БАКАЛАВРСЬКА РОБОТА

БР. ІП - 22.00.00.000 ПЗ

Група ІП-21-4

Нейлюк Андрій

2025

Івано-Франківський національний технічний університет нафти і газу
Інститут інформаційних технологій
Кафедра інженерії програмного забезпечення

Нейлюк Андрій Васильович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Програмування систем реального часу

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Нейлюк А. В._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Тимків Дмитро Федорович, д.т.н., проф._____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____202 р.

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ БАКАЛАВРА СТУДЕНТОВІ

Нейлюк Андрій Васильович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Програмування систем реального часу”.

керівник проекту (роботи) д.т.н., проф. Тимків Дмитро Федорович

затверджені наказом вищого навчального закладу від **“** 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1 Основи систем реального часу

2. Апаратне забезпечення для систем реального часу

3. Мови програмування для систем реального часу

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	15.02.2025	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	25.02.2025	виконано
3	Побудова моделі або алгоритму власного рішення	15.03.2025	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	25.04.2025	виконано
5	Оформлення пояснювальної записки кваліфікаційної роботи	10.06.2025	виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 89 сторінок, 19 рисунків, 4 таблиці, список використаних джерел із 26 найменування,

Метою роботи є аналіз програмування систем реального часу, включаючи їх теоретичні основи, апаратне забезпечення та мови програмування, з метою надання рекомендацій для розробки ефективних і надійних систем.

Об'єкт дослідження: Системи реального часу та програмне забезпечення, що забезпечує їх функціонування.

Предмет дослідження: Методи та засоби програмування систем реального часу, включаючи апаратну підтримку, архітектуру процесорів, мови програмування та підходи до дизайну таких систем.

Результати дослідження: Виявлення ключових критеріїв ефективності програмних рішень у системах реального часу.

У першому розділі - розглядаються основи систем реального часу, включаючи поняття, хибні уявлення, багатодисциплінарні проблеми дизайну та їх еволюцію.

Другий розділ - присвячений апаратному забезпеченню, охоплюючи базову архітектуру процесорів, технології пам’яті та сучасні архітектурні розвитки

Третій розділ аналізує мови програмування для систем реального часу, включаючи кодування, асемблераційні, процедурні та об’єктно-орієнтовані мови, а також їхній порівняльний огляд.

Висновок: узагальнено ключові результати та окреслено перспективи розвитку програмування систем реального часу

КЛЮЧОВІ СЛОВА: СИСТЕМИ РЕАЛЬНОГО ЧАСУ, ПРОГРАМУВАННЯ, АПАРАТНЕ ЗАБЕЗПЕЧЕННЯ, МОВИ ПРОГРАМУВАННЯ, АСЕМБЛЕР, C, ADA, C++, ВБУДОВАНІ СИСТЕМИ, ПОРІВНЯЛЬНИЙ АНАЛІЗ.

АНОТАЦІЯ

Бакалаврська робота містить 89 сторінок, 19 рисунків, 4 таблиці, список використаних джерел із 26 найменування,

Метою роботи є аналіз програмування систем реального часу, включаючи їх теоретичні основи, апаратне забезпечення та мови програмування, з метою надання рекомендацій для розробки ефективних і надійних систем.

Об'єкт дослідження: Системи реального часу та програмне забезпечення, що забезпечує їх функціонування.

Предмет дослідження: Методи та засоби програмування систем реального часу, включаючи апаратну підтримку, архітектуру процесорів, мови програмування та підходи до дизайну таких систем.

Результати дослідження: Виявлення ключових критеріїв ефективності програмних рішень у системах реального часу.

У першому розділі - розглядаються основи систем реального часу, включаючи поняття, хибні уявлення, багатодисциплінарні проблеми дизайну та їх еволюцію.

Другий розділ - присвячений апаратному забезпеченню, охоплюючи базову архітектуру процесорів, технології пам’яті та сучасні архітектурні розвитки

Третій розділ аналізує мови програмування для систем реального часу, включаючи кодування, асемблераційні, процедурні та об’єктно-орієнтовані мови, а також їхній порівняльний огляд.

Висновок: узагальнено ключові результати та окреслено перспективи розвитку програмування систем реального часу

КЛЮЧОВІ СЛОВА: СИСТЕМИ РЕАЛЬНОГО ЧАСУ, ПРОГРАМУВАННЯ, АПАРАТНЕ ЗАБЕЗПЕЧЕННЯ, МОВИ ПРОГРАМУВАННЯ, АСЕМБЛЕР, C, ADA, C++, ВБУДОВАНІ СИСТЕМИ, ПОРІВНЯЛЬНИЙ АНАЛІЗ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ЦП - центральний процесор

АЛП - арифметико-логічний пристрій

SRAM - статична оперативна пам'ять

DRAM - динамічна оперативна пам'ять

STL - стандартна мова шаблонів

VLIW - Архітектура дуже довгих слів інструкцій

RAM - Енергонезалежна оперативна пам'ять

ROM - енергонезалежна пам'ять

PI - регістра інструкцій

CORBA - common object request broker architecture

ЗМІСТ

ВСТУП

Програмування систем реального часу (real-time systems) є ключовою дисципліною в сучасній інформатиці та інженерії, яка забезпечує розробку програмного забезпечення, здатного виконувати завдання з чітко визначеними часовими обмеженнями. Такі системи широко застосовуються в критичних галузях, зокрема в аерокосмічній промисловості, автомобільному секторі, медичних пристроях, телекомунікаціях і робототехніці, де затримки або помилки можуть мати серйозні наслідки. Програмування систем реального часу вимагає глибокого розуміння апаратного забезпечення, архітектури процесорів, технологій пам’яті та спеціалізованих мов програмування, таких як асемблер, процедурні та об’єктно-орієнтовані мови, для забезпечення надійності, передбачуваності та ефективності.

Актуальність теми зумовлена швидким розвитком технологій і зростаючою потребою в системах, що функціонують у реальному часі, в умовах цифрової трансформації. За даними аналітичних звітів, ринок вбудованих систем і систем реального часу до 2025 року продовжує зростати, що підкреслює необхідність кваліфікованих розробників, здатних вирішувати багатодисциплінарні проблеми дизайну та адаптуватися до нових архітектурних і програмних викликів. Водночас хибні уявлення про системи реального часу, складність їх кодування та потреба в оптимізації апаратного забезпечення вказують на важливість систематичного дослідження цієї галузі.

Метою цього дослідження є аналіз програмування систем реального часу, включаючи їх теоретичні основи, апаратне забезпечення та мови програмування, з метою надання рекомендацій для розробки ефективних і надійних систем. Робота спрямована на систематизацію знань про принципи роботи систем реального часу, їх еволюцію, а також практичні аспекти створення програмного забезпечення для таких систем. Дослідження охоплює як теоретичні, так і прикладні аспекти, з акцентом на інженерні підходи до

розробки.

У роботі розглянуто основи систем реального часу, включаючи поняття, хибні уявлення, багатодисциплінарні проблеми дизайну та їх еволюцію. Проаналізовано апаратне забезпечення, зокрема базову архітектуру процесорів, технології пам’яті та сучасні архітектурні розвитку. Окремо досліджено мови програмування, такі як асемблер, процедурні (C, Ada) та об’єктно-орієнтовані (C++), а також їх порівняльний огляд для кодування програмного забезпечення реального часу.

Завданнями дослідження: проаналізувати поняття, міфи та основні характеристики систем реального часу, вивчити етапи розвитку та еволюції архітектур систем реального часу, дослідити апаратні компоненти, зокрема архітектуру процесорів і пам’яті, розглянути існуючі мови програмування, що використовуються для створення програмного забезпечення систем реального часу, визначити ефективні методи та підходи до розробки програмного забезпечення для систем із жорсткими часовими обмеженнями.

Об'єкт дослідження: системи реального часу та програмне забезпечення, що забезпечує їх функціонування.

Предмет дослідження: методи та засоби програмування систем реального часу, включаючи апаратну підтримку, архітектуру процесорів, мови програмування та підходи до дизайну таких систем.

Методи дослідження: теоретичний аналіз літератури, порівняльний аналіз, систематизація підходів, моделювання архітектури та програмного забезпечення.

Наукова новизна: виділено нові напрями розвитку апаратних і програмних платформ у сфері реального часу, що можуть бути застосовані в індустріальних системах та вбудованих пристроях.

Бакалаврська робота містить 89 сторінок, 19 рисунків, 4 таблиці, три розділи, список використаних джерел із 26 найменуванням.

РОЗДІЛ 1. ОСНОВИ СИСТЕМ РЕАЛЬНОГО ЧАСУ

1.1 Поняття та хибні уявлення

Фундаментальні визначення інженерії систем реального часу можуть відрізнятися залежно від використаного ресурсу. Наші прагматичні визначення були зібрані та уточнені до найменшої спільної підмножини узгодженості, щоб сформувати словник цього конкретного тексту. Ці визначення представлені у формі, яка має бути найбільш корисною для практикуючого інженера, на відміну від академічного теоретика.

Визначення для систем реального часу

Апаратне забезпечення комп'ютера вирішує проблеми шляхом багаторазового виконання машинно-мовних інструкцій, що разом відомі як програмне забезпечення. Програмне забезпечення, з іншого боку, традиційно поділяється на системні програми та прикладні програми. Системні програми складаються з програмного забезпечення, яке взаємодіє з базовим комп'ютерним обладнанням, таким як драйвери пристроїв, обробники переривань, планувальники завдань та різні програми, що діють як інструменти для розробки або аналізу прикладних програм. Ці програмні засоби включають компілятори, які перетворюють програми мовами високого рівня в код асемблера; асемблери, які перетворюють код асемблера у спеціальний двійковий формат, який називається об'єктним або машинним кодом; та компонувальники/локатори, які готують об'єктний код до виконання в певному апаратному середовищі . Операційна система - це спеціалізований набір системних програм, які керують фізичними ресурсами комп'ютера. Таким чином, операційна система реального часу є справді важливою системною програмою (Anh and Tan, 2009).

Прикладні програми – це програми, написані для вирішення конкретних проблем, таких як оптимальне розподілення викликів у ліфтовому банку висотної будівлі, інерційна навігація літака та підготовка заробітної плати для певної промислової компанії. Певні конструктивні міркування відіграють певну

роль у розробці системних програм та прикладного програмного забезпечення, призначених для роботи в умовах реального часу. Поняття «системи» є центральним для програмної інженерії, та й взагалі для всієї інженерії, і потребує формалізації.

Визначення: Система

Система – це відображення набору вхідних даних у набір виходів. Коли внутрішні деталі системи не становлять особливого інтересу, функцію відображення між вхідним та вихідним просторами можна розглядати як чорну скриньку з одним або кількома входами, що входять, та одним або кількома виходами, що виходять із системи (див. рис. 1.1). Більше того, Вернон перераховує п'ять загальних властивостей, що належать будь-якій «системі» (Вернон, 1989):

1. Система — це сукупність компонентів, з'єднаних між собою організованим чином.
2. Система фундаментально змінюється, якщо до неї приєднується або виходить компонент.
3. Це має мету.
4. Він має певний ступінь сталості.
5. Це було визначено як таке, що становить особливий інтерес.

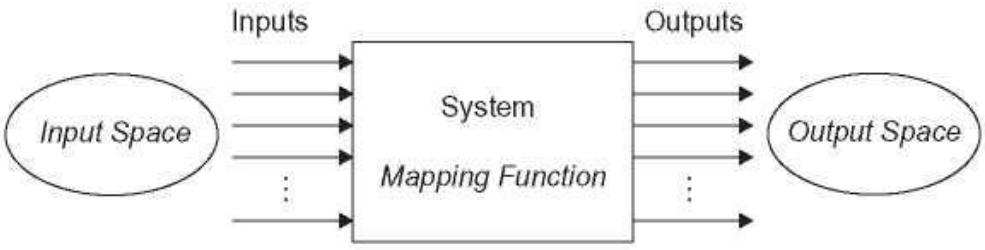

Рисунок 1.1 - Загальна система з входами та виходами.

Кожну реальну сутність, органічну чи синтетичну, можна змоделювати як систему. В обчислювальних системах входи представляють собою цифрові дані

з апаратних пристроїв або інших програмних систем. Входи часто пов'язані з датчиками, камерами та іншими пристроями, які забезпечують аналогові входи, що перетворюються на цифрові дані, або забезпечують прямі цифрові входи. Цифрові виходи комп'ютерних систем, з іншого боку, можна перетворити на аналогові виходи для керування зовнішніми апаратними пристроями, такими як виконавчі механізми та дисплеї, або використовувати безпосередньо без будь-якого перетворення (рис. 1.2).

Рисунок 1.2 - Система керування в реальному часі, що включає вхідні дані від камери та кількох датчиків, а також виходи на дисплей та кілька виконавчих механізмів.

Моделювання системи реального часу (керуючої), як показано на рисунку 1.2, дещо відрізняється від більш традиційної моделі системи реального часу як послідовності завдань, які потрібно запланувати, та продуктивності, яку потрібно передбачити, що можна порівняти з тим, що показано на рисунку 1.3. Останній погляд є спрощеним, оскільки він ігнорує звичайний факт, що джерела вхідних даних та апаратне забезпечення, що керується, можуть бути дуже складними. Крім того, існують інші, «широкі» міркування щодо розробки програмного забезпечення, які приховані моделлю, показаною на рисунку 1.3. Знову розглянемо модель системи реального часу, показану на рисунку 1.2. У її реалізації існує деяка невід'ємна затримка між представленням вхідних даних (збудження) та появою виходів (відгук).

Визначення: Час відгуку

Час між поданням набору вхідних даних до системи та реалізацією

необхідної поведінки, включаючи доступність усіх пов'язаних з ними -виходів, називається часом відгуку системи. Наскільки швидким і пунктуальним має бути час реагування, залежить від характеристик та призначення конкретної системи. Попередні визначення заклали основу для практичного визначення системи реального часу.

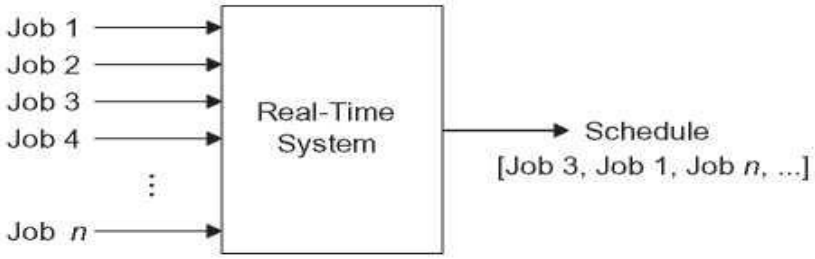

Рисунок 1.3 - Класичне представлення системи реального часу як послідовності планованих завдань .

Визначення: Система реального часу (I)

Система реального часу — це комп'ютерна система, яка повинна задовольняти обмеження часу відгуку, інакше вона ризикує серйозними наслідками, включаючи збій. Але що таке «відмовлена» система? Наприклад, у випадку космічного човника чи атомної електростанції, коли стався збій, це абсолютно очевидно. Для інших систем, таких як автоматичний банківський касир, поняття відмови менш очевидне. Наразі відмова буде визначатися як «нездатність системи працювати відповідно до системної специфікації ».

Визначення: Збій системи

Несправна система – це система, яка не може задовольнити одну або декілька вимог, зазначених у специфікації вимог до системи. Через таке визначення відмови необхідна сувора специфікація критеріїв роботи системи , включаючи часові обмеження. Це питання обговорюється пізніше в розділі 5. Існують різні інші визначення для «реального часу», залежно від джерела, з яким звертаються. Тим не менш, спільною темою всіх визначень є те, що система повинна відповідати обмеженням термінів, щоб бути правильною.

Визначення: Система реального часу (II)

Система реального часу — це така, логічна правильність якої базується як на правильності вихідних даних, так і на їх своєчасності. У будь-якому разі, роблячи непотрібним поняття своєчасності, кожна система стає системою реального часу. Системи реального часу часто є реактивними або вбудованими системами. Реактивні системи – це ті, в яких планування завдань керується постійною взаємодією з навколишнім середовищем; наприклад, система управління вогнем реагує на певні кнопки, натискані пілотом.

Визначення: Вбудована система

Вбудована система — це система, що містить один або декілька комп'ютерів (або процесорів), що відіграють центральну роль у функціональності системи, але сама система не називається комп'ютером. Наприклад, сучасний автомобіль містить багато вбудованих процесорів, які керують розгортанням подушок безпеки, антиблокувальною системою гальм, кондиціонером, уприскуванням палива тощо. Сьогодні численні побутові прилади, такі як мікрохвильові печі, рисоварки, стереосистеми, телевізори, пральні машини та навіть іграшки, містять вбудовані комп'ютери. Очевидно, що складні системи, такі як літаки, ліфти та папероробні машини, містять кілька вбудованих комп'ютерних систем.

Три системи, згадані на початку цього розділу, відповідають критеріям системи реального часу. Літак повинен обробляти дані акселерометра протягом певного періоду, який залежить від характеристик літака; наприклад, кожні 10 мс. Невиконання цієї вимоги може призвести до помилкового відображення місцезнаходження або швидкості, що в кращому випадку може призвести до відхилення літака від курсу або, в гіршому, до аварії. У разі теплової проблеми ядерного реактора нездатність швидко реагувати може призвести до його аварії. Нарешті, система бронювання авіаквитків повинна бути здатною обробляти сплеск запитів пасажирів у межах сприйняття пасажиром розумного часу (або до того, як рейси покинуть гейт). Коротше кажучи, система не повинна обробляти дані одночасно або миттєво, щоб вважатися такою, що працює в реальному часі;

вона просто повинна мати відповідним чином обмежений час реагування.

Коли система працює в режимі реального часу? Можна стверджувати, що всі практичні системи зрештою є системами реального часу. Навіть пакетно-орієнтована система, наприклад, обробка оцінок в кінці семестру або двомісячний цикл нарахування заробітної плати, працює в режимі реального часу. Хоча система може мати час відгуку в дні або навіть тижні (наприклад, час, що минає між поданням оцінки або інформації про заробітну плату та видачею табеля успішності або зарплатного листка), вона повинна реагувати протягом певного часу, інакше може статися академічна або фінансова катастрофа. Навіть програма для обробки текстів повинна реагувати на команди протягом розумного періоду часу, інакше її використання стане болісним. У більшості літератури такі системи називають м'якими системами реального часу.

Визначення: М'яка система реального часу

М'яка система реального часу - це система, в якій продуктивність знижується, але не руйнується через невиконання обмежень часу відгуку. І навпаки, системи, де невиконання обмежень часу відгуку призводить до повного або катастрофічного відмови системи, називаються системами жорсткого реального часу.

Визначення: Система жорсткого реального часу

Система жорсткого реального часу — це така, в якій недотримання навіть одного крайнього терміну може призвести до повного або катастрофічного виходу з ладу системи. Фіксовані системи реального часу — це системи з жорсткими дедлайнами, де можна допустити деяку довільно невелику кількість пропущених дедлайнів.

Визначення: система реального часу

Надійна система реального часу — це така, в якій кілька пропущених термінів не призведуть до повного збою, але пропуск більшої кількості може призвести до повного або катастрофічного виходу з ладу системи. Як зазначалося, всі практичні системи мінімально представляють собою м'які системи реального часу. У таблиці 1.1 наведено ілюстративний приклад

жорстких, стійких та м'яких систем реального часу.

Існує велика свобода для інтерпретації жорстких, стійких та м'яких систем реального часу. Наприклад, у банкоматі пропуск занадто великої кількості дедлайнів призведе до значного невдоволення клієнтів і потенційно навіть достатніх втрат бізнесу, щоб поставити під загрозу існування банку. Цей екстремальний сценарій відображає той факт, що кожну систему часто можна охарактеризувати будь-яким способом — м'яким, стійким або жорстким — реального часу шляхом побудови допоміжного сценарію. Ретельне визначення системних вимог (а отже, і очікувань) є ключем до встановлення та виконання реалістичних очікуваних дедлайнів. У будь-якому випадку, головною метою інженерії систем реального часу є пошук способів перетворення жорстких дедлайнів на тверді, а твердих — на м'які.

