівня мікропрограм процесора і має враховувати архітектуру процесора на рівні команд, а це значить, нема нічого спільного в асемблерах для процесорів типу CISC та типу RISC крім загальних понять мнемоніки команд. Слід відзначити також, що способи використання мови залежить також від операційної системи, в якій він встановлений, а також і від конкретної реалізації мови асемблера як програмного продукту. В даному посібнику розглянемо особливості використання мови асемблер як на архітектурі CISC, так і на архітектурі RISC.

Спочатку розглянемо асемблери, пристосованих до команд процесорів x86 (типу CISC) і використанні в середовищах WIN 32 та LINUX.

РОЗДІЛ 1. АМСЕМБЛЕРИ, ОРІЄНТОВАНІ НА CISC-АРХІТЕКТУРУ

Асемблери, орієнтовані на CISC-архітектуру мають реалізувати обчислювальні можливості процесорів такої архітектери. Самі процесори мають довгу історію свого розвитку з початку 70-х років XX століття від 4-розрядних до 64-розрядних, причому справедливо це повязують з досягненнями фірми Intel, хоча на даний час розробкою та виробництвом таких мікропроцесорів займається не тільки ця фірма. Крім Intel, цю архітектуру підтримують для своїх процесорів такі вендори як: AMD, VIA, Transmeta, IDT та інші. В середовищі IT ці процесори відносять до класу, відомого як x86. Вдосконалений варіант цієї архітектури із розрядністю 32 біти називається IA-32 (Intel Architecture 32). Треба підкреслити - у світі мікропропроцесорів архітектура x86 не була і не є наразі єдиним представником архітектури CISC.

Зокрема сімейство Motorola MC68000, або як її називають професіонали - M68K, було одним із перших так званих 16-розрядних мікропроцесорів зі складним набором команд(CISC) і сучасником Intel 8086. На відміну від процесора Intel, 68K не намагався бути зворотно сумісним із попередніми 8- розрядними чіпами компанії. 68K вперше з'явився на ринку в кінці 1980 року (через два роки після Intel 8086). 16-розрядний 68K мав 32-розрядну архітектуру ISA, чим випередив Intel. Важко повірити, що 68K продавався як 16- бітна машина, конкуруючи з іншими 16 -бітними машинами, коли це була справжня 32-бітна машина. Це була 16-бітна машина лише в тому сенсі, що шина даних мала 16-бітну ширину. Регістри та багато внутрішніх операцій (додавання, логіка, зсув) були 32-розрядними.

Процесори M68K спочатку користувалися значним попитом. Вони були обрані для робочих станцій Sun, комп'ютерів Apple Macintosh, Amiga та Atari. Sega прийняла M68K для своєї консолі Mega Drive. Тому не дивно, що Apple ще у першій половині 1980-х вдалося реалізувати на своїх машинах візуальний інтерфейс для користувачів. Як і інші компанії, Motorola випустила нові версії сімейства M68K з удосконаленнями ISA та більшою швидкістю. Однак 68K зрештою не став конкурентом Intel. Домінуючим став ПК IBM на базі архітектури Intel IA32. Навіть Apple зрештою на певний час прийняла архітектуру IA32. M68K знайшла свою нішу сьогодні як сімейство мікроконтролерів, які продаються фірмою Freescale Semiconductor.

Проте ідея архітектури CISC почалась задовго до ери персональних обчислень. Основоположником CISC-архітектури можна вважати IBM з її базовою архітектурою

IBM/360, ядро якої використовується з 1964 року і збереглося до наших днів, наприклад, в таких мейнфреймах, як IBM ES/9000. Велику роль у встановленні CISC-архітектури відіграла також серія комп'ютерів PDP фірми DEC, особливо її модель PDP-11: спочатку 16-бітна машина, а потім 32-бітна серія відома під назвою VAX. Ідеї системи команд PDP-11 вплинула на розробку мікропроцесорів різних розробників: Motorola, Intel, Renesas, Texas Instruments та багатьох інших, хоча процесори VAX і не були інтегральними – технології відійшли в минуле, а ідеї живуть. Сімейство PDP-11 було виключно довгоживучим, залишаючись на ринку понад 20 років.

Лідером в розробці мікропроцесорів з повним набором інструкцій (Complete Instruction Set Computer) справедливо довгий час була компанія Intel. Ця компанія і зараз залишається у першому ряду мікропроцесорної індустрії, а архітектура Intel є фактичним втіленням CISC-архітектури в технології інтегральних схем. Мікропроцесори Intel та її численних конкурентів пережили дивовижну еволюцію з початку 70-х до наших днів. Після більш ніж 30 років архітектура x86 продовжує розвиватися, тоді як більшість технологій застарівають протягом десяти років. Це тому, що це є адитивна технологія. Тобто, якщо Intel додавала нову технологію до своєї архітектури ЦП, то ніколи не позбавлялась старої. Зокрема на оригінальному 8086 не було можливості перемикання контексту, що не дозволяло запустити на ньому багатопроцесону ОС. Потім Intel додала захищений режим на 286 і 386 для багатопроцесорної обробки. Але команди і концептуальні принципи адресації попередньої версії збереглись. Однак при цьому значно ускладнилась архітектура, додались нові команди.

Тому не дивно, що і асемблери, які мали обслуговуати теж пережили відповідну еволюцію. Відтак зробимо короткий екскурс у історію розробок асемблерів для персональних комп'ютерів для архітектури Intel.

Першим у цьому ряду був асемблер для 8-розрядного процесора I8080 для операційної системи CP/M. За ним був асемблер для 16-розрядного I8086 ОС DOS, яким майже 30 років тішилися покоління програмістів, і ще продовжують навчатися студенти у деяких університетах.

І хоча базовий набір команд і принципи системи переривань залишалися незмінними, необхідно було її доповнювати з врахуванням ускладнення архітектури процесорів. Одним з перших удосконалень мови асемблера стала розробка набору команд для виконання операцій з плаваючою точкою в мікропроцесорі 8087.

Потім були реалізовані система команд SIMD і технологія Intel® MMX™. Система команд SIMD використовувалася корпорацією Intel для збільшення рівня паралелізму

при обробці цілочислових даних в мікроархітектурі P5 (січень 1997 р.) за допомогою застосування спеціальних команд (57 команд на першому етапі). Використовуючи набір команд технології Intel MMX, програмісти могли виконувати команди для декількох елементів даних, завантажених в регістри MMX (виконаних на основі стека співпроцесора), що забезпечувало підвищення продуктивності мультимедійних застосувань, зокрема застосувань для роботи з графікою, ігор, потокового відео і так далі.

У мікроархітектурі P6 корпорація Intel ввела набір команд Streaming SIMD Extensions (SSE) на основі нових регістрів XMM0-XMM7. Набір команд SSE, розроблений для мікропроцесора Intel® Pentium® III (початок 1999 р.), розширював можливості технології Intel MMX і дозволяв одночасно виконувати обчислення SIMD з чотирма елементами даних з плаваючою комою з одинарною точністю, використовуючи 128-розрядні регістри (назва XMM0- XMM7).

З мікроархітектурою Intel Netburst® (процесор Intel® Pentium® 4) корпорація Intel ввела набір команд SSE2 (червень 2000 р.) з метою розширення набору команд SSE (і технології Intel MMX). Набір команд SSE2 забезпечив можливість паралельно виконувати більше обчислень, розширюючи команди, введені в технології Intel MMX і набір команд SSE, та забезпечуючи підтримку 128-розрядних типів даних цілих чисел і упакованих чисел з плаваючою точкою з подвійною точністю. У набір команд SSE2 були додані 144 нові команди, що забезпечило підвищення продуктивності різноманітних застосувань. Зауважимо, що перший набір так званих базових команд теж містив 144 команди.

Після випуску мікропроцесора Intel® Pentium® 4 на базі 90-нанометрової виробничої технології був розроблений набір команд SSE3. У початковий набір команд SSE3 (лютий 2004 р.) входило 13 додаткових команд SIMD. Ці команди перш за все були направлені на поліпшення синхронізації потоків і розширення можливостей математичних операцій x87 і операцій з плаваючою точкою.

Наступний крок був зроблений із створенням додаткового набору команд SSE3, який реалізований сьогодні в мікроархітектурі Intel® Core™. Додатковий набір команд SSE3, використовуваний в мікропроцесорах Intel® Xeon® 5100 (для серверів і робочих станцій) і Intel® Core™2 Duo (для мобільних і настільних ПК), додає 32 нових коди операцій, включаючи операції приведення у відповідність, множення і додавання, що забезпечує ще більшу продуктивність системи.

Корпорація Intel для архітектури Intel® 64 (ISA), починаючи з мікропроцесора Core 2 Duo E6700 та корпорація AMD для МП Phenom 9600+, впровадили новий набір

команд SSE4, який включає компоненти, що дозволяють використовувати розширені функціональні можливості мікропроцесору, підвищену продуктивність і покращувану енергоекономічність для більшості застосувань. Крім того, вже з 2000 року використовується 64-розрядний режим, який, з початку, фірмою Intel в мікропроцесорах Itanium використовував нову архітектуру IA-64. В цій архітектурі використовувався режим емуляції 32-розрядної архітектури, який був надзвичайно повільним. Компанія AMD запропонувала альтернативне вирішення проблеми збільшення розрядності мікропроцесора: було запропоновано ввести 64-розрядне розширення до вже існуючої 32-розрядної архітектури x86. Нова архітектура спочатку називалася x86-64, а потім AMD64. Фірма Intel створила набір інструкцій, повністю сумісний з AMD64. При цьому були додані ряд специфічних інструкцій, відсутні у початковому наборі AMD64. Нова версія архітектури отримала назву EM64T.

З 2008 року на основі інструкцій SSE4 для мікропроцесорів Intel®Core™2 компанією Intel введені інструкції AVX (Advanced Vector Extensions), які збільшують ступінь паралелізму і пропускну спроможність в дійсних SIMD обчисленнях та зменшують навантаження на регістри завдяки неруйнуючим трьохоперандним операціям.

В 2011 р. корпорація Intel підтвердила, що її процесори з кодовим ім'ям Haswell отримали підтримку набору інструкцій AVX2:

AVX здатності розповсюдилися на цілочисельні (integer) типи і операції з ними;

з'явилися інструкції, які дозволяють робити тип GATHER операцій; при таких операціях не потрібно, щоб дані знаходилися в безперервних ділянках пам'яті;

FMA (Fused Multiply-Add) – одна інструкція виконує декілька операцій над упакованими даними. Додані спеціалізовані інструкції для маніпуляції бітами.

Цей процес продовжується далі.

Не можна забувати і про те, що з постійним введенням нових команд не припиняються роботи, які спрямовані на удосконалення модульності асемблера. З появою API-функції під 32 - та 64 - розрядні операційні системи Windows зробило програмування застосунків MS-DOS не тільки застарілим підходом, але і практично неможливим. І хоча не можна не визнати, що таке програмування було б корисним для навчання початківців, на зразок того, як курсантів вчать літати спершу на застарілих моделях літаків, тим не менше Microsoft рішуче відмовилась підтримувати 16-розрядні застосування і з цим доводиться рахуватися. А для операційних систем типу UNIX це питання взагалі не є актуальним. Процес створення нових процесорів продовжується,

відповідно змінюються командні набори для асемблерів. Обсяг посібника не дозволяє розвинути далі цю тему. Однак розробникам мовою асемблера слід бути уважним і враховувати з якою моделлю процесора вони мають справу, особливо коли це стосується 64-розрядних застосунків. Особливо важливо враховувати якій концептуальній основі підлягає архітектура даного процесора – Intel чи AMD? Відповіді на питання особливостей використання нових команд для сучасних процесорів треба шукати в технічній документації.

Природно, що у мови асемблера є й недоліки. Головним її недоліком є те, що програма, написана для одного типу комп'ютерів, не може бути перекомпільована і використана на інших типах процесорів. Для кожного сімейства мікропроцесорів використовується своя мова асемблера зі своїм синтаксисом. Це є наслідком того, що мова асемблера дуже близька до машинних команд і до архітектури мікропроцесора. Саме тому, щоб глибше зрозуміти архітектуру мікропроцесора, без мови асемблера не обійтися.

Наразі ніхто не пише великі закінчені проекти на "чистому" асемблері. На практиці великі за об'ємом програми пишуться мовою високого рівня, переважно в C/C++ з асемблерними вставками для критичних ділянок коду. Навіть коли програма середньої складності написана на мові високого рівня, то все одно більшість бібліотечних функцій, що входять в Visual C++ чи іншою мовою, які були підключені, написані в асемблері. А паралельні обчислення, що грунтуються на технології SIMD, написані тільки з використанням сучасних асемблерних команд.

Нарешті, без асемблера неможливо розібратися у програмі, де вихідний текст програми на мові високого рівня недоступний. А це є стандартною практикою для аналізу усіх підозрілих щодо безпеки програм, коли експерти визначають сигнатуру шкідливої програми. У цьому випадку тільки асемблер може допомогти розібратися в програмі.

1.1 Архітектурні особливості процесорів IA-32

Архітектура IA-32 являє собою характерні особливості CISC-архітектури, зорієнтованої на ідею реалізації багатозадачного операційного середовища, що значно відділяє її від попередниці IA-16, яка мала суттєві обмеження для реалізації багатозадачності. Разом з тим, ця архітектура несуттєво відрізняється від IA-64 з точки зору забезпечення багатозадачності, хоча остання має багато додаткових можливостей.

Тому це хороша основа для вивчення сучасних особливостей архітектури процесорів, а заодно особливостей їх програмування без надмірних ускладнень.

1.1.1 Регістри процесора

Сучасні мікропроцесори нараховують десятки мільйонів регістрів. Проте серд них є особливі, які один з класиків IT Пітер Нортон влучно назвав грифельною дошкою для програмістів. В архітектурі IA-32 є десять 32-бітних і шість 16-бітних процесорних регістрів важливих для програміста.

Регістри діляться на чотири категорії :

- Регістри загального призначення (General Registers)
- Регістри управління (Control Registers)
- Сегментні регістри (Segment Registers)
- Регістр прапорів (FLAGS (16 біт) / EFLAGS (32 біта) / RFLAGS (64 біта)) – містить поточний стан процесора.
- регістри співпроцесора ST(0), ST(l), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7) призначені для написання програм, які використовують тип даних з плаваючою точкою;

У свою чергу, регістри загального призначення діляться на наступні:

- Регістри даних (Data Registers)
- Регістри-вказівники (Pointer Registers)
- Індексні регістри (Index Registers)

Всі перелічені регістри будемо вважати базовими. Проте слід додати, що розвиток архітектури мікропроцесорів привів до появи додаткових груп регістрів, без яких функціонування сучасних багатозадачних операційних систем була б неможлиим, або суттєво знижена функціональність нових прикладних програм.. Це так звані недокументовані регістри. Вони поділяються на регістри керування, регістри налагодження, тестові регістри та регістри сегментації захищеного режиму.

- Регістри керування - від CR0 до CR4,
- регістри налагодження - від DR0 до DR7,
- регістри тестування - від TR3 до TR7,
- регістри сегментації захищеного режиму - GDTR (глобальний регістр таблиці дескрипторів), IDTR (регістр таблиці дескрипторів переривань), LDTR (локальний DTR) і TR.

Потреби впровадження мультимедіа приведи до появи цілочисельних регістрів MMX-розширення MMX0, MMX1, MMX2, MMX3, MMX4, MMX5, MMX6, MMX7 та

регістрів MMX-розширення з плаваючою точкою XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7;

Нові моделі x86_64 також визначають набір великих регістрів для операцій з плаваючою комою та операцій з однією інструкцією/багатьма даними (SIMD).

Однак у даному посібнику сконцетруємось на розгляді базових регістрів, маніпулювання якими і визначає сутність програми на асемблері, причому за основу візьмемо набір для IA – 32.

Регістри даних – це чотири 32-бітних регістри, які використовуються для арифметичних, логічних і інших операцій. Ці 32-бітові регістри можуть бути використані наступними трьома способами:

- Як повні 32-бітові регістри даних: EAX, EBX, ECX, EDX.
- Нижні половини 32-бітних регістрів можуть використовуватися як чотири 16-бітних регістра даних: AX, BX, CX і DX.
- Нижня і верхня половини зазначених вище чотирьох 16-бітових регістрів можуть використовуватися як вісім 8-бітних регістрів даних: AH, AL, BH, BL, CH, CL, DH і DL.

Треба звернути увагу на цей термін *регістри даних*, хоча в регістри насправді записуються як коди даних та і коди команд. Справа в тому, що для процесора будь який код є даними, на основі яких він виконує команди. Ідея поділу регістрів даних закладена ще у 70-х роках, була продиктована проблемою зворотньої сумісності, тобто можливістю виконання програм, написаними для процесорів попередніх версій процесорами наступних версій.

Цю ідею було відповідно реалізовано і для 64-розрядних процесорів, в яких 32-бітові регістри EAX, EBX, ECX, EDX виступають як молодші регістри 64-розрядних RAX, RBX, RCX, RDX. Роль цих регістрів для.програмування в архітектурі x86 – особлива. В літературі часто зустрічається назва «Регістри загального призначення» тільки стосовно цих чотирьох регістрів, тому, що саме маніпуляціями з ними забепечуються всі основні команди процесора. Крім того, в них, і тільки в них визначаються параметри системних викликів. На Рис. 1.1 та Рис. 1.2 показані регістри даних та ідея зворотньої переносимості.

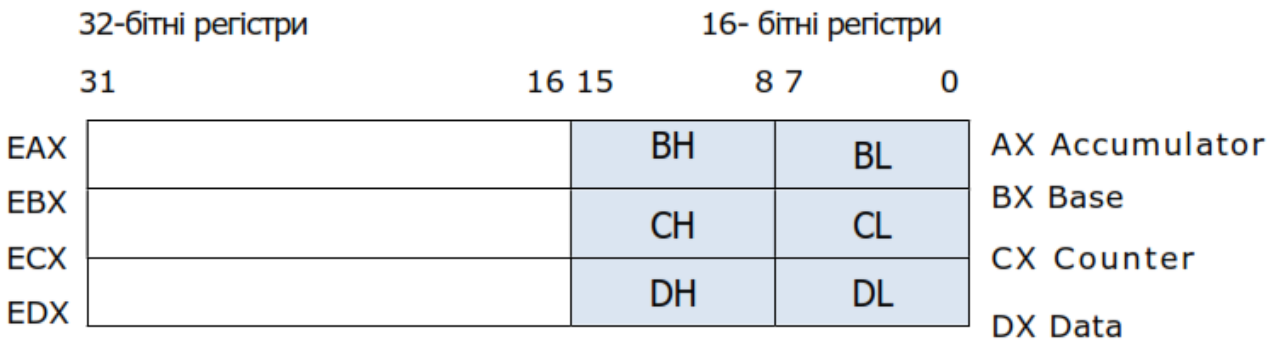

Рис 1.1 Регістри даних IA-32

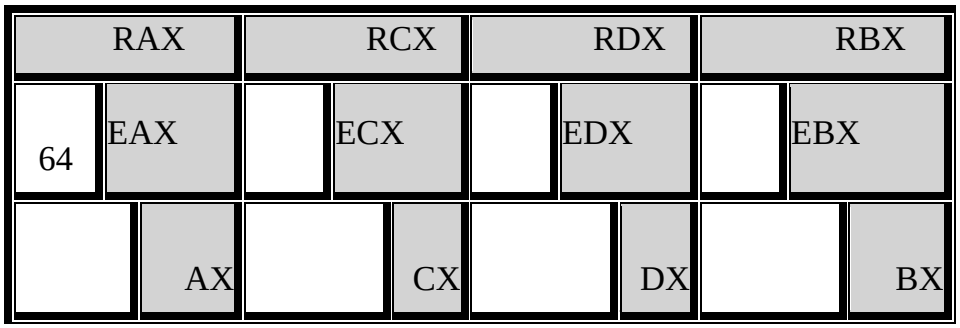

Рис. 1.2 Регістри даних для загальної архітектури x86

Деякі з цих регістрів даних мають специфічне застосування в арифметичних операціях.

EAX (primary accumulator) використовується для введення / виведення і в більшості арифметичних операцій. Наприклад, в операції множення один операнд зберігається в регістрі EAX або AX або AL відповідно до розміру операнда.

EBX (base register) використовується при індексованої адресації.

ECX (count register) зберігає кількість циклів в повторюваних операціях (також, як і регістри ECX і CX).

EDX (data register) використовується в операціях введення/виводу, а також з регістрами EAX і EDX для виконання операцій множення і ділення, пов'язаних з великими значеннями.

Регістри-вказівники є 32-бітові регістри EIP, ESP і EBP і відповідні їм 32-бітові регістри EIP, ESP і EBP. Є три категорії регістрів-вказівників:

Вказівник на інструкцію або команду (Instruction Pointer або EIP) - 32-бітний регістр EIP зберігає зсув адреси наступної команди, яка має бути виконана. EIP в поєднанні з регістром CS (як CS: EIP) надає можливість отримати адресу поточної

інструкції в сегменті коду. До нього неможливо отримати прямий доступ за допомогою команд. Цей регістр неявно контролюється інструкціями передачі керування (такими як JMP, Jcc, CALL і RET), перериваннями та винятками. Єдиний спосіб прочитати регістр EIP - це виконати інструкцію CALL, а потім прочитати значення вказівника інструкції повернення зі стеку процедур.

Вказівник на стек (Stack Pointer або ESP) - 32-бітний регістр SP забезпечує значення зміщення в програмному стеку. SP в поєднанні з регістром SS (SS: SP) відноситься до поточної позиції даних або адреси в програмному стеку.

Базовий вказівник (Base Pointer або EBP) - 32-бітний регістр EBP використовується в основному при передачі параметрів в підпрограми. Адреса в регістрі SS об'єднується зі зміщенням в EBP, щоб отримати місце розташування параметра. EBP також можна комбінувати з EDI і ESI як базовим регістром для спеціальної адресації.

Сегментні регістри. Наявність сегментних регістрів є характерною рисою архітектури x86. Основна ідея розробників полягала в тому, щоб на апаратному рівні розділити області пам’яті, де розміщується код програми, дані та стек. Сегментні регістри містять адреси сегментів різних елементів. Вони мають лише в 16 розрядів. Вони можуть бути встановлені лише загальним регістром або спеціальними інструкціями. Деякі з них мають вирішальне значення для якісного виконання програми. Сегментні регістри, що використовуються в сегментованому режимі, вказують на різні сегменти в пам'яті. Кожен 16-бітний сегментний регістр дає змогу переглядати 64 Кб (16 біт) даних. Після того, як сегментний регістр встановлено для вказівки на блок пам’яті, регістри (такі як BX, SI та DI) можна використовувати як зміщення сегментного регістра, щоб можна було отримати доступ до певних місць у просторі 64 Кб.

Процесори 86-64 мають загалом 6 сегментних регістрів: CS, SS, DS, ES, FS і GS. Операція залежить від режиму ЦП. У всіх режимах, крім тривалого режиму(long mode) для 64-розрядного процесора, кожен сегментний регістр містить селектор, який індексує GDT або LDT. Це дає дескриптор сегмента, який, серед іншого, надає базову адресу та розмір сегмента. Кожен дескриптор сегмента сам зберігається десь у пам'яті разом з усіма іншими дескрипторами сегмента. У сукупності це «таблиця дескрипторів сегментів». Половина кожного дескриптора містить 32-розрядну адресу початку сегмента. Двадцять біт визначають довжину сегмента, що дає однобайтову деталізацію до 1 МБ (2 20). Ще є так званий біт гранулярності в дескрипторі сегмента, що дозволяє перетворити поле довжини в одиниці розміром 4 КБ, дозволяючи програмісту визначати величезні сегменти пам’яті, хоча й із більш грубою деталізацію розміру. На Рис. 1.3

показана схема сегментації коду за допомогою дескриптора, початковим елемнтом якого є значення сегментного регістра CS.

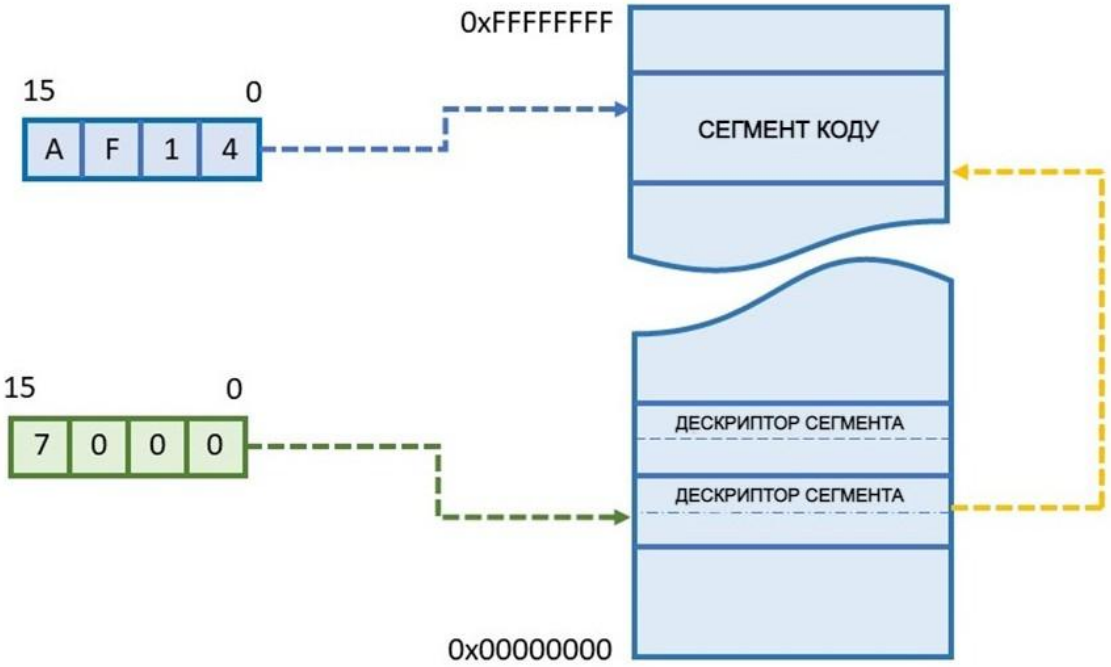

Рис. 1.3 Сегментація пам’яті

Значеннями 11 біт, що залишилися,визначення тип сегмента (код, дані, стек), чи є він лише для читання чи читання-запису, його рівня привілеїв (два біти), біт для того, чи застосовуються привілеї для цього сегмента, один для обробки цього сегмента як 16-бітного коду (для зворотної сумісності), а інший для сповіщення системи про фізичну присутність пам’яті (що допомагає при обробці віртуальної пам’яті та помилок сторінки).

У тривалому режимі всі сегментні регістри, крім FS і GS, розглядаються як такі, що мають нульову базову адресу та 64-бітний екстент, фактично створюючи плоский адресний простір. FS і GS зберігаються як спеціальні випадки, але більше не використовують таблиці дескрипторів сегментів: натомість вони отримують доступ до базових адрес, які зберігаються в регістрах FSBASE і GSBASE. CS містить сегмент коду, у якому виконується ваша програма.Зміна його значення може спричинити зависання комп’ютера.DS : містить сегмент даних, до якого має доступ ваша програма.Зміна його значення може дати помилкові дані.ES, FS, GS: це доступні додаткові сегментні регістри адресація далекого вказівника, як-от відеопам'ять тощо. SS: Зберігає сегмент стека, який використовує ваша програма. Іноді має те саме значення, що й DS.Зміна його значення може дати непередбачувані результати,в основному пов'язані з даними.

Регістри **EIP та FLAGS** часто об’єднують в одну групу стану і управління. При цюму, хоча EIP фактично виконує вкрай важливу роль лічильника команд, регістр EFLAGS набагато важливіший для операційної системи. Регістр EFLAGS (статус програми та керування) використовується для надання статусу програми та виконання відповідного керування. Регістр EFLGAS повідомляє про стан програми, що виконується, і дозволяє обмежено (на рівні прикладної програми) контролювати процес. У 64-розрядному режимі регістр EFLGAS розширюється на 64-розрядний регістр RFLGAS, верхні 32 біти зарезервовані, а молодші 32 біти такі ж, як і регістр EFLAGS. 32-розрядний регістр EFLAGS містить набір прапорів стану, прапорів системи та прапор керування. Після ініціалізації процесора x86 значення статусу регістра EFLAGS становить 0000 0002H. Біти 1, 3, 5, 15 і від 22 до 31 зарезервовані. Деякі прапорці в цьому регістрі можна безпосередньо змінити за допомогою спеціальних загальних інструкцій, але немає інструкцій для перевірки або зміни всього регістру. Використовуючи такі інструкції, як LAHF/SAHF/PUSHF/POPF/POPFD, можна перемістити біти прапорів регістра EFLAGS у стек програм або регістр EAX у групах або зберегти результат операції з цих засобів у регістр EFLAGS. Після того, як вміст регістра EFLAGS буде передано в стек або регістр EAX, ці біти прапорів можна перевірити або змінити за допомогою інструкцій обробки бітів (BT, BTS, BTR, BTC).

При виконанні кожної програми від початку і до її завершення значення цього регістра має бути збереженим. В умовах режиму багатозадачності це потребує спеціального механізму. Під час виклику обробника переривання або виняткової ситуації процесор автоматично збереже значення стан EFLAGS у програмному стеку. Якщо перемикання завдань відбувається під час обробки переривання або виняткової ситуації, стан регістра EFLAGS буде збережено в TSS. Слід зауважти, що стан EFLAGS регістр зберігається в TSS для призупиненого завдання. На Рис. 1.4 показані біти регістра EFLAGS від яких залежить статус програми. Всі ці біти у свій час розробники назвали прапорцями(FLAGS). Розміщення прапорців показано на Рис. 1.4.

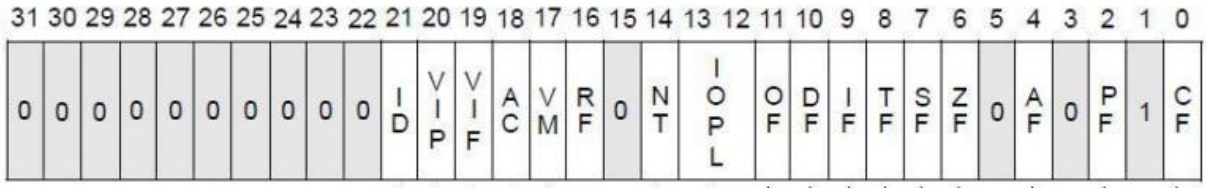

Рис. 1.4 Регістр EFLAGS

1. Прапори стану (біти **0, 2, 4, 6, 7** і 11 **)** регістра EFLAGS вказують на результати виконання арифметичних інструкцій (таких як інструкції ADD, SUB, MUL і DIV). Функції цих прапорців стану такі:

- ***CF(bit 0) [Carry flag]*** Якщо результат арифметичної операції міститься в старшому біті (найстаршому біті), йому буде встановлено значення 1, інакше скидається. Цей прапорець вказує на статус переповнення цілочисельних операцій без знаку. Цей прапор також використовується в арифметиці множинної точності.
- ***PF(bit 2) [Parity flag]*** Якщо найменший байт результату містить парну кількість біт 1, тоді цей біт встановлюється на 1, інакше він очищається.
- ***AF(bit 4) [Adjust flag]*** Якщо арифметична операція має перенесення або запозичення в третьому біті результату, установіть цей прапор на 1, інакше зніміть його. Цей прапорець використовується в арифметичних операціях BCD (десятковий двійковий код).
- ***ZF(біт 6) [Позначка нуля]*** Якщо результат дорівнює 0, встановіть його на 1, інакше очистіть його.
- ***SF(біт 7) [Прапор знаку]*** Цей прапорець встановлюється на старший біт цілого числа зі знаком. (0 означає, що результат позитивний, інакше він негативний)
- ***OF(біт 11) [Прапор переповнення]*** Якщо цілочисельний результат є більшим додатним числом або меншим від'ємним числом, а операнд призначення не може бути зіставлений, біт буде встановлено на 1, інакше Очищено. Цей прапорець вказує на статус переповнення для цілочисельних операцій зі знаком.

Серед цих прапорів стану лише прапор CF можна безпосередньо змінити за допомогою інструкцій STC, CLC і CMC, або вказаний біт можна скопіювати до прапора CF за допомогою інструкцій біта (BT, BTS, BTR і BTC).

Ці прапорці стану дозволяють одній арифметичній операції отримати результати трьох різних типів даних: ціле число без знака, ціле число зі знаком і ціле число BCD. Якщо результат розглядається як ціле число без знаку, тоді прапор CF вказує на стан поза межами діапазону, тобто перенесення або запозичення. Якщо він розглядається як ціле число зі знаком, прапор OF вказує на перенесення або запозичення. номер BCD, тоді прапорець AF вказує на перенесення або запозичення. Прапор SF вказує на знаковий біт цілого числа зі знаком, а ZF вказує на те, що результат дорівнює нулю. Крім того, під час виконання арифметичних операцій із багаторазовою точністю прапор CF використовується для перенесення переносу або запозичення, згенерованого додаванням

із переносом (ADC) або відніманням із запозиченням (SBB) під час однієї операції до наступної операції.

2. Прапор DF (прапор DF) Цей прапор напрямку (розташований у 10-му біті регістра EFLAGS) керує рядковими інструкціями (MOVS, CMPS, SCAS, LODS і STOS). Встановлення прапорця DF автоматично зменшує рядкову інструкцію (обробка рядка від старшої адреси до нижчого напрямку адреси), а зняття прапорця змушує рядкову інструкцію автоматично збільшуватися. Інструкції STD і CLD використовуються для встановлення та очищення прапора DF відповідно.

3. Системні прапори та поле IOPL Ця частина прапорів у регістрі EFLAGS використовується для керування операційною системою або виконання операцій, і **їх не дозволяється змінювати прикладною програмою**. Функції цих знаків такі:

TF(bit 8) [Trap flag] Встановіть цей біт на 1, щоб дозволити одноетапний режим налагодження, і зніміть його, щоб вимкнути цей режим.

IF (біт 9) [Прапор дозволу переривання] Цей прапор використовується для керування відповіддю процесора на запити переривання, що маскуються. Встановіть 1, щоб відповідати на масковані переривання, інакше вимкніть масковані переривання. ***IOPL (біти 12 і 13) [поле рівня привілеїв вводу-виводу]*** вказує ***рівень привілеїв вводу-виводу*** поточного завдання (рівень привілеїв вводу-виводу), поточний рівень привілеїв (CPL) поточного завдання має бути меншим за або дорівнює I/O Лише привілейовані рівні можуть дозволити доступ до адресного простору I/O. Це поле можна змінити лише за допомогою інструкцій POPF та IRET, коли CPL дорівнює 0.

NT (біт 14) [Прапор вкладеного завдання] Цей прапор керує ланцюжком переривань і викликаним завданням. Установіть значення 1, якщо поточне завдання пов’язане з попереднім завданням виконання, інакше зніміть.

RF(біт 16) [Прапор відновлення] контролює реакцію процесора на винятки налагодження.

VM (біт 17) [Прапор режиму віртуального 8086] встановлено на 1, щоб дозволити віртуальний режим 8086, скинуто, щоб повернутися до захищеного режиму.

- ***AC(біт 18) [Прапор перевірки вирівнювання]*** Цей прапорець і біт AM у регістрі CR0 дозволять перевірку вирівнювання посилань на пам’ять, якщо принаймні один із двох вищевказаних прапорців очищено, щоб вимкнути перевірку вирівнювання.

VIF(біт 19) [Прапор віртуального переривання] При встановлені прапора VIP та прапора VME у керуючому регістрі CR4, надається дозвіл розширення віртуального режиму (розширення віртуального режиму)

VIP (біт 20) [Прапор очікування віртуального переривання] Цей біт встановлено на 1, щоб вказати, що переривання очікує на розгляд. Цей біт очищається, якщо немає очікуваних переривань. [Програмне забезпечення встановлює та знімає цей прапорець; процесор лише читає його.] Використовується разом із прапором VIF. ***Ідентифікатор (біт 21) [Ідентифікаційний прапор]*** Програма може встановлювати або знімати цей прапорець, що вказує на підтримку процесором інструкції CPUID.

Питання для самоперевірки

Перелічіть категорії регістрів в архітектурі IA-32.

Яке функціональне призначення регістрів даних в архітектурі IA-32?

Яка роль регістрів даних при збільшенні розрядності в архітектурі IA-32?

Навіщо потрібні сегментні регістри в архітектурі IA-32?

Який максимальний обсяг памяті може адресувати 32 розрядний процесор, якщо для цього виділено один регістр?

Який з регістрів виконує роль лічильника команд в архітектурі IA-32?

За якими областями памяті закріплені кожен з сегментних регістрів в архітектурі IA-32?

Для чого призначені регістри EBP та ESP?

В чому призначення індексних регістрів в архітектурі IA-32?

Яку функціональну роль відіграє регістр EFLAGS в архітектурі IA-32?

Назвіть групи прапорів регістр EFLAGS.

В якому прапорі регістр EFLAGS відображається стан переривання?

1.1.2 Регістри співпроцесора

Необхідність розробки апаратної підтримки операцій з плаваючою точкою(комою) для процесорів x86 була усвідомлена досить давно із-за значної трудомісткості. На регістрах загального призначення операція множення дісних, наприклад, чисел існує і успішно виконується, проте повільніше два порядки ніж множення цілих чисел. Ветерани IT ще пам’ятають співпроцесор I8087, який вставлявся в окремий слот на платі персонального компютера, тобто являв собою окремо працюючий пристрій. Тобто окремо на материнську плату вставлявся пристрій - співпроцесор, який дозволяв робити обчислення з числами з точкою, що плаває. Тільки для I486 процесора з'явилося рішення, в якому на одному кристалі фізично об'єднувалися два пристрої - 486 і x87. Але тільки в Pentium співпроцесор був повністю інтегрований в кристал. Проте і зараз виконання команд на регістрах співпроцесора сприймається як робота з окремим пристроєм. Однак

у сучасних процесорах всі необхідні функціональні можливості, всі набори команд, придумані 40 років тому, не застаріли і успішно працюють. Співпроцесор x87 – це надзвичайно складний пристрій, майже як сам процесор. Тому розглянемо його дещо спрощено, як набір регістрів, що зберігають самі величини - числа з плаваючою точкою, і службових регістрів (керуючий регістр, регістр ознак, регістр стану, схема Рис..

Роботу із співпроцесором x87 забезпечує спеціальний набір команд, який починається з команди FINIT, яка виконує його ініціалізацію. Усі команди, що відносяться до x87, мають суфікс F. Команда FINIT скидає всі налаштування в початковий стан. Це здійснюється за допомогою запису в три службові регістри певних послідовностей бітів. Цікаво те, що x87 реалізує у собі стекову машину. Тут є підмножина команд, що дозволяє виконувати операції не специфікуючи операнди. Можливість такої роботи базується на тому, що в регістрі стану є поле, яке задає верхівку стека. Вісім регістрів, із якими ведеться робота, своєю чергою становлять стек регістрів. У спеціальному полі TOP (має три розряди) зберігається вказівник одного з регістрів (тобто якесь ім'я). Той регістр, на який посилається поле, називатиметься регістр ST0. Те, що знаходиться вище називається ST1, ST2 і так далі. На Рис.1.5 показана спрощена схема співпроцесора x87.

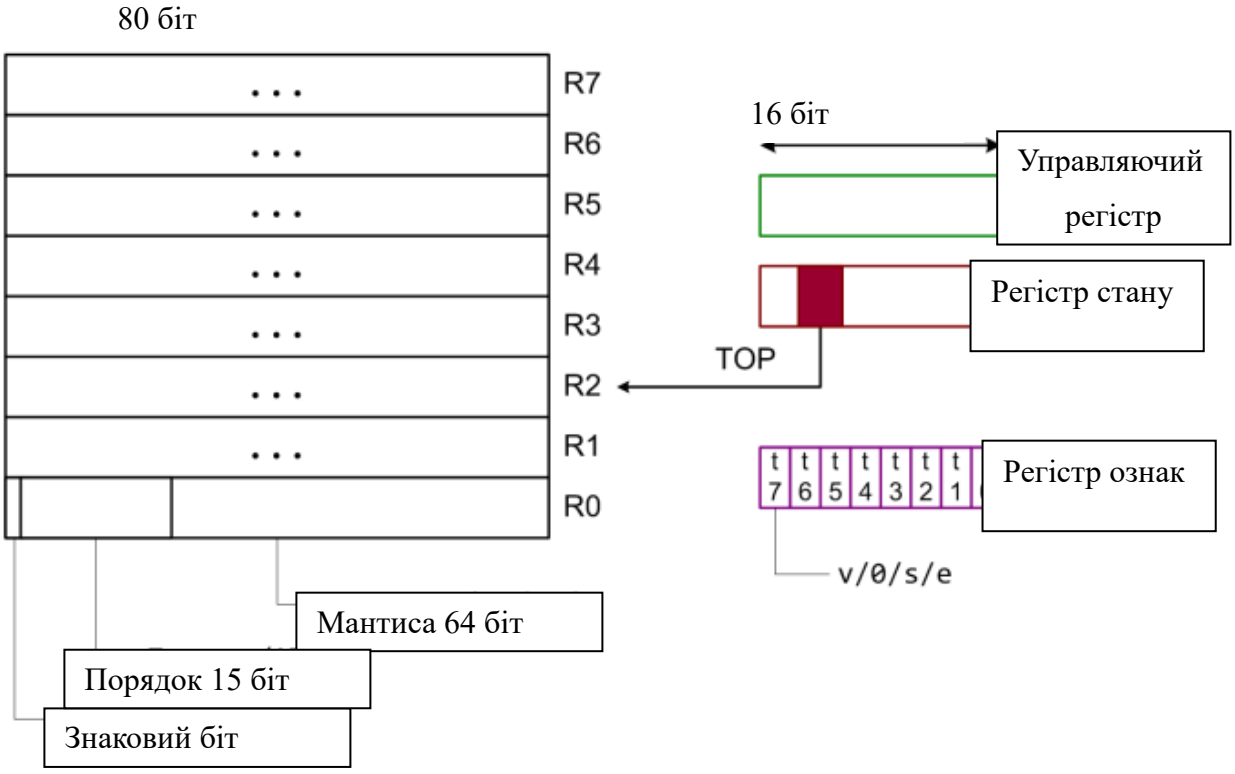

Рис.1.5 Спрощена схема співпроцесора x87.

