МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
«Прикарпатський національний університет
імені Василя Стефаника»

Фізико-технічний факультет

Голота В. І.

Курс лекцій з дисципліни

“СИСТЕМНЕ ПРОГРАМУВАННЯ. АСЕМБЛЕР x86-64”

Електронне мережеве навчальне видання

Івано-Франківськ, 2024

УДК 004.432.2
Г 61

Рекомендовано до друку Вченою радою фізико-технічного факультету Прикарпатського національного університету імені Василя Стефаника (протокол № 2 від 24 жовтня 2024 року)

Рецензенти:

Когут Ігор Тимофійович, професор, завідувач кафедри комп'ютерної інженерії та електроніки Прикарпатського національного університету імені Василя Стефаника, доктор технічних наук

Никируй Любомир Іванович, професор кафедри фізики і хімії твердого тіла Прикарпатського національного університету імені Василя Стефаника, кандидат фізико-математичних наук

Курс лекцій з дисципліни "Системне програмування. Асемблер x86-64". [Електронний ресурс] / Прикарпатський національний університет імені Василя Стефаника; уклад. Голота В. І. – Електронні текстові дані (1 файл: 10,6 Мбайт). – Івано-Франківськ: Прикарпатський національний університет імені Василя Стефаника, 2024. – 234 с. – Назва з екрану.

У курсі лекцій розглянуто програмну модель, адресацію пам'яті, базову систему команд мікроконтролерів з архітектурою x86-64, асемблер NASM, використання системних викликів ОС Linux в програмах асемблера, взаємодію асемблера з мовами програмування високого рівня, команди співпроцесора і розширень MMX, SSE, AVX.

Курс лекцій призначений для здобувачів ступеня бакалавра галузі знань 12 "Інформаційні технології" спеціальності 123 "Комп'ютерна інженерія". Може бути також корисним студентам, які вивчають системне програмування у курсах споріднених дисциплін відповідних спеціальностей.

УДК 004.42(076.6)

ЗМІСТ

1. ОСНОВНІ ПОНЯТТЯ СИСТЕМНОГО ПРОГРАМУВАННЯ

Мета. Ознайомлення з задачами, основними поняттями і особливостями системного програмування.

Вступ. Системне програмування пов’язане з розробленням програм, які взаємодіють з операційною системою і апаратним забезпеченням обчислювальної системи. Для розроблення системних програм використовують низькорівневі мови і асемблери. Асемблери використовують мнемоніку машинних команд і переводять їх у коди машинних команд, зрозумілих процесорові. Процесор обробляє як цілі числа, так і числа з плаваючою крапкою, подані у форматі стандарту IEEE-754.

Сучасні операційні системи працюють у багатозадачних режимах. Для підтримки таких режимів використовується віртуалізація і поділ системних програм на дві групи, одні з яких працюють в режимі суперкористувача, а інші в режимі звичайного користувача.

Для написання системних програм потрібно знати архітектуру процесорів, типи даних, способи адресації пам'яті, формати машинного подання чисел та команди асемблера.

План.

1. Особливості системного програмування.
2. Машинний код і асемблер
3. Особливості програмування багатозадачних ОС
4. Машинне подання цілих чисел
5. Машинне подання чисел з плаваючою крапкою в стандарті IEEE-754-1985
5.1. Перетворення числа формату 32 біт IEEE-754 в десяткове число
5.2. Формальне подання чисел в стандарті IEEE-754 для любого формату точності
5.3. Обчислення десяткових чисел з плаваючою крапкою з чисел поданих у стандарті IEEE-754
5.4. Подання денормалізованого числа та інших чисел у форматі IEEE-754
5.5. Межі діапазону для чисел одинарної та подвійної точності IEEE-754
5.6. Точність подання дійсних чисел у форматі IEEE7-54
5.7. Округлення чисел в стандарті IEEE-754
6. Архітектури процесорів
7. Історія розвитку процесорів IA-32 і Intel 64
8. Процесори i9
9. Середовище виконання програм

1. Особливості системного програмування

Системне програмування (або **програмування систем**) – це вид програмування, який полягає у розробленні програм, які взаємодіють з системним програмним забезпеченням (операційною системою), або апаратним забезпеченням обчислювальної системи. Головною відмінністю системного програмування в порівнянні з прикладним програмуванням є те, що прикладне програмне забезпечення призначене для кінцевих користувачів, тоді як результатом системного програмування є програми, які обслуговують апаратне забезпечення або операційну систему (наприклад, дефрагментація диску) що обумовлює значну залежність програм такого

типу від апаратури. Слід зазначити, що прикладні програми можуть використовувати у своїй роботі фрагменти коду, характерні для системних програм, і навпаки; тому чіткої межі між прикладним та системним програмуванням немає. Оскільки різні операційні системи (ОС) відрізняються як внутрішньою архітектурою, так і способами взаємодії з апаратним та програмним забезпеченням, то принципи системного програмування для різних ОС є відмінними. Тому розробка системних програм з однаковими функціями для різних ОС, може суттєво відрізнятися.

В загальному для системного програмування характерні наступні особливості:

• враховуються особливості ОС та/або апаратного забезпечення, на яких передбачається функціювання програми;

• використовуються низькорівневі мови програмування які можуть працювати у ресурсообмеженому середовищі, мають мінімальні затримки за часом виконання, використовують невеликі бібліотеки часу виконання (RTL), підтримують прямий доступ до пам'яті та керуючої логіки, дозволяють писати частини програми на асемблері;

• складне налагодження, якщо неможливо запустити програму через обмеження у ресурсах.

Основними задачами системного програмування є розроблення:

• базових систем введення/виведення (Base Input Output System, Bios);

• операційних систем;

• драйверів;

• засобів віртуалізації апаратних та програмних середовищ;

• трансляторів, компіляторів, симуляторів, інтерпретаторів, компонувачів, завантажувачів, зневадників, дизасемблерів, трасувальників;

• утиліт обслуговування програмної і апаратної частини;

• систем захисту від несанкційованого доступу та шкідливих програм.

Кожна із вказаних задач є самостійним напрямом в системному програмуванні, що змушує фахівців програмістів спеціалізуватися у одному або в декількох із них.

2. Машинний код і асемблер

Практично всі обчислювальні машини працюють за одним і тим же принципом. Обчислювальний пристрій складається з **центрального процесора**, **оперативної пам'яті** і **периферійних пристроїв**. У більшості випадків ці пристрої підключаються до спільної **шини**.

Оперативна пам'ять складається з **комірок пам'яті**, кожна з яких має унікальну **адресу**. Комірка містить декілька (частіше всього – вісім) двійкових розрядів, кожний з яких може знаходитися в одному з двох станів (які позначаються "нуль" або "один"). Це дозволяє комірці знаходитися в одному з 2^n станів, де n – кількість розрядів комірки. Якщо в комірці 8 розрядів, то число можливих станів 2^8=256 і в комірці можна "запам'ятати" число від 0 до 255. Якщо потрібно зберігати число з більшого діапазону, то використовують декілька послідовних комірок. У різних машинах використовується різна послідовність байтів для зберігання чисел.

Порядок від старшого до молодшого.

Порядок від старшого до молодшого (англ. **big-endian**, «тупокінцевий»): $A_n,\ldots,A_0$. Запис починається зі старшого розряду й закінчується молодшим. Так записують числа на папері й цей порядок є стандартним для протоколів TCP/IP, він застосовується в заголовках пакетів даних і в багатьох протоколах вищого рівня, розроблених для застосування поверх TCP/IP. Тому,

порядок байтів від старшого до молодшого часто називають мережевим порядком байтів (англ. network byte order). Обробку чисел із таким порядком байтів апаратно реалізовано в процесорах IBM 360/370/390, Motorola 68000, SPARC (звідси третя назва — порядок байтів Motorola, Motorola byte order).

Порядок від молодшого до старшого.

Порядок від молодшого до старшого або (англ. **little-endian**, «гострокінцевий»): $A_0,\ldots,A_n$, запис починається з наймолодшого розряду й закінчується найстаршим. Такий порядок запису прийнятий у пам'яті комп'ютерів з процесорами Intel, у яких його було апаратно реалізовано, у зв'язку з чим іноді його називають «інтелівський» порядок байтів (за назвою фірми-розробника архітектури x86). Такий порядок застосовується в USB, конфігурації PCI, таблиця розділів GUID, рекомендаціях FidoNet, однак маловживаний у крос-платформних протоколах і форматах даних.

Змінюваний порядок.

Деякі процесори можуть працювати і з порядком від молодшого до старшого, і зі зворотнім, наприклад, ARM, PowerPC, DEC Alpha, MIPS, PA-RISC і IA-64. Зазвичай порядок байтів вибирається програмно під час ініціалізації операційної системи, але може бути вибраний і апаратними перемичками на материнській платі. У цьому випадку правильніше говорити про порядок байтів операційної системи. Змінюваний порядок байтів (англ. **bi-endian)**.

Змішаний порядок.

Змішаний порядок байтів (англ. **middle-endian**) іноді застосовується при роботі з числами, довжина яких перевищує машинне слово. У машинному слові байти зберігаються в порядку, природному для даної архітектури, але самі слова йдуть у зворотному порядку.

Класичний приклад middle-endian – подання 4-байтових цілих чисел на 16-бітових процесорах родин PDP-11 (відомий як PDP-endian) . Для подання 2-байтових значень (слів) застосовувався апаратний порядок: спочатку молодший байт, потім – старший. Але в подвійному (4-байтовому) слові записувалося спочатку старше слово, а потім молодше. У процесорах VAX і ARM застосовується змішаний порядок для довгих дійсних чисел.

Порівняння.

Істотною перевагою little-endian у порівнянні з big-endian вважається можливість «неявної типізації» цілих чисел при читанні меншого обсягу байт (за умови, що прочитане число вміщується в діапазон). Так, якщо в комірці пам'яті міститься число 0x00000022, то прочитавши один байт отримаємо число 0x22, прочитавши два байти (int16) — число 0x0022 і т. д. Однак, це ж може вважатися одночасно недоліком, тому що може спричинити помилки втрати даних.

Недоліком little-endian (у порівнянні з big-endian) вважається «неочевидність» значення байтів пам'яті, наприклад, при налагодженні: послідовність байтів (A1, B2, C3, D4) означає число 0xD4C3B2A1, тоді як у big-endian ця послідовність (A1, B2, C3, D4) читається природнім чином – 0xA1B2C3D4.

Найменш зручним у роботі вважається middle-endian формат запису. Він зберігся тільки на старих платформах. Для запису довгих чисел (чисел, довжина яких істотно перевищує розрядність машини) зазвичай переважає порядок слів little-endian (оскільки більшість арифметичних операцій над довгими числами здійснюються від молодших розрядів до старших). Порядок байтів в слові — звичайний для такої архітектури.

Так для послідовностей чисел

```
m1 db 0xA1,0xB2,0xC3,0xD4
```

порядок розміщення байтів у пам’яті буде наступним:

	little-endian				**big-endian**			
Адреси пам'яті	0	1	2	3	0	1	2	3
Розрядові байти	0xD4	0xC3	0xB2	0xC3	0xA1	0xB2	0xC3	0xD4

```
m1 dw 0,0,0x43,0x21
```

	little-endian				big-endian			
Адреси пам'яті	0	1	2	3	0	1	2	3
Розрядові байти	00	0x21	00	0x43	0x43	00	0x21	00

Порядок байтів в конкретній машині можна визначити за допомогою програми на мові Сі

```
#include <stdio.h>
unsigned short x = 1; /* 0x0001 */
int main(void)
{
  printf("%s\n", *((unsigned char *) &x) == 0 ? "big-endian" : "little-
endian");
  return 0;
}
```

Структура даних у пам’яті Intel процесора x86-32/64 показана на рис. 1.

Рисунок 1 – Структури даних пам’яті Intel процесора

При необхідності послідовність комірок пам’яті можна розглядати як стрічку з двійкових розрядів, як ціле число, як число з плаваючою крапкою або машинну інструкцію. При цьому важливо вказати, що самі комірки “не знають”, як інтерпретувати інформацію, яка в них зберігається.

В процесорі є деяка обмежена кількість **регістрів**. Процесор може копіювати дані з оперативної пам’яті у регістри і з регістрів в оперативну пам’ять, виконувати над вмістом регістрів арифметичні та інші операції.

Кількість інформації, яку може обробити процесор в одній інструкції (команді) називається машинним словом. Розмір більшості регістрів відповідає машинному слову. В сучасних процесорах використовується розмір машинного слова 32 і 64 біти. З історичних

причин в Intel процесорах “словом, word” називають два байти (16 біт), а “подвійним словом, dword” – чотири байти (32 біт).

Програма, яка призначена для виконання, записується в оперативну пам'ять як послідовність машинних інструкцій (команд), тобто цифрових кодів, які позначають ті або інші операції. Один з регістрів процесора називається **лічильником команд** і містить адресу тієї комірки пам’яті в якій розміщується наступна до виконання команда.

Цикл оброблення команд:

1. Видобування інструкції з пам'яті.

Використовуючи поточне значення лічильника команд, процесор видобуває деяку кількість байт з пам'яті і поміщає їх в буфер команд.

2. Декодування команди

Процесор читає вміст буфера команд, визначає код операції і її операнди. Довжина декодованої команди додається до поточного значенню лічильника команд.

3. Завантаження операндів.

Видобуваються значення операндів. Якщо операнд розміщений в комірках пам'яті - обчислюється виконавча адреса.

4. Виконання операції над даними

5. Запис результату

Результат може бути записаний у тому числі і в лічильник команд для зміни послідовного порядку виконання команд.

Деякі машинні команди можуть змінити послідовність виконання команд, вказавши процесору на перехід в інше місце програми, тобто в явному вигляді змінивши значення лічильника команд. Такі команди називаються **командами переходу**. Розрізняють **умовні** і **безумовні** команди переходу. Команди умовного переходу спочатку перевіряють істинність деякої умови і виконують перехід тільки при її виконанні. Команди безумовного переходу завжди заставляють процесор продовжити виконання команд із заданої адреси. Процесори звичайно підтримують такі переходи із запам’ятовуванням точки повернення, що використовується при виклику підпрограм.

Програму, яку виконує процесор подають у вигляді **машинного коду**. Програма у машинному коді складається з **машинних команд**, які позначаються числами (кодами). Процесор може легко дешифрувати такі команди.

Асемблер – це програма, яка приймає на вхід текст, який містить умовні позначення машинних команд (**мнемонік**), зручних для людини, і переводить їх у послідовність відповідних кодів машинних команд, зрозумілих процесору. Мова таких умовних команд (мнемомік) називається **мовою асемблера**.

Програмування на асемблері суттєво відрізняється від програмування на мовах високого рівня. В програмі на асемблері потрібно однозначно вказати як машинні команди, регістри і комірки пам’яті, так і їх послідовність використання для реалізації заданого алгоритму.

Для одного і того ж мікропроцесора може існувати декілька різних асемблерів. Система машинних кодів (**система команд)** звичайно змінитися не може. Але для одних і тих же команд можна придумати різні позначення, наприклад `add eax, ebx` та `addl %ebx, %eax`, хоча машинний код для обох команд буде однаковий.

При програмуванні на асемблері використовуються не тільки мнемоніки, але і *директиви*, які є прямими наказами процесору. Виконуючи такі накази процесор може зарезервувати пам'ять, оголосити ту або іншу позначку видимою з інших модулів програми, перейти до

генерації іншої секції програми, обчислити (під час асемблювання) який-небудь вираз. Набір таких директив може бути різним, як за можливостями, так і за синтаксисом.

3. Особливості програмування багатозадачних ОС

Сучасні ОС запускають одночасно на виконання декілька задач (в термінах ОС процесів). Такий режим роботи обчислювальної системи називається *багатозадачним* і породжує необхідність апаратного захисту виконуваних програм одна від одної та захисту ОС від програм. Тому ЦП процесор підтримує механізм *захисту пам'яті*: кожній виконуваній програмі виділяється певна область пам'яті. Звертатися до комірок пам'яті за межами цієї області програма не може.

У багатозадачному режимі задачі (процеси) користувача не допускаються до прямої роботи із зовнішніми пристроями. Для обмеження можливостей користувача, частина машинних команд оголошена **привілейованою**. Процесор може працювати в **привілейованому режимі** (режим суперкористувача) або в **обмеженому режимі** (режим користувача, режим задачі). В обмеженому режимі привілейовані команди недоступні. В привілейованому режимі процесор може виконувати як звичайні так і привілейовані команди. Операційна система, природно, виконується у привілейованому режимі, а при передачі керування задачі користувача перемикається в обмежений режим. До привілейованих відносяться команди взаємодії із зовнішніми пристроями, команди для налаштування механізмів захисту пам'яті та деякі інші команди, які впливають на роботу всієї системи в цілому. В багатозадачній ОС задача користувача може тільки перетворювати інформацію у виділеній їй області пам'яті. Всю взаємодію із зовнішнім світом задача здійснює через звернення до ОС**.** Навіть завершення задачі здійснюється через звернення до ОС. Таке звернення задачі до ОС називається **системним викликом.**

Інший важливий момент, який необхідно згадати – це наявність в оперативній пам'яті механізму **віртуальної пам'яті**. Кожна комірка пам'яті має свій порядковий номер. Саме цей номер використовує ЦП для роботи з комірками пам'яті через спільну шину, щоб відрізнити їх одна від одної. Такий номер прийнято називати фізичною адресою пам'яті. У машинному коді програми використовуються саме фізичні адреси, які просто називають "адресами". З розвитком багатозадачного режиму ОС виявилося, що з ряду причин використання фізичних адрес незручне. Тому у сучасних ОС використовуються два види адрес. Сам процесор працює з фізичними адресами, а програми використовують віртуальні адреси. Віртуальна адреса – це число з деякого віртуального адресного простору. У 32-розрядних процесорів віртуальний простір є множиною цілих чисел від 0 до 2^{32}-1. Адреси звичайно записують у шістнадцятковій формі, так що адреса може бути числом від 0000_0000 до ffff_ffff. Віртуальна адреса може не відповідати якій-небудь фізичній адресі комірки пам'яті. Відповідність між віртуальними і фізичними адресами задається шляхом налаштування ЦП. За це налаштування відповідає ОС. При відповідному налаштуванні, ЦП, отримавши з чергової машинної команди віртуальну адресу, перетворює її у фізичну і тоді уже звертається до оперативної пам'яті. Це дозволяє кожній програмі мати свій адресний простір. ОС може налаштувати таке перетворення адрес, щоб віртуальний простір однієї задачі відображався в один фізичний адресний простір, а іншої задачі – зовсім у інший.

4. Машинне подання цілих чисел

Для подання інформації в обчислювальних системах переважно використовується двійкова система числення. Кожний із розрядів може приймати значення 0 або 1. На практиці для подання цілих чисел використовується одна комірка (8 біт), дві комірки (16 біт), чотири комірки (32 біт) або вісім комірок (64 біт). Обмеження розрядності зумовлює існування “найбільшого” числа 2^n-1, де n – кількість розрядів (2^8-1 = 11111111_2 = 255_{10} = ff_{16}). Якщо до найбільшого двійкового числа додати 1, то виникне переповнення, всі значущі розряди перетворяться у 0 і появиться 1 перенесення у неіснуючий старший розряд. Подібно, якщо із найменшого двійкового числа відняти 1, то виникне позика із неіснуючого старшого розряду.

```
  1111_1111          1_000_0000
+          1       -          1
----------------   ----------------
 1.0000+0000         1111_1111
```

При додаванні 8-розрядних беззнакових і знакових і цілих чисел може виникати переповнення, рис.2.

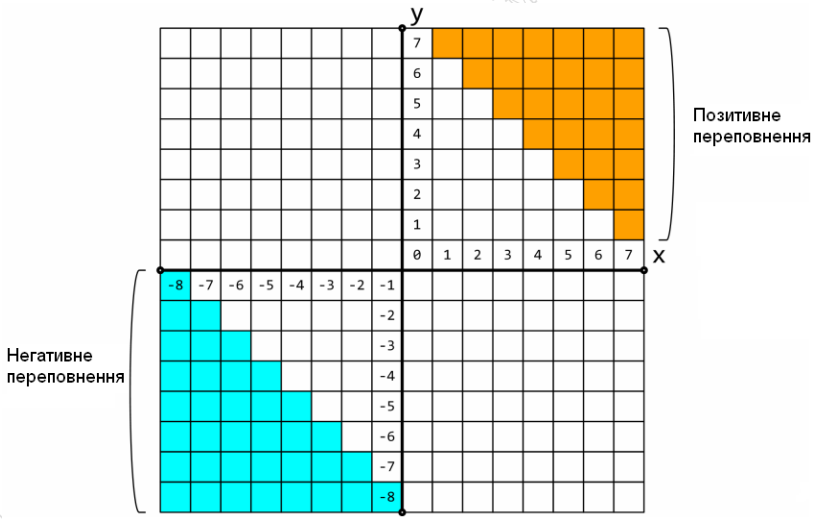


Рисунок 2 – Позитивне і негативне переповнення при додаванні 8-розрядних знакових і беззнакових чисел

Для подання знакових цілих чисел використовується доповняльний код. Доповняльний код можна уявити як відображення деякого діапазону, який включає позитивні і негативні цілі числа на інший діапазон, який містить тільки позитивні числа, рис. 3.

Один байт може містити числа в діапазоні від 0 до 255. Діапазон від 0 до 127 відображається сам на себе, а негативним числам відповідає діапазон від 128 до 255. Числу -1 відповідає число 255.

Доповняльний код може бути розширений до 2-байт (від 0 до 65535) або до 4-байт. Він буде охоплювати діапазон 2-байтових чисел від -32768 до 32767 і діапазон 4-байтових чисел -від 2 147 483 648 до 2 147 483 647.

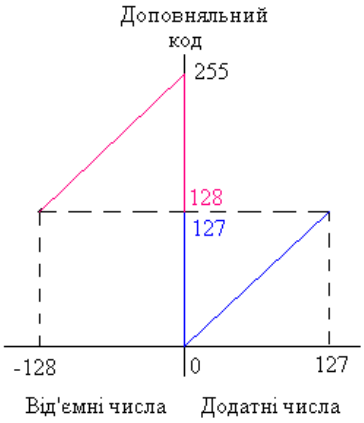


Рисунок 3 – Схема відображення чисел в доповняльний код

Для отримання доповняльного коду від’ємне знакове ціле число перетворюється в інверсний код (інвертуються всі розряди числа) і потім до молодшого розряду додається 1. Приклад доповняльного коду для чисел -1, -2, -3.

0000_0001 1111_1110 + 1 ---------- 1111_1111	0000_0010 1111_1101 + 1 ---------- 1111_1110	0000_0011 1111_1100 + 1 ---------- 1111_1101

Для подання -1 використовуються одиниці у всіх розрядах, незалежно від розрядності числа. Тобто для 8-розрядного числа 1111_1111 означають -1, а не 255. Число 1111_1110 буде означати -2, а не 255-1=254.

Таким чином прийнята наступна домовленість: якщо набір двійкових чисел розглядається як подання знакового числа, то **від’ємними** вважаються числа, старший біт яких дорівнює одиниці. Діапазон значень 8-розрядних знакових цілих чисел у прямому і доповняльному коді

Десятковий код	Прямий код	Доповняльний код
127	0111_1111	0111_1111
…	…	…
2	0000_0010	0000_0010
1	0000_0001	0000_0001
0	0000_0000	0000_0000
-1	1000_0001	1111_1111
-2	1000_0010	1111_1110
…	…	…
-127	1111_1111	1000_0001
-128	н/в	1000_0000

При такій домовленості дуже просто змінити знак числа. Щоб ***змінити знак числа*** на протилежний при використанні доповняльного коду, достатньо інвертувати значення всіх розрядів і до отриманого значення додати одиницю. Наприклад, числа 5 і -5:

$5_{10} \rightarrow 0000\_0101_2 \rightarrow 1111\_1010 \rightarrow 1111\_1011 \rightarrow -5$

$-5_{10} \rightarrow 1111\_1011_2 \rightarrow 0000\_0100 \rightarrow 0000\_0101 \rightarrow 5$

При зміні знаку нуля нуль залишається нулем:

$0_{10} \rightarrow 0000\_0000_2 \rightarrow 1111\_1111 \rightarrow 0000\_0000 \rightarrow 0$

Аналогічна ситуація виникає і для найменшого знакового числа -128, так як для нього відсутнє у даній розрядності додатне число з таким же модулем:

$-128_{10} \rightarrow 1000\_0000_2 \rightarrow 0111\_1111 \rightarrow 1000\_0000 \rightarrow -128_{10}$

5. Машинне подання чисел з плаваючою крапкою в стандарті IEEE-754-1985

Стандарт IEEE-754-1985 визначає:

• як подавати нормалізовані позитивні і негативні числа з плаваючою крапкою;

• як подавати денормалізовані позитивні і негативні числа з плаваючою крапкою;

• як подавати нульові числа;

• як подавати спеціальну величину нескінченість (`Infinity`);

• як подавати спеціальну величину "Не число" (`NaN` або `NaNs`);

• чотири режими округлення.

Стандарт визначає чотири формати подання чисел з плаваючою крапкою:

• з одинарною точністю (single-precision) 32 біта;

• з подвійною точністю (double-precision) 64 біта;

• з одинарною розширеною точністю (single-extended precision) >= 43 біт (рідко використовуваний);

• з подвійною розширеною точністю (double-extended precision) >= 79 біт (звичайно використовують 80 біт).

Приклади подання чисел в нормалізованому і денормалізованому експоненційному виді:

Десяткове число 155,625 $\rightarrow 1{,}55625 \cdot 10^{+2} = 1{,}55625 \cdot exp_{10}^{+2}$.

Число $1{,}55625 \cdot exp_{10}^{+2}$ складається з двох частин: мантиси M=1,55625 і експоненти exp_{10}=+2. Якщо мантиса знаходиться в діапазоні 1<=M<10, то число вважається **нормалізованим**.

Число $0{,}155625 \cdot exp_{10}^{+3}$ складається з двох частин: мантиси M=0,155625 і експоненти exp_{10}=+3. Якщо мантиса знаходиться у діапазоні 0,1<=M<1, то число вважається **денормалізованим**.

Перетворення десяткового числа в двійкове число з плаваючою крапкою:

$155{,}625_{10} = 10011011{,}101_2$ - число в десятковій і двійковій системі з плаваючою крапкою

$155{,}625 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$

155,625 =128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 + 0,5 + 0 + 0,125

Двійкове число є нормалізованим, якщо його мантиса 1 <= M < 2.

Нормалізований вид числа в десятковій і двійковій системі:

$1{,}55625 \cdot exp_{10}^{+2} = 1{,}0011011101 \cdot exp_2^{+111}$

Основні складові нормалізованого двійкового числа:

• мантиса M = 1,0011011101;

• експонента exp_2= +111.

В знаковий біт записується 0 для додатних чисел і 1 — для від'ємних чисел. В поле порядку записується зміщений порядок – до порядку нормалізованого числа додається така константа, так щоб всі порядки задавалися додатними числами. В поле мантиси записуються всі дробові цифри мантиси, крім першої одиниці, яка є цілою частиною. Ціла частина не входить до коду числа і є так званою прихованою одиницею.

5.1. Перетворення числа формату 32 біт IEEE-754 в десяткове число

Щоб записати число в стандарті IEEE-754 або відновити його, необхідно знати три параметри:

- S – біт знаку (31-й біт)
- E – зміщену експоненту (30-23 біти)
- M – залишок від мантиси (22-0 біти)

S, E, M це цілі числа записані як числа IEEE-754 у двійковому виді.

Формула для отримання десяткового числа із числа IEEE-754 одинарної точності:

$$F=(-1)^S 2^{E-127}(1 + M/2^{23}),$$

де F – десяткове число.

Приклад:

$F =(-1)^0 \cdot 2^{(134-127)} \cdot (1+ 1810432 / 2^{23}) = 2^7 \cdot (1+0{,}2158203125) = 128 \cdot 1{,}2158203125 = 155{,}625,$

де $(1+ M/2^{23})$ – мантиса (одиниця в цій формулі – це та одиниця, яку відкинули з 23 біт, а залишок мантиси в десятковому виді знаходиться відношенням двох цілих чисел – залишку мантиси до цілого).