Оскільки цей текст здебільшого стосується систем жорсткого реального часу, термін «система реального часу» використовуватиметься у значенні вбудованої системи жорсткого реального часу , якщо не зазначено інше. При вивченні систем реального часу типово розглядати природу часу, оскільки дедлайни – це моменти часу. Тим не менш, виникає питання: «Звідки беруться дедлайни?» Загалом кажучи, дедлайни ґрунтуються на основних фізичних явищах керованої системи.

Наприклад, на анімованих дисплеях зображення повинні оновлюватися щонайменше 30 кадрів на секунду, щоб забезпечити безперервний рух, оскільки людське око може розрізняти оновлення з меншою швидкістю. У навігаційних системах прискорення повинні зчитуватися зі швидкістю, яка є функцією максимальної швидкості транспортного засобу тощо. Однак у деяких випадках реальні системи мають встановлені на них терміни, які базуються не що інше, як здогадки або на якійсь забутій і, можливо, усунені вимозі. Проблема в цих випадках полягає в тому, що на системи можуть бути накладені надмірні обмеження. Це основна максима проектування систем реального часу — розуміти основу та природу часових обмежень, щоб їх можна було послабити за необхідності. У економічно ефективних та надійних системах реального часу

прагматичним правилом може бути: обробляти все якомога повільніше та повторювати завдання якомога рідше.

Таблиця 1.1

Вибірка жорстких, стійких та м'яких систем реального часу

Система	У режимі реального часу Класифікація	Пояснення
Система доставки авіоніки, в якій натисканням кнопки запускається ракета класу "повітря-повітря".	Важко	Пропуск терміну запуску ракети протягом заданого часу після натискання кнопки може призвести до промаху по цілі, що призведе до катастрофи.
Навігаційний контролер для автономного робота - винушильника	Фірма	Пропуск кількох термінів навігації призводить до того, що робот відхиляється від запланованого шляху та пошкоджує деякі посіви
Консольна хокейна гра	М'який	Пропуск навіть кількох дедлайнів лише погіршить продуктивність

Багато систем реального часу використовують глобальні годинники та мітки часу для синхронізації , ініціювання завдань та маркування даних. Однак слід зазначити, що всі годинники показують дещо неточний час — навіть офіційний атомний годинник США необхідно регулярно налаштовувати. Крім того, з годинниками пов'язана похибка квантування, яку, можливо, доведеться враховувати під час їх використання для міток часу. Окрім ступеня «реального часу» (тобто жорсткого, твердого або м’якого), у багатьох застосуваннях важлива також пунктуальність часу реагування.

У програмних системах зміна стану призводить до зміни потоку керування комп'ютерною програмою. Розглянемо блок-схему на рисунку 1.4. Блок рішення, представлений ромбом, свідчить про те, що потік інструкцій програми може обрати один із двох альтернативних шляхів, залежно від відповідної відповіді. Оператори case , if-then та while в будь-якій мові програмування представляють можливу зміну потоку керування.

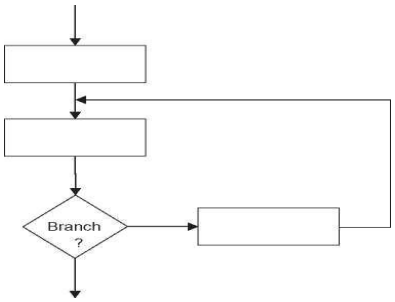

Рисунок 1.4 - Блок-схема часткової програми, що показує умовний перехід як зміну потоку керування.

Визначення: Час вивільнення

Час випуску – це час, коли екземпляр запланованого завдання готовий до виконання, і зазвичай пов'язаний з перериванням. Події дещо відрізняються від завдань тим, що події можуть бути спричинені перериваннями , а також розгалуженнями. Подія може бути синхронною або асинхронною. Синхронні події – це ті, що відбуваються в передбачуваний час у потоці керування, наприклад, представлені блоком рішення на блок-схемі на рисунку 1.4. Зміна потоку керування, представлена умовною інструкцією перемикання або виникненням внутрішнього переривання-пастки, може бути передбачена.

Асинхронні події виникають у непередбачуваних точках потоку керування та зазвичай спричинені зовнішніми джерелами. Годинник реального часу, який регулярно пульсує з частотою 5 мс, не є синхронною подією. Хоча він являє собою періодичну подію, навіть якби годинник міг цокати з ідеальною частотою 5 мс без дрейфу, точка, де відбувається цокання потоку керування, залежить від багатьох факторів.

Таблиця 1.2.

Таксономія подій та деякі типові приклади

	Періодичні	Аперіодичний	Спорадичні
Синхронний	Циклічний код	Умовне розгалуження	Ділення на нуль (пастка) переривання
Асинхронний	Переривання годинника	Звичайний, але не переривання з фіксованим періодом	Сигналізація про втрату живлення

Ці фактори включають час запуску годинника відносно програми та затримки поширення в самій комп'ютерній системі. Інженер ніколи не може розраховувати на те, що годинник цокає точно з заданою частотою, тому будь-яку подію, керовану годинником, слід розглядати як асинхронну. Події, які не відбуваються з регулярними періодами, називаються аперіодичними. Крім того, аперіодичні події, які відбуваються дуже рідко, називаються спорадичними. У таблиці 1.2 наведено вибірку подій.

У кожній системі, і особливо у вбудованій системі реального часу, - підтримка загального контролю є надзвичайно важливою. Для будь-якої фізичної системи існують певні стани, за яких система вважається некерованою; тому програмне забезпечення, що керує такою системою, повинно уникати цих станів. Наприклад, у деяких системах керування літаками швидке обертання на кут тангажу 180° може призвести до втрати гіроскопічного контролю. Отже, програмне забезпечення повинно бути здатним передбачати та запобігати всім таким сценаріям.

Ще однією характеристикою програмно-керованої системи є те, що процесор продовжує правильно вибирати, декодувати та виконувати інструкції з програмної області пам'яті, а не з даних чи інших небажаних областей пам'яті. Останній сценарій може статися в погано протестованих системах і є катастрофою, від якої майже немає надії на відновлення. Програмне керування будь-якою системою реального часу та пов'язаним з нею обладнанням підтримується, коли наступний стан системи, враховуючи поточний стан та набір вхідних даних, є передбачуваним. Іншими словами, метою є передбачити, як система поводитиметься за всіх можливих обставин.

Визначення: Детермінована система

Система є детермінованою, якщо для кожного можливого стану та кожного набору вхідних даних можна визначити унікальний набір виходів та наступний стан системи. Детермінізм подій означає, що наступні стани та виходи системи відомі для кожного набору вхідних даних, які запускають події. Таким чином, система, яка є детермінованою, також є детермінованою за подіями. Хоча

системі було б важко бути детермінованою лише для тих вхідних даних, які запускають події, це правдоподібно, і тому детермінізм подій може не означати детермінізм.

Цікаво відзначити, що хоча проектування систем, які є повністю подієво-детермінованими, є значним викликом, і, як згадувалося, можливо ненавмисно отримати систему, яка не є детермінованою, безумовно важко проектувати системи, які навмисно недетерміновані. Ця ситуація виникає через надзвичайні труднощі в проектуванні ідеальних генераторів випадкових чисел. Такі навмисно недетерміновані системи були б бажаними , наприклад, як ігрові автомати в казино. Зрештою, якщо в детермінованій системі час відгуку для кожного набору виходів відомий, то система також демонструє часовий детермінізм. Побічна перевага проектування детермінованих систем полягає в тому, що можна гарантувати, що система зможе реагувати в будь-який час, а у випадку часово детермінованих систем – коли вони реагуватимуть. Цей факт підсилює асоціацію «керування» з системами реального часу.

Останній і справді важливий термін, який потрібно визначити, – це критичний показник продуктивності системи реального часу. Оскільки центральний процесор (ЦП) продовжує отримувати, декодувати та виконувати інструкції доти, доки подається живлення, ЦП більш-менш часто виконуватиме або інструкції, що не пов'язані з виконанням певного терміну (наприклад, некритичне « ведення домашнього господарства»). Міра відносного часу, витраченого на обробку даних поза станом простою, показує, скільки часу виконується в режимі реального часу.

Звичайні помилки

Для повного розуміння природи систем реального часу важливо розглянути низку часто цитованих помилкових уявлень. Вони коротко викладені в наступному:

1. Системи реального часу є синонімом «швидких» систем.
2. Монотонний аналіз зі швидкістю вирішив «проблему реального часу».

3. Існують універсальні, широко прийняті методології для специфікації та проектування систем реального часу.

4. Більше немає потреби створювати операційну систему реального часу, оскільки існує багато комерційних продуктів.

5. Вивчення систем реального часу здебільшого стосується теорії планування.

Перша помилкова думка, що системи реального часу повинні бути швидкими, виникає через те, що багато систем жорсткого реального часу справді мають справу з дедлайнами в десятки мілісекунд, як-от навігаційна система літака. Однак у типовому застосуванні харчової промисловості банки з макаронами та соусом можуть рухатися конвеєрною стрічкою повз точку розливу зі швидкістю один раз кожні п'ять секунд. Крім того, система бронювання авіаквитків може мати дедлайн у 15 секунд. Ці останні дедлайни не є особливо швидкими, але їх дотримання визначає успіх чи невдачу системи.

Друга помилкова думка полягає в тому, що системи, що працюють за швидкістю, є простим рецептом для побудови систем реального часу. Системи, що працюють за швидкістю, - періодична система, в якій пріоритети переривань (або програмних завдань) призначаються таким чином, що чим вища швидкість виконання, тим вищий пріоритет, - отримали значну увагу з 1970-х років. Хоча вони надають цінні рекомендації щодо проектування систем реального часу, і хоча навколо них існує багато теорії, вони не є панацеєю.

А як щодо третьої помилки? На жаль, не існує загальноприйнятих та безпомилкових методів специфікації та проектування систем реального часу. Це не є провалом дослідників чи індустрії програмного забезпечення, а пов'язано зі складністю пошуку універсальних рішень для цієї вимогливої галузі. Після майже 40 років досліджень та розробок досі не існує методології, яка б відповідала на всі виклики специфікації та проектування систем реального часу завжди та для всіх застосувань.

1.2 Багатодисциплінарні проблеми дизайну

Вивчення систем реального часу – це справді багатовимірна піддисципліна комп'ютерної системної інженерії, на яку сильно впливають теорія керування, дослідження операцій та, природно, програмна інженерія. На рисунку 1.5 зображено деякі дисципліни інформатики, електротехніки, системної інженерії та прикладної статистики, які впливають на проектування та аналіз систем реального часу. Тим не менш, ці репрезентативні дисципліни – не єдині, що мають зв'язок із системами реального часу. Оскільки системна інженерія реального часу є настільки багатопрофільною, вона виділяється як захоплива галузь дослідження з багатим набором проектних завдань. Хоча основи систем реального часу добре розроблені та мають значну сталість, системи реального часу є активно розвиваючою галуззю , наприклад, завдяки розвитку архітектур процесорів, розподілених системних структур, універсальних бездротових мереж та нових застосувань .

Впливові дисципліни Проектування та впровадження систем реального часу вимагає уваги до численних практичних питань. До них належать:

- Вибір апаратного та системного програмного забезпечення, а також оцінка компромісу, необхідного для конкурентного рішення, включаючи роботу з розподіленими обчислювальними системами та питаннями паралельності та синхронізації.

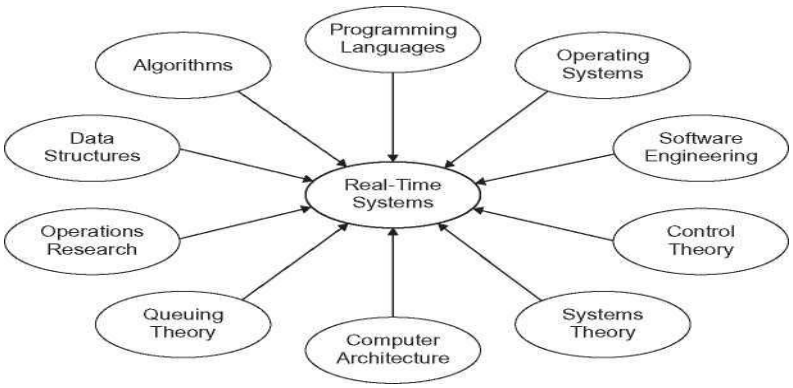

Рисунок 1.5 - Різноманітні дисципліни, що впливають на інженерію систем реального часу.

- Специфікація та проектування систем реального часу, а також правильне та інклюзивне представлення часової поведінки.
- Розуміння нюансів високорівневої(их) мови(мов) програмування та наслідків для реального часу, що виникають в результаті їх оптимізованої компіляції в машинний код.
- Оптимізація (з урахуванням цілей конкретного застосування) відмовостійкості та надійності системи шляхом ретельного проектування та аналізу.
- Розробка та проведення адекватних тестів на різних рівнях ієрархії, а також вибір відповідних інструментів розробки та тестового обладнання.
- Використання переваг технології відкритих систем та сумісності. Відкрита система — це розширювана колекція незалежно написаних програм, які співпрацюють, щоб функціонувати як інтегрована система. Наприклад, кілька версій відкритої операційної системи Linux були об'єднані для використання в різних програмах реального часу [8] Сумісність можна виміряти з точки зору відповідності стандартам відкритих систем, таким як стандарт реального часу CORBA (common object request broker architecture)–[2].

- Нарешті, оцінка та вимірювання часу реагування та (за потреби) його скорочення. Проведення аналізу планування, тобто визначення та гарантування дотримання термінів апріорі.

Очевидно, що інженерні методи, що використовуються для систем жорсткого реального часу, можуть бути використані також і в інженерії всіх інших типів систем, що супроводжується покращенням продуктивності та стійкості. Вже саме це є вагомою причиною для вивчення інженерії систем реального часу.

1.3 Народження та еволюція систем реального часу

Історія систем реального часу, що характеризується важливими розробками у Сполучених Штатах, нерозривно пов'язана з еволюцією комп'ютерів. Сучасні системи реального часу, такі як ті, що керують атомними електростанціями, військовими системами озброєння або медичним моніторинговим обладнанням, є складними, проте багато з них досі демонструють характеристики тих новаторських систем, розроблених у 1940-х – 1960-х роках.

Диверсифікація застосувань

Вбудовані системи реального часу настільки поширені та повсюдні, що їх можна знайти навіть у побутовій техніці, спортивному одязі та іграшках. Невелика вибірка областей реального часу та відповідних застосувань наведена в таблиці 1.4. Чудовим прикладом передової системи реального часу є марсохід NASA, показаний на рисунку 1.6. Це автономна система з надзвичайними вимогами до надійності; вона отримує команди та надсилає дані вимірювань через канали радіозв'язку; і виконує свої наукові місії за допомогою кількох датчиків, процесорів та виконавчих механізмів. У вступних абзацах цього розділу було згадано деякі системи реального часу. Наступні описи надають більше деталей для кожної системи, а інші надають додаткові приклади
Зрозуміло, що ці описи не є суворими специфікаціями.

Розглянемо інерційну вимірювальну систему для літака. У специфікації програмного забезпечення зазначено, що програмне забезпечення отримуватиме імпульси акселерометра x, y та z з частотою 10 мс від спеціального обладнання. Програмне забезпечення визначатиме компоненти прискорення в кожному напрямку, а також відповідні крен, тангаж та рискування літака. Програмне забезпечення також збиратиме іншу інформацію, таку як температура, з частотою 1 секунду. Завдання прикладного програмного забезпечення полягає в обчисленні фактичного вектора швидкості на основі поточної орієнтації, показань акселерометра та різних коефіцієнтів компенсації (наприклад, для

впливу температури) з частотою 40 мс. Система повинна виводити справжні вектори прискорення, швидкості та положення на дисплей пілота кожні 40 мс, але використовуючи інший тактовий генератор.

Таблиця 1.4.

Типові домени реального часу та різноманітні застосування

Домен	Застосування
Аерокосмічна галузь	Інтерфейс пілота навігації та керування польотом
Цивільний	Автомобільні системи Управління ліфтом Управління світлофорами
Промисловий	Автоматизована перевірка Роботизована складальна лінія Контроль зварювання
Медичний	Монітори інтенсивної терапії Магнітно-резонансна томографія Дистанційна хірургія
Мультимедіа	Консольні ігри Домашні кінотеатри Симулятори

Ці завдання виконуються з чотирма різними швидкостями в інерціальній вимірювальній системі та потребують обміну даними та синхронізації. Показники акселерометра повинні бути відносними до часу або корельованими; тобто не допускається змішування % імпульсу акселерометра дискретного моменту часу k з імпульсами y та z моменту k + 1. Це критичні проблеми проектування цієї системи.

Далі розглянемо систему моніторингу атомної електростанції, яка оброблятиме три події, що сигналізуються перериваннями. Перша подія ініціюється будь-яким із кількох сигналів у різних точках безпеки, що свідчитиме про порушення безпеки. Система повинна відреагувати на цей сигнал протягом однієї секунди. Друга і найважливіша подія вказує на те, що активна зона реактора досягла перегріву . Цей сигнал має бути оброблений протягом 1 мілісекунди (1 мс). Нарешті, дисплей оператора має оновлюватися приблизно 30 разів на секунду. Системі атомної електростанції потрібен надійний механізм, який гарантуватиме, що індикатор «неминучого розплавлення» може перервати

будь-яку іншу обробку з мінімальною затримкою.

Рисунок 1.6 - Марсохід Mars Exploration Rover; автономна система реального часу на сонячній енергії з радіозв'язком та різноманітними датчиками й виконавчими механізмами. Фото надано NASA.

Як інший приклад, згадаємо систему бронювання авіаквитків, згадану раніше. Керівництво вирішило, що для запобігання довгим чергам та невдоволенню клієнтів час обробки будь-якої транзакції має бути менше 15 секунд, і не допускається овербукінг. У будь-який час кілька туристичних агентів можуть спробувати отримати доступ до бази даних бронювань і, можливо, забронювати один і той самий рейс одночасно . Тут необхідні ефективні механізми блокування записів та безпечного зв'язку для захисту від зміни бази даних, що містить інформацію про бронювання, кількома співробітниками одночасно.

Тепер розглянемо систему реального часу, яка контролює всі фази розливу банок з соусом для пасти, коли вони рухаються конвеєрною стрічкою. Порожні банки спочатку проходять мікрохвильову піч для їх дезінфекції. Механізм наповнює кожну банку точною порцією певного соусу, коли вона проходить під нею. Інша станція закриває наповнені пляшки. Крім того, є дисплей оператора, який забезпечує анімоване відображення діяльності виробничої лінії. Існує безліч подій, що -викликаються винятковими умовами, такими як заклинювання конвеєрної стрічки та переповнення або розбиття

пляшки. Якщо конвеєрна стрічка рухається занадто швидко, пляшка передчасно пройде повз призначену для неї станцію. Таким чином, існує широкий спектр подій, як синхронних, так і асинхронних, з якими потрібно впоратися.

Як останній приклад, розглянемо систему, яка використовується для керування світлофорами на перехресті з чотиристороннім рухом (рух у північному, південному, східному та західному напрямках). Ця система керує світлофорами для руху транспортних засобів та пішоходів на перехресті з чотиристороннім рухом у жвавому місті, такому як Філадельфія. Вхідні дані можуть надходити з камер, транспондерів автомобілів екстреної допомоги, кнопок, датчиків під землею тощо. Світлофори повинні працювати синхронізовано, але реагувати на асинхронні події, такі як натискання кнопки пішоходом на пішохідному переході. Неналежне функціонування може призвести до автомобільних аварій і навіть смертельних випадків.