Цікаво, що змінивши стан стека і пересунувши поле TOP вниз, зміниться вся система найменування. Таким чином, ST0 виявиться рівнем нижче, те, що було ST0, стане ST1. Крім того, регістри містять 80 розрядів, таким чином, вони реалізують розширену точність. Також, у мантисі x87 явно кодується ціла частина. Так як історично x87 і x86 стояли в різних місцях на платі, то немає жодної можливості безпосередньо передавати інформацію з регістрів загального призначення x86 у ці регістри з плаваючою точкою. Обміни можуть відбуватися лише з пам'яттю.

Регістр стану містить у собі 3 розряди (з 11 по 13), які кодують поточний номер верхівки стека (з апаратних регістрів). Найстарший розряд (15) показує, чи у співпроцесорі якісь обчислення (біт зайнятості). Розряди 14, 10, 9, 8 - це біти, що показують спеціальні умови, пов'язані з порівнянням чисел із плаваючою точкою. Далі, від 0 до 7 йдуть розряди, що показують наявність тих чи інших ситуацій, що виникли під час обчислень.

В управляючому регістрі молодші 6 розрядів по суті управляють реакцією на винятки з регістру стану. Тобто якщо в масці стоїть одиниця, то відповідна виняткова ситуація ігнорується. На початку FINIT виставляє в усі розряди одиниці. Тобто за умовчанням винятки ігноруються. Також присутні поля, що керують точністю та округленням. За промовчанням FINIT виставляє нулі в поле округлення, що означає округлення до найближчого парного. У полі точності за замовчуванням встановлюються дві одиниці, що означає використання розширеної точності під час обчислень.

Регістр ознак показує, що зберігається у регістрах із даними. При ініціалізації регістр заповнюється одиницями (16 одиниць). На кожен регістр із числами припадає по два розряди. Ці два розряди кодують вміст регістрів з такими даними:

• 0 означає нормалізоване число з плаваючою точкою

• 1 означає, що в регістрі записано нуль

• 2 означає, що у регістрі записані особливі числа (такі як нескінченність)

• 3 означає, що у регістр вільний

Отже, у початковому стані видно, що це регістри вільні.

Питання для самоперевірки

В чому призначення співпроцесора x87?

Яким чином ініціалізується робота співпроцесора x87?

Перелічіть регістри співпроцесора x87?

Яку функцію виконує регістр ознак?

Що відображується в регістр стану співпроцесора x87?

Що собою являє стек регістрів співпроцесора x87?

Які дії виконує управляючий регістр x87?

1.1.3 Машинні команди процесора IA-32

Процесор IA-32, як представник CISC-процесорів, виконує великий набір команд із багатими можливостями адресації, надаючи програмісту можливість вибрати найбільш придатну команду для виконання необхідної операції. Як наслідок це привело до ускладнення формату та збільшення розміру машинної команди, натомість дозволяє програмісту писати досить лаконічно. При цьому не всі команди можуть використовувати кожен зі способів адресації стосовно до кожного з регістрів процесора. Вибірка команди на виконання здійснюється побайтно протягом декількох циклів роботи мікропроцесора. Час виконання команди може складати від 1 до 12 циклів. Для процесора розмір команди IA-32 складає від 1 до 15 байт, тобто в архітектурі IA-32 машинна команда має змінний розмір. Вона має наступний формат, він справді складний(Рис. 1.6):

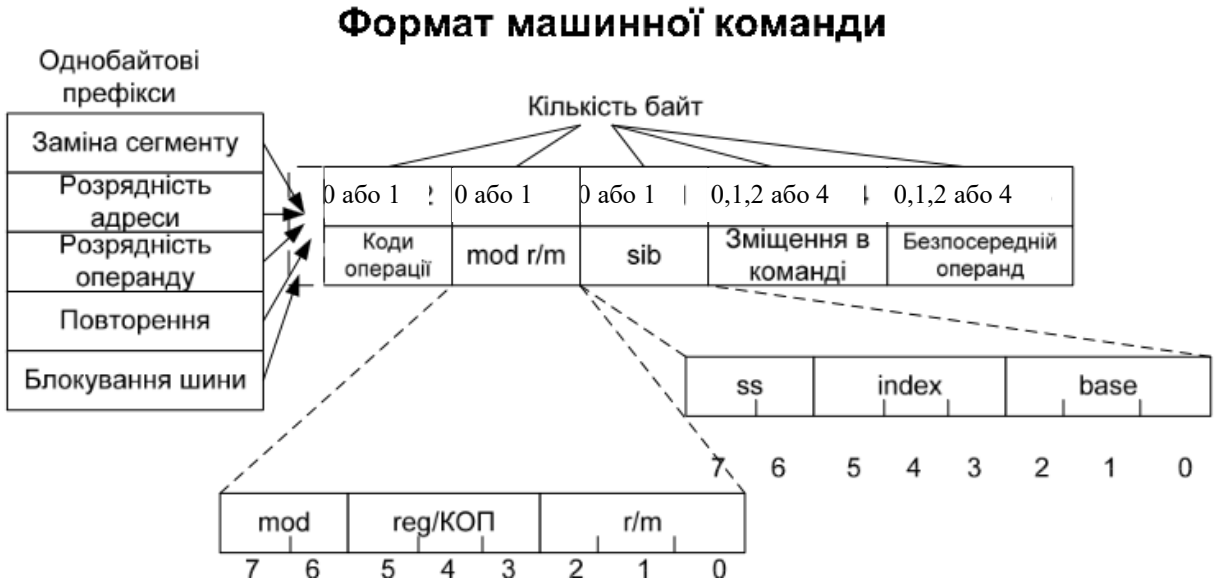

Рис.1.6 Формат машинної команди в IA - 32

- На Рис ми бачимо наступні поля:
- поле префіксів, необов’язкове,
- поле коду операції, обов’язкове,
- байт режиму адресації mod r/m,
- байт масштабу, індексу та бази,
- поле зміщення,
- поле безпосереднього операнда,

- префікс повторення - використовується тільки для рядкових команд;
- префікс розміру адреси (67h) - застосовується для зміни розміру зсуву: 16 біт при 32-х розрядної адресації;
- префікс розміру операнда (66h) - вказується, якщо замість 32-х розрядного регістра для зберігання операнда використовується 16-ти розрядний;
- префікс заміни сегмента - використовується при адресації даних будь-яким сегментом крім DS.

Поле коду операцій містить, власне код d - напрям обробки, наприклад, пересилання даних: 1 - в регістр, 0 - з регістра;

w - розмір операнда: 1 - операнди - подвійні слова, 0 - операнди - байти;

mod - режим:

00 - Disp = 0 - зсув в команді 0 байт;

01 - Disp = 1 - зміщення в команді 1 байт; 10 - Disp = 2 - зміщення в команді 2 байта;

11 - операнди-регістри.

Байт режиму адресації містить дані про операнди та режими адресації mod r/m. Операнди можуть перебувати в пам’яті, а також в одному або в двох регістрах, але обидва операнди у памяті бути не можуть. Відповідно це визначається трьома полями, показані на Рис.1.6

Байт масштабу, індексу та бази (Scale-Index-Base – sib) використовується для надання розширених можливостей адресації операндів. Байт sib складається з трьох елементів: поля масштабу (ss), поля index для зберження номера індексного регістра, поля base для зберження номера базового регістра.

За цим байтом ідуть поля зміщення та безпосереднього операнда для різних варіантів адресації

Регістри теж кодуються, причому в залежності від розміру операнда: w = 1 w = 0

reg 000 EAX

000 AL (R)

001 ECX

001 CL

010 EDX

010 DL

011 EBX

011 BL

100 ESP

100 AH

101 EBP

101 CH

110 ESI

110 DH

111 EDI

111 BH

Якщо в команді використовується певний регістр (наприклад, AX), то перед командою додається префікс зміни довжини операнда (66h).

Розрізняють два види команд, що обробляють операнд в пам'яті:

• команди без байта sib (див. Таблицю 1.1);

• команди, що містять байт sib (див. Таблицю 1.2).

Розрізняються ці команди по вміст поля m (r / m): якщо m=100, то байт sib в команді відсутній і використовується таблиця 1.1.

Таблиця 1.1 Схеми адресації пам'яті у відсутності байта sib

Поле r/m	Ефективна адреса другоого операнда		
	mod = 00B	mod = 01B	mod = 10B
000B	EAX	EAX+Disp8	EAX+Disp32
001B	ECX	ECX+Disp8	ECX+Disp32
010B	EDX	EDX+Disp8	EDX+Disp32
011B	EBX	EBX+Disp8	EBX+Disp32
100B	**Визначається Sib**	**Визначається Sib**	**Визначається Sib**
101B	Disp32	SS:[EBP+Disp8]	SS:[EBP+Disp32]
110B	ESI	ESI+Disp8	ESI+Disp32
111B	EDI	EDI+Disp8	EDI+Disp32

Таблиця 1.2 Схеми адресації пам’яті при наявності байта **sib**

Поле base	Ефективна адреса другого операнда		
	mod = 00B	mod = 01B	mod = 10B
000B	EAX+ss*index	EAX+ss*index +Disp8	EAX+ss*index +Disp32
001B	ECX+ss*index	ECX+ss*index +Disp8	ECX+ss*index +Disp32
010B	EDX+ss*index	EDX+ss*index +Disp8	EDX+ss*index +Disp32
011B	EBX+ss*index	EBX+ss*index +Disp8	EBX+ss*index +Disp32
100B	SS:[ESP+ss*index	SS:[ESP+ ss*index]+Disp8	SS:[ESP+ ss*index] +Disp32

	]		
101B	Disp32[1]+ss*index	SS:[EBP+ss*index +Disp8]	SS:[EBP+ss*index +Disp32]
110B	ESI+ss*index	ESI+ss*index +Disp8	ESI+ss*index +Disp32
111B	EDI+ss*index	EDI+ss*index +Disp8	EDI+ss*index +Disp32

ss - масштаб; Index - індексний регістр; Base - базовий регістр;1 - особливий випадок - адреса операнда не залежить від значення EBP, а визначається тільки зміщенням в команді (пряма адресація).

Приклади:

1) mov EBX,ECX 100010DW Mod Reg Reg

100010011100 1011

8 9 C B

1) mov BX,CX

префікс1 100010DW Mod Reg Reg

0110 0110 1000 1001 1100 1011

6 6 8 9 C B

2) mov ECX,DS:6[EBX] 100010DW Mod Reg Reg м.мол.байт

10001011010010110000011 0

8 B 4 B 0 6

3) mov CX,DS:6[EBX]

префікс 100010DW Mod Reg Reg Зм.мол.байт

01100110100010110100 1011 00000110

6 6 8 B 4 B 0 6

4) mov CX,ES:6[EBX]

префікс1 префікс2 100010DW Mod Reg Reg Зм.мол.байт

01100110001001101000101 1 0100101100000110

6 6 2 6 8 B 4 B 0 6

5) mov ECX,6[EBX+EDI*4]

100010DW Mod Reg Mem SS Ind Base Зм.мол.байт

10001011010011001011101100000110

8 B 4 C B B 0 6

Питання для самоперевірки

Що визначає префіксний байт?

Де розміщуються коди операцій?

Що визначають поля байту коду операцій?

Що міститься у байт режиму адресації?

Для яких випадків передбачений байт sib?

Яке призначення полів sib?

Чому поле зміщення може змінюватись від 0 до 4 байт?

Чому поле безпосередній операнд може змінюватись від 0 до 4 байт?

1.2 Середовища програмування для Асемблера

Конкретна реалізація мови Асемблер залежить не тільки від процесора, але також і від операційної системи, яка використовує цей процесор і від середовища програмування, в якому Асемблер реалізований. Більшість сучасних розробників середовищ програмування намагаються не прив’язуватись до конкретної операційної системи, а охопити коло найбільш поширених ОС.

1.2.1 Сучасні середовища

Для процесора x86-x32/64 існує більше десятка різних асемблеро-компіляторів, тобто компіляторів, які формують виконуваний код, починаючи прямо з мови асемблера. Вони відрізняються різними наборами функцій і синтаксисом. Який асемблер краще? З огляду на множину діалектів ассемблерів для x86-x64 і обмежену кількість часу для їх вивчення, обмежимося коротким переліком компіляторів: MASM, TASM, NASM, FASM, GoASM, Gas, HLA. Далі треба орієнтуватись на операційну систему, в якій пишуть програму.

У Таблиці 1.3 подані короткі характеристики можливостей асемблерів

Таблиця 1.3 Асемблери

	Windows	**DOS**	**Linux**	**BSD**	**QNX**	**MacOS, на Intel/AMD**
FASM	+	+	+	+		

GAS	+	+	+	+	+	+
GoAsm	+					
HLA	+		+			
MASM	+	+				
NASM	+	+	+	+	+	+
TASM	+	+				

Як ми бачимо різні асемблери по різному «заточені» під різні ОС.

З іншого боку є різні програмні середовища. Наприклад, RadASM - безкоштовне середовище(яке можна скачати з відповідного сайту) розробки програмного забезпечення для ОС Windows і не тільки, спочатку призначена для написання програм на мові ассемблера. Має гнучку систему файлів налаштувань, завдяки чому може бути використана як середовище розробки програмного забезпечення на високорівневих мовах, а також документів, що ґрунтуються на мовах розмітки. Більш детально деякі з цих середовищ описані в Додатку А.

Питання для самоперевірки

Які з асемблерів мають найбільше поширення для різних середовищ?

Наскільки вдалим є вибір TASM для вирішення сучасних задач?

Що ви знаєте про асемблер MASM?

1.3 Технологія асемблінгу

Терміном асемблер називають транслятор з мови асемблера, а часто і саму мову. Головною особливістю мови є використання в якості виконуваних операторів **мнемокод** машинних команд. Інакше кажучи, програмування мовою асемблерає програмуванням в машинних командах цільової архітектури, представлених мнемокодом, а не машинними кодами. Незалежно від використовуваного нами асемблера, формату об'єктного файлу, компонувальника або операційної системи, процес програмування завжди однаковий:

Вираз z = 2 * (z + 34) для випадку, коли z має тип long,

мовою асемблерадля IA-32 може бути представлено так:

```
add   dword ptr z, 34        ; додати 34 до z
shl   z, 1                   ; помножити на 2 через зсув вліво
```

Писати програми в асемблері набагато легше ніж в мікрокоді, проте і розуміння асемблера справа непроста.Тому порівняння кодів написаних на С і мовою асемблерає хорошим засобом для навчання.

Технологічні аспекти перетворення коду з мови асемблера у виконуваний можуть відрізнятися. Проте є два принципові моменти, які мають всі асемблери:

1. Вони мають підтримувати систему команд, зрозумілу процесору;
2. Вони мають визначену структуру програми, написаної на асемблерній мові.

Програма на мові асемблера поділяються орієнтовно на чотири розділи:

- заголовок
- дані
- тіло коду
- закриття

Це базові розділи. Проте оскільки стандарту не існує, можливі деякі відміності для реалізацій асемблера.

Асемблерна програма в цілому має таку загальну структуру:

```
Заголовок
; Вибір моделі пам’яті:
; Для 16 – розрядних програм це наприклад;
.model small ;
;Для 32-розрядних програм визначення моделі такі визначення не актуальні,
;натомість актуальним може бути оголошення моделі процесора:
.386
;а також плоскої моделі та формату виклику
.model flat, stdcall
.stack stack_size ; Визначення розміру стеку
.data ; Оголошення даних і масивів;
.code
main proc ; основний код програми на цьому рівні
main endp; закінчення коду на цьому рівні
; Інші процедури
; Бажано структурувати програму
; вхід в процедури
end main ;Визначення кінця вихідного файлу
```

Директиви сегментів: Сегменти оголошуються за допомогою директив. Наступні директиви використовуються для визначення наступні сегменти:

- стек
- дані

- код

Сегмент стека: Використовується для виділення місця для стека.

- Адреси стека обчислюються як зміщення в цьому сегменті
- Використовується:.*stack* із значенням, що вказує на розмір стека

сегменти даних

Сегмент даних: Використовується: директива *.data* з оголошеннями змінних або визначеннями констант.

Адреси змінних обчислюються як зміщення від початку цього сегмента. Проте в деяких асемблерах вводять окремі секції для констант та псевдооперації для оголошення іменованих констант. Тому треба уважно вчитуватись в документацію конкретного асемблера.

Кожен файл мови асемблера збирається у об'єктному файлі, і об'єктний файл пов'язаний з іншими об'єктними файлами, щоб утворити виконуваний файл. Статична бібліотека(Static Lib) насправді є не що інше, як колекція (можливо пов'язаних) об'єктних файлів. Прикладні програмісти зазвичай використовують бібліотеки для таких речей, як I/O і математичні функції. Після під’єднання статичних бібліотек може бути утворений виконуваний файл, який може бути завантажений на виконання. В сучасних операційних системах під час виконання може відбутися під’єднання динамічних бібліотек. Процес перетворення коду в асемблерній мові показано на Рис. 1.7. В цьому плані всі асемблери однакові.

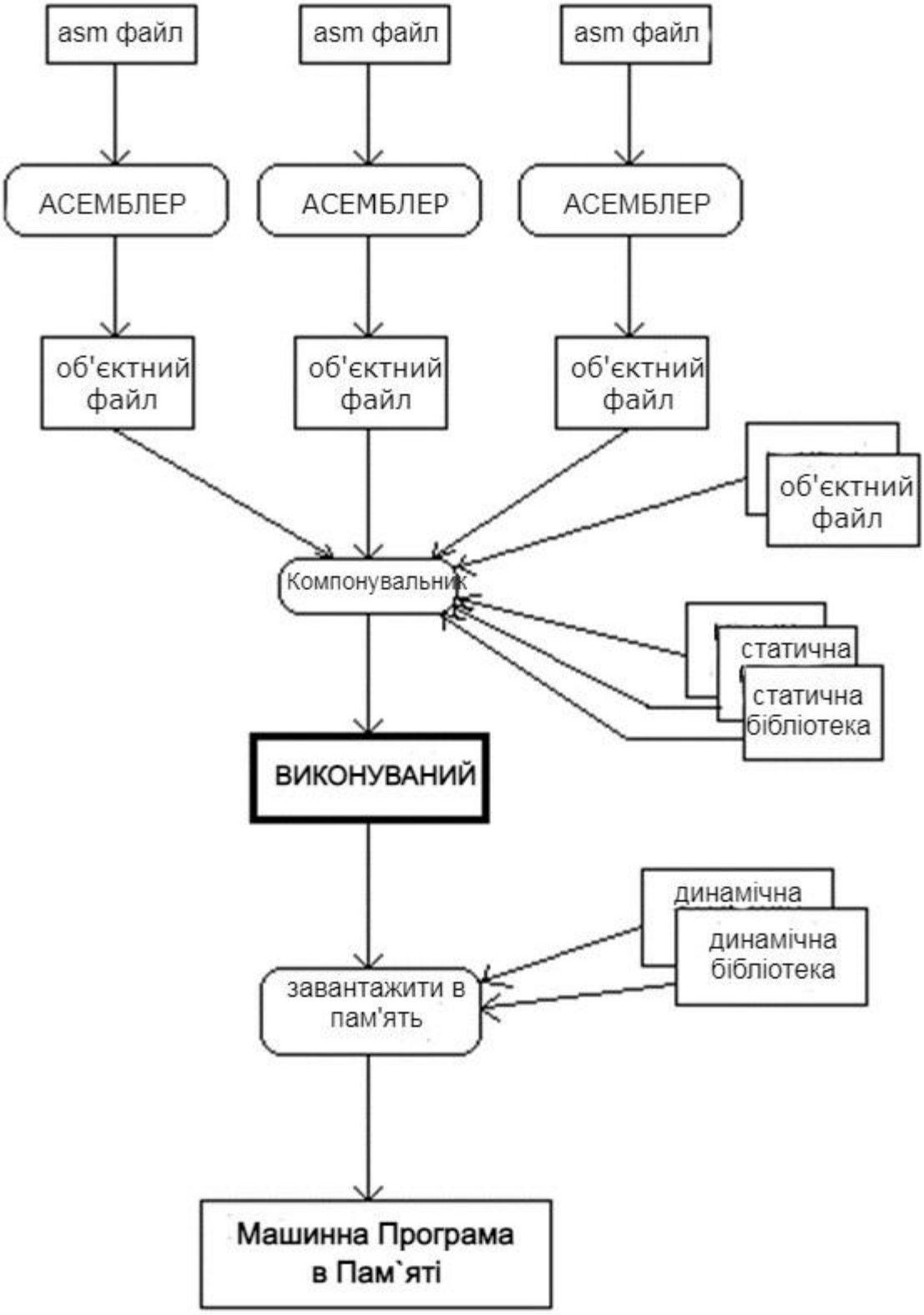

Рис. 1.7 Схема асемблінгу

Проте конкретна реалізація породжує певну специфіку реалізації цієї технології. Зокрема в реалізації MACM 32 основні типи вихідних модулів задаються наступними розширеннями:

.asm - файл асемблер-програми;

.inc - файл з операторами визначення загальних імен, перш за все іменованих констант, а також визначеннями макрокоманд;

У деяких системах програмування мовою асемблерамакроозначення концентрують у файлі з розширенням.mac;

*.Def і *.rc - файли визначень і ресурсні файли, які використовуються в системах програмування на мові високого рівня і підтримувані засобами макрообробки асемблера для забезпечення спільного використання асемблера і мови високого рівня.

Також, особливістю MACM 32 є приховування реального механізму виклику системних функцій Windows API, який базується на перериваних процесора за рахунок використання вбудованої в асемблер директиви «invoke». Ця директива дозволяє записувати виклик функцій, не програмуючи явно передачу параметрів через стек. Типовий варіант виведення на консоль рядка msg довжиною 20 байт при використанні цієї директиви має вигляд (символ "\" задає продовження операторно рядки):

*invoke WriteConsoleA, hStdout, ADDR msg, *

20, ADDR cWritten, 0

Цей оператор еквівалентний С-функції:

WriteConsoleA (hstdout, msg, 20, & cWritten, 0);

Якщо програмувати виклик цієї функції без використання директиви invoke, то буде потрібен наступний фрагмент асемблер-програми:

push 0

push offset cWritten push 20

push offset msg

push hStdout

call WriteConsoleA

Саме такий фрагмент і буде породжений в ході макрообробки рядка з директивою *invoke WriteConsoleA.*

Асемблер може створювати лістинг програми з номерами рядків, адресами змінних, операторами та таблицею перехресних посилань символів і змінних. Нарівні з асемблером використовується програма, яка називається компонувальником (linker), яка об’єднує окремі файли, створені асемблером, в єдину виконувану програму.

При вивченні мови асемблера та розробленні реальних програм цією мовою під операційну систему (ОС) Windows процес виділення пам’яті для збереження кодів команд та чисел виконується автоматизовано.

Етапи створення програми мовою асемблера такі:

- підготовка (або внесення змін) вихідного тексту програми;
- асемблінг програми (отримання об’єктного коду далі будемо називати ООК);
- компонування програми (отримання виконуваного файлу програми);
- завантаження і виконання.

Зазвичай ці етапи циклічно повторюються, оскільки при виявленні помилок на всіх етапах доводиться повертатися до першого етапу і вносити зміни в текст програми для виправлення помилок.

Асемблери бувають двох типів: однофазні та двофазні.

Однофазні асемблери можуть обробляти тільки такі програми, в яких символи з’являються в полі назви до того, як на них дається посилання в полі операндів.

Трансляція програми двофазним асемблером проводиться в два етапи: у першій фазі трансляції асемблер послідовно рахує кожне речення початкової програми, частково її транслює і будує повну таблицю символів; у другій фазі асемблер закінчує трансляцію програми, використовуючи таблицю символів першої фази як вхідну інформацію.

Для обох типів асемблерів символ, записаний у полі операндів, обов’язково має бути визначений у полі назви, інакше буде повідомлення про помилку.

Далі ми зосередимось на вивченні асемблерів, що орієнтуються на команди процесорів IA-32 та працюють у середовищі операційних систем орієнтованих на архітектуру IA-32, зокрема WIN 32 та Linux і легко адаптуються до 64-розрядної архітектури. І хоча ці операційні системи суттєво відрізняються, проте їх об’єднують спільна система команд мікропроцесора та багатозадачність, в якій мікропроцесор може переключатися з задачі на задачу, як показано на Рис:1.8

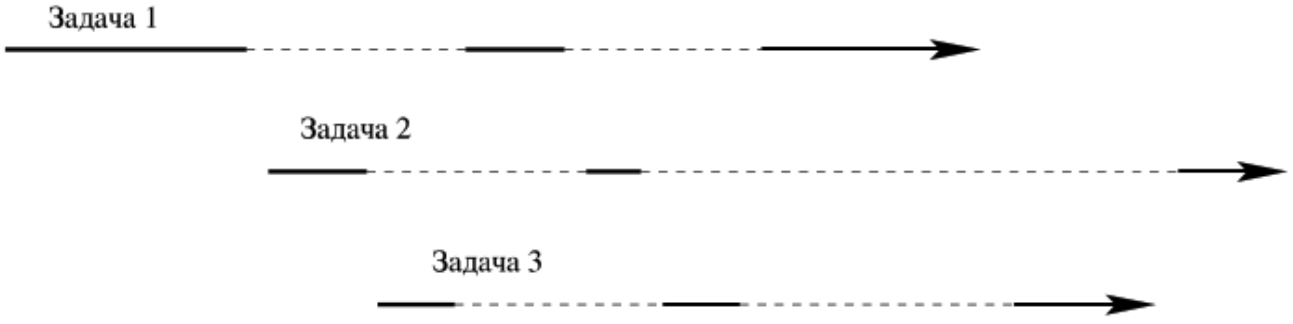

Рис. 1.8 Розподіл часу в багатозадачному середовищі

Питання для самоперевірки

Які дві базові ідеї асемблера?

В якості чином в тексті асемблера оперують регістрами?

З яких основних частин складається текст асемблерною мовою?

Для чого вказуються директиви в тексті асемблера?

Коли потрібно визначати модель памяті?

Що потрібно задавати для визначення памяті для архітектури IA-32?

Як визначати формат виклику?

Який формат файлу після виконання асемблера?

Які файли формуються під час асемблінгу у середовищі MACM 32?

Що виконує директива invoke?

Назовіть етапи створення програми мовою асемблера?

1.4. Середовища програмування в асемблері для IA-32

Як вже зазначалося у цьому розділі існує значна кількість реалізацій асемблера для платформи x86. Вони можуть досить помітно відрізнятися, але їх об’єднує те, що вони мають забезпечити функціонал процесора x86: розрядність, систему адресації та набір команд. Крім того, звичайно, мають підтримувати механізм переривань.

Як базову розрядність ми будемо розглядати тут 32-бітну. Система команд власне є завжди однаковою, бо вона належить не асемблеру і, навіть, не операційній системі, а тільки процесору. І хоча існують деякі відмінності наборі команд асемблера, то це можна розуміти як надбудову в самій організації асемблера, а не якісь нові команди, придумані в даному асемблері. Причому існує те, що фундаментально об’єднує всі без винятку асемблери для x86, так це спосіб взаємодії регістрів процесора з пам’яттю, що власне є характерною рисою архітектури CISC: наявність великої кількості методів адресації пам'яті з можливістю безпосередньої роботи з операндами в основній пам'яті комп'ютера. Причому шляхом виконання команд двоадресним способом.

Крім того, існує ще один аспект, який дає спільну основу для асемблерів, які далі розлянемо – нотація, тобто спосіб позначення виразів мови асемблер. У цьому посібнику ми будемо використовувати *синтаксис Intel*, але потрібноі знати і про *синтаксис AT&T,* оскільки він є стандартним для Linux, і тому багато онлайн-ресурсів використовують його.

Архітектура x86 (як 32-, так і 64-розрядна) має два альтернативні синтаксиси, доступні для нього. Деякі асемблери можуть працювати лише з одним або іншим, тоді як деякі можуть працювати з обома.

У таблиці 1.4 підсумовано основні відмінності між синтаксисом Intel і AT&T:

Таблиця1.4 Порівняння синтаксів Intel і AT&T

	Intel	**AT&T**
Коментарі	;	//
Інструкції	Без тегів add	Позначено розмірами операндів:addq
Регістри	eax, ebx, та інші	%eax, %ebx та інші
Безпрсередні	0x100	$0x100
Непрямий	[eax]	(%eax)
Загальні непрямі	[base + reg + reg * scale + displacement]	displacement(reg, reg, scale)

Зауважте, що в узагальненому непрямому форматі синтаксис Intel `base + displacement` дещо зручніший, оскільки коди операцій дійсно дозволяють лише одне безпосередне зміщення. У синтаксисі AT&T ви маєте зробити це самостійно, використовуючи результат суми як третє значення в дужках.

Основна відмінність між синтаксисом Intel і AT&T полягає в тому, що Intel залишає *розміри* операндів інструкцій неявними, тоді як AT&T робить їх явними, додаючи суфікси до імені інструкції. Прикладом є `add` інструкція, яка додає два значення разом:

add eax, ebx ;

Як зазначено, у синтаксисі Intel це додає два 32-розрядні регістри (обидва `eax`і `ebx`є 32-розрядними) і зберігає результат у `eax`. Інструкції Intel мають регістр призначення першим, а регістр джерела другим. Інструкції AT&T навпаки - регістр призначення другим, а регістр джерела першим.Оскільки і джерело, і місце призначення є 32-розрядними, асемблер знає, що йому потрібно виконувати 32-розрядну `add` інструкцію. У синтаксисі AT&T така ж інструкція буде

addq %ebx, %eax ;

Імена регістрів у синтаксисі AT&T мають префікс `%` та змінено порядок операндів, але найбільша різниця полягає в `q` суфіксі в `add` інструкції. Символ `q` – це скорочення від «квадрослова», тобто 32-бітове значення. Суфікс потрібен, щоб

повідомити асемблеру про розмір операндів; він не виведе правильну інструкцію для використання лише з розмірів регістру.

Інструкції, які можуть змішувати операнди різних розмірів, матимуть два (або іноді три) суфікси. Наприклад, щоб додати молодші 16 бітів `ebx`у `eax`, ми б зробили:

add eax, bx :

в Intel, але в AT&T це буде:

addzqd %bx, %eax ;

де `d` означає «подвійне слово». Указує `z`, що це додавання без знака (нульове розширення). Якби ми хотіли додати знак, ми б використали `s`(розширення знака).

Питання для самоперевірки

Перелічіть різниці між синтаксисом AT&T та Intel

1.4.1 Макроасемблер MASM 32

Спочатку розглянемо **макроасемблер masm32.** Фірма Microsoft давно, ще з часів DOS розробила і підтримує свій продукт під назвою Microsoft Macro Assembler. Потім він був перероблений для операційної системи Windows, причому слід відзначити програміста Стівена Хатчінсона (Сідней, Австралія), який вніс значний вклад у його модернізаціюдля сучасних обчислювальних можливостей сучасних Windows.

Текст програми мовою асемблера записується в один чи декілька текстових файлів. Імена файлів і їх розширення можуть бути будь-які, але прийнято використовувати розширення *.asm. Ці файли є текстовими, їх можна підготувати за допомогою будь-якого текстового редактора, наприклад блВікнота або NotePad, і зберігати у вигляді звичайних файлів у форматі ASCII. Використання складних текстових процесорів типу WinWord не рекомендується, тому що вони можуть ввести множину непотрібних прихованих службових символів, які можуть призвести до помилок трансляції.

Найпростіше при підготовці тексту можна використати саме середовище Masm32.

Структура програми під Win32 може бути такою:

TITLE <ім'я програми

.386; директива визначення типу мікропроцесора

.model flat; призначення плоскої моделі пам'яті

.stack

.data; директива визначення даних

; дані, які визначені- ініціалізовані

.data? ; неініціалізовані дані

;

.code; директива початку коду програми

label:; мітка початку програми

; <Code>; команди асемблера

ret; повернення управління ОС

end label; закінчення програми

Директива .386 задає область дозволених інструкцій. Можна вказати, наприклад, .486,.586 і т. д., Проте наразі це зайве - інструкцій і386 нам цілком достатньо, але може бути недостатнім для реальних задач, в яких треба писати код з врахуванням команд нових процесорів.

Модель пам'яті **flat** встановлює роботу процесора в захищеному режимі і відповідну йому схему адресації. Директива OPTION с атрибутом casemap встановлює чутливість компілятора до регістру букв. Це необов'язкова опція, проте допомагає програмісту контролювати імена ідентифікаторів і функцій API. Директива INCLUDE підключає файл з прототипами функцій. Зазвичай ці файли мають те ж ім'я, що і динамічна бібліотека (kernel32.dll).

Директива INCLUDELIB підключає спеціальним чином скомпоновані бібліотеки (з розширенням .lib), що містять опис функцій. Вони потрібні тільки на етапі компіляції; під час виконання програми викликаються вже динамічні бібліотеки. Знову розробники подбали про те, щоб заголовки (.inc) описували прототипи скомпонованих функцій в бібліотеках (.lib).Як видно з прикладу, "класичний" шаблон 32-розрядного програми містить область даних (яка визначається директивою .data), область стека (директива .stack) і область програмного коду (директива .code). Може трапитися так, що 32-розрядній програмі мовою асемблера потрібно кілька окремих сегментів даних і/або коду. В цьому випадку розробник може створити їх за допомогою директиви SEGMENT. Директива SEGMENT визначає логічний сегмент і може бути описана наступним чином:

Ім’я SEGMENT список атрибутів

. . .

Ім’я ENDS Зараз ми розглянемо програмний код 32-розрядної процедури на асемблері, в якій використовуються два логічних сегмента даних. Процедура виконує копіювання рядка src, що знаходиться в сегменті даних data1, в область пам'яті dst в сегменті даних data2 і містить один логічний сегмент програмного коду (code segment).

Наразі нема необхідності робиратися з принципами роботи процедур:

в даному випадку нас буде цікавити тільки код всередині процедури _seg_ex (команди, що знаходяться між директивами _seg_exproc і _seg_exendp). Оригінальний текст програмного коду представлений в лістингу.

```
.586
.model flat
 option casemap:none
data1 segment
src DB "Test STRING To Copy"
len EQU$-src
data1 ends
data2 segment public
dst DBlen+1 DUP('+')
data2 ends
code segment
_seg_ex proc
 assume CS:FLAT,DS:FLAT, SS:FLAT, ES:FLAT, FS:ERROR, GS:ERROR
 mov ESI, offset data1
 mov EDI, offset data2
 cld
 mov CX, len
 rep movsb
 mov EAX, offset data2
 ret
_seg_ex endp
code ends
end
```

Далі, щоб цей текст був перетворений у виконуваний процесором код у програмному середовищі середовощі, він має пройти три стадії:

- ассемблінг- перетворення в об’єктний код;
- лінкування - під’єднання бібліотек;
- компонування виконуваного файлу.

Подробиці цього процесу описані в Додатку А.

1.4.2 Команди асемблера MASM 32

Програма, написана мовою асемблераMASM 32, як і для інших асемблерів може складатися із декількох модулів. У кожному модулі можуть бути визначені один або кілька сегментів даних, стека та коду. Програма повинна включати один головний модуль, з якого починається виконання. Модуль може містити програмні сегменти, сегменти даних та стека, оголошені за допомогою відповідних директив.

Оголошення сегментів вимагає попереднє визначення моделі пам'яті. Основною моделлю пам'яті дляWin 32 є модель flat, в якій всі секції начеб то розміщені в одному величезному сегменті розміром 4Гб як дані, так і код. Ще однію особливістю є необхідність перед директивою .model flat розміщення з директиви моделі процесора: .386, .486, .586, .686. Цим самим програміст показує на які набори команд він розраховує.

.686p
.model flat, stdcall
option casemap: none
.data
ініціалізовані дані
data?
неініціалізовані дані
.const
константи
.code
мітка
код
end мітка

Як ми бачимо у приведеному коді можна використовувати практично всі команди останніх моделей процесорів x86. Але даному розділі обмежимось описом базового набору асемблерних команд інтелівської архітектури.

Команди цілочисельної арифметики IA-32

Процесори сімейства IA-32 підтримують арифметичні операції над однобайтовими, двобайтовими і чотирьохбайтовими цілими числами.

Розмір операндів при цьому визначається:

• обсягом регістра, що зберігає число - якщо хоча б один операнд знаходиться в регістрі;

• розміром числа, заданим директивою визначення даних;

• спеціальними описувачем.

Наприклад, BYTE PTR (байт), WORD PTR (слово) і DWORD PTR (подвійне слово), якщо жоден операнд не перебуває у регістрі і розмір операнда відмінний від розміру, визначеного директивою визначення даних.

Команда пересилання даних *mov* - пересилає число розміром 1, 2 або 4 байти з джерела в приймач:

mov Приймач, Джерело

Правильні варіанти:

mov reg, reg mov mem, reg mov reg, mem mov mem, imm mov reg, imm mov r/m16, sreg mov sreg, r/m16

Приклади:

mov AX, BX

mov ESI, 1000

mov 0[DI], AL

mov AX, code

mov DS, AX

Команда переміщення і доповнення нулями movzx - при переміщенні значення джерела поміщається в молодші розряди, а в старші заносяться нулі:

movzx Приймач, Джерело

Правильні варіанти:

movzx r16 / r32, r / m8 movzx r32, r / m16

приклади:

а) movzx EAX, BX

б) movzx SI, AH

Команда переміщення і доповнення знаковим розрядом movsx - команда виконується аналогічно, але в старші розряди заносяться знакові біти:

movsx Приймач, Джерело

До речі команда *mov* в IA-32 має специфічний привілей – можливість звернення до регістрів управління, зокрема до CR0. Наприклад призавантаженні оперативної системи процесор спочатку працює в режимі реальних адрес і тільки на певному етапі відбувається переключення в захищений режим. Перехіду захищений режим відбувається після виставлення молодшого біта в регістрі CR0 (MSW), потім виконати далекий міжсегментний перехід (FAR JUMP) для очищення кешу команд:

mov EAX,CR0; Читання cr0

and EAX,0fffffffeh ; Скидання біта 0

mov CR0,EAX

JMP DWORD PTR [PMEntry]

Команда обміну даних

XCHG операнд1, Операнд 2

Правильні варіанти:

xchg reg, reg xchg mem, reg xchg reg, mem

Команди додавання – додає операнди, а результат поміщає за адресою першого операнда. На відміну від ADD команда ADC додає до результату значення біта прапора перенесення CF.

ADD операнд1, операнд2

ADC операнд1, операнд2

Правильні варіанти:

add reg, reg

add mem, reg

add reg, mem

add mem, imm

add reg, imm

Команди віднімання - віднімає з першого операнда другий і результат поміщає за адресою першого операнда. На відміну від SUB команда SBB віднімає з результату значення біта прапора перенесення CF. Правильні варіанти ті ж, що і у додавання.

SUB операнд1, операнд2

SBB операнд1, Операнд 2

Команди додавання / віднімання одиниці(Інкремент/декремент)

INC reg / mem DEC reg / mem

приклади:

inc AX

dec byte ptr 8 [EBX, EDI]

Команди множення

MUL <операнд2>

IMUL <операнд2>

Правильні варіанти:

mul / imul r | m8;

*AX = AL * <операнд2> mul / imul r | m16;*

*DX: AX = AX * <операнд2> mul / imul r | m32;*

*EDX: EAX = EAX * <операнд2>*

Другим операндом не можна вказати безпосереднє значення !!!

Регістри першого операнда в команді не вказуються. Місцезнаходження і довжина результату операції залежить від розміру другого операнда.

приклад:

mov AX, 4
imul word ptr A;
*DX: AX: = AX * A*
команди ділення

DIV <операнд2> IDIV <операнд2>

Правильні варіанти:

div / idiv r | m8;
AL = AX: <операнд2>, AH - залишок div / idiv r | m16;
AX = (DX: AX): <операнд2>, DX – залишок div / idiv r | m32;
EAX = (EDX: EAX): <операнд2>, EDX - залишок
другим операндом не можна вказати безпосереднє значення !!!

приклад:

mov AX, 40 cwd
idiv word ptr A;
AX: = (DX: AX): A

Приклад лінійної програми

Розробити застосунок, що обчислює X = (A + B) (B-1) / (D + 8).

.DATA
A SWORD 25
B SWORD -6
D SWORD 11
.DATA?
X SWORD?
.CODE

Start: mov CX, D
add CX, 8; CX: = D + 8
mov BX, B
dec BX; BX: = B-1
mov AX, A
add AX, D; AX: = A + D
*imul BX; DX: AX: = (A + D) * (B-1)*
idiv CX; AX: = (DX: AX): CX
mov X, AX

Команди передачі управління

Кожна команда лінійної програми виконує деякі дії з перетворення або пересилання даних, після чого мікропроцесор передає управління наступній команді. Але дуже мало програм працюють у такий послідовний спосіб. Зазвичай у програмі є точки, в яких потрібно ухвалити рішення про те, яка команда виконуватиметься наступною. Це рішення може бути

- безумовним - у цій точці необхідно передати управління не тій команді, яка йде наступною, а іншою, яка знаходиться на деякій відстані від поточної команди;
- умовним — рішення про те, яка команда виконуватиметься наступною, приймається з урахуванням аналізу деяких умов чи даних.

За принципом дії команди мікропроцесора, що забезпечують організацію переходів у програмі, можна розділити на чотири групи:

- Команди безумовної передачі управління:
- команда безумовного переходу ;
- виклику процедури та повернення з процедури ;
- виклику програмних переривань та повернення з програмних переривань.
- Команди умовної передачі управління :
- команди переходу за результатом команди порівняння cmp ;
- команди переходу за станом певного прапора ;
- команди переходу за вмістом регістру ecx/cx .
- Команди управління циклом :
- команда організації циклу з лічильником ecx/cx ;
- команда організації циклу з лічильником ecx/cx з можливістю дострокового виходу з циклу за умовою .

Безумовні переходи

Попереднє обговорення виявило деякі деталі механізму переходу. Команди переходу модифікують регістр покажчика команди **eip/ip** і, можливо, сегментний регістр коду **cs** .

Команда безумовного переходу jmp

Синтаксис команди безумовного переходу **jmp [модифікатор] адреса_переходу** - безумовний перехід без збереження інформації про точку повернення.

Адреса_переходу є адресу як мітки чи адресу області пам'яті, де знаходиться покажчик переходу. Загалом у системі команд мікропроцесора є кілька кодів машинних команд безумовного переходу **jmp** . Їх відмінності визначаються дальністю переходу та способом завдання цільової адреси. Дальність переходу визначається місцем розташування операнда адрес_переходу. Ця адреса може знаходитися в поточному сегменті коду або в іншому сегменті. У першому випадку перехід називається внутрішньосегментним, або близьким, у другому міжсегментним, або далеким. Внутрішньосегментний перехід передбачає, що змінюється лише вміст регістру **eip/ip** .

Можна виділити три варіанти внутрішньосегментного використання команди jmp:

- прямий короткий;
- прямий;
- непрямий.