5.2. Формальне подання чисел в стандарті IEEE-754 для любого формату точності

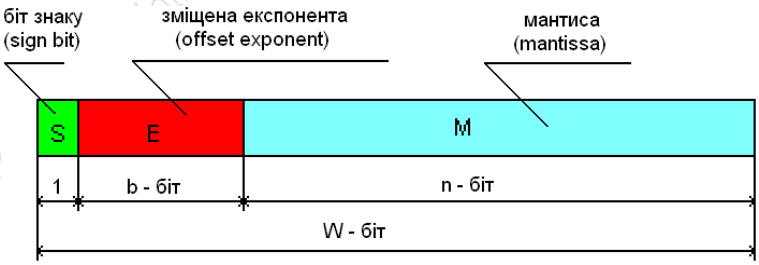


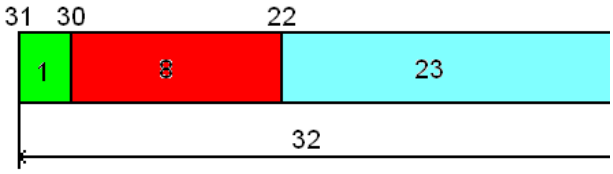
Рисунок 4 – Подання чисел в стандарті IEEE-754:

- S - біт знаку, якщо S=0 – позитивне число; S=1 – негативне число;
- E - зміщена експонента двійкового числа;exp_2 = E - $(2^{(b-1)} - 1)$ – експонента двійкового нормалізованого числа з плаваючою крапкою $(2^{(b-1)} - 1)$ – задане зміщення експоненти (в 32-бітовому IEEE-754 воно дорівнює +127, а в 64-бітовому +1023);
- M – залишок мантиси двійкового нормалізованого числа з плаваючою крапкою.

5.3. Обчислення десяткових чисел з плаваючою крапкою з чисел поданих у стандарті IEEE-754

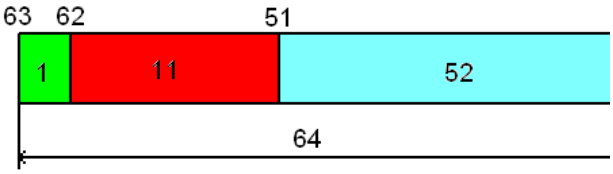
$$F=(-1)^S 2^{(E-2^{|b-1|}+1)}(1 + M/2^n) \quad (1)$$

З формули (1) можна отримати формули для знаходження десяткових чисел з форматів одинарної (32 біти) і подвійної (64 біти) точності IEEE-754:



$$F=(-1)^S 2^{E-127} (1+M/2^{23})$$

Рисунок 5 – Формат числа одинарної точності (single-precision) 32 біти



$$F=(-1)^S 2^{E-1023} (1+M/2^{52})$$

Рисунок 6 – Формат числа подвійної точності (double-precision) 64 біти

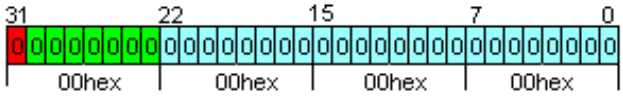
5.4. Подання денормалізованого числа та інших чисел у форматі IEEE-754

Якщо застосувати формулу (1) для обчислення мінімального і максимального числа одинарної точності у поданні IEEE-754, то можна отримати наступні результати:

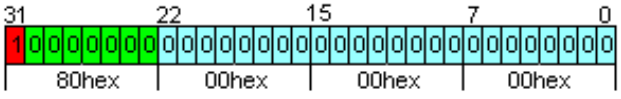
- 00 00 00 00 hex= 5,87747175411144e-39 (мінімальне позитивне число)
- 80 00 00 00 hex=-5,87747175411144e-39 (мінімальне негативне число)
- 7f ff ff ff hex= 6,80564693277058e+38 (максимальне позитивне число)
- ff ff ff ff hex=-6,80564693277058e+38 (максимальне негативне число)

Звідси видно, що неможливо подати число нуль або нескінченість у заданому форматі. Тому формула (1) не застосовується у наступних випадках:

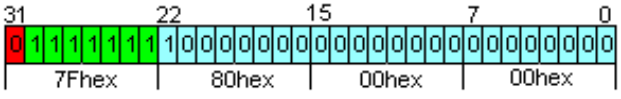
1. Число IEEE-754=00 00 00 00hex вважається числом +0:



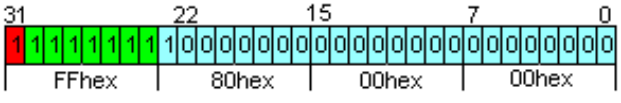
число IEEE-754=80 00 00 00hex вважається числом -0:



2. Число IEEE-754=7F 80 00 00hex вважається числом +∞:

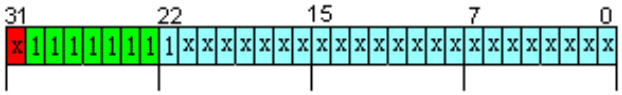


число IEEE-754=FF 80 00 00hex вважається числом -∞:

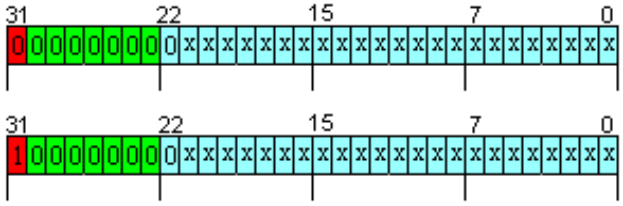


3. Числа IEEE-754=FF (1xxx)X XX XXhex не вважаються числами (NAN), крім випадку п.2, числа IEEE7- 54=7F (1xxx)X XX XXhex не вважаються числами (NAN), крім випадку п.2

Числа подані в бітах з 0...22 можуть бути любим числом крім 0 (тобто +∞ і -∞).



4. Числа IEEE754=(x000) (0000) (0xxx)X XX XXhex вважаються денормалізованими числами, за винятком чисел п.1 (тобто -0 і +0):



Формула розрахунку денормалізованих чисел

$$F=(-1)^S\ 2^{(E-2^{\wedge}|b-1|+2)}\ (1+M/2^n) \qquad (2)$$

Пояснення до виняткових чисел:

• Наявність двох нулів вносить симетричність.

• - ∞/ +∞ - числа, які більші границь діапазону подання чисел вважаються нескінченими.

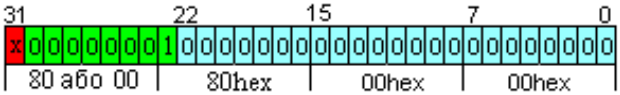
• Не числа NAN (No a Numbers) – до них відносяться символи, або результати недопустимих операцій.

• Денормалізовані числа – це числа, мантиси яких лежать в діапазоні 0.1<=M<1. Денормалізовані числа знаходяться ближче до нуля, ніж нормалізовані. Денормалізовані числа розбивають мінімальний розряд нормалізованого числа на підмножину. Це зв'язано з тим, що в технічній практиці частіше зустрічаються величини близькі до нуля.

5.5. Межі діапазону для чисел одинарної та подвійної точності IEEE-754

Знаючи формат чисел з одинарною точністю можна порахувати межі діапазону подання дійсних чисел у цьому форматі. Для цього потрібно підставити значення максимальних і мінімальних абсолютних чисел IEEE-754 у формули (1) і (2).

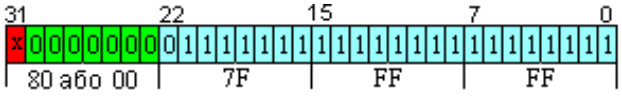
Мінімальне нормалізоване число (абсолютне):



00 80 00 00 = $2^{-126}\cdot(1+0/2^{23})= 2^{-126} \approx 1{,}17549435\cdot e^{-38}$

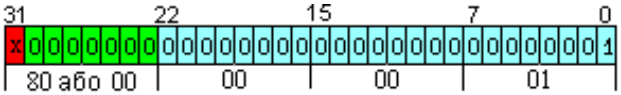
80 80 00 00 = $-2^{-126}\cdot(1+0/2^{23})=-2^{-126} \approx -1{,}17549435\cdot e^{-38}$

Максимальне денормалізоване число (абсолютне):



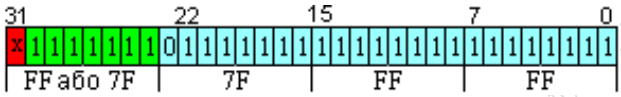
00 7F FF FF = $2^{-126}\cdot(1-2^{-23}) \approx 1{,}17549421\cdot e^{-38}$
80 7F FF FF = $-2^{-126}\cdot(1-2^{-23}) \approx -1{,}17549421\cdot e^{-38}$
Як видно, мінімальне нормалізоване число межує з максимальним денормалізованим.

Мінімальне денормалізоване число (абсолютне)



00 00 00 01 = $2^{-126}\ 2^{-23} = 2^{-149} \approx 1{,}40129846\cdot e^{-45}$
80 00 00 01 = $-2^{-126}\ 2^{-23} = 2^{-149} \approx -1{,}40129846\cdot e^{-45}$
Це число межує з нулем.

Максимальне нормалізоване число (абсолютне)



7F 7F FF FF = $2^{127}\cdot(2-2^{-23}) \approx 3{,}40282347\cdot e^{+38}$
FF 7F FF FF = $-2^{127}\cdot(2-2^{-23}) \approx -3{,}40282347\cdot e^{+38}$
Це число межує з нескінченістю.

Повний діапазон чисел одинарної точності (32 біт) за стандартом IEEE-754

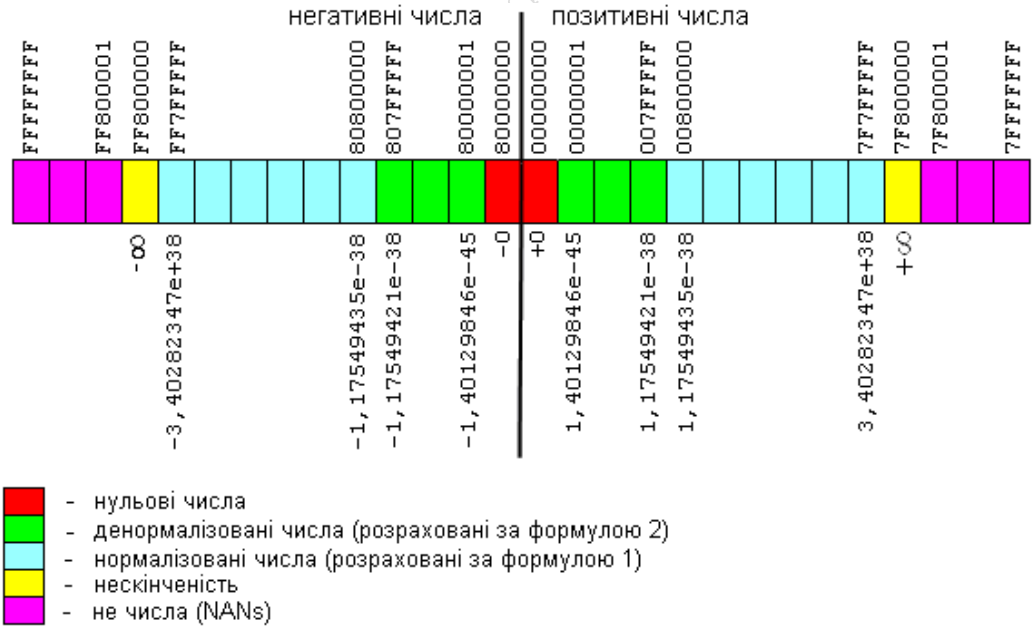


Рисунок 7 – Діапазон чисел формату одинарної точності (32 біти) за стандартом IEEE-754

Повний діапазон чисел подвійної точності (64 біт) за стандартом IEEE-754

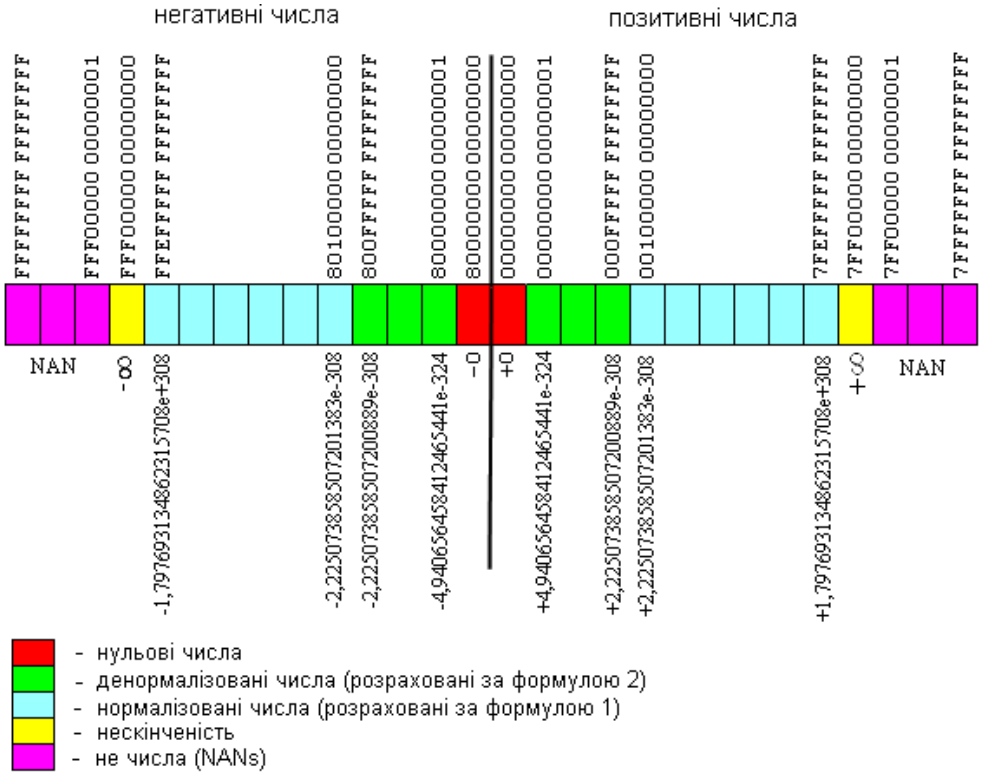


Рисунок 8 – Діапазон чисел формату подвійної точності (64 біт) за стандартом IEEE-754

5.6. Точність подання дійсних чисел у форматі IEEE-754

Числа подані у форматі IEEE-754 утворюють скінчену множину, на яку відображається нескінчена множина дійсних чисел. Тому початкове число може бути подане у формату IEEE-754 з похибкою.

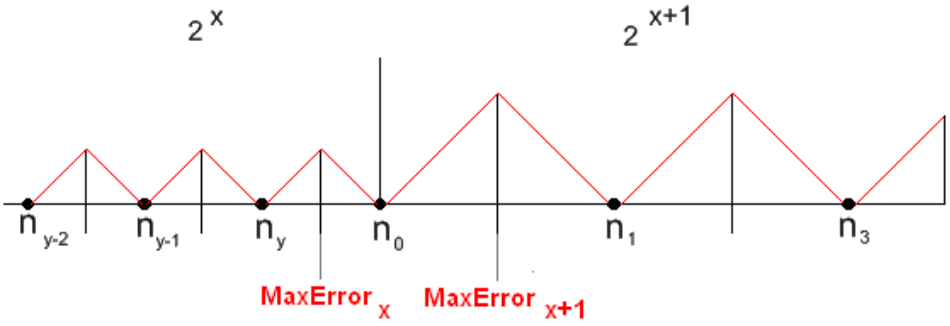


Рисунок 9 – Функція похибки точності подання числа у IEEE-754

Абсолютна максимальна похибка для числа у форматі IEEE-754 дорівнює в границі половині кроку чисел. Крок чисел подвоюється із збільшенням експоненти двійкового числа на одиницю. Тобто, чим далі від нуля, тем ширший крок чисел у форматі IEEE-754 на числовій вісі. Крок денормалізованих чисел дорівнює $2^{(E-149)}$ (Single) і $2^{(E-1074)}$ (Double). Відповідно границя максимальної абсолютної похибки буде дорівнювати 1/2 кроку числа: $2^{(E-150)}$ (Single) і $2^{(E-1075)}$ (Double). Відносна похибка в % буде дорівнювати: $(2^{(E-150)}/F)*100\%$ (Single) і $(2^{(E-1075)}/F)*100\%$ (Double). Крок нормалізованих чисел дорівнює $2^{(E-150)}$ (Single) і $2^{(E-1075)}$ (Double). Відповідно границя максимальної абсолютної похибки буде дорівнювати 1/2 кроку числа: $2^{(E-151)}$ (Single) і

$2^{(E-1076)}$ (Double). Відносна похибка в % буде дорівнювати: $(2^{(E-151)}/F)*100\%$ (Single) і $(2^{(E-1076)}/F)*100\%)$ (Double).

Максимальна відносна похибка для денормалізованого числа (single/double):

$$\frac{2^{(E-150)}}{2^{(E-126)}\frac{M}{2^{23}}}=\frac{1}{2M}$$

Максимальна відносна похибка нормалізованого числа (single):

$$\frac{2^{(E-151)}}{2^{(E-127)}\left(1+\frac{M}{2^{23}}\right)}=\frac{1}{2^{24}+2M}$$

Максимальна відносна похибка нормалізованого числа (double):

$$\frac{2^{(E-1076)}}{2^{(E-1023)}\left(1+\frac{M}{2^{52}}\right)}=\frac{1}{2^{53}+2M}$$

Таблиця 1 – Максимальна можлива похибка для чисел Single

IEEE-754, hex	Число, dec	Абсолютна похибка, dec	Відносна похибка, %
00000001	2^{-149} ≈1,401298e-45	2^{-150}≈0,700649e-45	=50
00000002	2^{-148} ≈2,802597e-45	2^{-150}≈0,700649e-45	=25
00000032	≈7,00649e-44	2^{-150}≈0,700649e-45	=1
007FFFFF	≈1,175494e-38	2^{-150}≈0,700649e-45	≈5,96e-6
00800001	≈1,175494e-38	2^{-149} ≈1,401298e-45	≈11,9209e-6
0DA24260	≈1,0e-30	2^{-123} ≈9,4039e-38	≈9,4039e-6
1E3CE508	≈1,0e-20	2^{-90} ≈8,0779e-28	≈8,0779e-6
2EDBE6FF	≈1,0e-10	2^{-57} ≈6,9389e-18	≈6,9389e-6
3F800000	≈1,0	2^{-23} ≈1,192e-7	≈11,9209e-6
41200000	≈10,0	2^{-20} ≈9,5367e-7	≈9,5367e-6
42C80000	≈1,0e+2	2^{-17} ≈7,6294e-6	≈7,62939e-6
501502F9	≈1,0e+10	2^{10} ≈1,024e+3	≈10,24e-6
60AD78EC	≈1,0e+20	2^{43} ≈8,7961e+12	≈8,7961e-6
7149F2CA	≈1,0e+30	2^{76} ≈7,5558e+22	≈7,5558e-6
7F7FFFFF	≈+3,40282e+38	2^{104} ≈2,02824e+31	≈5,96e-6

Таблиця 2 – Максимальна можлива похибка для чисел Double

IEEE754, hex	Число, dec	Абсолютна похибка, dec	Відносна похибка, %
00000000 00000001	2^{-1074} ≈4,940656e-324	2^{-1075}≈2,470328e-324	=50
00000000 00000002	2^{-1073} ≈9,881313e-324	2^{-1075}≈2,470328e-324	=25
00000000 00000032	≈2,470328e-322	2^{-1075}≈2,470328e-324	=1
000FFFFF FFFFFFFF	≈2,225073e-308	2^{-1075}≈2,470328e-324	≈1,110223e-14

00100000 00000001	≈2,225074e-308	2^{-1074} ≈4,940656e-324	≈2,220446e-14
2B2BFF2E E48E0530	≈1,0e-100	2^{-385} ≈1,268971e-116	≈1,268971e-14
3FF00000 00000000	=1,0	2^{-52} ≈2,220446e-16	≈2,220446e-14
54B249AD 2594C37D	≈1,0e+100	2^{280} ≈1,942669e+84	≈1,942669e-14
6974E718 D7D7625A	≈1,0e+200	2^{612} ≈1,699641e+184	≈1,699641e-14
7FEFFFFF FFFFFFFF	≈1,79769e+308	2^{971} ≈1,99584e+292	≈1,110223e-14

Як видно з вказаного, основна маса чисел у форматі IEEE-754 має стабільну невелику відносну похибку:

- максимально можлива відносна похибка для числа Single складає 2^{-23}*100% =11,920928955078125e-6 %;

- максимально можлива відносна похибка для числа Double складає 2^{-52}*100% =2,220446049250313080847263336181 6e-14 %.

Характеристики чисел одинарної і подвійної точності стандарту IEEE-754

Таблиця 3 – Характеристики чисел в форматі 32/64 біт у стандарті IEEE-754

Назва формату	single-precision	double-precision
Довжина числа, біт	32	64
Зміщена експонента (E), біт	8	11
Залишок від мантиси (M), біт	23	52
Зміщення	127	1023
Двійкове денормалізоване число	$(-1)^S \cdot 0, M \cdot exp_2^{-127}$, де M - бінарне	$(-1)^S \cdot 0, M \cdot exp_2^{-1023}$, де M- бінарне
Двійкове нормалізоване число	$(-1)^S \cdot 1, M \cdot exp_2^{(E-127)}$, де M - бінарне	$(-1)^S \cdot 1, M \cdot exp_2^{(E-1023)}$, де M- бінарне
Десяткове денормалізоване число	$F = (-1)^S \cdot 2^{(E-126)} \cdot M/2^{23}$	$F = (-1)^S \cdot 2^{(E-1022)} \cdot M/2^{52}$
Десяткове нормалізоване число	$F = (-1)^S \cdot 2^{(E-127)} \cdot (1+ M/2^{23})$	$F = (-1)^S \cdot 2^{(E-1023)} \cdot (1+M/2^{52})$
Абс. макс. можлива похибка числа	$2^{(E-150)}$	$2^{(E-1075)}$
Від. макс. можлива похибка денормалізованого числа	1/(2M)	1/(2M)
Від. макс. можлива похибка нормалізованого числа	$1/(2^{24}+2M)$	$1/(2^{53}+2M)$
Мінімальне число	$\pm 2^{-149} \approx \pm 1,40129846 \cdot e^{-45}$	$\pm 2^{-1074} \approx \pm 4,94065646 \cdot e^{-324}$
Максимальне число	$\pm 2^{127} \cdot (2-2^{-23}) \approx \pm 3,40282347 \cdot e^{+38}$	$\pm 2^{1023} \cdot (2-2^{-52}) \approx \pm 1,79769313 \cdot e^{+308}$

5.7. Округлення чисел в стандарті IEEE-754

Стандарт IEEE-754 передбачає чотири способи округлення чисел:

- округлення до найближчого цілого;
- округлення до нуля;
- округлення до +∞;
- округлення до -∞.

Таблиця 4 – Приклади округлення чисел до десятих

Початкове число	Округлення до найближчого цілого	Округлення до нуля	Округлення до +∞	Округлення до -∞
1,33	1,3	1,3	1,4	1,3
-1,33	-1,3	-1,3	-1,3	-1,4
1,37	1,4	1,3	1,4	1,3
-1,37	-1,4	-1,3	-1,3	-1,4
1,35	1,4	1,3	1,4	1,3
-1,35	-1,4	-1,3	-1,3	-1,4

Самий легкий в апаратній реалізації спосіб округлення до нуля.

6. Архітектури процесорів

Мікропроцесор (МП) – це пристрій, що являє собою одну або декілька великих інтегральних схем (ВІС), які виконують функції процесора обчислювального пристрою. Класичний обчислювальний пристрій складається з арифметичного пристрою (АП), пристрою керування (ПК), запам'ятовуючого пристрою (ЗП) і пристрою введення-виведення (ПВВ).

МП можуть мати різні архітектури – CISC, RISC, MISC.

CISC (англ. Complex Instruction Set Computing) – архітектури, які характеризуються наступним набором властивостей:

- різні формати і різні довжини команд;
- значна кількість різних режимів адресації;
- складне кодування інструкції;
- арифметичні дії виконуються в одній команді;
- невелике число регістрів, кожний з яких виконує строго визначену функцію.

Типовими представниками CISC-архітектури є процесори на основі команд x86, процесори Motorola MC680x0. Завдяки поширеності процесорів архітектур x86 і x86-64, CISC-системи найбільше використовуються в обчислювальній техніці – вони домінують в сегментах робочих станцій, персональних комп'ютерів, серверів початкового і середнього рівня. При цьому пізніші x86-процесори (Intel Pentium 4, Pentium D, Core, AMD Athlon, Phenom), хоча і CISC-сумісні, але є процесорами з RISC-ядром, і формально вважаються гібридними. В таких гібридних CISC-процесорах CISC-інструкції перетворюються в набір внутрішніх RISC-команд, при цьому одна команда x86 може породжувати декілька RISC-команд (наприклад для процесора типу P6 – до чотирьох), команди виконуються на суперскалярному конвеєрі одночасно по декілька команд. Основний недолік CISC-архітектури у порівнянні з RISC – більш складний підхід до розпаралелювання обчислень.

RISC (англ. Reduced Instruction Set Computing) – архітектури в яких процесор має скорочений набір команд. Система команд має спрощений вид. Всі команди мають однаковий формат з простим кодуванням.

Для звернення до пам'яті використовують команди завантаження і зберігання, інші команди мають тип регістр-регістр. Команда, яка поступає в центральний обчислювальний блок, уже розділена по полях і не потребує додаткової дешифрації.

Найбільш характерною особливістю RISC процесорів є розділення інструкцій для оброблення даних і звернення до пам'яті. Для звернення до пам'яті використовуються тільки

інструкції `load` і `store`, а всі інші інструкції обмежені внутрішніми регістрами. Це спрощує архітектуру процесорів: інструкції мають фіксовану довжину, спрощений конвеєр, логіка із затримками доступу до пам'яті ізольована тільки в двох інструкціях. В результаті RISC-архітектури ще називають архітектурами `load/store`.

RISC-архітектури мають більшу продуктивність, ніж CISC, за рахунок використання суперскалярного і VLIW-підходу, можливості значного підвищення тактової частоти і спрощення кристалу з вивільненням площі під кеш величезного розміру. Також RISC-архітектури мають менше енергоспоживання за рахунок меншої кількості транзисторів.

У суперскалярних архітектурах рішення про паралельне виконання двох або більше команд приймає апаратура процесора на етапі виконання. Ефективне використання такої архітектури потребує спеціальної оптимізації машинного коду в компіляторі для генерації пар незалежних команд (коли результат однієї команди не є аргументом іншої команди).

Архітектура VLIW (англ. very long instruction word – дуже довге слово команди) має декілька обчислювальних пристроїв. Характеризується тим, що одна інструкція містить декілька операцій, які повинні виконуватися паралельно. Відрізняються від суперскалярної архітектури тим, що рішення про розпаралелювання приймає не апаратура на етапі виконання, а компілятор на етапі генерації коду. Команди дуже довгі і містять явні інструкції по розпаралелюванню декількох підкоманд на декілька пристроїв виконання. Розробка ефективного компілятора для VLIW є дуже складною задачею. Перевага VLIW перед суперскалярною архітектурою полягає в тому, що компілятор може бути більш складним, ніж пристрої керування процесора, і може зберігати більше контекстної інформації для прийняття більш вірних рішень по оптимізації.

На даний час багато архітектур процесорів є RISC-подібними, наприклад, ARM, SPARC, AVR, MIPS, POWER і PowerPC. RISC-процесори переважають в сегментах мобільних пристроїв, мікроконтролерів і Unix-серверів вищого рівня.

Крім CISC і RISC архітектур є і інші альтернативи – наприклад, MISC, OISC, масово-паралельна обробка, систолічна матриця, реконфігуровні обчислення, потокова архітектура.

MISC (англ. Minimal (Multipurpose) lnstruction Set Computer) – комп'ютер з мінімальним (багатоцільовим) набором інструкцій.