1.4 Висновок по розділу

Системи реальної години — це складні комп'ютерні системи, які повинні виконувати завдання в межах суворих часових обмежень, причому їх правильність залежить не тільки від точності результату, а й від своєчасності його отримання. Розуміння природи дедлайнів, класифікації систем (м'які, тверді, жорсткі) та уникнення поширених хибних уявлень є ключем до успішного проектування таких систем.

Інженерія систем реальної години є міждисциплінарною та динамічною галуззю, що охоплює широкий спектр застосувань — від аерокосмічної промисловості до побутових приладів — і вимагає високої точності, надійності та вчасного реагування на події

РОЗДІЛ 2. АПАРАТНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ СИСТЕМ РЕАЛЬНОГО ЧАСУ

Існує очевидна потреба в базовому розумінні апаратного забезпечення серед розробників програмного забезпечення та системних інженерів, особливо під час проектування або аналізу вбудованих систем реального часу.. Отже, він також є корисним оглядом для фахівців, орієнтованих на апаратне забезпечення. У системах реального часу багатокрокові та змінні в часі шляхи затримки від входів (збуджень) до виходів (відповідей) створюють значні проблеми з часом та затримкою, які слід розуміти та належним чином керувати, наприклад, під час проектування програмного забезпечення реального часу або інтеграції програмного забезпечення з апаратним забезпеченням. Такі проблеми, природно, мають різну складність та важливість залежно від конкретного застосування, з яким ми маємо справу. Існує широкий спектр великомасштабних та більш компактних застосувань реального часу, від глобальних систем бронювання авіаквитків до нових повсюдних обчислень. Так само апаратні платформи можуть значно відрізнятися: від мережевих багатоядерних робочих станцій до окремих 8-бітних або навіть 4-бітних мікроконтролерів. Хоча питання, пов'язані з апаратним забезпеченням, є досить абстрактними для програмістів додатків, які розробляють програмне забезпечення для робочих станцій, вони є справді конкретними для системних програмістів та осіб, які працюють із вбудованими мікроконтролерами або цифровими сигнальними процесорами.

Існує очевидна потреба в базовому розумінні апаратного забезпечення серед розробників програмного забезпечення та системних інженерів, особливо під час проектування або аналізу вбудованих систем реального часу. Цей розділ пропонує цілеспрямований вступ до фундаментальних -питань, пов'язаних з апаратним забезпеченням, з точки зору реального часу. Отже, він також є корисним оглядом для фахівців, орієнтованих на апаратне забезпечення. У системах реального часу багатокрокові та змінні в часі шляхи затримки від

входів (збуджень) до виходів (відповідей) створюють значні проблеми з часом та затримкою, які слід розуміти та належним чином керувати, наприклад, під час проектування програмного забезпечення реального часу або інтеграції програмного забезпечення з апаратним забезпеченням. Такі проблеми, природно, мають різну складність та важливість залежно від конкретного застосування, з яким ми маємо справу. Існує широкий спектр великомасштабних та більш компактних застосувань реального часу, від глобальних систем бронювання авіаквитків до нових повсюдних обчислень. Так само апаратні платформи можуть значно відрізнятися: від мережевих багатоядерних робочих станцій до окремих 8-бітних або навіть 4-бітних мікроконтролерів. Хоча питання, пов'язані з апаратним забезпеченням, є досить абстрактними для програмістів додатків, які розробляють програмне забезпечення для робочих станцій, вони є справді конкретними для системних програмістів та осіб, які працюють із вбудованими мікроконтролерами або цифровими сигнальними процесорами.

2.1 Базова архітектура процесора

У наступних підрозділах ми спочатку представимо базову архітектуру процесора та визначимо деякі основні терміни щодо архітектур комп'ютерів, обробки інструкцій та організації вводу/виводу (введення/виведення). Цей вступ формує міцну основу для наступних розділів цього розділу, присвячених архітектурним та іншим удосконаленням апаратного забезпечення.

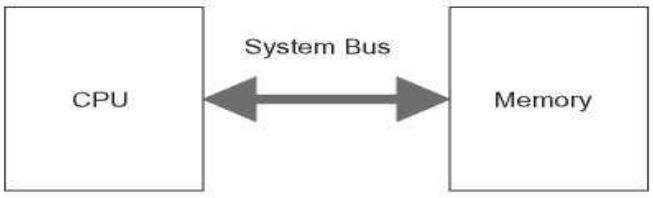

Рисунок 2.1 - Архітектура комп'ютера фон Неймана без явного елемента вводу/виводу.

Архітектура фон Неймана

Традиційна архітектура комп'ютерів фон Неймана, також відома як архітектура Принстона, використовується в численних комерційних процесорах і може бути зображена лише трьома елементами: центральним процесором (ЦП), системною шиною та пам'яттю. На рисунку 2.1 показано таку архітектуру, де ЦП підключено до пам'яті через системну шину. При більш детальному розгляді системна шина насправді являє собою набір із трьох окремих шин: адресної, даних та керування. З цих паралельних шин адресна шина є односпрямованою та керується ЦП; шина даних є двонаправленою та передає як інструкції, так і дані; а шина керування являє собою гетерогенний набір незалежних ліній керування, стану, тактової частоти та живлення. Процесор у застосунку реального часу має групу з 4, 8, 16, 24, 32, 64 або навіть більше ліній даних, які разом утворюють шину даних. З іншого боку, ширина адресної шини зазвичай становить від 16 до 32 біт. У базовій архітектурі фон Неймана регістри вводу/виводу називаються відображеними в пам'ять, оскільки доступ до них здійснюється так само, як і до звичайних комірок пам'яті. Як приклад багатьох варіантів реалізації для практичних комп'ютерів фон Неймана, протокол шини даних може бути синхронним або асинхронним; перший забезпечує простішу структуру реалізації, а другий є більш гнучким щодо різних часів доступу до пам'яті та пристроїв вводу/виводу.

Центральний процесор (ЦП) є основним пристроєм, де відбувається обробка інструкцій; він складається з блоку керування, внутрішньої шини та тракту даних, як показано на рисунку. 2.2 Крім того, тракт даних містить багатофункціональний арифметико-логічний пристрій (АЛП), банк робочих регістрів, а також регістр стану. Блок керування взаємодіє із системною шиною через регістр лічильника програм (ЛПК), який адресує зовнішню комірку пам'яті, з якої наступна інструкція буде вибрана до регістра інструкцій (РІ). Кожна вибрана інструкція спочатку декодується в блоці керування, де ідентифікується конкретний код інструкції. Після ідентифікації коду інструкції блок керування відповідно командує тракт даних, подібно до кінцевого автомата типу Мілі.

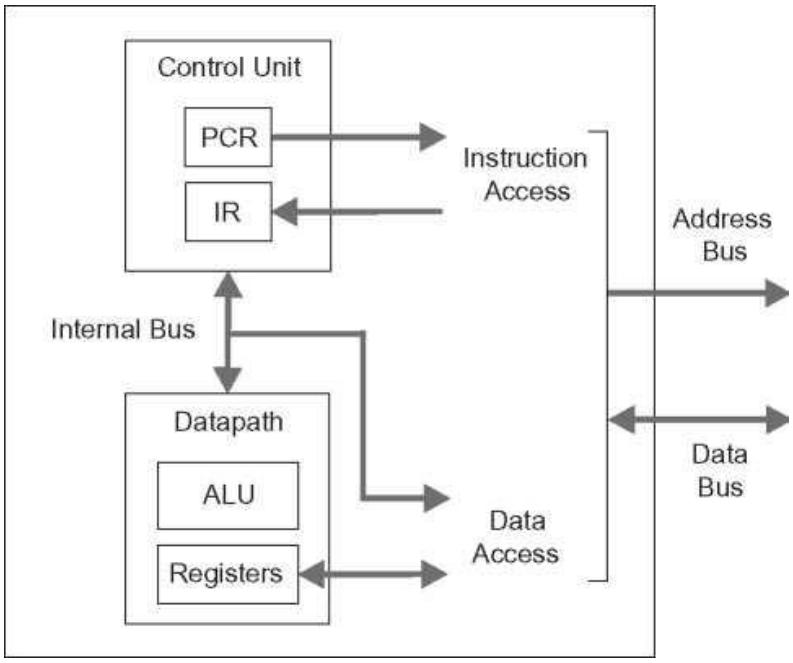

Рисунок 2.2 - Внутрішня структура спрощеного процесора. Доступ до інструкцій та доступ до даних попарно об'єднані для формування спільних шин адреси та даних.

У той час як цілочисельні дані зазвичай можна зберігати в 1, 2 або 4 байтах, числа з плаваючою комою зазвичай займають 4 або більше байтів пам'яті. Банк робочих регістрів утворює швидкий інтерфейсний буфер між АЛП та пам'яттю. Окремі біти або прапорці регістра стану оновлюються відповідно до результату попередньої операції АЛП та поточного стану процесора. Певні прапорці стану, такі як «нуль» та «перенесення/запозичення», використовуються для реалізації інструкцій умовного переміщення та додавання/віднімання з розширеною точністю. Існує внутрішній годинник та інші сигнали, що використовуються для синхронізації та передачі даних, а також численні приховані регістри, які знаходяться всередині процесора, але не показані на рисунку 2.2.

Ця архітектурна структура пропонує кілька параметрів проектування, які можна адаптувати до вимог конкретного застосування та обмежень реалізації : набір інструкцій, блок керування, функції АЛП, розмір банку регістрів, розрядність шини та тактова частота. Хоча архітектура фон Неймана широко використовується в різних процесорах, іноді вважається серйозним обмеженням

те, що інструкції та дані доступні лише послідовно, використовуючи єдину системну шину. З іншого боку, така проста структура шини є компактною для реалізації.

Обробка інструкцій

Обробка інструкцій складається з кількох послідовних фаз, що займають різну кількість тактів. Ці незалежні фази разом утворюють цикл інструкцій. У цьому тексті ми припускаємо п'ятифазний цикл інструкцій: інструкція вибірки , інструкція декодування , завантаження операнда, виконання функції АЛП та збереження результату. На рисунку 2.3 показано часову діаграму послідовного циклу інструкцій. Тривалість циклу інструкцій залежить від самої інструкції; множення зазвичай займає більше часу, ніж просте переміщення з регістра в регістр. Крім того, не всі інструкції потребують активних фаз завантаження, виконання та/або збереження, але ці відсутні фази або пропускаються, або заповнюються циклами простою.

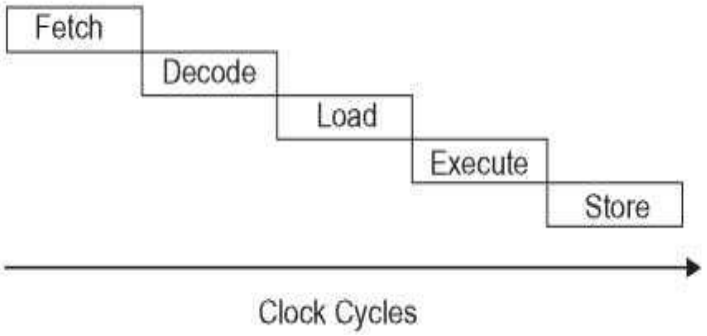

Рисунок 2.3 - Послідовний цикл команд з п'ятьма фазами.

Кожна інструкція представлена своїм унікальним двійковим кодом, який зберігається в пам'яті, і потік таких кодів утворює програму машинною мовою. Однак у наступних параграфах ми будемо використовувати мнемонічні коди для інструкцій замість двійкових кодів. Ці мнемонічні коди інструкцій більш відомі як інструкції асемблерної мови, і вони існують лише для того, щоб полегшити наше життя - процесор використовує лише двійкові коди. Щоб зрозуміти специфіку обробки інструкцій, корисно дати короткий вступ до інструкцій

асемблерної мови. Можна сказати, що набір інструкцій описує функціональність процесора. Він також тісно пов'язаний з архітектурою процесора.

Реальні процесори часто мають помірний набір режимів адресації та вичерпний набір інструкцій. Це, очевидно, призводить до значного навантаження на етапі декодування інструкцій, оскільки кожна інструкція з різним режимом адресації розглядається як окрема інструкція, коли ідентифікується код інструкції. Наприклад, одна інструкція ADD розглядається як чотири окремі інструкції, якщо доступні чотири режими адресації, які обговорювалися вище. Після ідентифікації коду інструкції блок керування створює відповідну послідовність команд для виконання цієї інструкції.

Існує два основних методи реалізації блоку керування: мікропрограмування та апаратно-програмна логіка. У мікропрограмуванні кожна інструкція визначається мікропрограмою, що складається з послідовності примітивних апаратних команд, мікроінструкцій, для активації відповідних функцій шляху даних та додаткових підоперацій. В принципі, побудувати інструкції машинною мовою за допомогою мікропрограмування просто, але такі послідовності мікроінструкцій, як правило, використовують кілька тактів. Це може стати перешкодою зі складними інструкціями, які вимагають відносно великої кількості тактів. Користувачі комерційних процесорів не мають доступу до пам'яті мікропрограм, але вона налаштовується постійно тими, хто реалізує набір інструкцій. У процесорах з невеликим набором інструкцій або вимогою дуже швидкої обробки інструкцій блок керування зазвичай реалізується з використанням апаратно-програмної логіки, яка складається з комбінаторних та послідовних цифрових схем. Цей низькорівневий варіант реалізації займає більше місця на інструкцію порівняно з мікропрограмуванням, але він може запропонувати помітно швидше виконання інструкцій. Тим не менш, створювати або змінювати інструкції машинною мовою складніше, коли використовується апаратно-програмний блок керування. Як ми побачимо пізніше, коли обговорюватимуться передові архітектури процесорів, у сучасних комерційних процесорах широко використовуються як мікропрограмування, так і апаратно-

програмна логіка. Ця ситуація головним чином пов'язана з проблемою реалізації, пов'язаною з розміром та складністю набору команд.

У попередніх абзацах ми представили основний принцип обробки інструкцій , який складається з п'яти послідовних фаз та відсутності паралелізму. Фактично, він, здається, спирається на неявне мислення, що послідовні інструкції в програмі мають як внутрішню, так і взаємну залежність, що запобігає будь-якому паралелізму на рівні інструкцій. Це нереалістичне обмеження робить низьким коефіцієнт використання ресурсів АЛП, і тому він значно полегшується в передових комп'ютерних архітектурах. Навіть з цією еталонною архітектурою все ще можливо підвищити обчислювальну продуктивність, використовуючи широкі внутрішні та системні шини, високу тактову частоту та великий банк робочих регістрів, щоб зменшити потребу в (повільнішому) доступі до зовнішньої пам'яті. Ці прості вдосконалення мають прямий зв'язок з апаратними обмеженнями: бажаними розмірами інтегральної схеми та використаною технологією виготовлення. Крім того, з точки зору систем реального часу (час відгуку та його пунктуальність), такі вдосконалення є цілком доцільними.

Щоб заощадити енергію та зробити програмне забезпечення «зеленим», багато сучасних процесорів мають режим уповільнення. Певні інструкції можуть знижувати напругу схеми та тактову частоту, тим самим уповільнюючи роботу комп'ютера, споживаючи менше енергії та -виділяючи менше тепла. Використання цієї функції є особливо складним для розробників реального часу, яким доводиться турбуватися про дотримання термінів та варіації часу виконання завдань.

2.2 Технології пам'яті

Розуміння центральних характеристик сучасних технологій пам'яті необхідне під час проектування та аналізу систем реального часу. Це особливо важливо, наприклад, для таких вбудованих програм, де коефіцієнт використання

процесора планується залишати в межах «небезпечної» зони 83-99% (див. Розділ 1). У цих майже перевантажених часом системах найгірша затримка доступу до ієрархічної архітектури пам'яті може призвести до неперіодичного пропуску термінів зі значною затримкою. Наступні підрозділи містять поведінкові та якісні обговорення, які орієнтовані радше на розробників програмного забезпечення та систем, ніж на розробників апаратного забезпечення. Детальний огляд пам'яті та систем пам'яті для вбудованих програм доступний у Peckol (2008).

Різні класи пам'яті

Енергонезалежна оперативна пам'ять (RAM) та енергонезалежна пам'ять (ROM) – це традиційна відмінність між двома основними групами напівпровідникової пам'яті, де RAM позначає пам'ять з довільним доступом, а ROM – пам'ять лише для читання. Протягом багатьох років ця відмінність була чіткою, доки пристрої ROM були виключно такого типу , що їхній вміст «програмувався» або під час виробничого процесу мікросхеми пам'яті, або на заводі-виробнику. Сьогодні межа між групами RAM та ROM вже не така чітка, оскільки зазвичай використовувані класи ROM, EEPROM та Flash, можна перезаписати без спеціального програмного блоку; таким чином, вони програмуються в системі. Хоча існує багато різних класів пам'яті в межах двох основних груп, нижче представлені лише найважливіші з них. На рисунку 2.5 зображено звичайні лінії інтерфейсу загального компонента пам'яті.

Електрично стираний програмований ПЗП (EEPROM) та близька до нього флеш-пам'ять базуються на принципі динамічного плаваючого затвора, і обидва можуть бути перезаписані подібно до пристроїв RAM. Однак процес стирання та запису в цих пристроях типу ROM набагато повільніший, ніж у випадку RAM.

Більше того, кожну комірку пам'яті зазвичай можна перезаписати лише 100 000-1 000 000 разів, оскільки стресовий процес перезапису зношує комірки пам'яті EEPROM та Flash-компонентів. Основні 1-бітні комірки пам'яті конфігуровані в масив, утворюючи практичний пристрій пам'яті.

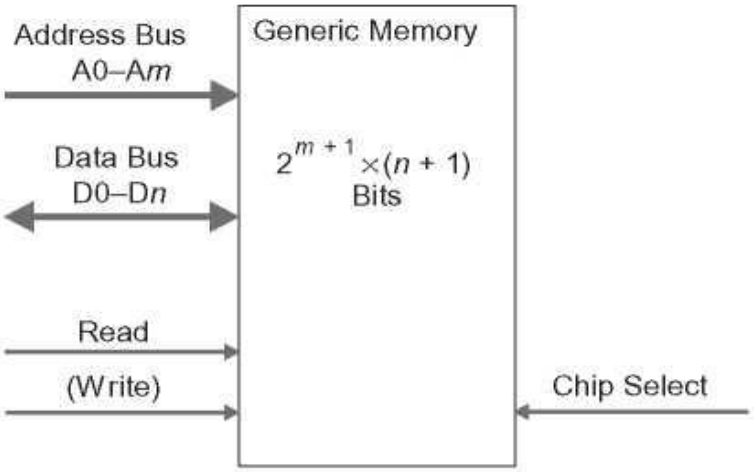

Рисунок 2.5 - Лінії інтерфейсу загального компонента пам'яті (запис не використовується з пристроями ROM).

Місткість пам'яті становить 2 m + 1 x (n + 1) бітів.Хоча окремі комірки пам'яті можна перезаписувати розріджено за допомогою EEPROM, Flash-пам'ять можна стерти лише великими блоками. Таким чином, ці перезаписувані ПЗП, які можуть зберігати свої дані приблизно 10 років, аж ніяк не є конкурентами пристроїв RAM, але вони призначені для інших цілей. EE PROM зазвичай використовуються як енергонезалежне сховище програм і параметрів, а Flash-пам'ять використовується для зберігання як прикладних програм, так і великих записів даних. Неперезаписуваний ПЗП з масковим програмуванням все ще використовується як недорога пам'ять програм у певних стандартизованих додатках з дуже великим обсягом виробництва. Нарешті, слід зазначити, що пам'ять типу ROM зчитується повільніше, ніж типові пристрої RAM. Тому в багатьох застосунках реального часу може бути доцільним або навіть необхідним запускати програми з швидшої оперативної пам'яті (RAM), а не з ПЗП (ROM). Іншою поширеною практикою є завантаження прикладної програми з рухомої карти флеш-пам'яті (або USB-накопичувача) в оперативну пам'ять для виконання. Таким чином, флеш-пристрій поводиться як надійна та недорога масова пам'ять для вбудованих систем.