Умовні переходи

Мікропроцесор має 18 команд умовного переходу. Ці команди дозволяють перевірити:

- відношення між операндами зі знаком (“більше – менше”);
- відношення між операндами без знаку (“вище – нижче”)2;
- стану арифметичних прапорів zf, sf, cf, of, pf (але не af)

Для того щоб прийняти рішення про те, куди буде передано управління командою умовного переходу, попередньо має бути сформована умова, на підставі якої прийматиметься рішення про передачу управління. Джерелами такої умови можуть бути:

- будь-яка команда, що змінює стан арифметичних прапорів;
- команда порівняння **cmp** , що порівнює значення двох операндів;
- стан регістра ecx/cx.

Команда порівняння cmp

Команда порівняння **cmp** має цікавий принцип роботи. Він абсолютно такий самий, як і у команди віднімання.

sub операнд_1,операнд_2 . Команда **cmp** так само, як і команда **sub** , виконує віднімання операндів і встановлює прапори. Єдине, чого вона не робить - це запис результату віднімання на місце першого операнда.

Синтаксис команди cmp:

cmp операнд_1,операнд_2 (compare) - порівнює два операнди і за результатами порівняння встановлює прапори. Прапори, що встановлюються командою **cmp** , можна аналізувати спеціальними командами умовного переходу.

Команди умовного переходу та прапори

Мнемонічне позначення деяких команд умовного переходу відображає назву прапора, з яким вони працюють, і має таку структуру: першим йде символ “ **j** ” (jump, перехід), другим – або позначення прапора, або символ заперечення “ **n** ”, після якого стоїть назва прапора . Така структура команди відображує її призначення. Якщо символу “n” немає, то перевіряється стан прапора, і він дорівнює 1, проводиться перехід на мітку переходу. Якщо символ “n” присутній, то перевіряється стан прапора на рівність 0, й у разі успіху відбувається перехід на мітку переходу.

Мнемокоди команд, та умови переходів легко вирахувати знаючи назв назви прапорів. Наприклад команда jc реагує на зміну прапора перенесення cf, а команда jz на зміну прапора нуля

При програмуванні мовою асемблера використовується модель пам'яті Flat. У цій моделі вважається, що всі сегменти програми (кодовий, сегменти даних і стека) починаються з нульової адреси і мають розмір, що дорівнює доступній пам'яті комп'ютера. Таким чином вони начебто накладаються на загальний простір адрес. Реальний поділ адресного простору кожного сегмента здійснюється за допомогою розміщення коду, даних і стека з різними зсувами щодо початку сегментів.

Та ж модель використовується при компіляції програм з мов високого рівня в основних програмних середовищах (наприклад, Delphi і Visual C ++), оскільки вона істотно спрощує адресацію програми.

Стекові команди

Стек (від англ. Stack) - спеціально відведене місце в пам'яті для зберігання тимчасових даних. Він підпорядковується наступним правилам:

- *LIFO* (*Last In First Out*), який має на увазі, що елемент, який був поміщений на стек останнім, буде першим звідти вилучений.
- Стек росте у бік початку адресного простору (ще кажуть, що стек росте вниз).

- Мінімальна одиниця, яка може бути видалена/вміщена зі\у стек(а) - 16 біт (2 байти).
- Максимальна одиниця, яка може бути видалена / поміщена зі/у стек (а) - 32 біта (4 байти).

Як видно з Рис 1.9 при розміщенні чогось у стек він збільшується вниз.

Рис.1.9 Улаштування стека IA - 32

Вказівник стека (*ESP - Stack Pointer*) містить адресу останнього елемента стека, що слідує за останнім елементом. Цю частину стека також називають вершиною стека (від англ. *TOS – Top Of Stack*).

Коли щось міститься у стеку, процесор декрементує значення в регістрі ESP і записує значення на вершину стека. У випадку, коли необхідно щось видалити зі стека, процесор спочатку копіює значення з вершини стека, а потім вже інкрементує значення в регістрі ESP.

Щоб процесор зміг зрозуміти, що йому потрібно зберегти на стек, використовується інструкція **push** в асемблерному коді, у разі видалення "витягування" значення зі стека **-pop.**

Інструкція push

Її синтаксис може відрізнятися від однієї мови асемблера до іншого, проте суть її залишається незмінною - розміщення будь-якого значення у стек.

PUSH r/m16

PUSH r/m32

PUSH r16

PUSH r32

PUSH imm8

PUSH imm16

PUSH imm32

Префікс *r/m*(від англ. *register/memory*) означає, що значення, яке необхідно помістити на стек знаходиться в пам'яті, яка в свою чергу знаходиться в регістрі, наприклад, в регістрі лежить значення 0x87654321- адреса пам'яті, за якою зберігається значення 0x11223344 відповідно на стек буде вміщено значення 0x11223344.

Префікс *r*(від англ. *Register*) означає, що значення, яке необхідно помістити на стек, знаходиться в регістрі.

Префікс imm(від англ. *immediate*), тобто. значення, яке безпосередньо передається інструкції як параметр.

Постфікси 8, 16, 32 означають скільки біт містить, що передається інструкції, значення, яке, у свою чергу, зазвичай називають *операндом.*

На даний момент може виникнути питання від того, що, як писалося вище, мінімальна одиниця, яка може бути поміщена на стек - 16 біт, але в синтаксисі інструкції push imm8, що говорить про те, що операндом інструкції може бути і 8-бітове значення. Насправді 8-бітові значення доповнюються до 16-бітних, т.щ. стек вирівняний по 16 біт, але це має значення і для знакових типів, які застосовують друге доповнення до двох.

Інструкція pop.

Синтаксис інструкції pop аналогічний тому, який використовується в інструкції push, за винятком того, що значення видаляється зі стека і поміщається в операнд інструкції.

POP r/m16

POP r/m32

POP r16

POP r32

І навіть, з очевидних причин, операндом інструкції pop може бути immediate value, бо. ми не можемо зберегти щось туди.

Регістри для керування стеком

Вже був згаданий один з регістрів, який дозволяє керувати стеком - *ESP* (*Stack Pointer*), мабуть, це найважливіший регістр, який зберігає вказівник на вершину стека, однак є ще кілька регістрів, пов'язаних зі стеком:

- *SS (Stack Segment)* - регістр, що вказує на певний сегмент пам'яті, в якому знаходиться сам стек, тільки одним сегментом стека можна маніпулювати в конкретний момент часу, цей регістр задіюється процесором для всіх операцій, пов'язаних зі стеком.
- *EBP (Base Pointer)* - регістр, показує поточний кадр, тобто. на початок стека для конкретної процедури\функції, зазвичай використовується для адресації локальних змінних та аргументів процедури\функції.

Питання для самоперевірки

- Над якими даними процесори сімейства IA-32 підтримують арифметичні операції?
- Які правильні варіанти команда переміщення і доповнення нулями?
- Які особливості виконання команди XCHG?
  - Що виконують команди INC, DEC? Як впливають ці команди на арифметичні прапори?
- Яка особливість формату команди множення ?
  - Для чого призначені команди пересилання даних? Звідки і куди пересилаються дані по цим командам?
- Як влаштований стек?
- Який формат команд для роботи зі стеком?
- Що змінюється в регістрі ESP при запису значення у вершині стеку?
- Які регістри підтримують керування стеком?

1.5 Організація виклику функцій(процедур)

Так само, як і в C, в асемблері передбачені виклики функцій, що робить структуру програми нелінійною, але в цілком інший спосіб ніж інструкції безумовного переходу. Залежно від моделі програмування, виклик функції може бути near(під час якого значеннння поточного IP/ EIP надсилається в стек і функція завершується за допомогою RET) та far(під час якого IP/EIP і CS надсилаються в стек, а функція виходить за допомогою RETF). Існує ще один спосіб зміни лінійного потоку команд, відомий як переривання(interrupt), але про нього буде мова дещо пізніше.

1.5.1 Виклик

Функція(процедура) викликається з програми за допомогою команди CALL, яка може виникнути будь-де в програмі, що робить виклик, і завершується інструкцією RET, яка повертає керування програмі, що викликає. Керування повертається до рядка в програмі, яка безпосередньо слідує за комадою CALL у програмі, що викликає. Викликаюча процедура може передавати параметри або адреси параметрів до стеку, які будуть використовуватися викликаною процедурою, або передавати параметри, розміщуючи їх у регістрах загального призначення, до яких може отримати доступ викликана процедура.

Інструкція CALL використовується для виклику процедури або в поточному сегменті коду (ближній виклик), або в іншому сегменті коду (дальній).

Синтаксис для інструкції CALL показано наступний:

CALL procedure_name_label

Близький виклик переміщує значення оновленого регістра (E)IP у стек; таким чином, він вказує на інструкцію, яка слідує безпосередньо за інструкцією CALL - ця інструкція виконується після повернення від викликаної процедури до програми, що викликає. Потім відбувається розгалуження до адреси призначення (цільової). Адреса призначення для найближчого виклику може бути абсолютним або відносним зміщенням.

Абсолютне зміщення вказує на адресу, яка є зміщенням від основи поточного сегмента коду. Абсолютний зсув отримується опосередковано через регістр загального призначення або з місця пам’яті, як показано нижче, використовуючи регістр EBX для непрямого виклику. Вміст регістра EBX містить зміщення в пам'яті.

CALL [EBX]

Абсолютний зсув зберігається в регістрі (E)IP; якщо атрибут розміру операнда дорівнює 16, то старша половина регістра EIP скидається. Відносне зміщення - це зміщення зі знаком, яке додається до регістру (E)IP і зазвичай визначається як мітка, яка кодується як безпосереднє значення. Регістр CS не змінюється для ближніх викликів, оскільки цільова адреса знаходиться в поточному сегменті коду.

Далекий виклик переміщує регістр CS у стек і оновлене значення регістра (E)IP у стек. Потім дальній виклик завантажує регістр CS із селектором сегментів викликаної процедури та завантажує зміщення викликаної процедури в регістр (E)IP. Цільова адреса для викликаної процедури - це дальня адреса, отримана безпосередньо з вказівника в інструкції або опосередковано через область пам'яті.

1.5.2 Повернення

Інструкція RET передає керування від викликаної процедури до інструкції, що йде безпосередньо після інструкції CALL у викликаючій процедурі. Під час виконання внутрішньосегментного повернення, процесор дістає значення на вершині стека для регістра EIP, додає додаткове безпосереднє значення до вказівника стека, а потім продовжує виконання процедури виклику. Під час виконання далекого міжсегментного повернення процесор дістає значення на вершині стеку в регістр EIP, потім виносить значення на вершині нового стеку в регістр CS і додає необов’язкове безпосереднє значення до вказівника стека, а потім продовжує виконання процедури виклику. Безпосереднє необов'язкове значення використовується для видалення параметрів зі стеку, які були розміщені в стеку процедурою виклику. Синтаксис для інструкції RET показано нижче.

RET необов'язкове_безпосереднє_значення

Параметри (аргументи) можна передати в підпрограму (процедуру) за допомогою стека, регістрів загального призначення або шляхом розміщення адрес параметрів у регістрах загального призначення. Оскільки вміст регістрів загального призначення не зберігається перед виконанням інструкції CALL, усі шість РЗП - EAX, EBX, ECX, EDX, ESI та EDI - можна використовувати для передачі параметрів викликаній процедурі. Однак регістри ESP і EBP не можна використовувати для передачі параметрів. Програма, що викликає, може зберегти РЗП у стеку, у пам’яті або в сегменті даних перед викликом процедури. Передача параметрів за допомогою стека. Сегмент коду нижче ілюструє використання стека для передачі параметрів викликаній процедурі. Викликаюча програма передає augend і addend до процедури, поміщаючи їх у стек. Потім викликана процедура виконує операцію додавання та повертає суму викликаючій програмі в регістрі EAX

```
PUSH AUGEND                 ADD_PROC PROC NEAR
PUSH ADDEND                 PUSH EBP
CALL ADD_PROC               MOV EBP, ESP
.                           MOV EAX, [EBP + 12]
.                           ADD EAX, [EBP + 8]
.                           POP EBP
                            RET 8
                            ADD_PROC END
```

Розміщення параметрів та регістрів у стеку показано на Рис.10

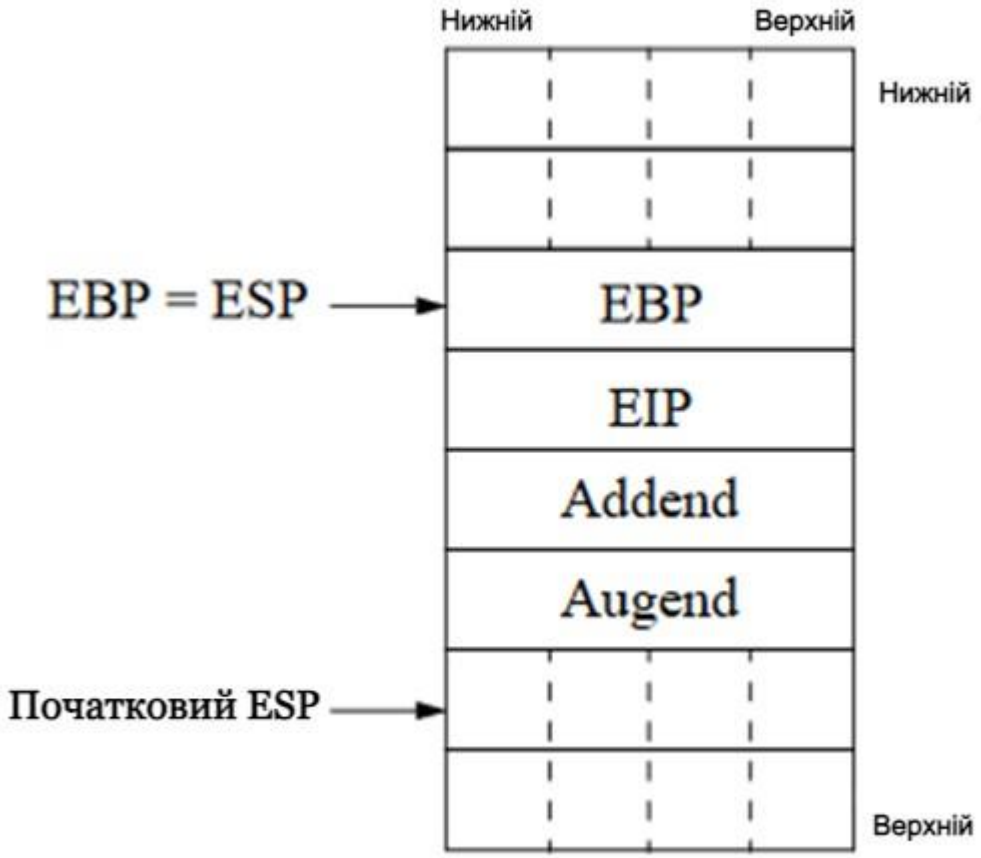

Рис. 1.10 Розміщення параметрів та регістрів в стеку

Викликаюча програма поміщає augend і addend у стек, а потім викликає процедуру ADD_PROC. Процедура є близькою процедурою; тому в стек поміщається лише регістр EIP. Викликана процедура надсилає регістр EBP у стек - регістр EBP буде використано для доступу до augend і addend.

Потім значення регістра ESP, яке вказує на розташування EBP у стеку, переміщується до EBP. Регістр EBP тепер можна використовувати для доступу до даних у стеку.

Потім augen переміщується до регістру EAX командою, наведеною нижче. Augend розташований на 12 байтів вище адреси EBP (ESP). Тому до вмісту регістру EBP додається значення 12.

MOV EAX, [EBP + 12]

Потім інструкція ADD додає вміст регістра EAX (augend) до вмісту стека в місці EBP + 8 (add) і поміщає суму в регістр EAX.

Потім початковий вміст регістра EBP витягується зі стеку та зберігається в EBP. Після цього виконується інструкція найближчого повернення, яка видаляє EIP зі стеку; register ESP тепер вказує на кінець додавання.

Щоб відновити початкове значення вмісту регістра ESP, після повернення до процедури виклику до регістра ESP безпосередно додається значення вісім. Викликана процедура розмежована директивами PROC і ENDP, обидві з яких містять назву викликаної процедури.

Дещо складніше використання стеку для виклику функцій проходить у випадку, коли для передачі параметрів використовуються регістри, а також у випадку непрямої адресації.

Функції(процедури), як окремі ділянки коду, зазвичай пишуть один раз для багаторазового використання як для даної програми, так і для використання в інших програмах. Що дає можливість створювати бібліотеки, які потім можуть використовувати інші програмісти. Очевидно, що це можливо, якщо програмісти приймають загальну угоду про *виклики*.

1.5.3 Угода про виклики

Тобто ця угода – це протокол про те, як викликати та повертатися з підпрограм. Наприклад, маючи набір правил угоди про виклики, програмісту не потрібно вивчати визначення підпрограми, щоб визначити, як параметри мають бути передані цій підпрограмі. Крім того, враховуючи набір правил угоди про виклики, компілятори мов високого рівня можна змусити дотримуватися цих правил, таким чином дозволяючи кодованим вручну підпрограмам мови асемблера та підпрограмам мови високого рівня викликати одна одну.

На практиці можливе багато умов викликів. Однак найбільш поширені тільки дві.

1.Угода в стилі мови Pascal (або нотація stdcall).

Стек відновлює викликану функцію, використовуючи команду ret m, де m=n*4. Ця нотація використовується в більшості функцій API Win32. Тут діє просте правило: "зліва направо, знизу-вгору". При виклику функції команди call, як відомо, у стек вкладається адреса повернення. При відсутності параметрів при відновленні з функцій (команда ret, правильніше було б сказати retn (повернення в межах одного сегмента)) адреса витягується зі стеку, тобто. положення вершини стеку відновлюється. Однак, якщо в стек попередньо поклали параметри (n - кількість), то після виклику параметрів функції вказівник стека залишається зміненим на величину 4*n. При багаторазовому

виклику функції з параметром стек може бути вичерпано, що приведе до краху програми. Для повернення стека в вихідний стан використовуються два механізми. Угода про виклики stdcall є різновидом угоди про виклики Pascal, у якій викликана особа відповідає за очищення стека, але параметри надсилаються в стек у порядку справа наліво, як для угоди про виклики cdecl. Регістри EAX, ECX і EDX призначені для використання в межах функції. Повернуті значення зберігаються в регістрі EAX. Угода stdcall - це стандартна угода про виклики для Microsoft Win32 API.

2. Угода в стилі мови C (або нотація cdecl). Після виклику команди call, для відновлення стека застосовується команда add esp,m, де m=4*n. Подібний механізм, який використовується для функцій зі змінною кількістю параметрів, наприклад, wsprintf (функція здійснює копіювання форматної строки в буфері, підставляючи туди значення параметрів). При цьому порядок відправки параметрів в стек інший. Тут працює правило "справа наліво, знизу-вгору".

Ми будемо використовувати широко поширену угоду про виклики мови C. Дотримання цієї угоди дозволить вам писати підпрограми мови асемблера, які можна безпечно викликати з коду C (і C++), а також дозволить вам викликати функції бібліотеки C з коду мови асемблера.

Угода про виклики C значною мірою базується на використанні стека, що підтримується апаратним забезпеченням. Він базується на інструкціях push, pop, call і ret. Параметри підпрограми передаються в стек. Регістри зберігаються в стеку, а локальні змінні, які використовуються підпрограмами, розміщуються в пам'яті стека. Переважна більшість процедурних мов високого рівня, реалізованих на більшості процесорів, використовують подібні угоди про виклики.

Угода про виклики розбита на два набори правил. Перший набір правил використовується тим, хто викликає підпрограму(абонент), а другий набір правил дотримується власником підпрограми (викликаним). Слід підкреслити, що помилки в недотриманні цих правил швидко призводять до фатальних програмних помилок, оскільки стек залишиться в невизначеному стані.Таким чином слід ретельно реалізовувати угоду про виклик у ваших власних підпрограмах.

1.5.4 Правила абонента

Щоб здійснити виклик субмаршрутизації, абонент повинен:

1. Перед викликом підпрограми абонент, який викликає, повинен зберегти вміст певних регістрів, які позначено як *caller-saved*. Регістри, збережені абонентом - це

EAX, ECX, EDX. Оскільки викликаній підпрограмі дозволено змінювати ці регістри, якщо програма, що викликає, покладається на їхні значення після повернення підпрограми, вона повинна перемістити значення в цих регістрах у стек (щоб їх можна було відновити після повернення підпрограми).

2. Щоб передати параметри підпрограмі, помістіть їх у стек перед викликом. Параметри слід вводити в зворотному порядку (тобто останній параметр спочатку). Оскільки стек зростає, перший параметр буде зберігатися за найнижчою адресою (ця інверсія параметрів історично використовувалася, щоб дозволити функціям передавати змінну кількість параметрів).

3. Щоб викликати підпрограму, використовуйте інструкцію виклику. Ця інструкція розміщує адресу повернення поверх параметрів у стеку та переходить до коду підпрограми. Це викликає підпрограму, яка має дотримуватися наведених нижче правил виклику.

Після повернення підпрограми (відразу після інструкції викликуабонент може очікувати знайти повернене значення підпрограми в регістрі EAX. Щоб відновити стан машини, абонент повинен:

1. Видалити параметри зі стеку. Це відновлює стек до його стану до виконання виклику.

2. Відновити вміст регістрів, збережених абонентом (EAX, ECX, EDX), вилучивши їх зі стеку. Виклик може припустити, що жодні інші регістри не були змінені підпрограмою.

Приклад. Наведений нижче код показує виклик функції, який відповідає правилам виклику. Виклик викликає функцію *_myFunc*, яка приймає три цілі параметри. Перший параметр знаходиться в EAX, другий параметр – константа 216; третій параметр знаходиться в пам'яті *var*.

```
push [var] ; Спершу натисніть останній параметр
push 216 ; Натисніть другий параметр
push eax; Натисніть перший параметр останні
викликати _myFunc ; Виклик функції (припустимо іменування C)
add esp, 12
```

Зауважте, що після повернення виклику абонент очищає стек за допомогою інструкції add. Ми маємо 12 байтів (3 параметри * 4 байти кожен) у стеку, і стек зростає. Таким чином, щоб позбутися параметрів, ми можемо просто додати 12 до вказівника стека.

Результат, створений *_myFunc*, тепер доступний для використання в регістрі EAX. Значення регістрів, збережених абонентом (ECX і EDX), можливо, були змінені. Якщо абонент використовує їх після виклику, йому потрібно було б зберегти їх у стеку перед викликом і відновити після нього.

1.5.5 Правила викликаного

Визначення підпрограми має відповідати наступним правилам на початку підпрограми:

1. Надішліть значення EBP у стек, а потім скопіюйте значення ESP у EBP, дотримуючись таких інструкцій:

push ebp

mov ebp, esp

Ця початкова дія підтримує *базовий вказівник* EBP. Базовий вказівник використовується за домовленістю як точка відліку для пошуку параметрів і локальних змінних у стеку. Під час виконання підпрограми базовий вказівник містить копію значення вказівника стека з моменту початку виконання підпрограми. Параметри та локальні змінні завжди будуть розташовані на відомих постійних зсувах від базового значення вказівника. Ми вставляємо старе базове значення вказівника на початку підпрограми, щоб пізніше ми могли відновити відповідне базове значення вказівника для абонента, коли підпрограма повернеться. Пам'ятайте, що абонент не очікує, що підпрограма змінить значення базового вказівника. Потім ми переміщуємо вказівник стека в EBP, щоб отримати нашу точку відліку для доступу до параметрів і локальних змінних.

2. Далі виділіть локальні змінні, звільнивши місце в стеку. Нагадаємо, стек росте вниз, тому, щоб звільнити простір у верхній частині стека, потрібно зменшити вказівник стека. Величина, на яку зменшується вказівник стека, залежить від кількості та розміру необхідних локальних змінних. Наприклад, якщо потрібні 3 локальні цілі числа (по 4 байти кожне), вказівник стека потрібно буде зменшити на 12, щоб звільнити місце для цих локальних змінних (тобто sub esp, 12). Як і параметри, локальні змінні будуть розташовані за відомими зміщеннями від базового вказівника.

3. Потім збережіть значення регістрів, *збережених викликом*, які використовуватимуться функцією. Щоб зберегти регістри, помістіть їх у стек. Регістри, збережені викликаним, - це EBX, EDI та ESI (ESP і EBP також будуть збережені

відповідно до угоди про виклики, але їх не потрібно надсилати в стек під час цього кроку).

Після виконання цих трьох дій тіло підпрограми може продовжувати роботу. Коли підпрограма повертається, вона повинна виконати такі дії:

1. Залишити значення, що повертається, у EAX.
2. Відновити старі значення будь-яких регістрів, збережених викликом (EDI та ESI), які були змінені. Вміст регістру відновлюється шляхом витягування його зі стеку. Регістри слід відкривати в порядку, зворотному порядку їх надсилання.
3. Звільнити локальні змінні. Очевидним способом зробити це може бути додавання відповідного значення до вказівника стека (оскільки простір було виділено шляхом віднімання необхідної кількості від вказівника стека). На практиці менш схильний до помилок спосіб звільнити змінні - це перемістити значення в базовому вказівнику в вказівник стека: mov esp, ebp. Це працює, оскільки базовий вказівник завжди містить значення, яке містив вказівник стека безпосередньо перед розподілом локальних змінних.
4. Безпосередньо перед поверненням відновіть базове значення вказівника абонента, вилучивши EBP зі стеку. Згадайте, що перше, що ми зробили при вході в підпрограму, це натиснути базовий вказівник, щоб зберегти його старе значення.
5. Нарешті, поверніться до абонента, виконавши інструкцію **ret**. Ця інструкція знайде та видалить відповідну адресу повернення зі стеку.

Зауважте, що правила абонента розпадаються на дві половини, які в основному є дзеркальним відображенням одна одної. Перша половина правил застосовується до початку функції, і зазвичай кажуть, що вона визначає *пролог* до функції. Остання половина правил застосовується до кінця дійства, і тому зазвичай кажуть, що вона визначає *епілог дійства*.

Приклад визначення функції, яка відповідає правилам викликаного:

```
.486
.model flat
.CODE
PUBLIC _myFunc
_myFunc PROC
 ; Пролог підпрограми
 push ebp ; Збережіть старе базове значення вказівника.
 mov ebp, esp; Встановіть нове значення базового вказівника.
```

```
 sub esp, 4 ; Звільніть місце для однієї 4-байтової локальної змінної.
 push edi ; Збережіть значення регістрів функції
 push esi ; буде змінено. Ця функція використовує EDI та ESI.
; (не потрібно зберігати EBX, EBP або ESP)

 ; Тіло підпрограми
 mov eax, [ebp+8] ; Перемістити значення параметра 1 в EAX
 mov esi, [ebp+12] ; Перемістити значення параметра 2 в ESI
 mov edi, [ebp+16] ; Перемістити значення параметра 3 в EDI
mov [ebp-4], edi ; Перемістіть EDI в локальну змінну
 add [ebp-4], esi ; Додайте ESI до локальної змінної
 add eax, [ebp-4] ; Додайте вміст локальної змінної; в EAX (кінцевий результат)
 ; Підпрограма Епілог
 pop esi ; Відновити значення регістру
 pop edi
 mov esp, ebp ; Звільнити локальні змінні
 pop ebp ; Відновити базове значення вказівника абонента
 ret
_myFunc ENDP
END
```

Пролог підпрограми виконує стандартні дії збереження моментального знімка вказівника стека в EBP (базовий вказівник), виділення локальних змінних шляхом зменшення вказівника стека та збереження значень регістру в стеку.

У тілі підпрограми ми бачимо використання базового вказівника. І параметри, і локальні змінні розташовані на постійних зсувах від базового вказівника на час виконання підпрограм. Зокрема, ми помітили, що оскільки параметри були розміщені в стеку до виклику підпрограми, вони завжди розташовані нижче базового вказівника (тобто за вищими адресами) у стеку. Перший параметр підпрограми завжди можна знайти в місці пам'яті EBP + 8, другий – у EBP + 12, третій – у EBP + 16. Так само, оскільки локальні змінні виділяються після встановлення базового вказівника, вони

завжди знаходяться над базовий вказівник (тобто на нижчі адреси) у стеку. Зокрема, перша локальна змінна завжди знаходиться в EBP - 4, друга в EBP - 8 і так далі.

Епілог функції є в основному дзеркальним відображенням прологу функції. Значення регістра виклику відновлюються зі стеку, локальні змінні звільняються шляхом скидання вказівника стека, базове значення вказівника виклику відновлюється, а інструкція ret використовується для повернення до відповідного розташування коду в викликаючому.

1.5.6 Кадри стека

Розглянутий нами механізм роботи зі стеком може здатися комусь дещо складним. Однак ми тут розібрали ідеальну ситуацію, в якій була викликана одна функція, яка коректно була викликана, коректно виконана і коректно завершена. Проте реальність набагато складніша, особливо для сучасних операційних систем. Реальність в тому, що при виконанні програми дуже ймовірна ситуація, коли відбуваються виклики декількох функцій. При цьому всі вони мають використати один стек. Для такої ситуації поняття стеку мало. Вводиться поняття фрейму стека. На Рис показано стан стеку, де викликані дві функції, причому одна виконується, а інша має чекати.

Стек викликів складається з кадрів стека (також званих записами активації або кадрами активації). Це машинно-залежні та ABI -залежні структури даних, що містять інформацію про стан підпрограми. Кожен кадр стека відповідає виклику підпрограми, яка ще не завершилася поверненням. Наприклад, якщо в якійсь програмі DrawSquare даний момент виконується підпрограма під назвою DrawLine, яка була викликана програмою DrawSquare, верхня частина стека викликів може бути розташована так, як показано на Рис.1.11.

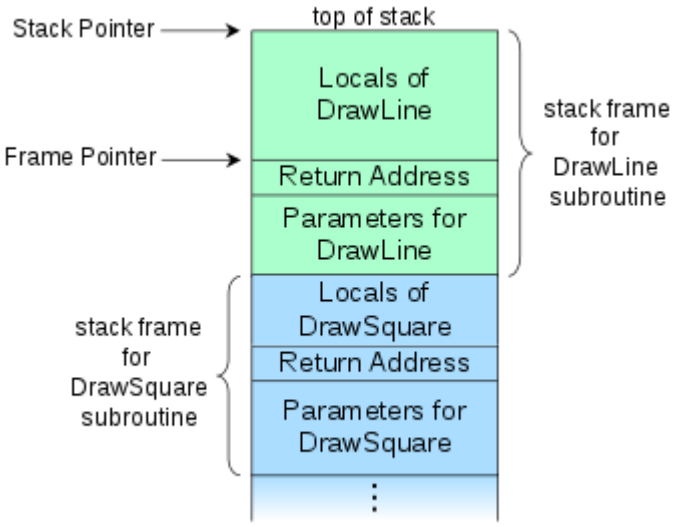

Рис.1.11 Стек з фреймами

Фрейм стека у верхній частині стека призначений для підпрограми, що виконується в даний момент, яка може отримати доступ до інформації у своєму фреймі (наприклад, параметрів або локальних змінних) у будь-якому порядку. Фрейм стека зазвичай включає принаймні такі елементи (у порядку проштовхування):

- аргументи (значення параметрів), передані підпрограмі (якщо є);
- зворотна адреса до виклику підпрограми (наприклад, у DrawLine кадрі стека, адреса в DrawSquare коді); і
- простір для локальних змінних підпрограми (якщо такі є).

Коли розміри кадрів стека можуть відрізнятися, наприклад, між різними функціями або між викликами певної функції, видалення кадру зі стеку не є фіксованим зменшенням **вказівника стека**. Після повернення функції вказівник стека натомість відновлюється до **вказівника кадру**, значення вказівника стека безпосередньо перед викликом функції. Кожен фрейм стека містить вказівник стека на вершину фрейму безпосередньо під ним. Вказівник стека - це змінний регістр, спільний для всіх викликів. Вказівник кадру даного виклику функції є копією вказівника стека, яким він був до виклику функції.

Розташування всіх інших полів у кадрі можна визначити відносно верхньої частини кадру, як від’ємні зсуви вказівника стека, або відносно верхньої частини кадру нижче, як додатні зсуви вказівника кадру. Розташування самого вказівника кадру за своєю суттю має бути визначено як негативне зміщення вказівника стека.

У більшості систем кадр стека має поле, яке містить попереднє значення регістра вказівника кадру, значення, яке він мав під час виконання виклику. Наприклад, фрейм стека `DrawLine` матиме ділянку пам'яті, що містить значення вказівника кадру, яке `DrawSquare` використовує (не показано на схемі вище). Значення зберігається при вході в підпрограму і відновлюється після повернення. Наявність такого поля у відомому розташуванні у фреймі стека дозволяє коду отримувати доступ до кожного фрейму послідовно під фреймом підпрограми, що виконується на даний момент, а також дозволяє підпрограмі легко відновити вказівник кадру на фрейм *абонента,* безпосередньо перед тим, як він повернеться.

У мові з вільними вказівниками або неперевіреними записами в масив (наприклад, у C), змішування даних потоку керування, що впливає на виконання коду (адреси повернення або вказівники збережених кадрів) і простих програмних даних (параметрів або повернених значень) у стеку викликів є ризиком для безпеки, який, можливо, можна використати через переповнення буфера стека.

1.5.7 Організація вводу-виводу

Операцій вводу-виводу процесори не мали і не мають, це - прерогатива виключно операційної системи. Для цього в асемблерах пишуть досить непрості підпрограми, які потребують системних викликів, що потім оформлюють в бібліотеки. Однак необхідно відмітити спочатку специфічний спосіб організації системних викликів шляхом використання макродирективи Invoke. Використання Invoke є одним з основних відмінностей MASM. Вона дозволяє викликати API-функції з перевіркою кількості та типу параметрів. Це майже той же виклик call, але ця макродиректива ще й перевіряє кількість параметрів та їх типи. Приклад виклику функції:

Invoke <функція>, <параметр1>, <параметр2>, <параметр3>.

Щоб використовувати invoke для виклику функції, необхідно визначити її прототип:

tproc PROTO STDCALL :DWORD, :DWORD, :DWORD

Ця макродиректива оголошує процедуру, названу tproc, яка має три параметри розміром DWORD.

Значить, якщо написати:

Invoke tproc, 0,1, 2, 3

MASM видасть помилку, бо процедура tproc має три параметри, а не чотири. MASM також контролює відповідність типів, тобто. перевіряє, чи параметри мають

правильний тип. Бібліотека MASM32.lib містить спеціальні підпрограми введення-виведення консольного режиму, приклади нижче:

Процедура введення:

StdIn PROC lpszBuffer: DWORD, bLen: DWORD

Перший операнд - адреса буфера введення, другий - розмір буфера введення (до 128 байт).

У буфері введення введений рядок завершується символом кінця рядка (нультермінальний символ (13, 10)).

Процедура заміни символів кінця рядка нультермінальним символом:

StripLF PROC lpszBuffer: DWORD; буфер введення

Функція перетворення завершується рядком з нультермінальним символом у вікно консолі:

atol proc lpszBuffer: DWORD; результат - в EAX

Приклад програмування введення:

```
.DATA
zapyt DB 'Input value:', 13,10,0; запит
buffer DB 10 dup ( '0'); буфер введення
.CODE
...
vvod: Invoke StdOut, ADDR zapyt
Invoke StdIn, ADDR buffer, LengthOf buffer Invoke StripLF, ADDR buffer
; Перетворення в SDWORD
Invoke atoi, ADDR buffer; результат в EAX
...
```

Процедура виведення завершується рядком з нультермінальним символом у вікно консолі:

StdOut PROC lpszBuffer: DWORD; буфер виведення, заверш. нулем

Процедура перетворення числа в рядок:

dwtoa PROC public dwValue: DWORD, lpBuffer: PTR BYTE

Програмування виведення:

```
.DATA
result DWORD? ; поле результату
string DB 13,10, 'Result ='; заголовок виведення
resstr DB 16 dup (?); виведене число
```

.CODE

...

; перетворення

Invoke dwtoa, result, ADDR resstr

; виведення

Invoke StdOut, ADDR string

Питання для самоперевірки

Які різновиди виклику функцій?

В яких регістрах містяться зміщення в памяті та абсолютний зсув?

Як змінюються значення регістрів при виконанні інструкції RET?

Які регістри загального призначення можна використовувати для передачі параметрів викликаній процедурі?

В чому суть угоди про виклики в стилі Pascal?

В чому різниця угоди про виклики в стилі Pascal з угодою в стилі C?

Які регістри мають бути збережені при викликаючою програмою(абонентом)?

Що має виконати абонент після повернення процедури для відновлення стану?

Які правила викликаного для коретного виконання процедури?

Які дії підпрограма повинна виконати, коли повертається?

Що таке кадр(фрейми) стеку?

Як розміщуються фрейми в стеку?

Які особливості системного виклику MASM?

Які особливості організації вводу/виводу MASM?

1.6 Опис даних в асемблерній програмі

Всі дані, які використовуються в програмах на асемблері, обов'язково мають бути оголошені з використанням відповідних директив, які визначають тип даних і кількість байт, необхідних для розміщення цих даних в пам'яті:

[<Ім'я>] <Директива> [<Константа> DUP (] <Список ініціалізаторів> [)]

. де <Ім'я> - ім'я поля даних, яке може не присвоюватися;

<Директива> - команда, що оголошує тип описуваних даних (див. Таблицю 1) <Константа> DUP - використовуються при опису повторюваних даних, тоді константа визначає кількість повторень;.

<Список ініціалізаторів> - послідовність дій, що ініціалізували константу через кому або символ «?», якщо ініціалізуюче значення не визначається. Нижче в Таблиці 1.5 надано типи даних MASM.

Таблиця 1.5 Типи даних MASM

Директива	**Опис типу даних**
BYTE / SBYTE	8- розрядне ціле без знаку / знаком
WORD / SWORD	16- розрядне ціле без знаку / знаком
DWORD / SDWORD	32-розрядне ціле без знаку /знаком или ближній вказівник
FWORD	48-разрядне ціле або дальній вказівник
QWORD	64-розрядне ціле
TBYTE	80- розрядне ціле
REAL4	32-х розрядне коротке дійсне
REAL8	64-х розрядне довге дійсне
REAL10	80-ти розрядне расширене дійсне

Як ініціалізатори при описі даних застосовуються:цілі константи

[<знак>] <ціле> [<основа системи числення>], наприклад:

43236, 236d - цілі десяткові числа,

23h, 0ADh - цілі шістнадцятиричні числа (якщо шістнадцятирична константа починається з букви, то перед нею вказується 0),

0111010b - ціле двійкове;

дійсні константи [<знак>] <ціле>. [E | e [<знак>] <ціле>], наприклад: -2., 34E-28;

символи в кодуванні ASCII (MS DOS) або ANSI (Windows) в апострофах або лапках, наприклад: 'A' або "A";

Програма обчислення виразу з вводом-виводом:

```
; Template for console application
.586
.MODEL flat, stdcall OPTION CASEMAP:NONE
Include kernel32.inc Include masm32.inc IncludeLib kernel32.lib IncludeLib masm32.lib
.CONST
MsgExit        DB      13,10,"Press Enter to Exit",0AH,0DH,0
               .DATA
 B             SWORD   -6
 D             SWORD   11
```

```
 X          SWORD     ?
 fX         SWORD     0       ; старше слово результата
 Zapyt      DB        13,10,'Input A',13,10,0
 Result     DB        'Result='
 ResStr     DB        16 DUP (' '),0
.DATA?
A   SWORD ?
fA  SWORD ?; старше слово змінної A
Buffer DB     10 DUP (?) inbuf     DB     100 DUP (?)
.CODE
Start:  Invoke StdOut,ADDR Zapyt
Invoke StdIn, ADDR Buffer, LengthOf Buffer Invoke StripLF, ADDR Buffer
; Перетворення в SDWORD
Invoke atol, ADDR Buffer ;результат в EAX
mov    DWORD PTR A,EAX
; Обчислення
mov    CX,D
add    CX,8; CX:=D+8
mov    BX,B
dec    BX     ; BX:=B-1
mov    AX,A
add    AX,D; AX:=A+D
imul   BX; DX:AX:=(A+D)*(B-1) idiv    CX     ; AX:=(DX:AX):CX
mov    X,AX
; Перетворення
Invoke dwtoa, X, ADDR ResStr
; Виведення
Invoke StdOut, ADDR Result XOR EAX, EAX
Invoke StdOut, ADDR MsgExit
Invoke StdIn,ADDR inbuf, LengthOf inbuf
Invoke ExitProcess,0 End      Start
```

Питання для самоперевірки

Який порядок використання даних в асемблернійі програмі?

Які типи даних можуть бути оголошені в середовищі MASM?

Який порядок використання констант в середовищі MASM?

1.7 Програмування в NASM

NASM (Netwide Assembler) - вільний асемблер для архітектури Intel x86. Використовується для написання 16-, 32- та 64-розрядних програм. Хоча може, правда без генерації коду, працювати на процесорах не x86. Цікавий у першу чергу тим, що ASM компілює програми під різні операційні системи в межах x86-сумісних процесорів, навіть такі як QUNIX та KolibriOS. Перебуваючи в одній операційній системі, можна безперешкодно відкомпілювати файл для іншої. Цьому сприяє ще й те, в NASM прийнято притримуватись Intel-синтаксису запису інструкцій, і не використовує синтаксис AT&T, який поширений для неінтеловських процесорів. Він цікавий тим, що його зручно використовувати для ОС типу UNIX, особливо таких, які підтримують процесорну архітектуру x86, зокрема Linux. Причому саме для Linux код NASM виглядає як справжній асемблер.

Технології розробки програми – асемблера в NASM описана в Додатку Б.

Стосовно набору команд, то тут не має суттєвих відмінностей, принаймні на рівні базового набору команд, розлянутого вище. Синтаксис може дещо відрізнятися, але інструкції x86/64 універсальні. NASM - це сучасний асемблер x64, NASM, який може генерувати файли з розширенням .o, які можна зв’язати. Хоча кожен асемблер має дещо інший синтаксис. MASM подекуди складніший і суперечливіший, ніж потрібно, але ви можете використати вже розібраний набір асемблерних команд та перетворивши код на синтаксис NASM. Якусь віртуальну машину налаштовувати не потрібно.

Основні відмінності, полягають у тому, що NASM постійно використовує нотацію в дужках для розіменування адреси замість PTR або того, що було в MASM.