Збільшення розрядності процесорів привело до ідеї пакування декількох команд в одне велике слово. Це дозволило збільшити продуктивність комп'ютера і обробляти одночасно декілька потоків даних. Крім цього, MISC використовує стекову модель обчислювального пристрою і основні команди роботи із стеком мови Forth.

Елементна база MISC систем складається з двох частин, які або виконані в окремих корпусах, або об'єднані. Основна частина – RISC CPU, розширюється шляхом підключення другої частини – ПЗП мікропрограмного керування. Система набуває властивості CISC. Основні команди працюють на RISC CPU, а команди розширення перетворюються в адресу мікропрограми. RISC CPU виконує всі команди за один такт, а друга частина еквівалентна CPU із складним набором команд. Наявність ПЗП усовує недолік RISC, коли при компіляції з мови високого рівня мікрокод генерується з бібліотеки стандартних функцій, яка займає багато місця в ОЗП. Оскільки мікропрограма уже дешифрована і відкрита для програміста, то час вибірки з ОЗП на дешифрацію не потрібний.

7. Історія розвитку процесорів IA-32 і Intel 64

7.1. 16-розрядні процесори і сегментація

16-розрядні процесори **8086** і **8088** появилися у 1978 році. Процесор 8086 мав 16-розрядні регістри і 16-розрядну зовнішню шину даних, з можливістю 20-розрядної адресації, що давало доступ до 1 Мбайт адресного простору. Процесор 8088 був подібний до процесор 8086 за винятком того, що він мав 8-розрядну зовнішню шину даних.

Процесори 8086/8088 впровадили сегментацію в архітектуру IA-32. Для підтримки сегментації у 16-розрядний сегментний регістр поміщають вказівник на сегмент пам'яті розміром до 64 Кбайт. Використовуючи одночасно чотири сегментні регістри процесори 8086/8088 могли адресувати до 256 Кбайт пам'яті без перемикання між сегментами. 20-розрядна адресація, яка формувалася з використанням сегментного регістру і додаткового 16-бітового вказівника забезпечувала доступ до адресного простору в 1 Мбайт.

7.2. Процесор 286

Процесор **286** появився у 1982 році і вніс в архітектуру IA-32 захищений режим роботи. В захищеному режимі використовувався вміст сегментного регістру як селектор або вказівник в дескрипторних таблицях. Дескриптори надавали 24-розрядні базові адреси фізичної пам'яті розміром до 16 Мбайт, підтримували керування віртуальною пам'яттю на основі перемикання сегментів і деякі механізми захисту. Ці механізми включали

- перевірка обмежень сегменту;
- опції сегменту "read-only" і "execute-only";
- чотири рівні привілеїв.

7.3. Процесор 386

Процесор **386** появився у 1985 році і був першим 32-розрядним процесором в архітектурі IA-32. Він використовував 32-розрядні регістри як для зберігання операндів, так і для адресації. Молодша половина кожного 32-розрядного регістра зберігала властивості 16-розрядних регістрів попередніх моделей процесорів, підтримуючи зворотну сумісність. Процесор також підтримував віртуальний режим процесора 8086, що дозволяло виконувати програми створені для МП 8086/8088.

Додатково процесор 386 підтримував:

- 32-розрядну адресну шину, що дозволяло адресувати до 4 Гбайт фізичної пам'яті.
- сегментовану і плоску модель пам'яті;
- розбиття пам'яті на фіксовані сторінки розміром 4 Кбайт, чим надавалася можливість керування віртуальною пам'яттю;
- підтримка паралельності виконання стадій конвеєра;

7.4. Процесор 486

Процесор **486** появився у 1989 році і додав більше можливостей у паралельність виконання стадій конвеєра шляхом розширення інструкцій декодування і блоків виконання у п'яти стадіях конвеєра. Кожна стадія функціонує паралельно з іншими, при цьому в різних стадіях виконується одночасно до п'яти інструкцій.

В процесорі добавлено:

• 8 Кбайт кеш першого рівня, який збільшує процент інструкцій, які можуть бути виконані з скалярним порядком за один такт;

• інтегрований процесор з плаваючою крапкою x87 FPU;

• можливості економії енергії та керування системою.

7.5. Процесор Pentium

Процесор **Pentium** появився у 1993 році і мав конвеєр другого рівня виконання для досягнення суперскалярної продуктивності (два конвеєри, відомі як *u* і *v*, разом могли виконувати дві інструкції за один машинний такт). Впроваджено два кеші 1-го рівня, один 8 Кбайт для коду, інший 8 Кбайт для даних. Кеш даних використовував протокол MESI для підтримки більш ефективного write-back кешу порівняно з попереднім write-through кешем. Для підвищення продуктивності конструкцій циклу добавлено передбачення галужень на основі таблиці галужень.

В процесорі добавлено:

• розширення у віртуальний режим процесора 8086, які роблять його більш ефективним і дозволяють використовувати сторінки розміром 4 Кбайт і 4 Мбайт;

• внутрішні шини даних розрядністю 128 і 256, які збільшили швидкість передачі внутрішніх даних;

• збільшена розрядність зовнішньої шини даних до 64;

• APIC для підтримки багатьох процесорів;

• режим подвійного процесора для двоядерних процесорних систем;

Послідовними кроками у наступні моделі Pentium добавлено MMX технологію. MMX використовує одну інструкції і множину даних (SIMD) для паралельних обчислень на пакетах цілих чисел, які містяться в 64-розрядних регістрах.

7.6. Родина процесорів P6

Родина процесорів **P6** появилася в 1995-1999 роках і базувалася на скалярній мікроархітектурі і встановлювала нові стандарти продуктивності. Одним із завдань при проектуванні процесорів P6 було перевершити продуктивність процесора Pentium використовуючи той самий 4-х шаровий, метал BICMOS технологічний процес з роздільною здатністю 0,6 мікрометрів. В родину цих процесорів входили:

• *Pentium Pro* процесор – суперскалярний, 3-стадієвий конвеєр. Використовуючи техніку паралельної обробки процесор міг в середньому декодувати, диспетчеризувати і виконати три інструкції за один машинний такт. В процесорі Pentium Pro появилося динамічне виконання (аналіз потоків мікроданих, надшвидке виконання, вищої якості передбачення галужень і спекулятивні обчислення) у суперскалярній реалізації. Процесор мав два 8 Кбайт кеші 1-го рівня і додаткові два 256 Кбайт кеші 2-го рівня.

• *Pentium II* процесор добавив MMX технології і нове розміщення зовнішніх контактів (SECC). Кеш даних і інструкцій 1-го рівня збільшився до 16 Кбайт і підтримувалися кеші 2-го рівня розмірами 256 Кбайт, 512 Кбайт і 1 Мбайт. Додаткова шина з півтактовою швидкістю з'єднувала кеш 2-го рівня з процесором. Введено стани з низьким споживанням енергії, такі як AutoHalt, Stop-Grant, Sleep і Deep Sleep для економії енергії в стані очікування.

• *Pentium II Xeon* процесор об'єднав найкращі характеристики попередніх процесорів – 4-х і 8-ми шляхове масштабування і кеші 2-го рівня розміром 2 Мбайт які функціювали на повній швидкості додаткової шини.

• родина *Celeron* процесорів була зорієнтована на комерційний сегмент. В ній використовувався вбудований кеш 2-го рівня розміром 128 Кбайт і масив контактних шпильок.

• *Pentium III* процесор добавив розширення потокового SIMD (SSE) в IA-32 архітектуру. Для підтримки SSE розширень добавлено набір 128-розрядних регістрів і здатність виконувати SIMD операції над числами з плаваючою крапкою звичайної точності.

• *Pentium III Xeon* процесор підвищив швидкодію процесорів IA-32 за рахунок повної швидкості, затримок і покращеного кешу передач.

7.7. Процесор Pentium IV

Процесори **Pentium IV** появилися в 2000-2006 роках і засновані на NetBurst архітектурі. Процесор Pentium IV використовував розширення потокового SIMD (SSE2). Процесор *Pentium IV 3.4 ГГц* використовував розширення SIMD (SSE3). Процесор *Pentium IV Extream Edition* підтримував багатопотокову технологію і 6xx та 5xx послідовності.

Технологія віртуалізації використовувалася в процесорах Pentium IV серії 671 і 662.

7.8. Процесор Xeon

Процесори **Xeon** появилися в 2001-2007 роках і засновані на мікроархітектурі NetBurst. Родина цих процесорів спроектована для використання в багатопроцесорних серверних системах і високопродуктивних робочих станціях. Процесори Xeon використовували технологію багатопотоковості.

64-розрядний процесор Xeon 3.60 ГГц (з 800 МГц системною шиною) започаткував архітектури Intel-64. Процесор *Dual-Core Xeon* започаткував двоядерну технологію. Xeon процесор серій 70xx використовував технологію віртуалізації.

В процесорах *Xeon серії 5100* використана енергоефективна, високопродуктивна мікроархітектура Intel Core. Процесор базувався на 64-розрядній архітектурі, технологіях віртуалізації і двоядерності. Процесор *Xeon серії 3000* також базувався на мікроархітектурі Intel Core. В процесорі *Xeon серії 5300* застосовано чотири ядра і він також базувався на мікроархітектурі Intel Core.

7.9. Процесор Pentium M

Процесори **Pentium M** появилася в 2003 році і є високопродуктивними, з низьким споживанням енергії і призначені для мобільних застосувань. Ця родина процесорів спроектована для збільшення терміну служби батарей живлення, має тонкий форм-фактор, можливості інтеграції в бездротові мережі.

До покращень мікроархітектури входить:

• підтримка Intel архітектури з динамічним виконанням;

• високопродуктивне ядро з малим енергоспоживанням, виготовлене по новій технології з мідними міжз'єднаннями.

• на пластині, основний 32 Кбайт кеш інструкцій і 32 Кбайт write-back кеш даних;

• на пластині, кеш другого рівня (до 2 Мбайт) з покращеною архітектурою передачі кеша;
• покращена логіка передбачення галужень і прогнозного вибору даних;
• підтримка MMX технології, інструкцій потокового SIMD і SSE2;
• 400 або 533 МГц синхронізована системна шина;
• просунута технологія керування живленням на основі Intel SpeedStep.

7.10. Процесор Pentium Extrim Edition

Процесор *Pentium Extrim Edition* появився в 2005-2007 роках і впровадив двоядерну технологію, яка забезпечує розширену апаратну підтримку багато потоковості. Процесор базується на мікроархітектурі NetBurst і підтримує SSE, SSE2, SSE3, багатопотоковість і 64-розрядну архітектуру.

7.11. Процесор Core Duo і Core Solo

Процесор появився в 2006-2007 роках і підтримує ефективність енергоспоживання і двоядерність. Родина двоядерних процесорів **Core Duo** і одноядерних **Core Solo** мають покращену мікроархітектуру порівняно з родиною процесорів Pentium M.

Покращення мікроархітектури наступні:

• Smart Cash, який підтримує ефективний розподіл даними між двома ядрами процесора;
• покращене декодування і SIMD виконання;
• покращене координування споживання енергії і стан спокою з меншим енергоспоживанням;
• покращене керування тепловиділенням з використанням цифрових термосенсорів;
• оптимізація споживання енергії 667 МГц шиною;

Двоядерний **Dual-core Intel Xeon LV** процесор має аналогічну мікроархітектуру, але підтримує архітектуру IA-32.

7.12. Процесори Xeon 5100, 5300 і Core 2

Процесори появилися в 2006 році. Процесори **Xeon** 3000, 3200, 5100, 5300, 7300 і Dual-Core, Core 2 Extreme, Core 2 Quad, Core 2 Duo підтримують 64-розрядну архітектуру. Вони основуються на енергоефективній Core мікроархітектурі виготовленій за 65 нм технологічним процесом.

Core мікроархітектура містить наступні інноваційні признаки:

• широкодинамічне виконання, яке збільшує продуктивність;
• інтелектуальне керування енергоспоживанням;
• покращений Smart Cashe, який забезпечує ефективний розподіл даними між двома ядрами;
• інтелектуальний доступ до пам’яті, який збільшує смугу пропускання даних і приховує затримки доступу до пам’яті;
• покращений цифровий медіа прискорювач, який покращує продуктивність за рахунок множинної генерації розширень потокового SIMD;

Процесори Xeon 5300, Core 2 Extreme QX6800 і Core 2 Quad підтримують 4-ядерну технологію.

7.13. Процесори Xeon 5200, 5400, 7400 і Core 2

Процесори появилися в 2007 році. Процесори **Xeon** 5200, 5400, 7400, *Core 2 Quad* Q9000, **Core 2 Duo** E8000 підтримують 64-розрядну архітектуру. Вони базуються на покращеній мікроархітектурі виготовленій за 45 нм технологічним процесом.

Мікроархітектура має наступні покращення:

• ділене з radix-16;

• покращений Smart Cashe, збільшений на 50% кеш 2-го рівня

• 128-розрядний суфлер значно покращив продуктивність медіа прискорювача і SSE4;

Процесори Xeon 5400, Core 2 Quad Q9000 підтримують 4-ядерну технологію. Процесор Xeon 7400 підтримує до шести процесорних ядер і кеш 3-го рівня розміром до 16 МБайт.

7.14. Процесори Atom

Процесори появилися в 2008 році. Процесори *Atom* побудовані за 45 нм технологічним процесом. Вони використовують нову Atom мікроархітектуру, яка оптимізована для надзвичайно низьковольтних пристроїв. В мікроархітектурі почергово використовуються два конвеєри, що понижає енергоспоживання. Процесор має надзвичайно малий форм фактор. Процесор підтримує наступні можливості:

• покращену SpeedStep технологію;

• багатопотокову технологію;

• технологія глибокого зниження енергоспоживання і динамічна зміна розміру кешу;

• підтримка нових інструкцій і розширень додаткового потокового SIMD (SSSE3);

• підтримка технології віртуалізації;

• підтримка 64-розрядної архітектури.

7.15. Процесори Core i7

Процесори появилися в 2008 році. Процесори **Core i7 900** підтримують 64-розрядну архітектуру, основані на мікроархітектурі під назвою Nehalem, виготовлені за 45 нм технологічним процесом. Процесори Core i7 і Xeon 5500 підтримують наступні можливості:

• Turbo Boost технологія, яка використовує тепло для підвищення продуктивності;

• гіперпотокова технологія у поєднання з 4-ма ядрами підтримує вісім потоків;

• спеціальний блок контролю енергії, який понижує її споживання як в а активному стані, так і в стані спокою;

• інтегрований з процесором контролер пам’яті, який підтримує три канали DDR3 пам’яті.

• 8 Мбайт інтелектуальний кеш;

• система швидкого з’єднання (QuickPath) точка-точка для набору IC;

• підтримка набору інструкцій для SSE4.1 і SSE4.2;

• друге покоління технології віртуалізації.

7.16. Xeon 7500

Процесори появилися в 2010 році. Процесори **Xeon** 7500, 6500 побудовані на мікроархітектурі Nehalen, виготовленій з використанням 45 нм технологічного процесу. В них добавлено наступні нововведення:

• до 8-ми ядер;

• до 24 Мбайт покращеного інтелектуального кешу;

• масштабований буфер пам'яті для під'єднання до системної пам'яті;

• покращений RAS для апаратного тестування апаратури з можливістю перепрограмування.

7.17. 2010 Core

2010 **Core** процесори об'єднують процесори Core i3, i5, i7. Вони базуються на мікроархітектурі Westmere, виготовленій з використанням 32 нм технологічного процесу. В них добавлено наступні нововведення:

• інтелектуальне керування продуктивністю з використання технологій гіперпотоковості та прискорювача Turbo Boost;

• покращений інтелектуальний кеш та інтегрований контролер пам'яті;

• інтелектуальний блок керування енергопостачанням;

• перепланована платформа з 45 нм інтегрованою графікою;

• набір інструкцій для роботи з AESNI, PCLMULQDQ, SSE4.1 і SSE4.2.

7.18. Xeon 5600

Процесори появилися в 2010 році. Процесори **Xeon 5600** базуються на мікроархітектурі Westmere, виготовленій з використанням 32 нм технологічного процесу. В них добавлено наступні нововведення:

• до 6-ти ядер;

• до 12 Мбайт покращеного інтелектуального кешу;

• набір інструкцій для роботи з AESNI, PCLMULQDQ, SSE4.1 і SSE4.2;

• гнучка технологія віртуалізації як для процесора так і для систем введення/виведення.

7.19. Друга генерація Core

Процесори появилися в 2011 році. Об'єднують процесори i3, i5, i7 побудовані на мікроархітектурі Sandy Bridge з використанням 32 нм технологічного процесу. В них добавлено наступні нововведення:

• технологія прискорювачів Turbo Boost для i5, i7;

• гіперпотокова технологія;

• покращений інтелектуальний кеш та інтегрований контролер пам'яті;

• графічний процесор з підтримкою відео;

• набір інструкцій для роботи з AVX, AESNI, PCLMULQDQ, SSE4.1 і SSE4.2.

7.20. Лінійка процесорів Intel

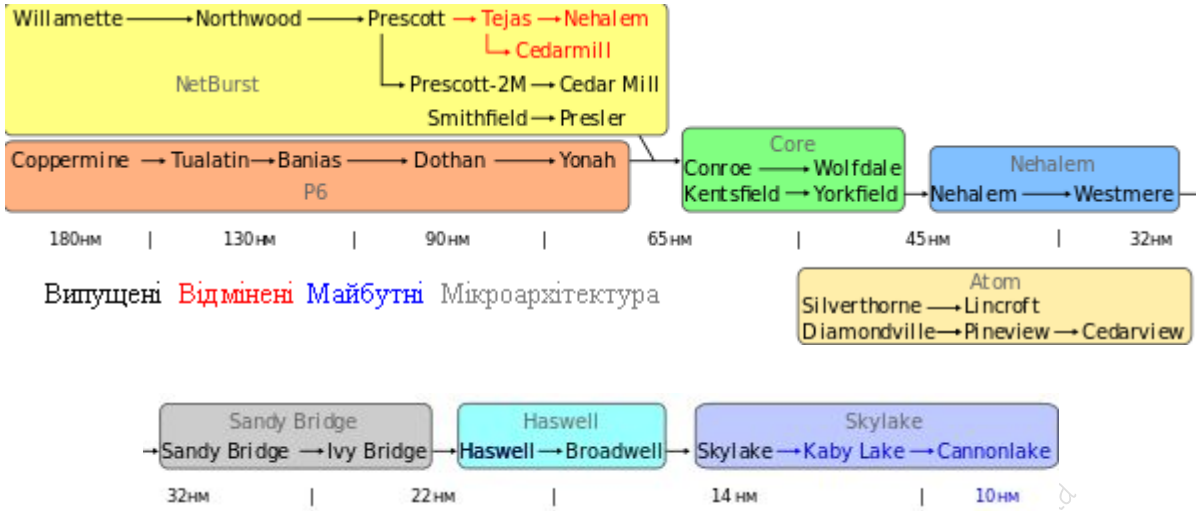


Рисунок 10 – Лінійка процесорів Intel

Архітектура x86-64 використовувалася в МП Celeron, Core2, Corei3, Corei5, Corei7, деякі моделі Inel Atom.

Архітектура IA-64 використовувалася в МП Itanium, Itanium2. Випуск лінійки Itanuim заснованій на VLIM (в термінах Intel EPIC) припиняється з 2021 року.

Архітектури x86-64 і IA-64 є різними і несумісними. Intel повертається до архітектури X86-64 на основі процесорів Xeon E7 (15-ядерних).

7.21. Процесори i9.

8-ядерний процесор Core-i9 призначений для використання high-end ноутбуках. Має 12 Мбайт кеш-пам'яті третього рівня і тактову частоту 5 ГГц, контролер пам'яті DDR4-2666 і відеоядро Intel UHD Graphics 630.

7.22. Процесори Intel Core 11

Процесори Core 11-го покоління виготовлені за 10-нанометровим техпроцесом. Нові процесори також отримали низку особливостей: підтримку PCIe 4.0, інтегрований контролер Thunderbolt 4, адаптер Wi-Fi 6 (Gig +), апаратну підтримку Dolby Vision, підтримку даних типу INT8, вбудовані механізми Control Flow Enforcement Technology (CET) і Total Memory Encryption (TME) , підтримку кодека AV1, підтримку Intel Adaptix і наявність апаратного блоку прискорення II Intel GNA 2.0 (Gaussian & Neural Acceleration).

8. Середовище виконання програм

До середовище виконання програм у процесорі x86-64 відноситься:

• Адресний простір – лінійний до 2^{64} байт, фізичний до 2^{40} байт.

• Основні регістри для виконання програм – 16 регістрів загального призначення (64-біт), 6 сегментних регістрів (16-біт), регістр прапорів (64-біт), регістр вказівник команд (64-біт),

• 8 регістрів процесора з плаваючою крапкою (80-біт);

• 8 MMX регістрів (64-біт) для підтримки операцій “одна інструкція – багато даних” (SIMD) при обробленні 64-бітного пакету цілих чисел розміром байт, слово, подвійне слово, почотирне слово.

• 16 XMM регістрів (128-біт) і MXCSR регістр для підтримки SIMD операцій з 128-бітними пакетами чисел з плаваючою крапкою звичайної і подвійної точності і також 128-бітних пакетів цілих чисел розміром байт, слово, подвійне слово, четвертне слово.

• Стек розміщується у пам’яті і призначений для підтримки викликів підпрограм та передачі параметрів між програмами та підпрограмами.

• Порти введення-виведення призначені для передавання/приймання даних. Для них виділена спеціальні область пам’яті.

• Контрольні регістри CR0-CR4 визначають режими роботи процесора і характеристики виконуваної задачі, підтримують віртуальну адресацію.

• Контрольний регістр CR8 підтримує механізм переривань.

• Регістр EFER ще один регістр прапорів, для контролю режимів роботи процесора і оброблення системних викликів.

• Регістри керування пам’яттю GDTR (регістр для зберігання адрес глобальної дескрипторної таблиці), IDTR (регістр для зберігання адрес таблиці дескрипторів переривань), TR (регістр задачі), LDTR (регістр для зберігання адрес локальної дескрипторної таблиці) – визначають розміщення структур даних і використовуються для керування пам’яттю у захищеному режимі.

• Регістри налагодження DR0-DR7, які використовуються для підтримки роботи процесора у режимі налагодження.

• Регістри діапазону типів пам’яті MTRR призначені для підтримки різних типів різних областей пам’яті.

• Машино специфічні регістри MSR використовуються для контролю і повідомлень про функціювання процесора.

• Регістри машинного тестування використовують набори повідомлень з MSR регістрів для виявлення і повідомлення про апаратні помилки.

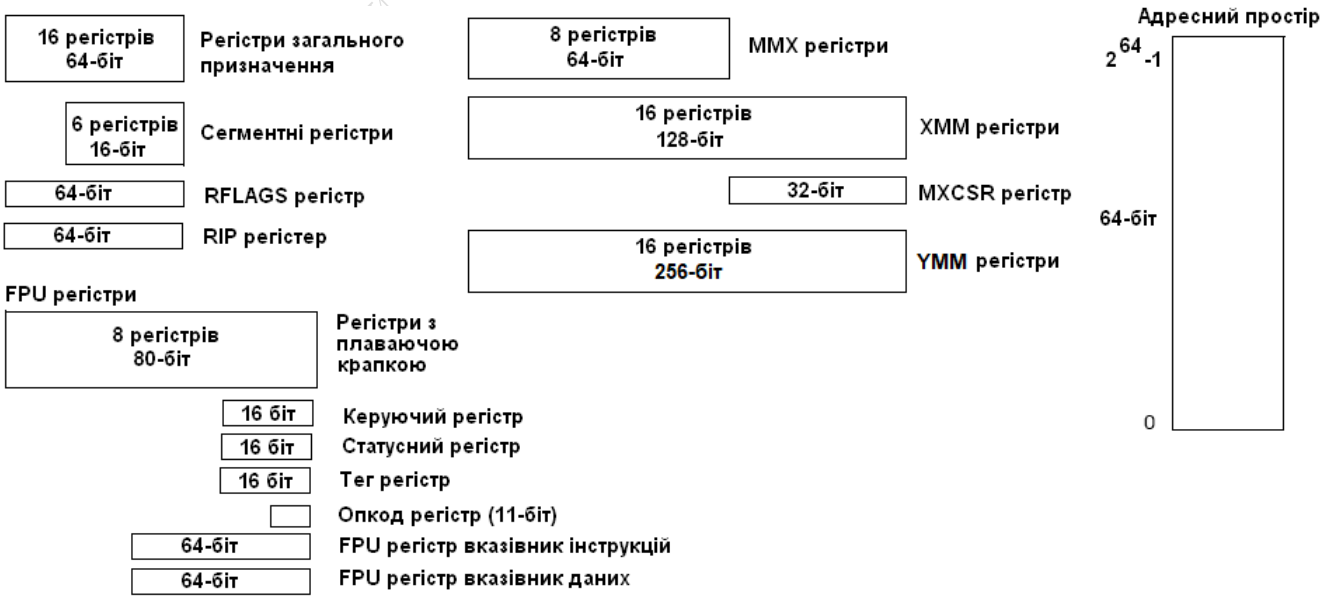


Рисунок 11 – Програмне середовище процесора x86-64

Контрольні запитання.

1. Чим відрізняється системне програмування від прикладного.
2. Що таке комірка пам’яті і машинне слово.
3. Які послідовності байтів використовуються для зберігання чисел.
4. Що таке асемблер.
5. В яких режимах може працювати процесор.
6. Що таке віртуальна пам’ять.
7. Машинне подання беззнакових і знакових цілих чисел.
8. Що таке доповняльний код числа і як його отримати.
9. Формат подання чисел з плаваючою крапкою одинарної і подвійної точності.
10. Особливості архітектури CISC, RISC і MISC процесорів.
11. Середовище виконання програм у процесорах Intel x86-64.

2. МОДЕЛІ ПАМ'ЯТІ І РЕГІСТРИ

Мета. Ознайомлення з моделями пам'яті і регістрами процесорів Intel x86-64.

Вступ. Фірма Intel випускає процесори, які використовуються у більшості персональних комп'ютерів. З розвитком технологій початкова розрядність процесорів (8-розрядів) подвоювалася до 16, 32 і на даний час до 64-розрядів. Відповідно змінювалася архітектура процесорів і ОС, які на них виконувалися. Для забезпечення зворотної сумісності програм сучасні процесори x86-64 підтримують три режими роботи – реальний, захищений і довгий. В цих режимах доступні як 32- так і 64-розрядні регістри.

План.

1. Типи адрес і розміри областей пам'яті
2. Режими роботи процесорів
3. Реальний режим
   3.1. Реальний режим плоскої моделі пам'яті
   3.2. Реальний режим сегментованої моделі пам'яті
4. Захищений 32-розрядний режим плоскої моделі пам'яті
5. Довгий 64-розрядний режим пам'яті
6. Регістри процесорів x86-64
   6.1. Регістри загального призначення
   6.2. Сегментні регістри
   6.3. Регістр вказівник команд і регістр прапорів
   6.4. Регістри математичного співпроцесора
   6.5. MMX регістри
   6.6. XMM регістри
   6.7. YMM регістри
7. Службові регістри
   7.1. Керуючі регістри
   7.2. Системні адресні регістри
   7.3. Регістри налагодження

1. Типи адрес і розміри областей пам'яті

В процесорах x86-32/64 використовується три типи адрес: фізична, логічна, лінійна (або віртуальна).

Фізична адреса – це адреса в системній пам'яті комп'ютера. Саме ця адреса виставляється на шину адрес.

Логічна адреса завжди задається у форматі "сегмент:зміщення".

Лінійна адреса – це логічна адреса яка перетворюється у абсолютну 20-, 32-, 64-розрядну адресу (в залежності від режиму процесора).

В режимі реальних адрес лінійна адреса зразу виставляється на шину адрес. В захищеному і 64-розрядному режимах лінійну адресу можна назвати *віртуальною*, якщо активований механізм трансляції сторінок. Якщо механізм трансляції сторінок не активований, то лінійна адреса стає фізичною, тобто без перетворень виставляється на шину адрес. Якщо ж механізм

трансляції сторінок включений, то лінійна (віртуальна) адреса спеціальним чином перетворюється у фізичну адресу; спосіб перетворення задається самою ОС. Процесор підтримує наступні розміри областей пам'яті:

Таблиця 1 – Розміри областей пам'яті

Назва	10-ве значення	16-ве значення
Byte (b)	1	01h
Word (w)	2	02h
Double word (d, l)	4	04h
Quad word (q)	8	08h
Ten byte (t)	10	0Ah
Double quad	16	10h
Paragraph	16	10h
Page	256	100h
Segment	65536	10000h

2. Режими роботи процесорів

На даний час одними з найпоширеніших є процесори x86-32/64, які можуть працювати в різних режимах і з різними моделями пам'яті.