Існує два класи пристроїв оперативної пам'яті: статична оперативна пам'ять (SRAM) та динамічна оперативна пам'ять (DRAM). Будь-який з цих

класів або обидва вони також використовуються в системах реального часу. Одна комірка пам'яті типу SRAM зазвичай потребує шести транзисторів для реалізації структури бістабільного тригера, тоді як комірка DRAM може бути реалізована лише з одним транзистором та конденсатором. Отже, якщо порівнювати пам'ять однакового розміру та подібної технології виготовлення, SRAM за своєю структурою є більш просторово місткими та дорожчими, але швидшими для доступу, а DRAM дуже компактні та дешевші, але повільніші для доступу.

Через властивий -витік заряду в їхніх конденсаторах зберігання, DRAM необхідно регулярно оновлювати, щоб уникнути втрати даних; період оновлення повинен бути не повільнішим за 3-4 мс. Схема оновлення логічно збільшує розміри мікросхем DRAM, але це зазвичай не є критичною проблемою, оскільки окремі пристрої DRAM містять набагато більше пам'яті, ніж пристрої SRAM, і, таким чином, відносна частка схем оновлення є занадто стерпною. Аналогічна схема керування існує також в EEPROM та флеш-пам'яті для керування процесом стирання та запису за високої напруги.

Під час проектування підсистеми оперативної пам'яті для певної програми реального часу існує основне емпіричне правило: якщо вам потрібен великий обсяг пам'яті, використовуйте DRAM; але якщо ваші потреби в пам'яті не більше ніж помірні, рекомендованою альтернативою є SRAM, особливо для невеликих вбудованих систем. Тим не менш, практика не завжди така проста, оскільки може існувати так званий розрив між процесором та пам'яттю — «зростаюча невідповідність між затримкою пам'яті та пропускною здатністю, необхідною для ефективного виконання інструкцій, та затримкою та пропускною здатністю, які фактично може забезпечити зовнішня система пам'яті» [10]

Іншими словами, найшвидший цикл шини процесора може бути (набагато) коротшим за мінімальний час доступу до доступних компонентів пам'яті. Якщо це так, процесор не може працювати на повній швидкості під час доступу до повільнішої пам'яті. Це створює вузьке місце між процесором та

пам'яттю у високопродуктивних програмах; і це можна усунути за допомогою ієрархічної організації пам'яті. У програмах з низькою продуктивністю можливий конфлікт можна подолати, просто збільшивши тривалість циклу шини, щоб вона відповідала специфікаціям часу доступу компонентів пам'яті.

Проблеми з доступом до пам'яті та макетуванням

Принципи доступу до пам'яті тісно пов'язані зі специфічним обчислювальним обладнанням. Тим не менш, їх не можуть ігнорувати навіть розробники програмного забезпечення реального часу або системні інженери. Недостатньо знати лише архітектуру та пікову продуктивність процесора через вузьке місце між процесором та пам'яттю, згадане вище. Досить часто системна шина не працює на повній швидкості через обмеження, встановлені часом доступу до пам'яті. Це негативно впливає на час відгуку системи реального часу. Час доступу для читання з пам'яті - це суттєва затримка часу між активацією адресованого компонента пам'яті та наявністю запитуваних даних на шині даних. Це проілюстровано на часовій діаграмі на рисунку 2.6, яка використовує сигнали загального компонента пам'яті (рис. 2.5). Час доступу для запису в пам'ять визначається відповідно. Типові цикли читання та запису містять встановлення з'єднання між процесором та пристроєм пам'яті. А час завершення встановлення з'єднання залежить від електричних -характеристик процесора, системної шини та пристрою пам'яті.

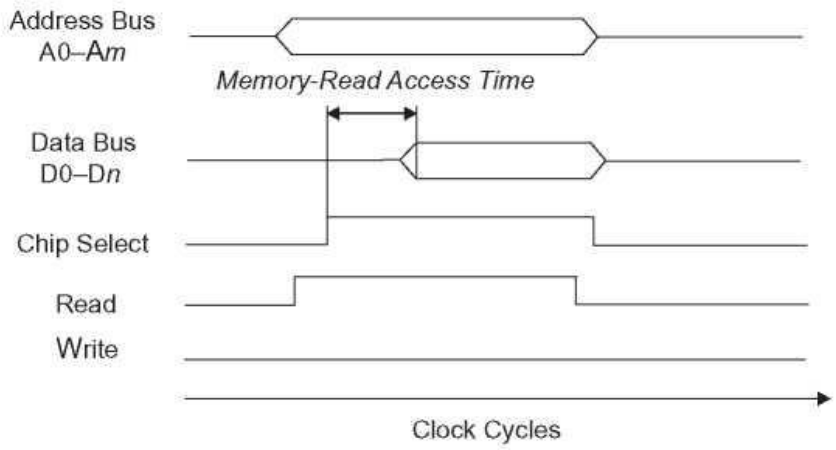

Рисунок 2.6 - Діаграма синхронізації циклу шини читання з пам'яті.

Кути “< >”, показані на шинах даних та адреси, вказують на те, що протягом цього періоду задіяно кілька ліній з різними логічними станами.

Визначаючи тривалість відповідного циклу шини, нам потрібно знати найгірші часи доступу до пам'яті та портів вводу/виводу, а також затримки схеми декодування адрес та можливі буфери на системній шині. За допомогою протоколу синхронної шини можна додавати стани очікування (або додаткові тактові цикли) до циклу шини за замовчуванням та динамічно адаптувати його до можливих різних часів доступу до пам'яті та компонентів вводу/виводу.

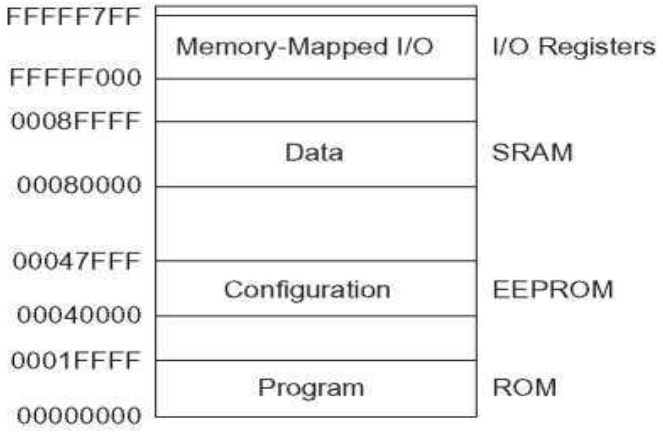

Рисунок 2.7 - Типова карта пам'яті, що показує виділені області (без масштабу).

Зверніть увагу, що велика частина простору пам'яті не виділена для жодної конкретної мети.

Ще одним критичним обмеженням іноді може бути загальне енергоспоживання апаратного забезпечення реального часу, яке зростає зі збільшенням тактової частоти процесора. З цієї точки зору, тактова частота повинна бути якомога нижчою, а пам'ять якомога повільнішою. Таким чином, відповідна тривалість циклу шини є параметром, специфічним для застосування, - можливо, критичним у вбудованих системах з батарейним живленням.

Для розробника програмного забезпечення реального часу розташування або карта пам'яті та вводу/виводу має велике значення. Розглянемо, наприклад, 16-бітний вбудований мікропроцесор, який підтримує 32-бітний адресний

простір, організований, як показано на рисунку 2.7. Ці початкова та кінцева адреси є довільними, але можуть бути репрезентативними для конкретної вбудованої системи. Наприклад, така карта може відповідати організації пам'яті контролера ліфта.

У нашому уявному контролері ліфта виконуваний програмний код знаходиться за адресами пам'яті від 00000000 до 0001FFFF у шістнадцятковій системі представлення. Ця стандартна система керування має настільки високий обсяг виробництва, що практично використовувати пристрої ROM, програмовані за маскою (128 тис. слів). Різні дані конфігурації, можливо, пов'язані з різними заводськими налаштуваннями та параметрами, специфічними для установки, зберігаються в місцях від 00040000 до 00047FFF в EEPROM (32 тис. слів), які можна перезаписати під час обслуговування або технічного обслуговування. Місця від 00080000 до 0008FFFF - це пам'ять SRAM (64 тис. слів), і вони використовуються для структур даних операційної системи реального часу та зберігання даних загального призначення. Нарешті, верхні місця від FFFFF000 до FFFFF7FF (2 тис. місць пам'яті) містять адреси, пов'язані з інтерфейсними модулями, доступ до яких здійснюється через відображення в пам'яті вводу/виводу, такі як паралельні входи та виходи для різних сигналів стану та команд; з’єднання польової шини для послідовного зв’язку з диспетчером групи та автомобільним комп’ютером; інтерфейс RS-232C для сервісного терміналу; а також функції годинника реального часу та таймера/лічильника. Ця карта пам’яті фіксує вільно переміщувані адреси системи та прикладного програмного забезпечення у фізичному апаратному середовищі.

Перш ніж перейти до важливого обговорення ієрархічних архітектур пам'яті , корисно розглянути деякі категорії DRAM відповідно до режиму доступу до даних . Хоча базовий пристрій DRAM призначений для випадкового доступу, різні модулі DRAM пропонують значне покращення продуктивності в спеціальних режимах доступу до даних, розроблених для швидкого доступу до послідовних комірок пам'яті. Ці типи пристроїв використовують такі методи, як доступ до рядків, конвеєрний доступ, синхронізований інтерфейс та чергування

доступу , для скорочення розриву між процесором та пам'яттю. Ці розширені режими є дуже цінними, коли організація пам'яті є ієрархічною, і потрібне швидке завантаження блоків даних у кеш-пам'ять з основної пам'яті на базі DRAM. З іншого боку, вражаюче покращення швидкості фактично не досягається, якщо доступ до розширених модулів DRAM здійснюється випадковим чином. Таким чином, модулі DRAM використовуються здебільшого в середовищах робочих станцій, де потрібна велика основна пам'ять. Нижче наведено приклад шляху еволюції модулів DRAM з розширеними режимами доступу в порядку їх появи:

- DRAM у режимі швидкого сторінкування (FPM)
- Розширена пам'ять виводу даних (EDO) DRAM
- Синхронна DRAM (SDRAM)
- Пряма Rambus DRAM (DRDRAM)
- Синхронна DRAM з подвійною швидкістю передачі даних 3 (DDR3 SDRAM)

У таких системах реального часу, що реалізуються на звичайних офісних або більш надійних промислових ПК, вдосконалені модулі DRAM природно використовуються на рівні основної пам'яті. Типові застосування включають централізовану систему моніторингу для групи ліфтів та розподілену систему бронювання авіаквитків. За певних обставин найдосконаліші модулі DRAM можуть пропонувати мінімальний час доступу , порівнянний з часом швидких SRAM.

2.3 Архітектурні розвитки

Архітектури процесорів значно розвинулися з моменту появи перших мікропроцесорів. Обмеження послідовного циклу команд базової архітектури фон Неймана призвели до розвитку різних архітектурних удосконалень. Більшість цих удосконалень побудовані на припущенні високої локальності посилання, яке є справедливим більшу частину часу з високою ймовірністю.

Хоча стабільний розвиток автоматизації проектування та технологій інтегральних схем дозволив розробляти та інтегрувати все більше функціональних можливостей в один чіп, новатори в галузі архітектури використали цю можливість для впровадження нових форм паралелізму в обробку команд. Таким чином, розуміння передових комп'ютерних архітектур є важливим для системного інженера реального часу. Хоча ми не маємо на меті надавати вичерпний огляд комп'ютерних архітектур, обговорення найважливіших питань є необхідним.

У розділі 2.1 ми представили послідовний цикл інструкцій: інструкція вибірки (F), інструкція декодування (D), завантаження операнда (L), виконання функції АЛП (E) та збереження результату (S). Цей цикл інструкцій містить два види звернень до пам'яті: вибірку інструкцій та завантаження/зберігання даних . У класичній архітектурі фон Неймана, показаній на рисунку 2.1 або 2.4, фази F та L/S не є незалежними одна від одної, оскільки вони використовують одну системну шину. Тому в конвеєрних архітектурах, які будуть коротко розглянуті , було б корисно мати окремі шини для інструкцій та даних, щоб мати можливість одночасно виконувати фази F та L/S. З іншого боку, дві паралельні шини адреси/даних займають значну площу кристала, але це ціна, яку доводиться платити за покращену продуктивність . Така архітектура називається гарвардською архітектурою, і вона вперше стала популярною в цифрових сигнальних процесорах. Багато сучасних процесорів поєднують характеристики як гарвардської, так і фон-нейманівської архітектури: окремі кеші інструкцій та даних на кристалі мають інтерфейс гарвардського типу, тоді як загальна позачіпова кеш-пам'ять підключена через єдину системну шину. Таким чином, цей вид гібридної архітектури внутрішньо є гарвардською, але зовні фон-нейманівською (тобто принстонською).

У архітектурі Гарварда можливо мати різну ширину шини для передачі інструкцій та даних. Наприклад, шина інструкцій може мати ширину 32 біти, а шина даних - лише 16 бітів. Більше того, шина адреси інструкцій може мати 20 бітів, а шина адреси даних 24 біти. Це означатиме 2 М слів пам'яті інструкцій та

16 М слів пам'яті даних. Отже, розробник архітектури має гнучкість при визначенні структур шини. На рисунку 2.9 зображено архітектуру Гарварда з можливостями паралельного виконання інструкцій та доступу до даних.

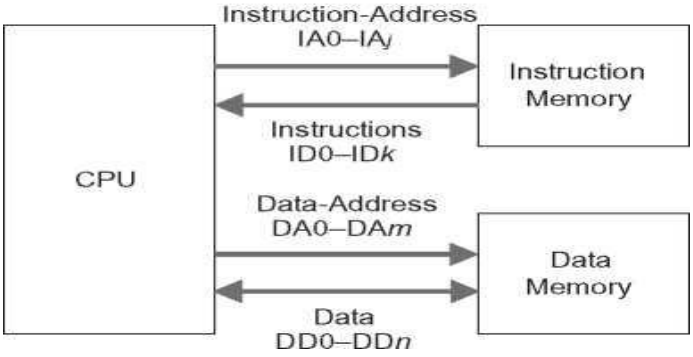

Рисунок 2.8 - Архітектура Гарварду з різною шириною шини.

Це навіть можна розглядати як потенційне полегшення вузького місця між процесором та пам'яттю; але це не той спосіб, яким зараз використовується архітектура Гарварду. Сьогодні як архітектура Гарварда, так і архітектура фон Неймана включають низку вдосконалень, що підвищують рівень паралелізму в обробці інструкцій. Найважливіші архітектурні вдосконалення обговорюються нижче. Незважаючи на їхні значні переваги в середньому випадку, вони зазвичай погіршують здатність прогнозування часу та продуктивність у найгіршому випадку, як обговорювалося в [12].

Конвеєрна обробка інструкцій

Конвеєрна обробка забезпечує неявний паралелізм виконання на різних фазах обробки інструкції, і тому має на меті збільшити продуктивність інструкції . Припустимо, що виконання інструкції складається з п'яти фаз, обговорених вище (FDLES). У послідовному (неконвеєрному) виконанні, запропонованому в розділі 2.1, одну інструкцію можна обробляти в одній фазі за раз. Завдяки конвеєрній обробці кілька інструкцій можна обробляти одночасно на різних фазах, що відповідно покращує продуктивність процесора.

Наприклад, розглянемо п'ятиетапний конвеєр, зображений на рисунку 2.10. На верхньому зображенні показано послідовне виконання фаз вибірки,

декодування, завантаження, виконання та збереження двох інструкцій, що вимагає 10 тактів. Під цією послідовністю знаходиться ще один набір тих самих двох інструкцій, плюс ще чотири інструкції , з перекриваючою обробкою окремих фаз FDLES. Цей конвеєр працює ідеально, якщо всі фази інструкцій мають однакову довжину, і кожна інструкція потребує однакового часу для виконання. Якщо припустити, що один етап конвеєра займає один такт, перші дві інструкції виконуються лише за шість тактів, а решта інструкцій виконуються -протягом 10 тактів. За ідеальних умов з безперервно заповненим конвеєром нова інструкція виконується зі швидкістю один такт. Загалом, найкращий можливий час виконання інструкції N- етапного конвеєра дорівнює 1/N, помноженому на час виконання неконвеєрного випадку.

Отже, АЛП та інші ресурси процесора використовуються ефективніше. Однак слід зазначити, що конвеєрна архітектура вимагає буферних регістрів між різними етапами обробки інструкцій. Це призводить до додаткової затримки конвеєрного циклу інструкцій порівняно з неконвеєрним циклом, де переходи від однієї фази до іншої можуть бути прямими без проміжного запису та читання буфера.

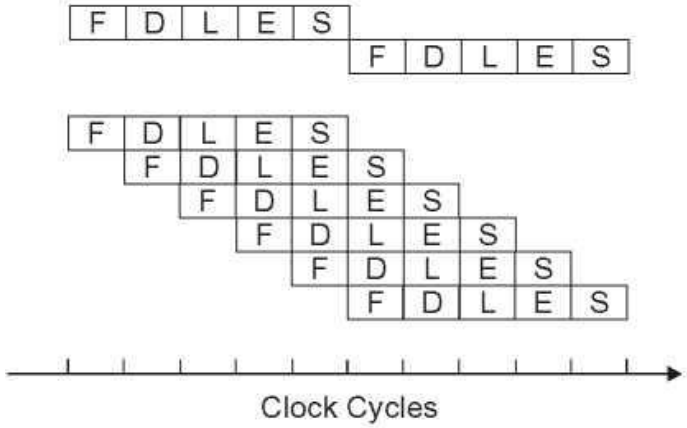

Рисунок 2.9 -. Конвеєрна обробка інструкцій у п'ятиетапному конвеєрі.

Ще одним недоліком конвеєрної обробки є те, що вона може фактично погіршити продуктивність у певних ситуаціях. Конвеєрна обробка є формою спекулятивного виконання, оскільки інструкції, які попередньо вибираються, вважаються наступними послідовними інструкціями. Спекулятивне виконання

добре працює, якщо локальність посилання залишається високою. Якщо інструкція в конвеєрі є умовною інструкцією перехідного виконання , наступні інструкції в конвеєрі можуть бути недійсними, і конвеєр необхідно очищати (всі регістри та прапорці конвеєра скидаються) та поповнювати поетапно. Щоб уникнути ймовірнісного негативного ефекту очищення/повторного заповнення конвеєра, багато процесорів мають розширені можливості прогнозування перехідних виконання та спекуляції. Подібна, але непередбачувана, ситуація виникає із зовнішніми перериваннями. Крім того, залежності даних та вхідних даних між послідовними інструкціями машинної мови можуть уповільнювати протікання конвеєра, вимагаючи тимчасових зупинок або втрачених тактів.

Конвеєри вищого рівня, або суперконвеєри, можна побудувати, якщо - цикл інструкцій додатково розкласти. Наприклад, можна побудувати шестиетапний конвеєр, що складається з етапу вибірки, двох етапів декодування (необхідних для підтримки режимів непрямої адресації), етапу виконання, етапу зворотного запису (який знаходить завершені операції в буфері та звільняє відповідні функціональні блоки) та етапу фіксації (на якому перевірені результати записуються назад у пам'ять). На практиці існують суперконвеєри з набагато більшою кількістю етапів у високопродуктивних процесорах з тактовими частотами рівня ГГц. Суперконвеєри з короткими етапами пропонують, в принципі, короткі затримки переривань. Однак ця потенційна перевага зазвичай прихована за неминучими промахами кешу та необхідним очищенням/повторним заповненням конвеєра, коли локальність посилання на інструкцію серйозно порушується. Таким чином, велика конвеєрна обробка є джерелом значного недетермінізму в системах реального часу.

