КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра комп'ютерної інженерії

До захисту допущено: «На правах рукопису»
Завідувач кафедри _____ Юрій Бойко
« _ » _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему:

«ДОСЛІДЖЕННЯ ВИКОРИСТАННЯ СЕРЕДОВИЩА ANDROID ДЛЯ РОЗРОБКИ НИЗКОРІВНЕВИХ ПРОГРАМ»

Виконав:
студент 4-го курсу бакалаврату
денної форми навчання
спеціальності 123 Комп'ютерна інженерія
ОНП «_____»
Володимир Богурський _____

Науковий керівник:
кандидат фізико-математичних наук, доцент
Сергій Загороднюк _____

Рецензент:
_____

Засвідчую, що у цій бакалаврській роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент _____

Робота допущена до захисту в ЕК рішенням кафедри _____
від «__»_____ 2023 р., протокол № __.

Завідувач кафедри _____,
кандидат фізико-математичних наук, доцент
Бойко Юрій Володимирович

(підпис)

Київ – 2023

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра містить 69 сторінок, 31 рисунки, 6 таблиць, використано 20 інформаційних джерел.

Об‘єкт дослідження – можливість використання Termux на платформі Android для розробки низкорівневого програмного забезпечення..

Мета роботи – дослідження можливостей використання мови асемблера для ARM архітектури в Android-середовищі та створення системного програмного забезпечення за допомогою цієї мови програмування..

Проаналізовано популярні архітектури процесорів, а також особливості асемблерів для написання програмного забезпечення. Проаналізовано інструменти та способи написання асемблер програм для пристроїв на ARM архітектурі. Розроблена та відлагоджена асемблер програма в емуляторі терміналу Termux для Android системи, відповідно до мети дослідження.

ASSEMBLER, ARM, ANDROID, TERMUX

ЗМІСТ

ВСТУП

У світі, де сучасні мобільні пристрої є невід'ємною частиною нашого повсякденного життя, розробка програмного забезпечення для них є важливою галуззю розробки програмного забезпечення. Сьогоднішні мобільні пристрої мають різноманітні процесори, такі як ARM, x86 і MIPS, та використовують різні операційні системи, такі як iOS, Android та Windows Mobile.

У зв'язку з цим, важливим аспектом розробки програмного забезпечення, зокрема й для мобільних пристроїв, є розуміння низькорівневого програмування, зокрема мови асемблера. Мова асемблера є мовою, яка використовується для написання низькорівневого програмного забезпечення та драйверів пристроїв. Вона дозволяє програмістам написати програму на мові, зрозумілій для процесора, і забезпечує більш точний контроль над функціонуванням обчислювальних процесів.

Ця робота присвячена дослідженню можливості написання програм за допомогою низкорівневих мов програмування, а саме мови асемблера для ARM процесорів. Для цього будуть розглянуті доступні середовища та інструменти для розробки таких програм.

Метою цієї роботи є вивчення можливостей використання мови асемблера в емуляторі терміналу Termux на платформі Android для розробки низькорівневих програм. В результаті цієї роботи буде розглянуто процес розробки програм на мові асемблера в Termux, а також будуть проаналізовані результати експериментів для оцінки продуктивності та ресурсів, необхідних для виконання програм.

РОЗДІЛ 1

Огляд архітектур процесорів та розробки програм на мові асемблер для ARM процесорів

1.1 Класифікація процесорів та їх архітектур

Процесор - це головний компонент комп'ютера, відповідальний за виконання обчислень та керування іншими компонентами. Тип процесора визначається його призначенням - від малих мікроконтролерів для простих пристроїв до потужних процесорів для серверів та суперкомп'ютерів.

Процесори за призначенням можна поділити на наступні категорії:

Процесори загального призначення: ці процесори призначені для виконання широкого кола завдань і зазвичай використовуються в настільних комп’ютерах, ноутбуках і серверах.

Графічні процесори (GPU): ці процесори призначені для виконання складних графічних обчислень і використовуються для ігор, рендерингу відео та інших додатків, що інтенсивно працюють із графікою.

Цифрові сигнальні процесори (DSP): Ці процесори призначені для виконання завдань обробки сигналів і зазвичай використовуються в телекомунікаціях, обробці аудіо та відео та обробці зображень.

Спеціальні інтегральні схеми (ASIC): Ці процесори розроблені для конкретного застосування чи завдання та оптимізовані для продуктивності та енергоспоживання.

Мікроконтролери: ці процесори призначені для керування невеликими вбудованими системами та зазвичай використовуються в інтелектуальних пристроях, датчиках та інших малопотужних пристроях.

В цій роботі основний акцент буде покладено на процесори загального використання та мікроконтролери.

В залежності від призначення процесорів та їх особливостей, які будуть розглянуті далі в цьому розділі, вони виконують різні завдання в різних пристроях, починаючи від мобільних пристроїв, таких як смартфони і планшети, до великих серверних систем.

Для цього в них застосовуються різні інженерні рішення, такі як розмір і форма корпусу, методи охолодження, використання різних типів кеш-пам'яті, кількість ядер та їх технології, такі як гіпер-потокування, та інші функції.

Розглянемо деякі основні складові процесорів

Основні складові процесора включають регістри, арифметико-логічний блок та управляючий блок.

Регістри - це швидкодіюча пам'ять в процесорі, призначена для зберігання даних та інструкцій.

Арифметико-логічний блок виконує математичні операції та логічні порівняння між даними.

Управляючий блок відповідає за виконання інструкцій та керування іншими складовими процесора.

Саме ці складові і є основою процесору, які визначають його можливості, а відповідно признаяення, вони мають найвищу складність реалізації і як відповідно є основною складовою ціни процесору.

Ці обставини і породжують таку велику кількість архітектур процесорів.

Архітектура процесорів та її вплив на розробку програм

Архітектура процесора - це загальна структура, конструкція та логіка роботи процесора, які визначають, які операції можуть виконуватися в процесорі та як це відбувається.

Архітектура процесора визначається такими факторами:

1.Набір інструкцій: це набір команд, які процесор може виконувати. Кожен процесор має свій власний набір інструкцій.

2.Розрядність: це кількість бітів, яку процесор може обробляти одночасно. Зазвичай розрядність процесора визначається кількістю бітів, які можуть бути оброблені в режимі одного такту.

3.Кількість ядер: це кількість окремих процесорів, які можуть працювати в одному процесорі. Чим більше ядер у процесорі, тим більше завдань він може виконувати одночасно.

4.Кеш-пам'ять: це невеликий, але дуже швидкий обсяг пам'яті, який зберігає найбільш часто використовувані дані для швидкого доступу до них. Кеш-пам'ять розташовується безпосередньо на процесорі і використовується для зменшення часу доступу до пам'яті.

5.Архітектурні особливості: це спеціальні функції та конструктивні рішення, які реалізуються в архітектурі процесора для покращення його продуктивності та ефективності роботи.

Основними складовими процесора є реєстри, арифметико-логічний блок (ALU), блок керування та блок звернення до пам'яті.

Реєстри - це найшвидший вид пам'яті в процесорі, який забезпечує збереження тимчасових значень та результатів обчислень. Реєстри можуть бути загального призначення або спеціалізовані для виконання певних завдань.

ALU - це блок, який відповідає за виконання арифметичних операцій (додавання, віднімання, множення, ділення) та логічних операцій (AND, OR, XOR, NOT). ALU виконує операції над даними.

Всі ці складові впливають на арітектуру процесорів та відповідно впливає на можливості, швидкодію та енергоефективність процесору, що диктує способи його використання.

Відповідно архітектура процесора впливає на мову асемблера, оскільки мова асемблера повинна бути зрозумілою для процесора, на якому вона буде виконуватися. Кожна архітектура процесора може мати свої власні інструкції, режими адресації, розміри операндів та інші особливості, які впливають на синтаксис мови асемблера.

Наприклад, архітектура x86 використовує складні інструкції, що дозволяють здійснювати багато операцій з однією інструкцією, тому синтаксис асемблера x86 є складним та має багато різних форм. З іншого боку, архітектура ARM використовує більш прості інструкції, що дозволяє їй працювати з меншими обсягами пам'яті та зменшує кількість інструкцій, необхідних для виконання певної операції. Тому синтаксис асемблера ARM є більш простим та легким для розуміння, ніж у випадку x86.

Отже, враховуючи відмінності в архітектурі процесорів, мови асемблера для різних архітектур також відрізняються за синтаксисом та операціями, які вони підтримують.

Процесори на основі ARM архітектури використовуються в різноманітних сферах, включаючи мобільні пристрої, інтернет речей, системи автоматизації, сервери та багато іншого. Оскільки ARM процесори є ефективними з точки зору енергоспоживання, вони особливо популярні в мобільних пристроях, таких як смартфони та планшети. Крім того, ARM процесори використовуються в системах автоматизації, таких як контролери промислового призначення, а також в інтернеті речей, де вони забезпечують обробку даних та забезпечують підтримку різноманітних протоколів зв'язку.

У сфері серверів ARM процесори також стають все більш популярними завдяки своїм перевагам у витраті енергії та обробки паралельних завдань. Наприклад, AWS пропонує ARM-базовані екземпляри для підвищення ефективності обчислень та зниження витрат на енергію. ARM процесори також використовуються в системах вбудованих обчислень, таких як автомобільні

системи, пристрої для домашньої автоматизації, системи безпеки та контролю доступу, розумні телевізори та багато іншого.

1.2 Огляд асемблерів та їх особливостей

Загальна інформація про мову асемблера

Мова асемблера - це низькорівнева мова програмування, яка працює безпосередньо з апаратною частинами комп'ютера, такими як процесор, пам'ять, регістри та інші периферійні пристрої. Вона дозволяє програмістам створювати ефективний і оптимізований код, який працює безпосередньо з апаратурою і забезпечує більш точний контроль над пристроєм.

Мова асемблера зазвичай складається зі списку інструкцій процесора, які виконують певні операції з даними та змінними. Код на мові асемблера зазвичай отримується шляхом трасляції із вищих мов програмування, але його так само можна записати у вигляді текстового файлу, який потім можна скомпілювати в машинний код.

Одним з головних переваг використання мови асемблера є те, що вона дозволяє програмістам прямо контролювати процесор і його ресурси. Крім того, вона дозволяє створювати дуже ефективний код, що працює дуже швидко та забезпечує точний контроль над апаратурою.

Проте, мова асемблера має свої недоліки, серед яких можна виділити складність та часову затратність розробки коду, яка збільшується зі зростанням складності проекту. Також, код на мові асемблера зазвичай є менш читабельним та менш зрозумілим для інших програмістів, що може ускладнити підтримку та розширення проекту.

Детальніше способи використання мови асемблеру буде розглянуто далі.

Види асемблерів та їх призначення

В першу чергу асемблери поділяються на цільові архітектури на які вони розраховані

• x86

• RISC

• CISC

Але якщо ділити асемблери за призначенням, то можна отримати наступні категорії:

1.Асемблери призначені для розробки програмного забезпечення для конкретної архітектури та операційної системи. Вони зазвичай використовуються в розробці вбудованих систем або драйверів для певного пристрою.

Приклади локальних асемблерів: NASM (Netwide Assembler), TASM (Turbo Assembler), GAS (GNU Assembler).

2.Асемблери призначені для розробки програмного забезпечення для різних архітектур та операційних систем. Вони зазвичай використовуються в розробці великих проектів, які повинні працювати на різних платформах.

Приклади крос-платформених асемблерів: FASM (Flat Assembler), Yasm (Yet another Assembler), HLA (High Level Assembler).

3.Асемблери призначені для розробки програмного забезпечення для мікроконтролерів, які зазвичай мають обмежені ресурси та працюють в реальному часі. Вони зазвичай мають спеціальні функції та директиви, які дозволяють легко робити доступ до ресурсів пристрою.