Процесори x86-32/64 переважно працюють у трьох основних режимах: реальному (real mode flat model, real mode segmented model), захищеному (protected mode flat model) і 64-розрядному (long mode flat model):

• **реальний режим** – це режим, в який переходить процесор після ввімкнення або перевантаження. Це стандартний 16-розрядний режим, у якому доступний тільки 1 МБайт фізичної пам'яті і можливості процесора майже не використовуються, а якщо і використовуються, то незначно. Іноді цей режим називають режимом реальних адрес, тому що в ньому не можна активувати механізм трансляції віртуальних адрес у фізичні. Це значить, що всі адреси, до яких звертаються програми, є фізичними, тобто без якого-небудь перетворення будуть виставлені на шину адрес. У цьому режимі процесор працює з 2-ма байтами (слово, word).

• **захищений режим** (protected mode або legacy mode) – це 32-розрядний режим, який для процесорів x86 є головним. У захищеному режимі 32-розрядна ОС може отримати максимальну віддачу від процесора. В цьому режимі є доступ до $2^{32} \approx 4$ Гбайт фізичного адресного простору, а при включенні спеціального механізму трансляції адрес можна отримати доступ до 64 Гб фізичної пам'яті. У захищений режим можна перейти тільки з реального режиму. Захищений режим називається так тому, що дозволяє захистити дані ОС від програм користувача. У цьому режимі процесор працює з 4-ма байтами (подвійне слово, dword). Всі операнди, які задають адреси у цьому режимі мають бути 32-бітовими.

• **64-розрядний (довгий) режим** (long mode або IA-32e) – за принципом роботи він майже подібний до захищеного режиму, крім деяких аспектів. У цьому режимі можна отримати доступ до 2^{52} байт фізичної пам'яті і до 2^{48} байт віртуальної пам'яті. В 64-розрядний режим можна перейти тільки із захищеного режиму. У цьому режимі процесор працює з 4-ма байтами (подвійне слово, dword), але може оперувати з даними розміром 8 байт (qword). Розмір адрес завжди 8-байтовий.

Крім вказаних процесор x86-64 підтримує два *підрежими*:

• **режим віртуального процесора 8086** – це підтримка захищеного режиму для старих 16-розрядних програм. Його можна включити для окремої задачі в багатозадачній ОС захищеного режиму;

• **режим сумісності для довгого режиму**. В режимі сумісності програмам доступні 4 Гбайт пам'яті і повна підтримка 16- і 32-розрядного коду. Режим сумісності можна включити для окремої задачі у багатозадачній 64-розрядній ОС. В режимі сумісності розмір адреси 32-бітовий, а розмір операнду не може бути 8-байтовим.

На рис. 1 показана діаграма режимів роботи процесора і можливості переходу з одного режиму у інший.

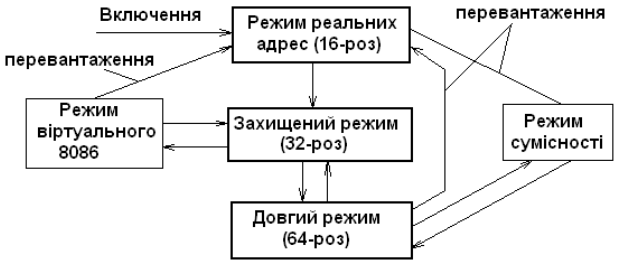


Рисунок 1 – Діаграма режимів роботи процесора x86-64

3. Реальний режим

Перші процесори 8080, 8086 і 80286 були 16-розрядними і мали *16-розрядні регістри* та забезпечували доступ до 1 Мбайт пам'яті. 16-розрядний регістр може забезпечити адресацію 2^{16}=65536 комірок, а потрібно адресувати 2^{20}=1_048_576 комірок. Виникає запитання, як 16-розрядним регістром адресувати 1_048_576 комірок пам'яті? Для цього прийнята домовленість, що кожна комірка знаходиться у сегменті, а її адреса складається з двох частин: **адреси сегменту і адреси зміщення** (віддалі в байтах комірки від початку сегменту). Таку адресація прийнято вказувати через двокрапку – **адреса_сегменту:адреса_зміщення**. Адреси завжди записують шістнадцяткові.

Адреси сегментів можуть починатися з параграфів, тобто через кожних 16 байт у межах реальної мегабайтової пам'яті, рис. 2.

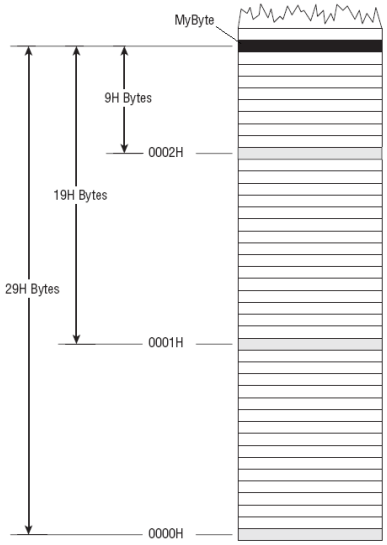


Рисунок 2 – Адресація комірки MyByte з використанням різних сегментів і зміщень

Адреса сегменту задає один з 65535 слотів з яких може починатися сегмент. Сегменти можуть перекриватися і тому адресу однієї комірки пам'яті можна задати з використанням різних сегментів і зміщень, наприклад для рис.2: 0000h:0028h, 0001h:0019h, 0002h:0009h.

Таким чином, для адресації мегабайтової пам'яті використовується два 16-розрядні регістри, один для зберігання адреси сегменту, а інший – для зберігання зміщення.

Процесори 8080, 8086, 80286 мали чотири сегментних регістри CS, DS, SS, ES для зберігання адрес сегментів. У наступних моделях процесорів 386 і x86 добавлено сегментні регістри FS і GS. Всі сегментні регістри є 16-розрядними.

Реальний режим підтримує дві моделі пам'яті:

– плоску;

– сегментовану.

Призначення сегментних регістрів:

• CS (code segment) – сегмент коду. Машинні інструкції розміщуються з деяким зміщенням у сегменті коду. Сегментний регістр містить адресу сегменту, інструкції якого виконуються в даний момент часу (використовується для IP).

• DS (data segment) – сегмент даних. Змінні і інші дані розміщуються з деяким зміщенням у одному сегменті даних. В програмі може використовуватися декілька сегментів даних. Центральний процесор (ЦП) може використовувати в один момент часу тільки один з сегментів, розміщуючи його адресу в сегментний регістр (використовується для MOV).

• SS (stack segment) – сегмент стеку. Стек використовується для тимчасового зберігання даних і адрес. Стек займає сегмент і його адреса зберігається в регістрі стеку.

• ES (extra segment, destination segment) – додатковий сегмент. Використовується як резервний для адресації пам'яті (використовується для MOVS).

• FS і GS – додаткові регістри (позначення взяті з послідовності букв E, F, G) не мають спеціального призначення і використовуються з різною метою.

3.1. Реальний режим плоскої моделі пам'яті

У реальному режимі **плоскої моделі пам'яті** програмі доступні тільки 64 Кбайт пам'яті з підтримуваного центральним процесором 1 Мбайт. Схема плоскої моделі пам'яті для реального режиму показана на рис. 3.

Всі сегментні регістри вказують на початок 64 Кбайт блоку пам'яті в якому буде виконуватися програма і розміщуватимуться дані. ОС заповнює ці сегменти коли вона завантажує програму і починає її виконання. За час виконання програми вони не змінюються. Так як 16-розрядний регістр, наприклад BX, може містити значення від 0 до 65535, то він може адресувати любу комірку з 64 Кбайт пам'яті де виконується програма без використання сегментних регістрів. У цьому режимі програміст не має доступу до сегментних регістрів.

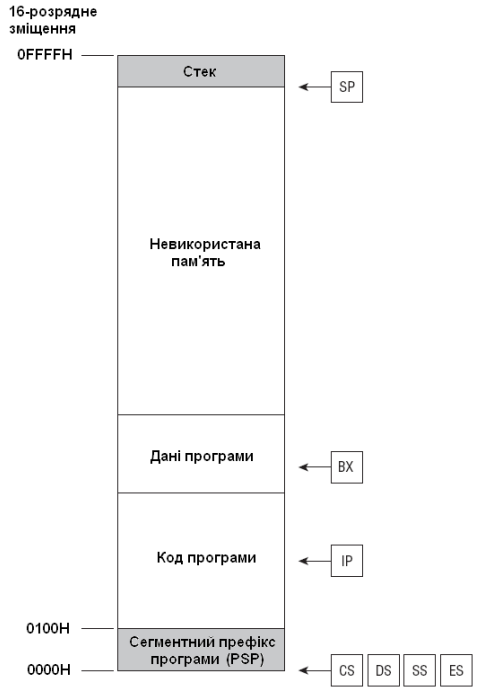


Рисунок 3 – Схема плоскої моделі пам’яті для реального режиму: SP – стек (LIFO буфер), BX – один з регістрів загального призначення, який містить адресу даних, IP – регістр, який містить адресу команди, як буде виконуватися наступною, CS, DS, SS, ES – сегментні регістри, значення яких завантажується ОС

3.2. Реальний режим сегментованої моделі пам’яті

У реальному режимі сегментованої моделі пам’яті програмі доступний увесь 1 Мбайт пам’яті, який підтримує ЦП у реальному режимі. Це досягається за рахунок комбінації 16-розрядної адреси сегменту і 16-розрядної адреси зміщення. ЦП перетворює комбінацію адрес сегменту і зміщення у внутрішню 20-розрядну адресу. Фізична адреса обчислюється апаратно за формулою:

```
фізична адреса = база сегмента * 16 + зміщення
```

В дійсності сегментний регістр містить не стартову адресу, а базу – старші чотири шістнадцяткові розряди. Додаючи нульовий шістнадцятковий розряд, що аналогічно множенню на 16, отримується реальна стартова адреса сегменту.

Для задання адрес сегменту і зміщення можуть використовуватися різні комбінації сегментних регістрів – SS:SP, SS:BP, ES:DI, DS:SI, CS:BX. Схема сегментованої моделі пам’яті для реального режиму показана на рис. 4.

На рис. 4 показано два сегменти даних і два сегменти коду. На практиці можна використати і більше число сегментів. Два різні сегментні регістри забезпечують одночасний доступ до двох сегментів даних. У процесорах 386 використовуючи сегментні регістри FS, GS можна отримати доступ ще до двох сегментів даних. Можна пересилати дані з одного сегментного регістра в інший.

Однак є тільки один сегментний регістр коду CS. CS завжди містить адресу сегменту з поточним кодом, а адреса інструкції, яка буде виконуватися наступною, міститься в регістрі IP (instruction pointer). Не можна безпосередньо завантажити значення в регістр CS для зміни

одного сегменту коду на інший. При необхідності, для зміни сегменту коду, використовуються інструкція `jumps`. Програма може займати декілька сегментів коду і коли інструкція `jumps` передає керування в інший сегмент коду, вона змінює значення сегментного регістра CS.

Програма має тільки один сегмент стеку, адреса якого зберігається в сегментному регістрі SP. При розміщенні значень у стек, їх адреси зменшуються.

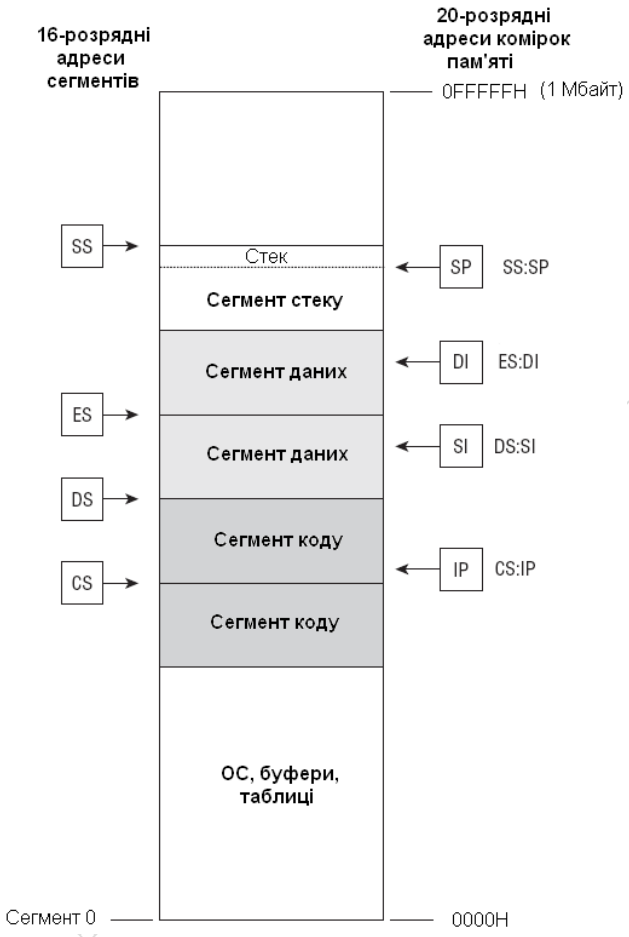


Рисунок 4 – Схема сегментованої моделі пам’яті для реального режиму:

Необхідно зазначити, що в реальному режимі в 1 Мбайт пам’яті розміщується як ОС з своїми буферами і системними таблицями, так і програма користувача. При некоректному використанні сегментних регістрів можна зруйнувати частину ОС, що спричинить її крах. З цієї причини в процесорах, починаючи з 80386, добавлено захищений режим. У захищеному режимі програма користувача не може зруйнувати ОС або інші програми.

4. Захищений 32-розрядний режим плоскої моделі пам’яті

Захищений режим плоскої моделі пам’яті з'явився в процесорах 386 у 1986 році і був реалізований в Linux, в 1992 році, та в Windows NT, в 1996 році. Найпростіше в захищеному режимі створювати консольні застосунки.

Схема захищеного режиму плоскої моделі пам’яті показана на рис. 5. Всі регістри загального призначення і регістр вказівник команд EIP (в позначеннях 32-розрядних регістрів

добавлена буква E (extended)) є 32-розрядними, тому вони потенційно мають доступ до всього адресного простору від 0 до 4 Гбайт пам’яті (2^{32}). Звичайно деяка частина адресного простору виділяється під ОС. Любий регістр загального призначення може містити адресу з області пам’яті для ОС, але спроба прочитати або записати у цю область викличе помилку часу виконання.

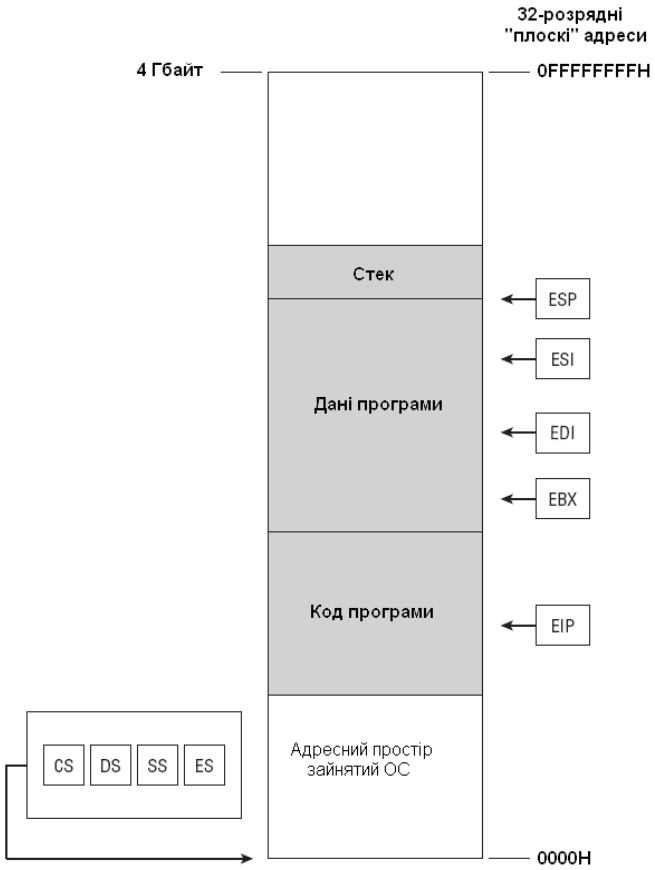


Рисунок 5 – Схема плоскої моделі пам’яті для захищеного режиму

Так як любий регістр загального призначення може адресувати увесь адресний простір, то сегменті регістри DS, CS, SS, ES використовуються для потреб ОС.

Захищений режим використовував нові механізми захисту: захисні кільця, віртуальну пам'ять, покращену сегментацію. Захищений режим дозволяє декільком програмам виконуватися в один і той самий момент часу. Для забезпечення цього майже всі низькорівневі звернення до апаратури здійснюються через інстальовані драйвери. Так доступ до відеопам’яті забезпечується програмами драйверами, які виконуються у просторі ядра. Доступ до портів здійснюється також за допомогою драйверів і бібліотек. Використання драйверів як інтерфейсу до портів, є значно простішим, ніж контроль самих портів. Виклики BIOS також виконує ОС. Linux надає список низькорівневих функцій, які можуть бути викликані через програмне переривання syscall.

У 32-бітовому захищеному режимі також використовується сегментована модель пам'яті, проте вже організована за іншим принципом: розташування сегментів описується спеціальними структурами (таблицями дескрипторів), розташованими в оперативній пам'яті. Стартова адреса сегмента обчислюється з використанням спеціальної таблиці:

```
Лінійна адреса = база сегменту (з системної таблиці) + зміщення
```

Сегменти пам'яті також вибираються все тими ж сегментними регістрами. Значення сегментного регістра (сегментний селектор) більше не є просто адресою, а є структурою, рис. 6.

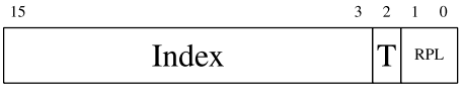


Рисунок 6 – структура сегментного селектора

Індекс задає номер дескриптора у глобальній чи локальній дескрипторній таблиці. Біт T=0 для глобальної таблиці і T=1 – для локальної. Поле RPL (Requested Privilege Level) визначає рівень привілеїв.

Кожний сегментний регістр CS, DS, SS, ES, GS і FS зберігає сегментний селектор, який містить індекс спеціальної таблиці сегментних дескрипторів і додаткову інформацію.

Є два типи дескрипторних таблиць: глобальна GDT (Global Descriptor Table) і локальна LDT (Local Descriptor Table). Глобальна таблиця описує сегменти ОС і спільно використовувані структури даних. У кожного ядра вона своя.

Локальна таблиця може бути визначена для кожного процесу і їх може бути багато. Вони призначені для апаратного механізму перемикання задач, але розробники ОС не використали їх. Тому зараз програми ізолюються за допомогою віртуальної пам’яті.

Таблиці GDT/LDT також зберігають інформацію про рівні привілеїв, які призначені сегментам. При зверненні до сегмента через селектор сегмента перевіряється значення Request Privilege Level (RPL) (яке зберігається у сегментному селекторі) з Descriptor Privilege Level (яке зберігається у дескрипторній таблиці). Якщо RPL не має достатньо привілеїв для доступу до сегменту з високими привілеями, то формується помилка. Таким чином можна створити багато сегментів з різними правами доступу і використовувати значення RPL у сегментних селекторах для визначення їх доступності для різних рівнів привілеїв. Рівні привілеїв є аналогом захисних кілець. Поточні привілеї зберігаються в молодших двох бітах регістрів CS і SS. На рис. 7 показано структуру сегментного дескриптора.

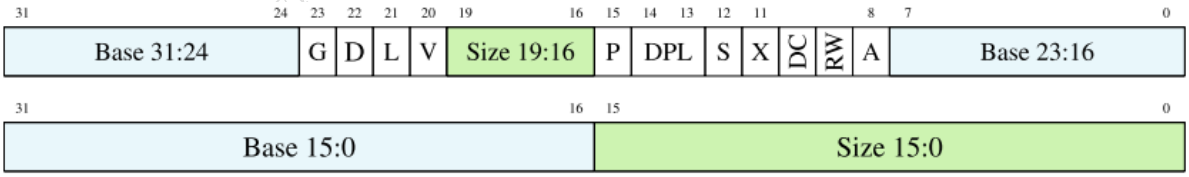


Рисунок 7 – Сегментний дескриптор (для GDT і LDT дескрипторних таблиць):

G – зернистість, 0 = bytes, 1 = сторінки розміром 4096 байтів;

D – розмір операнда за замовчуванням (0 = 16 bit, 1 = 32 bit);

L – чи сегмент є в 64-bit режимі?

V – доступний для програм;

P – є присутніми в пам'яті прямо зараз;

S – це кодовий/набір(1) або об’єкт деякої системної інформації(0);

X – data (0) або code (1);

RW – чи дозволений запис для сегменту data? (читання завжди дозволене); чи дозволене читання для сегменту code? (запис завжди заборонений);

DC – напрямок росту: до більших чи до менших адрес? (для сегменту data); чи можна виконати з більшими рівнями привілеїв? (для сегменту code);

A – чи було це доступно?

DPL – рівень привілеїв дескриптора (Descriptor Privilege Level).

Рівні привілеїв це механізм за допомогою якого ОС і ЦП обмежують можливості програм для користувача режиму. Існує чотири рівні привілеїв (0 – найбільший для ядра, 3 – найменший для користувача) і три основних ресурси: пам'ять, порти введення-виведення і можливість виконання певних машинних інструкцій. У будь-який момент часу процесор x86-64 працює на певному рівні привілеїв, який визначає, який код що може і не може зробити. Ці рівні привілеїв часто описуються як захисні кільця, причому саме внутрішнє кільце відповідає найвищим привілеям. Більшість сучасних ядер x86-64 використовують тільки Ring 0 і Ring 3.

Системні регістри GDTR, LDTR, IDTR призначені для зберігання базових адрес **таблиць дескрипторів** – найважливіших складових системної архітектури при роботі у захищеному режимі.

Регістр GDTR містить 32-розрядну базову адресу і 16-розрядний розмір **глобальної таблиці дескрипторів** GDT.

Видима частина регістра LDTR містить тільки селектор дескриптора **локальної таблиці дескрипторів** LDT. Сам дескриптор LDT автоматично завантажується у приховану частину LDTR із глобальної таблиці дескрипторів.

Регістр IDTR містить 32-розряду базову адресу і 16-розрядний розмір **таблиці дескрипторів переривань** IDT.

Видима частина регістра TR містить селектор дескриптора стану задачі TSS. Сам дескриптор автоматично завантажується у приховану частину TR з головної таблиці дескрипторів.

5. Довгий 64-розрядний режим пам'яті

Процесори x86-64 є 64-розрядними і для забезпечення зворотної сумісності підтримують реальний режим (плоскої і сегментованої моделі пам'яті), захищений режим і довгий режим. В реальному режимі процесор працює як 8086 або інші x86 процесори в реальному режимі. В захищеному режимі процесор працює як IA-32 процесори. В довгому 64-розрядному режимі сегментація не використовується.

Для чотирьох сегментних регістрів (CS, SS, DS і ES) базову адресу примусово виставляють в 0. Сегментні регістри FS і GS як і раніше можуть мати ненульову базову адресу. Це дозволяє ОС використовувати їх для службових цілей.

В довгому режимі можна адресувати величезний обсяг пам'яті – 2^{64} байт. На практиці в процесорах x86-64 реалізовано 48-розрядну адресацію віртуальної пам'яті і 40-розрядну адресацію фізичної пам'яті ($2^{40} = 10^{12}$ байт = 1 Тбайт).

6. Регістри процесорів x86-64

В архітектурі процесорі x86-64 є 64-, 32- і 16-розрядні регістри. Всі регістри поділяються на три групи:

• *регістри загального призначення (регістри даних);*

• *сегментні регістри;*

• *регістри стану і керуючі регістри.*

Регістри загального призначення в свою чергу також поділяються на три групи:

• *регістри даних;*

• *регістри вказівники;*

• *індексні регістри.*

В захищеному режимі, в режимі реальних адрес і режимі сумісності доступні наступні регістри:

• регістри загального призначення:

- 32-розрядні (8 шт) EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP;

- 16-розрядні (2^{16}=65536) AX, BX, CX, DX, SI, DI, SP, BP (вони є молодшими частинами 32-розрядних регістрів);

- 8-розрядні (2^8=256) регістри AH, BH, CH, DH і AL, BL, CL, DL (старші і молодші частини 16-розрядних регістрів відповідно);

• 32-розрядний EIP (16-розрядний IP в реальному режимі) – вказівник інструкції;

• 16-розрядні сегментні регістри: CS, DS, SS, ES, FS, GS;

• 32-розрядний регістр прапорів – EFLAGS;

• 80-розрядні регістри математичного співпроцесора ST0-ST7 та інші;

• 64-розрядні MMX-регістри MM0-MM7;

• 128-розрядні XMM-регістри XMM0-XMM7 і 32-розрядний MXCSR;

• 32-розрядні регістри керування CR0-CR4; регістр-вказівники системних таблиць GDTR, LDTR, IDTR і регістр задачі TR;

• 32-розрядні регістри налаштування DR0-DR3, DR6, DR7;

• MSR-регістри.

В режимі реальних адрес доступні не всі вищевказані регістри, але регістри керування доступні у любому випадку. В режимі реальних адрес не можна використовувати деякі регістри розміром більше 16 біт.

При перемиканні процесора у 64-розрядний режим програмі доступні наступні регістри:

• регістри загального призначення:

- 64-розрядні (16 шт) RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP і R8, R9,..., R15;

- 32-розрядні EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D-R15D (є молодшими частинами 64-розрядних регістрів);

- 16-розрядні AX, BX, CX, DX, SI, DI, SP, BP, R8W-R15W (є молодшими частинами 32-розрядних регістрів);

- 8-розрядні регістри AH, BH, CH, DH і AL, BL, CL, DL, SIL, DIL, SPL, BPL, R8L-R15L (старші і молодші частини 16-розрядних регістрів відповідно);

• 64-розрядний RIP – вказівник інструкції;

• 16-розрядні сегментні регістри: CS, DS, SS, ES, FS, GS;

• 64-розрядний регістр прапорів – RFLAGS;

• 80-розрядні регістри математичного співпроцесора ST0-ST7;

• 64-розрядні MMX-регістри (MM0-MM7);

• 128-розрядні XMM-регістри XMM0-XMM15 і 32-бітовий MXCSR;

• 256-розрядні YMM-регістри YMM0-YMM15;

• 64-розрядні регістри керування процесором CR0-CR4 і CR8;

• регістри-вказівники системних таблиць GDTR, LDTR, IDTR і регістр задачі TR;

• 64-розрядні регістри налагодження DR0-DR3, DR6, DR7;
• Тестувальні регістри TR3, TR4, TR5, TR6, TR7.
• MSR-регістри.

6.1. Регістри загального призначення

Регістри загального призначення використовуються для зберігання:
- операндів арифметичних і логічних виразів;
- компонент адрес;
- вказівників на комірки пам'яті.
64/32-розрядні регістри загального призначення використовуються як:

Регістри даних **rax/eax** (Accumulator register) – акумулятор, використовується для зберігання даних проміжних обчислень;	*Регістри вказівники для роботи зі стеком* **rsp/esp** (Stack pointer register) – регістр вказівник стеку. Містить адресу верхівки стеку.
rbx/ebx (Base register) – базовий регістр, використовується для зберігання базової адреси деякого об'єкта в пам'яті;	**rbp/ebp** (Base pointer register) – регістр вказівник бази кадру стека. Призначений для організації довільного доступу до даних всередині стеку.
rcx/ecx (Counter register) – регістр лічильник, неявно використовується в деяких командах для організації циклів;	*Індексні регістри для підтримки ланцюжкових операцій* **rsi/esi** (Source index register) – індекс джерела, в ланцюгових командах містить поточну адресу елемента джерела;
rdx/edx (Data register) – регістр даних, використовується для зберігання результатів проміжних обчислень і введення-виведення;	**rdi/edi** (Destination index register) – індекс приймача, в ланцюгових командах містить поточну адресу елемента приймача.