Архітектури суперскалярних та дуже довгих слів інструкцій

Суперскалярні архітектури ще більше підвищують рівень спекуляцій в - обробці інструкцій. Вони мають щонайменше два паралельні конвеєри для покращення пропускної здатності інструкцій. Один з цих конвеєрів може бути зарезервований лише для інструкцій з плаваючою комою, тоді як усі інші інструкції обробляються в окремому конвеєрі або навіть у кількох конвеєрах.

Рисунок 2.10 ілюструє роботу двох суперскалярних конвеєрів з п'ятьма етапами. Ці конвеєри підтримуються високонадлишковими АЛП та іншими апаратними ресурсами. Теоретично, час виконання інструкцій в K-конвеєрній архітектурі з N-етапними конвеєрами може бути всього в 1/(K • N) разів більшим за час виконання в неконвеєрному випадку, отже, за один такт може бути виконано більше однієї інструкції.

Така паралельна схема працювала б добре, якби виконувані інструкції були повністю незалежними одна від одної, а здатність прогнозування розгалужень була ідеальною. Тим не менш, це зазвичай не так з реальними програмами, і тому середній коефіцієнт використання паралельних ресурсів далекий від 100%. Якщо порівнювати з архітектурою з одним конвеєром або суперконвеєром, багатоконвеєрний -процесор має ще більшу різницю між найкращою та найгіршою продуктивністю.

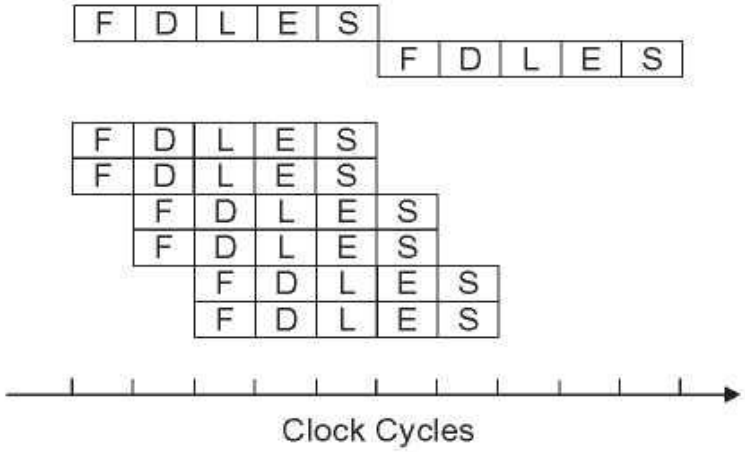

Рисунок 2.10 - Суперскалярна архітектура з двома паралельними конвеєрами інструкцій.

Суперскалярні процесори є складними реалізаціями інтегральних схем не лише тому, що вони мають значну функціональну надлишковість, але й через складну логіку перевірки взаємозалежності та диспетчеризації. Складність апаратного забезпечення може зростати, якщо для максимізації коефіцієнта використання дорогих ресурсів АЛП використовується виконання інструкцій поза порядком. Таким чином, суперскалярні процесори в основному

використовуються на робочих станціях та в програмах, що не працюють у реальному часі. Важко побудувати детерміновану вбудовану систему на суперскалярній платформі, хоча вона може запропонувати дуже високу пікову продуктивність.

Архітектура дуже довгих слів інструкцій (VLIW) подібна до суперскалярної архітектури тим, що обидві мають значну апаратну резервованість для підтримки паралельної обробки інструкцій. Однак існує фундаментальна різниця в процесі перевірки взаємозалежності між послідовними інструкціями та їх оптимального розподілу між відповідними функціональними блоками. У той час як суперскалярна архітектура повністю спирається на апаратну (онлайн) перевірку та розподіл залежностей, архітектура VLIW не потребує жодних апаратних ресурсів для цих цілей. Компілятори мов високого рівня процесорів VLIW обробляють як завдання перевірки залежностей, так і розподілу залежностей в автономному режимі, а дуже довгі коди інструкцій (зазвичай щонайменше 64 біти) складаються з кількох звичайних кодів інструкцій. Оскільки можна комбінувати лише взаємно незалежні інструкції, будь-які дві, що звертаються до шини даних, не можуть. В архітектурах VLIW немає онлайн-спекуляцій щодо розподілу інструкцій, але поведінка обробки інструкцій добре передбачувана.

Слід зазначити, однак, що ефективність архітектури VLIW залежить виключно від можливостей розширеного компілятора та властивостей власного набору інструкцій. Підтримка компілятором процесорів VLIW досліджується у роботі Ян та Чжан (2008). Процес генерації коду компілятором стає дуже складним через низку цілей паралелізації та обмежень взаємозалежності. Тому програміст додатків повинен допомагати компілятору у складній задачі диспетчеризації, адаптуючи критичні алгоритми до конкретної платформи VLIW. Загалом, програми, написані для одного процесора VLIW, досить погано переносяться в інші середовища VLIW. У той час як суперскалярні процесори використовуються в обчислювальних програмах загального призначення, процесори VLIW зазвичай налаштовуються для певного класу програм, таких як

обробка мультимедіа, і вони використовуються навіть у системах реального часу.

2.4 Висновки по розділі

Розуміння базової архітектури процесора, обробки інструкцій та організації пам'яті є критично важливим для інженерів систем реальної години, оскільки саме ці фактори визначають затримки, пропускну здатність та можливість дотримання жорстких часових обмежень. Ефективне поєднання апаратних та програмних рішень, зокрема використання оптимізованих архітектур та режимів енергозбереження, забезпечує продуктивність та надійність вбудованих систем у реальному часі.

Сучасні архітектурні та пам'ятні технології є ключовими для ефективної роботи систем реального часу, оскільки час доступу до пам'яті та структура процесора безпосередньо впливають на їх затримку та надежність.

РОЗДІЛ 3. МОВИ ПРОГРАМУВАННЯ ДЛЯ СИСТЕМ РЕАЛЬНОГО ЧАСУ

3.1 Кодування програмного забезпечення реального часу

Неправильне використання базової мови програмування може бути найбільшою причиною погіршення продуктивності та пропущених термінів у системах реального часу. Більше того, під час використання об'єктно-орієнтованих мов у системах реального часу такі проблеми з продуктивністю може бути складніше аналізувати та контролювати. Тим не менш, об'єктно-орієнтовані мови неухильно витісняють процедурні мови як мову вибору в розробці вбудованих систем реального часу. На рисунку 3.1 зображено основне використання мов програмування у вбудованих додатках реального часу з 1970-х років до поточного десятиліття.

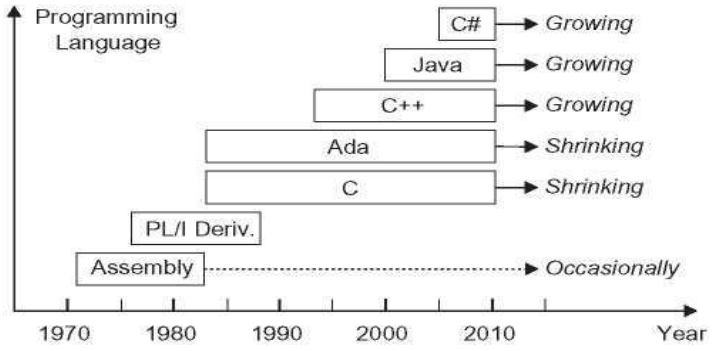

Рисунок 3.1 - Основне використання мов програмування реального часу за роками (межі років є приблизними).

Придатність мови програмування для застосувань реального часу

Мова програмування являє собою зв'язок між дизайном та структурою. Отже, оскільки фактична «збірка» програмного забезпечення залежить від інструментів для компіляції, генерації двійкового коду, компонування та створення двійкових об'єктів, «кодування» повинно займати пропорційно - менше часу, ніж зусилля з розробки вимог та дизайну. Тим не менш, «кодування»

(синонім «програмування» та «написання програмного забезпечення») традиційно було більше схожим на ремесло, ніж на виробництво, і, як і в будь-якому ремеслі, найкращі фахівці відомі якістю своїх інструментів та пов'язаними з ними навичками їх ефективного використання.

Основним інструментом у процесі генерації коду є компілятор мови. Системи реального часу зараз створюються за допомогою різноманітних мов програмування [16], включаючи різні діалекти C, C++, C#, Java, Ada, мови асемблера та навіть Fortran або Visual Basic. З цього неоднорідного списку C++, C# та Java є об'єктно-орієнтованими, тоді як інші є процедурними. Однак слід зазначити, що C++ можна зловживати таким чином, що всі переваги об'єктно-орієнтованого програмування втрачаються (наприклад, шляхом вбудовування старої програми на C в один клас "Бога"). Крім того, Ada 95 має елементи як об'єктно-орієнтованих, так і процедурних мов, і тому може використовуватися в будь-якому випадку, залежно від навичок та уподобань програміста, а також локальних політик проекту.

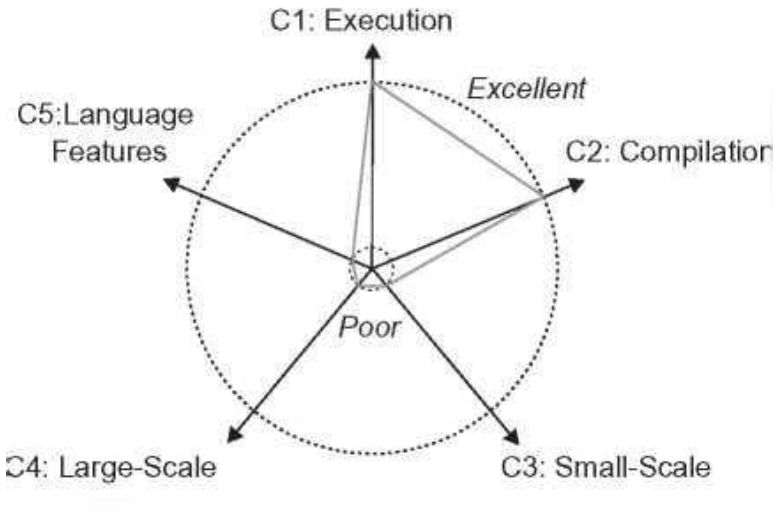

Рисунок 3.2 - П'ятикутна діаграма для ілюстрації критеріїв

Карделлі для різних економік (сірий п'ятикутник відповідає мові асемблера).

Кожна мова програмування, безсумнівно, має свої сильні та слабкі сторони стосовно систем реального часу, і ці якісні критерії, C1-C5, можна

використовувати для калібрування характеристик конкретної мови для порівняння «апельсинів з апельсинами» в межах заданої програми. Критерії Карделлі можна проілюструвати за допомогою діаграми пентакля [19], показаної на рисунку. 3.2 . Такі діаграми забезпечують простий спосіб візуального порівняння мов програмування-кандидатів.

У цьому розділі ми не маємо наміру проводити вичерпний огляд мов програмування; натомість ми зосереджуємося на тих функціях мови, які можна використовувати для мінімізації кінцевого часу виконання коду та які підходять для прогнозування продуктивності. Прогнозування продуктивності виконання під час компіляції безпосередньо підтримує аналіз планування. Під час розробки спеціальних мов програмування реального часу акцент робиться на усуненні тих конструкцій, які роблять мову неможливою для аналізу, наприклад, необмежена рекурсія та необмежені цикли while . Більшість так званих «мов реального часу» прагнуть усунути все це. З іншого боку, коли для програмування в реальному часі використовуються основні мови, певні проблемні структури коду можуть бути просто заборонені стандартами кодування.

Стандарти кодування для програмного забезпечення реального часу

Стандарти кодування [4] відрізняються від мовних стандартів. Мовний стандарт, наприклад, C++ ANSI/ISO/IEC 14882:2003, втілює синтаксичні правила мови програмування C++. Вихідна програма, що -порушує будь-яке з цих правил, буде відхилена компілятором. Стандарт кодування, з іншого боку, є набором стилістичних умовностей або «найкращих практик». Порушення цих умовностей не призведе до відхилення компілятором. В іншому сенсі, дотримання мовних стандартів є обов'язковим, тоді як дотримання стандартів кодування є, принаймні в принципі, добровільним.

Дотримання мовних стандартів сприяє переносимості між різними компіляторами , а отже, і апаратними середовищами. З іншого боку, дотримання стандартів кодування не сприятиме переносимості, а в багатьох випадках, радше читабельності, зручності обслуговування та повторного використання. Деякі практики навіть стверджують, що використання суворих стандартів кодування

може підвищити надійність програмного забезпечення. Стандарти кодування також можуть бути використані для покращення продуктивності шляхом заохочення або обов'язкового -використання певних мовних конструкцій, які, як відомо, генерують більш ефективний код. Багато гнучких методологій, наприклад, eXtreme Programming [18] використовують спеціальні стандарти кодування.

Стандарти кодування зазвичай передбачають стандартизацію деяких або всіх наступних елементів використання мови програмування :

- Формат заголовка
- Частота, довжина та стиль коментарів
- Іменування класів, даних, файлів, методів, процедур, змінних тощо
- Форматування вихідного коду програми, включаючи використання пробілів та відступів
- Обмеження розміру одиниць коду, включаючи максимальну та мінімальну кількість рядків коду та кількість використаних методів
- Правила вибору мовної конструкції, яка буде використовуватися; наприклад, коли використовувати оператори case замість вкладених операторів if-then-else

Хоча неясно, чи сприяє дотримання цих правил значному підвищенню надійності, очевидно, що суворе дотримання може зробити програми легшими для читання та розуміння, а отже, ймовірно, більш зручними для повторного використання та підтримки [3]. Існує багато різних стандартів кодування, які є або мовно-незалежними, або специфічними для певної мови. Стандарти кодування можуть бути корпоративними, командними, специфічними для певної групи користувачів (наприклад, група розробників програмного забезпечення GNU має стандарти для C та C++), або клієнти можуть вимагати відповідності певному власному стандарту. Крім того, інші стандарти стали загальнодоступними. Одним із прикладів є угорський стандарт нотації [9], названий на честь Чарльза Сімоньї, якому приписують перше поширення його використання. Угорська нотація - це стандарт загальнодоступного типу,

призначений для використання з об'єктно-орієнтованими мовами, зокрема C++. Стандарт використовує цілеспрямовану схему іменування для вбудовування інформації про тип об'єктів, методів, атрибутів та змінних в ім'я. Оскільки стандарт, по суті, надає набір правил щодо іменування різних ідентифікаторів, він може використовуватися і використовувався з іншими мовами, такими як Ada, Java і навіть C. Інший приклад - Java, яка, за домовленістю, використовує всі великі літери для констант, таких як PI та E. Більше того, деякі класи використовують символ підкреслення в кінці, щоб відрізнити атрибут, такий як x_ , від методу, такого як x() .

Загальна проблема зі стандартами стилю, такими як угорська нотація, полягає в тому, що вони можуть призвести до спотворених імен змінних і спрямовують увагу програміста на те, як називати змінні «угорською», а не на вибір змістовної назви змінної для її використання в коді. Іншими словами, бажання відповідати стандарту не завжди може призвести до особливо змістовної назви змінної. Ще одна проблема полягає в тому, що сама сила стандарту кодування може також бути його руйнівною. Наприклад, що робити, якщо в угорській нотації інформація про тип, -вбудована в назву об'єкта, насправді неправильна? Жоден компілятор не може розпізнати цю помилку. Існують комерційні майстри правил , що -нагадують інструмент перевірки мови C, lint, який можна налаштувати для забезпечення дотримання стандартів кодування, але він повинен бути запрограмований для роботи разом з компілятором. Крім того, вони можуть пропускати певні невідповідності, що призводить розробників до відчуття хибної впевненості. Зрештою, не рекомендується впроваджувати стандарти кодування на середині проекту. Легше та мотивуюче почати дотримуватися стандартів з самого початку, ніж потім змінювати існуючий стиль, щоб відповідати вимогам. Рішення про використання певного стандарту кодування є організаційним рішенням, яке вимагає значного попереднього обмірковування та відкритого обговорення.

3.2 Асемблераційна мова

У середині-кінці 1970-х років, коли перші мови програмування високого рівня стали доступними для мікропроцесорів, деякі викладачі коледжів казали своїм студентам, що «через п'ять років ніхто не писатиме реальні програми мовою асемблера». Однак, після більш ніж 30 років з тих днів, мова асемблера все ще відіграє обмежену , але постійну роль у програмуванні реального часу.

Які причини цього? Хоча мова асемблера не має зручності для користувача та функцій продуктивності, властивих мовам високого рівня, вона має особливу перевагу для використання в програмуванні реального часу; вона забезпечує найпряміший контроль над комп'ютерним обладнанням порівняно з мовами високого рівня. Ця перевага розширила використання мови асемблера в системах реального часу, незважаючи на те, що мова асемблера неструктурована та має дуже обмежені властивості абстракції. Більше того, синтаксис мови асемблера сильно відрізняється від процесора до процесора. Кодування мовою асемблера, як правило, займає багато часу для вивчення, є виснажливим та схильним до помилок. Зрештою, отриманий код нелегко перенести на різні процесори, тому використання мови асемблера у вбудованих системах реального часу – або в будь-якій професійній системі – не рекомендується. Покоління тому найкращі програмісти могли генерувати асемблерний код, який часто був ефективнішим, ніж код, згенерований процедурною мовою програмування.

З точки зору критеріїв Карделлі щодо різних економічних показників, мови асемблера мають чудову економічність виконання, а також порожню економічність компіляції, оскільки вони не компілюються. Однак мови асемблера мають низьку економічність як маломасштабної, так і великомасштабної розробки, а також мовних можливостей (див. рис. 3.2). Отже, програмування мовою асемблера має бути обмежене використанням у дуже стислих часових ситуаціях або для керування апаратними функціями, які не підтримуються компілятором. Роль мови асемблера, яка продовжує існувати в цьому десятилітті, підсумована нижче:

- Для певних видів коду, таких як обробники переривань та драйвери пристроїв для унікального обладнання, де необхідно мінімізувати «інтелектуальну відстань» між апаратним та програмним забезпеченням.
- Для ситуацій, коли передбачувану продуктивність коду надзвичайно важко або неможливо досягти через небажану взаємодію мови програмування та компілятора.
- Для ефективного використання всіх архітектурних особливостей процесора, наприклад, паралельних суматорів та помножувачів.
- Для написання коду з мінімальним часом виконання, що досягається для критично важливих застосувань, таких як складні алгоритми обробки сигналів з високою частотою дискретизації.
- Для написання всього програмного забезпечення для спеціально розроблених процесорів з невеликим набором інструкцій (див. розділ 2.5.3) - якщо підтримка мов високого рівня недоступна.
- Для налагодження складних проблем нижче рівня коду мови програмування високого рівня та трасування потоку отриманих інструкцій логічним аналізатором.
- Для навчання та вивчення архітектур комп'ютерів та внутрішньої роботи процесорів.

Щоб впоратися з цими особливими ситуаціями, розробник програмного забезпечення зазвичай пише оболонку програми мовою високого рівня та компілює код у проміжне представлення асемблера, яке потім налаштовується вручну для досягнення бажаного ефекту. Деякі мови програмування, такі як Ada, надають спосіб розміщення асемблерного коду в коді мови високого рівня. У будь-якому випадку, використання мови асемблера в реальній системі має здійснюватися неохоче та з надзвичайною обережністю.

3.3 Процедурні мови

Процедурні мови, такі як Ada, C, Fortran та Visual Basic, – це мови, в яких

дія програми визначається набором операцій, що виконуються послідовно. Ці мови характеризуються можливостями, які дозволяють групувати інструкції в процедури або модулі. Відповідна структурування процедур дозволяє досягти бажаних властивостей програмного забезпечення, наприклад, модульності, надійності та можливості повторного використання. Існує кілька особливостей мов програмування, що виділяються в процедурних мовах програмування та становлять інтерес для систем реального часу, зокрема:

- Модульність
- Стійкий текст
- Абстрактна типізація даних
- Універсальні механізми передачі параметрів
- Засоби динамічного розподілу пам'яті
- Обробка винятків

Ці мовні особливості, які будуть обговорені найближчим часом, допомагають просувати бажані властивості розробки програмного забезпечення та найкращі практики впровадження в реальному часі.