Приклад простої програми

Подивимось далі, як треба розробляти програму на прикладі простої програми, щоб зрозуміти, як використовуються регістри в програмуванні на асемблері. Ця програма виводить 9 зірочок з простим повідомленням:

```
section .text
global _start; має бути оголошено для лінкера
```

```
    (gcc)
_start:; повідомляємо лінкеру точку входу
mov edx, len; довжина повідомлення
mov ecx, msg; повідомлення для написання
mov ebx, 1; файловий дескриптор (stdout)
mov eax, 4; номер системного виклику
    (sys_write)
int 0x80; виклик ядра
mov edx, 9; довжина повідомлення
mov ecx, s2; повідомлення для написання
mov ebx, 1; файловий дескриптор (stdout)
mov eax, 4; номер системного виклику
    (sys_write)
int 0x80; виклик ядра
mov eax, 1; номер системного виклику (sys_exit)
int 0x80; виклик ядра
section .data
msg db 'Displaying 9 stars', 0xa; наше
    повідомлення
len equ $ - msg; довжина нашого повідомлення
s2 times 9 db '*'
```

Результат виконання цієї програми:

```
Displaying 9 stars
*********
```

Ми розглянули базові поняття та команди для процесора IA – 32. Далі розглянемо деякі розширені можливості на прикладі NASM, які стосуються всіх інших реалізацій асемблерів x86.

1.7.1 Виконання операцій з плаваючою точкою(комою) за допомогою інструкцій співпроцесора x87

Поява співпроцесора додала понад 80 нових інструкцій до набору інструкцій 80x86 для роботи з даними з плаваючою точкою. Ми можемо класифікувати ці інструкції наступним чином:

- переміщення даних,
- перетворення,
- арифметичні інструкції,
- порівняння,
- константні інструкції,
- трансцендентні інструкції
- інші інструкції

Розглянемо програмування з застосуванням цих нових команд на прикладі арифметичних операцій. Однак спочатку розглянемо операції пертміщення, без яких всі інші операції просто неможливі.

Інструкції FINIT і FNINIT

Інструкція FINIT ініціалізує FPU для належної роботи. Ваші програми мають виконати цю інструкцію перед виконанням будь-яких інших інструкцій FPU.

Інструкції щодо переміщення даних FPU

Інструкції переміщення даних передають дані між внутрішніми регістрами FPU та пам’яттю, але не з регістрами самого процесора. Інструкції в цій категорії: FLD, FST, FSTP і FXCH. Інструкція FLD завжди поміщає свій операнд у стек регістрів із плаваючою комою. Інструкція FSTP завжди відкриває верхню частину стека після збереження верхньої частини стека (tos). Решта інструкцій не впливають на кількість елементів у стеку.

Інструкція FLD завантажує 32-, 64- або 80-бітне значення з плаваючою комою в стек. Ця інструкція перетворює 32- та 64-бітні операнди на 80-бітове значення розширеної точності перед тим, як надсилати значення в стек із плаваючою комою.

Інструкції FST і FSTP копіюють значення на вершині стека регістрів з плаваючою комою в інший регістр з плаваючою комою або в 32-, 64- або 80-бітну змінну пам'яті. Під час копіювання даних до 32- або 64-розрядної змінної пам’яті 80-розрядне значення розширеної точності у верхній частині стека округлюється до меншого формату, як зазначено

Арифметичні інструкції співпроцесора x87

Арифметичні інструкції складають невелику, але важливу підмножину набору інструкцій FPU. Обмежимось розглядом декількох операцій.

Інструкції FADD та FADDP

Ці дві інструкції мають такі форми:

fadd ()

faddp()

fadd(st0,st i);

fadd(st i,st0);

faddp(st0,st i);

fadd(mem_32_64);

fadd(дійсна_константа);

Перші дві форми рівнозначні. Вони отримують два значення на вершині стека, додають їх і повНаступні дві форми інструкції FADD, з двома операндами регістра FPU, поводяться як інструкція ADD 80x86. Вони додають значення в операнді вихідного регістра до значення в операнді регістра призначення. Зауважте, що один із операндів регістру має бути ST0.

Інструкція FADDP із двома операндами додає ST0 (який завжди має бути вихідним операндом) до операнду призначення, а потім вириває ST0. Операнд призначення має бути одним із інших регістрів FPU.

Остання форма вище, FADD з операндом пам’яті, додає 32- або 64-бітну змінну з плаваючою комою до значення в ST0. Ця інструкція перетворить 32- або 64-бітні операнди на 80-бітове значення розширеної точності перед виконанням додаванняертають їх суму назад у стек.

Інструкції FMUL та FMULP

Інструкції FMUL і FMUL P множать два значення з плаваючою комою. Ці інструкції

дозволяють такі форми:

fmul()

fmulp()

fmul(st i, st0);

fmul(st0, st i);

fmul(mem_32_64);

fmul(дійсна_константа);

fmulp(st0,st i);

Без операндів і FMUL, і FMULP роблять те саме: вони отримують значення в ST0 і ST1, множать ці значення та повертають свій добуток назад у стек. Інструкції FMUL із двома регістровими операндами обчислюють призначення := призначення * джерело. Один із регістрів (джерело або призначення) має бути ST0.

Інструкція FMULP(ST0, ST i) обчислює ST i := ST i * ST0, а потім видаляє ST0. Ця інструкція використовує значення для і перед видаленням ST0. Для інструкції FMUL(mem) потрібен 32- або 64-бітовий операнд пам'яті. Він перетворює вказану змінну пам'яті на 80-бітне значення розширеної точності та множить ST0 на це значення.

Співпроцесор має також відповідні операції для віднімання і ділення з подібним синтаксисом.

Інструкція FSQRT

До множини арифметичних операцій належить і така операція як FSQRT, причому не потребує жодних операндів. Вона обчислює квадратний корінь із значення на вершині стека (TOS) і замінює ST0 цим результатом. Значення TOS має бути нульовим або позитивним, інакше FSQRT створить виняток недійсної операції.

Трансцендентні інструкції

Виходячи з власних міркувань розробники віднесли обчислення квадратного кореня до арифметичних функцій. А вісім трансцендентних (логарифмічних і тригонометричних) інструкцій для обчислення sin, cos, часткового тангенса, часткового арктангенса, 2x -1, y*log2(x) і y*log2(x+1) віднесли в окремий розділ. Використовуючи різні алгебраїчні тотожності, можна легко обчислити більшість інших поширених трансцендентних функцій за допомогою цих інструкцій. Застосування цих інструкцій відповідає логіці обчислення трансцендентних виразів у математичних виразах. Наприклад, інструкції FSIN, FCOS виконуються наступним чином:

Отримують значення з вершини стека регістрів і обчислюють синус, косинус, а потім повертають результат(и) назад у стек. Ці інструкції припускають, що ST0 визначає кут у радіанах, і цей кут має бути в діапазоні -2 63 < ST0 < +2 63. Якщо вихідний операнд виходить за межі діапазону, ці інструкції встановлюють прапор C 2 і залишають ST0 без змін.

Ці інструкції встановлюють прапори помилки стека, точності, недоповнення, денормалізації та недійсної операції відповідно до результату обчислення.

Приклад: оцінка закону косинуса:

*;; $c^2 = a^2 + b^2$ - cos(C)*2*a*b*

;; C значення кута в радіанах

global _start

section.data

a: dq 4.56 ;довжина сторони a

b: dq 7.89 ; довжина сторони b

ang: dq 1.5 ;кут протилежний стороні c (біля 85.94 градусів)

section.bss

c: resq 1 ;the result – length of side c

section.text

_start:

fld qword [a] ; завантажимо a в регістр st0 формат квадроворд

*fmul st0, st0 ;st0 = a * a = a^2*

fld qword [b] ; завантажимо b в регістр st1 формат квадроворд

*fmul st1, st1 ;st1 = b * b = b^2*

fadd st0, st1 ;st0 = $a^2 + b^2$

fld qword [ang] ;завантажимо кут в st0

fcos ;st0 = cos(ang)

*fmul qword [a] ;st0 = cos(ang) * a*

*fmul qword [b] ;st0 = cos(ang) * a * b*

*fadd st0, st0 ;st0 = cos(ang) * a * b + cos(ang) * a * b = 2(cos(ang) * a * b)*

*fsubp st1, st0 ;st1 = st1 - st0 = ($a^2 + b^2$) - (2 * a * b * cos(ang))*

;and pop st0

fsqrt ;отримання квадратного корня з st0 = c

fst qword [c] ;store st0 in c – отримано!

;end program

Питання для самоперевірки

Які особливості виконання операцій з плаваючою комою для процесора IA-32?

Якими інструкціями переміщуються дані у співпроцесорі?

Які особливості виконання інструкцій переміщування даних у співпрцесорі?

Які особливості арифметичних інструкцій у співпроцесорі?

Що можна обраховувати за допомогою трансцедентний операцій?

1.7.2 Переривання та системні виклики

З точки зору виконання коду програми - переривання є механізмом, за допомогою якого можна змінити керування потоком програми. Ми знаємо два інших механізми для того ж: процедури та безумовні переходи на мітку(стрибки). У той час як переходи забезпечують односторонню передачу керування, процедури забезпечують механізм повернення керування до точки виклику, коли викликана процедура завершена.

Переривання забезпечують механізм, подібний до механізму виклику процедури. Викликання переривання передає керування процедурі, яка називається програмою обслуговування переривань (ISR). ISR також називають обробником. Коли ISR завершено, перервана програма відновлює виконання так, ніби її не було перервано. Ця поведінка аналогічна виклику процедури. Однак існують деякі основні відмінності між процедурами та перериваннями, які роблять переривання майже незамінними.Слід зазначити, що виключно всі ISR належать операційній системі, причому його ядру на відміну від звичайних процедур, які компілюються разом з основною програмою, або належать прикладним бібліотекам.

Однією з головних відмінностей є те, що переривання можуть бути ініційовані як програмним, так і апаратним забезпеченням. Навпаки, процедури ініціюються виключно програмним забезпеченням. Той факт, що переривання можуть бути ініційовані апаратним забезпеченням, є головним фактором значної частини потужності переривань. Ця можливість дає нам ефективний спосіб, за допомогою якого зовнішні пристрої (поза процесором) можуть привернути увагу процесора.

Ще одна відмінність між процедурами та перериваннями полягає в тому, що ISR зазвичай є резидентними в пам'яті. Навпаки, процедури завантажуються в пам'ять разом із прикладними програмами.

Деякі інші відмінності, такі як використання номерів для ідентифікації переривань, а не імен, використання механізму виклику, який автоматично надсилає регістр прапорів у стек, і тому подібне, вказано далі.

Третя категорія називається винятками. Винятки обробляють помилки команд. Прикладом винятку є помилка ділення, яка генерується під час кожної спроби поділити на 0. Ця помилка виникає під час div або idiv виконання інструкції, якщо дільник дорівнює 0.

Програмні переривання записуються в програму за допомогою інструкції int. Основне використання програмних переривань полягає в доступі до пристроїв вводу/виводу, таких як клавіатура, принтер, екран дисплея, дисковод тощо. Програмні переривання можна далі класифікувати на визначені системою та визначені користувачем. Апаратні переривання генеруються апаратними пристроями, щоб привернути увагу процесора. Наприклад, коли ви натискаєте клавішу, апаратне забезпечення клавіатури генерує зовнішнє переривання, змушуючи процесор призупинити свою поточну діяльність і виконати програму обслуговування переривання клавіатури для обробки клавіші. Після завершення ISR клавіатури процесор продовжує роботу, яку він робив до переривання. Апаратні переривання можуть маскуватися або не маскуватися. Процесор завжди звертається до немаскованого переривання (NMI) негайно. Одним із прикладів NMI є помилка парності оперативної пам’яті, яка вказує на несправність пам’яті. Масковані переривання можуть бути відкладені, до настання слушного моменту. Для прикладу нехай процесор виконує основну програму. Виникає переривання. У результаті процесор призупиняє основну роботу, як тільки він завершує поточну інструкцію, і передає керування ISR1. Якщо ISR1 має бути виконано без будь-яких переривань, процесор може маскувати подальші переривання, доки воно не буде завершено. Припустимо, що під час виконання ISR1 виникає ще одне маскуване переривання. Обслуговування цього переривання має чекати, доки не завершиться ISR1.

Розглянемо обробку переривання в захищеному режимі. На відміну від процедур, де ім'я дається для ідентифікації процедури, переривання ідентифікуються номером типу. Процесор IA-32 підтримує 256 різних типів переривань. Номер типу переривання, який також називають вектором, використовується як індекс у таблиці, яка зберігає адреси ISR.

Ця таблиця називається таблицею дескрипторів переривань (IDT). Подібно до глобальних і локальних таблиць дескрипторів GDT і LDT, кожен дескриптор по суті є вказівником на ISR і вимагає восьми байтів. Номер типу переривання масштабується на 8 для формування індексу в IDT. Головне питання полягає в тому, щоб знайти адресу відповідного коду в ISR, а потім завантажити в регістри процесора замість наступної команди, яка виконувалась у момент виклику переривання. Але не менш важливим є питання зберегти адресу наступної команди, а також стан процесора, який був у момент виконання поточної команди.Якщо цього не зпробити, то це буде не переривання, а крах програми. Це непроста задача навіть для однозадачної ОС в режимі реальної адресації. Для обслуговуання багатьох задач використовуються таблиці IDT, які можуть

знаходитися будь-де у фізичній пам'яті. Розташування IDT зберігається в регістрі IDT IDTR. IDT не вимагає більше 2048 байт, оскільки може бути не більше 256 дескрипторів. У системі кількість дескрипторів може бути набагато меншою за максимально допустиму. У цьому випадку межа IDT може бути встановлена на необхідний розмір. Якщо дескриптор, на який посилається, виходить за межі IDT, процесор переходить у режим вимкнення. У цьому режимі виконання інструкції зупиняється, доки не буде отримано немаскувальне переривання або сигнал скидання. IDT може мати три типи дескрипторів: шлюз переривання, шлюз перехоплення та шлюз завдання. Ми не будемо обговорювати шлюз завдань, оскільки він не має прямого відношення до механізму переривання, який ми використовуємо.

Обидва шлюзи зберігають ідентичну інформацію: 16-бітний селектор сегмента, 32-бітове зміщення, рівень привілеїв дескриптора (DPL) і біт P, щоб вказати, присутній сегмент чи ні. Коли виникає переривання, селектор сегмента використовується для вибору дескриптора сегмента, який знаходиться або в GDT, або в поточному LDT. Дескриптор сегмента надає базову адресу сегмента, який містить програму обслуговування переривання. Частина зміщення надходить від шлюзу переривання. З цього моменту запускається переривання і виконуються наступні дії.

1. Регістр EFLAGS заштовується у стек;
2. Очищаються прапори переривання та перехоплення;
3. У стек заштовхуються регістри CS та EIP;
4. CS завантажується 16-бітним селектором сегментів із шлюзу переривання;
5. EIP завантажується з 32-розрядним значенням зсуву зі шлюзу переривання.

Після отримання переривання регістр прапорів автоматично зберігається в стеку. Прапорці переривання очищаються, щоб вимкнути подальші переривання. Зазвичай цей прапорець встановлюється в ISR, якщо тільки є особлива причина відключити інші переривання. Прапор переривання можна встановити sti і очищено за допомогою інструкцій мови асемблера cli. Обидві ці інструкції не вимагають операндів. Ми повинні використовувати popf і pushf, щоб змінити прапор перехоплення. Поточні значення CS і EIP надсилаються в стек. Регістри CS та EIP відповідно завантажуються селектором сегментів і зміщенням від шлюзу переривання. На Рис. 1.12 показана схема виклику у захищеному режимі адресації.

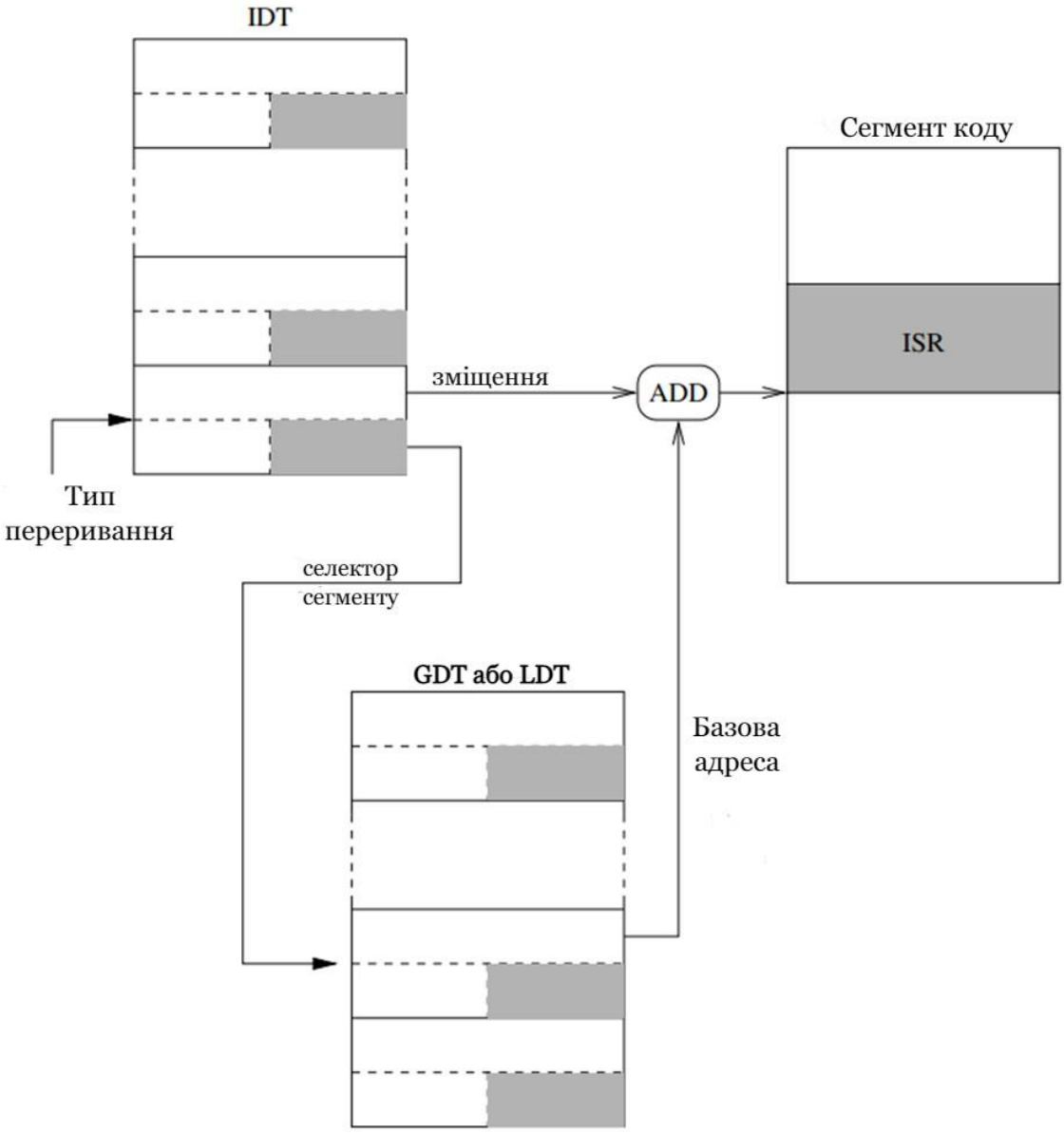

Рис. 1.12 Виклик переривань в захищеному режимі IA-32

Повернення від обробника переривань Так само, як і процедури, виклики переривань повинні закінчуватися оператором return, щоб відправити керування назад до перерваної програми. Для цього використовується повернення переривання (iret). Останньою інструкцією ISR має бути інструкція iret. Він служить для тих же цілей, що і ret для процедур. Дії, вжиті в результаті iret наступні.

1. Повернення 32-розрядне значення зі стека в регістр EIP;
2. Повернення 16-бітове значення зі стека в регістр CS;
3. Повернення 32-розрядне значення зі стека в регістр EFLAGS.

Програмні переривання

Програмні переривання ініціюються виконанням інструкції переривання. Формат цієї інструкції:

interrupt-type int

де interrupt-type є цілим числом у діапазоні від 0 до 255 (обидва включно). Таким чином, загалом можливо 256 різних типів. Це досить велика кількість, оскільки кожен тип переривання може бути параметризований для надання кількох послуг. Як правило програмні переривання використовуються для організації системних викликів.

1.7.3 Організація системного виклику

Системний виклик - це інтерфейс, через який процес спілкується із системним викликом.

Оскільки сучасні *к*омп'ютерні система працюють в двох режимах: режим користувача та режим ядра, то кожна програма користувача – це процес, що виконується в режимі користувача, і коли здійснюється системний виклик, режим перемикається в режим ядра. Після завершення виконання системного виклику керування передається назад процесу в режимі користувача. Системний виклик здійснюється шляхом надсилання сигналу перехоплення ядру, яке зчитує код системного виклику з регістру та виконує системний виклик. Основними типами системних викликів є керування процесами, керування файлами, керування пристроями, обслуговування інформації та зв’язок.

Правила для передачі параметрів під час здійснення системного виклику полягають у тому, що це не може бути число з плаваючою точкою, може передаватися обмежена кількість аргументів, а операції push, pop зі стеку виконуватиме лише операційна система. На Рис. 1.13 показана схема організації системного виклику на прикладі запиту на читання файлу.

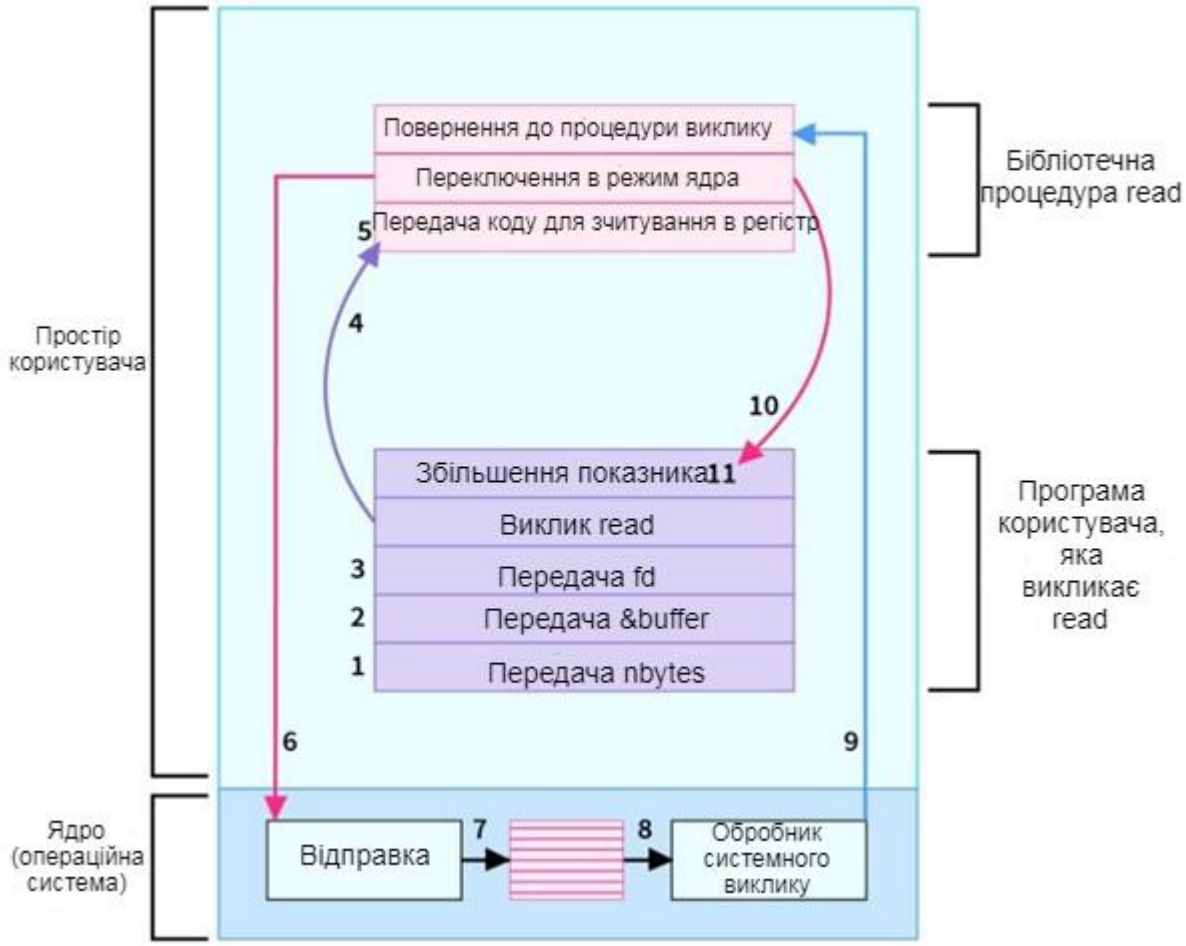

Рис. 1.13 Схема організації системного виклику запиту читання файлу read

Наприклад, Linux надає велику кількість послуг через int 0x80. Фактично, він забезпечує понад 180 різних системних викликів!

Усі ці системні виклики викликаються int 0x80. Потрібна послуга визначається шляхом розміщення номера системного виклику в регістрі EAX. Якщо кількість аргументів, необхідних для системного виклику, менше шести, вони розміщуються в інших регістрах. Зазвичай системний виклик також повертає значення в регістрах. У наступному розділі ми надамо детальну інформацію про деякі служби доступу до файлів, які надає int 0x80.

В ОС Linux є 256 векторів переривань, які підтримує IA - 32, причому Linux використовує перші 32 вектори (тобто від 0 до 31) для винятків і немаскованих переривань. Наступні 16 векторів (від 32 до 47) використовуються для апаратних переривань, створених через лінії запиту на переривання (IRQ). Він використовує один вектор (128 або 0x80) для програмного переривання для надання системних послуг.

Незважаючи на те, що для системних служб використовується лише один вектор переривання, Linux надає кілька служб, використовуючи це переривання.

Однак, слід зазначити, що у нових версіях операційних систем як Unix, так і Windows, орієнтованих на нові моделі процесорів x86 спостерігається тенденція відходу від прямого застосування переривань для організації системного виклику. В ОС Linux крім описаного переривання int 128 :

В архітектурі x86-64 ОС Linux існує ще декілька різних способів системних викликів:

через команди:

- sysenter/sysexit
- syscall/sysret
- vsyscall
- vDSO

У реалізації кожного системного виклику є свої особливості, але загалом обробник в Linux має приблизно однакову структуру:

- Включається захист від читання/запису/виконання коду користувальницького простору.
- Замінюється користувача стек на стек ядра, зберігаються регістри.
- Виконується обробка системного виклику
- Відновлюється стекта регістрів
- Вимкнення захисту
- Вихід із системного виклику

Ще сильніше ця тенденція прослідковується в ОС Windows

Зокрема у Windows -10 є тільки три непрівелейовані переривання. зокрема:

- int 1 (як кодування CD 01, так і переривання налагодження відбувається після однієї інструкції, якщо EFLAGS_TF встановлено в eflags)
- int 3 (обидва кодування CC і CD 03)
- int 2Eh, (системний виклик Windows)

Усі інші переривання є привілейованими, і їх виклик спричиняє виняток «недійсна інструкція» замість цього. INT - інструкція з «контролем привілеїв». Це має бути таким чином, щоб ядро захистило себе від режиму користувача. INT проходить ті самі вектори перехоплення, через які проходять апаратні переривання та виняткові ситуації процесора, тому, і якби режим користувача міг довільно ініціювати ці винятки, код диспетчеризації переривань заплутався б.

Переривання int 1 та int 3 використовують для системного відлагодження, а не для програм користувачів. Щодо int 2Eh, яке було основним способом організації системного виклику для 32-розрядних Windows, то у сучасних 64-розряних версіях ОС Windows більшою мірою рекомендується для використання інструкцію SYSCALL, або SYSENTER на інших моделя процесорів.

SYSCALL викликає обробник системних викликів ОС на рівні привілеїв 0. Це робиться шляхом завантаження RIP з регістра MSR IA32_LSTAR (після збереження адреси інструкції, наступної за SYSCALL, у RCX). (Інструкція WRMSR гарантує, що MSR IA32_LSTAR завжди містить правильну адресу.) SYSCALL також зберігає RFLAGS у регістрі R11, а потім маскує RFLAGS за допомогою IA32_FMASK MSR (адреса MSR C0000084H); Зокрема, процесор очищає в RFLAGS кожен біт, що відповідає біту, встановленому в IA32_FMASK MSR. Далі SYSCALL завантажує селектори CS і SS значеннями, отриманими з бітів 47:32 MSR IA32_STAR. Однак кеші дескрипторів CS і SS не завантажуються з дескрипторів (у GDT або LDT), на які посилаються ці селектори. Замість цього кеші дескрипторів завантажуються з фіксованими значеннями. Операційна система відповідає за те, щоб дескриптори (у GDT або LDT), на які посилаються ці значення селектора, відповідали фіксованим значенням, завантаженим у кеші дескрипторів; інструкція SYSCALL не забезпечує цю відповідність.

Інструкція SYSCALL не зберігає вказівник стека (RSP). Якщо обробник системного виклику ОС змінить вказівник стека, програмне забезпечення несе відповідальність за збереження попереднього значення вказівника стека. Це можна зробити до виконання SYSCALL, програмне забезпечення відновить вказівник стека за допомогою інструкції після SYSCALL (яка буде виконана після SYSRET). Крім того, обробник системних викликів ОС може зберегти вказівник стека та відновить його перед виконанням інструкції SYSRET, яка переводить виконання коду програми з інструкціїї, що слідує за SYSCALL в режимі привілеїв 3.

Спеціальні дослідження показують, що використання інструкцій SYSCALL SYSRET суттєво прискорює реалізацію системного виклику у порівнянні з використанням класичного механізму переривань. Однак слід зазначити, яким чином не був організований системний виклик для захищеного режиму, його реалізація може опиратися тільки на один механізм - механізм переключення контексту.

1.7.4 Управління файлами

Система розглядає будь-які вхідні або вихідні дані як потік байтів. Є три стандартних файлових потоки:

- Стандартне введення (**stdin**),
- Стандартне виведення (**stdout**) і
- Виведення помилок (**stderr**).

Операційна система завжди їх відкриває при завантаженні. Всі інші файли мають бути створені і відкриті для роботи.

Файловий дескриптор - це 16-розрядне ціле число, яке призначається файлу як ідентифікатор файлу в ОС Linux. Коли створюється новий файл або відкривається існуючий файл, дескриптор файлу використовується для доступу до файлу.

Файловий дескриптор стандартних файлових потоків - **stdin**, **stdout** і **stderr** - дорівнює **0**, **1** і **2** відповідно.

Файловий вказівник визначає місце розташування для наступної операції читання/запису в файлі у вигляді байтів. Кожен файл розглядається як послідовність байтів. Кожен відкритий файл пов'язаний з вказівником файлу, який задає зміщення в байтах відносно початку файлу. Коли файл відкритий, вказівник файлу встановлюється в нуль.

Системні виклики обробки файлів в Linux

У Таблиці 1.6 коротко описані системні виклики, пов'язані з обробкою файлів.

Таблиця 1.6 Системні виклики, пов'язані з обробкою файлів.

eax	ім'я	ebx	ecx	edx
2	sys_fork	struct pt_regs	-	-
3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t
5	sys_open	const char *	int	int
6	sys_close	unsigned int	-	-
8	sys_creat	const char *	int	-
19	sys_lseek	unsigned int	off_t	unsigned int

Кроки, необхідні для використання системних викликів, такі ж як ми обговорювали раніше:

- Помістіть номер системного виклику в регістр EAX.
- Збережіть аргументи системного виклику в регістрах EBX, ECX і т. д.

- Викличте відповідне переривання (80h).
- Результат зазвичай повертається в регістр EAX.

Створення та відкриття файлу

Для створення і відкриття файлу виконайте наступні завдання:

- Помістіть номер 8 системного виклику sys_creat () в регістр EAX.
- Помістіть ім'я файлу в регістр EBX.
- Помістіть права доступу до файлу в регістр ECX.

Системний виклик повертає дескриптор файлу створеного файлу в регістр EAX, в разі помилки код помилки знаходиться в регістрі EAX.

Відкриття існуючого файлу

Щоб відкрити існуючий файл, виконайте наступні завдання:

- Помістіть номер 5 системного виклику sys_open () в регістр EAX.
- Помістіть ім'я файлу в регістр EBX.
- Помістіть режим доступу до файлу в регістр ECX.
- Помістіть права доступу до файлу в регістр EDX.

Системний виклик повертає дескриптор файлу створеного файлу в регістрі EAX, в разі помилки код помилки знаходиться в регістрі EAX.

Серед режимів доступу до файлів найчастіше використовуються: тільки читання (0), тільки запис (1) і читання-запис (2).

Читання з файлу

Для читання з файлу виконайте наступні завдання:

- Помістіть номер 3 системного виклику sys_read () в регістр EAX.
- Помістіть дескриптор файлу в регістр EBX.
- Помістіть вказівник на вхідний буфер в регістр ECX.
- Помістіть розмір буфера, тобто кількість байтів для читання, в регістр EDX.

Системний виклик повертає кількість байтів, лічених в регістрі EAX, в разі помилки код помилки знаходиться в регістрі EAX.

Запис у файл

Для запису в файл виконайте наступні завдання:

- Помістіть номер 4 системного виклику sys_write () в регістр EAX.
- Помістіть дескриптор файлу в регістр EBX.
- Помістіть вказівник на вихідний буфер в регістр ECX.
- Помістіть розмір буфера, тобто кількість байтів для запису, в регістр EDX.

Системний виклик повертає фактичну кількість байтів, записаних в регістр EAX, в разі помилки код помилки знаходиться в регістрі EAX.

закриття файлу

Для закриття файлу виконайте наступні завдання:

- Помістіть номер 6 системного виклику sys_close () в регістр EAX.
- Помістіть дескриптор файлу в регістр EBX.

Системний виклик повертає, в разі помилки, код помилки в регістрі EAX.

Оновлення файлу

Для оновлення файлу виконайте наступні завдання:

- Помістіть номер 19 системного виклику sys_lseek() в регістр EAX.
- Помістіть дескриптор файлу в регістр EBX.
- Помістіть значення зміщення в регістр ECX.
- Помістіть референтну позицію для зміщення в регістр EDX.

Вихідна позиція може бути:

- Початок файлу - значення 0
- Поточна позиція - значення 1
- Кінець файлу - значення 2

Системний виклик повертає, в разі помилки, код помилки в регістрі EAX.

Приклад

Наступна програма створює і відкриває файл з ім'ям myfile.txt і записує текст «Привіт Фізтех» в цьому файлі. Далі програма читає файл і зберігає дані в буфері з ім'ям info. Нарешті, відображає текст як збережений в info.

```
section.text
global _start; потрібно декларувати для використання gcc
_start:; точка входу для лінкера
; створюємо файл
mov eax, 8
mov ebx, file_name
mov ecx, 0777; читання, запис і виконання всіма
int 0x80; виклик ядра
mov [fd_out], eax
; пишемо в файл
Mov edx, len; кількість байт
```

```
Mov ecx, msg; повідомлення для запису
Mov ebx, [fd_out]; файловий дескриптор
Mov eax, 4; номер системного виклику (sys_write)
int 0x80; виклик ядра
; закриваємо файл
mov eax, 6
mov ebx, [fd_out]
int 0x80; виклик ядра
; виведення повідомлення про те, що закінчено запис у файл
mov eax, 4
mov ebx, 1
mov ecx, msg_done
mov edx, len_done
int 0x80
; відкриваємо файл для читання
mov eax, 5
mov ebx, file_name
mov ecx, 0; доступ тільки для читання
mov edx, 0777; читання, запис і виконання для всіх
int 0x80
mov [fd_in], eax
; читаємо з файлу
mov eax, 3
mov ebx, [fd_in]
mov ecx, info
mov edx, 200
int 0x80
; закриваємо файл
mov eax, 6
mov ebx, [fd_in]
int 0x80
; виводимо info
mov eax, 4
mov ebx, 1
```

mov ecx, info

mov edx, 200

int 0x80

move ax, 1; номер системного виклику (sys_exit)

int 0x80; виклик ядра

section .data

msg db 'Привіт Фізтех!'

len equ $ -msg

msg_done db 'Записано в файл', 0xa

len_done equ $ -msg_done

file_name db 'myfile.txt'

section .bss

fd_out resb 1

fd_inresb 1

info resb200

Результат Виконання програми:

1

2 Записано в файл

Привіт Фізтех!

Питання для самоперевірки

Які суттєві відмінності між між процедурами та перериваннями?

Які категорії переривань?

Що таке немасковані переривання?

Де зберігаються дескриптори переривань?

Які дії викликає виникнення переривання в захищеному режимі IA-32?

Як ініціюються програмні переривання в захищеному режимі IA-32?

Яка схема організації системного виклику в NASM?

Які варіанти системного виклику для сучасних мікропроцесорів?

В чому відмінність у виконанні системного виклику спеціальними інструкціями типу SYSCALL і через використання переривання int 80h або int 2Eh?

Які особливості системних викликів для управавління файлами в NASM?

2.АСЕМБЛЕРИ, ОРІЄНТОВАНІ НА RISC-АРХІТЕКТУРУ

RISC(Reduced Instruction Set Computer) - це термін, введений Девідом Паттерсоном і Девідом Дітцелем у їхній основоположній статті 1981 року «Приклади комп'ютера зі скороченим набором інструкцій». Ці двоє авторів запропонували новий підхід до проектування напівпровідникових пристроїв, який ґрунтується на тенденціях, які спостерігалися наприкінці 1970-х років, і проблемах масштабування, з якими стикаються сучасні процесори. Вони запропонували також і термін «CISC» - Complex (Complete) Instruction Set Computer - для опису багатьох різноманітних архітектур центрального процесора, які вже існують і не відповідають принципам RISC. Концепція RISC в його сучасному розумінні остаточно сформувалася на основі трьох дослідницьких проектів комп'ютерів: процесора 801 від IBM, процесора RISC від Університету Берклі і процесора MIPS від Стенфордського університету. У 1980 році Д.Паттерсон і його колеги з Берклі почали свій проект і виготовили дві машини, які отримали назви RISC-I і RISC-II. Основними ідеями цих машин було відділення повільної пам'яті від швидкодіючих регістрів і використання регістрових вікон.

Ця очевидна потреба в новому підході до проектування ЦП виникла, коли змінилися вузькі місця, що обмежують продуктивність ЦП. Так звані проекти CISC, включаючи оригінальний 8086, були розроблені, щоб мати справу з високою вартістю пам'яті шляхом переміщення складності в апаратне забезпечення. Вони наголошували на щільності коду, а деякі інструкції виконували кілька операцій послідовно над змінною. Філософія дизайну CISC намагалася підвищити продуктивність шляхом мінімізації кількості інструкцій, які ЦП мав виконати для виконання певного завдання. Архітектури набору інструкцій CISC зазвичай пропонують широкий спектр спеціалізованих інструкцій.

Простота архітектури і її ефективність, підтверджена цими проектами, викликала великий інтерес в комп'ютерній індустрії і з 1986 року почалося активне промислове впровадження архітектури RISC. На сьогоднішній день ця архітектура займає лідируючі позиції на ринку мобільних обчислень і міцні позиції на світовому комп'ютерному ринку робочих станцій і серверів. Розвиток архітектури RISC багато в чому визначалося прогресом у створенні оптимізуючих компіляторів. Саме сучасна техніка компіляції дозволяє ефективно використовувати переваги великого регістрового файлу, конвеєрної організації і більшої швидкості виконання команд. Сучасні компілятори також

використовують іншу методику оптимізації, зазвичай використовувану в RISC-процесорах: реалізацію відкладених стрибків і суперскалярну обробку, що дозволяє виконувати кілька інструкцій одночасно. Слід зазначити, що в останніх розробках Intel, а також її наступників-конкурентів (AMD R5, Cyrix M1, NexGen Nx586 і ін.) широко використовуються ідеї, реалізовані в мікропроцесорах RISC, завдяки чому стираються багато відмінностей між CISC і RISC. Однак складність архітектури x86 і набору інструкцій залишається основним фактором, що обмежує продуктивність процесорів на базі організація конвеєра типу x86.

Особливістю процесорів з RISC-архітектурою набір команд, які виконуються, скорочений до мінімуму і всі команди мають одинакову довжину. В цьому принципова різниця з архітектурами CISC. Тому для реалізації більш складних операцій доводиться комбінувати команди з готового набору. При цьому всі команди мають формат фіксованої довжини (наприклад, 12, 14 або 16 біт), вибірка команди з пам'яті і її виконання можуть здійснюється за один цикл (такт) синхронізації. Система команд RISC-процесора, на відміну від команд CISC-процесора, надає можливість рівноправного використання всіх регістрів процесора, тобто всі ці регістри не є тільки регістрами загального призначення, але й регістрами даних, хоча деякі з них дещо спеціалізовані. Це забезпечує додаткову гнучкість при виконанні ряду операцій. Система команд такої архітектури не включає арифметико-логічних операції з операндами у пам'яті. Головними ознаками є фіксований розмір команди, за винятком команд переходу, які можуть міститися в 2 - 3 командних словах (містять адресу). Також ключовою характеристикою є фіксований час виконання команд - кожна команда виконується, наприклад за 1 машинний такт (AVR до 20Мгц) чи за 4 машинних такти (PIC до 80Мгц). Продуктивність програм сильно залежить від якості компілятора (при використанні асемблера та машинних команд більше залежить від програміста). На Рис. 2.1 показано порівняння форматів команд архітектур CISC та RISC.

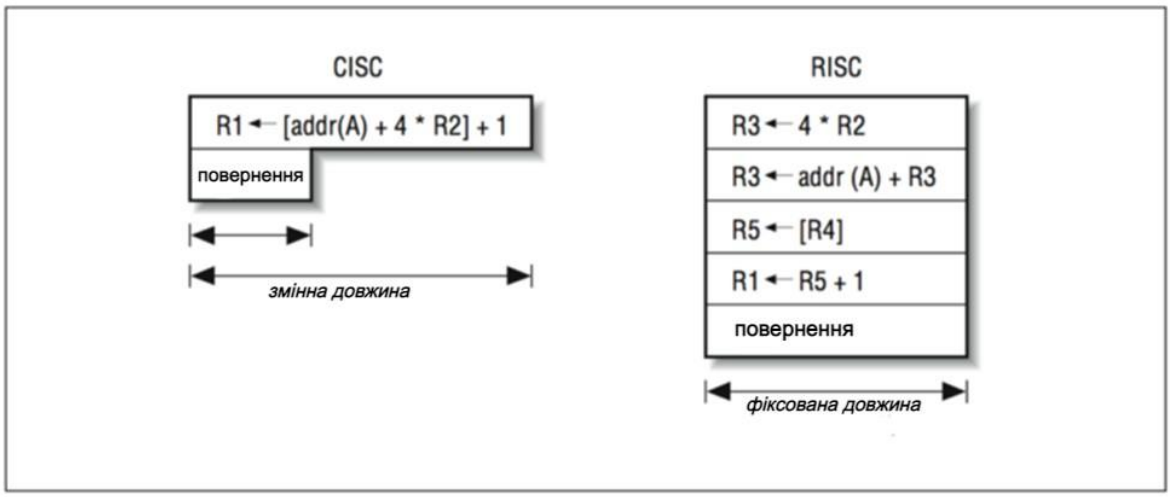

Рис. 2.1 Порівняльні формати команд архітектур CISC та RISC

Інструкції CISC можуть бути будь-якої довжини. Максимальна теоретична довжина інструкції x86 може бути необмеженою, але практично не перевищує 15 байт. Інструкції RISC мають обмежену довжину.