Приклади асемблерів для мікроконтролерів: Keil μVision, MPLAB X, AVR Studio. В даній роботі особлива увага буде приділена процесорам саме на ARM архітектурі, причина цього буде наведена в розділі 1.4.

Особливості синтаксису асемблера для ARM архітектури

Всі асемблери в першу чергу відрізняються синтаксично один від одного в першу чергу, як і було описано через відмінності цільових архітектур.

Але також кожна цільова архітектура процесорі має свої режими.

Наприклад таких режимів для ARM архітектури може бути до 8, які наведені в таблиці 1.

Таблиця 1

Режим	Функція режиму
User (USR)	Режим, у якому працює більшість програм
Fast Interrupt (FIQ)	режим швидкого переривання
Interrupt (IRQ)	основний режим переривання.
Supervisor (SVC)	Вводиться під час скидання або під час виконання інструкції виклику супервізора (SVC).
Abort(ABT)	Режим, у який процесор переходить у разі виникнення помилки доступу до пам'яті
Undef (UND)	Вводиться під час виконання невизначеної інструкції
System (SYS)	Режим, у якому працює ОС, перегляд реєстру спільно з режимом користувача. (захищений режим використання операційною системою)
THUMB та THUMB 2	режим процесорів ARM, у якому використовується скорочена система команд. Вона складається з 36 команд, взятих із стандартного набору 32-розрядних команд архітектури ARM та перетворених до 16-розрядних кодів.

В залежності від режиму може навіть відрізнятися синтаксис асемблеру.

Так наприклад для режиму THUMB можна використовувати окремий синтаксис, відмінний від звичайного режиму, оскільки в ньому використовується обмежений набір інструкцій.

1.3 Системне програмне забезпечення для систем на базі ARM процесорів

Системне програмне забезпечення та його призначення

Системне програмне забезпечення (СПЗ) - це програмне забезпечення, призначене для забезпечення роботи комп'ютерної системи в цілому. Його функції можуть включати в себе керування апаратним забезпеченням,

забезпечення безпеки, управління ресурсами, віддаленого доступу та багато іншого.

Оскільки системне програмне забезпечення працює на більш низькому рівні, ніж більшість програм, його можна розглядати як набір інструментів для розробки програм. Часто системне програмне забезпечення має доступ до апаратного забезпечення, що дозволяє йому керувати ним та забезпечувати оптимальну роботу.

Системне програмне забезпечення може бути написане на різних мовах програмування, включаючи асемблер, C, C++ та інші. Програмування на асемблері є важливим для розробки системного програмного забезпечення, оскільки це дає можливість більш точного контролю апаратного забезпечення та оптимізації роботи програми.

Важливість системного програмного забезпечення полягає в тому, що воно дозволяє програмістам розробляти різноманітні програми та додатки, які можуть взаємодіяти з апаратним забезпеченням комп'ютера або іншого електронного пристрою. Системне програмне забезпечення забезпечує основні функції, що потрібні для працездатності операційних систем, драйверів, мов програмування та інших програм.

Основні функції системного програмного забезпечення включають створення та управління процесами, взаємодію з апаратним забезпеченням, керування пам'яттю, роботу з мережею та файловою системою. Без системного програмного забезпечення, комп'ютери та інші електронні пристрої не можуть виконувати ніяких завдань.

Для створення системного програмного забезпечення, часто необхідно використовувати програмування за допомогою асемблера та мов низького рівня, оскільки це дозволяє більш ефективно використовувати ресурси

комп'ютера. Наприклад, програмування драйверів вимагає знання апаратного забезпечення та асемблера, щоб забезпечити оптимальну роботу пристрою.

Роль мови асемблера в системному програмуванні та її переваги

Один з найбільш відомих прикладів програмного забезпечення, що працює на рівні операційної системи, - це драйвер пристрою. Драйвер - це програма, яка дозволяє операційній системі взаємодіяти з певним пристроєм, таким як принтер, сканер, монітор, дискові накопичувачі тощо.

Драйвери використовуються для забезпечення взаємодії між операційною системою та пристроями, що підключаються до комп'ютера. Вони дозволяють операційній системі коректно ідентифікувати та належним чином працювати з пристроями, забезпечуючи їхню правильну роботу та оптимальну продуктивність.

Також до прикладів програмного забезпечення, яке працює на рівні операційної системи, можна віднести різноманітні системні утиліти, такі як диспетчер завдань, редактор реєстру, засоби архівації тощо. Ці програми забезпечують доступ до різноманітних системних ресурсів та дозволяють користувачеві виконувати різні завдання на комп'ютері.

Крім того для програмування системних і не тільки програм за допомогою мови С, часто використовуються асеблерні вставки.

Асемблерні вставки – це частини в програмі які реалізовані саме за допомогою асмеблеру, для більш вузького контролю ресурсів, або підвищення швидкодії.

1.4 Особливості використання мови асемблера на ARM архітектурі

Процесори на ARM-архітектурі є досить популярними завдяки своїм перевагам порівняно з іншими процесорами. Основні переваги процесорів на ARM-архітектурі включають:

1.Енергоефективність: Процесори на ARM-архітектурі є дуже енергоефективними, що означає, що вони споживають менше енергії під час роботи, ніж інші процесори. Це особливо важливо для портативних пристроїв, таких як смартфони, планшети та інші мобільні пристрої.

2.Низька вартість: Процесори на ARM-архітектурі зазвичай коштують дешевше, ніж інші процесори, що робить їх дуже популярними для використання в пристроях з обмеженим бюджетом.

3.Висока продуктивність: Хоча процесори на ARM-архітектурі є менш потужними, ніж деякі інші процесори, вони забезпечують достатню продуктивність для більшості завдань, які потребуються в портативних пристроях.

Процесори на ARM-архітектурі використовуються в багатьох пристроях, таких як:

1.Смартфони та планшети: Більшість сучасних смартфонів та планшетів використовують процесори на ARM-архітектурі.

2.Комп'ютери одноплатні: Одноплатні комп'ютери, такі як Raspberry Pi та BeagleBone Black, використовують процесори на ARM-архітектурі.

3.Мікроконтролери: ARM-процесори використовуються для створення мікроконтролерів для різноманітних пристроїв, таких як автомобілі, IOT пристрої

4.Датчики, тощо

Гарним показником популярності ARM процесорів є рекордна кількість чіпів які реалізуються на даній архітектурі. Так у звіті компанії arm повідомляють, що тільки за 1 квартал 2020 року було відвантажено 6.7 міліарди чіпів на базі Arm. Також 4.4 міліарди чіпів для IoT, а також графічних чіпів.

Що підтверджує популярність та актуальність роботи з чіпами ARM архітектури та розробку не тільки звичайного програмного забезпечення, а й системного програмного забезпечення під ARM процесори.

Застосування мови асемблера для ARM архітектури

Існує кілька причин, чому приймається рішення написати програму на мові асемблера для ARM архітектури, або з використанням вставок. Ось деякі з них:

1.Продуктивність: Мова асемблера дає можливість безпосередньо керувати процесором та дозволяє писати більш ефективний код для ARM процесорів, що може підвищити продуктивність програми.

2.Доступ до апаратних можливостей: Деякі апаратні можливості пристроїв на ARM архітектурі можуть бути недоступними через високий рівень абстракції, що надають високорівневі мови програмування. Написання програм на мові асемблера дозволяє безпосередньо отримати доступ до цих можливостей та їх використовувати.

3.Вбудовані системи: ARM процесори широко використовуються в вбудованих системах, де потрібна висока продуктивність та ефективність роботи з обмеженими ресурсами. Написання програм на мові асемблера може бути найбільш ефективним способом реалізації вбудованих систем з високими вимогами до продуктивності та низькими вимогами до ресурсів.

4.Операційні системи: Операційні системи, такі як Linux та Android, Windows, тощо - підтримують написання драйверів на мові асемблера для ARM процесорів. Це може забезпечити більш точний та ефективний контроль над апаратурою пристроїв, що працюють на ARM архітектурі.

Загалом, написання програм на мові асемблера для ARM архітектури може бути корисним у випадках, коли потрібна максимальна продуктивність та точність керування апаратними можливостями пристроїв.

Розглянемо деякі приклади використання мови асемблера для ARM архітектури.

Існує багато прикладів використання мови асемблера для ARM архітектури, особливо в області вбудованих-систем та вбудованих пристроїв. Деякі з них:

1.Розробка драйверів пристроїв: Мова асемблера є чудовим інструментом для розробки драйверів пристроїв на ARM процесорах, оскільки вона дозволяє точну маніпуляцію з реєстрами пристроїв та обмеженнями зниження витрат ресурсів.

2.Розробка системних додатків: Мова асемблера також використовується для розробки системних додатків, які працюють на рівні операційної системи, такі як драйвери файлової системи, сигнальні обробники та інші.

3.Швидкодія: В деяких випадках, де потрібна максимальна швидкість виконання, мова асемблера може бути кращим вибором, оскільки вона дозволяє точно оптимізувати код для ARM процесора, що знижує час виконання та підвищує продуктивність.

4.Взаємодія з апаратним забезпеченням: Мова асемблера дозволяє здійснювати прямий доступ до апаратного забезпечення, такого як вхід/вихід, пам'ять та інші ресурси, що робить її корисною для вбудованих систем та систем, які вимагають низького рівня доступу до апаратури.

5.Навчання: Мова асемблера є чудовим інструментом для навчання архітектури ARM та принципів роботи процесора, оскільки дозволяє отримати точне розуміння того, як працюють різні команди та інструкції процесора.

Цілі використання мови асемблера для ARM архітектури

Однією з основних відмінностей між програмами, написаними на мові асемблера та вищих мовах програмування для ARM архітектури є рівень абстракції та зручність написання коду.

Мова асемблера є найнижчим рівнем програмування та вимагає безпосереднього звернення до апаратної частини пристрою.Саме тому програми, написані на мові асемблера, мають вищу продуктивність та точність управління апаратурою, оскільки програміст має повний контроль над кожним байтом коду та може взаємодіяти безпосередньо з апаратурою.

З іншого боку, програми, написані на вищих мовах програмування, таких як C/C++, Python, Java, мають більш високий рівень абстракції та спрощують процес написання коду. Це дає можливість розробникам швидко створювати програми, проте це може призвести до втрати контролю над окремими ділянками коду та зниження продуктивності програми.

У випадку ARM архітектури, мова асемблера може бути використана для написання своїх міні-операційних систем, драйверів пристроїв та інших системних програм, де важлива швидкість та точність управління апаратурою. Однак, для розробки застосунків на вищому рівні абстракції, використання вищих мов програмування, таких як C/C++, може бути більш зручним та продуктивним.

Переваги та недоліки використання мови асемблера для ARM архітектури та підходів до використання її в програмуванні.

Переваги використання мови асемблера для ARM архітектури:

1.Швидкість: програми, написані на мові асемблера, зазвичай працюють швидше, ніж ті, що написані на вищих мовах програмування, оскільки вони можуть бути оптимізовані для конкретної архітектури.

2.Прямий доступ до ресурсів: програми на мові асемблера можуть працювати з пристроями безпосередньо із рівня апаратури, що дає можливість прямого доступу до ресурсів.

3.Менша вимога до ресурсів: програми на мові асемблера зазвичай потребують менше ресурсів, таких як пам'ять і процесорний час, що є важливим фактором для пристроїв з обмеженими ресурсами.

Недоліки використання мови асемблера для ARM архітектури:

1.Складність: програмування на мові асемблера вимагає високої кваліфікації і досвіду в програмуванні, що може бути складним для початківців.