До регістрів `rax`, `rbx`, `rcx`, `rdx` можна звертатися по "частинах". Наприклад до молодшого 32-біт регістра `rax` можна звернутися як до `eax`, до молодшого 16-біт регістра `eax` можна звернутися як до `ax`, `ax` свою чергу містять дві однобайтні половинки, до яких можна звернутися як до `ah` (старшого), `al` (молодшого) байту.

Інструкції, в яких використовуються регістри `rsp`, `sbp`, `rsi`, `rdi`, `rx (x=8...10)`, дають доступ до молодших частин регістрів `spl, bpl, sil, dil, rxb.`

В процесорах x86-64 регістри 64-розрядні і відповідно позначаються з префіксом "r": `rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi.` Додатково добавлено вісім нових 64-розрядних регістрів `r8, r9,…, r15` в яких префікс "`r`" вказує, що в не 64-розрядних інструкціях довжина інструкції може мати різні розміри: `r8b` – байт (8-бітів), `r8w` – слово (16-бітів), `r8d` – подвійне слово (32- біти), `r8` – почотирне слово (64-біти).

```
64-bit регістри |Молодші 32 біти| Молодші 16 біт| Молодші 8 біт
```

```
-------------------------------------------------------------
rax             | eax (r0d)     | ax (r0w)      | al  (r0b)
rbx             | ebx (r1d)     | bx (r1w)      | bl  (r1b)
rcx             | ecx (r2d)     | cx (r2w)      | cl  (r2b)
rdx             | edx (r3d)     | dx (r3w)      | dl  (r3b)
rsi             | esi (r4d)     | si (r4w)      | sil (r4b)
rdi             | edi (r5d)     | di (r5w)      | dil (r5b)
rbp             | ebp (r6d)     | bp (r6w)      | bpl (r6)
rsp             | esp (r7d)     | sp (r7w)      | spl (r7b)
r8              | r8d           | r8w           | r8b (r8b)
r9              | r9d           | r9w           | r9b (r9b)
r10             | r10d          | r10w          | r10b
r11             | r11d          | r11w          | r11b
r12             | r12d          | r12w          | r12b
r13             | r13d          | r13w          | r13b
r14             | r14d          | r14w          | r14b
r15             | r15d          | r15w          | r15b
```

Однією з особливостей роботи з 64-розрядними регістрами є «затирання» їх верхньої половини командами, які оперують з 32-розрядними операндами:

```
mov eax, 0F0F0AABBh
shl rax, 32   ; rax = F0F0AABB00000000h
mov eax, 2    ; rax = 0000000000000002h
```

6.2. Сегментні регістри

До сегментних регістрів відносяться наступні:

- **cs** (Code segment) – сегмент коду, містить адресу сегменту з машинними командами. Зв'язка `cs:ip` (`cs:eip`/`cs:rip` – в захищеному/64-бітному режимі) вказує на адресу в пам'яті наступної команди.
- **ds** (Data segment) – сегмент даних, містить адресу сегменту даних, які обробляє програма.
- **ss** (Stack segment) – сегмент стеку, містить адресу області пам’яті, яку використовує стек.

Додаткові сегменти даних. Якщо програмі недостатньо одного сегменту даних, то вона може використати ще три додаткових сегменти даних, адреси яких зберігаються в регістрах `es, fs, gs`.

- **es** (Extra segment) – додатковий сегмент, який використовується неявно в командах роботи з символьними рядками як сегмент отримувач.
- **fs** (F segment) – додатковий сегментний регістр без спеціального призначення.
- **gs** (G segment) – додатковий сегментний регістр без спеціального призначення.

Сегментні регістри містять адреси сегментів пам’яті, а саме: CS – сегмент коду, DS – сегмент даних, SS – сегмент стеку; решта три регістри додаткові і можуть не використовуватися програмою. Вільна робота з ними не завжди можлива; наприклад в захищеному і 64-розрядному режимах завантажувати у них можна лише певні значення. В захищеному і 64-розрядному режимах доступність регістрів залежить від рівня привілей, на яких виконується програма. Регістр загального призначення RSP (ESP) завжди вказує на верхівку стеку, але при цьому ніщо не заважає використовувати його з іншою метою, хоча тоді буде втрачена можливість нормальної роботи зі стеком. Взагалі всі регістри загального призначення можна вільно використовувати у своїх цілях, але слід пам’ятати, що деякі регістри використовуються деякими

командами: наприклад, RBP (EBP) звичайно вказує на початок фрейму у стеку, де зберігаються локальні дані підпрограм. Вказівник інструкції RIP (EIP) напряму використовувати не можна, так як він використовується процесором.

6.3. Регістри стану - вказівник команд і регістр прапорів

До регістрів стану відносяться регістр вказівник інструкцій і регістр прапорів.

Регістр вказівник інструкцій *rip/eip/ip* (Instruction Pointer register) – містить зміщення відносно вмісту сегментного регістра `cs` (`cs+offset` задають адресу) наступної команди, яка буде виконуватися.

Регістр прапорів *rflags/eflags/flags* містить інформацію про стан як самого МП, так і виконуваної програми. Найбільш важливі прапори:

• **cf** (*Carry flag*, номер біту 0) – прапор перенесення:

1 – під час арифметичної операції було перенесення із старшого біту (7-го, 15-го, 31-го, 63-го для 8, 16, 32, 64 розрядних операндів) результату;

0 – перенесення не було.

• **pf** (*Parity flag*, номер біту 2) – прапор паритету:

0 – 8-м молодших розрядів результату містить непарне число одиниць;

1 – 8-м молодших розрядів результату містить парне число одиниць.

• **af** (*Auxiliary carry flag,* номер біту 4), фіксація факту перенесення (позики) в (з) молодшої тетради при роботі з BCD-числами.

• **zf** (*Zero flag*, номер біту 6) – прапор нуля:

0 – результат останньої операції не нульовий.

1 – результат останньої операції нульовий;

• **sf** (*Sign flag*, номер біту 7) – прапор знаку:

0 – старший біт результату дорівнює 0;

1 – старший біт результату дорівнює 1;

• **tf** (*Trap flag,* номер біту 8) – дозволяє переведення інструкцій процесора в однокроковий (DEBUG) режим:

0 – звичайний режим;

1 – однокроковий режим.

• **if** (*Interrup flag*, номер біту 9) – визначає чи зовнішні переривання ігноруються чи обробляються:

0 – ігноруються;

1 – обробляються.

• **df** (*Direction flag*, номер біту 10) – прапор напрямок символьних рядків:

1 – напрямок “назад”, від старших адрес до молодших;

0 – напрямок “вперед”, від молодших адрес до старших.

• **of** (*Overflow flag*, номер біту 11) – прапор втрати значущого біту при арифметичних операціях:

1 - під час арифметичної операції було перенесення (позика) в (з) старшого (знакового) біта результату;

0 – не було перенесення/позики із старшого (знакового) біту.

• **iopl** (*Input/output Privilege Level*, номери бітів 12-13) – рівень привілеїв введення/виведення. Використовується у захищеному режимі процесора для контролю доступу до команд введення-виведення в залежності від привілейованості задачі.

• **nt** (*Nested Task*, номер біту 14) – прапор вкладеності задач. Використовується у захищеному режимі роботи процесора для фіксації того факту, що одна задача вкладена в іншу.

• **rf** (*Resume flag*, номер біту 16) – прапор відновлення. Використовується при обробленні переривань від регістрів налагодження.

• **vm** (*Virtual 8086 mode*, номер біту 17) – прапор віртуального 8086. Признак роботи процесора в режимі віртуального 8086: 1 – режим віртуального 8086, 0 – захищений режим.

• **ac** (*Alignment check*, номер біту 18) – прапор контролю вирівнювання. Призначений для дозволу контролю вирівнювання при зверненнях до пам'яті. Використовується сумісно з бітом `am` в системному регістрі `cr0`. Якщо потрібно контролювати вирівнювання даних і команд за адресами, кратними 2, 4, 8 то встановлення даних бітів приведе до того, що всі звернення за некратними адресами будуть збуджувати виняткову ситуацію.

• **vif** (*Virtual interrupt flag*, номер біту 19) – прапор віртуального переривання. При певних умовах є аналогом прапора `if`. Використовується сумісно з прапором `vip`.

• **vip** (*Virtual interrupt pending flag*, номер біта 20) – прапор віртуального відкладеного переривання. Встановлюється в 1 для індикації відкладеного переривання. Використовується при роботі у V-режимі сумісно з прапором `vif`.

• **id** (*Identification flag*, номер біту 21) – прапор ідентифікації. Показує факт підтримки процесором інструкції cpuid.

6.4. Регістри математичного співпроцесора

Вісім 80-бітових регістрів ST0,...ST7 для операцій з плаваючою крапкою.

6.5. MMX регістри

Розширення MMX включає в себе вісім 64-бітових регістрів загального використання MM0-MM7. Регістри MMX об'єднані з мантисами восьми регістрів математичного співпроцесора (FPU). Апаратно це можуть бути різні пристрої, але з точки зору програміста – це одні й ті ж регістри. Таким чином, не можна одночасно користуватися командами математичного співпроцесора і MMX.

6.6. XMM регістри

128-розрядні регістри XMM0,...XMM15 є частиною розширення SSE (де SSE є скороченням від Streaming SIMD Extension, а SIMD, у свою чергу, означає одну інструкцію з кількома даними). XMM регістри дозволяють одночасні операції над: 16 byte, 8 words, 4 double words, 2 quad words, 4 floats, 2 doubles.

6.7. YMM регістри

256-розрядні регісти YMM0,...YMM15 є частиною розширення AVX (Advanced Vector

Extensions). YMM регістри дозволяють одночасні операції над:

- 8-ма 32-бітовими числами з плаваючою крапкою звичайної точності;
- 4-ма 64-бітовими числами з плаваючою крапкою подвійної точності.

7. Службові регістри

7.1. Керуючі регістри

До керуючих відносяться вісім (32-бітові) регістри CR0, CR1, CR2, CR3, CR4, CR5, CR6, CR7.

Регістр CR0:

• 0-біт, дозвіл захисту. Переводить процесор у захищений режим;

• 1-біт, моніторинг співпроцесора (MP). Викликає виняток 7 для кожної команди `wait`;

• 2-біт, емуляція співпроцесора (EM). Викликає виняток 7 для кожної команди співпроцесора;

• 3-біт, перемикання задач (TS). Дозволяє визначити, чи відноситься даний контекст співпроцесора до поточної задачі чи ні. Викликає виняток 7 при виконанні наступної команди співпроцесора;

• 4-біт, індикатор підтримки інструкцій співпроцесора (ET);

• 5-біт, дозвіл стандартного механізму повідомлень про помилку співпроцесора (NE);

• 6-15 біти, не використовуються;

• 16-біт, дозвіл захисту від запису на рівні привілеїв супервізора (WB);

• 17-біт, не використовується;

• 18-біт, дозвіл контролю вирівнювання (AM);

• 19-28 біти, не використовуються;

• 29-біт, заборона наскрізного запису кешу і циклів анулювання (NW);

• 30-біт, заборона використання кешу (CD);

• 31-біт, включення механізму сторінкової переадресації.

Регістр CR1 зарезервований.

Регістр CR2 зберігає 32-бітову лінійну адресу, для якої була отримана остання відмова сторінки пам’яті.

Регістр CR3:

• 3-біт, кешування сторінок із наскрізним записом (PWT);

• 4-біт, заборона кешування сторінки (PCD);

• 11-31, 20 старших бітів фізичної адреси таблиць каталогу сторінок при умові, що 5-й біт регістра CR4 дорівнює 1.

Регістр CR4:

• 0-біт, дозвіл використання віртуального прапора переривань в режимі V8086 (VME);

• 1-біт, дозвіл використання віртуального прапора переривань в захищеному режимі (PVI);

• 2-біт, перетворення інструкції RDTSC в привілейовану (TSD);

• 3-біт, дозвіл точок зупинки при зверненні до портів введення-виведення (DE);

• 4-біт, включає режим адресації з 4 мегабітовими сторінками (PSE);

• 5-біт, включає 36-бітовий фізичний адресний простір (PAE);

• 6-біт, дозвіл винятку MC (MCE);

• 7-біт, дозвіл глобальної сторінки (PGE);
• 8-біт, дозвіл виконання команди RDPMC (PMC);
• 9-біт, дозволяє команди швидкого збереження/відновлення стану співпроцесора (FSR).

7.2. Системні адресні регістри

До системних адресних регістрів відносяться чотири (16-бітові) регістри таблиць GDTR, IDTR, LDTR і TR.

• GDTR – 6-байтовий регістр, в якому міститься лінійна адреса глобальної дескрипторної таблиці;

• IDTR – 6-байтовий регістр, який містить 32-бітову лінійну адресу таблиці дескрипторів обробників переривань;

• LDTR – 10-байтовий регістр, який містить 16-бітовий селектор (індекс) для GDT і 8-байтовий дескриптор;

• TR – 10-байтовий регістр, який містить 16-бітовий селектор (індекс) для GDT і 8-байтовий дескриптор з GDT, який описує TSS поточної задачі.

7.3. Регістри налагодження

• DR0-DR3 – зберігають 32-бітові лінійні адреси точок зупинки.
• DR6 (еквівалентно DR4) – відображає стан контрольних точок.
• DR7 (еквівалентно DR5) – керує встановленням контрольних точок.

Контрольні запитання.

1. Що таке фізична, логічна і лінійна адреса.
2. Які розміри областей пам’яті використовуються для адресації пам’яті.
3. Які режими роботи і моделі пам’яті підтримують процесори x86-32/64.
4. Які особливості роботи процесора в реальному режимі плоскої моделі пам’яті.
5. Які особливості роботи процесора в реальному режимі сегментованої моделі пам’яті.
6. Які особливості роботи процесора в захищеному режимі плоскої моделі пам’яті.
7. Які особливості роботи процесора в довгому режимі плоскої моделі пам’яті.
8. Які групи регістрів є в процесорі x86-64 і їх призначення.
9. Як використовуються регістри загального призначення.
10. Як використовуються сегментні регістри.
11. Як використовується регістр вказівник команд і регістр прапорів.

3. АСЕМБЛЕР NASM

Мета. Ознайомлення з засобами компіляції, налагодження і виконання асемблерних програм, синтаксисом асемблера Nasm, типами даних, директивами, способами адресації пам'яті.

Вступ. Кожний персональний комп'ютер має мікропроцесор, а кожна родина мікропроцесорів має свій набір інструкцій. Цей набір інструкцій називається "мовою машинних інструкцій". Мікропроцесор розуміє тільки мову машинних інструкцій, яка є послідовностями бітів із значеннями '0' і '1'. Але з такими послідовностями бітів дуже важко працювати при розробленні програм. Тому для кожної родини мікропроцесорів розроблена своя мова асемблера, яка подає машинні інструкції в мнемонічних кодах і в більш зрозумілій формі. Асемблер вважається самою низькорівневою мовою програмування. Асемблер переводить мнемокоди команд, зрозумілі людині, безпосередньо у машинні інструкції.

План.

1. Асемблювання, компонування і налагодження програм
2. Послідовність створення виконуваних файлів
3. Синтаксис асемблера
4. Константні типи
   4.1. Числа
   4.2. Символьні стрічки
   4.3. Символьні константи
   4.4. Стрічкові константи
   4.5. Константи з плаваючою крапкою
   4.6. Запаковані BCD константи
5. Псевдоінструкції і директиви
   5.1. Директиви
   5.2. Псевдоінструкції оголошення, ініціалізації і резервування пам'яті
6. Використання позначок
7. Способи адресації пам'яті
   7.1. Безпосередня адресація
   7.2. Регістрова адресація
   7.3. Пряма і непряма адресація
8. Обчислення адреси під час виконання програми
   8.1. Команда lea

1. Асемблювання, компонування і налагодження програм

Для розроблення програм на асемблері для процесорів x86-64 використовуються різні асемблери NASM, YASM, GAS (GNU AS є back-end в компіляторі gcc), as86, Microsoft Assembler (MASM), Borland Turbo Assembler (TASM), FASM.

Для навчальної мети широко використовується асемблер `nasm`, який підтримує Linux і Windows платформи, а також є вільнодоступним у дистрибутивах Linux. Після інсталяції ОС Linux можна вияснити де знаходиться `nasm` командою консолі:

```
$ whereis nasm
/usr/bin/nasm
```

Якщо `nasm` відсутній то потрібно його інсталювати із сайту розробників www.nasm.us. Обновити `nasm` до новішої версії можна командою `rpm`:

```
$ rpm -F nasm-2.13.03-0.fc24.x86_64.rpm
```

Якщо `nasm` інстальований, то можна визначити формат його об'єктного файлу:

```
$ cd /usr/bin/;file nasm;cd ~
nasm: ELF 64-bit LSB executable, x86-64
```

Результат виводу означає, що об'єктний файл має формат ELF (*executable and linkable format*), тому при асемблюванні потрібно використовувати ключ `-f elf`. `Nasm` використовує наступні ключі:

`-h` – довідка про використання `nasm`

`-f` – довідка про вихідні формати файлів

`-g` – генерувати налагоджувальну інформацію у вибраному форматі

`-e, -E` – виконати тільки передпроцесорну обробку

`-a` – тільки асемблювати (без передпроцесорної обробки)

`-o` – задання імені вихідного файлу;

`-f <xxx>`– задання вихідного формату

`-F` – вибір формату налагоджувальної інформації

`-l` – генерація файлу роздруку;

`-I` – добавити каталоги для пошуку файлів, які підключаються

`-M` – генерувати `makefile`

`-v` – версія програми

Приклади **асемблювання програм** з використанням асемблера `nasm` у консолі Linux:

- загальний формат асемблювання файлів

```
$nasm –f <format> <filename> [-o <output>]
```

- підтримувані формати вихідних файлів:

```
>nasm -hf
       * bin       flat-form binary files (e.g. DOS .COM, .SYS)
         ith       Intel hex
         srec      Motorola S-records
         aout      Linux a.out object files
         aoutb     NetBSD/FreeBSD a.out object files
         coff      COFF (i386) object files (e.g. DJGPP for DOS)
         elf32     ELF32 (i386) object files (e.g. Linux)
         elf64     ELF64 (x86_64) object files (e.g. Linux)
         elfx32    ELFX32 (x86_64) object files (e.g. Linux)
         as86      Linux as86 (bin86 version 0.3) object files
         obj       MS-DOS 16-bit/32-bit OMF object files
         win32     Microsoft Win32 (i386) object files
         win64     Microsoft Win64 (x86-64) object files
         rdf       Relocatable Dynamic Object File Format v2.0
         ieee      IEEE-695 (LADsoft variant) object file format
         macho32   NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (i386) object files
       macho64   NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (x86_64) object files
         dbg       Trace of all info passed to output stage
```

```
elf       ELF (short name for ELF32)
macho     MACHO (short name for MACHO32)
win       WIN (short name for WIN32)
```

Плоский бінарний формат bin не створює об'єктних файлів: у вихідний файл не генерується нічого, крім написаного коду. Такі "чисто бінарні" файли використовуються в MS-DOS: як файли .COM і драйвери пристроїв .SYS. Бінарний формат корисний також під час розробки операційних систем та завантажувачів.

Формат `bin` підтримує лише три стандартні секції з іменами `.text`, `.data` та `.bss`. У файлі, згенерованому NASM, спочатку йтиме вміст секції `.text`, а потім вміст секції `.data`, вирівняний на 4-байтову межу. Секція `.bss` не зберігається у вихідному файлі, але передбачається, що вона буде розташована відразу після секції `.data`, знову ж таки вирівняна на 4-байтову межу. Якщо явно не вказати директиву SECTION, написаний код буде направлений за замовчуванням у секцію `.text`.

Використання формату `bin` переводить NASM за замовчанням у 16-бітовий режим. Щоб використовувати його для написання 32-бітового коду (наприклад, ядра ОС), необхідно явно вказати директиву BITS 32. За замовчуванням `bin` не створює розширення файлу.:

Об'єктні файли формату `obj` за замовчуванням мають розширення `.obj`. NASM підтримує 16-, 32- і 64-бітові розширення цього формату. Формат `obj` не визначає спеціальних імен сегментів, їх можна назвати як завгодно. Типовими іменами сегментів формату `obj` є CODE, DATA та BSS.

Вихідний формат coff створює COFF-об'єктні файли, що обробляються компонувальником DJGPP. Цей формат передбачає за замовчанням розширення вихідних файлів `.o`.

Вихідний формат elf генерує об'єктні файли ELF32 (Executable and Linkable Format), використовувані Лінукс. Цей формат використовує за замовчанням розширення вихідних файлів `.o`.

Формат aout генерує об'єктні файли `a.out` у формі, що використовується застарілими системами Лінукс. Формат `a.out` передбачає розширення вихідних файлів за замовчанням `.o`. Цей формат дуже простий і підтримує лише три стандартні секції з іменами `.text`, `.data` та `.bss`.

Вихідний формат dbg у конфігурації NASM за замовчуванням відсутній. Формат `dbg` не створює об'єктних файлів, а створює текстовий файл, що містить повний список усіх транзакцій між ядром NASM та модулем вихідних форматів. Це звичайно потрібно при написанні власних вихідних драйверів. Цей формат дозволяє отримати картину різних запитів основної програми до вихідного драйвера і побачити, як вони виконуються.

- асемблювання програми в 32-розрядний об’єктний файл формату ELF в 64-розр. ОС

```
$nasm –f elf32 program.asm –o program.o
```

- асемблювання програми в 64-розрядний об’єктний файл формату ELF в 64-розр. ОС

```
$nasm –f elf64 program.asm –o program.o
```

- асемблювання програми в об’єктний файл формату ELF з вставленням налагоджувальної інформації для налагоджувача (опція `-g`), формат генерування налагоджувальної інформації (`-F dwarf`)

```
$nasm –f elf64 –g –F dwarf program.asm
```

- підтримувані формати налагоджувальної інформації:

```
$nasm -f ELF -y
```

```
valid debug formats for 'elf32' output format are ('*' denotes default):
    dwarf     ELF32 (i386) dwarf debug format for Linux/Unix
  * stabs     ELF32 (i386) stabs debug format for Linux/Unix
```

- асемблювання програми в ”сирий” бінарний файл (16-біт)

```
$nasm –f bin myfile.asm –o myfile.com
```

- асемблювання програми з отриманням файлу роздруку з hex-кодами (32-біт)

```
$nasm –f coff myfile.asm –l myfile.lst
```

Компонування програм для отримання виконуваних (бінарних) файлів:

- 32-розрядна ОС, 32-розрядний виконуваний файл

```
$ld program.o -o program
```

- 64-розрядна ОС, 32-розрядний виконуваний файл

```
$ld –m elf_i386 program.o -o program
```

- 64-розрядна ОС, 64-розрядний виконуваний файл

```
$ld -m elf_x86_64 program.o -o program
```

Взнати тип ОС, архітектуру і розрядність процесора можна за допомогою команди

```
$uname -a
linux linux-dell 4.1.13-5-default … UTC 2015 x86_64 GNU/Linux
```

Архітектури i386, i586, i686, x86 вказують на 32-бітові процесори, а x86_64, amd64 – на 64-бітові .

Запуск програми на виконання

```
$./program
```

Асемблювання, компонування і виконання програми

```
$nasm –f elf64 program.asm && ld program.o && ./a.out
```

Асемблювання, компонування і виконання програми, яка викликає Сі функції

```
$nasm –f elf64 program.asm && gcc program.o && ./a.out
```

Запуск програми на виконання у консольному налагоджувачі KDBG ttps://www.kdbg.org

```
$gdb program
```

Команди для асемблювання і компонування можна оформити як сценарії. Приклад сценаріїв для асемблювання (`debug.sh`) і компонування 64-розрядного бінарного файлу для роботи із графічним налагоджувачем KDbg (`load.sh`). Після асемблювання створюється об’єктний файл з розширенням *.o.

Сценарій асемблювання `debug.sh`:

```
#!/bin/bash
echo "Програма " $1
nasm -f elf64 -g -F dwarf $1
```

Запуск сценарію на асемблювання

```
$./debug.sh 1.asm
```

Сценарій компонування `load.sh`:

```
#!/bin/bash
ld -m elf_x86_64 $1 -o test
```

Запуск сценарію на компонування

```
$./load.sh 1.o
```

Сценарій асемблювання і компонування

Асемблювання і компонування програм зручно виконувати запуском одного Bash сценарію:

```
#-------------------------------------------------------------
# Сценарій асемблювання і компонування програм Nasm асемблера
# Copyright (C) Голота В.І., 2020
#-------------------------------------------------------------
#!/bin/bash
rm *.o *.map *.lst
echo "-----------------------------------------------------"
echo "Сценарій: $0      Дата-Час: `date +%d.%m.%y-%T`"
echo "-----------------------------------------------------"

declare -i code

if [ ! -f $1 ]
then
   echo "Файл $1 відсутній"
   exit 1
fi

s=$1
name=${s%.*m}
nasm -f elf64 -g -F dwarf  $1 -l $name".lst" -o $name".o"
code=$?
if [ $code -eq 0 ]
then
    echo "Асемблювання $1 успішне - $code"
else
    echo "Асемблювання $1 не успішне - $code"
    exit $code
fi

#ld -m elf_x86_64 -M -Map $name".map" $name".o" -o $name
gcc -o $name  $name".o" -no-pie
code=$?
if [ $code -eq  0 ]
then
   echo "Компонування $1 успішне - $code"
   rm $name".o"
else
   echo "Компонування $1 не успішне - $code"
   exit $code
fi
```

Сценарій можна записати під іменем `asm_ld.sh` у каталог де знаходяться асемблерні програми і зробити його виконуваним

```
$ chmod u+x asm_ld.sh
```

Асемблювання і запуск програм на виконання:

```
$ ./asm_ld.sh program.asm
```

Налагодження програм

Після компонування створюється бінарний файл, який можна завантажити у різні налагоджувачі GDB, DDD, KDBG, SASM (рис. 1).

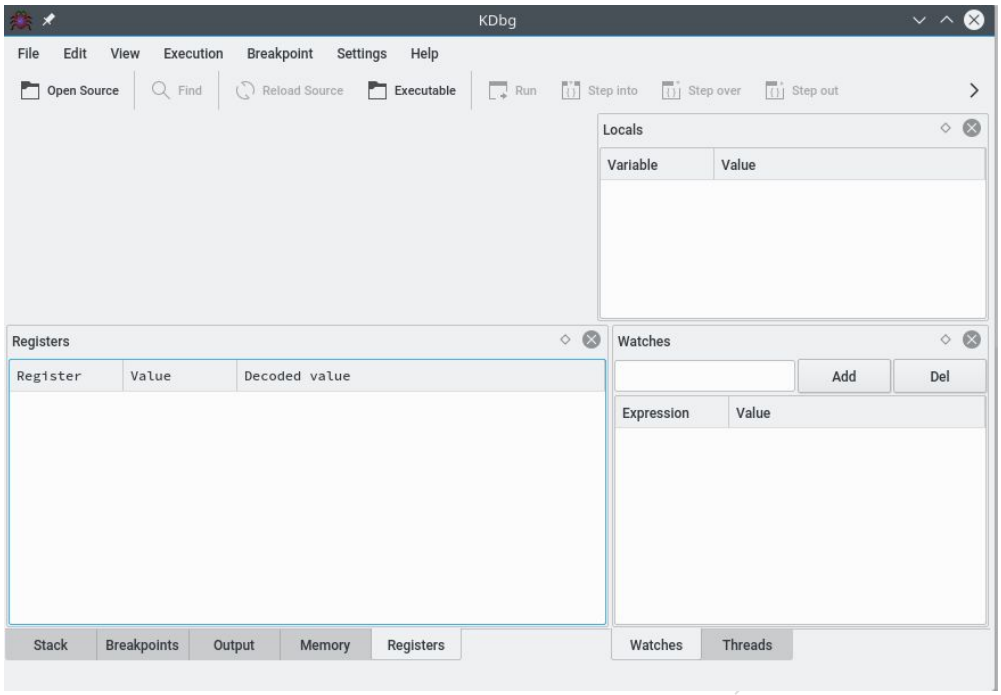


Рисунок 1 – Графічне середовище налагоджувача KDBG

Вибором меню **Open Source** завантажити асемблерну програму і задати в ній точки запинки (Breakpoint/Set/Clear). Вибором меню **Executable** завантажити бінарний файл і запустити його на виконання **Execution/Step into**.