Найвідоміші представники: ARC, Alpha, ARM, AVR, MIPS, PA-RISC, Power Architecture (включаючи PowerPC), SPARC. До сегменту мікроконтролерів із RISC-процесором відносяться AVR фірми Atmel, МК PIC16 і PIC17 фірми Microchip та інші.

2.1 Архітектурні особливості процесорів ARM

Архітектура ARM (Advanced RISC Machine, Acorn RISC Machine, вдосконалена RISC-машина) - сімейство 32-бітових і 64-бітових мікропроцесорних ядер, що ліцензуються розробки компанії ARM Limited. Компанія займається виключно розробкою ядер та інструментів для них (компілятори, засоби налагодження тощо), заробляючи на ліцензуванні архітектури стороннім виробникам. Наразі більшість мобільних пристроїв, планшетів розроблені саме на цій архітектурі процесорів. Основною перевагою даного сімейства є низьке енергоспоживання, завдяки чому часто використовується в різних вбудованих системах, особливо у сучасних смартфонах. Саме тому ця архітектура розвивалася протягом довгого часу, і починаючи з ARMv7 були визначені три сфери застосувань: 'A'(application) - застосунки, 'R'(real time) – системи в реальному часі, 'M'(microcontroller) – мікроконтролери.

На Рис. 2.2 представлена структурна схема ARM7. Дещо спрощено вона складається з наступних функціональних елементів:

Декодер інструкцій і пристрій керування. Функцією декодера інструкцій і блоку управління є декодування інструкцій і генерація керуючих сигналів для інших частин процесора для виконання інструкцій.

Адресний регістр. Виконує функцію відображення 32-розрядної адреси адресної шини.

Інкреметер адрес. Він використовується для збільшення адреси на 4 та розміщення її в регістрі адрес.

Банк(файл) регістрів. Банк регістрів містить тридцять один 32-розрядний регістр і 6 регістрів стану.

Шіфтер Барела. Використовується для швидкого перемикання.

32-розрядний Арифметично-логічний пристрій АЛП(ALU) для виконання арифметичних і логічних операцій.

Регістр запису даних Служить для запису даних в цей регістр при виконанні операції запису.

Регістр читання даних Служить для читання з пам'яті, процесор розміщує результат у цей регістр.

Множник Бута - це пристрій, де електронним способом реалізовано модифікований алгоритм Бута. Алгоритм Бута – це цікавий алгоритм множення для бінарних чисел в додатковому коді. Він однаково обробляє додатні та від’ємні числа. Таке множення закінчується лише за 16 тактів.

Регістр CPSR (Current Program Status Register) - містить прапори, що описують поточний стан процесора. На Рис. 2.2 показаноструктурна схема процесора архітектури ARM7

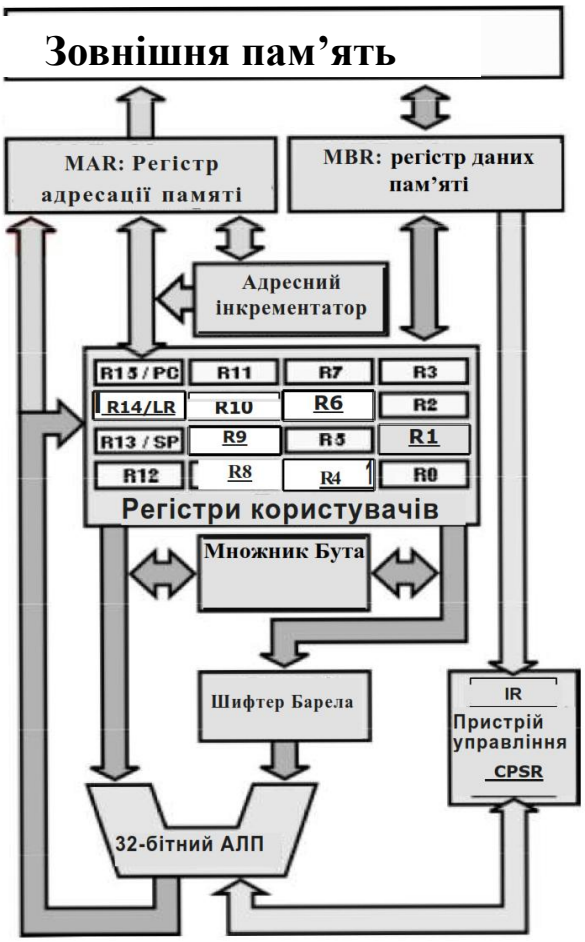

Рис. 2.2 Структурна схема процесораархітектури ARM7

Питання для самоперевірки

Які основні функціональні елементи архітектури ARM7?

Які їх призначення?

Що таке файл регістів?

2.2 Стани процесора ARM

Процесор може бути в одному з наступних режимів виконання операцій:

- User mode - звичайний режим виконання програм. У цьому режимі виконується більшість програм .
- Fast Interrupt (FIQ) - режим оптимізований для передачі даних. Запит на швидке переривання (FIQ) - це спеціалізований тип запиту на переривання, який є стандартною технікою, яка використовується в процесорах комп’ютерів для роботи з подіями, які потрібно обробляти, коли вони відбуваються, наприклад отримання даних від мережевої карти або дії клавіатури чи миші. FIQ є специфічними для архітектури

ARM, яка підтримує два типи переривань; FIQ для швидкої обробки переривань із малою затримкою та стандартні запити на переривання (IRQ) для більш загальних переривань. FIQ має пріоритет над IRQ у системі ARM. Крім того, одночасно підтримується лише одне джерело FIQ. Це допомагає зменшити затримку переривання, оскільки програму обслуговування переривання можна виконувати безпосередньо без визначення джерела переривання. Збереження контексту не потрібне для обслуговування FIQ, оскільки він має власний набір банківських регістрів.

- Interrupt (IRQ) - основний режим керування перериваннями.
- Supervisor mode – захищений режим для використання операційною системою.
- Abort mode - режим, у який процесор переходить у разі виникнення помилки доступу до пам'яті (доступ до даних чи інструкції на етапі prefetch конвеєра).
- System mode - привілейований режим користувача.

Undefined mode - режим, до якого процесор входить під час спроби виконати невідому йому інструкцію. Перемикання режиму процесора відбувається при виникненні відповідного виключення або модифікації регістру статусу.

Питання для самоперевірки

В яких режимах працює процесор

Які особливості режиму Supervisor mode

Які особливості режиму User mode

В чому відмінність між режимами Supervisor mode та System mode

Для чого існує режим FIQ.

2.3 Виконання машинних команд на основі принципу Load / Store

Якщо розглянути функціонування ARM RISC-процесора, ми виявимо п'ятиступінчастий конвеєр інструкцій.

- **(Fetch) Вибірка** інструкції з пам'яті та збільшення лічильника команд, щоб отримати наступну інструкцію в наступному такті.
- **(Decode) Декодування** інструкції – визначення, що ця інструкція робить. Тобто активація необхідних виконання цієї інструкції частин мікропроцесора.
- **(Execute) Виконання** включає використання арифметико-логічного пристрою (АЛП) або здійснення зсувних операцій.
- **(Memory) Доступ до пам'яті**, якщо потрібно. Це те, що робить інструкція *load*.

- **(Write Back) Запис результатів** у відповідний регістр.

Інструкції ARM складаються з секцій, кожна з яких працює з одним із цих етапів, а виконання етапу зазвичай займає один такт. Тобто інструкції ARM дуже зручно конвеєризувати. Більше того, кожна інструкція має однаковий розмір, тобто етап Fetch знає, де розташовуватиметься наступна інструкція, і йому не потрібно проводити декодування.

Щоб кількість тактів, необхідних для кожної інструкції, була приблизно однаковою та зручною для конвеєризації, набір інструкцій RISC чітко відокремлює завантаження з пам'яті та збереження у пам’яті від інших інструкцій.

Наприклад, в CISC може існувати інструкція, яка завантажує дані з пам'яті, виконує додавання, множення, щось ще й записує результат назад у пам'ять. У світі RISC такого не може бути. Операції типу додавання, зсуву та множення виконуються лише з регістрами. Вони не мають доступу до пам'яті. Це дуже важливий момент для конвеєризації. Інакше інструкції у конвеєрі можуть залежати одна від одної.

Особливості архітектури ARM полягають в тому, що є тільки один набір спеціальних команд, які мають право працювати з пам’яттю, тобто як завантажувати у пам’ять, так і робити вибірку з пам’яті. Це пояснюється архітектурними особливостями процесора ARM. На Рис.2.3 приведена с блок-схема CPU з 3-адресною адресацією і специфікою архітектури Load/Store.

На цій схемі виділені компоненти :

- Програмний лічильник PC
- Пам’ять команд
- interrupt-type даних
- Банк Регістрів
- Арифметично-логічний пристрій

Шини A і B Верхній набір ліній від банку регістрів до ALU називається шиною A, а нижні від банку регістрів до ALU називається шиною B.

На схемі (Рис. 2.3) також показані значення регістрів, які переміщаються по шинах:

Rm значення номера регістру. Якщо шина A отримує своє значення з регістра, тоді ідентифікатор регістру Rm буде використано для визначення номера регістру.

- Rd - регістр мети
- Rt - регістр в пам’ять
- Rn - регістр значення на шині B

- ShAmt – значення зміщення зсуву (ShAm). ShAmt - це 5-розрядне число від 0 до 31, включене в інструкцію.

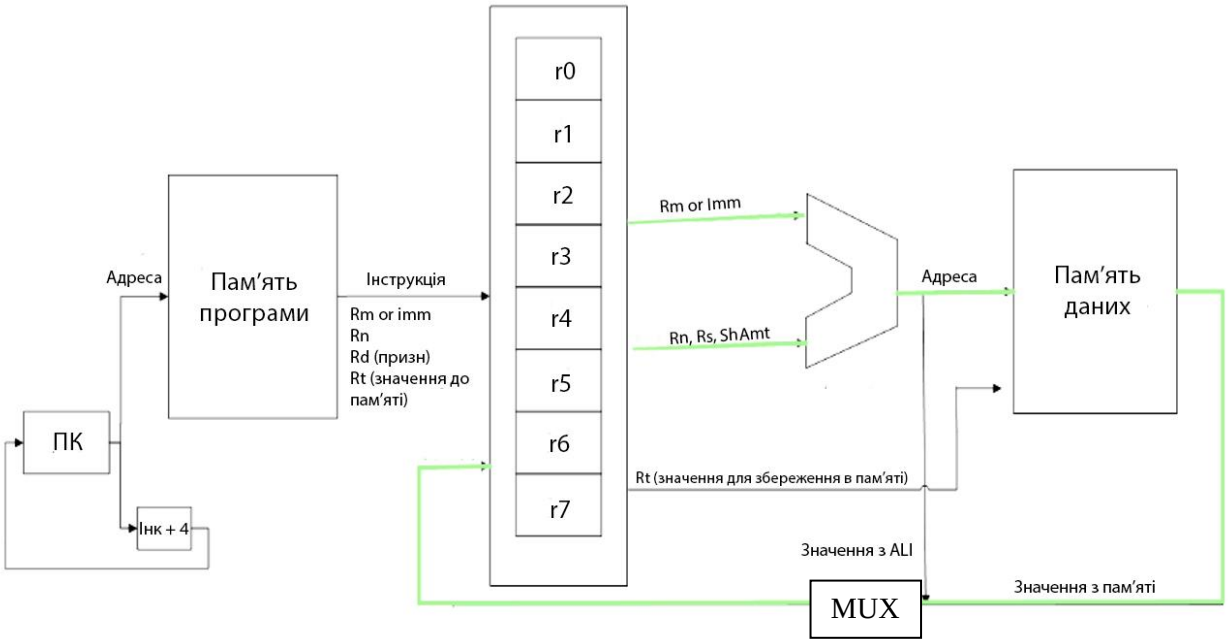

Рис. 2.3 3-адресне завантаження та зберігання ЦП

Кольором виділено операцію зберігання. Операція завантаження реалізується на одну шину вище. На Рис.2.3 показано, що для 3-адресних інструкцій ALU значення з регістрів або безпосередніх значень, створює вихідні дані та надсилає їх назад до банку регістру. Крім того реалізована операція завантаження та зберігання даних з пам'яті в регістр. Як показано на цій діаграмі, регістр Rn додається до регістра Rm або безпосереднього значення для обчислення адреси пам’яті, з якої завантажуються дані, і дані з Rn. Показана також зворотна операція завантаження даних у пам'ять. Адреса обчислюється в ALU і передається в пам'ять даних. Пам'ять даних зчитує значення за цією адресою та поміщає це значення на шину, щоб відправити його назад до банку регістрів. Зауважте, що це значення пам’яті розміщується на тій самій шині, що й вихідні дані ALU. У точці, де результати ALU та результат читання пам’яті знаходиться мультиплексор(MUX) буде вставлено, яке значення передати до банку регістрів. Rt регістр поміщається на шину C для запису в пам'ять.

Таким чином можна зрозуміти, чому тільки спеціальні інструкції мають доступ до пам’яті, бо тільки для них існує привілей доступу до шини C. Набір таких інструкцій буде розглянуте нижче.

Таким чином вирішується фундаментальна проблема кожного процесора поділу пам’яті на область даних і область коду, яка у архітектура Гарварду на визначенні

окремих розділів пам’яті для даних і коду на фізичному рівні. А в архітектура фон Неймана, розробники кожного процесора вирішують цю проблему по різному. Процесор ARM - це архітектура фон Неймана, і це може мати помітний вплив на машинний і виконуваний код, який створюється. Однак реалізація пам’яті ARM дозволяє фактично отримати доступ до пам’яті, який може виглядати так, ніби пам’ять розділена між кодом і даними. Це означає, що процесор ARM може діяти так, ніби він має гарвардську архітектуру. Це має великий вплив на конструкцію ЦП, оскільки 3-адресна схема на Рис. 2.3 розділяє пам’ять на текстову та адресну.

Питання для самоперевірки

До якого різновиду належить архітектура процесора ARM - фон Неймана чи гарвардської?

Які інструкції спеціалізуються на роботі з памяттю?

Яке архітектурне рішення в процесор ARM підтримує функції?

2.4 Конвеєрне виконання команд в процесорі ARM

Процес отримання наступної інструкції під час виконання поточної інструкції називається «конвеєрною обробкою». Для збільшення швидкості виконання програми процесор підтримує конвеєрну роботу. Це збільшує пропускну здатність. Кілька операцій виконуються одночасно, а не послідовно в конвеєрі. Конвеєр має три етапи вибірки, декодування та виконання, як показано на Рис. 2.4

Рис. 2.4 Схема конвеєрного виконання команд

У конвеєрі використовуються три етапи:

(i) Отримання: на цьому етапі процесор ARM отримує інструкцію з пам’яті.

(ii) Декодування: на цьому етапі розпізнає інструкцію, яка має бути виконана.

(iii) Виконати 2. На цьому етапі процесор обробляє інструкцію та записує результат назад у потрібний регістр.

Якщо ці три етапи виконання перекриваються, ми досягнемо вищої швидкості виконання. Такий конвеєр існує у версії 7 процесора ARM. Після заповнення конвеєра

кожна інструкція потребує одного циклу для завершення виконання. Нижче на Рис. 2.5 показано триетапні конвеєрні інструкції.

Для простих інструкцій стадія виконання триває лише один такт. Але є деякі інструкції, які мають багатоцикловий етап виконання. Наприклад, STR.

У першому циклі процесор отримує інструкцію 1 із пам'яті. У другому циклі процесор отримує інструкцію 2 із пам'яті та декодує інструкцію 1. У третьому циклі процесор отримує інструкцію 3 із пам'яті, декодує інструкцію 2 та виконує інструкцію 1. у четвертому циклі процесор отримує інструкцію 4, декодує інструкцію 3 і виконує інструкцію 2. Конвеєр, таким чином, виконує інструкцію за три цикли, тобто він забезпечує пропускну здатність, що дорівнює одній інструкції за цикл.

У випадку багатоциклової інструкції, як показано на Рис. 2.5, інструкція 2 (тобто STR інструкції збереження) потребує 4 тактових циклів і, отже, конвеєр зупиняється на одному тактовому імпульсі. Перша інструкція завершує виконання в третьому тактовому імпульсі, тоді як друга інструкція замість завершення виконання в четвертому тактовому імпульсі завершує те саме в п'ятому тактовому імпульсі. Після цього кожна інструкція завершує виконання за один тактовий імпульс, як показано на цьому Рис.2.5.

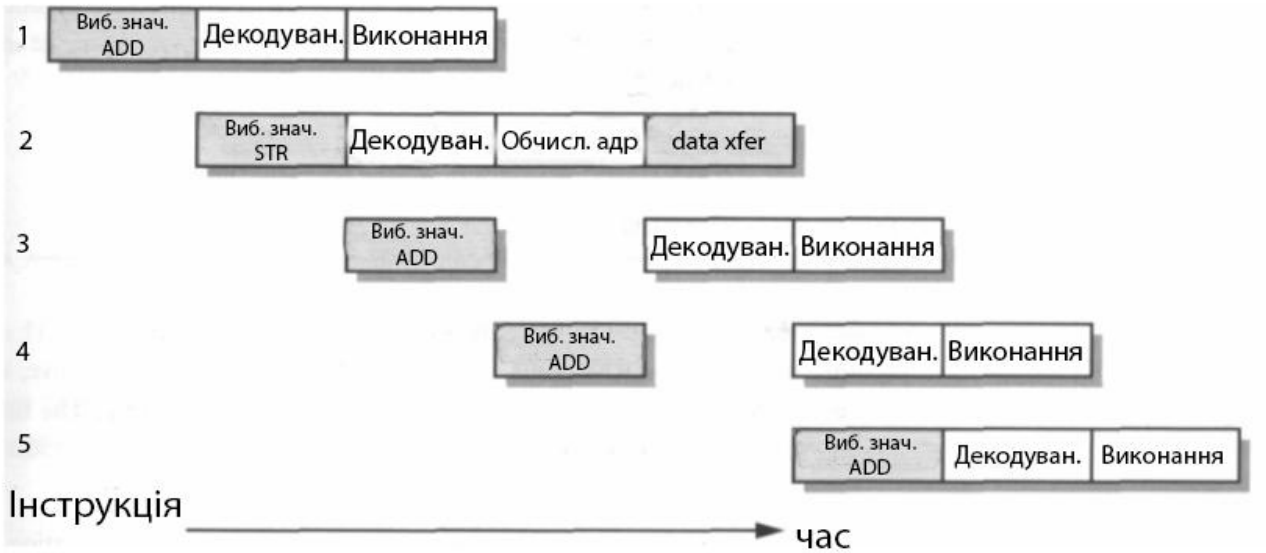

Рис. 2.5 Виконання багатоциклової інструкції

Обсяг роботи, що виконується на кожному етапі, можна зменшити шляхом збільшення кількості етапів у конвеєрі. Щоб покращити продуктивність, процесор може працювати на вищій робочій частоті.

Оскільки для заповнення конвеєра потрібна більша кількість циклів, затримка системи також збільшується. Залежність даних між етапами також може бути збільшена

зі збільшенням етапів конвеєра. Отже, інструкції мають бути заплановані під час написання коду, щоб зменшити залежність від даних.

Питання для самоперевірки

Як влаштований конвеєр команд ARM

Які особливості конвеєра для багатоциклової інструкції

Чому в архітектурі ARM конвеєр команд працює успішно?

2.5 Регістри ARM

Велика проблема для RISC - це спрощення інструкцій, що веде до збільшення їх кількості. Більше інструкцій вимагає більше пам'яті - недорогої, але повільної. Якщо програма RISC споживає більше пам'яті, ніж програма CISC, вона буде повільнішою, оскільки процесор постійно чекатиме повільного читання з пам'яті. Проектувальники RISC реалізували декілька рішень, які дозволили вирішити цю проблему. Вони помітили, що множина інструкцій переміщує дані між пам'яттю та регістрами, щоб підготуватися до виконання. Маючи велику кількість регістрів, вони змогли скоротити кількість звернень до пам'яті.

Ця особливість потребувала оптимізації компіляторів. Компілятори мають добре аналізувати програми, щоб розуміти, коли змінні можна зберігати у регістрі, а коли їх варто записати у пам’ять. Робота з множиною регістрів стала важливим завданням для компіляторів, що дозволяє прискорити роботу на RISC-процесорах.

Інструкції в RISC не мають великої кількості різних режимів адресації, тому, наприклад, серед 32-бітових команд є більше біт, щоб вказати номер регістру. Це дуже важливо. У процесорі легко можуть розміститися сотні регістрів. Це не так складно і не потребує витрат на велику кількість транзисторів. Проблема полягає у нестачі біт, що вказують адресу регістру. Так, наприклад, в x86 є лише 3 біти для вказівника регістра. Це дає нам всього $2^3 = 8$ регістрів. Процесори RISC заощаджують біти через меншу кількість способів адресації. Таким чином, для адресації використовується 5 біт, що дає $2^5 = 32$ регістра. Очевидно, що ці значення можуть відрізнятися, але тенденція зберігається.

На Рис.2.6 показана організаця регістрів ARM, де світлим фоном виділені активні регістри для даного стану процесора.

Рис. 2.6 Організація регістрів ARM

Залежно від режиму та стану процесора користувач має доступ тільки до певного набору регістрів. В ARM розробнику постійно доступні 17 регістрів:

- 13 регістрів загального призначення (r0..r12). _
- Stack Pointer (r13) - містить вказівник стека програми, що виконується, в мові асемблера часто вживається його синонім (sp).
- Link register (r14) - містить адресу повернення в інструкціях розгалуження мовою асемблера часто вживається його синонім(lr).
- Program Counter (r15) – лічильник команд біти [31:1] містять адресу виконуваної інструкції в мові асемблера часто вживається його синонім(pc).

Коли процесор працює в стані ARM: Усі інструкції мають довжину 32 біти

Усі інструкції мають бути вирівняні на довжину слова. Тому значення PC зберігається в бітах від 31 до 2 зі значеннями біт [1:0], що дорівнюють нулю (оскільки інструкція не може бути вирівняна напівсловом або байтом).

Регістр r14 використовується для зв’язку з підпрограми (LR) і зберігає адресу повернення під час виконання операцій розгалуження з посиланням, обчислену з PC.

Таким чином, щоб повернутися з пов’язаної частини коду треба виконати:

MOV r15,r14

або

MOV pc, lr

Current Program Status Register (CPSR) - містить прапори, що описують поточний стан процесора. Модифікується при виконанні багатьох інструкцій: логічних, арифметичних, та ін.

Перші чотири, r0 - r3, є регістрами аргументів та/або тимчасовими регістрами ; параметри функції надходять сюди (або в стек), і очікується, що ці регістри будуть зайняті викликаною функцією. r12 також потрапляє в цю категорію. Решта, r4-r11, також відомі як регістри змінних.

Немає обмежень до доступних на даний момент регістрів. Усі інструкції мають прямий доступ до r0 - r14. Більшість інструкцій також дозволяють використовувати лічильник команд. Щодо надання доступу до CPSR і SPSR потрібні спеціальні інструкції.

Інструкції прямого управління регістром cpsr

MRS – передача значення cpsr або spsr в регістр

MSR - передача значення регістра в cpsr або spsr

MRS{<cond>} Rd,<cpsr|spsr>

MSR{<cond>} <cpsr|spsr>_<fields>,Rm

MSR{<cond>} <cpsr|spsr>_<fields>,#immediate

В Табліці 2.1показані варіанти використання cpsr та spsr

Таблиця 2.1 Застосування команди MRS

MRS	копіювання psr до gpr	Rd = psr
MSR	пересилання gpr до psr	psr[field] = Rm
MSR	пересилання безпосереднього значення до psr	psr[field] = безпосереднє значення

У всіх режимах, крім User mode та System mode доступний також Saved Program Status Register (SPSR). Після виключення регістр CPSR зберігається в SPSR. Тим самим фіксується стан процесора (режим, стан; прапори арифметичних, логічних операцій, дозволу переривань) на момент безпосередньо перед перериванням. На Рис. 2.7 показано перехід процесора з режиму User в режим FIQ і відповідний стан регістрів ARM

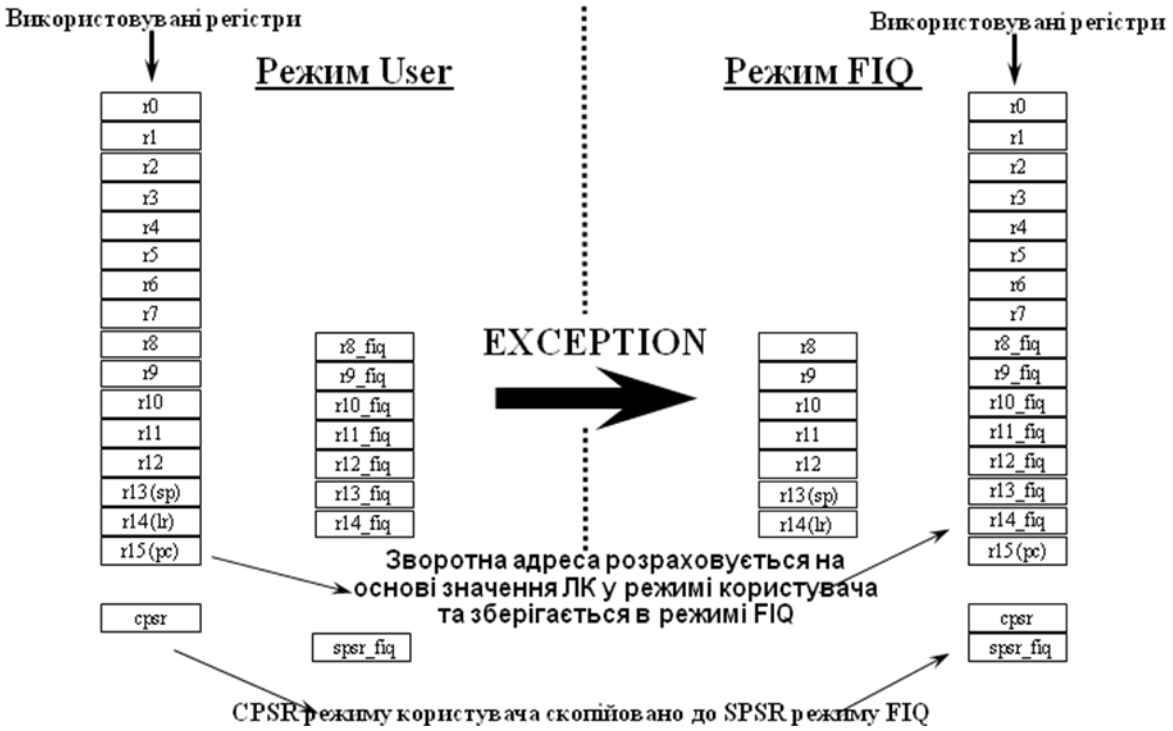

Рис. 2.7 Перехід з режиму User до режиму FIQ

На Рис. 2.8 зображено регістри статусу програми **CPSR** Current(поточний) та **SPSR** Saved(збережений. Ці регістри використовуються для зберігання бітів керування та бітів прапорів. Бітами прапорців є N, Z, C і V, а бітами керування є I, F і M0 до M4. Біти прапорців можуть бути змінені під час логічної, арифметичної операції та операції порівняння.

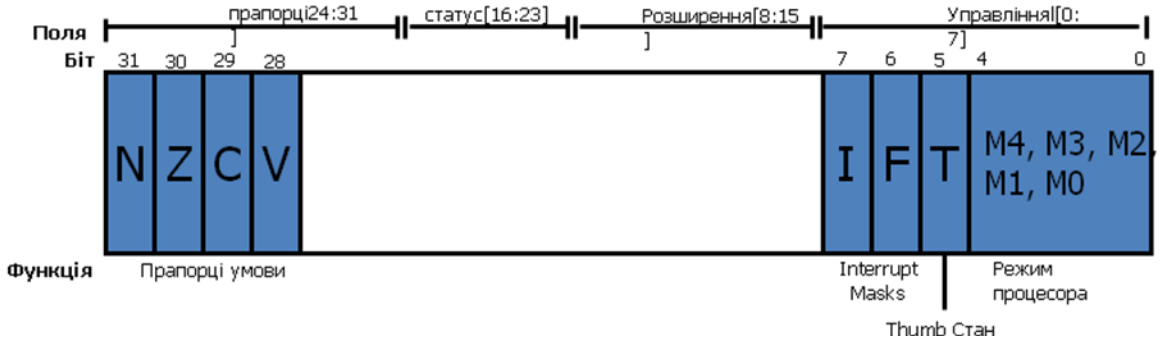

Рис. 2.8 Структура регістра CPSR (SPSR)

Біти прапорців

N (neative):N = 1 означає, що результат операції негативний, а N=0 операції є позитивним.

0 означає результат Z (нуль):

Z = 1 означає, що результат операції дорівнює нулю, а Z = 0 результат операції не нуль.

C (carry):C = 1 означає результат операції, згенерованої переносом, а C = 0 означає результат операції не призвело до перенесення.

V (переповнення): V =1 означає результат операції, яка викликала переповнення, а V=0 означає результат операції не спричинив переповнення.

Управляючі біти

I (біт переривання):Коли цей біт встановлюється в одиницю, він вимикає interrupt, а це означає процесор не приймає жодних програмних переривань.

F біт використовується для вимкнення та ввімкнення режим запиту швидкого переривання (режим FIQ). Коли цей біт встановлюється в одиницю, він вимикає FIQ, а і це приводить до того, що процесор не приймає жодних швидких переривань. Це означає, що коли ядро приймає виняток FIQ, воно автоматично маскує IRQ. IRQ не може перервати обробник FIQ. Зворотне не вірно - IRQ не маскує FIQ, тому обробник FIQ (якщо використовується) може перервати IRQ. Крім того, якщо запити IRQ і FIQ виникають одночасно, ядро спочатку оброблятиме FIQ. Код обробника FIQ зазвичай не можна написати мовою C - його потрібно написати безпосередньо мовою асемблера. Якщо необхідно використовувати FIQ, компілятор C не створюватиме код із-за обмеження на використання тільки регістрів r8-r13. Код, створений компілятором C, має бути сумісним зі ATPCS стандартом виклику процедури ARM, натомість використовуватиме регістри r0-r3для початкових значень і не створюватиме правильний cpsrкод повернення в кінці функції.

M4, M3, M2, M1 та M0 це біти режиму, і вони дорівнюють 10000 для режиму користувача.

T (Біт стану): T = 1 процесор, що виконує інструкції Thumb, T = 0 процесор виконанує інструкції ARM.

Під час виконання попередньої команди лічильник команд містить значення, відповідне адресі того слова пам'яті, по якому розташовується перший байт наступної призначеної для виконання команди. Перший байт команди відводиться для запису коду операції (КОП). Код операції вказує, які дії мають бути виконані над даними, а також вид оброблюваних даних. Таким чином, цикл виконання команди починається зі зчитування з пам'яті першого байта команди, який містить код операції. Якщо відповідно до коду операції виявляється, що другий і третій байти команди разом утворюють адресу даних, призначених для обробки, то ці 2 байти мають бути переписані в ЦП. Після цього ЦП знає, де слід шукати необхідні дані.

Питання для самоперевірки

В яких режимах працює процесор ARM7

Скільки регістрів загального призначення має ARM7

Чи є якісь умовні правила використання регістрів загального призначення ARM7?

Чи є строга спеціалізація серед регістрів загального призначення r0-r15 ARM7?

Яке стандартне використання регістрів r13, r14, r15 в ?

Яке призначення регістра Current Program Status Register (CPSR)?

Якими команди мають доступ до CPSR?

Навіщо потрібен регістр SPSR?

В чому призначення біт I та F регістра CPSR?

2.6 Режими адресації ARM

Центральний процесор може отримувати доступ до операндів (даних) різними способами, які називаються режимами адресації. Кількість режимів адресації визначається при проектуванні мікропроцесора і не може бути змінена. За допомогою розширених режимів адресації можливий доступ до різних типів даних і структур даних (наприклад, масивів, вказівників, класів), але ми тут не будемо це обговорювати, обмежимось базовими:

1.регістровий

2. безпосередній

3. непрямий регістровий (режим індексованої адресації)

Режим адресації регістрів

Режим адресації регістрів передбачає використання регістрів для зберігання даних, якими потрібно маніпулювати. Пам'ять не доступна, коли виконується цей режим адресації; отже, це відносно швидко. На Рис. 2.9 показана регістрова адресація d – кількість біт для адресації.

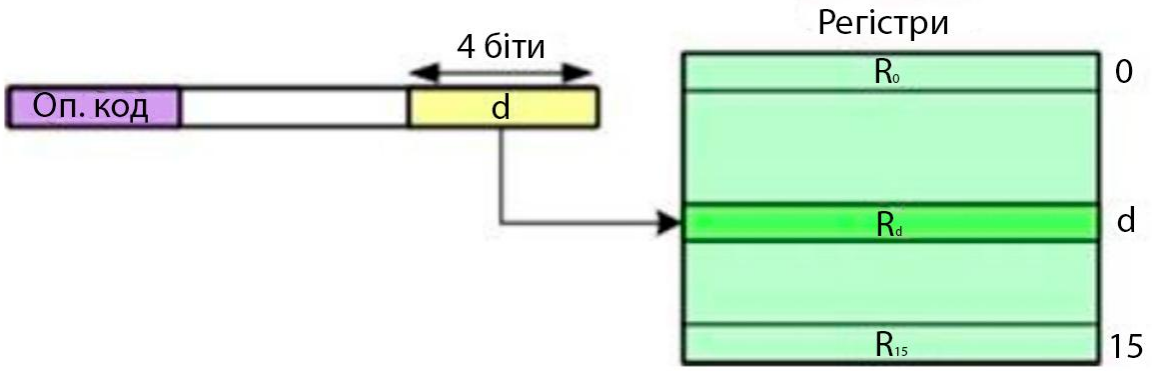

Рис. 2.9 Режим адресації регістрів

Програма мовою асемблера ARM для додавання деяких даних і збереження SUM у R3.

MOV R6,R2 ;скопіювати вміст R2 у R6

ADD R1,R1,R3 ;додати вміст R3 до вмісту R1

SUB R7,R7,R2 ;відніміть R2 від R7

Режим безпосередньої адресації У режимі безпосередньої адресації операнд джерела є константою. У режимі безпосередньо адресації, як випливає з назви, коли інструкція збирається, операнд приходить відразу після коду операції. З цієї причини цей режим адресації виконується швидко. На Рис. 2.10 показана схему безпосередньої адресації, де К- кількість біт для адресації.

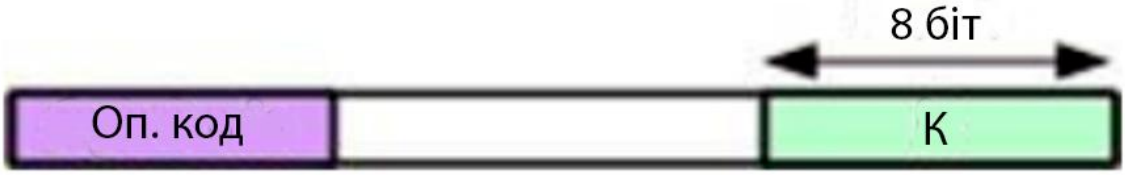

Рис. 2.10 Схема безпосередньої адресації

Приклади:

MOV R9,#0x25 ;перемістити 0x25 у R9

MOV R3,#62 ;завантажити десяткове значення 62 у R3

ADD R6,R6,#0x40 ;додати 0x40 до R6

У перших двох режимах адресації операнди знаходяться або всередині мікропроцесора, або позначені тегами разом з інструкцією. У більшості програм дані, що підлягають обробці, часто знаходяться в якому місці пам’яті поза ЦП. Існує багато способів доступу до даних у просторі пам’яті даних. Далі описується один із методів.

Режим непрямої адресації (режим індексованої адресації)

У режимі непрямої адресації регістр адреса ділянки пам’яті, де знаходиться операнд, міститься у регістрі. Див. Рис. 2.11

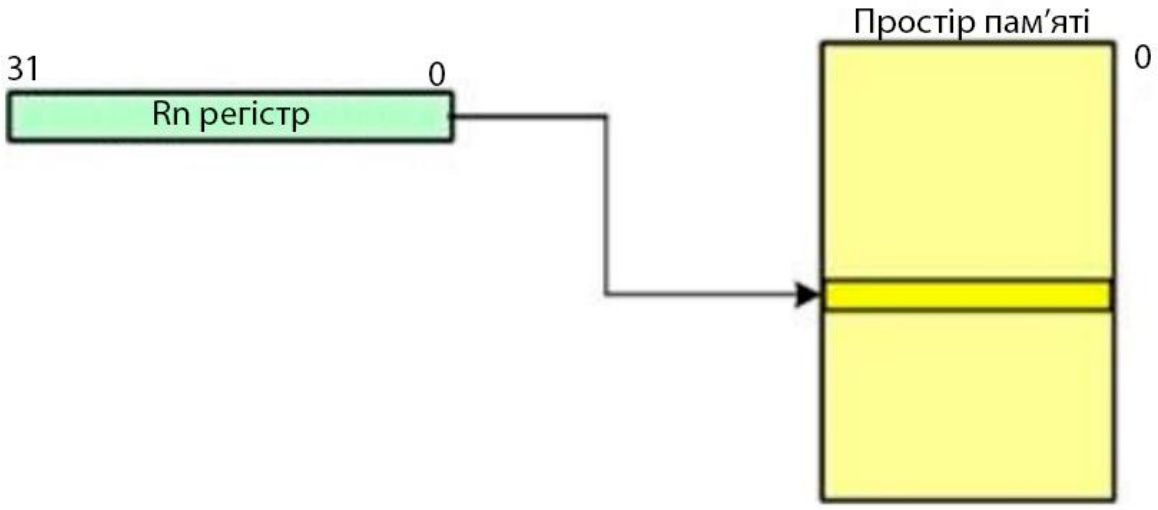

Рис. 2.11 Схема непрямої адресації

Приклад використання: режим непрямої адресації

Використовуючи режим непрямої адресації, ми можемо реалізувати різні вказівники. Оскільки регістри 32-розрядні, вони можуть адресувати весь простір пам'яті.

Тут ви бачите простий код на С та його еквівалент на Ассемблері:

мовою С:

```
char *ourPointer;
ourPointer = (char*) 0x12456; //Вказати на розташування 12456
*ourPointer = 25; //зберігаємо 25 у розташуванні 0x12456
ourPointer ++; //вказати на наступне місце
```

мовою асемблера:

```
LDR R2,=0x12456 ;вказати на адресу 0x12456
MOV R0,#25 ;R0 = 25
STRB R0,[R2] ;зберігати R0 у адресі 0x12456
ADD R2,R2,#1 ;збільшити R2, щоб вказати на наступне адресу.
```

Залежно від типу даних, на який вказує вказівник, можуть використовуватися різні варіанти команд завантаження: STR/LDR, STRH/LDRH або STRB/LDRB. У наведеному вище прикладі, оскільки він вказує на char (який є 8-бітним), використовується STRB.

Питання для самоперевірки

Чому для режиму адресації регістрів достатньо виділити не більше чотирьох біт?

Чому для режиму безпосередньої адресації в у форматі команди достньо поля розміром один байт?

Який адресний простір адресюється в непрямому режимі?

2.7 Машинні команди(інструкції) процесора ARM

Архітектура ARM підтримує 16-розрядний набір інструкцій Thumb-2 та 32-розрядний набір інструкцій. Більшість інструкцій ARM використовують три операнди. Ці інструкції класифікуються на основі формату інструкцій і операцій, які перераховані нижче:

- Інструкції з обробки даних.
- Інструкції одиночного обміну даними.
- Інструкції зміщення та обертання.
- Безумовні інструкції та умовні інструкції.
- Операції зі стеком.
- Розгалудження(Переходу).
- Інструкції множення.
- Інструкції передачі даних.

Особливістю набору інструкцій ARM в тому, що коли коли процесор працює в стані User ARM, то:

Виконання інструкцій відбувається за принципом load-store.

Кожна інструкція має 3-адресацію.

Умовне виконання кожної інструкції.

Можливе завантаження/збереження кількох регістрів одночасно.

Можливість комбінування операцій shift і ALU в одній інструкції.