2.Стрімке застарівання: мова асемблера має тенденцію застарівати дуже швидко, оскільки вона прив'язана до конкретної архітектури і піддається впливу нових технологій.

3.Підтримка: підтримка програм на мові асемблера зазвичай вимагає більше зусиль, оскільки вони складніші для розуміння і зміни, що може призвести до зниження продуктивності і підвищення витрат.

Підходи до використання мови асемблера для ARM архітектури:

1.Використання в критичних випадках: мова асемблера може використовуватись для написання коду, що обробляє критичні завдання, що потребують високої швидкості та прямого доступу до ресурсів.

2.Використання для оптимізації: мова асемблера може бути використана для оптимізації вузьких місць у коді, написаному на вищих мовах програмування.

3.Використання для драйверів та пристроїв: мова асемблера може бути використана для написання драйверів та програмного забезпечення для пристроїв, що потребують безпосереднього доступу до ресурсів.

4.Загалом, використання мови асемблера для ARM архітектури має свої переваги та недоліки, і використовується переважно для критичних завдань та оптимізації коду. Однак, для розробки повноцінних програм, зазвичай використовують вищі мови програмування, які є більш зручними для програмістів та мають широку підтримку та інструментарій.

1.5 Висновки до розділу

У даному розділі була розглянута основна інформація про процесори, та наведена їх класифікація. Наведено опис архітектур процесорів та їх сфер використання в залежності від архітектури. Роглянуто питання розробки системного програмного забезпечення з використанням мови асемблеру. Було визначено поняття системного програмного забезпечення, його основні функції та завдання. Було описано цілі використання мови асемблера для ARM архітектури, такі як оптимізація швидкодії та зменшення розміру програм.

Сьогодні найпополярнішою та найперспективнішою є саме ARM архітектура, яка використовується всюди. Відповідно стоїть Актуальність написання системного програмного забезпечення для пристроїв на процесорах цієї архітектури є безумовною. В наступних розділах буде оглянуто способи написання системного програмного забезпечення для пристроїв на ARM процесорах.

РОЗДІЛ 2

ВИБІР АРХІТЕКТУРИ ТА ЗАСОБІВ РЕАЛІЗАЦІЇ НИЗЬКОРІВНЕВОЇ ПРОГРАМИ ДЛЯ ARM СИСТЕМ

2.1 Огляд ARM архітектури

ARM архітектури.

ARM (Advanced RISC Machines) - це сімейство архітектур мікропроцесорів, що розробляються компанією ARM Holdings. ARM є одним з найпоширеніших архітектурних рішень в електронній промисловості, і його можна знайти в різноманітних пристроях, від мобільних телефонів до автомобільних систем управління.

Архітектура ARM - це архітектура зі зведенням на RISC (Reduced Instruction Set Computing), що означає, що процесор виконує складні функції, комбінуючи більш прості інструкції. У порівнянні з іншими архітектурами, ARM відзначається високою продуктивністю при низькому споживанні енергії, що робить його популярним в пристроях з обмеженою потужністю батареї.

ARM архітектура має високу продуктивність та низьке енергоспоживання завдяки кільком факторам. Перш за все, процесори ARM виготовляються з використанням технології зменшення розміру транзисторів, що дозволяє розміщувати більше транзисторів на кристалі і, отже, підвищує їх продуктивність. Крім того, вони використовують простішу архітектуру, ніж процесори інших архітектур, що зменшує їх енергоспоживання.

Також ARM архітектура має досить низьку кількість операцій, що виконуються за один тактовий імпульс, що зменшує частоту роботи процесора, а отже його енергоспоживання. Більшість процесорів ARM також підтримують технологію зменшення напруги (наприклад, технологію "big.LITTLE"), яка

дозволяє їм працювати в різних режимах, залежно від завдання, що виконується, що також зменшує їх енергоспоживання.

Крім того, ARM архітектура є модульною та масштабованою, що дозволяє розробникам використовувати різні конфігурації для різних пристроїв, що відповідає їх потребам щодо продуктивності та енергоспоживання. Наприклад, процесори ARM можуть бути використані в мобільних пристроях з обмеженим енергоспоживанням, а також у потужних серверах.

Таким чином, ARM архітектура має високу продуктивність та низьке енергоспоживання завдяки використанню новітніх технологій, ефективній архітектурі та можливості масштабуван

Одна з головних переваг ARM архітектури полягає в її масштабованості. ARM може бути використаний для виробництва мікроконтролерів, мікропроцесорів та серверів. Крім того, ARM має дуже широку підтримку з боку відомих виробників, таких як Samsung, Qualcomm, Apple, NVIDIA та інших.

Оглядаючи ARM архітектуру, важливо зазначити, що вона розділена на різні версії, що мають різну функціональність та підтримку інструкцій. Наприклад, ARMv7 зазвичай використовується в сучасних мобільних телефонах та планшетах, тоді як ARMv8 зазвичай використовується в сучасних серверних процесорах.

Версії архітектур в свою чергу поділяються від розрядності процесору

До 32-бітних ARM архітектур належать: ARMv1, ARMv2,ARMv3, ARMv4,ARMv5,ARMv6,ARMv7

До 64- бітних ARM архітектур належать:ARMv8-A

Останн версії ARM архітектур наведено в таблиці 2.

Таблиця 2

Архітектура	Рік випуску	32/64-бітна	Набір інструкцій	Кількість ядер	Кількість регістрів	Технологія виготовлення	Підтримка віртуалізації
ARMv6	2002	32-бітна	ARMv6	1	16	90 нм	Ні
ARMv7	2004-2011	32/64-бітна	ARMv7	1-4	31	45-28 нм	Так
ARMv8-A	2011-2014	64-бітна	ARMv8	1-8	31	16-7 нм	Так

Ці архітектури мають різні особливості та призначення. ARMv6 була розроблена для використання в пристроях з обмеженими можливостями, таких як мобільні телефони(2002 рік), але більш нові архітектури, такі як ARMv7 та ARMv8-A, стали домінувати в більш потужних пристроях, таких як смартфони та сервери.

ARMv7 стала дуже популярною в наступні десятиліття(після v6) та використовується в багатьох мобільних та вбудованих системах. Вона підтримує 32-бітні та 64-бітні операційні системи та має багато додаткових можливостей, таких як підтримка віртуалізації.

ARMv8-A є останньою версією архітектури ARM і є 64-бітною. Вона має підтримку більш ефективної обробки даних та кращої підтримки віртуалізації. ARMv8-A часто використовується в потужних пристроях, таких як сервери та смартфони високого класу.

Характеристиками для версії ARMv8-A наведено в таблиці 3:

Таблиця 3

Характеристика	Опис
Архітектура	ARMv8-A
Ядер	До 8
Частота	До 3 ГГц
Кеш-пам'ять L1	Інструкційна - 64 КБ, даних - 64 КБ на ядро
Кеш-пам'ять L2	До 4 МБ на чіпі
Кеш-пам'ять L3	Може бути присутнім на рівні чипу
Розрядність	64 біт
Виконання інструкцій	Одночасне виконання 2 інструкцій (одну на відгалуження та одну на обчислення)
Підтримка віртуалізації	ARM Virtualization Extensions

Шифрування	AES, SHA-1, SHA-256
Підтримка ОС	Android, iOS, Windows 10

Сама ж версія архітектури ARMv8-A має вже свої власні версії:

ARMv6 використовується в основному в старіших мобільних пристроях, наприклад, Nokia N95 та Sony Ericsson W995. Також ARMv6 використовується в низькопотужних смартфонах та інших пристроях Інтернету речей.

ARMv7 використовується в багатьох пристроях, таких як смартфони, планшети, ігрові консолі та інші пристрої. ARMv7 забезпечує підтримку 32-бітних операційних систем, таких як Android та iOS, та забезпечує високу продуктивність в області мобільних пристроїв.

ARMv8-A використовується в багатьох новіших смартфонах та інших пристроях, таких як Apple iPhone 8 і Samsung Galaxy S8. ARMv8-A забезпечує 64-бітну адресацію пам'яті та підтримку 64-бітних операційних систем, таких як iOS та Android.

ARMv8.1-A використовується в деяких нових смартфонах, таких як Samsung Galaxy S9 і OnePlus 6. ARMv8.1-A має деякі покращення порівняно з ARMv8-A, включаючи покращення продуктивності та підтримку нових технологій.

ARMv8.2-A використовується в деяких нових смартфонах та інших пристроях, таких як Samsung Galaxy S10 та Huawei P30. ARMv8.2-A має покращені можливості в області шифрування та дешифрування даних.

ARMv8.3 була випущена в 2016 році і містила вдосконалення для обробки векторів даних, зокрема збільшення ширини векторів на 128 бітів, що зменшило кількість виконуваних операцій і поліпшило продуктивність векторної обробки даних. ARMv8.3 також додала нові інструкції для роботи з пам'яттю, такі як "PAC" (pointer authentication code) і "PAN" (privilege attribute and names).

ARMv8.4 була випущена в 2017 році і містила додаткові покращення в області обробки векторів даних, включаючи підтримку змінної довжини векторів, що дозволяє ефективніше виконувати різноманітні операції з даними, включаючи машинне навчання та обробку сигналів. Крім того, ARMv8.4 додала нові інструкції для роботи з шифруванням і дешифруванням даних, що підвищило рівень безпеки.

ARMv8.5 була випущена в 2018 році і містила нові функції для покращення продуктивності, зокрема зміну формату даних для зменшення розміру коду і підвищення швидкодії. Також вона містить додаткові інструкції для обробки векторів даних і нові інструкції для роботи з пам'яттю, що забезпечують більшу ефективність при обробці даних.

ARMv8.6 була випущена в 2020 році і містила нові функції для покращ ення продуктивності, безпеки та ефективності енергоспоживання. Одним із основних покращень ARMv8.6 є технологія "Confidential Compute Architecture" (CCA), яка дозволяє обчислювальним процесорам зберігати та оброблювати дані в безпечному середовищі, що захищає дані від вторгнення ззовні.

Крім того, ARMv8.6 має покращену підтримку апаратної віртуалізації та мережевих функцій, що дозволяє ефективніше виконувати обчислення в хмарних сервісах та віртуальних середовищах.

Загалом, кожна нова версія ARM архітектури має свої вдосконалення, які дозволяють підвищувати продуктивність, безпеку та енергоефективність обчислювальних систем. Відповідно до цього, різні версії ARM архітектури використовуються в різноманітних пристроях, від мобільних телефонів і планшетів до серверів та обчислювальних систем великої потужності.

Огляд ассемблерів для написання програм для Arm архітектури

На сьогоднішній день існує велика кількість різних асемблерів для все-можливих задач та сфер використання. Найпопулярніші асемблери розгялнуті в таблиці 4:

Таблиця 4

Асемблер	Сфера використання	Які архітектури підтримує	Розрядність	ОС
NASM	- Розробка операційних систем - Розробка драйверів - Вбудовані системи - Ігри - Криптографія	x86, x86-64, IA-64, ARM, ARM64, PowerPC, SPARC	16/32/64-bit	Linux, Windows, macOS, FreeBSD, NetBSD, OpenBSD
MASM	- Розробка драйверів - Створення компонентів Windows	x86, x86-64	16/32-bit	Windows
FASM	- Розробка операційних систем - Розробка драйверів - Вбудовані системи - Ігри	x86, x86-64, ARM, ARM64, MIPS, PowerPC, SPARC	16/32/64-bit	Windows, Linux, macOS, FreeBSD
GAS (GNU Assembler)	- Розробка драйверів - Створення компонентів Linux	x86, x86-64, ARM, PowerPC, SPARC, MIPS, RISC-V, 68k	32/64-bit	Linux, macOS, FreeBSD
YASM	- Розробка операційних систем - Розробка драйверів - Вбудовані системи - Ігри	x86, x86-64	32/64-bit	Linux, Windows, macOS, FreeBSD
HLA	- Навчання асемблеру - Розробка додатків	x86, x86-64	32/64-bit	Windows

NASM (Netwide Assembler) - це крос-платформовий асемблер, який підтримує багато архітектур, включаючи x86, x86-64, ARM, PowerPC та багато інших. Цей асемблер часто використовується для створення операційних систем, драйверів та інших системних програм.