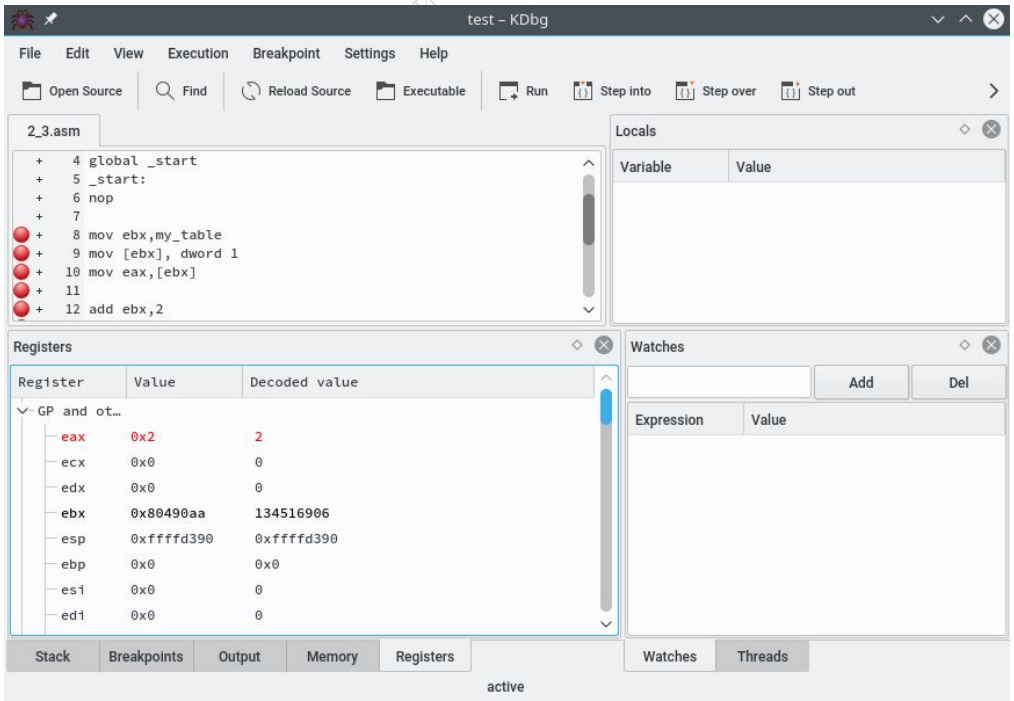


Рисунок 2 – Вікно налагодження програми

2. Послідовність створення виконуваних файлів

При створенні виконуваних файлів можуть використовуватися статичні і динамічні бібліотеки. Загальна послідовність створення виконуваних файлів з використанням бібліотек показана на рис. 3.

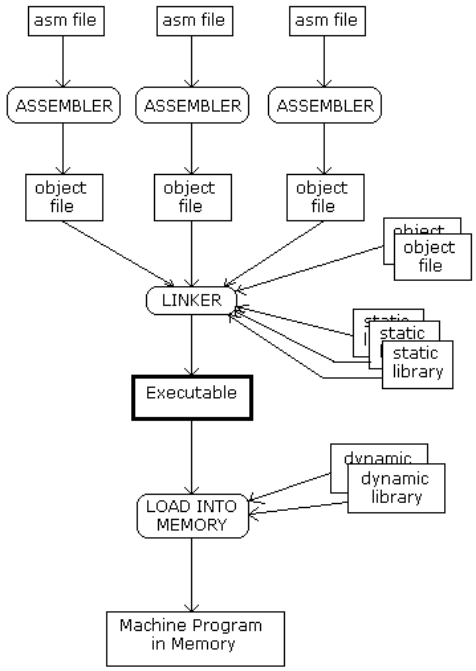


Рисунок 3 – Послідовність створення виконуваних файлів асемблера

При створенні виконуваного файлу можуть підключатися як статичні так і динамічні бібліотеки на відповідних кроках:

- початкові (сирцеві) файли асемблюються асемблером (`assembler`) в об'єктні файли;
- компонувач (`linker`) збирає об'єктні файли і файли із статичних бібліотек у виконуваний файл з віртуальними (переміщуваними) адресами;
- завантажувач (`loader`) завантажує виконуваний модуль у пам'ять, підключає динамічні бібліотеки і налаштовує фізичні адреси. Отриманий таким чином модуль готовий до виконання.

3. Синтаксис асемблера

Мова асемблера складається з речень, які записуються в один рядок. Речення можуть містити наступні синтаксичні конструкції:

• **Команди** (інструкції) є символьними аналогами (мнемоніками) машинних команд. В процесі асемблювання команди перетворюються у відповідні команди процесора.

• **Макрокоманди** – це оформлені певним чином послідовності коротких команд, які замінюються під час асемблювання іншими довшими командами.

• **Директиви** є вказівками асемблеру на виконання деяких дій. Директиви не мають відповідних машинних команд.

• **Коментарі** можуть містити любі символи. Коментарі починаються з символу ; та не обробляються асемблером.

Формат команди асемблера:

```
[позначка:] команда [ операнд1 [,операнд2]]  [; коментар]
```

Позначка – це ідентифікатор, з яким асоціюється адреса в пам’яті.
Операнди – це об’єкти над якими виконаються дії, які задає команда.

3. Константні типи даних

Асемблер розпізнає наступні константні типи: числові, символьні, стрічкові, числа з плаваючою крапкою.

4.1. Числові константи

Числові константи – це просто числа. NASM дозволяє використовувати числа в різних системах числення і різними способами: можна використовувати префікси і суфікси:

Числова константа	Префікс	Суфікс
шістнадцяткова	0x, 0h або $	h, x
десяткова	0d, 0t	d, t
вісімкова	0q, 0o	q, o
двійкова	0b, 0y	b, y

Примітка. Якщо після префіксу $ слідує буква, то перед нею потрібно вставити 0, наприклад $0a1. Це зв'язано з тим, що символ $ відміняє дію зарезервованих слів. Так $eax позначає символ eax, а не регістр eax. Аналогічно необхідно слідкувати і за першим символом при використанні суфікса h, наприклад, 0a51h, а не a51h).

Приклади задання одного і того ж числа 200:

```
mov ax,200   ; десяткове
mov ax,0200  ; десяткове
mov ax,0200d ; явне десяткове
mov ax,0d200 ; явне десяткове
mov ax,0c8h  ; шістнадцяткове
mov ax,$0c8  ; шістнадцяткове: перед буквою потрібний 0
mov ax,0xc8  ; шістнадцяткове
mov ax,0hc8  ; шістнадцяткове
mov ax,310q  ; вісімкове
mov ax,310o  ; вісімкове
mov ax,0o310 ; вісімкове
mov ax,0q310 ; вісімкове
mov ax,11001000b   ; двійкове
mov ax,1100_1000b  ; двійкове
mov ax,1100_1000y  ; двійкове
mov ax,0b1100_1000 ; двійкове
mov ax,0y1100_1000 ; двійкове
```

4.2. Символьні стрічки

Символьні стрічки містять до 8-ми (x86-64) символів взятих у одинарні (‘…’), подвійні (“…”) або ліво нахилені лапки (`…`).

У стрічки, які знаходяться у ліво нахилених лапках, можна поміщати esc-символ (екрануючий символ) \ (як у Сі):

```
\’ одинарні лапки (’)
\" подвійні лапки (")
\‘ одинарні ліво нахилені лапки (‘)
\\ зворотній slash (\)
\? знак запитання (?)
\a BEL (ASCII 7)
\b BS  (ASCII 8)
\t TAB (ASCII 9)
\n LF  (ASCII 10)
\v VT  (ASCII 11)
\f FF  (ASCII 12)
\r CR  (ASCII 13)
\e ESC (ASCII 27)
\377 до 3-х вісімкових цифр – літерний byte
\xFF до 2-х шістнадцяткових цифр – літерний byte
\u1234   4-ри шістнадцяткові цифри – символи Unicode
\U12345678 8-м шістнадцяткових цифр – символи Unicode
```

Unicode символи задані з \U, \u конвертуються у UTF-8. Наприклад наступні рядки є еквівалентними (символ посмішки).

```
db ‘\u263a‘                    ; UTF-8
db ‘\xe2\x98\xba‘   ; UTF-8
db 0E2h, 098h, 0BAh ; UTF-8
```

4.3. Символьні константи (character constant)

Символьна константа може містити від одного до 8 символів, взятих в одинарні або подвійні лапки (тип лапок для NASM неважливий), трактується як число integer і використовується як операнд. Символьна константа довжиною більшою як один байт розміщується у регістрі і пам’яті у зворотному порядку байтів (“little-endian”):

```
mov eax,’abcd’  ; 0x61626364 => 0x64636261
```

Але якщо цю константу записати у пам’ять і знову прочитати то отримується ‘abcd’, а не ‘dcba’.

4.4. Стрічкові константи (string constant)

Стрічкові константи використовуються тільки у директивах виділення і ініціалізації пам’яті db і як імена файлів в INCBIN. Вони обробляються як зчеплені між собою символьні константи.

```
db ’hello’ ; стрічкова константа
db ’h’,’e’,’l’,’l’,’o’ ; еквівалентна символьна константа
dd ’ninechars’ ; стрічкова константа double word
dd ’nine’,’char’,’s’ ; стрічкова константа з трьох double words
db ’ninechars’,0,0,0 ; яка реально в пам’яті розміщується так
```

4.5. Константи з плаваючою крапкою

Константи з плаваючою крапкою сприймаються тільки як аргументи псевдоінструкцій DB, DW, DD, DQ, DT і DO або як директиви спеціальних інструкцій __float8__, __float16__,

__float32__, __float64__, __float80m__, __float80e__, __float128l__, and __float128h__.:

```
db -0.2 ; "1/4 точність"
dw -0.5 ; IEEE 754r/SSE5 1/2 точність
dd 1.2 ; 1-на точність
dd 1.222_222_222 ; '_' дозволено розбивати на групи
dd 0x1p+2 ; 1.0x2^2 = 4.0
dq 0x1p+32 ; 1.0x2^32 = 4 294 967 296.0
dq 1.e10 ; 10 000 000 000.0
dq 1.e+10 ; синонім до 1.e10
dq 1.e-10 ; 0.000 000 000 1
dt 3.141592653589793238462 ; pi
do 1.e+4000 ; IEEE 754r чотирна точність
mov rax,__float64__(3.141592653589793238462)
```

4.6. Запаковані BCD константи

Запаковані BCD константи можуть використовуватися як 80-бітові числа з плаваючою крапкою. Вони вказуються префіксом `0p` або суфіксом `p`, і можуть містити до 18 десяткових цифр:

```
dt  12_345_678_901_245_678p
dt -12_345_678_901_245_678p
dt 33p
dt +0p33
```

5. Псевдоінструкції і директиви

Псевдоінструкції і директиви не виконуються процесором. Вони самі виконують якусь дію, яка не транслюється в машинний код або інформує процесор. Псевдоінструкції і директиви використовуються для:

- визначення констант;
- визначення і резервування пам'яті для зберігання даних;
- розбивки пам'яті на сегменти;
- включення початкових файлів за умовою;
- включення інших файлів.

5.1. Директиви

До директив відносяться `BITS`, `DEFAULT`, `SECTION` (або `SEGMENT`), `ABSOLUTE`, `EXTERN`, `GLOBAL`, `COMMON`, `CPU`, `FLOAT`.

`BITS xx`, де `xx`=16, 32, 64 задає розрядність коду, який генерує асемблер.

`DEFAULTS` – змінює значення параметрів асемблера за замовчування.

`SECTION` (або `SEGMENT`) – задає тип сегментів у пам'яті.

`ABSOLUTE` – є альтернативною директивою до `SECTION`, створюючи сегмент, який починається з абсолютної адреси:

```
absolute 0x1A
kbuf_chr resw 1
```

```
kbuf_free resw 1
kbuf resw 16
```

EXTERN – імпортує символи з інших модулів:

```
extern _printf
extern _sscanf,_fscanf
```

GLOBAL – експортує символи в інші модулі:

```
global _main
_main:
; деякий код
```

COMMON – оголошує загальну область пам'яті для декількох модулів:

```
common intvar 4
```

CPU xx, де xx=8086, 186, 286, 386, 486, 586, 686, P2, P3, P4, X64, IA64 – обмежує асемблер інструкціями вказаного процесора.

FLOAT – задає оброблення констант з плаваючою крапкою.

Адреси сегментів пам'яті різного призначення зберігаються у сегментних регістрах. Типи сегментів:

.text – сегмент коду

.data – сегмент ініціалізованих даних

.bss – сегмент неініціалізованих даних (в ньому розміщуються буфери)

.stack – сегмент стеку

Директиви задання сегменту

```
segment <.ім'я сегменту>
section <.ім'я сегменту>
```


5.2. Псевдоінструкції

До псевдоінструкцій відносяться:

- оголошення і ініціалізації пам'яті в секції .data:

DB (8-біт), DW (16-біт), DD (32-біти), DQ (64-біти), DT (80-бітів), DDQ, DO 128-біт), DY (YMM регістри), DZ (ZMM регістри);

- резервування пам'яті в секції неініціалізованих даних .bss:

RESB (8-біт), RESW (16-біт), RESD (32-біти), RESQ (64-біти), REST (80-бітів), RESDDQ, RESO (128-біт), RESY (YMM регістри), RESZ (ZMM регістри);

- INCBIN підключення зовнішнього двійкового файлу (графічного або акустичного);

- EQU визначення символу для заданого константного значення;

- префікс TIMES повторює інструкцію, яка асемблюється, задане число разів.

Псевдоінструкції оголошення і ініціалізації не просто резервують пам'ять, а вказують, які значення в цій пам'яті повинні бути до моменту запуску програми. Псевдоінструкції **db**, **dw**, **dd**, **dq**, **dt** виділяють і ініціалізують області пам'яті розміром байт, слово, подвійне слово, чотири слова, десять байтів. Прийнято позначати виділені області пам'яті позначками (змінними), які будуть асоційовані з адресою першого байта.

```
section .data
L1 db 0x55    ; byte позначений L1 і ініціалізований значенням 0x55
L2 db 0, 1, 2, 3 ; визначено 4 байти із значеннями 0, 1, 2, 3
L3 db 110101b ; byte ініціалізований бінарним значенням 110101 (53₁₀)
```

```
L4 db 12h     ; byte ініціалізований шістнадцятковим значенням 12 (18₁₀)
L5 db 17o     ; byte ініціалізований вісімковим значенням 17 (15₁₀)
L6 dw 0x1234  ; word 0x34 0x12
L7 dw 'a'     ; word 0x61 0x00
L8 dw 'ab'    ; word 0x61 0x62 (символьна константа)
L9 dw 'abc'   ; word 0x61 0x62 0x63 0x00 (символьна константа)
L1 dd 0x12345678           ; 0x78 0x56 0x34 0x12
L2 dq 0x1122334455667788  ;0x88 0x77 0x66 0x55 0x44 0x33 0x22 0x11
L3 ddq 0x112233445566778899aabbccddeeff00
       ; 0x00 0xff 0xee 0xdd 0xcc 0xbb 0xaa 0x99
        ; 0x88 0x77 0x66 0x55 0x44 0x33 0x22 0x11
L4 do 0x112233445566778899aabbccddeeff00 ;те ж саме як попереднє
L5 dd 0x12345678  ; 0x78 0x56 0x34 0x12
L6 dd 1.234567e20 ; floating-point константа
L7 dd 1A92h   ; double word ініціалізований шістнадцятковим значенням 1A92
L8 dq 0x123456789abcdef0 ; 8-байтова константа
L9 dd 1.234567e20 ; константа з плаваючою крапкою
L9 dq 1.234567e20 ; -"- подвійної точності
L1 dt 1.234567e20 ; -"- розширеної точності
L2 db ? ; байт з неініціалізованим значенням
```

Директиви `DT, DO, DY, DZ` – не сприймають цілочислові константи.

Для задання символу, його потрібно взяти в одинарні або подвійні лапки. Аналогічно можна задати символьний рядок. Всередині подвійних лапок, одинарні лапки розглядаються як звичайний символ. Те саме можна сказати і про подвійні лапки всередині одинарних.

```
section    .data
L1 db 'A' ; byte ініціалізований ASCII кодом для A (65)
L2 db "A" ; byte ініціалізований ASCII кодом для A (65)
L3 db 'Hello word'
L4 db "Hello word"
L5 db 'So I say: "Don', "'", 't panic"'
L6 db "w", "o", "r", 'd', 0 ; визначено символьну стрічку C = "word"
L7 db 'word', 0 ; так само як L6

hexstr  db "01 02 03 04 05 06 0A",10  ; 10 -> '\n'
hexlen  equ $-hexstr       ; $ - поточна адреса
digits  db "0123456789"
ClearTern db 27  ;  <ESC>
```

Псевдоінструкція `dd` може визначати як цілі числа, так і константи звичайної точності з плаваючою крапкою. Псевдоінструкція `dq` визначає тільки константи подвійної точності з плаваючою крапкою.

Псевдоінструкції резервування неініціалізованої пам'яті повідомляють асемблеру, що потрібно зарезервувати задану кількість комірок пам'яті. Псевдоінструкції **`resb`**, **`resw`**, **`resd`**, **`resq`**, **`rest`**, **`reddq`**, **`reso`** резервують неініціалізовані комірки пам'яті розміром байт, слово, подвійне слово, чотирне слово, десять байт, шістнадцять байт в секції `.bss`. Після псевдоінструкцій вказується число, яке задає кількість комірок пам'яті. Перед псевдоінструкцією ставиться позначка.

```
buffer    resb 64    ; резервувати 64 bytes
wordvar   resw 1     ; резервувати word
```

```
realarray resq 10   ; масив 10 значень чотирної точності
ymmval    resy 1    ; один YMM регістр
ymmval    resz 32   ; 32 ZMM регістри
```

З версії NASM 2.15 підтримується синтаксис використання `?` і `dup` в директивах `d*`. Так вищевказаний приклад так може бути переписаний:

```
buffer:   db  64 dup (?)  ; резервувати 64 bytes
wordvar:  dw  ?           ; резервувати 4 байти
realarray dq  10 dup (?)  ; масив 10 значень чотирної точності
ymmval:   dy  ?           ; один YMM регістр
zmmvals:  dz  32 dup (?)  ; 32 ZMM регістри
```

5.2.1. Псевдоінструкція INCBIN

Псевдоінструкція **`INCBIN`** включає бінарний (графічний або звуковий) файл у вихідний файл:

```
incbin "file.dat"      ; включити увесь файл
incbin "file.dat",1024 ; пропустити перші 1024 байти
incbin  "file.dat",1024,512  ;пропустити перші 1024 і включити наступні 512 байт
```

5.2.2. Псевдоінструкція EQU.

Псевдоінструкція **`equ`** *назначає символ для заданої константи,* яка не може надалі змінюватися.

```
message db 'hello, world',0
msglen equ $-message   ; msglen є константою із значенням 12
```

5.2.3. Префікс TIMES

Префікс **`times`** повторює інструкції або дані задане число разів:

```
zerobuf times 64 db 0; виділення 64 байтів ініціалізованих нулями
```

Аргумент `TIMES` може бути не тільки числовою константою, але і числовим виразом:

```
buffer db 'hello, world'
times 64-$+buffer db ' '   ; виділення буфера довжиною 64 байти
```

Аргументом TIMES може бути і звичайна інструкція:

```
times 100 movsb
```

6. Використання позначок

Позначки використовуються в секції коду для галуження програми. Позначка **`label:`** є змінною (покажчиком, вказівником, посиланням) і **задає адресу пам'яті**. Позначка **`.label:`** є локальною і асоціюється з попередньою нелокальною позначкою:

```
label1: ; some code
.loop
```

```
        ; some more code
        jne     .loop   ; label1.loop
        ret
label2: ; some code
.loop
        ; some more code
        jne .loop ; label2.loop
        ret
```

У наведеному вище фрагменті коду кожна інструкція `jne` переходить до рядка безпосередньо перед нею, оскільки два визначення `.loop` зберігаються окремо завдяки тому, що кожне з них пов'язане з попередньою нелокальною позначкою.

Деякі асемблери розрізняють позначки з двокрапками і без. NASM такі позначки не розрізняє. Звичайно програмісти ставлять *двокрапку після позначок, якими позначені машинні команди,* на які можна передати керування, але не ставлять *двокрапку після позначок, які оголошують або резервують пам'ять (змінні).*

Якщо позначка, яка асоціюється з оголошенням пам'яті, взята в квадратні дужки **`[label]`**, то вона задає **значення за адресою** `label`. Приклад використання позначок в 32-бітовому режимі:

```
mov al, [L1]  ; копіювати в регістр al дані (байт) за адресою L1
mov eax, L1   ; копіювати в регістр EAX адресу байта з позначкою L1
mov [L1], ah  ; копіювати регістр AH у комірку з адресою L1
```

При записуванні безпосереднього значення у пам'ять без задання його розміру асемблер видає помилку:

```
mov [L6], 5 ; записати число 5 у пам'ять за адресою L6 – помилка. Не вказано як записати число 5 – як byte, word, чи double word
```

Для вказування розміру записуваного безпосереднього значення потрібно вказати один із специфікаторів **`byte`**, **`word`**, **`dword`**, **`qword`**, **`tword`**.

```
mov dword [L6], 5 ; записати 5, як подвійне слово, за адресою L6
mov [L6], dword 5 ; або так
```

Спеціальна псевдо позначка **`$`**, містить поточну адресу:

```
message db 'hello, world'
msglen equ $-message        ; визначення довжини стрічки
```

Спеціальна псевдо позначка **`$$`**, містить адресу початку сегмента:

`$$-$` – віддаль від початку сегменту до поточної адреси

7. Вирази

Операнди команд можуть містити вирази з наступними інструкціями:

| - побітове OR;

^ - побітове XOR;

& - побітове AND;

m<<*n*, *m*>>*n* – зсув *m* вліво/вправо на *n* розрядів;

+, -, *, /, //, %, %% - додавання, віднімання, множення, беззнакове ділення, знакове ділення, беззнакове ділення по модулю, знакове ділення по модулю.

Унарні оператори:

+, -, ~, ! – зміна знаку на +, зміна знаку на -, інверсія числа (доповнення до 1) ~, логічне заперечення !.

8. Види адресації в командах асемблера

Команда може не мати операндів або мати один або два операнди. Як операнд можна задати безпосереднє значення, ім'я регістру або посилання на комірку пам'яті. Найбільш зручний і швидкий варіант розміщення операндів у регістрах, найбільш поширений – в системній пам'яті. Дані можуть також знаходиться у пристроях введення/виведення. Місце знаходження операндів задається кодом команди. Для кожної команди методи адресації визначають звідки взяти вхідний і куди помістити вихідний операнд. Операнди можуть бути 8-, 16-, 32- і 64-розрядні. Майже кожна команда вимагає щоб операнди були однакового розміру.

8.1 Безпосередня адресація

При *безпосередній адресації* команда має операнд з безпосереднім значенням або виразом:

```
value db 5      ; виділення і ініціалізація байту
mov rax, 45h    ; скопіювати безпосереднє значення 45h у регістр rax
mov rax, value+2 ; скопіювати адресу value+2 у регістр rax
```

Безпосередня адресація дозволяє підвищити швидкість виконання операції, так як у цьому випадку вся команда, включно з операндом, зчитується з пам'яті одночасно і на час виконання команди зберігається в процесорі у спеціальному регістрі команд (РК). При використанні безпосередньої адресації появляється залежність кодів команд від даних, що потребує зміни програми при кожній зміні безпосереднього операнда.

7.2. Регістрова адресація

При *регістровій адресації* операнд знаходиться у регістрі (регістровий операнд):

```
mov rax, rcx      ; скопіювати значення з регістра ecx в регістр eax
mov rdx, tax_rate ; скопіювати  адресу комірки пам'яті у регістр
mov count, rcx    ; скопіювати значення з регістра у пам'ять
```

7.3. Пряма і непряма адресація пам'яті

При *прямій адресації* операнд містить адресу (зміщення) комірки пам'яті. Таке зміщення називається *ефективною* адресою, так як вона відраховується від початку сегменту, адреса якого знаходиться у регістрі `ds`. Так як для обчислення ефективної адреси необхідна і адреса сегменту, тому така адресація є повільною.

```
section .data
addr1 dq 0
section .text
mov rax,addr1   ; скопіювати в регістр rax адресу addr1
add rax,8       ; збільшити адресу на 8
```

При *прямій адресації із зміщенням* використовуються арифметичні операції для модифікації адреси. Приклад копіювання даних з таблиці у регістри:

```
byte_table db 14, 15, 22, 45       ; таблиця байтів
word_table dw 134, 345, 564, 123   ; таблиця слів
mov cl, byte_table+2   ; скопіювати адресу 3-го елементу з byte_table
mov cx, word_table+3   ; скопіювати адресу 4-го елементу з word_table
```

Непряма адресація пам'яті використовує базові (RBX, RBP) і індексні регістри (RDI, RSI) для адресації комірок пам'яті. Для доступу до значень комірок регістри, вирази або позначки записуються в квадратних дужка, наприклад [RBX]**.**

```
segment .data
num dq 1,2,3,4,5
segment .text
global _start
_start:
mov rax,[num]     ; завантажити в регістр rax значення за адресою num
mov rbx,[num+2]   ; nasm - завантажити в регістр rbx значення,
                  ; яке знаходиться за адресою num+2
mov rcx,[rax]     ; завантажити в регістр rcx значення, яке знаходиться
                  ; за адресою rax
```

Звичайно непряма адресація використовується для доступу до масивів. *Стартова адреса масиву зберігається у базовому регістрі* `ebx`.

```
my_table times 10 dq 0 ; виділення 10 word з ініціалізацією 0
mov rbx, my_table      ; ефективна адреса my_table в rbx
mov [rbx], qword 1     ; my_table[0] = 1
add rbx, 8             ; збільшення адреси rbx = адреса rbx+8
mov [rbx],qword 3      ; my_table[1] = 3
```

Використання непрямої адресації операнда в оперативній пам'яті, при зберіганні його адреси в регістровій пам'яті, суттєво скорочує довжину поля адреси, одночасно зберігаючи можливість використання для фізичної адреси усієї розрядності регістра. Недолік цього способу – необхідність додаткового часу для читання адреси операнда. Разом з тим він суттєво підвищує гнучкість програмування. Змінюючи вміст комірки пам'яті або регістра, через які здійснюється адресація, можна, не міняючи команди в програмі, обробляти операнди, за різним адресами.

8. Обчислення адреси під час виконання програми

Існують і інші більш складні способи обчислення виконавчої адреси під час виконання, наприклад відносна, адресація по базі із зміщенням, адресація по базі з індексуванням і адресація з масштабуванням.

Відносна адресація використовується тоді, коли пам'ять логічно розбивається на сегменти. В цьому випадку адреса комірки пам'яті складається з двох частин: адреси початку сегмента (базова адреса), яка зберігається в регістрі, і зміщення, яке визначає положення комірки відносно початку сегмента. Адреса комірки пам'яті визначається як сума адреси сегменту і зміщення. Головний недолік відносної адресації – великий час обчислення виконавчої (фізичної) адреси операнда. Але суттєвою перевагою цього способу адресації є можливість створення "переміщуваних" програм, які можна розмістити в різних частинах пам'яті без зміни команд програми.

При адресації *по базі із зміщенням*, *індексуванням і масштабуванням* виконавча адреса обчислюється під час виконання програми.

Повна (виконавча, фізична) адреса у пам'яті обчислюється за виразом:

```
[  SELECTOR: BASE + INDEX*SCALE + OFFSET ]
SELECTOR = { CS, DS, ES, SS, FS, GS }
BASE =  { RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP}
INDEX = { RAX, RBX, RCX, RDX, RSI, RDI, RBP }
SCALE = { 1, 2, 4, 8 }
OFFSET = CONSTANT
```

де `SELECTOR` – один з шести сегментних регістрів `cs, ds, es, ss, fs, gs`.