На Рис. 2.12 представлено формат машинної команди процесора ARM

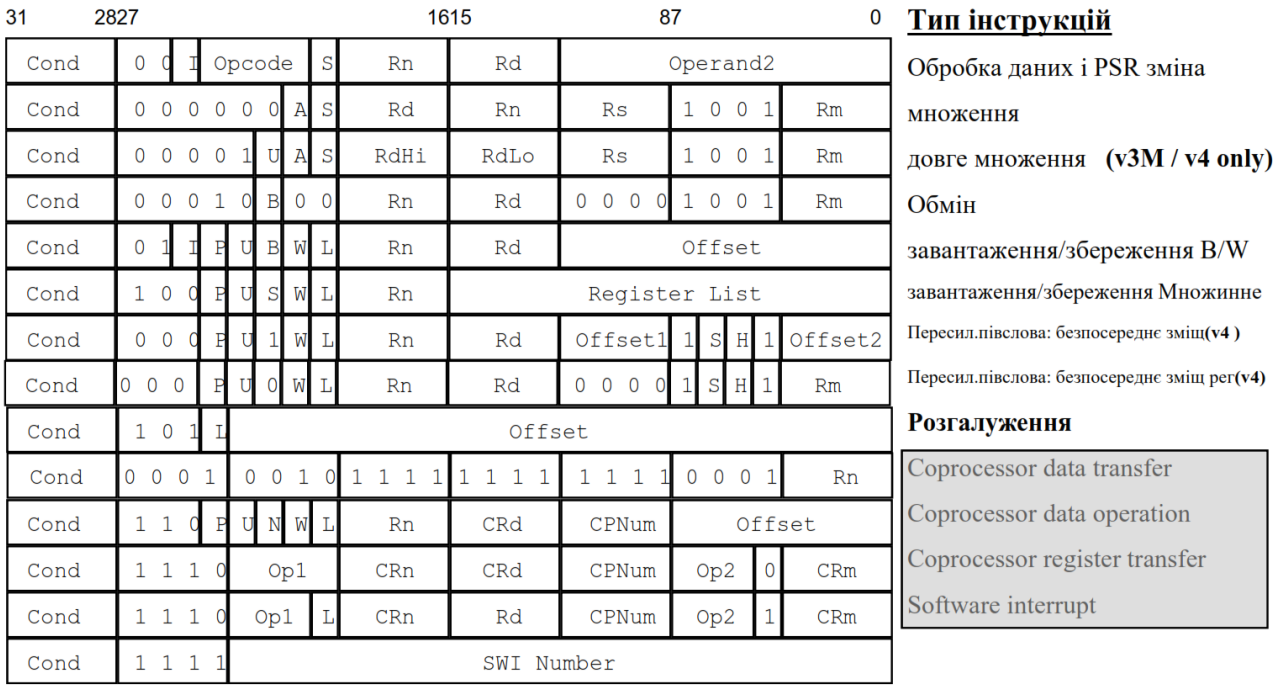

Рис. 2.12 Формат машинної команди процесора ARM

Як ми бачимо всі команди мають фіксовану довжину розміром 4 байти. Також всі мають поле умови (31-28 біти) виконання кожної інструкції. Цей факт здається досить дивним. Проте це цікава ідея розробників ARM. Насправді більшість наборів інструкцій дозволяють лише умовне виконання розгалужень. Однак, повторно використовуючи апаратне забезпечення оцінки умов, ARM ефективно збільшує кількість інструкцій. Усі інструкції містять поле умови, яке визначає, чи буде процесор їх виконувати. Невиконані інструкції поглинають один цикл. А ще потрібно завершити цикл, щоб отримати та декодувати наступні інструкції. Це усуває потребу у багатьох відгалуженнях, які зупиняють ковеєр (3 цикли для заповнення).В результаті отримється дуже щільний вбудований код без розгалужень. Втрати на невиконання кількох умовних інструкцій часто менший, ніж накладні витрати на розгалуження або виклик підпрограми, які були б необхідні в іншому випадку. Щоб умовно виконати інструкцію, просто зафіксуйте її відповідною умовою:

Наприклад, інструкція додавання має такий вигляд:

ADD r0,r1,r2 ; r0 = r1 + r2 (ADD)

Щоб виконати це, лише якщо встановлено нульовий прапор:

ADDEQ r0,r1,r2 ; Якщо встановлено нульовий прапор, тоді… ; ... *r0 = r1 + r2*

За замовчуванням операції обробки даних не впливають на позначки умов (за винятком порівнянь, де це єдиний ефект). Щоб спричинити оновлення позначок умови,

біт S інструкції має бути встановлений постфіксом інструкції (і будь-якого коду умови) літерою «S».

Наприклад, щоб додати два числа та встановити позначки умови:

ADD r0,r1,r2 ; r0 = r1 + r2; . і встановити прапори.

Інструкції розгалудження

Змінюють потік виконання або використовується для виклику процедури. На Рис. Приведена структура команди

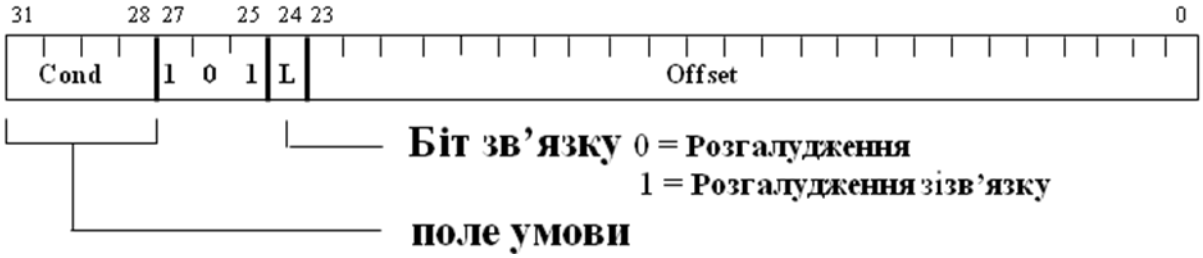

Рис. 2.13 Структура команди розгалудження

Розгалуження мітка: B{<cond>}

Розгалуження з посиланням: BL{<cond>} sub_routine_label

Зсув для інструкцій розгалуження обчислюється асемблером:

Взявши різницю між інструкцією розгалуження та цільовою адресою мінус 8 (щоб врахувати конвеєр).

Зсув для інструкцій розгалуження обчислюється асемблером:

Взявши різницю між інструкцією розгалуження та цільовою адресою мінус 8 (щоб врахувати конвеєр).

Це дає 26-бітове зміщення, яке зсувається праворуч на 2 біти (оскільки нижні два біти завжди дорівнюють нулю, оскільки інструкції вирівнюються за словом) і зберігається в кодуванні інструкцій.

Це дає діапазон ± 32 Мбайт

Приклади:

Інструкція BL зберігає адресу повернення на R14 (lr)

BL sub @ call sub CMP R1, #5 @ повернутися сюди

MOVEQ R1, # 0 ... sub: ... @ sub точка входу ...

MOV PK, LR @ повернення

Інструкція розгалуження з посиланням, або BL, подібна до інструкції B, але перезаписує регістр посилання адресою повернення. Щоб повернутися з підпрограми, регістр посилань слід скопіювати в PC підпрограма BL ;

розгалуження до підпрограми

CMP r1, #5 ; порівняння r1 з 5

MOVEQ r1, #0 ;

if (r1 == 5) then r1 = 0 повернення шляхом переміщення pc = lr

Це дає 26-бітове зміщення, яке зсувається праворуч на 2 біти (оскільки нижні два біти завжди дорівнюють нулю, оскільки інструкції вирівнюються за словом) і зберігається в кодуванні інструкцій.

Це дає діапазон ± 32 Мбайт

Інструкції з обробки даних

Найбільше сімейство інструкцій ARM, усі мають однаковий формат інструкцій.

Містить:

- Арифметичні дії
- Порівняння (немає результатів - просто встановіть коди умов)
- Логічні операції
- Переміщення даних між регістрами

Інструкції з обробки даних такі:

AND, EOR, SUB, RSB, ADD, ADC, SBC, RSB, TST, TEQ, CMP, CMN, ORR, MOV, BIC і MNW.

Вони виконуються за принципом load/stor, Причому ці інструкції працюють лише з регістрами, а не з пам’яттю. Кожен з них виконує певну операцію над одним або двома операндами. Перший операнд завжди регістр – Rn. Другий операнд, надісланий до ALU через шіфтер Барела.

Дані інструкції обробки використовують регістрові операнди та безпосередній операнд. Загальний формат інструкцій з обробки даних:

Мнемоніка {S}{Умова} Rd, Rn, операнд 2

Мнемоніка: Мнемоніка – це абревіатура операції додавання, наприклад ADD – додавання.

Якщо інструкція містить S, це означає оновлення бітів прапора регістра стану процесора (PSR).

Умова: Умова, за якої інструкція буде виконана, якщо умова відповідає

Rd: Rd це регістр призначення

Rn: Rn є операндом

Операнд 2: Операнд 2 може бути регістром або безпосереднім значенням

А. Регістри Операнди Операнди знаходяться в регістрах. Перший регістр є регістром призначення, другий регістр є operandl і третій регістр є operand2.

Нижче наведено інструкції арифметичних і логічних операцій із операндами регістрів.

Арифметичні інструкції

ADD операнд1 + операнд2

ADC операнд1 + операнд2 + перенесення

SUB операнд1 - операнд2

SBC операнд1 - операнд2 + перенесення -1

RSB операнд2 - операнд1

RSC операнд2 - операнд1 + перенесення - 1

Синтаксис:

• <Операція>{<cond>}{S} Rd,

Приклади

ADD r0, r1, r2 ;r0=r1+r2 Додавання вміст регістра r1 до регістра r2 і помістіть результат у регістр *r0.*

ADC r0, r1, r2; ;r0 = r1+r2+C Додавання з перенесенням С - біт перенесення.

ABC r0, r2, r3; ;r0=r2-r3+C-1 SUB з переносом.

SUB r0, r2, r3 ;r0=r2–r3 r0=r2-r3, Віднімання, де r2 - перший операнд, а r3 - другий операнд

SBC r0, r2, r3; ;r0=r2-r3C-1 Віднімання SUB з перенесенням.

RSB r0, r2, r5 ;r0= r5-r2 Зворотнє віднімання SUB.

RSC r0, r2, r5 ;r0=r5-r2 C-1 Зворотнє віднімання з перенесенням.

Логічні операції

AND логічне побітове AND of two 32 bit values Rd = Rn & N

ORR логічне побітове OR of two 32 bit values Rd = Rn | N

EOR logical exclusive OR of two 32 bit valuesRd = Rn ^ N

BIC Logical bit clear (AND NOT) Rd = Rn & ~ N

Синтаксис:

<Операція>{<cond>}{S} Rd, Rn, Операнд2

приклади:

I r0, r1, r2

BICEQ r2, r3, # 7

EOPC r1,r3,r0

AND r0, r3, r5 ;r0= r3 AND r5.

ORR r7, r3, r5; ;r7=r3 OR r5.

EOR r0, r1, r2 ;r0 = r1 Exclusive OR with r2.

BIC r0, r1, r2; Біт очистки. Перший у другому операнді очищає відповідний біт та зберігає результати в регістрі призначення.

Нехай вміст r1 становить 1111111111011111, а r2 становить 10000100 1110 0011 після виконання *BIC r0. r1,r2 r0* містить 0111 101100011100.

Інструкції переміщення даних

* Операції:

MOV операнд2

MVN не операнд2

Зверніть увагу, що вони не використовують операнд 1.

Синтаксис:

<Операція>{<cond>}{S} Rd, Операнд2

* приклади:

MOV r0, r1

MOVS r2, # 10

MVNEQ r2, # 10 r1,#0

Приклад :

MOV R2, # 0x45, вміст R2 буде 0x00000045

B. MOV Ro, Rm, 1s1 # n ; зрушити Rm n разів вліво і зберегти результат в Rn

C. Умовний MOV

MOVEQ R2, 0x56; якщо встановлено нульовий біт, виконується MOVEQ

Інструкції зі зсуву та повороту

ARM поєднав операції повороту та зсуву з іншими інструкціями;

Процесор ARM виконує наступні операції зсуву:

LSL, LSR, ASR,ROR

Немає єдиної інструкції, яка завантажувала б 32-розрядну безпосередню константу в регістр без виконання завантаження даних із пам’яті. Усі інструкції ARM мають 32 біти, причому інструкції ARM не використовують потік інструкцій як дані.

Формат інструкції обробки даних має 12 біт, доступних для операнда 2. Якщо використовувати напряму, це дасть лише діапазон 4096. Замість цього він використовується для зберігання 8-бітових констант, що дають діапазон від 0 до 255. Ці 8 бітів потім можна обертати на парну кількість позицій.

Це дає набагато більший діапазон констант, які можна безпосередньо завантажити, хоча деякі константи все одно потрібно буде завантажити з пам’яті.

Це дає нам:

0 – 255 [0 - 0xff]

256,260,264,..,1020 [0x100-0x3fc, крок 4, 0x40-0xff або 30]

1024,1040,1056,..,4080 0x400-0xff0, крок 16, 0x40-0xff або 28]

4096,4160, 4224,..,16320 [0x1000-0x3fc0, крок 64, 0x40-0xff або 26]

Їх можна завантажити, наприклад, за допомогою:

MOV r0, #0x40, 26; => MOV r0, #0x1000 (тобто 4096)..

Щоб зробити це легше, асемблер перетворить для нас цю форму, якщо просто дати необхідну константу:

MOV r0, #4096 ; => MOV r0, #0x1000 (тобто 0x40 ror 26)

Порозрядні доповнення також можна сформувати за допомогою MVN:

MOV r0, #0xFFFFFFFF ; збирається до MVN r0, #0

Якщо потрібну константу неможливо створити, буде повідомлено про помилку.

Однак це не всі операції переміщення. Значна частина інструкцій переміщення базується на безпосередньому використанню шифтера Барела перед виконанням в АЛП На Рис. 2.14 показана схема використання шифтера Барела.

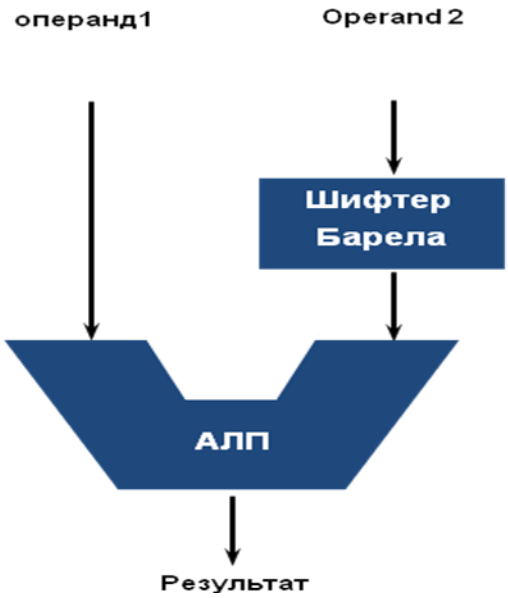

Рис. 2.14 Схема використання шифтера Барела

Значення зсуву може бути таким: 5-бітне ціле число без знаку Вказано в нижньому байті іншого регістра. Використовується для множення на константу Безпосереднє значення 8-розрядне число з діапазоном 0 - 255. Повернуто праворуч на парну кількість

позицій, що дозволяє завантажувати розширений діапазон 32-розрядних констант безпосередньо в регістри. В Таблиці 2.2 надані варіанти використання інструкцій шифтера Барела.

Таблиця 2.2 Інструкції шифтера Барела

Інструкція шифтера Барела	**Синтаксис**	**Семантика**
LSL	Rm, LSL #shift_imm	Логічний зсув вліво на константу
LSL	Rm, LSL Rs	Логічний зсув вліво за значення регістру
LLS	Rm, LSR #shift_imm	Логічний зсув праворуч на константу
LSR	Rm, LSR Rs	Логічний зсув праворуч за значення регістру
ASR	Rm, ASR #shift_imm	Арифметичний зсув ASR праворуч на константу
ASR	Rm, ASR Rs	Арифметичний зсув праворуч за значення регістру
ROR	Rm, ROR #shift_imm	Обертати праворуч за допомогою константу
ROR	Rm, ROR Rs	Обертання праворуч за значення регістру
ROR	Rm, RRX	Поверніть праворуч із розширенням

Використання інструкції множення для множення на константу означає спочатку завантаження константи в регістр, а потім очікування певної кількості внутрішніх циклів для завершення інструкції. Краще рішення часто можна знайти, використовуючи певну комбінацію MOV, ADD, SUB та RSB зі зсувами.

Множення на константу, що дорівнює а $^{((\text{степінь } 2) \pm 1),}$ можна виконати за один цикл.

Приклад:

*r0 = r1 * 5*

*r0 = r1 + (r1 * 4)*

ADD r0, r1, r1, LSL #2

Приклад:

*r2 = r3 * 105*

*= r3 * 15 * 7*

*= r3 * (16 - 1) * (8 - 1)*

*RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15*

*RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7*

Встановлення прапорців PSR. Наведені вище інструкції не впливають на біт прапора PSR, оскільки інструкції не мають опції S. Додавши суфікс S до інструкції, інструкція вплине на біт прапора.

Інструкції порівняння та тестування

* **Operations are:**
- CMP operand1 - operand2, but result not written
- CMN operand1 + operand2, but result not written
- TST operand1 AND operand2, but result not written
- TEQ operand1 EOR operand2, but result not written

* **Синтакс:**
- <Operation>{<cond>} Rn, Operand2

* **Examples:**
- CMP r0, r1
- TSTEQ r2, #5

Інструкції та тестування

Єдиним ефектом порівнянь є оновлення прапорців умов.Тому не потрібно встановлювати біт S.

Операції:

CMP операнд1 - операнд2, але результат не записується.

CMN операнд1 + операнд2, але результат не записується.

TST I операнд2, але результат не записується.

TEQ операнд1 операнд1 EOR операнд2, але результат не записується.

Інструкції порівнячння та перевірки

Процесор ARM використовує інструкції порівняння та тестування для встановлення бітів прапора PSR, а далі є інструкції порівняння та тестування. CMP, CMN, TST і TEQ, ці інструкції використовують два операнди для порівняння та перевірки, результат їх операцій не записується в жодний регістр. Інструкція CMP (Інструкція порівняння) Інструкція CMP має такий формат:

CMP Операнд1, Операнд2

Інструкція CMP порівнює операнд1 з операндом2; ця інструкція віднімає Операнд2 від Операнда1 та встановлює відповідний прапор. Біт прапора встановлюється на основі результату операції наступним чином

Прапор Z встановлюється, якщо операнд2 дорівнює операнду1

Прапор N встановлюється, якщо операнд1 менший за операнд2

Прапор C встановлюється, якщо результат операції генерує перенесення

Приклад. Нехай R1 містить 0x00000024, а R2 містить 0x00000078; операція CMP R1, R2 встановить прапор N на 1.

CMP Rd, безпосереднє значення, безпосереднє значення може бути 8 біт, наприклад

CMP R1, #0xFF

CMN Compare Negate CMN має такий формат

CMN Операнд1, Операнд2

Інструкція додасть операнд 1 до операнда 2 і встановить відповідний біт прапора.

Приклад. Нехай R1 містить 0x00000024, а R2 містить 0x13458978: операція CMN R1, R2 із перенесенням результату та встановленням прапорця C на 1.

TST (тестова інструкція) Тестова інструкція має такий формат TST Oprand1, Operand2

Тестова інструкція виконує операцію I між операндом1 і операндом2 і встановлює відповідний біт прапора. Операнд може бути безпосереднім значенням або регістром, таким як *TST R1, R2*;

Ця інструкція виконує операції *R1 AND R2* і встановлює відповідний прапор.

.Інструкції множення

Певним парадоксом виглядає винесення інструкцій множення в окремий розділ. Проте - це насправді глибоко продумана ідея, пов’язана з необхідністю прискорення виконання конвеєра команд. Як відомо саме множення є однією з найбільш трудомістких операцій. Тому цю операцію винесли з межі АЛП і її виконує схема, яка реалізує алгоритм Бута

Базовий ARM забезпечує дві інструкції множення.

Помножити *MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs*

Multiply Accumulate - робить додавання ще й додавання

*MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn*

Обмеження щодо використання: Rd і Rm не можуть бути одним і тим же регістром. Можна уникнути, помінявши місцями Rm і Rs. Це працює, оскільки множення

комутативне. Не можна використовувати PC. Операнди можна вважати знаковими або беззнаковими. Правильне тлумачення залежить від користувача. ARM використовує алгоритм Бута для множення цілих чисел. На не-M ARM це працює з 2 бітами Rs одночасно. Для кожної пари бітів це займає 1 цикл (плюс 1 цикл для початку).

Приклад: помножте 18 на -1: Rd = Rm * Rs

Примітка. Компілятор не використовує критерії раннього завершення, щоб вирішити, в якому порядку розмістити операнди.

MUL R0, R1, R2 @ R0 = (R1xR2)[31:0]

Особливості:

– Другий операнд не може бути безпосереднім

– Регістр результату має відрізнятися від першого операнда

– Цикли залежать від типу ядра

– Якщо встановлено біт S, прапор C не має значення

Множення-накопичення (індексація двовимірного масиву)

MLA R4, R3, R2, R1 @ R4 = R3xR2+R1

Множення на константу часто може бути ефективніше реалізовано за допомогою операнда зміщеного регістра

MOV R1, №35 MUL R2, R0, R1

or

ADD R0, R0, R0, LSL #2 @ R0'=5xR0

RSB R2, R0, R0, LSL #3 @ R2 =7xR0'

Розширені інструкції множення

M варіанти ядер ARM містять розширене апаратне забезпечення множення. Це забезпечує три покращення:

- використовується 8-розрядний алгоритм Бута
- множення виконується швидше (максимум для стандартних інструкцій тепер становить 5 циклів).
- метод раннього завершення покращено, тепер множення завершується, коли містяться всі інші набори бітів усі нулі (як у не-M ARM), або всі одиниці.

Таким чином, попередній приклад завершився б достроково через 2 цикли в обох випадках.

64-розрядні результати тепер можна отримати з двох 32-розрядних операндів з більшою точністю. Необхідна пара регістрів, що використовуються для зберігання результату. Однак усі 64 біти результату тепер мають значення (інструкції множення з нижчою

точністю просто відкидають верхні 32 біти). Тому необхідно вказати, чи є операнди знаковими чи беззнаковими.

Тому синтаксис нових інструкцій такий:

UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs

Приклад: Множення для 32 розрядів

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2   ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

Приклад: Множення для 64 розрядів

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3   ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Інструкції завантаження/збереження (Load / Store)

ARM - це архітектура завантаження/збереження, що означає наступне:

1.В процесорі ARM не підтримується операції обробки даних бепосередньо з пам’яті.

2.Перед їх використанням необхідно перемістити значення даних у регістри.

Це може здатися неефективним, але на практиці це не так:

Завантаження значення даних із пам’яті в регістри.

Дані обробляються в регістрах за допомогою спеціального набору інструкцій обробки даних, які не сповільнюються доступом до пам'яті.

Результати зберігаються з регістрів у пам'яті.

ARM має три набори інструкцій, які взаємодіють з основною пам'яттю. Це:

- Однорегістрова передача даних (LDR / STR).

- Блокування передачі даних (LDM/STM).
- Одиночний обмін даними (SWP).

Основні інструкції щодо завантаження та зберігання

Завантажити та зберегти слово або байт:

LDR / STR / LDRB / STRB

Архітектура ARM версії 4 також додає підтримку напівслів і підписаних даних.

Завантажити та зберегти півслова:

LDRH / STRH

Завантажити байт зі знаком або півслова - значення завантаження та знак розширюють його до 32 бітів.

LDRSB / LDRH

Усі ці інструкції можна умовно виконати, вставивши відповідний код умови після STR / *LDR.*

Наприклад

LDREQB

Синтаксис:

<LDR|STR>{<cond>}{<size>} Rd, <address>

Завантажити та зберегти слово або байт: базовий регістр

На Рис. 2.15 показана схема завантаження у базовий регістр. Область пам'яті, до якої потрібно отримати доступ, зберігається в базовому регістрі STR r0, [r1] ; Збереження значення r0 у вказаному місці ; за значенням r1.

LDR r2, [r1] ; Завантажте r2 значенням місця пам'яті на що вказує значення r1.

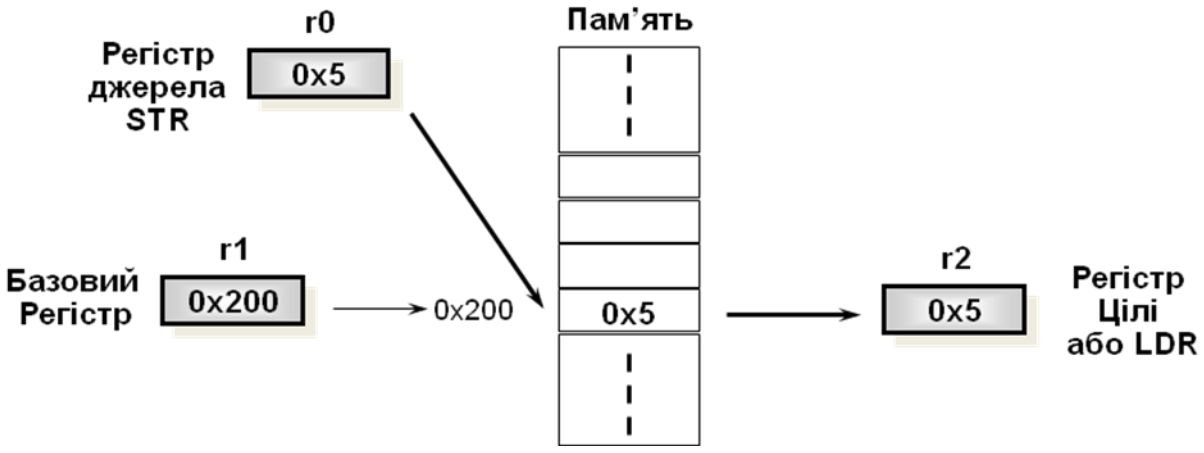

Рис. 2.15 Завантаження у базовий регістр

Завантаження та збереження слова або байта: зміщення з базового регістру

Окрім доступу до фактичного розташування, що міститься в базовому регістрі, ці інструкції можуть отримати доступ до зсуву розташування відносно вказівника базового регістру.

Це зміщення може бути беззнаковим 12-бітним безпосереднім значенням (тобто 0 - 4095 байт). Регістр, необов'язково зміщений на безпосереднє значення. Це можна або додати, або відняти з базового регістру:

Додати перед значенням зсуву «+» (за замовчуванням) або відняти«-».

Це зміщення можна застосувати до здійснення передачі.

Попередньо проіндексована адресація

Необов'язково автоматично збільшувати базовий регістр, постфіксуючи інструкцію «!».

після здійснення передачі: Постіндексована адресація

внаслідок чого базовий регістр автоматично збільшується

Завантаження та збереження Word або Byte: попередньо проіндексована адресація

Приклад: STR r0, [r1,#12] показано на Рис.2.16

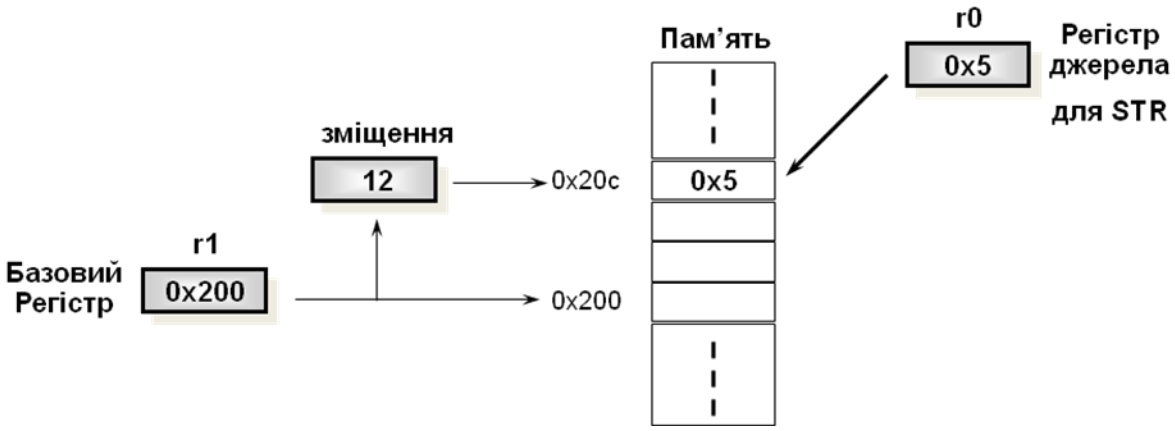

Рис. 2.16 Попередньо проіндексована адресація

Для збереження до розташування 0x1f4 натомість використовуйте:

STR r0, [r1,#-12]

Щоб автоматично збільшити базовий вказівник до 0x20c, використовуйте:

STR r0, [r1, #12]!

Якщо r2 містить 3, отримати доступ до 0x20c, помноживши це на 4:

STR r0, [r1, r2, LSL #2]

Завантажити та зберегти Word або Byte: попередньо проіндексовану адресацію

На Рис 2.17 показаний приклад виконання: STR r0, [r1], #12

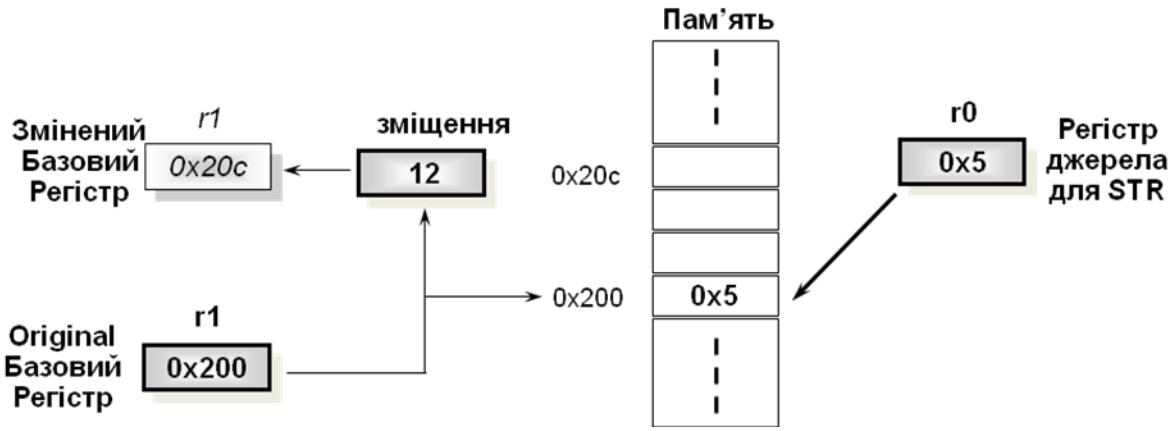

Рис.2.17 Зміщення з базового регістру

Щоб автоматично збільшити базовий регістр до розташування 0x1f4, треба використати: STR r0, [r1], #-12

Якщо r2 містить 3, автоматично збільшити базовий регістр до 0x20c, помноживши це на 4:

STR r0, [r1], r2, LSL #2

На Рис. 2.18 показано приклад постіндексної адресації для масиву.

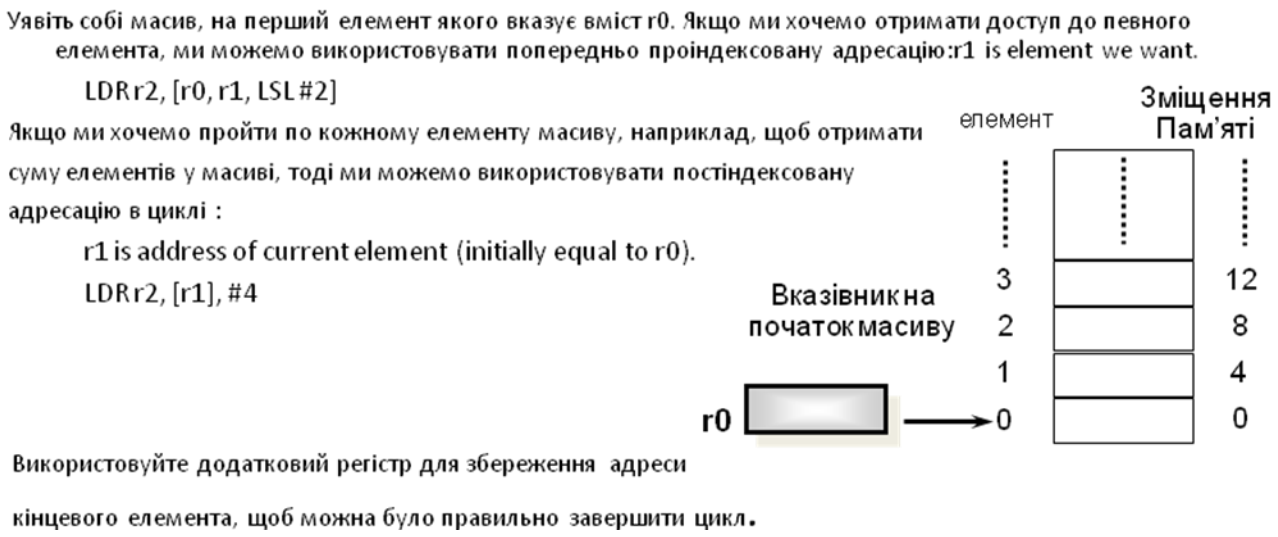

Рис. 2.18 Використання адресації для масиву

Питання для самоперевірки

В чому полягає особливість формату команд процесора ARM?

Чому всі команди процесора ARM умовні?

Який формат команди додавання процесора ARM?

Який формат команди розгалудження процесора ARM?

З чого складається набір логічних операцій процесора ARM?

Яка особливість формату переміщення даних процесора ARM?

Як інструкції зсуву використовують шіфтер Барела ?

Який синтакс команд порівняння процесора ARM?

Чому команди множення процесора винесені в окремий підрозділ?

Які існують варіанти для використання інструкцій завантаження/збереження (Load / Store)?

2.8 Операції зі стеком

Традиційно стек зростає в пам'яті вниз, а останнє «передане» значення знаходиться за найнижчою адресою. ARM також підтримує висхідні стеки, де структура стека зростає в пам'яті.

Значення вказівника стека може вказати на останню зайняту адресу (Повний стек), а тому потребує попереднього декрементування (тобто перед заштовхуванням).

Може вказати на наступну зайняту адресу (порожній стек)

– тому потребує постдекрементування (тобто після заштовхування)

Тип стека, який буде використовуватися, визначається постфіксом інструкції:

STMFD / LDMFD : повний спадний стек

STMFA / LDMFA: повний стек за зростанням.

STMED / LDMED: порожній спадний стек

STMEA / LDMEA : порожній за зростанням стек

* Примітка: компілятор ARM завжди використовуватиме повний спадний стек.

Архітектура ARM використовує кілька інструкцій для зберігання навантажень для виконання операцій зі стеком

За зростанням – стек зростає до вищої адреси пам'яті

За спаданням – стек зростає до меншої адреси пам'яті

Повний – sp вказує на останній елемент у стеку

Пусто - sp точки після останнього елемента в стеку ARM Thumb Procedure Call Standard (ATPCS) визначає, як викликаються процедури та як розподіляються регістри.

В ATPCS стеки визначаються як повні спадні стеки.

Інструкції LDMFD і STMFD забезпечують функції виймання та заштовхування стека

Методи адресації для операцій зі стеком подані у Таблиці 2.3.

Таблиця 2.3 Методи адресації для операцій зі стеком

Режим додавання	**Опис**	**Pop Заштовхування**	**=LDM**	**Push Заштовхування**	**=STM**
FA	Повне зростання	LDMFA	LDMDA	STMFA	STMIB
FD	Повне спадання	LDMFD	LDMIA	STMFD	STMDB
EA	Порожній зростаючий	LDMEA	LDMDB	STMEA	STMIA
ED	Порожній спадаючий	LDMED	LDMIB	STMED	STMDA

На Рис 2.19 показані приклади виконання стекових операцій.

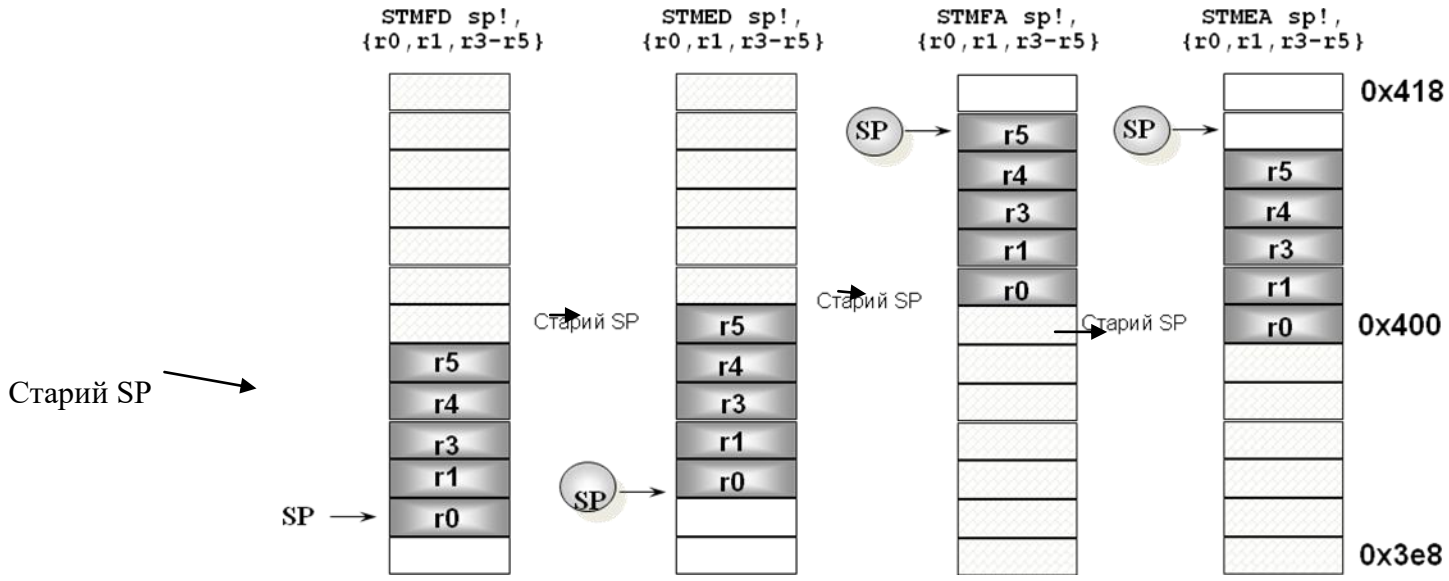

Рис. 2.19 Приклади виконання стекових операцій

Питання для самоперевірки

З якими стеками можуть працювати команди процесора ARM?

Який регістр показує на останній елемент у стеку?

Які інструкції застосовуються для спадних стеків?

Які інструкції застосовуються для стеків за зростанням?

Які методи адресації для операцій зі стеком?

Чи є в наборі команд ARM інструкція типу ret?

2.9 Організація переривань для ARM

Процесори ARM працюють під управлінням різних операційних систем, зокрема для Linux та Windows. Це означає, що вони мають підтримувати такі особливості таких систем як захищений режим адресації та механізм переривань.

ОС Linux для ARM інструкцію SWI для входу в простір ядра з простору користувача, Команда SWI використовується для генерації програмного переривання, щоб перейти з режиму користувача в режим управління, CPSR зберігається в SPSR в режимі управління, а виконання передається у вектор SWI. Інструкція SWI також може використовуватися в інших режимах і процесор аналогічно перемикається в режим керування. Формат команди наступний:

SWI{cond} immed_24

де: immed_24 24-бітне безпосереднє значення, ціле від 0 до 16777215.

При використанні інструкції SWI передачі параметрів зазвичай використовуються два методи: обробник винятків SWI може надавати пов'язані послуги. Обидва методи є угодами про програмне забезпечення користувача.:

1.OABI (Old ABI)

SWI<суфікс> <номер>

2.EABI (Extended ABI)

mov r7, #num

swi 0x0

Вихідний системний виклик такий,

swi (#num | 0x900000) (0x900000 - магічне значення)

Тобто оригінальний метод виклику (Old ABI) виконується за наступним номером виклику в інструкції swi, а поточний базується на значенні r7.

SWI : програмне переривання

SWI<суфікс> <номер>

Це простий засіб, але, можливо, найбільш використовуваний. Багато засобів операційної системи надаються через виконання SWI. SWI означає програмне переривання. В RISC OS SWI використовуються для доступу до програм операційної

системи або модулів, створених третьою стороною. Багато програм використовують модулі для забезпечення зовнішнього доступу низького рівня для інших програм.
Прикладами SWI є:

- Інтерфейси Filer SWI, які допомагають читати диск і з нього, налаштовувати атрибути тощо.
- SWI драйвера принтера, який використовується для допомоги у використанні паралельного порту для друку.
- SWIs FreeNet/Acorn TCP/IP стек SWIs, які використовуються для передачі та отримання даних за допомогою протоколу TCP/IP, який зазвичай використовується для надсилання даних через Інтернет.

При такому використанні SWIs дозволяє операційній системі мати модульну структуру, тобто код, необхідний для створення повної операційної системи, можна розбити на кілька невеликих частин (модулів) і обробник модуля. Коли обробник SWI отримує запит на певний номер підпрограми, він знаходить позицію підпрограми та виконує її, передаючи будь-які дані.

Давайте подивимося, як ви ним користуєтеся. Інструкція SWI (на асемблері) виглядає так:

SWI &02

або

SWI "OS_Write0"

Обидві ці інструкції фактично однакові, і, отже, складатимуться з однією інструкцією. Єдина відмінність полягає в тому, що друга інструкція використовує рядок для представлення номера SWI, який є &02. Коли використовується програма, написана з використанням рядка, рядок спочатку шукається перед виконанням. Наразі опустимо подробиці роботи з рядками. Вони часто використовуються для забезпечення ясності програми, але не є фактичними інструкціями, які виконуються.

Давайте знову подивимося на першу інструкцію:

SWI &02

Що це означає? Формально це означає необхідність ввести обробник SWI та передати значення &02. В RISC OS ARM це означає виконання підпрограми номер &02. Питання в тому, як він це робить, як він передає номер SWI і вводить обробник SWI? Подивіться на простому прикладі дизасемблінг перших 32 байтів пам’яті (розташування 0-&1C) і можна побачити фактичні інструкції ARM з адресами, щось на зразок цього:

```
Адреса          Зміст          Дизасемблінг
00000000 : E5000030 : STR      R0,[R0,#-48]
00000004 : E59FF31C : LDR      PC,&00000328
00000008 : E59FF31C : LDR      PC,&0000032C
0000000C : E59FF31C : LDR      PC,&00000330
00000010 : E59FF31C : LDR      PC,&00000334
00000014 : E59FF31C : LDR      PC,&00000338
00000018 : E59FF31C : LDR      PC,&0000033C
0000001C : E3A0A632 : MOV      R10,#&3200000
```

Виключаючи першу та останню інструкції (які є особливими випадками), можна побачити, що всі інструкції завантажують PC (лічильник програм), який повідомляє комп'ютеру, звідки виконати наступну інструкцію, з новим значенням. Значення береться з адреси в пам'яті, яка також відображається.

Все, що SWI робить, це змінює режим на Supervisor і налаштовує ПК на виконання наступної інструкції за адресою &08! Переведення процесора в режим Supervisor вимикає 2 регістри r13 і r14 і замінює їх на r13_svc і r14_svc. Під час входу в режим супервізора r14_svc також буде встановлено на адресу після інструкції SWI.

Як раніше вж вказувалось, адреса &08 містить інструкцію, яка переходить на іншу адресу, це та адреса, де знаходиться справжній обробник SWI!

Насправді саме значення значення з регістру процесор ігнорує. Обробник SWI отримує його, використовуючи передане значення r14_svc.

Послідовність дій обробника переривань (після збереження регістрів r0-r12):

1. Він віднімає 4 від r14, щоб отримати адресу інструкції SWI.
2. Завантажує інструкцію в реєстр.
3. Очищає останні 8 бітів інструкції, позбавляючись від OpCode і надаючи лише номер SWI.
4. Використовує це значення для пошуку адреси підпрограми коду, який буде виконуватися (за допомогою таблиць пошуку тощо).
5. Відновлює регістри r0-r12.
6. Виводить процесор із режиму супервізора.
7. Перехід до адреси підпрограми.