MASM (Microsoft Macro Assembler) - це асемблер, створений компанією Microsoft, який використовується для розробки програм під архітектуру x86 та x86-64. Він використовується для створення програм на мовах програмування, таких як Сі та C++.

FASM (Flat Assembler) - це крос-платформовий асемблер, який підтримує багато архітектур, включаючи x86, x86-64, ARM, PowerPC та багато інших. Цей асемблер відомий своєю швидкодією та ефективністю, що дозволяє використовувати його для створення оптимізованих програм.

GAS (GNU Assembler) - це асемблер, розроблений як частина GNU Compiler Collection. Він підтримує багато архітектур, включаючи x86, ARM та MIPS, та використовується для розробки операційних систем, драйверів та інших системних програм.

TASM (Turbo Assembler) - це асемблер, розроблений компанією Borland. Він використовується для розробки програм на архітектурі x86 та x86-64 і відомий своєю швидкодією та можливістю роботи з великими проектами.

YASM (Yet Another Assembler) - це крос-платформовий асемблер, який підтримує багато архітектур, включаючи x86, x86-64, ARM, PowerPC та багато інших. Він відомий своєю підтримкою векторних операцій та SSE, що дозволяє використовувати

Але серед всіх них для розробки для ARM процесорів можна виділити тільки GAS, оскільки він є безкоштовним та вже є вбудованим в GCC та поставляється разом з пакетом GNU Binutils.

Є кросплатформеним для більшості процесорних архітектур.

А отже це найкращий вибір для швидкої та ефективної розробки програм за допомогою асемблеру для ARM архітектур.

2.2 Огляд способів написання програм на ассемблері для ARM архітектури

Використання текстових редакторів та компіляторів для написання та компіляції програм на ассемблері для ARM архітектури

Для написання програм на асемблері можна використовувати абсолютно будь-який текстовий редактор. Його вибір абсолютно не впливає на кінцевий результат.

Натомість без компілятору обійтися не вийде

Для процесорів ARM існує досить широкий вибір компіляторів, внутрішня реалізація яких залежить безпосередньо від виробника даного ARM-процесора чи розробника IDE до роботи з ARM-процесорами. Офіційним компілятором ARM, безпосередньо від компанії ARM, є ARM Compiler 6, який входить до IDE DS-5 Development Studio та підтримує компіляцію програм мовами Сі та C++. Порівняння компіляторів наведено в таблиці 5.

Таблиця 5

Компілятор	Виробник	Підтримувані мови	Наявність комерційної ліцензії	Підтримка архітектур	Операційні системи
MDK-ARM	ARM	C, C++, Assembler, Python	Так	ARM, Cortex-M, Cortex-A/R	Windows, Linux
IAR Compiler	IAR Systems	C, C++, Assembler	Так	ARM, AVR, MSP430, RISC-V	Windows, Linux, macOS
GCC Compiler	GNU Project	C, C++, Assembler	Ні	ARM, AVR, MIPS, PowerPC, RISC-V, x86 та інші	Windows, Linux, macOS
Precompiled GCC Compiler	ARM, Linaro	C, C++, Assembler	Ні	ARM, Cortex-M, Cortex-A/R	Windows, Linux, macOS

Серед всіх доступних компіляторів варто виділити саме GCC, який в свою чергу використовує GAS.

GAS має велику кількість можливостей, включаючи підтримку різних форматів файлів об'єктного коду, таких як ELF, COFF, a.out, а також виведення лінійних адрес і переклад до різних форматів, таких як Intel Hex і S-Record.

GAS можна використовувати як з командним рядком, так і з різними інтегрованими середовищами розробки (IDE), такими як Eclipse і Code::Blocks.

Одним з переваг використання GAS є те, що він є безкоштовним і відкритим, тому він доступний для всіх користувачів Linux, і може бути легко налаштований з іншими інструментами з відкритим кодом, такими як GDB (GNU Debugger) для налагодження програм.

Узагальнюючи, GAS є потужним інструментом для розробки програм на асемблері для ARM архітектури на операційній системі Linux, і може бути використаний з будь-яким текстовим редактором або IDE, залежно від вподобань розробника.

Використання інтегрованих середовищ розробки (IDE) для написання та компіляції програм на ассемблері для ARM архітектури

Інтегроване середовище розробки (IDE) - це програмне забезпечення, яке надає розробникам зручне та повне середовище для написання коду, компіляції та налагодження програм. Використання IDE може значно спростити процес розробки, зменшити кількість помилок та покращити продуктивність розробника.

Існує декілька інтегрованих середовищ розробки, які підтримують розробку програм на ассемблері для ARM архітектури. Деякі з найпопулярніших інструментів наведені нижче:

Keil MDK-ARM: Інтегроване середовище розробки, розроблене компанією ARM, що спеціалізується на розробці програмного забезпечення для ARM мікроконтролерів. MDK-ARM включає в себе текстовий редактор, асемблер, компілятор C / C ++ та інші інструменти для розробки вбудованого програмного забезпечення на ARM архітектурі.

IAR Compiler: Інтегроване середовище розробки, розроблене компанією IAR Systems, яке підтримує розробку програмного забезпечення для багатьох

мікроконтролерів, включаючи ARM. IAR Compiler має багатофункціональний текстовий редактор, асемблер, компілятор C / C ++ та інші інструменти для розробки вбудованого програмного забезпечення.

Eclipse IDE: Інтегроване середовище розробки, яке підтримує розробку програмного забезпечення для ARM, а також для інших архітектур. Eclipse IDE має вбудований текстовий редактор, підтримує роботу з асемблером та мовами програмування, такими як C / C ++, Java та інші. Воно також надає можливість підключення різних плагінів та інструментів для розробки програмного забезпечення. Але для написання ARM ассемблер коду потрібно встановити Android NDK.

GCC Compiler: Це безкоштовний компілятор, що підтримує багато архітектур, включаючи ARM. GCC має текстовий редактор, асемблер та компілятор C / C ++. Воно також надає можливість підключення до інших інструментів для розробки програмного забезпечення. Нажаль на відміну від інших – він не має свого графічного інтерфейсу і робота з ним буде менш зручна в порівнянні з описаними вище.

Серед всіх повноцінних IDE та просто компіляторів можна виділити саме Keil MDK-ARM та GCC.

Keil MDK-ARM – офіційне IDE від компанії ARM, яке має дуже велику вартість та сильно урізану безкоштовну версію.

GCC – в свою чергу безкоштовний. Його можна встановити на обрану операційну систему, а для Linux він вже предвстановлений.

2.3 Огляд застосунків для Android, для написання низькорівневих програм

Як вже відомо смартфони на базі операційних систем Android, IOS та деяких Windows Phone мають процесори ARM архітектури, що дозволяє

виконувати прямо на них асемблерні програми з використанням ARM інструкцій.

Але оскільки продукція компанії Apple має ліценцію на зміну та використання ARM архітектури, то процесори компанії Apple хоч і мають arm процесори в основі, але ці процесори мають свої власні набори інструкцій, що унеможливлює використання техніки Apple для проведення даного дослідження.

В свою чергу перші смартфони на Windows Phone та їх наступники вже відійшли у вічність через свою неактуальність, тому в нашому дослідженні їх розглядати недоречно.

Таким чином залишається використання або Android систем або одноплатових комп'ютерів такі як сімейство Raspberry Pi,(окремо варто звернути увагу, що процес вітуалізації в IDE не дозволить в повній мірі використати можливості ARM архітектури та не буде розглянуто далі) .

Оскільки майже кожен читач має у вільному доступі смартфон на базі Android доречно буде дослідити можливість запуску assembler коду саме на Android системі.

Обмеження Android систем.

Кореневий доступ в Android означає можливість доступу до прав адміністратора або «суперкористувача» на пристрої Android. За замовчуванням пристрої Android мають набір обмежень і дозволів, які обмежують дії програм і користувачів на пристрої. Ці обмеження призначені для забезпечення безпеки та стабільності пристрою, а також для запобігання користувачам від випадкового чи навмисного заподіяння шкоди системі.

Огляд Android-терміналів та застосунків, які підтримують виконання ассемблерних інструкцій.

Перед початком розгляду даних застосунків звернемо увагу, що після оновлення політики Play Market від 2021 року у програм відібрали більшість прав таких як внутрішній запуск, внутрішні оновлення та запуск виконуючих файлів в середині середовища android (в першу чергу стосується емуляторів терміналу). Тому деякі застосунки які присутні в Play Market мають обрізаний функціонал і мають бути встановлені з офіційних сайтів або репозиторіїв розробників, щоб обійти ці обмеження.

Для початку розглянемо застосунки які позиціонують себе як мобільні компілятори та текстові редактори з можливістю компілювати код

Dcoder - це зручний редактор коду для Android, який підтримує багато мов програмування, включаючи асемблер(NASM). За допомогою цього додатку можна не тільки редагувати код, але й виконувати його.

Під час запуску коду Dcoder використовуються хмарні компілятори. Таким чином код запускається не на пристрої з ARM процесором, а десь у хмарі за допомогою віртуалізацій. Це пояснює яким чином можна запускати NASM код на телефоні – він не запускається локально, бо це не можливо, а виконується на сервері та повертає результат.

Code Editor - це зручний текстовий редактор, що дозволяє редагувати код із підсвічуванням синтаксису для різних мов програмування, включаючи асемблер. Застосунок Code Editor також має онлайн компілятор, наприклад він підтримує також тільки nasm, тому використання даного застосунку для розробки коду для ARM архітектури неможливе .

Compile+ та багато інших компіляторів, що можна знайти в Play Market, під Android не підтримують написання коду на асемблері. За допомогою них часто можна написати код на різних мовах програмування, таких як C++, C#, Java, Python, Ruby, Perl, PHP, JavaScript і багатьох інших, але не на асемблері. Ці застосунки в свою чергу також використовують хмарні компілятори

Тепер розглянемо декілька застосунків які позиціонують себе як емулятори терміналу, що дозволяють напряму працювати із linux системою на телефоні.

Qute — це емулятор терміналу, недоступний для Android. Однак для Android доступні інші програми емулятора терміналу, які можуть надавати подібні функції.

Будучи програмою емулятором терміналу для Android, Qute дозволяє користувачам взаємодіяти з системою Android через інтерфейс командного рядка. Він підтримує низку поширених команд і утиліт терміналу, таких як ls, cd, pwd і grep, і може використовуватися для таких завдань, як запуск сценаріїв, доступ до системних журналів і виконання завдань системного адміністрування.

Qute для Android також має деякі унікальні функції, такі як жести пальця для полегшення навігації, плаваюче вікно терміналу, яке можна переміщати по екрану, і настроювані колірні схеми та розмір шрифту. Крім того, він підтримує підключення SSH і може використовуватися для підключення до віддалених серверів.

Взаємодія між Qute і системою Android подібна до інших програм емулятора терміналу. Користувачі вводять команди в термінал, і Qute надсилає ці команди в систему Android. Результат цих команд відображається в програмі емулятора терміналу Qute. Проте в даному застосунку не доступна більшість команд для компіляції та запуску кодів безпосередньо в системі. В першу чергу це пов’язано з політикою Play Market .