`BASE` – один з регістрів загального призначення `rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, rbp, rsp`.

`INDEX` – один з регістрів загального призначення, за винятком `rsp`.

`SCALE` – масштабний множник, 1, 2, 4, 8.

`OFFSET` – любе 32-бітове число.

Щоб зрозуміти, для чого потрібна така складна адресація, достатньо розглянути масив `mas` з 2 рядків, кожний з яких містить 4 елементів розміром 8 байтів. Для доступу до елементу і-го рядка, потрібно отримати адресу початку масиву, а потім обчислити зміщення до бажаного елементу:

```
; 1.asm
segment .data
mas dq 1,2,3,4
    dq 5,6,7,8
segment .text
global main
main:
mov rbx,mas  ; адреса початку масиву
mov rsi,5    ; 6-й елемент
mov rcx,[rbx+rsi*8]

mov rax,60  ; повернення коду завершення в ОС
mov rdi,rcx
syscall

./1
echo $?
6
```

Nasm підтримує новий синтаксис розбиття виконавчої адреси бази і індексу:

```
mov rax,[rbx+16,rcx*8]   ; rbx=base, 16=disp, rcx=index, 8=scale,
```

8.1. Команда lea

Для отримання доступу до комірки пам'яті змінної `var` використовуються квадратні дужки [`var`]. Якщо квадратні дужки не вказати, то буде отримано не значення змінної, а її адреса:

```
mov rax, qword [var] ; значення комірки var в rax
mov rbx, var ; адреса комірки var в rbx
```

Адресу змінної можна визначити завантажуючи виконавчу (ефективну) адресу командою `lea`:

```
lea <reg>, mem  ; reg := регістр ЗП 16,32,64 біт
```

```
segment .data
var1 dq 1
var2 dq 0x00000072
 segment .text
lea rcx, [var1]
lea rsi, [var2]
```

Щоб зрозуміти роботу команди `lea`, для обчислення адреси `rdi` і поміщення у `rax`:

```
mov rax, rdi        ; інструкція mov
inc rax;
```

```
lea  rax, [rdi+1]   ; інструкція lea
```

Для комірки з адресою `[bp+si+4]`:

```
mov eax, bp         ; інструкція mov
add eax, si
add eax, 4;
```

```
lea  rax, [bp+si+4] ; інструкція lea
```

Контрольні запитання.

1. Які основні ключі використовуються при асемблюванні програм.
2. Як асемблювати і компонувати `nasm` програму.
3. Синтаксис асемблера.
4. Оголошення констант в асемблері (стрічки, цілі числа, числа з плаваючою крапкою, BCD числа).
5. Директиви асемблера `BITS, DEFAULT, SECTION або SEGMENT, ABSOLUTE, EXTERN, GLOBAL, COMMON, CPU, FLOAT`.
6. Псевдоінструкції асемблера `DB, DW, DD, DQ, DT, DDQ, DO; RESB, RESW, RESD, RESQ, REST, RESDDQ, RESO, INCBIN, EQU, TIMES`.
7. Позначки в асемблерній програмі.
8. Види адресації в командах асемблера.
9. Обчислення виконавчої адреси під час виконання програми.
10. Обчислення виконавчої адреси без звернення до пам’яті.

4. МАКРОПРОЦЕСОР І МАКРОДИРЕКТИВИ

Мета. Вивчення макропроцесора і макродиректив асемблера Nasm

Вступ. Так як асемблери мають недостатні можливості (порівняно з мовами високого рівня), то їх доповнюють потужними макропроцесорами. Макропроцесор перетворює текст програми як за допомогою макровикликів, так і безпосередніх вказівок макродиректив.

План.

1. Основні поняття
2. Однорядкові макровизначення
   2.1. Директиви `%define, %undef, %xdefine`
   2.2. Макрос контекстного розширення `%[…]`
   2.3. Макрос зчеплення виразів `%+`
   2.4. Макроси самопосилання `%?, %??`
   2.5. Директива `%assign (%iassign)`
   2.6. Макрос визначення стрічок `%defstr`
   2.7. Директиви зчеплення стрічок `%strcat, %strlen, %substr`
3. Багаторядкові макроси `%macro`
   3.1. Перевантаження макросів
   3.2. Локальні позначки у макросах
   3.3. Скупі макроси
   3.4. Змінне число і діапазон параметрів
   3.5. Лічильник параметрів макросу `%0`
   3.6. Прокручування параметрів макросу
   3.7. Зчеплення макропараметрів
   3.8. Коди умов як макропараметри
   3.9. Відміна макросів
4. Директиви асемблювання з умовами
   4.1. Директиви з логічними умовами
      4.1.1 Директиви `%if, %elif, %else, %endif`
      4.1.2 Директиви `%ifmacro, %else, %endif`
      4.1.3 Директива `%ifcntx`
      4.1.4 Директиви `%if expr, %ifn expr`
      4.1.5 Директиви `%ifid, %ifnum, %ifstr`
      4.1.6 Директиви `%ifempy, iftoken`
5. Передпроцесорний цикл `%rep, %endrep, %exitrep`
6. Директиви підключення зовнішніх ресурсів `%include, %use`
7. Директива задання шляху пошуку `%pathsearch`
8. Передпроцесорний стек контекстів
9. Стандартні макроси
   9.1. Створення екземплярів структури у пам’яті даних, `istruc, at, iend`
   9.2. Макроси вирівнювання коду `align, alignb`
10. Приклади простих макросів

11. Директиви асемблера
12. Дизасемблер `NDIASM`

1. Основні поняття

Під **макропроцесором** розуміють програмний засіб, який отримує на вхід деякий текст і, використовуючи вказівки, що задані в тексті, частково перетворює його, формуючи на виході розширений текст, але вже без вказівок на перетворення. Результатом роботи макропроцесора є текст на асемблері, який вже асемблюється згідно з правилами мови, рис. 1.

Рисунок 1 – Схема роботи макропроцесора

Так як асемблери мають недостатні можливості введення і виведення даних (порівняно з мовами високого рівня), то їх доповнюють потужними макропроцесорами.

Макрокоманда, макровизначення, макрос – послідовність інструкцій з іменем, яка може використовуватися в любому місці програми.

Перед використанням макросу його потрібно визначити, тобто вказати макропроцесору, що деякий ідентифікатор є іменем макросу.

Коли макропроцесор зустрічає у програмі ім'я макросу і параметри (так званий **макровиклик**), він *замінює* ім'я макросу (і можливо параметри) фрагментом тексту, отриманим у відповідності із визначенням макросу. Така заміна називається **макророзширенням**.

Приклад макросу для виклику підпрограми з одним аргументом

```
%macro pcal1 2  ; кількість параметрів макросу
push %2
call %1
add rsp 8       ; звільнення стеку від аргументу
%endmacro
```

Макрос з іменем `pcal1` має два параметри – ім'я підпрограми і її аргумент. В тілі макросу `%1` і `%2` будуть замінені відповідно на перший і другий параметри у виклику макросу. Якщо в тексті асемблерної програми зустрінеться рядок `pcal1 myproc, rax` то макропроцесор сприйме його як макровиклик і виконає макрос згідно макровизначення. В результаті макрос буде замінено на наступний фрагмент

```
push rax
call myproc
add rsp 8
```

Звичайно макрос поміщають в окремий файл, наприклад `stud.inc`. Для того щоб можна було викликати макроси з цього файлу, його необхідно підключити макродирективою `%include`. Макродиректива вказує макропроцесору на заміну її самою вмістом файлу вказаним як параметр:

```
%include "mymacro.inc"
```

2. Однорядкові макроси

2.1. Директиви `%define, %xdefine, %undef`

Для описання однорядкового макросу використовується передпроцесорна директива `%define`.

Директива **`%define`** розширює (розгортає) перший параметр у другий *під час її виклику*:

```
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
…
    mov byte [param(2,ebx)], ctrl 'D'
```

що буде розширено до

```
    mov byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

Коли однорядковий макрос містить символи, які викликають інший макрос, то розширення першого параметра відбувається під час *виклику*, а не визначення:

```
%define a(x) 1+b(x)
%define b(x) 2*x
mov ax,a(8)
```

макрос буде розширено під час виклику до 1+2*8, незважаючи на те, що при описанні макросу `a` макрос `b` ще не визначений.

Макроси визначені директивою `%define` є регістро залежними, а директивою `%idefine` – регістро незалежними

```
%define size 100     ;  стосується тільки size
%idefine Size 100    ;  стосується size, SIZE, Size
```

Для запобігання рекурсивних розширень у макросах використовується механізм розширення тільки для першого входження макросу:

```
%define a(x) 1+a(x)
mov ax,a(3)      ; 1+a(3) без подальшого розширення
```

Однорядкові макроси можна перевантажувати:

```
%define fun(x) 1+x
%define fun(x,y) 1+x*y
```

Макропроцесор зможе обробити обидва макроси, підраховуючи число параметрів:

`fun(3)` перетвориться в `1+3`, а `fun(x,y)` – в `1+3*y`

Макрос без параметрів не можна перевизначити у макрос з параметрами і навпаки. Макрос без параметрів можна перевизначити в одному і тому файлі :

```
%define fun bar
…
%define fun baz
```

Для того, щоб макроси розширювалися під час *визначення* (*негайно*), а не під час виклику, використовується директива **`%xdefine`** (або її регістро незалежна пара **`%ixdefine`**).

```
;Розширення під час виклику
```

```
%define isTrue 1
%define isFalse isTrue
%define isTrue 0  ----
…                    |
  val1: db isFalse <--
…
%define isTrue 1  ----
…                    |
  val2: db isFalse <--
```

У цьому випадку позначка `val1` має значення 0, а `val2 - 1`, це тому що однорядковий макрос розширюється тільки при його виклику. Так як `isFalse` розширюється до `isTrue`, то розширення матиме значення `isTrue`. У позначці `val1` розширення матиме значення 0, а у позначці `val2 - 1`.

Для того, щоб `isFalse` розширювався до значення присвоєному вбудованому макросу `isTrue` у момент визначення `isFalse`, потрібно використати директиву `%xdefine`.

```
; розширення під час визначення
%xdefine isTrue 1        ---
%xdefine isFalse isTrue  <-|
%xdefine isTrue 0
...
  val1: db isFalse
...
%xdefine isTrue 1
...
  val2: db isFalse
```

Тепер, при кожному виклику `isFalse` воно буде розширюватися до 1 і позначки `val1 i` і `val2` матимуть значення 1.

Директива **`%undef`** відміняє попереднє визначення однорядкового макросу, наприклад:

```
%define foo bar
%undef foo

    mov eax, foo
```

виклик макросу буде розширено до

```
    mov eax.
```

2.2. Макрос контекстного розширення %[…]

Макрос контекстного розширення %[…] розширюється в залежності від імен інших макросів. Наприклад набір макросів `Fun16`, `Fun32`, `Fun64` можна замінити одним

```
mov ax,Foo%[__BITS__],
```

де вбудований макрос `__BITS__` може мати значення 16, 32, 64.

2.3. Макрос зчеплення виразів `%+`

Окремі вирази однорядкових макросів можуть бути зчеплені з використанням виразу "%+" (після якого потрібно вставити символ пропуску, щоб відрізнити від синтаксису багаторядкових макросів виду %+1). Розглянемо фрагмент коду

```
%define BDASTART 400h  ; початок області даних BIOS
struc tBIOSDA          ; конструкція описання структури
.COM1addr RESW 1
.COM2addr RESW 1
; ..
endstruc               ; конструкція описання структури
```

Доступ до різних елементів структури даних дають інструкції

```
mov ax,BDASTART + tBIOSDA.COM1addr
mov bx,BDASTART + tBIOSDA.COM2addr
```

Ці інструкції можна записати як макрос

```
; Макрос для доступу до BIOS змінних за їх іменами (відносно tBDA):
%define BDA(x) BDASTART + tBIOSDA. %+ x
```

Тоді доступ до різних елементів структури з використанням макросу матиме коротший запис:

```
mov ax,BDA(COM1addr)
mov bx,BDA(COM2addr)
```

2.4. Макроси самопосилання %?, %??

Спеціальні символи %?, %?? можуть використовуватися для самопосилання на свої імена макросів всередині макросів. %? посилається на ім'я макросу вказане при *виклику*, а %?? – вказане при *оголошенні*, наприклад:

```
%idefine Foo mov %?,%??

foo
FOO
```

буде розширена до:

```
mov foo,Foo
mov FOO,Foo
```

2.5. Директива %assign (%iassign)

%assign (регістро незалежна версія %iassign) є альтернативою для визначення однорядкових макросів, які не мають параметрів, а мають числове значення. Це значення можна задати як вираз, який оцінюється один раз при обробці директиви. Макрос визначений %assign можна перевизначити.

```
%assign i 25
%assign i i+1
```

З директивою %assign часто використовується директива макроповторень %rep ... %endrep. В тілі макроповторення можна застосувати директиву %exitrep, яка за умовою припиняє виконання макроповторення, наприклад:

```
%assign i 0
%rep 10
   %if i > 10
       %exitrep
   %endif
   %assign j i+1
   %assign i j
%end rep
```

2.6. Макрос визначення стрічок `%defstr`

`%defstr` (регістро нечутлива версія `%idefstr`) визначає однорядковий макрос, який перетворює праву частину у стрічку

```
%defstr test TEST
```

що еквівалентно до

```
%define test 'TEST'
%defstr PATH %!PATH      ; змінна ОС PATH
```

2.7. Директиви зчеплення стрічок `%strcat, %strlen, %substr`

Директива `%strcat` зчіплює символьні стрічки і назначає їх однорядковому макросу:

```
%strcat alpha "Alpha: ", '12" screen'
```

`Alpha` матиме значення `"Alpha: '12" screen'`

Директива `%strlen` назначає довжину стрічки макросу

```
%strlen charcnt 'my string'
```

В результаті `charcnt` буде присвоєно значення 9.

Директива `%substr` видобуває підстрічку із стрічки:

```
%substr mychar 'xyzw' 1    ; аналогічно до %define mychar 'x'
%substr mychar 'xyzw' 2    ; аналогічно до %define mychar 'y'
%substr mychar 'xyzw' 3    ; аналогічно до %define mychar 'z'
%substr mychar 'xyzw' 2,2  ; аналогічно до %define mychar 'yz'
%substr mychar 'xyzw' 2,-1 ; аналогічно до %define mychar 'yzw'
%substr mychar 'xyzw' 2,-2 ; аналогічно до %define mychar 'yz'
```

3. Багаторядкові макроси `%macro`

В багаторядкових макросах макровизначення поміщається між директивами `%macro` і `%endmacro`. Після директиви `%macro` задається ім'я макросу і кількість параметрів, наприклад:

```
%macro prolog 1
    push rbp
    mov rbp,rsp
    sub rsp,%1
%endmacro
```

де `prolog` – ім'я багаторядкового макросу, а `1` – число параметрів, який отримує макрос, `%1` – операнд, в який підставляється перший параметр.

Виклик макросу:

```
myfunc: prolog 16
```

буде розширений до наступних інструкцій:

```
myfunc: push rbp
mov rbp,rsp
sub rsp,16
```

Приклад макросу з двома параметрами:

```
%macro silly 2
   %2: db %1
%endmacro
```

Якщо потрібно передати кому, як частину параметра, то увесь параметр вказується у фігурних дужках:

```
silly 'a',  letter_a   ;  =>  letter_a:  db 'a'
silly 'ab', string_ab  ;  =>  string_ab: db 'ab'
silly {13,10}, crlf    ;  =>  crlf: db 13,10
```

Приклад макросу з двома параметрами, який реалізує системний виклик `write`:

```
%macro write_string 2
mov rax, 1 ; 1 = write
mov rdi, 1 ; 1 = to stdout
mov rsi, %1; string to display in rsi
mov rdx, %2; length of the string, without 0
syscall ; display the string
%endmacro

section .data
msg1 db 'Hello, programmers!',0xA,0xD
len1 equ $ - msg1
msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $- msg2
msg3 db 'Linux assembly programming! '
len3 equ $- msg3

section .text
global main  ; оголошення для gcc
main:        ; оголошення точки входу для компонувальника
write_string msg1, len1
write_string msg2, len2
write_string msg3, len3
mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = success exit code
syscall ; quit
```

Багаторядкові макроси можуть містити всередині позначки. Якщо в програмі буде декілька макровикликів, то будуть згенеровані тексти програми які міститимуть однакові позначки. В процесі асемблювання це спричинить повідомлення про помилки. Для того, щоб такі позначки не конфліктували одна з одною використовується механізм "локальних позначок у макросах" – позначки, які починаються з символів `%%`. Така позначка в кожному макровиклику буде

замінюватися на унікальний ідентифікатор, наприклад `%%lp` при першому макровиклику буде замінено на `@1.lp`, а при другому – на `@2.lp`.

3.1. Перевантаження макросів

Багаторядкові макроси можна перевантажити використовуючи одне і те ж ім'я, але різну кількість параметрів. Наприклад, макрос без параметрів:

```
%macro push 0
   push rbp
   mov rbp,rsp
%endmacro
```

і макрос з двома параметрами:

```
%macro push 2
   push %1
   push %2
%endmacro
```

Тоді виклики:

```
push rbx     ; макрос не викликається, а виконується команда push rbx
push rax,rcx ; це виклик макросу
```

3.2. Локальні позначки у макросах

Макроси можуть містити локальні позначки, перед якими ставиться знак `%%`:

```
%macro retz 0
   jnz %%skip
   ret
%%skip:
%endmacro
```

3.3. Скупі макроси

Іноді потрібно визначити макрос, який виділяє декілька параметрів, а решту параметрів об'єднує з останнім разом із комами. Для цьому після числа, яке задає кількість параметрів ставиться символ '+'. Нехай є макрос:

```
%macro writefile 2+
   jmp %%endstr
%%str: db %2
%%endstr:
   mov rdx,%%str
   mov rcx,%%endstr-%%str
   mov rbx,%1
   mov rax, 1 ; 1 = write file
   mov rdi, 0 ; 0 = success exit code
   syscall ; quit
%endmacro
```

При виклику макросу з параметрами

```
writefile [filehandle],"hello, world",13,10
```

перший параметр сприймається як `[filehandle]`, а другий параметр об'єднує увесь залишок – `"hello, world",13,10` (розміщується після `db`).

Вищевказаний макрос можна реалізувати як не "скупий", тоді його виклик матиме наступний вид:

```
writefile [filehandle],{"hello, world",13,10}
```

3.4. Змінне число і діапазон параметрів

Для багаторядкових макросів можна задати змінне число параметрів через тире. Наприклад, макрос з числом параметрів від 0 до 1:

```
%macro DIE 0-1 "Аварійне завершення програми"
   writefile 2,%1
   mov rax, 0x4c01
   mov rdi, 0 ; 0 = success exit code
   syscall ; quit
%endmacro
```

Так при виклику макросу без параметрів, видається повідомлення `"Аварійне завершення програми"`, а з параметром виводиться текст заданого повідомлення у `STDERR`.

Якщо верхнє обмеження на число параметрів відсутнє, то це задається символом `"*"`:

```
%macro die 1-*
```

Для розширення параметрів у заданому діапазоні використовується спеціальна конструкція `%{start:end}`, де `start`, `end` – індекси першого і останнього параметрів (індекс може бути позитивний, негативний, але не нуль), наприклад виклики:

```
%macro mpar 1-*
db %{3:5}
%endmacro

mpar 1,2,3,4,5,6
```

розшириться до діапазону 3,4,5;

```
%macro mpar 1-*
db %{5:3}
%endmacro
mpar 1,2,3,4,5,6
```

розшириться до діапазону 5,4,3;

```
%macro mpar 1-*
db %{-1:-3}
%endmacro
mpar 1,2,3,4,5,6
```

розшириться до діапазону 6,5,4.

3.5. Лічильник параметрів макросу `%0`

Параметр макросу `%0` повертає числову константу з числом отриманих параметрів, тобто якщо `%0` дорівнює `n`, то `%n` є останній параметр.

3.6. Прокручування параметрів макросу

Директива `%rotate 1` прокручує параметри макросу по колу вліво на 1 позицію, тому першим появляється перший зліва параметр. Директива `%rotate -1` прокручує параметри макросу по колу вправо на 1 позицію, тому першим появляється перший справа параметр.

```
%macro multipush 1-*    ; відсутнє обмеження на верхнє число параметрів
%rep %0
   push %1
%rotate 1
%endrep
%endmacro
```

3.7. Зчеплення макропараметрів

Можна зчеплювати макропараметри і вбудовані в макроси конструкції з іншим оточуючим тестом. Це дозволяє оголошувати родини символів у макросах. Наприклад, якщо потрібно згенерувати таблицю ключових кодів разом із зміщенням у таблиці, то це можна зробити наступним чином:

```
%macro keytab_entry 2
keypos%1 equ $-keytab
db %2
%endmacro
```

Макровиклики

```
keytab:
     keytab_entry F1,128+1
     keytab_entry F2,128+2
     keytab_entry Return,13
```

розширяться до:

```
keytab:
keyposF1 equ $-keytab
         db 128+1
keyposF2 equ $-keytab
         db 128+2
keyposReturn equ $-keytab
       db 13
```

3.8. Коди умов як макропараметри

Для використання кодів умов як макропараметрів використовується синтаксис %+1, %-1. Синтаксис %+1 вказує, що він посилається на параметр %1, який містить код умови, якщо ж він міститиме не код умови, то препроцесор буде видавати повідомлення про помилку.

Так макрос

```
%macro retc 1
   j%+1 %%skip
   ret
%%skip:
%endmacro
```

при виклику `retc ne` розширить аргумент до `ne`, як до дійсної умови.

Синтаксис %-1 буде посилатися на параметр %1, який містить код умови, яка при розширенні буде інвертуватися. Так макрос

```
%macro retc 1
   j%-1 %%skip
   ret
%%skip:
%endmacro
```

при виклику `retc ne` розширить аргумент до `je`, як до інверсної умови

3.9. Відміна макросів

Для відміни багаторядкових макросів (з параметрами) використовується директива `%unmacro` (з параметрами):

```
%macro foo 1-3
   ; Do something
%endmacro

%unmacro foo 1-3  ; вилучає макрос

%unmacro foo 1    ; не вилучає макрос, на співпадають параметри
```

4. Директиви асемблювання з умовами

При розробленні програм часто виникає необхідність у створенні різних версій виконуваного файлу з використанням одного і того ж початкового (сирцевого) коду. У таких випадках використовуються директиви асемблювання з умовами, які дозволяють вибирати потрібні фрагменти коду.

4.1. Директиви з логічними умовами

4.1.1 Директиви `%if, %elif, %else, %endif`

Директива з логічними умовами має наступний синтаксис:

```
%if<умова1>
; фрагмент коду для умови1, яка виконується
%elif<умова2>
; фрагмент коду для умови2, яка виконується
%else
; фрагмент коду, якщо умова1 і умова2 не виконуються
%endif
```

Підтримується також інверсна форма директив `%ifn` і `%elifn`.

Наприклад, якщо визначене ім'я `DEBUG`

```
%define DEBUG
```

то умова перевірки існування *однорядкового* макросу задається директивою `%ifdef`:

```
%ifdef DEBUG
```

```
   writefile 2,"Function performed successfully",13,10
%endif
```

Якщо визначення `DEBUG` закоментувати:

```
; %define DEBUG
```

то макрос макропроцесором ігнорується і не виконується, тому його можна залишити у тексті програми. Розкоментовуючи або закоментовуючи ім'я макросу можна керувати налагоджувальним друком.

Крім `%ifdef` можна використовувати директиву `%ifndef`, у цьому випадку блок інструкцій буде оброблятися, якщо макровизначення не існує.

Керувати включенням або виключенням імені макровизначення можна при виклику `nasm`:

```
nasm -f  elf  -dDEBUG prog.asm
```

Можна обробляти і більш складні умови. Наприклад, якщо потрібно вибірково вставляти в текст програми фрагмент коду програміста1 або програміста2:

```
%ifdef Programer1
; код програміста 1
%elifdef Programer2
; код програміста 2
%else
; якщо не визначені імена програмістів, то виводиться повідомлення
%error Please define Programer1 or Programer2
%endif
```

При компіляції такої програми необхідно вказати ключ `-dProgramer1` або `-dProgramer2`. Крім наявності імені макросу можна перевіряти і факт його відсутності директивами `%ifndef`, `elifndef`.

4.1.2 Директиви `%ifmacro`, `%else`, `%endif`

Для вибору за умовою багаторядкового макросу використовується директива **`%ifmacro`**:

```
%ifmacro MyMacro 1-3
    %error "MyMacro 1-3" помилка виклику макросу
%else
   %macro MyMacro 1-3
      ; код макросу
   %endmacro
%endif
```

4.1.3 Директива `%ifcntx`

Для тестування *передпроцесорного стеку контекстів* використовується директива **`%ifcntx`** (`%ifnctx`, `%elifctx`, `%elifnctx`). Директива асемблює код, якщо вміст верхівки передпроцесорного стеку контекстів має таке ж ім'я, як один із аргументів.

4.1.4 Директиви `%if expr`, `%ifn expr`

Для тестування довільних *числових виразів* використовується директиви **%if expr**, %elif expr, **%ifn expr**, %elifn expr. %if expr розширює синтаксис Nasm, дозволяючи набір наступних умов порівняння =(==), <, >, <=, >=, <>(!=), &&(and), ||(or), ^^(xor).

Для тестування *ідентичності виразів* text1 і text2, які отримуються після розширення однорядкових макросів, використовується директива **%ifidn text1,text2 (%ifidni** – регістро незалежна). Конструкція %ifidn text1,text2 виконує один код при ідентичності аргументів і інший код – при не ідентичності аргументів.

```
%macro pushparam 1
   %ifidni %1,ip
      call %%label
   %%label:
   %else
      push %1
   %endif
%endmacro
```

Так виклик pusparam eax спричиняє виконання call %%label, а виклик pushparam 5 – push 5.

4.1.5 Директиви %ifid, %ifnum, %ifstr

Для визначення, *який параметр передається у макрос – ідентифікатор (id), число (num) або стрічка (str)* використовуються директиви **%ifid, %ifnum, %ifstr**.

```
%macro writefile 2-3+
   %ifstr %2
      jmp %%endstr
   %if %0 = 3
      %%str: db %2,%3
   %else
      %%str: db %2
   %endif
      %%endstr: mov dx,%%str
                mov cx,%%endstr-%%str
   %else
                mov dx,%2
                mov cx,%3
   %endif
                mov rax, 1 ; 1 = write
                     mov rdi, 1 ; 1 = to stdout
                     mov rsi, msg2 ; string to display in rsi
                     mov rdx, len2 ; length of the string, without 0
                syscall ; display the string
%endmacro
```

Можливі виклики макросу

```
writefile [file], strpointer, length
writefile [file], "hello", 13, 10
```

У першому виклику задається адреса зовнішньої існуючої стрічки і її довжина, а у другому виклику – виділяється пам’ять із ознакою кінця стрічки.

4.1.6 Директиви `%ifempy, iftoken`

Для тестування *значень параметрів*, які передаються в макрос використовуються директиви:

`%ifempy` – оброблення наступного коду макросу, якщо параметр після розширення порожній.

`%iftoken` `1` – оброблення наступного коду макросу, якщо параметр після розширення містить тільки один символ.

`%iftoken` `-1` – не оброблюється наступний код макросу, так як параметр після розширення містить два символи "–" і "1".

5. Передпроцесорний цикл %rep, %endrep, %exitrep

Директиви **%rep, %endrep** використовуються для дублювання частини асемблерного коду задане число разів.