Ось приклад коду з таблиці даних ARM:

0x08 B Supervisor

```
EntryTable
 DCD ZeroRtn
 DCD ReadCRtn
 DCD WriteIRtn
...
Read EQU 0
Прочитайте C EQU 256
Write EQU 512
; SWI має процедуру, необхідну для бітів 8-23 і даних
; (якщо є) у бітах 0-7.
; Припускає, що R13_svc вказує на відповідний стек
STMFD R13, {r0-r2, R14}
 ; Збережіть робочі реєстри та зворотню адресу
LDR R0,[R14,#-4]
 ; Отримати інструкцію SWI.
BIC R0, R0, #0xFF000000
 ; Очистити верхні 8 бітів.
MOV R1, R0, LSR #8
 ; Отримайте регулярне зміщення.
ADR R2, EntryTable
 ; Отримати початкову адресу входу
 ; стіл.
LDR R15,[R2,R1,LSL #2]
 ; Перейдіть до відповідної процедури.
WriteIRtn
 ; Wnte із символом у бітах R0 0–7.
.....
 LDMFD R13, {r0-r2, R15}^
 ; Відновлення робочої області та повернення, відновлення
 ; режим і прапори процесора.
```

Виклик супервізора (SVC) і виклик служби очікування (PendSV) є двома винятками, націленими на програмне забезпечення та операційні системи. SVC призначений для генерації викликів системних функцій. Наприклад, замість того, щоб дозволяти програмам користувача отримувати прямий доступ до обладнання, операційна система може надавати доступ до обладнання через SVC. Отже, коли користувальницька програма хоче використовувати певне апаратне забезпечення, вона генерує виняток SVC за допомогою інструкцій SVC, який переводить процесор в в режим супервізора, а потім виконується програмний обробник винятків в операційній системі та надає послугу, яку запитує програма користувача. Коректність виконання переривань в режимі супервізора забезпечують з боку ОС набір функцій API, з боку процесора блок управління пам’яттю. На Рис. 2.20 показана схема реалізації переривання для архітектури ARM

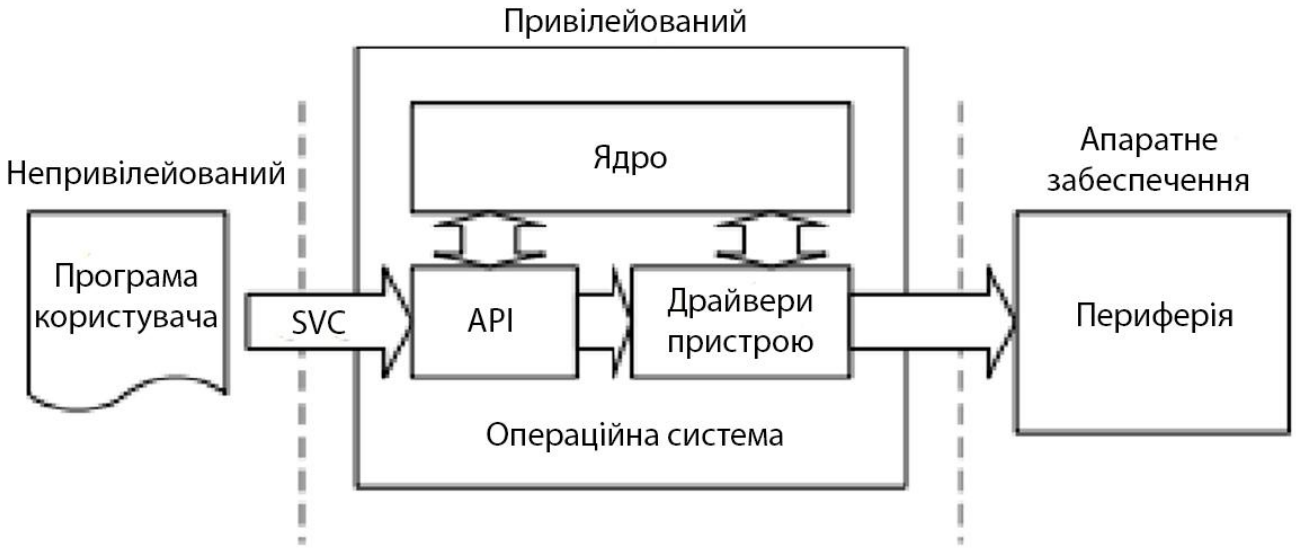

Рис.2.20 Реалізація переривання для архітектури ARM

Таким чином, доступ до обладнання знаходиться під контролем ОС, яка може забезпечити більш надійну систему, запобігаючи користувацьким програмам від прямого доступу до обладнання. SVC також може зробити програмне забезпечення більш портативним, оскільки програмі користувача не потрібно знати деталі програмування апаратного забезпечення. Програмі користувача знадобиться лише ідентифікатор функції та параметри інтерфейсу прикладного програмування (API). Фактичне програмування на апаратному рівні обробляється драйверами пристроїв. Виняток SVC генерується за

допомогою інструкції SVC. Для цієї інструкції потрібне безпосередне значення, який працює як метод передачі параметрів. Потім обробник винятків SVC може витягнути параметр і визначити, яку дію він повинен виконати.

Наприклад,

SVC #0x3; Виклик функції SVC 3

Традиційний синтаксис для SVC також прийнятний (без «#»):

SVC 0x3; Виклик функції SVC 3

Приклад: У цій програмі демонструються операції з файлом та використовується стек як буфер.

```
.global _start
_start:
;Системний виклик відкриття файлу
ldr r0, =filename
mov r1, #0101 ; O_WRONLY | O_CREAT
ldr r2, =0666 ; Дозвіл
mov r7, #5 ; 5 номер на відкриття
svc #0; застосування svc по EABI
cmp r0, #0; застосування svc по EABI
blt exit
;r0 містить fd (дескриптор файлу)
mov r5, r0
;системний виклик
;використання стека як буфера
sub sp, #8 ; стек повністю спадає, таким чином ми залишаємо простір
strb r1, [sp]
mov r1, #'2'
strb r1, [sp, #1]
mov r1, #'\n'
strb r1, [sp, #2]
mov r1, sp
mov r2, #3
```

```
mov r7, #4 ; 4 номер виклику на запис
svc #0
;fsync приведення у стан синхронізованого завершення введення-виведення
mov r0, r5; повернення дескриптора
mov r7, #118; номер fsync
svc #0
;close syscall
mov r7, #6 ; 6 номер на закриття
svc #0
exit:
mov r0, #0
mov r7, #1
svc #0
.data
.align 2
filename: .asciz "out.txt"
```

Питання для самоперевірки

Якою командою генерується програмне переривання в ARM?

Який синтаксис команди SWI?

Послідовність дій обробника переривань після збереження регістрів r0-r12?

Яка схема реалізації переривання для архітектури ARM?

Яку роль в реалізації переривань грає переключення режиму процесора?

Як застосовується інструкцій SVC для доступу до апаратного забезпечення?

2.10 Директиви асемблера ARM

У той час як інструкції повідомляють центральному процесору, що робити, директиви (також звані псевдоінструкціями) дають вказівки асемблеру. Саме це спільне застосування власне і є асемблером. Наприклад, інструкції MOV і ADD є командами для центрального процесора, але EQU, END і ENTRY є директивами для асемблера. Далі будуть представлені деякі широко використовувані директиви ARM і те, як вони використовуються. Директиви допомагають нам не тільки легше розробляти нашу

програму та роблять нашу програму зрозумілішою (більш читабельною), але що ще більш важливіше забезпечують утворити правильний машинний код.

AREA Наказує асемблеру зібрати новий код або розділ даниха END інформує асемблер, що він досяг кінця вихідного файлу.

Директива AREA вказує асемблеру визначити новий розділ пам'яті. Пам’ять може бути кодом (інструкцією) або даними та може мати такі атрибути, як ReadOnly, ReadWrite тощо. Це широко використовується для визначення одного або кількох блоків неподільного пам'ять для коду або даних, які будуть використовуватися компонувальником. Кожна програма мови асемблера має принаймні одну AREA. Нижче наведено формат:

AREA назва розділу, атрибут, атрибут, …

Наступний рядок визначає нову область під назвою MY_ASM_PROG1, яка має атрибути CODE і READONLY:

AREA MY_ASM_PROG1, КОД, READONLY:

Серед широко використовуваних атрибутів є CODE, DATA, READONLY, READWRITE, COMMON і ALIGN. Нижче наведено опис цих широко використовуваних атрибутів.

READWRITE - це атрибут, наданий області пам’яті, з якої можна читати та записувати. Оскільки це розділ програми READWRITE, він за замовчуванням призначений для DATA.

У мові асемблера ARM ми використовуємо цю область для виділення пам’яті SRAM для блокнота та стека. Асемблер розміщує розділи READWRITE один біля одного в SRAM-пам'ять.

READONLY - це атрибут, наданий області пам’яті, з якої можна лише читати. Оскільки це розділ програми READONLY за замовчуванням, він призначений для CODE. У мові асемблера ARM ми використовуємо цю область для написання наших інструкцій для виконання машинного коду.

Розділи READONLY розміщені поруч один з одним у флеш-пам’яті.

У мові асемблера ARM ми використовуємо цю область для написання наших інструкцій. Наступний рядок визначає нову область для написання програм:

AREA OUR_ASM_PROG, CODE, READONLY

DATA - це атрибут, наданий області пам’яті, яка використовується для даних, і жодні інструкції (машинні інструкції) не можуть бути розміщені в цій області. Оскільки він використовується для розділу даних програми, за замовчуванням це пам’ять

READWRITE. У мові асемблера ARM ми використовуємо цю область для виділення пам’яті SRAM для блокнота та стека. Наступний рядок визначає нову область для визначення змінних:

ОБЛАСТЬ OUR_VARIABLES, DATA, READWRITE

Щоб визначити постійні значення у флеш-пам'яті, ми пишемо наступне:

AREA OUR_CONSTS, DATA I, READONLY

COMMON – це атрибут, наданий для області розділу пам’яті DATA, який може бути зазвичай використовується кількома програмними кодами. Ми не ініціалізуємо розділ COMMON пам’яті, оскільки він використовується виключно компілятором. Компілятор ініціалізує COMMON область пам'яті з усіма нулями.

ALIGN - ще один атрибут, наданий області пам’яті, щоб вказати, як пам’ять має розподілятися відповідно до адрес. Коли ALIGN використовується для CODE і

READONLY він вирівнюється за 4-байтовою межею адреси за замовчуванням, оскільки всі інструкції ARM є 32-бітними (4-байтовими) словами. Атрибут ALIGN AREA має номер після like ALIGN=3, значить, що інформація має бути розміщена в пам’яті з адресами кратними 2^3, тобто 0x50000, 0x50008, 0x50010, 0x50020 тощо.

ENTRY

Іншим важливим псевдокодом є директива ENTRY. Це вказує асемблеру на початок виконуваного коду. Директива ENTRY є першим рядком розділу коду мови асемблера ARM у програмі, що означає, що все, що йде після директиви ENTRY у вихідному коді, вважається фактичною машинною інструкцією, яку має виконати ЦП. Для даної програми мови асемблера ARM ми можемо мати лише одну точку ENTRY. Наявність кількох директив ENTRY у програмі на мові асемблера викличе помилку асемблера.

Директива EQU дає символічне ім'я числової константи, відносного значення регістру або відносного значення ПК.

Директива INCLUDE додає вміст файлу до нашої програми.

Іншим важливим псевдокодом є директива END. Це вказує асемблеру на кінець вихідного (asm) файлу. Директива END є останнім рядком асемблера ARM

це означає, що все, що йде після директиви END у вихідному коді, ігнорується асемблером.

Приклад Програма на мові асемблера ARM для додавання даних і збереження суми у R3

AREA PROG_2_1, CODE, READONLY

ENTRY

MOV R1, #0x25 ;R1 = 0x25

MOV R2, #0x34 ;R2 = 0x34

ADD R3, R2,R1 ;R3 = R2 + R1

END

Реалізація програм мовою асемблера ARM потребує його присутності на платформі, де ця програма створюється. Однак такі платформи далеко не всім доступні. Проте є простий вихід – скористатися одним з емуляторів у вільному доступі. Наразі їх декілька. Як встановити та використовуати для програмування описано у Додатку В цього посібника.

Питання для самоперевірки

Для чого призначена директива AREA?

З якими атрибутами може використовуватись директива AREA?

Яке призначеннядирективи ENTRY?

Яке призначення директиви EQU?

Яке призначення директиви INCLUDE?

Яке призначення директиви END?

СПИСОК ЛІТЕРАТУРИ

1. Ata Elahi Computer Systems Digital Design, Fundamentals of Computer Architecture and Assembly Language © Springer International Publishing AG 2018, 269 p.
2. Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. Publication No. 24592. Revision Date 3.22. December 2017.
3. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Submitted: May 01, 2018 Last updated: May 27, 2020. – режим доступу: https://software.intel.com/content/www/us/en/develop/download/ intel-64-and-ia-32-architectures- sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.
4. Jim Ledin Modern Computer Architecture And Organization, Year: 2020 – режим доступу: https://vdoc.pub/documents/modern-computer-architecture-and-organization-2vg32cu96brg
5. Code: The Hidden Language of Computer Hardware and Software Kindle Edition by Petzold Charles (Author) 2019 576 p.
6. IA-32 Intel Architecture Software Developer's Manual: Instruction Set Reference A-M. —Order Number: 253666.
7. IA-32 Intel STRUCTURED COMPUTER ORGANIZATION ANDREW S. TANENBAUM Vrije Universiteit Amsterdam, The Netherlands TODD AUSTIN University of Michigan Ann Arbor, Michigan, United States 801 p. SIXTH EDITION 2013.
8. David A. Patterson. David R. Ditzel, The case for the reduced instruction set computer ACM SIGARCH Computer Architecture NewsVolume 8Issue 6October 1980 pp 25–33https://doi.org/10.1145
9. Paul A. Carter November PC Assembly Language, 2019 190 p – режим доступу : https://pacman128.github.io/static/pcasm-book.pdf
10. IA-32 Intel Architecture Software Developer's Manual: Basic Architecture. — Order Number: 253665.
11. Architecture Software Developer's Manual: Instruction Set Reference N-Z. —Order Number: 253667.
12. IA-32 Intel Architecture Software Developer's Manual: System Programming Guide. — Order Number: 253668.
13. P6 Family of Processors. Hardware Developer's Manual. — September 1998. — Order Number: 244001-001.
14. Small Computer System Interface – 2. Working Draft X3T9.2. Project 375D. — Revision 10L.
15. Рисований О.М. Системне програмування: підручник для студентів напрямку "Компютерна інженерія" вищих навчальних закладів в 2-х томах. Том 1. – Видання четверте: виправлено та доповнено – Х.: "Слово", 2015. – 576 с.

16. Ирвин К.Р. Язык ассемблера для процессоров Intel. - 4-е изд. - Москва: Издательский дом «Вильямс», 2005. - 912 с.
17. IA-32 Intel Architecture Software Developer’s Manual: Basic Architecture. - Order Number: 253665.
18. IA-32 Intel Architecture Software Developer’s Manual: Instruction Set Reference A-M. – Order Number: 253666.
19. IA-32 Intel Architecture Software Developer’s Manual: Instruction Set Reference N-Z. – Order Number: 253667.
20. IA-32 Intel Architecture Software Developer’s Manual: System Programming Guide. – Order Number: 253668.
21. P6 Family of Processors. Hardware Developer’s Manual. - September 1998. - Order Number: 244001-001.
22. Small Computer System Interface – 2. Working Draft X3T9.2. Project 375D. - Revision 10L.
23. https://www.ics.uci.edu/~aburtsev/143A/lectures/ https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Book%3A_x86-64_Assembly_Language_Programming_with_Ubuntu_(Jorgensen)/18%3A_Floating-Point_Instructions
24. https://www.singlix.com/trdos/archive/pdf_archive/X86%20Assembly%20float%20point.pdf
25. http://csapp.cs.cmu.edu/2e/waside/waside-x87.pdf
26. https://pdos.csail.mit.edu/6.828/2018/readings/ia32/IA32-1.pdf
27. https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
28. https://cs.lmu.edu/~ray/notes/nasmtutorial/
29. https://poli.cs.vsb.cz/edu/soj/down/soj-syllabus.pdf
30. ACM, IEEE Computer Society, "Computing Curriculum 2001".
31. Nelson, V,P, Theys, M,D, Clements, A, “Computer Architecture and Organization in the Model Computer Engineering Curriculum”, Frontiers in Education, Boulder 2003.
32. Clements, A, “The undergraduate curriculum in computer architecture”, IEEE Micro, Volume 20, No. 3, pp13-22
33. Patterson, D,A, Hennessy, J, L, “Computer Organization and Design”, Morgan Kaufmannn 4th edition 2009
34. Clements, A, “Principles of Computer Hardware”, Oxford University Press 3rd edition, 2004.

35. Brewer, E,W, “Professor’s Role in Motivating Students to Attend Class”, Journal of Industrial Teacher Education, Vol. 42, No. 3, 2005.
36. Hohl, W, “ARM Assembly Language – Fundamentals and Techniques”, CRC Press, 2009
37. http://en.wikipedia.org/wiki/ARM_Holdings
38. Goudge, L, Segars, S, “Thumb: Reducing the cost of 32-bit RISC performance in portable and consumer applications”, Proc. COMPCON ’96.

ДОДАТОК А. ТЕХНОЛОГІЯ ПРОГРАМУВАННЯ В MASM32

А.1 ОТРИМАННЯ ОБ'ЄКТНОГО КОДУ В MASM32

Підготовлені тексти програми з розширенням *.asm є вхідними даними для програм, які називаються асемблерами (наприклад, програми Masm, Tasm, Wasm, Fasm). Завдання програми асемблера – перетворити текст програми з набором символічних команд у форму двійкових команд, які може виконати МП. Але насправді цього недостатньо, бо на етапі перетворення тексту асемблера в набір машинних команд (ПАО) отримують файли об'єктних модулів, що мають розширення *.obj. Процес асемблінгу частіше називають ***трансляцією***.

Для простоти використання асемблера Masm32 (www.masm32. com) його необхідно інсталювати на системний диск c:/. Якщо поставити асемблер Masm32 не на системний диск, а на інший логічний диск, то в такому випадку необхідно конкретно вказувати шляхи підключення бібліотек розташування API-функцій.

Програма ml.exe з Masm32 виконує трансляцію початкового тексту програми в проміжний об'єктний файл. Програма link.exe виконує компонування об'єктного (об'єктних) файлу (файлів) і бібліотек в єдину виконувану програму. В асемблері masm32 трансляцію файлів із розширенням asm можна виконати **з командного рядка** (якщо програма складається з одного файлу):

ml.exe /c /coff ім'я_файлу.asm

Створений за допомогою цієї команди об'єктний файл має формат COFF. Якщо параметр /coff не заданий, то форматом створеного об'єктного файлу буде OMF.

Параметр /C означає, що виконується тільки асемблінг. Виклик лінкеру link.exe поки не виконується, тому що його викликаємо вручну пізніше.

Опції компіляції ML.EXE:

ML [/ опції] список_файлів [/link опції_лінкеру]

/AT– дозволити надмаленьку модель пам'яті tiny (.COM файл);

/Bl<linker> – використовувати альтернативний лінкер;

/c – асемблювати без лінкування (використання зовнішнього лінкеру (наприклад link.exe) для компонування файлів);

/Cp – зберегти регістр ідентифікаторів користувача;

/Cu – перевести всі ідентифікатори у верхній регістр;

/Cx – зберегти регістр publics, externs;
/coff – генерувати об’єктний файл у форматі COFF;
/D<name>[=text] – визначити текст макросу;
/EP – виводити лістинг препроцесора в стандартний потік виведення;
/F <hex> – встановити розмір стека (в байтах);
/Fe<file> – встановити ім’я виконуваного файлу;
/Fl[file] – генерувати лістинг;
/Fm[file] – генерувати карту пам’яті;
/Fo<file> – ім’я об’єктного файлу;
/FPi – генерувати 80x87 емулятор;
/Fr[file] – встановити обмежену інформацію;
/FR[file] – встановити повну інформацію;
/G<c|d|z> – використовувати формат виклику в стилі Pascal, C або Stdcall;
/H<number> – встановити максимальну довжину зовнішніх імен;
/I<name> – додати include-шлях;
/link <опції лінкеру та бібліотеки>;
/nologo – заборонити повідомлення ML.EXE про авторське право;
/Sa – максимізувати лістинг;
/Sc – видавати таймінги в лістинг;
/Sf – видавати лістинг першого проходження асемблера;
/Sl<width> – встановити ширину рядка;
/Sn – подавити лістинг таблиці символів;
/Sp<length> – встановити довжину сторінки;
/Ss<string> – встановити підзаголовок;
/St<string> – встановити заголовок;
/Sx – скласти список умовних виразів помилок;
/Ta<file> – асемблювати не .ASM файли;
/w – аналогічно /W0 /WX;
/WX – розцінювати попередження як помилку;
/W<number> – встановити рівень попереджень;
/X – проігнорувати INCLUDE шлях оточення;
/Zd – додати номери рядків у налагоджувальну інформацію;
/Zf – зробити всі символи загальнодоступними;
/Zi – додати символічну налагоджувальну інформацію;

/Zm – забезпечити сумісність із MASM5.10;

/Zp[n] – встановити вирівнювання структур;

/Zs – виконати тільки перевірку синтаксису.

Для трансляції та лінкування *простих* програм **зручно** використовувати середовище **masm32**. Для цього необхідно запустити програму **masm32 Editor**, вставити або набрати свою програму, зберегти її з розширенням asm та вибрати шлях меню **Project/ Assemble & Link**. Якщо помилок не буде, то буде створений exe-файл.

А.2 Отримання виконуваного файлу програми в Masm32

Після успішної трансляції початкового тексту асемблерної програми результат у вигляді об'єктного файлу передається компонувальнику link.exe. Він виконує об'єднання об'єктних модулів в один файл. Сегменти, визначені в програмі, групуються відповідно до інструкцій, що містяться в об'єктному файлі. Вся інформація про розміщення сегментів записується в заголовок виконуваного файлу. Результатом компонування є виконувані файли, що мають розширення *.exe.

Компонувальник link.exe оперує з OBJ-файлами як у форматі COFF, так і у форматі OMF, при цьому виконується автоматичне перетворення формату файлу з OMF в COFF. Зазвичай при генерації виконуваних файлів використовується формат COFF. Якщо файл об'єктного модуля повинен використовуватися у застосуванні, написаному на Visual C++ .NET, то формат його обов'язково повинен бути COFF. У той же час при використанні об'єктного файлу у застосуванні, розробленому в Borland Delphi 2005, єдиним прийнятним форматом є OMF.

У процесі трансляції початкового тексту програми виконуються такі дії:

1. Аналізуються директиви умовного асемблінгу, і у випадку істинності вказаних у них умов виконуються ті або інші кроки.
2. Розгортаються макроси.
3. Обчислюються константні вирази, при цьому вони заміщаються обчисленими значеннями.
4. Декодуються команди та операнди, що не знаходяться в пам'яті. Наприклад, декодується команда mov EAX, 2008, оскільки вона не має операндів, розташованих у пам'яті.
5. Зберігаються зсуви змінних у пам'яті відносно зміщення у сегментах, в яких ці змінні розташовані.
6. Сегменти та їх атрибути розміщуються в об'єктному файлі.

7. В об’єктному файлі зберігаються адреси, що переміщуються (relocatable addresses).

8. За необхідності створюється файл лістингу.

9. Безпосередньо програмі link.exe передаються деякі директиви (наприклад, INCLUDELIB і DOSSEG).

В асемблері masm32 для генерування 32-розрядних EXE-файлів може бути використаний один із командних рядків:

link.exe /SUBSYSTEM:WINDOWS /OPT:NOREF ім’я_файлу.obj link.exe

/SUBSYSTEM:CONSOLE ім’я_файлу.obj

А.3 Середовища розробки програм Masm32

Асемблер Masm32 може бути реалізований в різних програмних середовищах під управлінням операційної системи Windows. При цьому технологія отримання виконуваного коду дещо відрізняється, не змінюючи суті виконання отриманої програми. Програмісти самі обирають інструмент реалізації, виходячи зі своїх смаків та можливостей.

А.3.1 Середовище Visual Studio

Для створення та налагоджування програми в Visual Studio 201X необхідно виконати наступні дії.

1.Створити проект:

File/ New/ Project.

У лівій колонці вікна, що відкрилося, виділяємо Win32, а потім Win32 Console Application. Після чого вказуємо ім'я проекту і директорію його розташування (рис. А.1)

Рис. А.1.1 Вікно **Create Project**

У вікні, що відкрилося **Vizard Win32** підсвічуємо мишкою пункт **Application Settings** і відзначаємо пункт **Empty project**. І натискаємо кнопку **Finish** (рис. А.2).

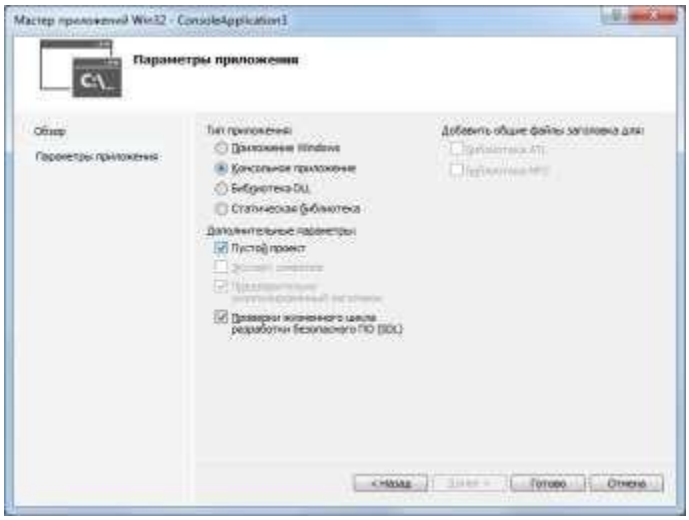

Рис. А.2 Вікно **Vizard …**

1.У головному меню вибираємо **PROJECT/Build** Customizations і у вікні, що відкрилося, вибираємо значення masm() (рис. А.3).

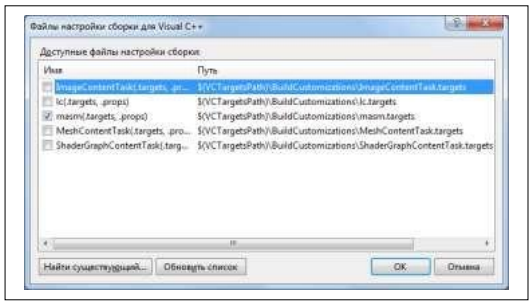

Рис. А.3. Вікно **Configuration Files…**

2.Налаштовуємо значення платформи. Для цього підводимо вказівник мишки до значка. Внаслідок чого з'являється ім'я **Toolbar Options Standard**. Ставимо відмітку в пункті **Solution platforms** (Рис. А.4).

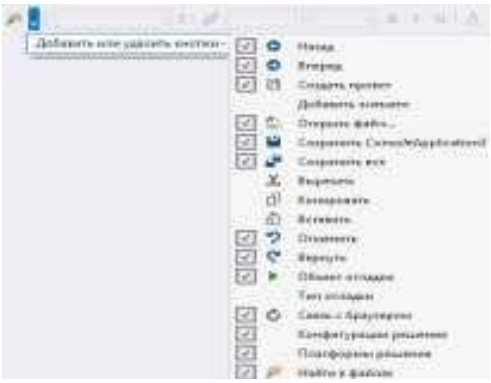

Рис. А.4 Вікно обрання **Solution platforms**

З'явилося вікно Win32. У цьому вікні в правій її частині вибираємо значення **Configuration Manager** (рис.А.5).

Рис. А.5. Вікно **Configuration Manager**

У вікні диспетчера конфігурації замість активної платформи рішення вибираємо значення Create.

Далі вибираємо значення x64 (рис. А.6).

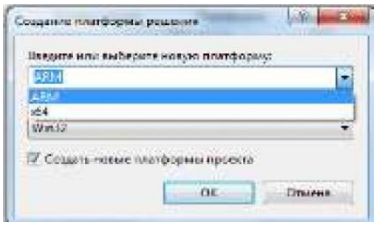

Рис. А.6. Вікно **Create solution platforms**

В меню головного вікна вибрати **PROJECT/ Properties/ Linker/ Advanced** (рис. А.7).

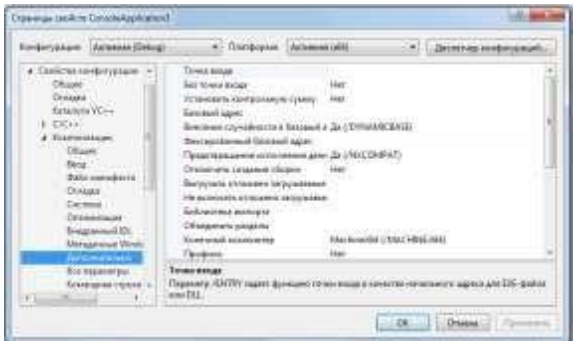

Рис. А.7. Вікно **Properties page…**

У правій колонці вікна вибрати Point of entry, а в кінці цього рядка, натиснувши на зображення трикутника вибрати значення Change.

У вікні, що з'явилося, ввести значення точки входу в програму, наприклад, main. Натиснути **Apply** та **ОК**. Обираємо **View/ Solution Explorer**. Підсвічуємо мишкою Resource files, натискаємо праву клавішу мишки і обираємо **Add/Create** element (Рис. А.8).

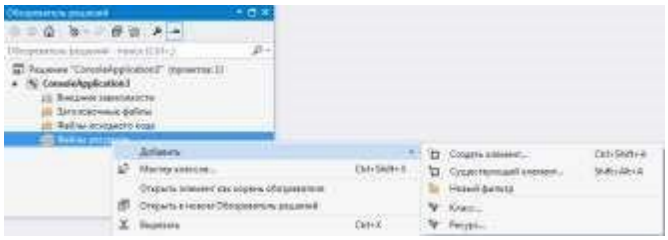

Рис. А.8. Вікно обрання **Solution Explorer**

У новому вікні вибираємо ім'я програми, наприклад myAsm.asm та визначаємося з директорією її розташування. Натискаємо **Add**.

У основному вікні пишемо програму. Для відображення нумерації рядків необхідно вибрати: SERVICE / Options / Text editor / All languages / General та включити Line numbers.

Для виконання програми вибираємо пункти **ASSEMBLY / Assemble solution** (або Ctrl+Shift+B).

Для налагодження програми вибираємо пункти **DEBUG/Step bypass** (F10). Стани регістрів відображаються, якщо в середині вікна **Registers** натиснути праву клавішу мишки та вибрати ЦП (центральний процесор) (рис. А.9).

Рис. А.9. Вікна налагоджувача середовища **Visual Studio 201X**

А.4.2 Робота з середовищем RADAsm

Серед альтернативних програмних середовищ для розробки програм мовою асемблера розглянемо середовище RADAsm (Рис. А.10)

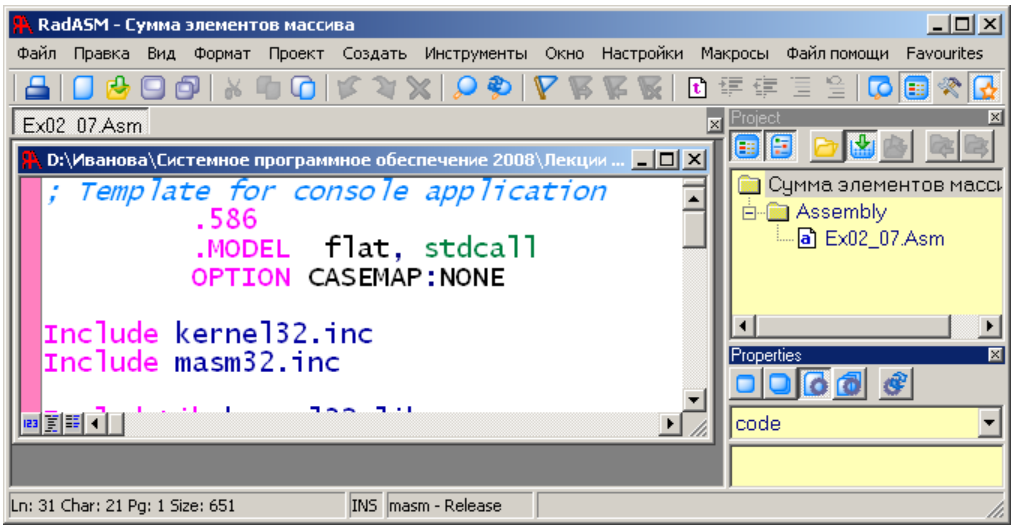

Рис. А.10 - Вікно інтегрованого середовища **RADAsm**

Програмне середовище ініціюється запуском програми RADAsm.exe(https://wikiprograms.org/radasm/). Після виклику на екрані з'являється вікно середовища RADAsm, в якому зазвичай висвічується остання програма, налагоджувати попередній раз.

Для створення нового проекту необхідно вибрати пункт меню **File/ New**, після чого на екрані з'явиться перше вікно чотирьохвіконної Майстра створення проекту).

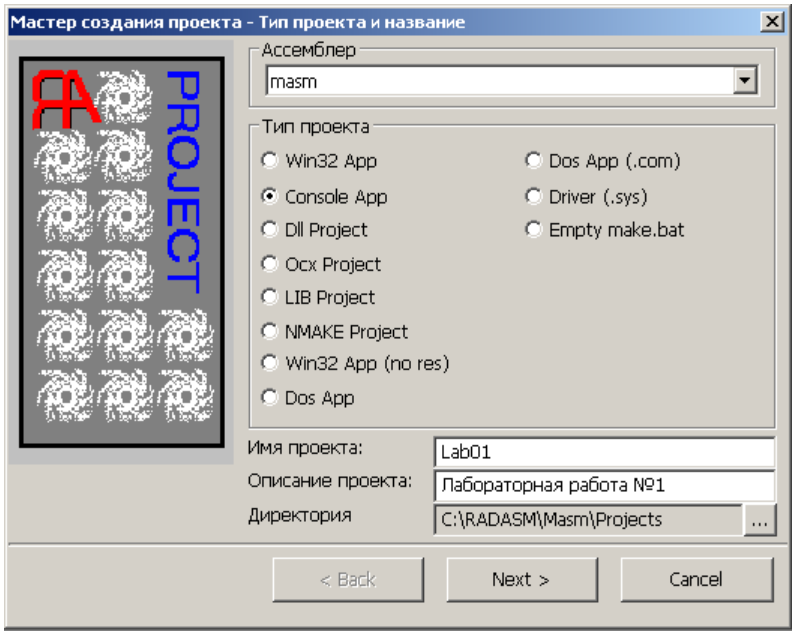

Рис. А.11 Вікно майстра створення нового проекту

У цьому вікні необхідно вибрати тип проекту - в нашому випадку **Console App** (консольний застосунок), а також ввести його ім'я, наприклад, Lab01, опис, наприклад,

«Лабораторна робота № 1», і шлях до створюваної середовищем нової папки з ім'ям проекту.

У наступному вікні Майстра вибирається шаблон проекту - conapp.tpl - спеціально створений для лабораторних робіт шаблон консольного застосування.

Передостаннє вікно Майстра пропонує вибрати типи створюваних файлів - вибираємо **Asm** (вихідні файли асемблера), і папки - вибираємо папку Bak, використовувану для розміщення попередніх копій файлів.

Останнє вікно **Майстра** визначає доступні для роботи з проектом пункти меню запуску додатка Create і виконувані команди. У даній асемблері виконувані команди вже налаштовані, тому в ньому можна нічого не міняти, хоча використовуватися будуть не всі пункти створеного меню, а тільки такі: **Assemble** (транслювати), **Link** (компонувати), **Run** (виконати) і **Run w/Debug** (виконати в підключеному відладчику).В результаті буде отримана заготовка консольного додатку Windows. Переглянути цю заготовку можна, двічі клацнувши лівою клавішею миші по файлу Lab01.asm у вікні навігатора Project, розташованому праворуч вгорі.

Заготовка містить:

• директиви, що визначають набір команд, модель пам'яті, конвенцію передачі даних і опцію відмінності малих і великих літер;

• директиви підключення бібліотек;
• розділи констант, ініціалізованих і неініціалізованих даних з мінімально необхідними директивами визначення даних;
• розділ коду, процедури в якому забезпечують затримку закриття вікна до натискання будь-якої клавіші.;

```
Template for console application - комментарий
.586 ; підключення набора команд Pentium
.MODEL flat, stdcall ; модель пам'яти и
                                ; конвенція про передачу параметрів
OPTION CASEMAP:NONE ; опція рядкових
                                ; і заголовних букв
Include kernel32.inc ; підключення опису процедур і
Include masm32.inc ; констант
IncludeLib kernel32.lib ; підключеня бібліотек
IncludeLib masm32.lib
.CONST ; початок розділу констант
MsgExit DB "Press Enter to Exit",0AH,0DH,0
.DATA ;розділ ініціалізованих зміних
.DATA? ;розділ неініціалізованих зміних
inbuf DB 100 DUP (?)
.CODE ; початок сегмента кода
Start:
;                                Місце, куди додається код
; Add you statements
;
Invoke StdOut,ADDR MsgExit ; вивід повідомлення
Invoke StdIn,ADDR inbuf,LengthOf inbuf ; введення рядка
Invoke ExitProcess,0 ; завершення програми
End Start ; кінець модуля
```

Заготовку, як в інших середовищах програмування, можна запустити на виконання. Вона містить виклики процедур, що забезпечують виведення на екран запиту «Press Enter to Exit» (Натисніть Enter для виходу) і введення рядка. Це зроблено для того, щоб затримати автоматичне закриття вікна консолі при завершенні програми до натискання клавіші Enter.

Запуск заготовки застосунка

Для запуску заготовки необхідно виконати:

- трансляцію Create / Assemble,
- компонування Create / Link,
- запуск на виконання Create / Run.

У процесі трансляції (асемблінгу) вихідна програма мовою асемблераперетворюється в двійковий еквівалент. Якщо трансляція проходить нормально, то у вікні Output, яке з'являється під вікном програми, виводиться текст:

C:\Masm32\Bin\ML.EXE /c /coff /Cp /nologo
/I"C:\Masm32\Include" "Lab01.asm"
Assembling: Lab01.asm
Make finished.
Total compile time 78 ms

Примітка - Вікно Output з'являється і закривається. Щоб повторно подивитися результати, необхідно встановити курсор миші на верхню частину рядка стану під вікном тексту програми (нижню рамку вікна Output).

Перший рядок повідомлення про асемблінгу - виклик асемблера:

C: \ Masm32 \ Bin \ ML.EXE - повне ім'я файлу транслятора асемблера masm32 (шлях + ім'я), за яким слідують опції:

/ C – замовляє асемблінг без автоматичного компонування,

/ Coff - визначає формат об'єктного модуля Microsoft (coff),

/ Cp - означає збереження регістра заголовних і прописних букв усіх ідентифікаторів програми,

/ Nologo - здійснює придушення виведення повідомлень на екран в разі успішного завершенняасемблінгу,

/ I "C: \ Masm32 \ Include" - визначає місцезнаходження вставляються (.inc) файлів,

і параметр "Lab01.asm" - задає ім'я оброблюваного файлу.

Решта рядків - повідомлення про початок і завершення процесуасемблінгу і часу виконання цього процесу.

Результатом нормального завершенняасемблінгу є створення файлу, що містить об'єктний модуль програми, - файлу Lab01.obj.

Якщо при асемблінгу виявлені помилки, то об'єктний модуль не створюється і після повідомлення про початок асемблінгу йдуть повідомлення про помилки, наприклад:

Lab01.asm (26): error A2006: undefined symbol: EAY

У повідомленні вказується:

• номер рядка вихідного тексту (в дужках),

• номер помилки, під яким вона описана в документації,

• можлива причина.

Після виправлення помилок, процес асемблінгу повторюють.

Наступний етап - компоновка програми. На цьому етапі до об'єктного (бінарного) коду програми додаються об'єктні коди використовуваних підпрограм. При цьому в тих місцях програми, де відбувається виклик процедур, вказується їх відносна адреса в модулі. Відомості про компонуванні також виводяться у вікно Output:

C:\Masm32\Bin\LINK.EXE /SUBSYSTEM:CONSOLE /RELEASE

/VERSION:4.0 /LIBPATH:"C:\Masm32\Lib" /OUT:"Lab01.exe"

"Lab01.obj"

Microsoft (R) Incremental Linker Version 5.12.8078

Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Make finished.

Total compile time 109 ms

Перший рядок виводу також є командним рядком виклику компоновщика

C: \ Masm32 \ Bin \ LINK.EXE - повне ім'я компоновщика, за яким слідують опції:

/ SUBSYSTEM: CONSOLE - підключити стандартне вікно консолі,

/ RELEASE - створити реалізацію (а не оцінний варіант),

/VERSION:4.0 - мінімальна версія компоновщика,

/ LIBPATH: "C: \ Masm32 \ Lib" - шлях до файлів бібліотек,

/OUT:"Lab01.exe "- ім'я результату компонування - завантажувального файлу

і параметр "*Lab01.obj*" - ім'я об'єктного файлу.

У наступних рядках також можливі повідомлення про помилки. В основному будете отримувати повідомлення про недозволені зовнішні посилання, наприклад:Lab01.obj:*errorLNK2001:unresolved external symbol _ExitProcess@4*

Lab01.exe : fatal error LNK1120: 1 unresolved externals Make error(s) occured.

Як правило, таке повідомлення говорить про наявність в програмі викликів процедур, для яких в вказівки бібліотеках не знайдені коди. В даному прикладі не підключена бібліотека, в якій знаходиться процедура ExitProcess. Після усунення помилки програму необхідно перетранслювати і заново скомпонувати.

Якщо процеси трансляції і компонування пройшли нормально, то її можна запустити на виконання. При цьому відкривається вікно консолі, в яке виводиться рядок запиту (див. Рис. А.12).

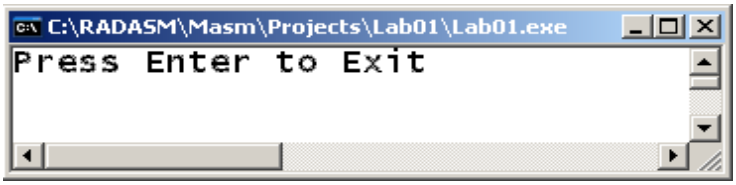

Рисунок А.12 – Вікно консолі

Створення найпростішої програми

Для вивчення можливостей відладчика необхідно ввести програму, яка ви заповнює кілька простих команд, наприклад, обчислює результат наступного виразу

X = A + 5-B

Дані для програми задамо константами, помістивши їх опис в розділ ініціювання Даних:

DATA

ASDWORD ***-30***

B SDWORD ***21***

Для результату обчислень - змінної X - необхідно зарезервувати місце, помістивши опис відповідної неініціалізованной змінної в розділ неініціалізованих Даних:

.DATA?