Terminal Emulator for Android – позиціонує себе як емулятор терміналу для Android, але для роботи потрібно вибрати директорію на пристрої, яку система дозволить змінювати. Обрана директорія і стане кореневою для емулятора. Але такий підхід позбавляє емуляцію будь-якого сенсу, оскільки термінал не буде мати доступ до всієї системи, не кажучи вже про збірку та виконання коду.

Таким чином поки жоден із розглянутих засобів на операційній системі Android не відповідає нашим потребам.

Але в наступній частині буде розглянуто застосунок, що дозволить абсолютно вільно працювати із Android системою без всяких завад та перепон.

Використання Termux для написання та компіляції програм на ассемблері для ARM архітектури

Перед розглядом Termux маємо акцентувати увагу, що застосунок варто встановлювати не із Play Market, а з офіційного сайту (причиною є все таж політика безпеки).

Також Termux не вирішує проблему супер користувача для Android системи, тому для абсолютно повного контролю над системою вам потрібно буде отримати рут-доступ на ваш пристрій. Але для розглянутих далі задач із написання, компіляції та запуску коду - рут-доступ не потрібен.

Termux — це емулятор терміналу та середовище Linux для Android, яке забезпечує повнофункціональний інтерфейс командного рядка. Він дозволяє користувачам встановлювати та запускати пакети та програми Linux на своїх пристроях Android, надаючи потужне середовище розробки, яке можна використовувати для таких завдань, як програмування, веб-розробка та тестування на проникнення.

Однією з ключових особливостей Termux є його здатність отримувати доступ до апаратного забезпечення та системних ресурсів пристрою Android, що дозволяє користувачам запускати команди та сценарії, які взаємодіють із датчиками пристрою, камерою та іншими апаратними компонентами. Termux також підтримує низку мов програмування та фреймворків, включаючи Python, Ruby, Node.js і PHP, і може використовуватися для розробки та запуску мобільних програм на платформі Android.

Взаємодія між Termux і системою Android подібна до інших програм емулятора терміналу. Користувачі вводять команди в термінал, і Termux

надсилає ці команди в систему Android. Результат цих команд відображається в програмі емулятора терміналу Termux.

Однак Termux також надає додаткову функціональність завдяки своїй здатності встановлювати та запускати пакети та програми Linux. Це стало можливим завдяки системі керування пакетами, яка дозволяє користувачам завантажувати та встановлювати пакети програмного забезпечення з різних джерел, включаючи власний репозиторій Termux, а також сторонні репозиторії, такі як Debian і Ubuntu.

Загалом Termux надає потужне та гнучке середовище розробки для Android, яке дозволяє користувачам отримувати доступ і використовувати повний спектр інструментів і утиліт Linux на своїх мобільних пристроях.

Оскільки Termux має свій власний пакетний менеджер і дозволяє встановлювати нові пакети на Android систему, то за його допомогою можна скомпілювати абсолютно будь-який мову, якщо попередньо був встановлений компілятор.

Тобто можна скомпілювати і запустити будь-який із вищерозглянутих асемблерів, але за однієї умови – вони не будуть мати arm інструкцій.

Єдиний компілятор, який підтримує компіляцію і запуск інструкцій, що будуть підтримуватися процесором самої системи це вже встановлений GCC який використовує GAS.

Також Termux підтримує наступні додаткові можливості, які не доступні серед інших емуляторів терміналу:

1.Termux API яке надає можливості з отримання системної інформації про пристрій, а саме:

a.Отримати стан акумулятора пристрою.

b.Встановіть яскравість екрана від 0 до 255.

c.Список історії викликів.

d.Отримати інформацію про камери пристрою.

e.Зробіть фотографію та збережіть її у файл у форматі JPEG.

f.Отримати текст системного буфера обміну.

g.Встановити текст системного буфера обміну.

h.Список усіх контактів.

i.Показати діалогове вікно введення тексту.

j.Завантажити ресурс за допомогою системного менеджера завантажень.

k.Використовувати датчик відбитків пальців на пристрої, щоб перевірити автентифікацію.

l.Запланувати виконання сценарію Termux пізніше або періодично.

m.Отримати місцезнаходження пристрою.

n.Відтворення медіафайлів.

o.Інтерфейс MediaScanner, зробити зміни файлів видимими в Android Gallery

p.Запис за допомогою мікрофона на вашому пристрої.

q.Відображення системного сповіщення.

r.Отримайте інформацію про типи датчиків, а також поточні дані.

s.Поділитися файлом, указаним як аргумент, або текстом, отриманим на stdin.

t.Список SMS-повідомлень.

u.Надішліть SMS-повідомлення на вказані номери одержувачів.

v.Запитати файл із системи та виводити його у вказаний файл.

w.Телефонувати за номером телефону.

x.Отримати інформацію про всю спостережувану інформацію про комірки з усіх радіостанцій на пристрої, включаючи основні та сусідні комірки.

y.Отримати інформацію про пристрій телефонії.

z.Показати тимчасове спливаюче сповіщення.

aa.Увімкнути світлодіодний ліхтарик на пристрої.

bb.Отримайте інформацію про доступні механізми синтезу мовлення.

cc.Промовляйте текст за допомогою системи перетворення тексту в мовлення.

dd.Список пристроїв USB або доступ до них.

ee.Запустити вібрацію.

ff.Зміна гучності аудіопотоку.

gg.Змініть шпалер на пристрої.

hh.Отримати інформацію про поточне з'єднання Wi-Fi.

ii.Увімкнути/вимкнути Wi-Fi.

jj.Отримати інформацію про останнє сканування Wi-Fi.

Та багато іншого, що дозволить перетворити телефон на повноцінний пристрій або використовувати як передавач/датчик.

2.Запускати сценарій(ї) під час завантаження пристрою.

3.Termux підтримує майже всі інсталяції валідні для Ubuntu/Debian що дозволить запустити на телефоні сервер або віддалену базу даних або ж просто сканер мережі.

4.Підключення по ssh як до самого Android пристрою так із Android пристрою до віддалених клієнтів.

Таким чином Termux є потужним інструкметом для керування Adnroid системою, а також може використовуватися для написання та компіляції програм, які будуть мати доступ до внутрішніх ресурсів системи, як то мережа чи показання датчиків.

2.4 Порівняння способів та застосунків для написання низькорівневих програм для ARM архітектури

Написання програм на мові ассемблеру для ARM архітектури може бути здійснене різними способами, які мають свої переваги та недоліки. У цьому розділі ми порівняємо декілька різних способів написання програм на мові ассемблеру для ARM архітектури.

Написання програм на мові ассемблеру безпосередньо на Android-пристрої.

Цей спосіб передбачає написання коду ассемблера безпосередньо на Android-пристрої за допомогою текстового редактора або спеціальної інтегрованої середовища розробки (IDE) для ассемблера. Цей спосіб може бути зручним для коротких програм або для тестування алгоритмів. Однак, цей спосіб може бути обмеженим з точки зору можливостей редагування та відлагодження коду. А також серед всіх доступних застосунків для Android тільки 1 може використовуватися для запуску ассемблер коду безпосередньо на пристрої.

Написання програм на мові ассемблеру на ПК та перенесення на Android-пристрій.

Цей спосіб передбачає написання коду ассемблера на ПК за допомогою спеціальних програм для розробки ассемблерних програм, таких як FASMARM або Android NDK, а потім перенесення програми на Android-пристрій. Цей спосіб може бути зручним для написання складних програм, оскільки на ПК можна використовувати більш потужні інструменти розробки. Однак, цей спосіб вимагає додаткових зусиль для перенесення програми на Android-

пристрій. А також позбавляє сенсу розробку під конкретний процесор, адже використовує емуляцію середовища для запуску асемблер коду.

Використання інтерпретаторів ассемблерного коду.

Інтерпретатори ассемблерного коду, такі як QEMU, можуть бути використані для запуску ассемблерних програм на Android-пристрої. Цей спосіб може бути зручним для тестування ассемблерних програм або для навчання. Однак, емуляція не відповідає темі дослідження, оскільки не дає доступу до реальних системних ресурсів, а отже не дає можливості написання та відлагодження системного програмного забезпечення.

Переваги та недоліки кожного способу написання програм на ассемблері для ARM архітектури

Існує декілька способів написання програм на мові ассемблеру для ARM архітектури, кожен з яких має свої переваги та недоліки. Давайте розглянемо їх детальніше.

1.Написання програм на ассемблері безпосередньо на Android-пристрої. Цей підхід є найбільш прямолінійним та незалежним від інших програмних засобів. Він дозволяє розробникам бути повністю залежними від своїх вмінь і знань у написанні ассемблерного коду. Однак, цей підхід може бути складним для новачків, оскільки вимагає глибоких знань з ассемблеру та розуміння специфіки ARM архітектури та умінь роботи з компіляторами.

2.Використання спеціалізованих інструментів, таких як Dcoder, Code Editor, Assembler IDE, для написання та виконання ассемблерного коду на Android-пристрої. Ці інструменти дозволяють зробити процес написання коду більш простим та зручним, пропонуючи користувачам широкий спектр функцій та можливостей для створення програм за рахунок хмарного виконання коду, що переносить навантаження із системи на хмару, але такий підхід не

дозволить керувати файлами/процесами/станом пристрою/виконання довільної програми на самому пристрої, оскільки пристрій ніяк не пов’язаний із хмарою.

3.Використання компіляторів, таких як FASMARM, для збірки ассемблерного коду на комп'ютері та перенесення скомпільованого коду на Android-пристрій. Цей підхід дозволяє розробникам використовувати різні інструменти, які можуть полегшити процес написання коду, такі як текстові редактори та інші інтегровані середовища розробки. Однак, цей підхід може бути складним для новачків, оскільки вимагає додаткових знань з компіляції та на ARM архітектурі та налаштування середовища розробки для компіляції коду. Потрібно абсолютно точно виставити налаштування емуляції та збірки на комп’ютері, щоб виконання виконуючого файлу було можливе на android пристрої. Також, збірка та перенесення скомпільованого коду можуть зайняти більше часу, ніж написання та виконання просто навчального або тестового коду безпосередньо на Android-пристрої.

Кожен з цих підходів має свої переваги та недоліки, тому вибір підходу залежить від потреб розробника та його рівня вмінь та знань з ассемблеру та ARM архітектури. Для простих та тестових програм, які не потребують взаємодії із системою, її датчиками та файловою системою пристрою – рекомендується використовувати спеціалізовані інструменти, такі як Dcoder та Code Editor, для того, щоб полегшити процес написання та виконання коду.

Для всіх інших випадків підіде написання ассемблер коду та його компіляція безпосередньо на Android пристрої за допомогою Termux.

Відповідність способу написання програм конкретній задачі

Termux дозволяє розробляти різноманітне системне програмне забезпечення для використання в Android системі. Ось деякі приклади таких програм:

1.Консольні програми: Термінал Termux дає можливість написання різноманітних консольних програм, за допомогою всеможливих мов

програмування і навіть асемблеру відповідно до архітектури системи. Таким чином можуть бути реалізовані безпосередньо на телефоні утиліти для обробки текстових файлів, інструменти для моніторингу системи і т.д.

2.Системні утиліти: Термінал Termux може бути використаний для розробки системних утиліт, таких як програми для роботи з файловою системою, керування процесами, моніторингу мережі, роботи з Bluetooth і т.д. Звернемо увагу, що за замовчуванням без root-доступу Termux може керувати лише своїми процесами, тобто якщо запустити веб сервер – він буде запущений із Termux і відповідно процес буде мати тіж самі права доступу, що й батьківський, тобто вікно Termux. Таким чином із Termux без root-доступу можна керувати лише власними процесами, для доступу до системних процесів, повного стану мережі та повного доступу до вбудованих модулів і передачі інформації по Bluetooth, тощо – необхідно встановити root для Android пристрою.