Проблеми модульності та типізації

Процедурні мови, що піддаються принципу приховування інформації, як правило, сприяють побудові високоінтегрованих систем реального часу. Хоча C та Fortran мають механізми, які можуть підтримувати приховування інформації (процедури та підпрограми), інші мови, такі як Ada, як правило, сприяють більш модульному проектуванню через вимогу мати чітко визначені вхідні та вихідні дані у списках параметрів модуля.

В мові Ada поняття пакета вишукано втілює концепцію приховування інформації Парнаса [7]. Пакет Ada складається зі специфікації та оголошень, які включають його публічний або видимий інтерфейс та його приватні або невидимі елементи. Крім того, тіло пакета, яке має більше зовні невидимих компонентів, містить робочий код пакета. Окремі пакети є окремо компільованими сутностями, що ще більше розширює їхнє застосування як чорних скриньок. Крім того, мова C передбачає окремо -компільовані модулі та

інші функції, що сприяють суворому низхідному підходу до проектування, який має призвести до надійної модульної конструкції. Хоча модульне програмне забезпечення є бажаним з багатьох причин, воно має свою ціну у вигляді накладних витрат, пов'язаних з викликами процедур та передачею важливих параметрів. Цей негативний ефект слід ретельно враховувати під час визначення розміру модулів.

Типізовані мови вимагають, щоб кожна змінна та константа були певного типу (наприклад, логічного, цілочисельного або дійсного), і щоб кожна з них була оголошена як така перед використанням. Мови із строгою типізацією забороняють змішування різних типів в операціях та присвоєннях, і таким чином змушують програміста точно визначати, як мають оброблятися дані. Точна типізація може запобігти пошкодженню даних через небажане або непотрібне перетворення типів. Крім того, перевірка типів компілятором є важливим кроком для пошуку помилок під час компіляції, а не під час виконання, коли їх виправлення є більш витратним. Отже, мови із строгою типізацією дійсно бажані для систем реального часу.

Зазвичай, мови програмування високого рівня пропонують цілочисельні та дійсні типи, а також логічні, символьні та рядкові типи. У деяких випадках також підтримуються абстрактні типи даних . Це дозволяє програмістам визначати власні типи разом із пов'язаними з ними операціями. Однак використання абстрактних типів даних може призвести до зменшення часу виконання, оскільки для підтримки абстракції часто потрібні складні внутрішні представлення. Деякі мови є типізованими, але не забороняють змішування типів в арифметичних операціях. Оскільки ці мови зазвичай виконують змішані обчислення, використовуючи тип, який має найвищу складність зберігання , вони повинні підвищувати всі змінні до цього найвищого типу.

Передача параметрів та динамічне виділення пам'яті

Існує кілька методів передачі параметрів , включаючи використання - списків параметрів та глобальних змінних. Хоча кожен із цих методів має переважні способи використання, кожен також має різний вплив на

продуктивність. Зауважте, що ці механізми передачі параметрів також зустрічаються в об'єктно-орієнтованих мовах програмування.

Двома найпоширенішими методами передачі параметрів є виклик за значенням та виклик за посиланням. При передачі параметрів за значенням значення фактичного параметра у виклику процедури копіюється у формальний параметр процедури. Оскільки процедура маніпулює лише формальним параметром, фактичний параметр не змінюється. Цей метод корисний або під час виконання тесту, або коли вихідний результат є функцією вхідних параметрів. Наприклад, під час передачі показників акселерометра з циклу 10 мс у цикл 40 мс, необроблені дані не потрібно повертати до викликаючої процедури у зміненому вигляді. Коли параметри передаються за допомогою виклику за значенням, вони копіюються на стек виконання, що призводить до додаткових витрат часу виконання.

При виклику за посиланням (або виклику за адресою) адреса параметра передається викликаючою процедурою викликаній процедурі, щоб там можна було змінити відповідний вміст пам'яті. Виконання процедури з використанням виклику за посиланням може тривати довше, ніж виконання процедури з використанням виклику за значенням, оскільки при виклику за посиланням для будь-яких операцій, що включають передані змінні, потрібні інструкції режиму непрямої адресації . Однак у випадку передачі великих структур даних, таких як буфери між процедурами, бажаніше використовувати виклик за посиланням, оскільки передача вказівника є ефективнішою, ніж передача даних побайтово.

Списки параметрів, ймовірно, сприятимуть модульному проектуванню, оскільки інтерфейси між модулями чітко визначені. Чітко визначені інтерфейси можуть зменшити потенціал невідстежуваного пошкодження даних процедурами, що використовують глобальний доступ. Однак, як методи передачі параметрів за значенням, так і за посиланням можуть впливати на продуктивність у реальному часі, коли списки довгі, оскільки переривання часто вимикаються під час передачі параметрів, щоб зберегти цілісність переданих даних. Крім того, виклик за посиланням може призвести до незначних побічних

ефектів функцій , залежно від компілятора.

Перш ніж визначитися з конкретним набором правил щодо передачі параметрів для оптимальної продуктивності, доцільно створити набір тестових випадків, які перевіряють різні альтернативи. Ці тестові випадки необхідно повторно запускати щоразу, коли змінюється компілятор, апаратне забезпечення або програма, щоб оновити правила. Глобальні змінні – це змінні, які знаходяться в межах області видимості всього коду. «В межах області видимості» зазвичай означає, що посилання на ці змінні можуть бути зроблені з мінімальним обсягом вибірки пам'яті для визначення цільової адреси, і таким чином вони швидші, ніж посилання на змінні, що передаються через списки параметрів, які потребують додаткових посилань на пам'ять. Наприклад, у багатьох програмах обробки зображень глобальні масиви визначені для представлення цілих зображень, що дозволяє уникнути дорогої передачі параметрів.

Однак глобальні змінні небезпечні, оскільки посилання на них можуть бути зроблені неавторизованим кодом, що потенційно може призвести до помилок, які важко ізолювати. Використання глобальних змінних також порушує принцип приховування інформації, що ускладнює розуміння та підтримку коду. Тому слід уникати непотрібного та безпідставного використання глобальних змінних. Передача глобальних параметрів рекомендується лише тоді, коли цього вимагають обмеження часу, або якщо використання списків параметрів призводить до заплутування коду. У будь-якому випадку використання глобальних змінних має бути суворо узгодженим та чітко задокументованим. Рішення використовувати один метод передачі параметрів або інший може являти собою компроміс між належною практикою розробки програмного забезпечення та потребами продуктивності. Наприклад, часто обмеження часу змушують використовувати глобальну передачу параметрів у випадках, коли списки параметрів були б кращими для ясності та зручності обслуговування.

Більшість мов програмування забезпечують рекурсію, оскільки процедура може або викликати саму себе, або використовувати себе у своїй

конструкції. Хоча рекурсія може бути елегантною та іноді необхідною, слід враховувати її негативний вплив на продуктивність у реальному часі. Виклики процедур вимагають виділення пам'яті на стеку для передачі параметрів та для зберігання локальних змінних. Час виконання, необхідний для виділення та звільнення пам'яті, а також для зберігання та отримання цих параметрів та локальних змінних, може бути дорогим. Крім того, рекурсія вимагає використання великої кількості дорогих інструкцій, непрямих до пам'яті та регістрів. Крім того, необхідно вжити запобіжних заходів, щоб гарантувати, що рекурсивна процедура завершиться, інакше стек виконання зрештою переповниться. Використання рекурсії часто унеможливлює визначення точного розміру вимог до пам'яті виконання. Таким чином, ітераційні методи, такі як цикли while та for , повинні використовуватися там, де продуктивність та детермінізм є критично важливими, або, природно, у тих мовах, які не підтримують рекурсію.

Можливість динамічного розподілу пам'яті важлива для побудови та підтримки багатьох структур даних, необхідних у системах реального часу. Хоча динамічний розподіл пам'яті може займати багато часу, він необхідний, особливо для побудови обробників переривань, менеджерів пам'яті тощо. Зв'язані списки, дерева, купи та інші динамічні структури даних можуть виграти від ясності та економії, що вносяться динамічним розподілом пам'яті. Крім того, у випадках, коли для передачі структури даних використовується лише вказівник, накладні витрати на динамічний розподіл можуть бути розумними. Однак, під час кодування систем реального часу слід подбати про те, щоб компілятор завжди передавав вказівники на великі структури даних, а не самі структури даних. Мови, які не дозволяють динамічного розподілу пам'яті, наприклад, деякі примітивні мови високого рівня або мова асемблера, вимагають структур даних фіксованого розміру. Хоча це може бути швидше, гнучкість приноситься в жертву, а вимоги до пам'яті мають бути визначені заздалегідь. Сучасні процедурні мови, такі як Ada, C та Fortran 2003, мають можливості динамічного розподілу.

Метрики та процедурні мови Карделлі

Розглядаючи загальний набір процедурних мов як єдине ціле, Карделлі розглядав їх для використання в системах реального часу відповідно до своїх критеріїв. Його коментарі перефразовані у попередньому обговоренні. По-перше, він зазначає, що типізація змінних була введена для покращення генерації коду. Отже, економічність виконання є високою для процедурних мов за умови ефективності компілятора. Крім того, оскільки модулі можна компілювати незалежно, компіляція великих систем є ефективною, принаймні, коли інтерфейси стабільні. Таким чином, усуваються більш складні аспекти системної інтеграції.

Розробка в невеликому масштабі є економічно вигідною, оскільки перевірка типів може виявляти багато помилок кодування, що зменшує зусилля на тестування та налагодження. Помилки, які трапляються, легше налагоджувати, просто тому, що великі класи інших помилок виключені. Зрештою, досвідчені програмісти зазвичай застосовують стиль кодування, який призводить до того, що деякі логічні помилки відображаються як помилки перевірки типів; отже, вони можуть використовувати перевірку типів як інструмент розробки. Наприклад, зміна назви типу, коли його інваріанти змінюються, навіть якщо структура типу залишається незмінною, призводить до звітів про помилки щодо всіх його попередніх використань.

Більше того, абстракція даних та модуляризація мають методологічні переваги для розробки великомасштабного коду. Великі команди програмістів можуть узгоджувати інтерфейси, що будуть реалізовані, а потім окремо продовжувати реалізацію відповідних фрагментів коду. Залежності між такими фрагментами коду мінімізуються, і код можна локально перебудовувати без будь-яких побоювань глобальних наслідків. Зрештою, процедурні мови є економічними, оскільки деякі добре розроблені конструкції можна природним чином компонувати ортогональним чином. Наприклад, у мові С масив масивів моделює двовимірні масиви. Ортогональність мовних особливостей зменшує складність мови програмування. Таким чином, крива навчання для програмістів

зменшується , а зусилля на повторне навчання, які постійно необхідні для використання складних мов, мінімізуються [20].

3.4 Об'єктно-орієнтовані мови

Переваги об'єктно-орієнтованих мов, такі як підвищена продуктивність програмістів, підвищена надійність програмного забезпечення та вищий потенціал для повторного використання коду, добре відомі та цінуються. До об'єктно-орієнтованих мов належать Ada, C++, C# та Java. Формально об'єктно-орієнтовані мови програмування - це ті, що підтримують абстракцію даних, успадкування, поліморфізм та обмін повідомленнями.

Об'єкти є ефективним способом керування зростаючою складністю - систем реального часу, оскільки вони забезпечують природне середовище для приховування інформації або захищеної варіації та інкапсуляції. В інкапсуляції клас об'єктів та пов'язані з ними методи вкладаються або інкапсулюються у визначення класів . Об'єкт може використовувати інкапсульовані дані іншого об'єкта, лише надіславши цьому об'єкту повідомлення з назвою методу, який потрібно застосувати. Наприклад, розглянемо проблему сортування об'єктів. Може існувати метод для сортування класу об'єктів цілих чисел у порядку зростання. Клас людей може бути відсортований за їхнім зростом. Клас об'єктів зображень, що має атрибут кольору, може бути відсортований за цим атрибутом. Всі ці об'єкти мають метод повідомлення порівняння з різними реалізаціями. Тому, якщо клієнт надсилає повідомлення для порівняння одного з цих об'єктів з іншим, код середовища виконання повинен вирішити, який метод застосовувати динамічно, що з очевидним зниженням часу виконання. Це питання буде обговорено найближчим часом.

Об'єктно-орієнтовані мови програмування забезпечують плідне середовище для приховування інформації; наприклад, у системах обробки зображень може бути корисним визначити клас типу піксель з атрибутами, що описують його положення, колір та яскравість , а також операціями, які можна

застосовувати до пікселя, такими як додавання, активація та деактивація. Також може бути бажаним визначити об'єкти типу зображення як набір пікселів з іншими атрибутами ширини, висоти тощо . У певних випадках вираження функціональності системи легше виконати об'єктно-орієнтованим способом.

Синхронізація об'єктів та збирання сміття

Замість розширення класів шляхом успадкування, на практиці часто краще -використовувати композицію. Однак, роблячи це, існує потреба підтримувати різні політики синхронізації для об'єктів через різні контексти використання. Зокрема, розглянемо такі поширені політики синхронізації для об'єктів:

- Синхронізовані об'єкти. Об'єкт синхронізації, такий як м'ютекс, пов'язаний з об'єктом, до якого можуть одночасно звертатися кілька потоків. Якщо використовується внутрішнє блокування, то при вході в метод кожен публічний метод отримує блокування на пов'язаному об'єкті синхронізації та знімає блокування при виході з методу. Якщо використовується зовнішнє блокування, то клієнти несуть відповідальність за отримання блокування на пов'язаному об'єкті синхронізації перед доступом до об'єкта та подальше зняття блокування після завершення.
- Інкапсульовані об'єкти. Коли об'єкт інкапсульовано всередині іншого об'єкта (тобто інкапсульований об'єкт недоступний поза межами об'єкта, що його оточує), отримання блокування на інкапсульованому об'єкті є зайвим, оскільки блокування об'єкта, що його оточує, також захищає інкапсульований об'єкт . Тому операції над інкапсульованими об'єктами не потребують синхронізації.
- Потік – Локальні об’єкти. Об’єкти, до яких звертається лише один потік, не потребують синхронізації.
- Міграція об'єктів між потоками. У цій політиці право власності на мігруючий об'єкт передається між потоками. Коли потік передає право власності на мігруючий об'єкт, він більше не може отримати до нього доступ. Коли потік отримує право власності на мігруючий об'єкт, йому гарантовано надається

ексклюзивний доступ до нього (тобто мігруючий об'єкт є локальним для потоку). Отже, мігруючі об'єкти не потребують синхронізації. Однак передача права власності вимагає синхронізації.

- Незмінні об'єкти. Стан незмінного об'єкта ніколи не може бути змінений після його створення. Таким чином, незмінні об'єкти не потребують синхронізації під час доступу до них кількома потоками, оскільки всі звернення доступні лише для читання.
- Несинхронізовані об'єкти. Об'єкти в однопотоковій програмі не потребують синхронізації.

Щоб проілюструвати необхідність підтримки параметризації політик синхронізації, розглянемо бібліотеку класів. Розробник бібліотеки класів хоче забезпечити якомога ширшу аудиторію для цієї бібліотеки, тому він синхронізує всі класи, щоб їх можна було безпечно використовувати як в однопотокових, так і в багатопотокових програмах. Однак клієнти бібліотеки, чиї програми є однопотоковими, надмірно караються непотрібним виконанням понад синхронізацію, яка їм не потрібна. Навіть багатопотокові програми можуть бути надмірно карані, якщо об'єкти не потребують синхронізації (наприклад, об'єкти є локальними для потоків). Тому, щоб сприяти повторному використанню бібліотеки класів без шкоди для продуктивності, класи в бібліотеці в ідеалі повинні дозволяти клієнтам вибирати для кожного об'єкта, яку політику синхронізації використовувати.

Сміття стосується виділеної пам'яті, яка більше не використовується, але й недоступна для інших цілей. Надмірне накопичення сміття може бути шкідливим , тому сміття необхідно регулярно відновлювати. Алгоритми збору сміття зазвичай мають непередбачувану продуктивність, хоча середня продуктивність може бути відома. Втрата детермінізму є результатом невідомої кількості сміття, часу тегування недетермінованих структур даних та того факту, що багато інкрементальних збирачів сміття вимагають, щоб кожне виділення або звільнення пам'яті з купи було готове обслуговувати обробник перехоплення помилок сторінок.

Крім того, сміття може створюватися як у процедурних, так і в об'єктно-орієнтованих мовах. Наприклад, у C сміття створюється шляхом виділення пам'яті, але не шляхом її належного звільнення. Тим не менш, сміття зазвичай асоціюється з об'єктно-орієнтованими мовами, такими як C++ та Java. Java примітна тим, що стандартне середовище включає збирання сміття, тоді як C++ цього не робить.

Об'єктно-орієнтовані та процедурні мови

Немає загальної згоди щодо того, що краще для систем реального часу — об'єктно-орієнтовані чи процедурні мови. Частково це пов'язано з тим, що існує величезна різноманітність програм реального часу - від невбудованих систем бронювання авіаквитків до вбудованих бездротових датчиків у працюючих системах.

Переваги об'єктно-орієнтованого підходу до вирішення проблем та використання об'єктно-орієнтованих мов очевидні та вже були описані. Більше того, можна уявити певні аспекти операційної системи реального часу, які виграли б від об'єктивації, такі як процес, потік, файл або пристрій. Крім того, певні області застосування можуть явно виграти від об'єктно-орієнтованого підходу. Однак основними аргументами проти об'єктно-орієнтованих мов програмування для систем реального часу є те, що вони можуть призвести до непередбачуваних та неефективних систем, а також те, що їх важко оптимізувати. Тим не менш, ми можемо впевнено рекомендувати об'єктно-орієнтовані мови для м'яких та стійких систем реального часу.

Однак аргумент про непередбачуваність важко захистити, принаймні стосовно об'єктно-орієнтованих мов, таких як C++, які не використовують збирання сміття . Цілком ймовірно, що передбачувану систему -також систему жорсткого реального часу - можна так само легко побудувати на C++, як і на C. Аналогічно, ймовірно, так само легко побудувати непередбачувану систему на C, як і на C++. Аргументи на користь більш непередбачуваних систем, що використовують об'єктно-орієнтовані мови, легше обґрунтувати, коли йдеться про мови збирання сміття, такі як Java.

У будь-якому випадку, аргумент неефективності проти об'єктно-орієнтованих мов є вагомим. Як правило, об'єктно-орієнтовані мови мають зниження часу виконання порівняно з процедурними мовами. Це зниження частково зумовлене пізнім зв'язуванням (визначення адрес пам'яті під час виконання, а не під час компіляції), що зумовлене поліморфізмом функцій, успадкуванням та композицією. Ці ефекти призводять до значних і часто невизначених факторів затримки. Ще одна проблема виникає через накладні витрати на процедури збору сміття. Один зі способів зменшити ці зниження - не визначати занадто багато класів і визначати лише ті класи, які містять грубі деталі та високорівневу функціональність.

3.5 Огляд мов програмування

Для ілюстрації вищезгаданих властивостей мови корисно розглянути деякі мови, які наразі використовуються в програмуванні систем реального часу. Вибрані процедурні та об'єктно-орієнтовані мови обговорюються в алфавітному порядку, а не в порядку схвалення чи визначальних властивостей.