X SDWORD ?

Розміщення неініціалізованої змінної в цьому розділі не є обов'язковим, але більш грамотно, оскільки неініціалізовані змінні в образі виконуваної програми на диску не зберігаються, а розміщуються при завантаженні на виконання. Фрагмент коду програми, яка виконує додавання і віднімання, необхідно помістити в сегменті кодів після мітки Start:

Start: mov EAX,A ; вмістити число в регістр EAX

add EAX, 5; скласти EAX і 5, результат в EAX

sub EAX, B; відняти B, результат в EAX

mov X, EAX; зберегти результат в пам'яті

Оскільки програма нічого не виводить, результат операції слід подивитися в відладчику**.**

Для запуску програми в режимі налагодження слід послідовно ініціювати такі пункти меню:

• трансляцію **Create / Assemble**

• компонування **Create / Link**,

• запуск на виконання в відладчик **Create/Run w/Debug**.

Після запуску на екрані з'являється вікно відладчика **OlleDBG** (див. Рис. А.13).

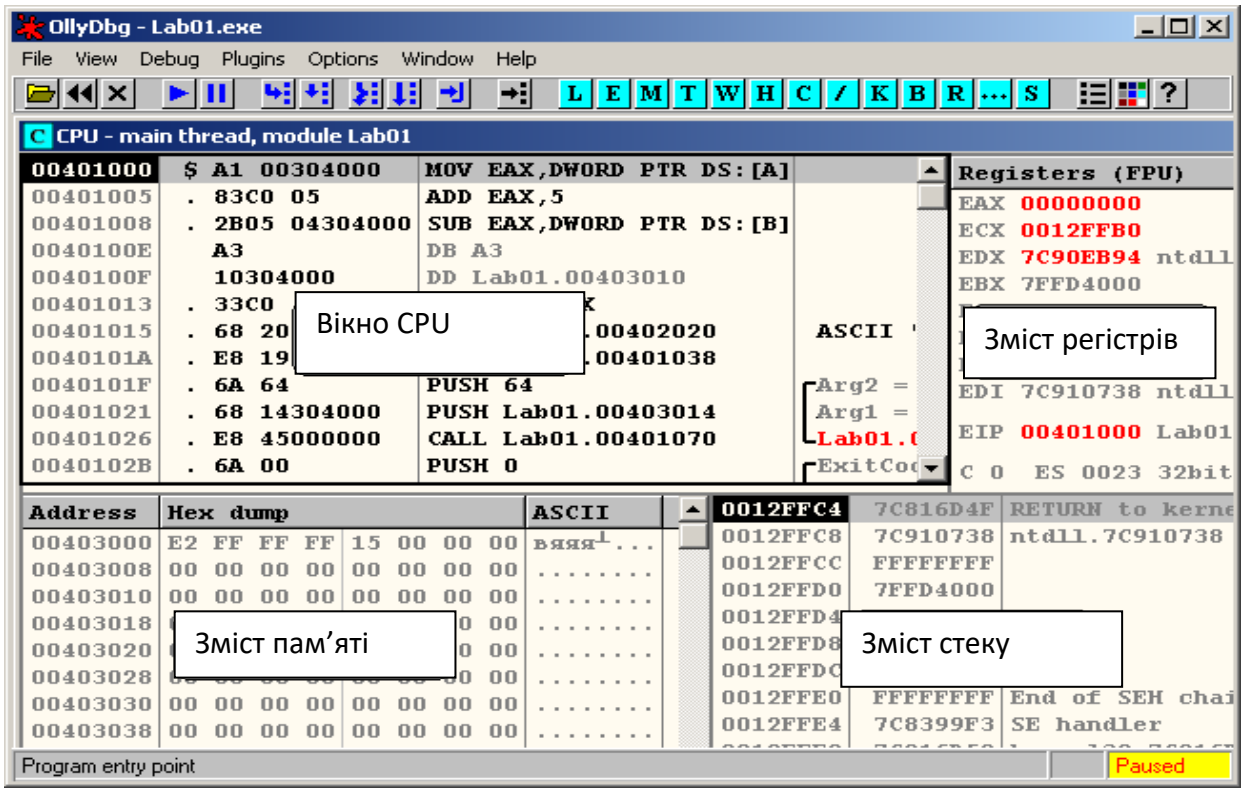

Рис. А.13 – Вікно OlleDBG.

У вікні можна виділити чотири області:

- дочірнє вікно CPU - вікно процесора, в якому висвічуються адреси команд, їх шістнадцяткові коди, результати дізасемблінгу і результати виділення параметрів процедур;
- область Registers - вікно значенню регістрів процесора,
- вікно Даних, в якому висвічуються адреси пам'яті (Address) і її шістнадцятковий (Hex dump) і символьний (ASCII) дампи;
- вікно стека, в якому показується значення вершини стека.

У вихідний момент часу курсор знаходиться в вікні CPU, тобто програма готова до виконання. Для виконання команди відладчика використовують такі комбінації клавіш:

F7 - виконати крок з заходом в тіло процедури; F8 - виконати крок, не заходячи в тіло процедури.

Адреса початку програми 00401000h.

На жаль, відладчик не завжди правильно дизасемблює двійковий код програми, і якісь команди можуть залишитися не розпізнані. Так на рисунку 4 видно, що команда mov X, EAX виявилася не розпізнаною. Замість неї в відладчику вказано код, що

відповідає визначенню даних: DB A3 і т.д. Однак виконання програми буде здійснюватися правильно.

Адреса початку розділу ініціювання даних - 00403000h. Кожне значення займає стільки байт, скільки резервується директивою. У відладчику кожен байт представлений двома шістнадцятеричними цифрами. Крім того, використовується зворотний порядок байт, тобто молодший байт числа знаходиться в молодших адресах пам'яті (перед старшим). Якщо шістнадцятерична комбінація відповідає коду символу, то він висвічується в наступній колонці(ASCII), інакше в ньому висвічується точка, не ініційовані дані розташовуються після ініціювання з адреси, кратної 16.

При кожному виконанні команди ми можемо спостерігати зміну даних до пам'яті та/або в регістрах, відстежуючи процес виконання програми і контролюючи правильность проміжного результатів.

Так після виконання першої команди число A копіюється в регістр EAX, який при цьому підсвічується червоним. При запису в регістр порядок байт змінюється на прямий, при якому першим записаний старший байт.

ДОДАТОК Б. РОЗШИРЕНИЙ АСЕМБЛЕР NASM

Розширений асемблер NASM - це 80x86 асемблер, розроблений на основі принципів переносимості і модульності. Він підтримує широкий діапазон форматів об'єктних файлів, включаючи формати Linux a.out і ELF, NetBSD / FreeBSD, COFF, Microsoft 16-bit OBJ і Win32. Він здатний також створювати прості бінарні файли. Синтакс NASM максимально спрощений для розуміння і схожий на синтакс Intel, але дещо складніший. Він підтримує інструкції Pentium, P6 і MMX, а також має макророзширення.

Б.1 НЕОБХІДНІСТЬ NASM

Виникає запитання: навіщо ще один асемблер? Подамо короткий аналіз.

- a86 - хороший асемблер, але не безкоштовний, і якщо ви не заплатите, то 32-бітний код писати не зможете - тільки DOS.
- gas знаходиться у вільному доступі і портований під DOS і Unix, але розроблений для забезпечення сумісності з gcc. Тому перевірка помилок мінімальна, до того ж, з точки зору будь-кого, хто спробував що-небудь *написати* в ньому - синтаксис жахливий. Плюс до всього ви не можете в ньому написати 16-розрядний код.
- as86 - тільки під Linux і недостатньо добре документований.
- MASM дуже хороший і працює в середовищі Visual Studio, але мало придатний для розуміння початківцями суті асемблера
- TASM суперник MASM за сумісність, що означає суттєве ускладнення. Колись він найкращий, але тільки для DOS, але наразі це не має жодного значення.

Отже, представляємо NASM.

Сторінка NASM в інтернеті WWW http://www.cryogen.com/Nasm.

Як вже зазпочатоксь, NASM може використовуватись як на платформі WINDOWS так і на всіх ОС Unix, розроблених під Intel-архітектуру, наприклад Linux, що рекомендовано для використання для лабораторних та практичних робіт.

Б.2 ВИКОРИСТАННЯ NASM В ОС LINUX

В NASM Linux, використовується Intel синтаксис. Компіляція і лінковка:

- nasm -f elf -o hello.o hello.asm
- ld -o hello hello.o

Б.3 Інсталяція NASM під Unix

При отриманні Unix-архіву початкових кодів NASM, nasm-X.XX.tar.gz (де X.XX означає номер версії NASM в архіві) розпакуйте його в каталог типу / usr / local / src. Архів за розпакуванню створить власний підкаталог nasm-X.XX.

NASM - автоконфігуруємий пакет: як тільки ви розпакуєте його, перейдіть до каталогу, куди він був розпакований і введіть./configure. Даний шелл-скрипт знайде найкращий компілятор C для збірки NASM і, відповідно, настройки Make-файлів.

Як тільки NASM сконфігурує, ви можете ввести make для збірки бінарних файлів nasm і ndisasm, а потім ввести make install для установки їх в / usr / local / bin і установки man-сторінок nasm.1 і ndisasm.1 в / usr / local / man / man1. Як альтернативу ви можете вказати опції типу --prefix до команди configure скрипта (подробиці див. У файлі INSTALL) або встановити програми самостійно. NASM також має набір утиліт для обробки замовленого формату об'єктних файлів RDOFF, що знаходяться в підкаталозі rdoff архіву NASM. Ви можете зібрати їх за допомогою make rdf і встановити за допомогою make rdf_install, якщо звичайно вони вам потрібні.

Якщо NASM буде не в змозі автоматично вибрати конфігурацію, ви все-таки зможете скомпілювати його за допомогою make-файлу Makefile.unx. Скопіюйте або перейменуйте цей файл в Makefile і спробуйте ввести make. Є також файл Makefile.unx в підкаталозі rdoff.

Якщо при установці Linux ви встановлювали «Інструменти розробки», то NASM у вас вже встановлений. Щоб перевірити, чи встановлений у вас NASM, виконайте наступні дії:

- відкрийте термінал;
- введіть команду whereis nasmi натисніть ENTER.

Якщо він у вас встановлений, то ви побачите приблизно наступне (Рис Б.1).:

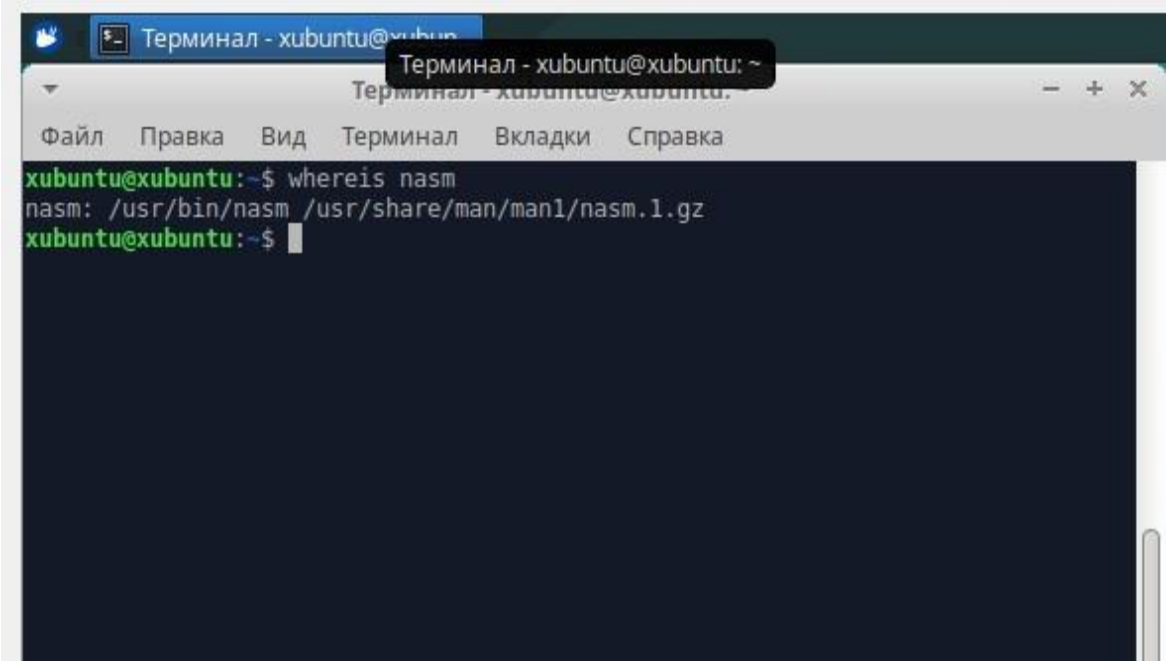

Рис Б.1 Знаходження NASM

Якщо ж ми побачили просто: nasm:

То NASM у вас не встановлений, і його потрібно встановити.

Щоб встановити NASM, виконайте наступні кроки:

- відкрийте термінал;
- введіть sudo apt install nasm.

Рекомендовано встановити універсальний редактор коду Visual Studio Code. Він дозволить вам не тільки зручно писати мовою асемблерапід Linux, але і на інших мовах. Для установки вам потрібно перейти на офіційний сайт Microsoft Visual Studio Code і завантажити версію.deb (Рис. Б.2:)

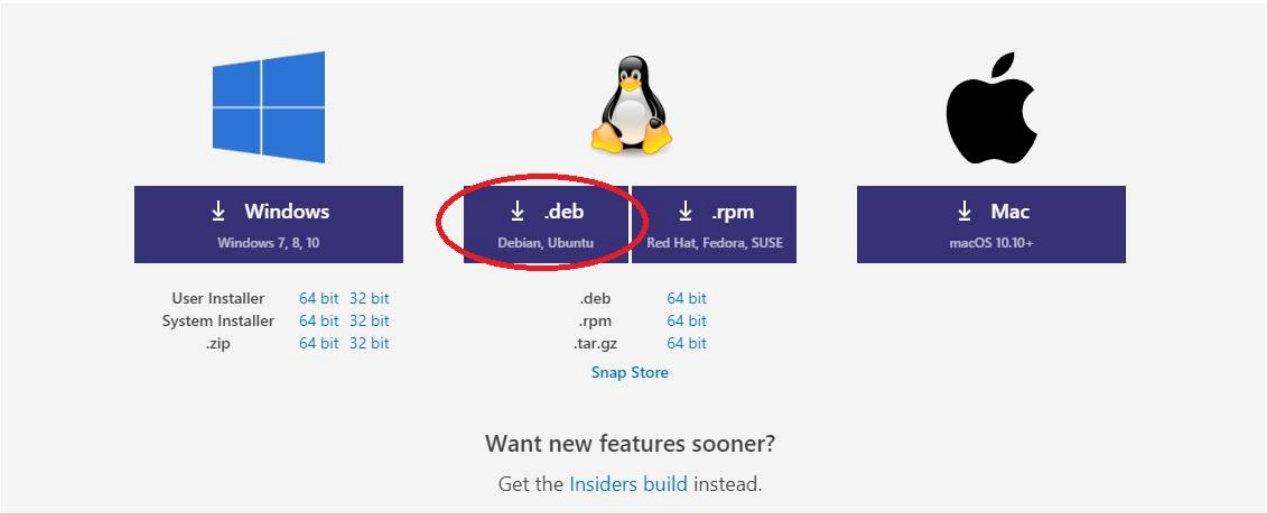

Рис. Б.2 Встановлення версії .deb

Після того, як VS Code буде встановлено, вам потрібно буде:

- запустити його;
- натиснути комбінацію клавіш Ctrl+P;
- ввести у вікні ext install 13xforever.language-x86-64-assembly.

Б.4 СИНТАКСИС КОМАНДНОГО РЯДКА NASM

Для асемблінгу файлу треба ввести наступну команду:

nasm -f <format> <filename> [-o <output>]

Наприклад,

nasm -f elf myfile.asm

буде збирати myfile.asm в ELF-об'єктний файл myfile.o. А рядок

nasm -f bin myfile.asm -o myfile.com

буде збирати myfile.asm у звичайний бінарний файл myfile.com. Для отримання файлу-лістингу, що містить ліворуч від оригінального вихідного тексту шістнадцяткові коди, що генеруються NASM, використовуйте ключ -l, що позначає ім'я файлу-лістингу, наприклад:

nasm -f coff myfile.asm -l myfile.lst

Для отримання довідки по командному рядку NASM вкажіть наступний ключ:

nasm –h.

При цьому ви також отримаєте список доступних форматів вихідних файлів і що вони означають. Якщо ви використовуєте Linux, але не впевнені, яка ваша система - a.out або ELF, введіть:

file nasm

у каталозі, де знаходяться бінарні файли NASM. У відповідь ви отримаєте щось на зразок

nasm: ELF 32-bit LSB executable i386 (386 and up) Version 1

Це означає, що ваша система - ELF і ви повинні при асемблінгу використовувати ключ -f elf. Якщо ж ви побачите:

nasm: Linux/i386 demand-paged executable (QMAGIC)

або щось на зразок цього, ваша система - a.out, і при асемблінгу потрібно буде вказати ключ -f aout . (В Linux системах a.out вважаються застарілими і сьогодні зустрічаються рідко). Подібно до Unix компіляторів і асемблерів, NASM "безшумний": ви не будете бачити жодних повідомлень взагалі, якщо тільки це не повідомлення про помилки.

Ключ -o : Вказівка імені вихідного файлу

Зазвичай NASM вибирає ім'я вихідного файлу самостійно; оскільки це залежить від формату файлу об'єкта. Якщо формат об'єктного файлу - Microsoft (obj і win32), він видаляє розширення.asm (або будь-яке інше, яке вам подобається використовувати - NASM все одно) з імені вихідного файлу і замінить його на .obj. У об'єктних файлів Unix-формату (aout, coff, elf і as86) він замінюватиме розширення на.o. Для формату rdf він буде використовувати розширення . _myfile.

Якщо вихідний файл вже існує, NASM перезапише його, якщо його ім'я не збігається з ім'ям вхідного файлу — у цьому випадку з'явиться попередження і як вихідний файл буде використано ім'я nasm.out.

У випадках, коли стандартне ім'я неприпустиме, використовуйте ключ -o командного рядка, що дозволяє визначити необхідне вам ім'я вихідного файлу. Ім'я вихідного файлу має йти за ключем -o, неважливо з пробілом між ними чи без. Наприклад:

nasm -f bin program.asm -o program.com

nasm -f bin driver.asm -odriver.sys

Ключ -f : Вказує формат вихідного файлу

Якщо ключ -f у командному рядку відсутній, NASM вибиратиме формат вихідного файлу самостійно. У поширеній версії NASM формат за замовчуванням завжди bin ; якщо ви створюєте власну копію NASM, при компіляції можете перевизначити значення OF_DEFAULT на те, що вам потрібно за замовчуванням.

Як і для ключа -o розділовий пробіл між -f і форматом вихідного файлу необов'язковий: -f elf і -felf для NASM ідентичні.

Повний список доступних вихідних форматів можна отримати за допомогою команди nasm -h.

Ключ -l : Генерація файлу-лістингу

Якщо в командному рядку ви вкажете ключ -l і ім'я файлу (як зазвичай, пробіл необов'язковий), NASM буде генерувати з вихідника файл-листинг, де адреси і код, що генерується, будуть розташовані зліва, а вихідний код з розгорнутими багаторядковими макросами - Праворуч. Наприклад:

nasm -f elf myfile.asm -l myfile.lst

Ключ -E : Перенаправлення помилок у файл

Позаяк NASM зазвичай направляє попередження та повідомлення про помилки в потік stderr, перехоплення цих повідомлень (наприклад, для подальшого перегляду в редакторі) може викликати складності.

У зв'язку з цим є спеціальний ключ -E після якого вказується ім'я файлу (пробіл необов'язковий). Цей ключ перенаправляє стандартний потік помилок у вказаний файл. Таким чином, ви можете запустити NASM, наприклад:

nasm -E myfile.err -f obj myfile.asm

Ключ -s : Перенаправлення помилок у stdout

Ключ -s перенаправляє стандартний потік помилок stderr у вихідний потік stdout (природно, MS-DOS). Наприклад, для асемблювання файлу myfile.asm та передачі помилок програмі more, ви можете ввести наступне:

nasm-s-f obj myfile.asm | more

також ключ -E,

Ключ -i : Каталоги пошуку файлів, що включаються.

Коли NASM зустрічає у вихіднику директиву %include, він шукатиме вказаний у ній файл у поточному каталозі, а й у всіх каталогах, вказаних у командному рядку з допомогою ключа -i. Отже, ви можете увімкнути файли з, наприклад, бібліотеки макросів, ввівши таку команду:

nasm -ic:\macrolib\ -f obj myfile.asm

З метою переносимості коду в NASM не "зашито" угоду про іменування файлів тієї чи іншої ОС, під якою він запущений; рядок, який ви вкажете як аргумент ключа, буде оброблено точно так, як є. Завершальний зворотний слеш у наведеному вище прикладі під Unix необхідний, т.щ. у цій ОС заключні слеші зазвичай потрібні.

(Ви можете отримати з цієї обставини вигоду, використовуючи цей ключ "не за призначенням" - наприклад ключ -ifoo буде змушувати директиву %include "bar.i" шукати файл foobar.i...)

Якщо ви хочете описати *стандартний* шлях пошуку файлів, що включаються, такий як /usr/include в системі Unix, ви повинні помістити одну або більше директив -i в змінну оточення NASM.

Для сумісності з make-файлами більшості компіляторів, даний ключ може бути також заданий як -I.

Ключ -p : Попередні файли

За допомогою ключа -p NASM дозволяє *попередньо* включити деякі файли у ваш

вихідний. Так, рядок запуску:

nasm myfile.asm -p myinc.inc

еквівалентний рядку nasm myfile.asm та розміщенням на початок файлу директиви %include "myinc.inc". Для цілей симетричності з ключами I, -D і -U даний ключ може бути також заданий як -P.

Ключ -d : Призначення макросу

Аналогічно тому, як ключ -p дає альтернативу приміщення початку вихідного файлу директиви %include, ключ -d дає альтернативу директиві %define. Таким чином, команда

nasm myfile.asm -dFOO=100

альтернативна приміщенню на початок файлу директиви

%define FOO 100

Ви також можете опустити значення константи: ключ -dFOO еквівалентний рядку %define FOO. Ця можливість може бути корисною при асемблінгу для включення/вимкнення опцій, що перевіряються за допомогою директиви %ifdef, наприклад -dDEBUG. Для сумісності з make-файлами більшості компіляторів, даний ключ може бути також заданий як -D.

Ключ -u : Скасування визначення макросу

Ключ -u скасовує визначення раніше визначеного макросу. Наприклад, у результаті виконання наступного командного рядка:

nasm myfile.asm -dFOO=100 -uFOO

FOO *не буде* зумовленим макросом для програми. Це корисно для тимчасового вимкнення опцій, заданих у make-файлах. Для сумісності з make-файлами більшості компіляторів, даний ключ може бути також заданий як -U.

Ключ -e : Тільки препроцесування

NASM допускає виконання лише препроцесування вхідного файлу. Використання ключа -e (не потребує параметрів) змусить NASM перепроцесувати вхідний файл, розгорнути всі макро посилання, видалити всі коментарі та директиви препроцесора і вивести результуючий файл у стандартний вихідний потік (або зберегти його як окремий файл, якщо також використовується опція -o).

Цей ключ не застосовується для програм, де препроцесор повинен обчислити вирази, що залежать від значень адрес: такий код як

%assign tablesize ($-tablestart)

буде викликати помилку в режимі "тільки препроцесування".

Ключ -a: Вимкнення препроцесора

Якщо NASM використовується як вихідна частина компілятора, то для збільшення швидкості компіляції бажано повністю придушити препроцесування у випадку, якщо компілятор вже робить його. Ключ -a, що не вимагає жодних параметрів, змушує NASM замінити свій препроцесор нічого не робить "заглушкою".

Ключ -w: Дозвіл/Заборона попереджень під час асемблінгу

В процесі асемблювання NASM може помічати безліч речей, що заслуговують на увагу користувача, але не являють собою таких серйозних помилок, щоб NASM не зміг згенерувати вихідний файл. Ці попередження видаються так само, як і помилки, але перед повідомленням додається слово "warning". Попередження не є причиною переривання процесу створення вихідного файлу.

Деякі події менш серйозні, ніж інші: вони лише іноді заслуговують на увагу користувача. У зв'язку з вищесказаним NASM підтримує ключ командного рядка -w, що включає або вимикає певні класи попереджень. Ці класи попереджень описуються на ім'я, наприклад orphan-labels ; Ви можете дозволити попередження цього класу за допомогою ключа -w+orphan-labels та заборонити їх ключем -w-orphan-labels.

Нижче перераховані класи попереджувань, що придушуються:

macro-params включає попередження про багаторядкові макроси, яким передається неправильне число параметрів. Цей клас попереджень за замовчуванням дозволено; orphan-labels включає попередження про рядки вихідного коду, що не містять інструкцій, але в яких описуються мітки без завершального двокрапки. NASM за замовчуванням не попереджає про ці речі.

number-overflow включає попередження про числові константи, що не вписуються в діапазон 32 біта (0x7fffffff). Цей клас попереджень за замовчуванням дозволено.

Змінна оточення NASM

Якщо ви визначите змінну оточення NASM, програма інтерпретуватиме її як список додаткових ключів командного рядка, що обробляються *раніше* "справжніх" параметрів командного рядка. Ви можете використовувати цю можливість, наприклад для опису стандартних каталогів пошуку файлів, що включаються, помістивши в преміальну NASM ключ -i.

Значення змінної поділяється пробілами, тому значення -s -ic:\nasmlib буде оброблено як два окремі ключі. В той же час значення -dNAME="my name" не буде

сприйнято так, як ви хочете (всередині є пробіл) і командний процесор NASMa відповідно не зрозуміє двох безглуздих параметрів -dNAME="my та name".

Для вирішення цього введено таке правило: якщо ви починаєте змінну NASM деяким символом, що не є знаком "мінус", NASM сприйматиме цей символ як роздільник опцій. Таким чином, значення !-s!-ic:\nasmlib змінної NASM еквівалентно -s -ic:\nasmlib, зате тепер !-dNAME="my name" буде працювати правильно.

Б.5 Користувачам MASM: Відмінності

Якщо ви використовували для написання програм MASM, або TASM у режимі сумісності з MASM, або a86, прочитайте цей розділ, в якому наведено основні відмінності синтаксису MASM та NASM. Якщо ви взагалі не використовували раніше MASM, просто пропустіть цей розділ і читайте далі.

NASM чутливий до регістру символів

Найпростішою відмінністю є регістро-чутливість NASM - він розрізняє звернення до таких міток, як foo, Foo або FOO. Якщо ви збираєте .obj -файли для DOS або OS/2, то для переведення всіх символів, що експортуються в інші модулі, у верхній регістр можете активізувати директиву UPPERCASE. Однак *у межах* одного модуля NASM розрізняє мітки, що відрізняються один від одного тільки регістром.

NASM вимагає квадратних дужок для посилань на пам'ять

NASM був розроблений, крім іншого, для спрощення запам'ятовування синтаксису. Одна з цілей проекту NASM полягає в тому, щоб скрізь, де це можливо, була взаємно однозначна відповідність між окремим рядком коду NASM і інструкцією, що генерується з неї. У MASM ви цього зробити не можете: якщо визначите, наприклад,

```
foo equ 1
bar dw 2
```

потім напишете два рядки коду:

```
mov ax,foo
mov ax,bar
```

то будуть згенеровані дві абсолютно різні інструкції, незважаючи на те, що синтаксис на вигляд цілком ідентичний.

NASM уникає цієї небажаної ситуації шляхом спрощення синтаксису для посилань на пам'ять. Правило елементарне — *будь* -який доступ до вмісту пам'яті вимагає встановлення квадратних дужок навколо адреси, а за будь-якого доступу до *адреси* змінної квадратні дужки не ставляться. Таким чином, інструкція виду mov ax, foo

завжди буде посилатися на константу часу компіляції, неважливо чи це EQU або адресу змінної, в той же час для отримання доступу до *вмісту* змінної bar ви повинні використовувати код mov ax, [bar]

З цього також випливає, що NASM не потребує ключового слова MASMa OFFSET, т.щ. MASMівський код mov ax, offset bar означає те саме, що й mov ax, bar для NASM. Якщо ви накопичили досить багато коду, написаного під MASM і хочете використовувати його в NASM, ви завжди можете вставити рядок виду %idefine offset, що вказує препроцесору, що ключове слово OFFSET є інструкцією, що нічого не робить.

Ця невідповідність ще більшою мірою присутня в a86, де оголошення мітки з завершальним двокрапкою визначає власне мітку, а без двокрапки - змінну. Так, в a86 інструкція mov ax,var поводитиметься по-різному, залежно від оголошення мітки var : як var: dw 0 (це мітка) або як var dw 0 (а це вже змінна розміром у слово). NASM у порівнянні з цим дуже простий: *все* є мітками.

NASM, з метою спрощення, не підтримує гібридний синтаксис MASMa та її клонів, такий, як наприклад mov ax,table[bx], де посилання пам'ять позначена частиною всередині квадратних дужок, а частиною — поза ними. Правильний синтаксис NASM для вказаної вище інструкції буде mov ax,[table+bx]. Відповідно, інструкція виду mov ax, es: [di] не підтримується, правильна інструкція - mov ax, [es: di].

NASM не зберігає типи змінних

NASM не запам'ятовує типи змінних, що визначаються вами. Оскільки MASM ці речі запам'ятовує, то при зустрічі var dw 0 він запам'ятає, що ви визначили var як замірний розмір у слово і потім буде здатний дозволити невизначеність при появі інструкції mov var,2. NASM ж навмисно не пам'ятатиме ніщо щодо символу var крім того, де він починається, тому ви повинні явно вказувати mov word [var],2.

Відповідно, NASM не підтримує інструкції LODS, MOVS, STOS, SCAS, CMPS, INS або OUTS, підтримуються лише їх форми виду LODSB, MOVSW і SCASD, де явно задається розмір компонентів рядка, що обробляється.

NASM не підтримує ASSUME

Як частина загальної ідеології спрощення NASM не підтримує директиву ASSUME. NASM не стежитиме за тим, які значення ви поміщаєте в сегментні регістри і тому ніколи автоматично не генеруватиме префікс заміни сегмента.

NASM не підтримує моделі пам'яті

NASM не має жодних директив для підтримки різних 16-розрядних моделей пам'яті. Програміст повинен самостійно стежити, які функції передбачається викликати

"далеким викликом", а які - ближнім, і відповідно поміщати правильну форму інструкції RET (RETN або RETF ; NASM допускає застосування RET як альтернативну форму RETN); Крім того, програміст відповідає за кодування інструкцій CALL FAR, де це необхідно при виклику зовнішніх функцій, і повинен також стежити, які описи зовнішніх змінних є дальніми, а які ближніми.

Відмінності в обробці чисел із плаваючою точкою

NASM, на відміну MASM, використовує інші імена посилань на регістри співпроцесора: MASM посилається ці регістри як ST(0), ST(1) тощо., в NASMі цієї мети використовуються імена st0, st1 тощо. буд.

Починаючи з версії 0.96, NASM обробляє інструкції форм "nowait" так само, як і MASM-сумісні асемблери. Особлива обробка, використана у версіях 0.95 і молодше, була заснована на неправильному розумінні авторами.

Інші відмінності

З історичних причин NASM використовує ключове слово TWORD там, де MASM та сумісні з ним асемблери використовують TBYTE.

NASM оголошує резервований простір (неініціалізовані дані) не так, як MASM: там, де MASM-програміст може написати stack db 64 dup (?), NASM вимагає наступне - stack resb 64, що інтерпретується як "резервування 64 байт". Так як NASM обробляє ? як звичайний символ, ви можете написати щось на зразок ? equ 0, а потім використовувати dw?, Що буде можливо корисно для якихось цілей. Однак DUP залишається непідтримуваним синтаксисом.

ДОДАТОК В. Короткий посібник із використання симулятора Keil ARM

Середовище µVision IDE об’єднує керування проектами, середовище виконання, засоби асемблера, редагування вихідного коду та налагодження програм в одному потужному середовищі. µVision просте у використанні та прискорює розробку вбудованого програмного забезпечення. µVision підтримує кілька екранів і дозволяє створювати індивідуальні макети вікон у будь-якому місці екрану.

1. Завантажти симулятор Keil ARM ви зможете з веб-сайту Keil (https://www2.keil.com/mdk5/uvision/). Після входу на сайт необхідно обрати сам продукт для завантаження. Треба обрати ARM-MDK. Розробники досить інтенсивно оновлюють цей продукт. Остання версія на травень 2022 року була µVision V5.37. Запустіть пакет IDE, який ви завантажили.

2. Клацніть **select New µMicrovision Project**. На Рис.В.1 показано вікно нового проекту µVision.

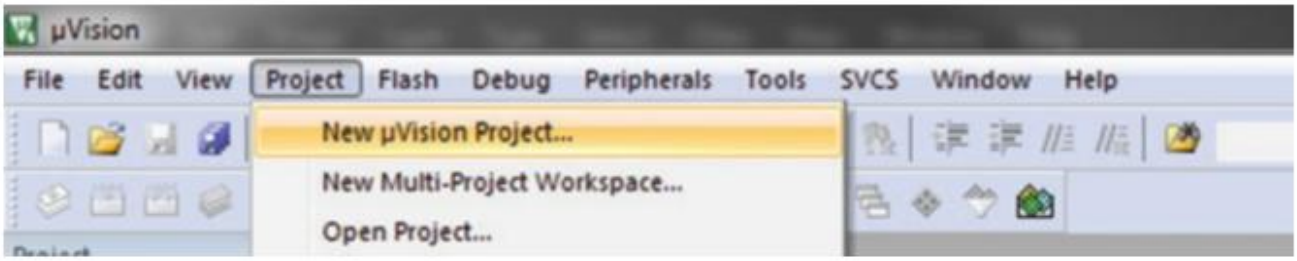

Рис. В.1 Створення нового проекту Keil µVision® IDE v5.37

3. Введіть ім’я файлу в поле Ім’я файлу. Скажімо, **MyFirstExample**

4. Натисніть **Save**.

5. Після цього з’явиться вікно з написом **Select Device Target 1**(Рис.В.2). Тепер вам потрібно сказати, яке сімейство процесорів і яку версію ви збираєтеся використовувати.

6. У списку пристроїв виберіть **ARM**, а потім у новому списку виберіть **ARM7 (Big Endian)**

Вибір процесора показано на Рис.В.2

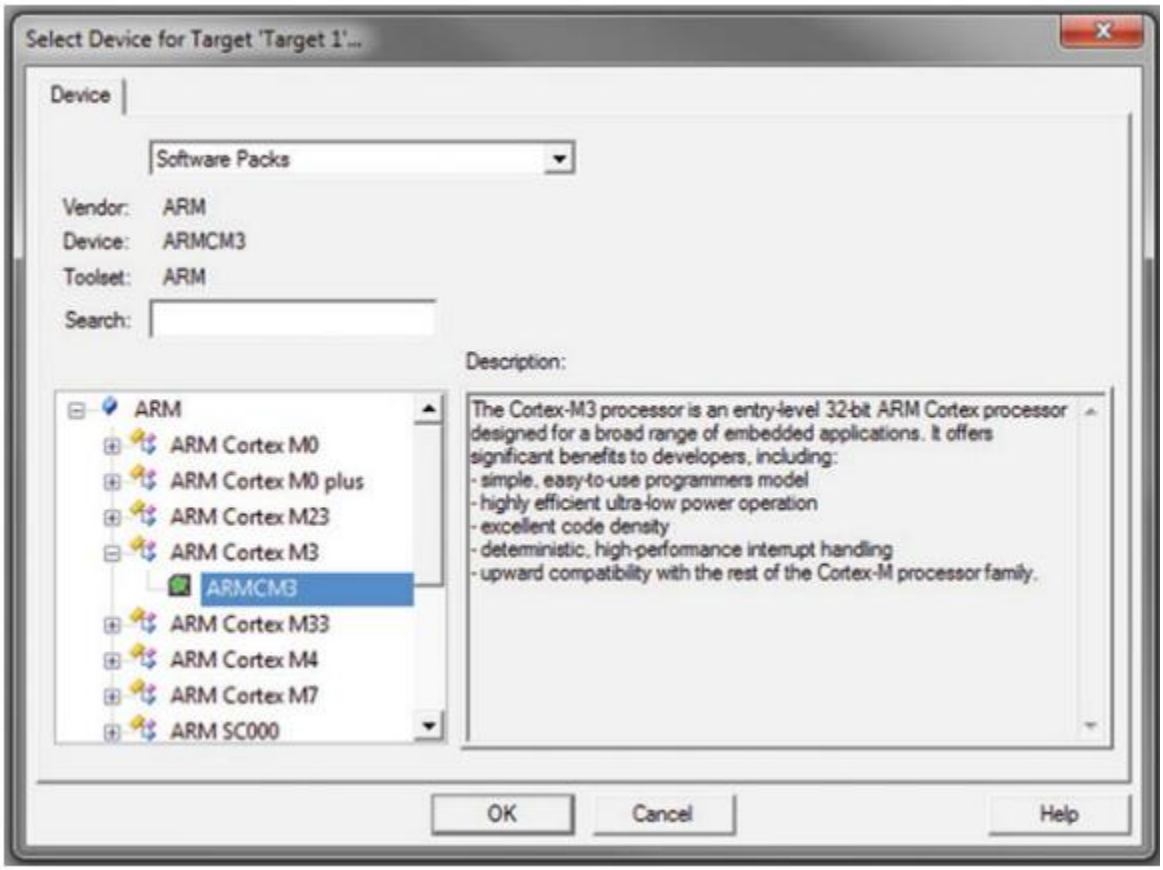

Рис. В.2 Вибір процесора ARM Cortex M3

7. Натисніть **OK**. Кнопка-бокс зникає. Ви повернетесь до головного вікна µVision.

8. Нам потрібно зайти в вихідну програму. Натисніть **File.** Виберіть **New** і натисніть на нього. Відкриється вікно редагування з позначкою **Text1**.. Тепер ми можемо написати просту програму. Напиклад:

AREA MyFirstExample, CODE, READONLY

ENTRY

MOV r0,#4 ;завантажити 4 у r0

MOV r1,#5 ;завантажити 5 у r1

ADD r2,r0,e1 ;додайте r0 до r1 і помістіть результат у r2

SBS ;примусовий нескінченний цикл шляхом розгалуження до цього рядка

END ;кінець програми

9. Після входу в програму виберіть у меню **File**, а потім **Save**. Вам буде запропоновано ввести назву файлу. Використайте MyFirstExample.s. Суфікс.s вказує на вихідний код.

10. Це поверне вас до вікна, яке тепер називається MyFirstExample і з кодом, встановленим з використанням угод ARM для виділення коду, чисел і коментарів.

11. Тепер нам потрібно налаштува середовище. Натисніть **Project** у головному меню. Зі списку, що розкриється виберіть **Manage**. Це дасть вам новий список. Виберіть **Components, Environment,Books**..

12. Тепер ви отримуєте форму з трьома вікнами. Під вікном праворуч виберіть **Add Files**.

13. Це дає вам звичайний вигляд файлів Windows. Клацніть стрілку розширення типу файлу та виберіть вихідний файл Asm (*.s*; *.src; *.a*). Тепер у вікні має з'явитися ваш власний файл MyFirstExample.s. Виберіть це та натисніть вкладку **Add**. Це додасть ваш вихідний файл до проекту. Потім натисніть **Close**. Тепер ви побачите свій файл у крайньому правому вікні. Натисніть **OK**, щоб вийти.

14. Ось і все. Ви готові зібрати файл.

15. Виберіть Project у верхньому рядку, а потім клацніть **Built target**.

16. У нижньому вікні під назвою **Build Output** ви побачите результат процесу асемблінгу.

17. Ви маєте побачити щось на зразок:

Build target 'Target 1'

assembling MyFirstExample.s...

linking...

Program Size: Code=16 RO-data=0 RW-data=0 ZI-data=0

"MyFirstExample.axf" - 0 Error(s), 0 Warning(s).

Чудове повідомлення 0 Error(s)! Якщо його ви не отримаєте, вам доведеться повторно відредагувати вихідний файл. А потім перейдіть до проекту та знову побудуйте ціль.

Запуск програми в Keil Simulator

Завантаживши програму у вигляді коду та зібравши її, потрібно її запустити. Ви можете або продовжити з того місця, на якому зупинилися після складання коду за допомогою **Build target**, або почати заново та завантажити код.

Якщо ви завантажите симулятор ARM, він відкриється в тому самому стані, у якому ви його закрили (тобто проект і вихідний файл завантажено). Якщо проект не відкрито, виберіть тег Project, а потім виберіть **Open Project**.. у вікні, що відкриється знизу. Якщо ви перебуваєте не в правильному каталозі, виберіть відповідний каталог звичайним способом. Потім клацніть **MyFirstExample.uvproj,** що є назвою проекту, який ми налаштували, а потім клацніть вкладку **Open**. Це завантажує ваш проект, і ви готові до роботи.

Щоб запустити код, виберіть **Debug** у верхньому меню. У спадному меню виберіть **Start/Stop Debug Session**. З’являється повідомлення про те, що ви перебуваєте в РЕЖИМІ ОЦІНЮВАННЯ, і ви натискаєте ОК, щоб закрити його. Ви повинні побачити екран, з яким ви можете працювати з ним так само, як і з будь-якою іншою програмою Windows, і використовувати вкладку View щоб відкривати інші вікна (наприклад, пам’ять дисплея).

Тепер ви можете виконувати код. Нас цікавить режим інструкції за інструкцією, який дозволяє виконувати інструкцію за раз. Якщо ви натиснете кнопку введення, ви виконаєте одну інструкцію.

Будь-які зміни ви зможете побачити в регістрах ліворуч. Ви також зможете побачити значення програмного лічильника PC та регістра стану CPSR. Зверніть увагу, що функціональна клавіша F11 виконує ту саму операцію.

Коли ви закінчите, ви повинні клацнути пункт **Start/Stop Debug Session** налагодження в меню **Project**. Це поверне вас до вихідного коду, який ви можете змінити за потреби. Після того, як ви його змінили, ви маєте знову використати команду Build target, щоб виконати повторний асемблінг.