3.Мережеві програми: В Termux можна запускати та реалізовувати різноманітні мережеві програми, такі як сервери і клієнти для різних протоколів мережі, утиліти для моніторингу мережі і т.д. Для деяких все так само потрібен root.

4.Автоматизація задач: Termux може бути використаний для написання скриптів та інструментів для автоматизації різних задач на Android пристрої, наприклад, для автоматичного копіювання даних з пам'яті телефону на карти пам'яті, резервного копіювання даних, роботи зі звуком і т.д.

В загальному, Termux надає широкі можливості для розробки різноманітних системних програм та інструментів, які можуть бути використані для вирішення різних завдань на пристроях, що мають в основі ARM процесори, такі як Android пристрої чи мікроконтролери типу Raspberry.

2.5 Висновки до розділу

В цьому розділі ми дослідили різні способи написання програм на мові асемблеру для ARM архітектури на платформі Android. Ми розглянули такі підходи, як написання програм безпосередньо на Android-пристрої, використання спеціалізованих інструментів та компіляцію коду на комп'ютері.

Кожен з цих підходів має свої переваги та недоліки. Написання програм безпосередньо на Android-пристрої є найбільш прямолінійним та незалежним від інших програмних засобів, але вимагає глибоких знань з асемблеру та розуміння специфіки ARM архітектури. Використання спеціалізованих інструментів дозволяє зробити процес написання коду більш простим та зручним, пропонуючи широкий спектр функцій та можливостей для створення програм. Але деякі з цих інструментів можуть бути обмеженими в плані функціональності та можливості оптимізації коду. Компіляція коду на комп'ютері дозволяє використовувати різні інструменти, які можуть полегшити процес написання коду, але вимагає додаткових знань з компіляції та перенесення коду на Android-пристрій.

Зважаючи на переваги та недоліки кожного з цих підходів, в наступному розділі буде продемонстровано процес написання, компіляції асемблер коду безпосередньо на android пристрої за допомогою емулятора терміналу Termux.

РОЗДІЛ 3

РОЗРОБКА НИЗЬКОРІВНЕВОЇ ПРОГРАМИ В СЕРЕДОВИЩІ ANDROID

3.1 Компіляція Assembler програм для ARM64 архітектури Android

Огляд процесу компіляції програм на мові ассемблера в Termux

Перед початком написання програм за допомогою асемблеру в Termux необхідно встановити останню версію з офіційного сайту та надати йому доступ до спільного сховища за допомогою :

termux-setup-storage

Під час виконання подальших кроків в залежності від версії Android системи та самого Termux потрібно буде встановити додаткові linux утиліти, такі як архіватори, мережеві утиліти, тощо. Всі ці маніпуляції легко виконати за допомогою підказок, які буде пропонувати сам Termux.

Проте варто зазначити, що таким чином потрібно мати запас внутрішнього сховища приблизно до 500 мб. в залежності від цілей.

Далі необхідно визначити архітектуру системи Android. Це можна зробити за допомогою будь-якої утиліти із Play Market, такі як Inware або CPU-z, тощо.

Для Inware потрібно звернути увагу на поле «Suppoerted ABIS» в меню «Hardware».

Для CPU-z у вкладці «Система» поле «Kernel Arhitecture»

Також, бажано варто було б дізнатися які набори інструкції підтримуються процесором обраного смартфону, це найлегше знайти в застосунку Inware у полі «Suppoerted ABIS» в меню «Hardware».

З інформацією про апаратним забезпеченням та набором інструкцій для процесору можна буде з легкістю знайти необхідну офіційну arm документацію. Відповідність набору інструкцій до ABI наведено в таблиці 6

Таблиця 6

ABI	Набір інструкцій, який підтримується	Опис
armeabi-v7a	armeabi Thumb-2 VFPv3-D16	Несумісний з пристроями ARMv5/v6.
arm64-v8a	AArch64	Тільки Armv8.0.
x86	x86 (IA-32) MMX SSE/2/3 SSSE3	Немає підтримки MOVBE або SSE4.
x86_64	x86-64 MMX SSE/2/3 SSSE3 SSE4.1, 4.2 POPCNT	x86-64-v1 тільки.

Після того, як буде визначено розрядність процесора та архітектуру системи 32/64, а також набір інструкцій, що підтримуються системою - можна приступити до безпосередньої розробки в емуляторі терміналу Termux.

В нашому випадку ядро підтримує: arm64-v8a та armeabi-v7a і відповідно має 64 бітну архітекруту.

Розгляд основних проблем, що можуть виникнути при компіляції програм для ARM64 архітектури

В залежності від розрядності ядра та набору інструкцій, що подтримуються -- синтаксис ARM асемблеру буде суттєво відрізнятися.

ARM32 та ARM64 є різними архітектурами процесорів, тому синтаксис мови асемблера для них теж різний. Основні відмінності між синтаксисами ARM32 та ARM64 наступні:

1.Розмір регістрів: в ARM32 розмір регістрів 32 біти, тоді як у ARM64 вони мають розмір 64 біти.

2.Назви регістрів: у ARM32 регістри мають назви від r0 до r15, тоді як у ARM64 вони мають назви від x0 до x30, причому x30 використовується як регістр-адрес повернення.

3.Інструкції: у ARM32 використовуються 32-бітні інструкції, тоді як у ARM64 вони мають довжину 32 або 64 біти. Крім того, деякі інструкції можуть мати різні назви та аргументи у ARM32 та ARM64.

4.Використання регістрів для передачі параметрів: у ARM32 перші чотири параметри передаються через регістри r0-r3, тоді як у ARM64 перші вісім параметрів передаються через регістри x0-x7.

5.Режими роботи процесора: ARM32 підтримує три режими роботи процесора (user, system та supervisor), тоді як ARM64 підтримує вісім режимів роботи процесора (EL0-EL7).

6.Підтримка інструкцій: ARM64 підтримує більш широкий набір інструкцій, включаючи інструкції з розширеними можливостями SIMD та криптографічні інструкції.

Ці відмінності можуть вплинути на написання та компіляцію програм на мові асемблера для ARM32 та ARM64. Тому, якщо при переході до розробки для ARM32 до ARM64, необхідно звернути увагу на ці відмінності та адаптувати свій код до відповідного синтаксису. В іншому випадку обов’язково виникнуть проблеми на моменті компіляції вашого асемблер файлу.

Опис засобів для компіляції програм на мові ассемблера для ARM64 архітектури в Termux

Будь-яка Android система з самого початку вже має вбудований асемблер та компілятор за допомогою якого можна компілювати асемблер код.

Проте з офіційного сайту ARM можна завантажити будь-який GNU toolchain, який підтримується набором інструкцій пристрою.

Наприклад такі такі пакети необхідні для компіляції та генерації виконуючого коду можна встановити під будь-яку операційну систему.

Для Windows систем можна встановити .exe компілятори які можна зпускати із cmd.

Для Android достатньо просто завантажити відповідний пакет (наприклад з розширенням .tar) через wget а потім встановити за допомогою вбудованих і вже доступних засобів наявних в Termux.

При чому варто зазначити, що для жодного з описаних кроків права root доступу не потрібні, що значно полегшує розробку. Але варто звертати увагу на набір інструкцій, які підтримує ваш процесор та компілятор, який ви плануєте встановити.

Таким чином знаючи архітектуру системи можна відразу з самого початку перейти до написання і компіляції коду на асемблері для конкретної архітектури без всяких проміжних кроків. Абож можна встановити компілятор під цільову архітектуру з офіційного сайту виробника і вже потім перейти до розробки.

3.2 Особливості розробленої програми

Для демонстрації можливості компіляції та виконання асемблер програм за допомогою Termux в операційній системі Android було реалізовано невелику демонстраційну програму.

В даній програмі реалізовано роботу з функціями, стеком та викликом функцій із бібліотеки libc в arm64.

Для демонстраційної програми було вибрано обчислення наступного виразу:

3f(x-1,y+1)+6f(x,y)+f(x,2y)

Де f – функція двох аргументів:

f(x,y)=2x+3y-5

```
.global f
.type f, %function

f:
 // зберігаємо значення регістрів x0 та x1 на стеку
 sub sp, sp, #16
 str x0, [sp, #8]
 str x1, [sp, #0]

 // виконуємо обчислення
 add x0, x0, x0  // 2x
 add x0, x0, x1  // 2x + y
 add x0, x0, x1  // 2x + 2y
 add x0, x0, x1  // 2x + 3y
 sub x0, x0, #5  // 2x + 3y - 5
 mov x4,x0        // записуємо в регістр для повернення результату із функції

 // відновлюємо значення регістрів x0 та x1 зі стеку
 ldr x1, [sp, #0]
 ldr x0, [sp, #8]
 add sp, sp, #16
 ret
```

Рисунок 3.1 код функції реалізованої за допомогою асемблеру

Для позначення функції потрібно визначити її назву як глобальну змінну, а також встановити тип через .type.

Сам процес написання функції майже ніяк не відрізняється від інших асемблерів. Для повернення із функції використовується мнемоніка ret.

Для виклику функції та повернення з неї використовується регістр LR(link register), він же X30.

Для використання регістру LR застосовується інструкція переходу BL (branch with link), яка виконує перехід і поміщає адресу наступної інструкції в

регістр LR. Завдяки чому після завершення функції виконання програми перейде до інструкції, яка слідує за викликом функції.

Однак тут є ще одна проблема – необхідно повернутися з функції. Для повернення з функції використовується інструкція RET (return). Фактично, ця інструкція виконує перехід до тієї адреси, яка збережена в регістрі LR. Більше того, визначення функції має завершуватися цією інструкцією.

Також в реалізованій функції показано приклад роботи зі стеком.

Коли Linux запускає програму, цій програмі виділяється стек розміром 8 мегабайт. Для роботи зі стеком у програмі на асемблері застосовується регістр X31. Цей регістр має спеціальне призначення: він може виступати як нульовий регістр і як покажчик стека (SP). Якщо інструкція не працює зі стеком, вона розглядає цей регістр як нульовий регістр.

ARM-процесор вимагає, щоб регістр SP завжди мав вирівнювання в 16 байт. Це означає, що ми можемо додавати або віднімати з SP тільки ті дані, які є кратними 16. В іншому випадку ми зіткнемося з помилкою.

Таким чином ми спочатку виділяємо в стеку 16 байт та зберігаємо туди регістри X0 та X1 які в нашому прикладі використовуються, як аргументи x,y для функції відповідно.

```
// зберігаємо значення регістрів x0 та x1 на стеку
sub sp, sp, #16
str x0, [sp, #8]
str x1, [sp, #0]
```

Рисунок 3.2 команди для роботи з регістром

Так само, перед вихідом із функції(після збереження результату функції в регістр повернення) ви відновлюємо із стеку значення регістрів та обов’язково маємо вирівняти стек

```
// відновлюємо значення регістрів x0 та x1 зі стеку
ldr x1, [sp, #0]
ldr x0, [sp, #8]
add sp, sp, #16
```

Рисунок 3.3 команди для роботи з регістром

Точка входу в програму та використання функцій із бібліотеки libc.

Зазвичай, точкою входу а програму ARM асемблеру є фунція, яка визначається назвою «_start».

Але в нашому випадку ця функція має назву «main».

Причина цього полягає в використання функцій із бібліотеки libc, яка має свою точку входу _start, та підключається автоматично при компіляції(можна і обійтися).

Arm асмеблери вміють виводити на потік вводу-виводу тільки ascii символи. В такому випадку константний та сформований текст виводиться без проблем. Але вивест числа такий метод не довзоляє. З першого погляду це унеможливлює виведення чисел нашому асмеблер програмою, але це не так і нище ми розглянемо декілька варіантів вирішення даної проблеми.