Ада

Спочатку планувалося, що Ada стане обов'язковою мовою для всіх проектів Міністерства оборони США, які включали значну частку вбудованих систем. Перша версія, яка була стандартизована до 1983 року, мала досить серйозні проблеми. Ada призначалася для використання спеціально для програмування систем реального часу, але на той час розробники систем вважали отриманий виконуваний код громіздким та неефективним. Більше того, були виявлені серйозні проблеми під час спроби реалізації -багатозадачності за допомогою обмежених інструментів, що надаються мовою, таких як механізм зустрічей, що зазнавав різкої критики. Спільнота розробників мов програмування знала про ці проблеми і практично з першої ж поставки компілятора Ada 83 прагнула їх вирішити. Ці зусилля з реформування зрештою призвели до появи нової версії мови. Ретельно перероблена мова під назвою

«Ada 95» вважається першою міжнародно стандартизованою об'єктно-орієнтованою мовою програмування, і, власне, деякі люди називають Ada 95 «першою мовою реального часу».

У Ada 95 було введено три особливо корисні конструкції для вирішення недоліків Ada 83 у плануванні, конкуренції за ресурси та синхронізації:

1. Прагма, яка контролює, як завдання розподіляються.
2. Прагма, яка контролює взаємодію між плануванням завдань.
3. Прагма, яка контролює політику постановки в чергу черг завдань та ресурсів.

Більше того, інші доповнення до мови прагнули зробити Ada 95 повністю об'єктно-орієнтованою. До них належать:

- Типи з тегами
- Пакети
- Захищені одиниці

Правильне використання цих конструкцій дозволяє створювати об'єкти, які демонструють чотири характеристики об'єктно-орієнтованих мов: абстрактну типізацію даних, успадкування, поліморфізм та обмін повідомленнями. У жовтні 2001 року ISO/IEC оголосила про Технічну поправку до стандарту Ada 95, а важливу поправку до міжнародного стандарту було опубліковано в березні 2007 року. Ця остання версія Ada називається «Ada 2005». Відмінності між Ada 95 та Ada 2005 не є значними - у будь-якому разі, не такими суттєвими, як зміни між Ada 83 та Ada 95. Тому, коли ми посилаємося на «Ada» в решті цієї книги, ми маємо на увазі Ada 95, оскільки Ada 2005 — це не новий стандарт, а лише поправка.

Ada ніколи не виправдовувала своєї обіцянки універсальності. Тим не менш, переглянута мова переживає певне повернення, зокрема тому, що Ada використовується в окремих нових системах Міністерства оборони США та багатьох застарілих системах, а також через наявність версій Ada з відкритим вихідним кодом для популярного середовища Linux.

C Мова програмування C, винайдена приблизно у 1972 році в Bell

Laboratories, є гарною мовою для «низькорівневого» програмування. Причина цього полягає в тому, що вона походить від чіткої мови BCPL (наступником якої, батьком C, була мова «B»), яка підтримувала лише один тип – машинне слово. Отже, C підтримує елементи, пов'язані з машиною, такі як адреси, біти, байти та символи, які обробляються безпосередньо цією мовою високого рівня. Ці базові сутності можна ефективно використовувати для керування робочими регістрами процесора, периферійними інтерфейсними блоками та іншим обладнанням, що відображає пам'ять, необхідним у системах реального часу.

Мова C надає спеціальні типи змінних, такі як register , volatile , static та constant , які дозволяють ефективно контролювати генерацію коду на процедурному рівні мови. Наприклад, оголошення змінної як регістрового типу вказує на те, що вона буде часто використовуватися. Це спрямовує компілятор до розміщення такої оголошеної змінної в робочому регістрі, що часто призводить до швидших та менших програм. Крім того, C підтримує лише виклик за значенням, але виклик за посиланням можна легко реалізувати, передавши вказівник на будь-що як значення. Змінні, оголошені як тип volatile , взагалі не оптимізуються компілятором. Ця функція необхідна для обробки вводу-виводу, що відображаються в пам'яті, та інших спеціальних випадків, коли код не повинен бути оптимізований.

Автоматичне приведення стосується неявного перетворення типів даних, яке іноді трапляється в C. Наприклад, значення типу "float" може бути присвоєно цілочисельній змінній, що може призвести до втрати інформації через усікання. Крім того, C надає функції, такі як printf, які приймають змінну кількість аргументів. Хоча це зручна функція, компілятору неможливо ретельно перевірити типи аргументів, що означає, що під час виконання можуть виникати проблеми.

Мова C забезпечує обробку винятків за допомогою сигналів, а також два інші механізми, setjmp та longjmp ', дозволяють процедурі швидко повертатися з глибокого рівня вкладеності — особливо корисна функція в процедурах, що потребують переривання. Виклик процедури setjmp, який насправді є макросом

(але часто реалізується як функція), зберігає інформацію про середовище, яка може бути використана наступним викликом функції бібліотеки longjmp. Виклик longjmp відновлює програму до стану на момент останнього виклику setjmp. Наприклад, припустимо, що процедура викликається для виконання певної обробки та перевірки на помилки. Якщо виявлено помилку, longjmp можна використовувати для переходу до першого оператора після setjmp. Загалом, мова С особливо гарна для вбудованого програмування, оскільки вона забезпечує структуру та гнучкість без складних мовних обмежень. Остання версія міжнародного стандарту мови С вийшла у 1999 році (ANSI/ISO/IEC 9899:1999).

C++ — це гібридна об'єктно-орієнтована мова програмування, яка спочатку була реалізована як макророзширення С у 1980-х роках. Сьогодні С++ є окремою компільованою мовою, хоча компілятори С++ також повинні приймати стандартний код С. С++ має всі характеристики об'єктно-орієнтованої мови та сприяє кращій практиці розробки програмного забезпечення завдяки інкапсуляції та більш просунутим механізмам абстракції, ніж С.

Компілятори С++ реалізують етап попередньої обробки, який, по суті, виконує інтелектуальний пошук та заміну ідентифікаторів, оголошених за допомогою директив #define або #typedef . Хоча більшість прихильників С++ не рекомендують використовувати препроцесор, успадкований від С, він досить широко використовується. Більшість визначень препроцесора в С++ зберігаються у заголовкових файлах, які доповнюють фактичні файли вихідного коду. Проблема з препроцесорним -підходом полягає в тому, що він надає програмістам можливість ненавмисно додавати непотрібну складність до програми. Додатковою проблемою з препроцесорним підходом є його слабка перевірка та валідація типів.

Більшість розробників програмного забезпечення погоджуються, що неправильне використання вказівників спричиняє більшість помилок у програмуванні на С/С++. Раніше програмісти на С++ використовували складну арифметику вказівників для створення та підтримки динамічних структур даних, особливо під час маніпулювання рядками. Як наслідок, вони витрачали багато

часу на пошук складних помилок для простого керування рядками. Однак сьогодні доступні стандартні бібліотеки динамічних структур даних. Наприклад, стандартна мова шаблонів (STL) – це стандартна бібліотека C++, яка має як рядковий, так і wstring тип даних для звичайних і широких символьних рядків відповідно. Ці типи даних нейтралізують будь-які аргументи проти ранніх версій C++, які ґрунтувалися на проблемах маніпулювання рядками.

У C++ є три складні типи даних: класи, структури та об'єднання. Однак C++ не має вбудованої підтримки текстових рядків. Стандартний метод полягає у використанні масивів символів, що завершуються нулем, для представлення рядків. Звичайний код на C організовано у функції, які є глобальними підпрограмами, доступними для програми. C++ додає класи та методи класів, які насправді є функціями, пов'язаними з класами. Однак, оскільки C++ все ще підтримує C, ніщо, в принципі, не заважає програмістам на C++ використовувати звичайні функції. Однак це призведе до змішування використання функцій та методів, що створить заплутані програми.

Множинне успадкування - це корисна функція C++, яка дозволяє створювати клас з кількох батьківських класів. Хоча множинне успадкування справді є потужним інструментом, його може бути важко використовувати правильно, і воно спричиняє багато проблем в іншому випадку. Його також складно реалізувати з точки зору компілятора.

Сьогодні все більше вбудованих систем створюється на C++, і багато практиків запитують: «Чи варто мені реалізовувати свою систему на C чи C++?» Безпосередня відповідь завжди «це залежить». Вибір C замість C++ у вбудованих застосунках є складним компромісом: програма на C буде швидшою та передбачуванішою, але складнішою в обслуговуванні, а програма на C++ буде повільнішою та менш передбачуваною, але потенційно легшою в обслуговуванні. Отже, вибір мови рівносильний питанню, що мені з'їсти: «зелене яблуко» чи «червоне яблуко»? C++ все ще дозволяє низькорівневий контроль; наприклад, він може використовувати вбудовані методи, а не виклики під час виконання. Такий тип реалізації не є особливо абстрактним і не є повністю

низькорівневим, але прийнятний у типових вбудованих середовищах.

На шкоду цьому процесу, може існувати певна тенденція брати існуючий код на C та об'єктивувати його шляхом простого обгортання процедурного коду в об'єкти без урахування найкращих практик об'єктно-орієнтованого проектування. Такого підходу слід категорично уникати, оскільки він може поєднати всі недоліки C++ та жодної з його переваг. Крім того, C++ не забезпечує автоматичного збирання сміття, що означає, що динамічною пам'яттю потрібно керувати вручну або збирати сміття самостійно. Тому під час перетворення програми на C на C++ потрібне повне перероблення, щоб повністю врахувати всі переваги об'єктно-орієнтованого проектування, мінімізуючи при цьому недоліки середовища виконання.

C# (вимовляється «ді-шарп») — це мова програмування, подібна до C++, яка разом зі своїм операційним середовищем має схожість з Java та віртуальною машиною Java відповідно. Таким чином, C# спочатку компілюється в проміжну мову, яка потім використовується для створення власного образу під час виконання. C# асоціюється з платформою Microsoft .NET Framework для зменшених операційних систем, таких як Windows CE. Windows CE має високі можливості налаштування, здатна масштабуватися від невеликих розмірів вбудованої системи (<1 Мбайт) і вище (наприклад, для систем реального часу, що потребують підтримки інтерфейсу користувача). Мінімальна конфігурація ядра забезпечує базову підтримку мережі, управління потоками, підтримку бібліотек динамічного компонування та управління віртуальною пам'яттю. Хоча детальне обговорення виходить за рамки цього тексту, очевидно, що Windows CE спочатку задумувався як операційна система реального часу для платформи NET .

C# підтримує «небезпечний код», дозволяючи вказівникам посилатися на певні місця в пам'яті. Об'єкти, на які посилаються вказівники, повинні бути явно «закріплені», що забороняє збирачу сміття змінювати їхнє розташування в пам'яті. Збирач сміття збирає закріплені об'єкти; він просто не переміщує їх. Ця можливість може підвищити планування, а також дозволяє прямий доступ до

пам'яті (DMA) для запису в певні місця в пам'яті; необхідна можливість у вбудованих системах реального часу. .NET пропонує поколінний підхід до збирання сміття, призначений для мінімізації блокування потоків під час мітки та очищення. Наприклад, засіб для створення потоку в певний момент часу та гарантування завершення потоку до певного моменту часу не підтримується.

Крім того, C# надає багато механізмів синхронізації потоків, але жоден з них не має такого рівня точності. C# підтримує масив конструкцій синхронізації потоків: lock, monitor, mutex та interlock. Lock семантично ідентичний критичній секції - сегменту коду, який гарантує вхід у себе лише одному потоку одночасно. Lock - це скорочена нотація для типу класу monitor. М'ютекс семантично еквівалентний блокуванню з додатковою можливістю роботи в різних просторах процесів. Недоліком м'ютексів є зниження продуктивності. Нарешті, блокування, набір перевантажених статичних методів, використовується для збільшення та зменшення числових значень потокобезпечним способом з метою реалізації протоколу успадкування пріоритетів.

У C# існують таймери, подібні за функціональністю до широко використовуваного таймера Win32. Під час створення таймерів налаштовується час очікування в мілісекундах перед їхнім першим викликом, а також їм надається інтервал, знову ж таки в мілісекундах, що вказує період між наступними викликами. Точність цих таймерів залежить від машини і тому не гарантується, що зменшує їхню корисність у системах реального часу, які використовуються на різних апаратних платформах. C# та платформа .NET не підходять для більшості систем жорсткого реального часу з кількох причин, включаючи необмежене виконання середовища збирання сміття та відсутність потокових конструкцій для належної підтримки планування та детермінізму. Тим не менш, здатність C# ефективно взаємодіяти з API операційної системи захищає розробників від складної логіки управління пам'яттю, а разом з хорошою продуктивністю C# для операцій з плаваючою комою робить її мовою програмування з високим потенціалом для м'яких та навіть стійких застосунків реального часу. Однак потрібен дисциплінований стиль програмування [13]

Java

Java, так само як і C#, є інтерпретованою мовою, тобто код компілюється в машинно-незалежний проміжний код, який виконується в керованому середовищі виконання. Це середовище є віртуальною машиною (див. рис. 3.3), яка виконує інструкції «об'єктного» коду як послідовність програмних директив.

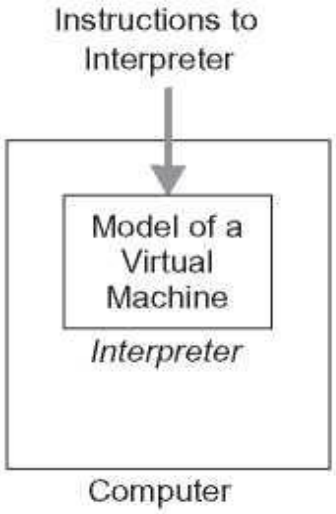

Рисунок 3.3 - Інтерпретатор Java як модель віртуальної машини.

Очевидна перевага такої схеми полягає в тому, що код Java може виконуватися на будь-якому пристрої, який реалізує віртуальну машину. Ця філософія «пиши один раз, запускай будь-де» має важливі застосування в мобільних та портативних обчисленнях, таких як стільникові телефони та смарт-картки, а також у веб-обчисленнях.

Однак, існують також компілятори Java з нативним кодом, які дозволяють Java запускатися безпосередньо «на голому залізі», тобто компілятори конвертують Java в код асемблера або об'єктний код. Наприклад, починаючи з Java 2, віртуальні машини Java підтримують спеціальні компілятори, які компілюють у машинний код для кількох стандартних архітектур. Крім того, існують навіть спеціальні мікропроцесори Java, які безпосередньо виконують байт-код Java на апаратному забезпеченні [15]. Java — це об'єктно-орієнтована мова програмування, і її код дуже схожий на C++. Як і C, Java підтримує виклик за значенням, але виклик за посиланням можна імітувати, що буде обговорено

пізніше. Але Java - це чисто об'єктно-орієнтована мова, тобто вся функціональність у Java має бути реалізована шляхом створення об'єктних класів, створення екземплярів об'єктів цих класів (або базових класів) та маніпулювання атрибутами об'єктів за допомогою методів. Таким чином, практично неможливо взяти застарілий код, написаний процедурною мовою, скажімо, C, та «конвертувати» цей код на Java без справжнього втілення об'єктно-орієнтованого підходу до проектування. Звичайно, хороший об'єктно-орієнтований дизайн не гарантований, але дизайн, отриманий в результаті перетворення, буде справжнім об'єктно-орієнтованим, заснованим на правилах мови. Ця ситуація досить сильно відрізняється від того хибного об'єктно-орієнтованого перетворення, яке можна отримати з C на C++ у прямолінійний спосіб, про який йшлося раніше.

Java дійсно надає препроцесор. Константні члени даних використовуються замість директиви #define, а визначення класів використовуються замість директиви #typedef. В результаті вихідний код Java зазвичай є більш узгодженим і легшим для читання, ніж вихідний код C++. Компілятор Java створює визначення класів безпосередньо з файлів вихідного коду, які містять як визначення класів, так і реалізації методів. Однак, існують природні обмеження продуктивності через результуючу портативність.

Мова Java не підтримує вказівники, але вона забезпечує подібну функціональність через посилання. Java передає всі масиви та об'єкти за посиланням, що запобігає поширеним помилкам через неправильне керування вказівниками. Відсутність вказівників може здатися перешкодою для реалізації структур даних, таких як динамічні масиви. Однак, будь-яку функціональність вказівників можна зручно реалізувати за допомогою посилань, з безпекою, яку забезпечує система виконання Java, наприклад, перевірка меж під час операцій індексації масивів - і все це зі зниженням продуктивності.

Java реалізує лише один складний тип даних: класи. Програмісти Java використовують класи, коли потрібна функціональність структур та об'єднань. Ця узгодженість досягається ціною збільшення часу виконання порівняно з

простими структурами даних. Мова Java не підтримує окремі функції. Натомість, Java вимагає від програмістів знову об'єднувати всі процедури в методи класу, що призводить до значних витрат.

Більше того, Java не має прямої підтримки множинного успадкування. Однак інтерфейси дозволяють реалізацію множинного успадкування. Інтерфейси Java надають описи об'єктів і методів, але не містять реалізацій. У Java рядки реалізовані як об'єкти першого класу (string та StringBuffer), що означає, що вони є ядром мови Java. Реалізація рядків як об'єктів у Java надає кілька переваг. По-перше, створення рядків та доступ до них є узгодженими на всіх системах. По-друге, оскільки класи рядків Java визначені як частина мови Java, рядки функціонують передбачувано щоразу. Нарешті, класи рядків Java виконують розширену перевірку під час виконання, що допомагає усунути помилки. Але всі ці операції збільшують час виконання.

Перевантаження операторів не підтримується в Java. Однак у класі рядків Java «+» позначає об'єднання рядків, а також числове додавання. Мова Java не підтримує автоматичне приведення. В Java, якщо приведення призведе до втрати даних, необхідно явно привести елемент даних до нового типу. Java має неявне «приведення догори». Однак будь-який екземпляр може бути приведений до Object ', який є батьківським класом для всіх об'єктів. Приведення догори є явним і вимагає приведення. Ця явність важлива для запобігання прихованій втраті точності. Аргументи командного рядка, що передаються із системи в програму Java, відрізняються від звичайних аргументів командного рядка, що передаються в програму C++. У C та C++ система передає програмі два аргументи: argc та argv. argc визначає кількість аргументів, що зберігаються в argv, а argv — це вказівник на масив символів, що містить фактичні аргументи. З іншого боку, в Java система передає програмі одне значення: args. args — це масив рядків, що містить аргументи командного рядка.

Java в реальному часі

Цей розділ присвячений адаптації Java для роботи в реальному часі. Хоча Java для роботи в реальному часі є лише модифікацією стандартної мови

Java, вона заслуговує на окрему розмову, оскільки вона все частіше використовується для реалізації м'яких, стійких і навіть жорстких систем реального часу, тоді як стандартна Java в основному використовується лише для м'яких систем реального часу. Хоча ми включаємо обговорення Java для роботи в реальному часі для повноти картини та оскільки воно ілюструє кілька цікавих моментів, ми повторюємо нашу перевагу C++ над версіями Java у більшості випадків.

Окрім непередбачуваної продуктивності збирання сміття, специфікація Java надає лише загальні рекомендації щодо планування. Наприклад, коли існує конкуренція за ресурси обробки, потоки з вищим пріоритетом зазвичай виконуються з перевагою над потоками з нижчим пріоритетом. Однак ця перевага не гарантує, що поток з найвищим пріоритетом завжди буде виконуватися, а пріоритети потоків не можна використовувати для надійної реалізації взаємного виключення. Незабаром було визнано, що цей та інші недоліки роблять стандартну Java неадекватною для більшості систем реального часу.

У відповідь на цю проблему робочій групі Національного інституту стандартів і технологій (NIST) було доручено розробити версію Java, яка була б особливо придатною для вбудованих програм реального часу. Заключний звіт семінару, опублікований ще у вересні 1999 року, визначає дев'ять основних вимог до специфікації реального часу Java (RTSJ 1.0):

R1. Специфікація повинна містити структуру для пошуку та виявлення - доступних профілів.

R2. Будь-яке надане збирання сміття повинно мати обмежену затримку витіснення.