Найперше рішення, яке спадає на думку – реалізувати свою функцію переведення чисел в набір ascii символів що було б не дуже доречно, оскільки в стандартній бібліотеці C вже є функція, яка може виводити і текст і числа (printf)

Другий варіант – це використовувати якраз функції із бібліотеки libc.

Третій варіант, який також має право на життя це запис числа у спецільний 32 бітний регістр, який використовується як регістр куди записується код помилки. Зазвичай туди поміщається 0 як показник нормального завершення програми, але якщо виникає помилка – її код поміщають саме в цей регістр.

Таким чином можна перед виходом із програми помістити в цей регістр будь-яке число, яке вміщається в X0/W0 і отримати код «помилки», який повертає програма.

До демонстрації двох останній методів виводу результату програми ми повернемося пізніше.

Отже точкою входу в програму буде функція «main» яка визначається просто як глобальна і не потребує вказання типу через інструкцію «.type»

Як вже було описано вище, для виклику функцій в arm64 асемблері використовується інструкція «bl»

```
sub x0, x0, #1 // x - 1
add x1, x1, #1 // y + 1
bl f
mov x5, x4      // записуємо в регістр результату значення, що повертає функція
add x5, x5, x4  // 2f(x-1,y+1)
add x5, x5, x4  // 3f(x-1,y+1)
```

Рисунок 3.4 виклик функції за допомогою bl

Виведення результату на консоль. Для цього в першому випадку використаємо виклик функції «printf». Ця функція бере данні для виводу із регістру X1, тому спершу поміщаємо в нього результат роботи нашої програми.

```
// передаємо результат у регістр x0 для виведення на екран
mov x1, x5
adr x0, .LC0
bl printf
```

Рисунок 3.5 виклик функції printf

Для виведення даних функція потребує формат-рядка. Адреса рядка поміщається в регістр X0. Таким чином функція «printf» бере свої параметри із перших двох регістрів так само, як реалізована нами функція.

Такий формат-рядок зберігається в змінній LC0, яка являє собою набір ascii символів з символом форматування «%ld» -- цей означає, що число яке буде виводитися буде «long decimal». Перед створення змінної було виконане вирівнювання від початку секції 2^2.

Такі змінні створюються в секцій «.data».

Інший спосіб виведення числа на консоль ми вже описували вище – це повернення числа як коду помилки. Для цього просто помістити значення в регістр X0/W0 і завершити програму.

```
// Завершуємо програму
mov X0, X5       // поміщаємо результуюче значення в регістр з кодом помилки
MOV X8, #93      // встановлюємо функцію Linux для виходу із програми
SVC 0           // викликаємо супервізор
```

Рисунок 3.6 Вихід із програми

Тепер наша програма нічого не виводить на консоль і щоб отримати бажане число нам потрібно отримати код помилки який повертає наша програма.

Зробити це можна за допомогою команди Linux «echo $?» після запуску виконуючого файлу, або в тому самому рядку запуску програми через

«./executable_file_name ; echo $?»

Компіляція та запуск виконуючого файлу

Для компіляції програм, які використовують бібліотеки як от наприклад libc компілювати потрібно за допомогою вже наявного компілятора gcc, який викличе асемблер та лінковщик і створить виконуючий файл.

Якщо було встановлені інші компілятори описані в попередній частині 3 розділу, то варто додати шлях до нього в SYSPATH і далі спокійно його використовувати.

Можна використовувати як повну назву «aarch64-linux-android-gcc» так і просто скорочення «gcc»

Команда для компіляції:

aarch64-linux-android-gcc -o main main.s

```
~/storage/diplom $ ls
main.s  test  test.s
~/storage/diplom $ aarch64-linux-android-gcc -o main main.s
~/storage/diplom $ ls
main  main.s  test  test.s
~/storage/diplom $
```

Рисунок 3.7 Приклад компіляції

Для компіляції чистого асмеблер коду без сторонніх залежностей можна таксамо використовувати gcc або власноруч викликати асемблер збірку а потім лінковку.

Це можна зробити за допомогою наступних команд:

as simple.s -o simple.o

ld simple.o -o simple

Запуск програм

1.Запуск програми, яка використовує функцію printf

```
~/storage/diplom $ ./main
165
~/storage/diplom $
```

Рисунок 3.8 Приклад виводу чисел за допомогою функції printf

2.Запуск програми, яка повертає значення як код помилки в регістр X0/W0

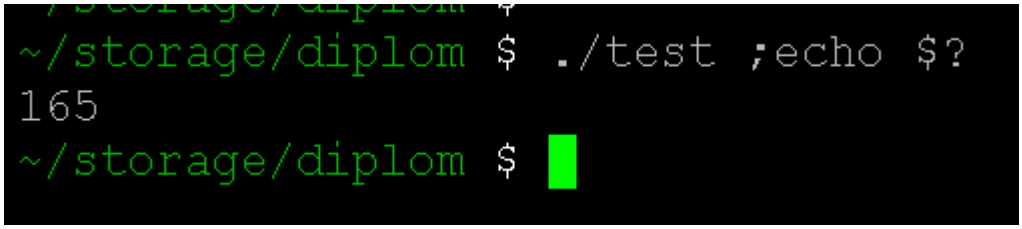

Рисунок 3.9 Приклад виводу чисел за допомогою поміщення в регістр

В прикладі який розглянутий нище аргументи мають наступні значення:

X = 4

Y = 4

Результат обчислення 165, що збігаєтсья з тим, що вивела наша програма.

Перед тим, як перейти до розділу в якому буде розглянуто відлагодження програми, зазначемо, що в даній програмі не використовувалися такі інструкції як множення чи ділення, тощо для покрокової демонстрації обчислення результату в програмі під час відлагодження.

3.3 Відлагодження Assembler програм

Розглянемо декілька популярних дебагерів.

Для того, щоб можна було під’єднати вбудовані дебагери, компіляція програм має відбуватися з прапором «-g».

GDB (GNU Debugger): Це популярний і потужний налагоджувач, який може налагоджувати C, C++ та інші мови. Він підтримує як локальне, так і віддалене налагодження та може використовуватися з багатьма різними середовищами розробки.

LLDB: Це налагоджувач, який є частиною проекту LLVM і може налагоджувати код C, C++, Objective-C та інші. Він розроблений як розширюваний і має архітектуру плагінів.

Valgrind: Це інструмент, який може виявляти витоки пам’яті та інші пов’язані з пам’яттю помилки в програмах. Він також може профілювати код і допомогти виявити вузькі місця продуктивності.

DDD (Data Display Debugger): Це графічний інтерфейс для GDB, який забезпечує більш зручний інтерфейс для налагодження.

Nemiver: Це ще один графічний налагоджувач, який підтримує код C і C++. Він має простий та інтуїтивно зрозумілий інтерфейс і може використовуватися з різними IDE.

Серед розглянутих останні два є просто графічними покращеннями дебагерів gdb та lldb. Дебагер Valgrind є більш вузьконаправленим тому для нашого відлагодження будемо використовувати перші два.

Для встановлення gdb та lldb варто скористатися наступними командами:

pkg install gdb

pkg install lldb

Використання gdb для відлагодження assembler програм для ARM64 архітектури

Після компілювання програми з мітками для відлагодження можна перейтини безпосередньо до дебагу.

1. Підключаємось до дебагу файлу за допомогою «gdb main.py»

2. Перед початком відлагодження можна подивитися код за допомогою «list n,m» де n рядок з якого почати m – рядок до якого дивитися:

```
(gdb) list 25,35
25      .globl main
26
27      main:
28        // передаємо параметри у регістри x0 та x1
29        mov x0, #4 // x
30        mov x1, #4 // y
31        mov x5, #0 // res = 0
32
33        sub x0, x0, #1 // x - 1
34        add x1, x1, #1 // y + 1
35        bl f
(gdb)
```

Рисунок 3.10 Лістинг коду програми з 25 по 35 рядки

3. Далі встановлюємо точки зупину по номеру рядку, або по назві мітки:

```
(gdb) b main
Breakpoint 1 at 0x16f0: file main.s, line 29.
(gdb)
```

Рисунок 3.11 Встановлення точки зупину

4. Після того, як всі точки зупину виставлені можна перейти до самого дебагу за допомогою «run» і програма буде виконуватися до першої встановленої точки:

```
(gdb) run
Starting program: /data/data/com.termux/files/home/storage/diplom/main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/data/data/com.termux/files/usr/lib/libthread_db.so".

Breakpoint 1, main () at main.s:29
29        mov x0, #4 // x
(gdb)
```

Рисунок 3.12 Запуск програми і зупинка на першій точці зупину

5. Перед тим як ми продовжимо відлагодження програми і покищо знаходимося в стартовій точці варто переглянути регістри за допомогою «info

register». Початковий стан регістрів:

```
(gdb) info register
x0             0x1                 1
x1             0x7ffffff148        549755810120
x2             0x7ffffff158        549755810136
x3             0x7ffffff118        549755810072
x4             0x7fbe4e0080        548653629568
x5             0x7fbe464a4a        548653124170
x6             0x34                52
x7             0x7fbe4e00c0        548653629632
x8             0x0                 0
x9             0x2                 2
x10            0x1                 1
x11            0x0                 0
x12            0x0                 0
x13            0x0                 0
x14            0x7                 7
x15            0xffffffffffffffff  -1
x16            0x7fbdfffaa8        548648516264
x17            0x7fbf637934        548671813940
x18            0x7fbf4d4000        548670357504
x19            0x55555566f0        366503880432
x20            0x7ffffff148        549755810120
x21            0x7ffffff158        549755810136
x22            0x1                 1
x23            0x0                 0
x24            0x0                 0
x25            0x0                 0
x26            0x0                 0
x27            0x0                 0
x28            0x0                 0
x29            0x7ffffff0e0        549755810016
x30            0x7fbdf8d070        548648046704
sp             0x7ffffff0e0        0x7ffffff0e0
pc             0x55555566f0        0x55555566f0 <main>
cpsr           0x80000000          [ EL=0 BTYPE=0 N ]
fpsr           0x0                 [ ]
fpcr           0x0                 [ Len=0 Stride=0 RMode=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
tpidr          0x7fbe469008        0x7fbe469008
tpidr2         0x0                 0x0
```

Рисунок 3.13 значення регістрів в точці входу програми

6. Для переходу на наступну інструкцію (і відповідно виконання поточної) використовується команда next. На рисунку нище видно стан регістрів X0 та X1

після завантаження в них значень x та y

```
(gdb) next
30          mov x1, #4 // y
(gdb)
31          mov x5, #0 // res = 0
(gdb)
33          sub x0, x0, #1 // x - 1
(gdb) info register x0 x1 x4 x5
x0             0x4                4
x1             0x4                4
x4             0x7fbe4e0080       548653629568
x5             0x0                0
```

Рисунок 3.14 Значення регістрів після завантаження параметрів

7. При послідовному виконанні інструкцій перехід в функцію, яка викликається – не відбувається, тому за наступний крок відбувається виконання функції і відбувається перехід на наступну інструкцію. В регістрі X4 можемо бачити результат виконання функції:

```
35          bl f
(gdb)
36          mov x5, x4      // записуємо в регістр результату значення, що повертає функція
(gdb) info register x0 x1 x4 x5
x0             0x3                3
x1             0x5                5
x4             0x10               16
x5             0x0                0
(gdb)
```

Рисунок 3.15 Результат виконання функції в регістрі X4

8. Таким чином можна покроково відлагоджувати програму. Якщо необхідно завершити виконання програми можна скористати командою «continue»

Використання lldb для відлагодження assembler програм для ARM64 архітектури

Для відлагодження програми за допомогою lldb потрібно так само скомпілювати програму з мітками відлагодження після чого можна перейти до дебагу:

1. Підключаємося за допомогою «lldb ./main» а далі схема роботи схожа із gdb

2. Для встановлення точки зупину потрібна команда «breakpoint», а для точки зупину по імені «breakpoint set -n main»

```
(lldb) breakpoint set -n main
Breakpoint 1: where = main`main + 4, address = 0x00000000000016f4
(lldb)
```

Рисунок 3.16 Встановлення точки зупину

3. Після чого можна запустити виконання до точки зупину за допомогою «process launch» і чекаємо поки виконання не дійде до першої точки:

```
(lldb) process launch
Process 25600 launched: '/data/data/com.termux/files/home/storage/diplom/main' (aarch64)
Process 25600 stopped
* thread #1, name = 'main', stop reason = breakpoint 1.1
    frame #0: 0x00000055555566f4 main`main at main.s:30
   27   main:
   28     // передаємо параметри у регістри x0 та x1
   29     mov x0, #4 // x
-> 30     mov x1, #4 // y
   31     mov x5, #0 // res = 0
   32
   33     sub x0, x0, #1 // x - 1
warning: This version of LLDB has no plugin for the language "assembler". Inspection of frame variables will be limited.
(lldb)
```

Рисунок 3.17 Зупинка виконання на першій точці зупину

4. Перевіряємо стан регістрів за допомогою «register read -all» або «register read x0 x1 x4 x5»

```
(lldb) register read -all
General Purpose Registers:
        x0 = 0x0000000000000004
        x1 = 0x0000007ffffff168
        x2 = 0x0000007ffffff178
        x3 = 0x0000007ffffff138
        x4 = 0x0000007fbe4e0080
        x5 = 0x0000007fbe464a4a
        x6 = 0x0000000000000034
        x7 = 0x0000007fbe4e00c0
        x8 = 0x0000000000000000
        x9 = 0x0000000000000002
       x10 = 0x0000000000000001
       x11 = 0x0000000000000000
       x12 = 0x000000000000000a
       x13 = 0x0000000000000000
       x14 = 0x09bf19f9b0b90aed
       x15 = 0xffffffffffffffff
       x16 = 0x0000007fbe01eaa8
       x17 = 0x0000007fbf637934
       x18 = 0x0000007fbf0c6000
       x19 = 0x00000055555566f0  main`main
       x20 = 0x0000007ffffff168
       x21 = 0x0000007ffffff178
       x22 = 0x0000000000000001
       x23 = 0x0000000000000000
       x24 = 0x0000000000000000
       x25 = 0x0000000000000000
       x26 = 0x0000000000000000
       x27 = 0x0000000000000000
       x28 = 0x0000000000000000
        fp = 0x0000007ffffff100
        lr = 0x0000007fbdfac070  libc.so`__libc_init + 112
        sp = 0x0000007ffffff100
        pc = 0x00000055555566f4  main`main + 4
      cpsr = 0x80000000
        w0 = 0x00000004
```

Рисунок 3.18 Стан регістрів при вході в програму

5. Для виконання поточної інструкції та переходу до наступної необхідна команда «thread step-over» Як можемо бачити із рисунка нище в регістрах x0 та

x1 знаходяться значення наших параметрів (4 та 4 відповідно)

```
(lldb) thread step-over
Process 25600 stopped
* thread #1, name = 'main', stop reason = step over
    frame #0: 0x00000055555566f8 main`main at main.s:31
   28     // передаємо параметри у регістри x0 та x1
   29     mov x0, #4 // x
   30     mov x1, #4 // y
-> 31     mov x5, #0 // res = 0
   32
   33     sub x0, x0, #1 // x - 1
   34     add x1, x1, #1 // y + 1
(lldb)
Process 25600 stopped
* thread #1, name = 'main', stop reason = step over
    frame #0: 0x00000055555566fc main`main at main.s:33
   30     mov x1, #4 // y
   31     mov x5, #0 // res = 0
   32
-> 33     sub x0, x0, #1 // x - 1
   34     add x1, x1, #1 // y + 1
   35     bl f
   36     mov x5, x4      // записуємо в регістр результату значення, що повертає функція
(lldb) register read x0 x1 x4 x5
      x0 = 0x0000000000000004
      x1 = 0x0000000000000004
      x4 = 0x0000007fbe4e0080
      x5 = 0x0000000000000000
(lldb)
```

Рисунок 3.19 Завантаження параметрів в регістри

6. Дійдемо до моменту виклику функції та зробимо перехід

```
(lldb) thread step-over
Process 25600 stopped
* thread #1, name = 'main', stop reason = step over
    frame #0: 0x0000005555556704 main`main at main.s:35
   32
   33     sub x0, x0, #1 // x - 1
   34     add x1, x1, #1 // y + 1
-> 35     bl f
   36     mov x5, x4      // записуємо в регістр результа
   37     add x5, x5, x4  // 2f(x-1,y+1)
   38     add x5, x5, x4  // 3f(x-1,y+1)
(lldb) register read x0 x1 x4 x5
      x0 = 0x0000000000000003
      x1 = 0x0000000000000005
      x4 = 0x0000007fbe4e0080
      x5 = 0x0000000000000000
```

Рисунок 3.20 Виклик функції

7. Як можемо бачити виконання перекидає нас прямо в нашу функцію що звісно може слугувати великою перевагою на відміну від gdb

```
(lldb) thread step-over
Process 25600 stopped
* thread #1, name = 'main', stop reason = step over
    frame #0: 0x00000055555566bc main`f at main.s:7
   4
   5    f:
   6      // зберігаємо значення регістрів x0 та x1 на стеку
-> 7      sub sp, sp, #16
   8      str x0, [sp, #8]
   9      str x1, [sp, #0]
   10
(lldb)
```

Рисунок 3.21 Перехід в функцію

8. Дійдемо до моменту виклику функції printf із сторонньої бібліотеки. В x0 адреса для формат-рядку, а в x1 данні, які ми будемо виводити

```
Process 25600 stopped
* thread #1, name = 'main', stop reason = step over
    frame #0: 0x000000555555674c main`main at main.s:58
   55     // передаємо результат у регістр x0 для виведення на екран
   56     mov x1, x5
   57     adr x0, .LC0
-> 58     bl printf
   59
   60     // Завершуємо програму
   61     mov w0, #0
(lldb) register read x0 x1 x4 x5
      x0 = 0x0000005555558a28
      x1 = 0x00000000000000a5
      x4 = 0x000000000000001b
      x5 = 0x00000000000000a5
(lldb)
```

Рисунок 3.22 Завантаження параметрів в регістри для фунцій printf

10. При наступному кроці викличиться функція printf без заходу в неї. Але зазначимо, що останні 2 команди викликають вихід із програми, тому були

автоматично пропущені

```
Process 25600 stopped
* thread #1, name = 'main', stop reason = step over
    frame #0: 0x000000555555674c main`main at main.s:58
   55      // передаємо результат у регістр x0 для виведення на екран
   56      mov x1, x5
   57      adr x0, .LC0
-> 58      bl printf
   59
   60      // Завершуємо програму
   61      mov w0, #0
(lldb) register read x0 x1 x4 x5
      x0 = 0x0000005555558a28
      x1 = 0x00000000000000a5
      x4 = 0x000000000000001b
      x5 = 0x00000000000000a5
(lldb) thread step-over
165
Process 25600 exited with status = 0 (0x00000000)
(lldb)
```

Рисунок 3.23 Значення регістрів-параметрів для виклику сторонньої функції

3.4 Висновки до розділу

У цьому розділі було розглянуто базові концепції та інструменти для програмування на мові асемблера для архітектури ARM64 на платформі Android. Ми розглянули різноманітні способи компіляції програм, відлагодження та оптимізації коду.

Також було розглянуто деякі відмінності між синтаксисом ARM32 та ARM64, описали набір інструкцій AArch64 та розглянули різні засоби для компіляції програм на мові асемблера для ARM64 архітектури в Termux. Також, було детально розглянуто використання інструментів відлагодження, таких як gdb та lldb.

Як можна було зрозуміти процес компіляції та відлагодження програм на мові асемблера для ARM архітектури за допомогою Termux на Android смартфоні є досить легким та швидким.

ВИСНОВКИ

В ході виконання дипломної роботи було проведено дослідження можливостей використання Termux на Android-пристроях для розробки та запуску низкорівневого програмного забезпечення.

В першому розділі було проведено огляд архітектур процесорів та вибрано актуальну та перспективну ARM архітектуру для подальшого написання низкорівневого програмого забезпечення.

Були проаналізовані можливості встановлення та налаштування необхідних інструментів, таких як компілятори та редактори коду, і описано процеси розробки програм на мові асемблеру для ARM архітектури.

Було розглянуто переваги та недоліки різних підходів до написання програм на мові асемблеру для ARM архітектури на Android-пристроях.

Було визначено, що Termux є потужним інструментом для розробки та виконання низкорівневого програмного забезпечення на Android-пристроях. Він надає можливість встановлення та використання широкого спектру інструментів та бібліотек, що дозволяє створювати програми для різних цілей, в тому числі й для розробки системного програмого забезпечення .

Отже, в результаті проведеного дослідження було встановлено, що використання Termux на Android-пристроях може бути вигідним для розробки та запуску системного програмного забезпечення, а також додатків, що вимагають високої продуктивності та оптимізації.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 [Електронний ресурс]. Режим доступу: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html?wapkw=%20Architectures%20Software%20Developer%20Manuals
2. Learn the architecture - Introducing the Arm architecture [Електронний ресурс]. Режим доступу: https://developer.arm.com/documentation/102404/0201/?lang=en
3. Learn the architecture - A64 Instruction Set Architecture [Електронний ресурс]. Режим доступу: https://developer.arm.com/documentation/102374/0101
4. Introduction to Assembly Language Programming [Електронний ресурс]. Режим доступу: https://www.tutorialspoint.com/assembly_programming/index.htm
5. How Microprocessors Work [Електронний ресурс]. Режим доступу: https://computer.howstuffworks.com/microprocessor.htm
6. Linux Assembly [Електронний ресурс]. Режим доступу: https://asm.sourceforge.net/
7. PROGRAMMING FROM THE GROUND UP [Електронний ресурс]. Режим доступу: http://programminggroundup.blogspot.com/2007/01/chapter-2-computer-architecture.html
8. Termux official documentation [Електронний ресурс]. Режим доступу: https://termux.com/
9. Android NDK DOCUMENTATION [Електронний ресурс]. Режим доступу: https://developer.android.com/ndk
10. ARM System Developer's Guide Designing and Optimizing System Software Andrew Sloss, Dominic Symes, Chris Wright [Електронний ресурс]. Режим доступу: https://www.perlego.com/book/1810182/arm-system-developers-guide-designing-and-optimizing-system-software-pdf
11. Introduction to ARM Assembly Language [Електронний ресурс]. Режим доступу: https://azeria-labs.com/writing-arm-assembly-part-1/
12. GDB documentation [Електронний ресурс]. Режим доступу: https://www.sourceware.org/gdb/documentation/
13. LLDB documentation [Електронний ресурс]. Режим доступу: https://lldb.llvm.org/
14. ARMv8-A Assembly Language Programming [Електронний ресурс]. Режим доступу: https://developer.arm.com/documentation/dui0801/f/

15. NASM documentation [Електронний ресурс]. Режим доступу: https://www.nasm.us/docs.php
16. MASM documentation [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference?view=msvc-170
17. FASM documentation [Електронний ресурс]. Режим доступу: https://flatassembler.net/docs.php
18. GAS documentation [Електронний ресурс]. Режим доступу: https://sourceware.org/binutils/docs-2.37/as/
19. The Yasm Modular Assembler Project [Електронний ресурс]. Режим доступу: http://yasm.tortall.net/