R3. Специфікація повинна визначати зв'язки між потоками Java реального часу на тому ж рівні деталізації, який наразі доступний в існуючих стандартних документах.

R4. Специфікація повинна містити API, що дозволяють зв'язок та – синхронізацію між завданнями Java та не-Java.

R5. Специфікація повинна включати обробку як внутрішніх, так і зовнішніх асинхронних подій.

R6. Специфікація повинна включати певну форму асинхронного завершення потоків.

R7. Ядро повинно забезпечувати механізми для забезпечення взаємного виключення без блокування.

R8. Специфікація повинна забезпечувати механізм, який дозволяє коду запитувати, чи виконується він у потоці Java реального часу, чи в нереальному потоці Java.

R9. Специфікація повинна визначати зв'язки, що існують між потоками Java реального часу та потоками Java нереального часу.

RTSJ 1.0 задовольняє всі вимоги, крім першої, яку вважали неактуальною, оскільки доступ до фізичної пам'яті не є частиною вимог NIST, але внесок промисловості спонукав групу включити її [6]. У 2006 році було анонсовано вдосконалену версію специфікації реального часу, RTSJ 1.1 [14]. Більша частина наступного обговорення адаптована з роботи [25]. RTSJ визначає клас потоків реального часу для створення потоків, які виконує резидентний планувальник. Потоки реального часу можуть отримувати доступ до об'єктів у купі, і тому можуть спричиняти затримки через збирання сміття.

RTSJ 1.0 задовольняє всі вимоги, крім першої, яку вважали неактуальною, оскільки доступ до фізичної пам'яті не є частиною вимог NIST, але внесок промисловості спонукав групу включити її [17]. У 2006 році було анонсовано вдосконалену версію специфікації реального часу, RTSJ 1.1 [5]. Більша частина наступного обговорення адаптована з роботи [5]. RTSJ визначає клас потоків реального часу для створення потоків, які виконує резидентний планувальник. Потоки реального часу можуть отримувати доступ до об'єктів у купі, і тому можуть спричиняти затримки через збирання сміття.

Система повинна поставити в чергу всі потоки, що очікують на отримання ресурсу, в порядку пріоритетності. Ці ресурси включають процесор, а також синхронізовані блоки. Якщо активна політика планування дозволяє потоки з

однаковим пріоритетом, потоки ставляться в чергу за принципом FIFO. Зокрема, система (1) наказує потокам, що очікують, вводити синхронізовані блоки в чергу пріоритетів; (2) додає заблокований потік, який стає готовим до виконання, в кінець черги готовності для цього пріоритету; (3) додає потік, пріоритет якого явно встановлено ним самим або іншим потоком, в кінець черги готовності для нового пріоритету; та (4) поміщає потік, який виконує операцію «yield», в кінець своєї черги пріоритетів. Протокол успадкування пріоритетів -реалізовано за замовчуванням. Специфікація реального часу також надає механізм, за допомогою якого можна реалізувати загальносистемну політику за замовчуванням.

Засіб асинхронної обробки подій складається з двох класів: AsyncEvent та AsyncEventHandler. Об'єкт AsyncEvent представляє щось, що може статися, наприклад, апаратне переривання, або обчислювану подію, наприклад, в'їзд літака в контрольовану область. Коли відбувається одна з цих подій, що вказується викликом методу fire() , система планує пов'язані AsyncEventHandlers . AsyncEvent керує двома речами: відправкою обробників під час спрацьовування події та набором обробників, пов'язаних з подією. Програма може запитувати цей набір та додавати або видаляти обробники. AsyncEventHandler - це планований об'єкт, приблизно подібний до потоку. Коли подія спрацьовує, система викликає методи run() пов'язаних обробників.

Однак, на відміну від інших виконуваних об'єктів, AsyncEventHandler має пов'язані параметри планування, випуску та пам'яті, які керують фактичним виконанням читання або запису. Асинхронна передача керування дозволяє ідентифікувати певні методи, оголошуючи їх такими, що викликають виняток AsynchronouslyInterrupted Exception (AIE). Коли такий метод виконується на вершині стеку виконання потоку, і система викликає java.lang.Thread.interrupt() у потоці, метод негайно діятиме так, ніби система викликала AIE. Якщо система викликає переривання у потоці, який не виконує такий метод, система встановить AIE у стан очікування для потоку та викличе його наступного разу, коли керування перейде до такого методу, або викликавши його, або

повернувшись до нього. Система також встановлює стан AIE у стан «очікування», коли керування знаходиться в синхронізованих блоках, повертається до них або входить до них.

RTSJ визначає два класи для програмістів, які хочуть отримувати доступ до фізичної пам'яті безпосередньо з коду Java. Перший клас, RawMemoryAccess, визначає методи, які дозволяють створювати об'єкт, що представляє діапазон фізичних адрес, а потім отримувати доступ до фізичної пам'яті з гранулярністю byte, word , long та multiple byte . RTSJ не передбачає жодної семантики, окрім методів set та get. Другий клас, PhysicalMemory, дозволяє створювати PhysicalMemoryArea, що представляє діапазон адрес фізичної пам'яті, де система може знаходити об'єкти Java. Наприклад, новий об'єкт Java у певному об'єкті PhysicalMemory можна створити за допомогою методів newInstance() або newArray() . Екземпляр RawMemoryAccess моделює область зберігання даних як послідовність байтів фіксованого розміру. Фабричні методи дозволяють створювати об'єкти RawMemoryAccess з пам'яті в певному діапазоні адрес або за допомогою певного типу пам'яті. Реалізація повинна забезпечити та встановити фабричний метод, який відповідно інтерпретує ці запити. Повний набір методів get та set дозволяє системі отримувати доступ до вмісту області фізичної пам'яті через зміщення від базового масиву, що інтерпретуються як значення даних byte, short, int або long , та копіювати їх до або з масивів byte , short , int або long .

Спеціальні мови реального часу

Протягом останніх десятиліть з'явилася та здобула більший успіх велика різноманітність спеціалізованих мов для програмування в реальному часі. До них належать, наприклад:

- PEARL. Мова програмування для автоматизації процесів та експериментів у реальному часі була розроблена на початку 1970-х років групою німецьких дослідників. PEARL використовує стратегію доповнення та має досить широке застосування в Німеччині, особливо в промисловому управлінні. Поточна версія - PEARL-90.
- в реальному часі Euclid. Експериментальна мова, також створена в

1970-х роках, яка має відзнаку як одна з небагатьох мов, повністю придатних для аналізу планування. Цього досягають за допомогою мовних обмежень. Вона походить від мови програмування Pascal, але так і не знайшла свого поширення в основних додатках.

- Оккам 2. Мова програмування, заснована на формалізмі послідовних процесів комунікації, розроблена для підтримки паралельності на трансп'ютерах (див. розділ 2.5.2). Вона з'явилася наприкінці вісімдесятих років і знайшла практичне впровадження переважно у Великій Британії, але зникла разом із трансп'ютером.
- С у реальному часі. Насправді це загальна назва для будь-якого з різноманітних пакетів макророзширень С. Ці макророзширення зазвичай надають конструкції синхронізації та керування, яких немає у стандартному С.
- Neuron® С. Удосконалення стандарту С з розширеннями для обробки подій, мережевого зв'язку та апаратного вводу/виводу. Він призначений для підтримки застосунків LonWorks (стандарт польової шини для мереж керування) для процесорів Neuron® та відповідних інтелектуальних приймачів-передавачів і широко використовується в спільноті автоматизації будівель.
- реального часу. Загальна назва однієї з кількох бібліотек об'єктних класів, спеціально розроблених для С++. Ці бібліотеки доповнюють стандартний С++, забезпечуючи підвищений рівень синхронізації та контролю.

Крім того, існує безліч інших мов програмування реального часу та відповідних операційних середовищ, таких як Anima, DROL, Erlang, Esterel, Hume, JO VIAL, LUSTRE, Maruti, RLUCID, RSPL та Timber. Деякі з них використовуються для вузькоспеціалізованих застосувань або лише в дослідженнях.

Мови програмування продовжують відігравати життєво важливу роль у розробці систем реального часу, оскільки вони формують явний інтерфейс між розробниками програмного забезпечення та апаратним забезпеченням реального часу. Існує чітка тенденція, пов'язана з обсягом коду у вбудованих додатках: нові продукти матимуть значно більше коду, ніж їхні попередники. Це збільшення

головним чином пов'язане з покращенням функціональності та використанням складніших обчислювальних алгоритмів.

Спільнота розробників апаратного забезпечення відреагувала на ці вимоги просторово та часово розподіленими системними архітектурами, передовими комунікаційними мережами, процесорами з вищою пропускною здатністю інструкцій та більшим обсягом пам'яті. Вони необхідні для запуску складних алгоритмів обробки сигналів та диспетчерського керування, інтелектуального прогнозування несправностей та функцій самодіагностики, віртуальних датчиків на основі моделей, що замінюють фізичні, більш комплексних інтерфейсів на рівні користувача та системи тощо. Як ця глобальна тенденція впливає на використання мов програмування та написання програмного забезпечення реального часу? Що ж, поточний рішучий перехід від процедурних мов до об'єктно-орієнтованих мов є однією з відповідей на зростання кількості програмного забезпечення, що зустрічається в типових вбудованих додатках. Об'єктно-орієнтовані мови, такі як Ada, C++, C# та Java (або Real-Time Java), забезпечують основні засоби для підвищення продуктивності програмістів, а також зручності підтримки та повторного використання розробленого коду.

Повторне використання програмного забезпечення, яке розглядається як можливість зменшити обсяг надлишкової роботи з кодування в програмних проектах, має й зворотний бік: надмірне повторне використання існуючого коду може навіть обмежити інноваційність нових продуктів. Крім того, існує ризик поширення поганого коду. Отже, повторне використання коду має бути зосереджене на природно незмінних у часі та добре перевірених модулях, які існують у конкретному застосуванні з покоління в покоління. Типові приклади включають локальні алгоритми керування, генератори опорних сигналів, обробку аналогових та цифрових вводів/виводів, а також стандартні інтерфейси польової шини.

З точки зору практикуючого інженера, автоматичну генерацію коду все ще можна розглядати як лише нову технологію - навіть після кількох десятиліть

еволюції. Тим не менш, потреба в автоматичних генераторах коду зараз більша, ніж будь-коли, через зростання розміру коду в системах реального часу. Однак малоймовірно, що автоматичні генератори коду стануть основним інструментом для практиків у найближчому майбутньому. Але вони, безумовно, матимуть постійно зростаюче використання в кодуванні таких рутинних структур, як кінцеві автомати та деякі числові алгоритми, які не потребують здатності програмістів вирішувати проблеми. Хоча продуктивність, зручність обслуговування та можливість повторного використання є справді важливими факторами, вони не мають значення, якщо не виконуються вимоги системи до роботи в реальному часі. Запускати складні алгоритми з високою частотою дискретизації на економічно ефективній апаратній платформі складно. Тому в певних прикладних областях необхідно використовувати спеціальні мови програмування в реальному часі замість загальних. Такі адаптовані мови призводять до високопередбачуваної поведінки в реальному часі та мінімальних мовних витрат. Крім того, все ще виправдано використовувати процедурні мови, такі як мова С, для кодування менших та критичних за часом програм або навіть використовувати мову асемблера в рідкісних випадках.

Кінцевою метою програмістів реального часу є стандартизована мова програмування та відповідний компілятор з високим рівнем абстракції, суворою передбачуваністю в реальному часі та ефективно оптимізованою генерацією об'єктного коду. У наступному розділі ми обговоримо методології розробки вимог, які мають очевидний зв'язок зі зростанням рівня абстракції, що, здається, є центральним питанням у підвищенні продуктивності розробників програмного забезпечення.

3.6 Висновки по розділу

Сучасні мови програмування, зокрема об'єктно-орієнтовані, відіграють ключову роль у розробці систем реального часу, забезпечуючи підвищену продуктивність, повторне використання коду та масштабованість. Однак

критичним залишається дотримання часових обмежень, що часто вимагає застосування спеціалізованих мов або низькорівневого програмування для забезпечення максиму.

У розробці систем реального часу важливо обирати мову програмування, яка забезпечує передбачувану продуктивність, ефективне використання ресурсів і підтримує стандарти кодування. Хоча мова асемблера все ще актуальна для вузькоспеціалізованих завдань, основний акцент нині робиться на об'єктно-орієнтовані мови, які забезпечують баланс між продуктивністю та зручністю розробки.

ВИСНОВКИ

Дослідження, проведене в рамках цієї роботи, підкреслює критичну роль програмування систем реального часу в забезпеченні надійності, передбачуваності та ефективності в критичних галузях, таких як аерокосмічна промисловість, автомобілебудування, медицина та телекомунікації. Аналіз основ систем реального часу показав, що чітке розуміння їхніх концепцій, подолання хибних уявлень і вирішення багатодисциплінарних проблем дизайну є ключовими для створення стабільних систем. Еволюція систем реального часу від простих вбудованих пристроїв до складних розподілених архітектур підтверджує необхідність адаптації програмних і апаратних рішень до сучасних вимог.

Другий розділ, присвячений апаратному забезпеченню, продемонстрував, що базова архітектура процесорів, технології пам’яті та архітектурні інновації, такі як багатоядерні процесори та кеш-пам’ять з низькою затримкою, відіграють вирішальну роль у забезпеченні часової детермінованості систем реального часу. Оптимізація апаратного забезпечення для зменшення затримок і підвищення продуктивності є необхідною умовою для ефективного програмування таких систем.

Третій розділ, що аналізує мови програмування для систем реального часу, виявив, що вибір мови суттєво впливає на якість і надійність програмного забезпечення. Кодування на асемблераційних мовах забезпечує максимальний контроль над апаратним забезпеченням, але є трудомістким. Процедурні мови, такі як C і Ada, пропонують баланс між продуктивністю та зручністю розробки, тоді як об’єктно-орієнтовані мови, такі як C++, дозволяють створювати модульні та масштабовані системи, хоча потребують додаткової оптимізації для дотримання часових обмежень. Огляд мов програмування підкреслив важливість вибору інструментів, що відповідають специфіці проєкту, з урахуванням вимог до швидкості, безпеки та масштабованості.

Узагальнюючи, програмування систем реального часу є складною, але перспективною галуззю, яка вимагає інтеграції знань з апаратного забезпечення, програмування та системного дизайну. Результати дослідження можуть бути використані розробниками, інженерами та дослідниками для створення надійних систем реального часу, що відповідають сучасним галузевим стандартам. Перспективи подальших досліджень включають розробку нових мов програмування, оптимізованих для реального часу, інтеграцію штучного інтелекту для динамічного керування ресурсами та вдосконалення архітектур для підтримки розподілених систем реального часу. Ці напрямки сприятимуть розвитку більш ефективних і доступних рішень для критичних застосувань у майбутньому.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛА

1. AdaCore. Ada Programming Language for Real-Time Systems. *adacore.com*, 2024. — Режим доступу: https://www.adacore.com/ada

2. ARM. Cortex-R Series for Real-Time Applications. *arm.com*, 2024. — Режим доступу: https://www.arm.com/products/silicon-ip-cpu/cortex-r

3. Ben-Ari, M. *Principles of Concurrent and Distributed Programming.* — 2nd ed. — Harlow, UK: Pearson Education, 2006. — 384 с.

4. Burns, A., & Wellings, A. *Real-Time Systems and Programming Languages: Ada, Real-Time Java, and C/Real-Time POSIX*. — 4th ed. — Boston, MA: Addison-Wesley, 2009. — 611 с. — Режим доступу: https://www.pearson.com/store/p/real-time-systems-and-programming-languages/P100000245813

5. Buttazzo, G. C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* — 3rd ed. — New York, NY: Springer, 2011. — 524 с. — Режим доступу: https://doi.org/10.1007/978-1-4614-0676-1

6. Cooling, J. E. *Real-Time Operating Systems Book 1: The Theory.* — 2nd ed. — Raleigh, NC: Lulu Press, 2017. — 252 с. — Режим доступу: https://www.realtimeoperatingsystems.com

7. DOU. Real-Time Systems in Embedded Development. *dou.ua*, 2024. — Режим доступу: https://dou.ua/lenta/articles/real-time-embedded

8. FreeRTOS. FreeRTOS: Real-Time Operating System for Microcontrollers. *freertos.org*, 2024. — Режим доступу: https://www.freertos.org

9. Ganssle, J. G. *The Art of Designing Embedded Systems.* — 2nd ed. — Burlington, MA: Newnes, 2008. — 312 с. — Режим доступу: https://www.elsevier.com/books/the-art-of-designing-embedded-systems/ganssle/978-0-7506-8644-0

10. IEEE Computer Society. Real-Time Systems: Design and Challenges. *IEEE Transactions on Computers*, 2024, 73(5), 102–115. — Режим доступу: https://doi.org/10.1109/TC.2024.3357892

11. Klein, M. H., Ralya, T., Pollak, B., Obenza, R., & Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* — Boston, MA: Kluwer Academic Publishers, 1993. — 704 с.

12. Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* — 2nd ed. — New York, NY: Springer, 2011. — 376 с. — Режим доступу: https://doi.org/10.1007/978-1-4419-8237-7

13. Laplante, P. A., & Ovaska, S. J. *Real-Time Systems Design and Analysis: Tools for the Practitioner.* — 4th ed. — Hoboken, NJ: Wiley-IEEE Press, 2011. — 584 с. — Режим доступу: https://doi.org/10.1002/9781118136607

14. Liu, C. L., & Layland, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 1973, 20(1), 46–61. — Режим доступу: https://doi.org/10.1145/321738.321743

15. Liu, J. W. S. *Real-Time Systems.* — Upper Saddle River, NJ: Prentice Hall, 2000. — 592 с.

16. Mall, R. *Real-Time Systems: Theory and Practice.* — New Delhi, India: Pearson Education, 2007. — 240 с.

17. Mentor Graphics. Embedded Software Development for Real-Time Systems. *mentor.com*, 2024. — Режим доступу: https://www.mentor.com/embedded-software/real-time

18. Microchip Technology. Real-Time Embedded Systems Design. *microchip.com*, 2024. — Режим доступу: https://www.microchip.com/en-us/solutions/embedded/real-time

19. Obermaisser, R. *Event-Triggered and Time-Triggered Control Paradigms.* — New York, NY: Springer, 2005. — 159 с. — Режим доступу: https://doi.org/10.1007/b137582

20. Shaw, A. C. *Real-Time Systems and Software.* — New York, NY: Wiley, 2001. — 224 с.

21. Stankovic, J. A., & Ramamritham, K. *Advances in Real-Time Systems.* — Los Alamitos, CA: IEEE Computer Society Press, 1993. — 784 с.

22. Tanenbaum, A. S., & Woodhull, A. S. *Operating Systems: Design and Implementation.* — 3rd ed. — Upper Saddle River, NJ: Prentice Hall, 2006. - 1088 с.

23. Vahid, F., & Givargis, T. *Embedded System Design: A Unified Hardware/Software Introduction.* — Hoboken, NJ: Wiley, 2002. — 352 с. — Режим доступу: https://www.wiley.com/en-us/Embedded+System+Design-p-9780471386780

24. Ward, P. T., & Mellor, S. J. *Structured Development for Real-Time Systems.* — Englewood Cliffs, NJ: Prentice Hall, 1985. — 156 с.

25. Wikipedia. Системи реального часу. *uk.wikipedia.org*, 2024. — Режим доступу: https://uk.wikipedia.org/wiki/Системи_реального_часу

26. X Post by @embedded_dev. Real-time programming with FreeRTOS: Tips for optimizing latency in IoT devices. *X*, 2025. — Режим доступу: https://t.co/7kpRBfh9X

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Програмування систем реального часу"

Обсяг пояснювальної записки: 89 аркушів

Дата закінчення дипломної роботи 10 червня 2025р.

Підпис студента _____