```
%assign i 0
%rep    64
        inc word [table+2*i]
%assign i i+1
%endrep
```

Ця директива згенерує 64 інструкції `inc` збільшуючи адресу кожного слова від `table` до `table+2*64`.

У наступному прикладі генерується список 16-розрядних чисел Фібоначчі:

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
%if j > 65535
    %exitrep
%endif
        dw j
%assign k j+i
%assign i j
%assign j k
%endrep

fib_number equ ($-fibonacci)/2
```

6. Директиви підключення зовнішніх ресурсів `%include, %use`

Директива **`%include`** дозволяє підключити зовнішні макроси або сирцеві коди асемблера в основний файл.

```
%include "macros.mac"
```

Для запобігання повторного підключення файлів використовується макрос:

```
%ifndef MACROS_MAC
    %define MACROS_MAC
    ; визначення макросу
```

```
%endif
```

Директива **%use** дозволяє підключити зовнішні макропакети:

```
%use altreg
%use 'altreg'   ; еквівалентний виклик
```

7. Директива задання шляху пошуку %pathsearch

Директива задає однорядковий макрос з іменем шляху пошуку

```
%pathsearch MyPath "foo.bin"
```

8. Передпроцесорний стек контекстів

Асемблер Nasm підтримує передпроцесорний стек контекстів, кожен з яких задається своїм іменем і є доступним для всіх макросів. Можна додати новий контекст у стек або вилучити директивами **%push** і **%pop.** Директива %push створює новий контекст і поміщає на його верхівку ім'я контексту:

```
%push context1
```

Директива %pop вилучає верхній контекст стеку і руйнує його разом з усіма асоційованими позначками.

Контекстно локальні позначки у певному макросі задаються як %%label. Подібно визначаються позначки директивою %$label, які є локальними до контексту на верхівці контекстного стеку.

Приклади макровизначень з контекстним стеком:

```
%macro repeat 0
   %push repeat
   %$begin:
%endmacro

%macro until 1
   j%-1 %$begin
   %pop
%endmacro
```

Виклик макросів:

```
mov cx,string
repeat
add cx,3
scasb
until e
```

які сканують кожний четвертий байт стрічки у пошуку байту в регістрі al.

Якщо потрібне визначення (або доступ до) локальних позначок контексту нижче від першої верхівки стеку можна використати %$$label, або %$$$label і так далі.

9. Стандартні макроси

Ядро NASM не має засобів для описання структур, тому структури можна реалізувати за допомогою макросів **struc** і **endstruc**. Макроси **struc** і **endstruc** визначають типи даних структури.

Макрос **struc** може мати один або два параметри. Перший параметр задає ім'я типу даних, а другий (необов'язковий) – зміщення бази структури. Всередині структури необхідно визначити поля з використанням псевдоінструкцій **resb**.

```
struc mytype
   mt_long: resd 1    ; 0
   mt_word: resw 1    ; 0+4
   mt_byte: resb 1    ; 4+2=6
   mt_str:  resb 32   ; 6+1=7
endstruc
```

У структурі визначено наступні символи, які мають зміщення від початку структури `mytype`: `mt_long` – 0, `mt_word` – 4, `mt_byte` – 6, `mt_str` – 7 і `mytype_size` – 39 (визначається як EQU `mytype + _size`).

Якщо імена полів структури співпадають з іменами у інших структурах, то можна визначити структуру наступним чином:

```
struc mytype
   .long: resd 1
   .word: resw 1
   .byte: resb 1
   .str:  resb 32
endstruc
```

Це визначає зміщення полів структури як `mytype.long`, `mytype.word`, `mytype.byte`, `mytype.str`.

Іноді відоме тільки зміщення структури, наприклад у стандартному стековому фреймі:

```
push ebp
mov ebp,esp
sub esp,40
```

У цьому випадку можна доступитися до елемента структури віднімаючи зміщення:

```
mov [ebp - 40 + mytype.word],ax
```

Щоб не вказувати зміщення в команді, його можна задати у визначенні структури:

```
struc mytype,-40
```

Тоді доступ до елементів структури буде наступним:

```
mov [ebp + mytype.word],ax
```

9.1. Створення екземплярів структури у пам'яті даних, istruc, at, iend

Маючи визначений тип структури можна створити екземпляр структури і ініціалізувати його у сегменті даних використовуючи конструкцію **istruc ... iend**:

```
mystruc
   istruc mytype
      at mt_long, dd 123456
      at mt_word, dw 1024
```

```
    at mt_byte, db 'x'
    at mt_str, db 'hello, world', 13, 10, 0
iend
```

Макрос **at** використовує префікс `times` для переміщення вказівника асемблера у потрібну точку поля структури, де можна резервувати пам'ять для даних. Тому тут поля мають бути записані у такому ж порядку, як і у визначенні структури.

Дані для одного поля структури можуть записуватися в декількох рядках:

```
at mt_str,  db  123,134,145,156,167,178,189
            db  190,100,0

at mt_str
            db  'hello, world'   ; еквівалентний синтаксис
            db  13,10,0
```

9.2. Макроси вирівнювання коду align, alignb

Макроси **align** або **alignb** вирівнюють код або дані на `word`, `doubleword`, `longword`, `paragraph` або інші границі. Звичайно `align` використовується у секції даних і коду, а `alignb` – у секції BSS.

```
align 4         ; вирівняти на 4-байтову межу
align 16        ; вирівняти на 16-байтову межу
align 8,db 0    ; вирівняти і заповнити нулями, а не NOPs
align 4,resb 1 ; вирівняти на 4 у секції BSS
alignb 4        ; еквівалент попереднього рядка
```

`Alignb` може використовуватися у визначенні структури:

```
struc mytype2
  mt_byte:
        resb 1
        alignb 2
  mt_word:
        resw 1
        alignb 4
  mt_long:
        resd 1
  mt_str:
       resb 32
endstruc
```

10. Приклади простих макросів

Використовуючи макроси можна значно скоротити виклики підпрограм. Розглянемо виклики підпрограм з двома і трьома параметрами. Макроси з іменами `call_2` і `call_3` мають перший параметр – ім'я викликуваної підпрограми для команди `call`, а решта параметрів – аргументи для занесення у стек:

```
%macro call_2 3
    push %3
    push %2
```

```
%macro
call_3 4
        push %4
```

```
    call %1
    add rsp,16
%endmacro
```

```
    push %3
    push %2
    call %1
          add
esp,24
%endmacro
```

Виклики макросів матимуть вид:

```
call_2 myproc rax, 1, 2
```

```
call_3 myproc rax, 1, 2, 3
```

Саме макророзширення буде наступним: макропроцесор замінить входження `%1, %2, %3` на відповідні параметри. В результаті макророзширення буде отримано наступний текст:

```
push rax
push 1
push 2
call myproc
add rsp 16
```

```
push rax
call myproc
add rsp 8
```

11. Директиви асемблера

Директиви асемблера – це макровизначення, визначені самим асемблером.

Директива **`BITS N`** – задає режим роботи процесора (16-, 32- або 64-розрядний).

Директива **`SECTION`** або **`SEGMENT`** – визначає сегменти програми: **`.text`** – сегмент коду, **`.data`** – сегмент статичних даних, **`.bss`** – сегмент динамічних даних. Статичні дані – це дані відомі під час компіляції. Динамічні дані – це неініціалізовані дані, яким під час компіляції відводиться тільки пам'ять, а значення не присвоюються.

Директива **`ABSOLUTE`** задає абсолютну адресу не фізичної, а уявної секції. Використовується тільки з псевдоінструкціями родини `resxx`:

```
absolute 0x1A
    kbuf_chr    resw    1
    kbuf_free   resw    1
    kbuf        resw    16
```

Директива **`EXTERN`** – визначає зовнішні ідентифікатори для "імпорту" у програму:

```
extern  _printf
extern  _sscanf,_fscanf
```

Директива **`GLOBAL`** – визначає ідентифікатори для "експорту", вони позначаються як глобальні і можуть використовуватися іншими модулями (програмами).

```
global _main
_main:
        ; some code
```

Директива **`COMMON`** – використовується замість **`GLOBAL`** для оголошення спільних змінних у секції **`.bss`**.

```
common  intvar  4
; є подібним до
global  intvar
section .bss
```

```
intvar  resd    1
```

Директива **CPU** – вказує процесору генерувати команди тільки для вказаного типу процесора.

```
CPU 486 486 instruction set
```

Директива **ORG** – встановлює початкову адресу для завантаження програми.

12. Дизасемблер NDIASM

NDIASM призначений тільки для дизасемблювання бінарних файлів. Виклик **NDIASM**:

```
ndisasm –b {16|32|64} filename
```

Контрольні запитання.

1. Макропроцесор, макрос, ім’я макросу і макророзширення.
2. Однорядкові макроси.
3. Багаторядкові макроси `%macro … %endmacro`.
4. Директиви асемблювання з умовами.
5. Передпроцесорний цикл `%rep, %endrep, %exitrep`
5. Визначення і створення структур `struc, endstruc, istruc, iend`.
6. Директиви асемблювання.

5. ЦІЛОЧИСЛОВІ КОМАНДИ

Мета. Вивчення функціональної класифікації команд для роботи з цілими числами, команд пересилання даних, арифметичних команд, логічних команд і операцій, ланцюгових команд

Вступ. Кожна родина процесорів має свій набір машинних команд. З появленням нових моделей процесорів зростає і кількість команд, яка відображає архітектурні нововведення. Набори машинних команд можна класифікувати за функціональним призначенням і структурувати за групами. Машинні команди у виділених групах мають подібний синтаксис і призначення, що дозволяє краще вивчити можливості окремих команд. Кожна машинна команда записується у відповідному форматі.

Одними із найбільш використовуваних є команди пересилання даних, арифметичні команди, логічні команди і операції, ланцюгові команди.

План.

1. Функціональна класифікація цілочислові машинних команд
2. Команди пересилання даних
   2.1. Команди одно- і двонаправленого пересилання даних
   2.2. Введення-виведення у порт
   2.3. Робота з адресами і вказівниками
   2.4. Перекодування даних
   2.5. Робота зі стеком
3. Арифметичні команди
   3.1. Команди перетворення типу
   3.2. Арифметичні операції над цілими числами
   3.3. Арифметичні операції над двійково-десятковими числами
   3.4. Інші команди з арифметичним принципом
4. Класифікація логічних команд і операцій
   4.1. Логічні побітові команди
   4.2. Бітові операції над виразами
   4.3. Обчислення поточних адрес $, $$
   4.4. Сканування, перевірка і модифікація бітів
   4.5. Команди зсуву
   4.6. Лінійний зсув
   4.7. Циклічний зсув
5. Ланцюгові команди

1. Функціональна класифікація цілочислових машинних команд

Машинна команда є закодованою за певними правилами інструкцією процесору на виконання деякої операції або дії. Кожна команда містить елементи, які визначають:

- що робити (задається кодом операції);
- об’єкти, над якими потрібно щось робити (операнди);
- як робити (задається типами операндів, які звичайно задаються неявно).

Машинні команди розділити за наступними групами:

• General purpose (команди загального призначення цілочислові);
• x87 FPU (команди співпроцесора для роботи з числами з плаваючою крапкою);
• x87 FPU (команди співпроцесора для роботи з командами SIMD);
• Intel® MMX technology (команди цілочислового розширення MMX);
• SSE extensions (команди розширення SSE);
• SSE2 extensions (команди розширення SSE2);
• SSE3 extensions (команди розширення SSE3);
• SSSE3 extensions (команди розширення SSSE3);
• SSE4 extensions (команди розширення SSE4);
• AESNI and PCLMULQDQ
• Intel® AVX extensions (команди розширення AVX);
• F16C, RDRAND, RDSEED, FS/GS base access
• FMA extensions (команди розширення FMS);
• Intel® AVX2 extensions (команди розширення AVX2);
• Intel® Transactional Synchronization extensions (команди розширення синхронізацій);
• System instructions (системні команди);
• IA-32e mode: 64-bit mode instructions (команди 64-розрядні);
• VMX instructions (команди розширення VMX);
• SMX instructions (команди розширення SMX);
• ADCX and ADOX
• Intel® Memory Protection Extensions (команди розширення захисту пам’яті);
• Intel® Security Guard Extensions (команди розширення безпеки).

Функціональна класифікація цілочислових команд показана на рис. 1.

Рисунок 1 – Класифікація цілочислових команд

2. Команди пересилання даних

До цієї групи відносяться команди:
- пересилання даних;
- введення-виведення у порт;
- робота з вказівниками і адресами;

- перетворення даних;
- робота зі стеком

2.1.Команди одно- і двонаправленого пересилання даних

До цієї групи відносяться наступні команди:

• однонаправленого пересилання даних `mov`;

• двонаправленого пересилання даних `xchg`.

Команда `mov` має два операнди `mov <операнд призначення>, <операнд джерело>`.

Перший операнд задає те місце, коди будуть скопійовані дані, а другий операнд – те місце, звідки будуть копіюватися дані. Команда **`mov`** тільки **копіює** дані не виконуючи ніяких перетворень. В команді `mov`, а також в інших командах, не можна використовувати одночасно два операнди типу пам'ять. Варіанти використання команда `mov`:

```
     mov destreg, constant          ;destreg := constant
                                    ;destreg, регістр ЗП 8, 16, 32, 64 біт
     mov type [destmem], constant   ;destmem := constant
                                       ;destmem, комірка пам'яті розміром 8,16,32,64
біти
                                    ;type=byte, word, dword
     mov destreg, srcreg            ;destreg := srcreg
                                    ;destreg і srcreg,  регістр ЗП 8,16,32,64 біт
                                    ;регістри мають бути одного розміру
     mov destreg, [srcmem]          ;destreg := srcmem
                                    ;destreg, регістр ЗП 8,16,32,64 біт
                                    ;srcmemn, комірка пам'яті того ж розміру
     mov destmem, [srcreg]          ;desttmem := srcreg
                                    ;srcreg, регістр ЗП 8,16,32,64 біт
                                    ;destmemn, комірка пам'яті того ж розміру
```

Для команди `mov` є допустимі наступні комбінації операндів:

• `регістр, безпосереднє значення`
• `пам'ять, безпосереднє значення`
• `регістр, регістр`
• `регістр, пам'ять`
• `пам'ять, регістр`

У комбінації операндів `пам'ять, безпосереднє значення` потрібно вказати, скільки байт, починаючи із заданої адреси пам'яті потрібно записати. Для цього використовується специфікатор розміру – `byte, word, dword, qword`. Наприклад, записати число 15 у 4-байтову область пам'яті, яка знаходиться за адресою `x`, можна так:

```
mov [x], dword 15
; або так
mov dword [x], 15
```

Приклад, як записати у регістр число або вміст іншого регістру:

```
mov eax, 100   ; eax = 0x0000_0064, старша половина rax = 0
mov rcx, -1    ; rcx = 0xffff_ffff_ffff_ffff
mov ecx, eax   ; ecx = 0x0000_0064
```

Приклад, виконання базових операцій над даними. Нехай є оголошення даних:

```
dValue dd 0
bNum db 42
wNum dw 5000
dNum dd 73000
qNum dq 73000000
bRes db 0
wRes dw 0
dRes dd 0
qRes dq 0
```

Над даними потрібно виконати операції:

```
dValue = 27
bRes = bNum
wRes = wNum
dRes = dNum
qRes = qNum
```

Наступні команди реалізують ці операції:

```
mov dword [dValue], 27 ; dValue = 27
mov al, byte [bNum]
mov byte [bRes], al    ; bRes = bNum
mov ax, word [wNum]
mov word [wRes], ax    ; wRes = wNum
mov eax, dword [dNum]
mov dword [dREs], eax  ; dRes = dNum
mov rax, qword [qNum]
mov qword [qRes], rax  ; qRes = qNum
```

Команда двонаправленого пересилання (обміну місцями значень) даних `xchg` має два операнди `xchg <операнд 1>, <операнд 2>`. Операнди мають бути одного типу. Не допускається обмін вмісту двох комірок пам'яті.

Варіанти використання команди `xchg`:

```
xchg reg, reg  ; reg := регістр ЗП 8,16,32,64 біт
xchg reg, mem  ; reg := регістр ЗП 8,16,32,64 біт
               ; mem комірка пам'яті того ж розміру
xchg mem, reg  ; reg := регістр ЗП 8,16,32,64 біт
               ; mem комірка пам'яті того ж розміру
```

Приклади використання команди `xchg`:

```
xchg rax,rbx          ; обмін вмісту регістрів rax, rbx
xchg eax,ebx          ; обмін вмісту регістрів eax, ebx

xchg al,ah            ; обмін місцями значень al, ah
xchg ax, word [si]    ; обмін вмісту регістра ax і слова в пам'яті за
                      ; адресою [si]
```

2.2. Введення-виведення у порт

Кожний пристрій введення-виведення має один або декілька регістрів, доступних через *адресний простір введення-виведення*. Ці регістри мають розрядність 8-, 16-, 32-біти. Адресний простір *введення-виведення* фізично незалежний від простору оперативної пам'яті і має

обмежений обсяг з 2^{16}=65536 адрес. Таким чином поняття порту можна визначити як 8-, 16-, 32-розрядний апаратний регістр, який має певну адресу в адресному просторі *введення-виведення*.

Варіанти використання команд введення у порт `in` і виведення з порту `out`:

```
in акумулятор, порт
out порт, акумулятор
in reg_ax, imm       ; ax  := al, ax, eax
in reg_ax, reg_dx    ; imm := resb, resw, resd, resq
out imm, reg_ax
out reg_dx, reg_ax
```

Команда `in` читає дані з порту введення/виведення, номер якого міститься в регістрі `dx`, і поміщає дані в регістри `al/ax/eax`. Інші регістри крім `al/ax/eax` і `dx` використовувати не можна.

Номери портів від 0 до 255 вказуються безпосередньо, а більші номери задаються у регістрі `dx`.

```
mov dx,300
in eax,dx
```

Команда `out` out пересилає дані у порт. Типи її операндів такі ж як і в команди `in`, але вони вказуються у зворотному порядку.

Діапазони портів введення/виведення, які зв'язані з різними портами, показані у табл. 1.

Таблиця 1 – Діапазони портів введення виведення

0000-001f: dma1	Перший контролер DMA
0020-003f: pid	Перший апаратний контролер переривань
0040-005f: timer	Системний таймер
0060-006f: keyboard	Клавіатура
0070-007f: rtc	Годинник реального часу (RTC)
0080-008f: dma page reg	Регістр сторінок DMA
00a0-00bf: pic2	Другий апаратний контролер переривань
00c0-00df: dma2	Другий контролер DMA
00f0-00ff: fpu	Математичний співпроцесор
0170-0177: ide1	Другий IDE контролер
01f0-01f7: ide0	Перший IDE контролер
0213-0213: isapnp read	Інтерфейс PnP (plug-end-play) шини ISA
0220-022f: soundblaster	Звукова карта
0290-0297: w83781d	Апаратний монітор температури і напруги
0376-0376: idel	Другий IDE контролер (продовження)
03c0-03df: vga+	Відеоадаптер
03f2-03f5: floppy	Дисковод для гнучких дисків
03f6-03f6: ide0	Перший IDE контролер (продовження)
03f7-03f7: floppy DIR	Дисковод для гнучких дисків (продовження)
03f8-03ff: lirc_serial	Послідовний порт
0a79-0a79: isapnp write	Інтерфейс PnP (plug-end-play) шини ISA (продовження)
0cf8-0cff: PCI conf 1	Перший конфігураційний регістр шини PCI
4000-403f: Intel 82371AB/EB/MB PIIX4 ACPI	Набір мікросхем ACPI
5000-501f: Intel 82371AB/EB/MB PIIX4 ACPI	Набір мікросхем ACPI
e000-e01f: Intel 82371AB/EB/MB PIIX4 USB	Набір мікросхем USB
f000-f00f: Intel 82371AB/EB/MB PIIX4 IDE	Набір мікросхем контролера дисків

2.3.Робота з адресами і вказівниками

Асемблер інтенсивно працює з адресами операндів в пам'яті. Для підтримки таких операцій є спеціальна група команд завантаження `far` вказівника (сегмент_зміщення):

• `lea reg, джерело` – завантаження виконавчої (ефективної) адреси, `reg=16|32|64`, `джерело=mem|imm`.

• `lds reg, mem` – завантаження вказівника у регістр сегмента даних `ds`, `reg=16|32`;

• `les reg, mem` – завантаження вказівника у регістр додаткового сегмента даних `es`, `reg=16|32`;

• `lfs reg, mem` – завантаження вказівника у регістр додаткового сегмента даних `fs`, `reg=16|32|64`;

• `lgs reg, mem` – завантаження вказівника у регістр додаткового сегмента даних `gs`, `reg=16|32|64`;

• `lss reg, mem` – завантаження вказівника у регістр додаткового сегмента даних `ss`, `reg=16|32|64`;

Команда `lea` (мнемоніка від англ. Load Effective Address) обчислює виконавчу (ефективну) адресу (тобто зміщення даних від початку сегмента даних) і пересилає її в перший операнд. Команда зручна у тих випадках коли не потрібно звертатися до пам'яті, а тільки обчислити адресу:

```
Msg db ”Hello world”,10
lea rax,[Msg]    ; адреса Msg
lea rax,[1000+ebx+8*rcx]   ; адреса комірки пам'яті
```

Команда множить значення регістра `rcx` на 8, додає до добутку значення з регістра `rbx` та число 1000 і отриманий результат пересилає у регістр `rax`.

Альтернативно, команда `lea` може використовуватися для виконання арифметичних операцій +, -, *, / над цілими числами.

```
// функція на Сі
int simpleArithmetic(int x) {
    return 5 * x + 7;
}
; підпрограма на асемблері з використанням команди leas
simpleArithmetic:
        lea     eax, [rdi+7+rdi*4]
        ret
;  підпрограма на асемблері з використанням команд mul, add
simpleArithmetic:
        mov     eax, rdi
        mul     eax, 5
        add     eax, 7
        ret
```

2.4. Перекодування даних

Команда `xlat` замінює значення в регістрі `al` іншим байтом з таблиці перекодування, яка розміщена у пам'яті. Адреса цієї таблиці має бути попередньо завантажена у регістр `bx`.

Значення у регістрі `al` використовується як індекс (зміщення) для пошуку значень заміни з 256-байтової таблиці перекодування.

```
hex_table times 256 db 'a'  ; таблиця перекодування
symbol db 'x'               ; символ для перекодування
...
mov rbx,hex_table    ; адреса таблиці
mov al,[symbol]      ; заміна символу в al
xlat
```

2.5. Робота із стеком

Стек – область пам'яті, спеціально виділена для тимчасового зберігання вмісту регістрів або адрес пам'яті. Стек працює за принципом LIFO (дані поміщені у стек останніми, будуть "виштовхнуті" із стеку першими).

Стек зберігається у пам'яті і розміщується в окремому сегменті. Верхівка стеку задається парою `ss:rsp/ss:esp/ss:sp` .

Для роботи зі стеком призначені наступні регістри:

- `ss` – сегментний регістр стеку;
- `rsp/esp/sp` – регістр вказівник верхівки стеку;
- `rbp/ebp/bp` – регістр вказівник бази кадру стека.

Розмір стеку залежить від режиму роботи процесора (4 Гбайт в захищеному режимі). В кожний момент часу доступний тільки один стек, адреса сегменту якого знаходиться у регістрі `ss`. Для того щоб перемкнутися в інший стек, необхідно завантажити у регістр `ss` іншу адресу. Регістр `ss` автоматично використовується для виконання всіх команд, які працюють зі стеком.

Особливості роботи зі стеком:

- записування і читання даних в стек здійснюється за принципом "останнім прийшов, першим пішов";
- при записуванні даних у стек, він зростає у сторону зменшення адрес (зверху вниз);
- стек може містити тільки 16-, 32- або 64-регістри;
- при використанні регістрів `rsp/esp/sp, rbp/ebp/bp` для адресації пам'яті, асемблер автоматично приймає їх вміст як зміщення відносно регістра `ss`.
- регістр `rsp/esp/sp` завжди вказує на верхівку стеку, тобто містить зміщення за яким був записаний останній елемент;
- регістр `rbp/ebp/bp` – вказівник бази кадра стеку.

Стек використовується:

- для зберігання/відновлення значень любих регістрів;
- при викликах підпрограм, для збереження адрес повернення, для передачі фактичних параметрів в підпрограми та для зберігання локальних змінних.

Саме використання стеку дозволяє реалізувати механізм рекурсії.

Виклик підпрограм з передачею їм параметрів може бути вкладеним. Для розділення параметрів і локальних змінних різних підпрограм вводять поняття кадра стеку. Він містить адресу повернення, аргументи підпрограми, локальні змінні підпрограми. В `ebp` пересилається адреса верхівки стеку попереднього фрейму, яка стає адресою початку нового кадра стеку. `ebp` використовується для доступу до любих елементів нового кадра стеку..

Команди роботи зі стеком:

- занесення у стек регістрів загального призначення у наступному порядку (r|e)ax, (r|e)cx, (r|e)dx, (r|e)bx, (r|e)sp, (r|e)bp, (r|e)si, (r|e)di:

• push reg ; де reg := 16-,32-,64-біт,ds,cs,ss,es,fs,gs регістри

- регістрів (e)ax, (e)cx, (e)dx, (e)bx, (e)sp, (e)bp, (e)si, (e)di (у вказаному порядку, значення esp у стані до виконання команди):

• pusha занесення у стек – всі регістри 16-бітові

• pushad – всі регістри 32-бітові

• pushf, pushfw, pusfd,pushfq ; занесення у стек регістра прапорів 16-, 32- і 64-бітового

– занесення у стек значень із комірок пам'яті і безпосередніх значень:

• push mem ; де mem := 8-,16-,32-,64-біт комірки пам'яті

• push imm ; де imm := 8-,16-,32 безпосереднє значення

- видобування даних зі стеку:

• pop reg ; де reg := 16-,32-,64-біт,ds,cs,ss,es,fs,gs регістри

• popa, popaw,popad ; відновлення вмісту регістрів (r|e)ax, (r|e)cx, (r|e)dx, (r|e)bx, (r|e)sp, (r|e)bp, (r|e)si, (r|e)di

• popf, popfw,popfd,popfq ; відновлення вмісту регістра прапорів 16-, 32- і 64-бітового

Команди push rax і pop rax можна реалізувати за допомогою пари команд:

```
sub rsp,8          ; rax 8-байтовий регістр
mov [ss:rsp],rax   ; запис eax в стек

mov rax,[ss:rsp]   ; помістити в rax верхівку стеку
add rsp,8          ; вилучити останнє значення типу qword із стеку
```

3. Арифметичні команди

Процесор може виконувати цілочислові операції і операції з плаваючою крапкою. Для цього у його архітектурі є два окремих блоки:

- пристрій для виконання цілочислових операцій;
- пристрій для виконання операцій з плаваючою крапкою.

Кожен з цих пристроїв моє свою систему команд. Пристрій для виконання цілочислових команд підтримує команди для роботи з двома типами чисел:

- цілі двійкові числа із знаком або без знаку;
- цілі десяткові числа.

Арифметичні команди асемблера поділяються на наступні групи:

• цілочислова арифметики:

  • перетворення типу (cbw, cwd, cwde, cdq, movsx, movzx);

  • двійкової арифметики:

    - додавання add, adc, inc;
    - віднімання sub, sbb,dec;
    - множення mul, imul;
    - ділення div, idiv;
    - зміни знаку neg;
    - інверсії бітів not;

  • десяткової арифметики:

- корекції додавання `aaa, daa`;
- корекції віднімання `aas, das`;
- корекції множення `aam`;
- корекції ділення `aad`;

• інші команди з арифметичним принципом:

- зміна знаку `neg;`
- інверсії бітів `not`;
- переставлення байтів (32-біт) `bswap`

```
00 66 6E 69 вхід
69 6E 66 00 вихід
```

- порівняння операндів `cmp;`
- порівняння операндів з обміном `cmpxchg;`
- атомарне додавання з можливістю lock `xadd`.

Ціле двійкове число – це число, закодоване у двійковій системі. Розмір цілого двійкового числа може бути 8-, 16-, 32- або 64-біти. Числа з фіксованою крапкою оголошуються і ініціалізуються директивами `db, dw, dd, dq`.

Десяткові числа – це спеціальний вид подання числової інформації, в основу якого покладений принцип кодування кожної десяткової цифри числа групою з чотирьох біт. При цьому кожний байт числа містить одну або дві десяткові цифри у так званому двійково-десятковому коді (BCD – binary coded decimal). Процесор зберігає BCD-числа у двох форматах:

• *запакований формат* – кожний байт містить дві десяткові цифри. Десяткова цифра задається двійковим значенням у діапазоні від 0 до 9 розміром 4 біти. При цьому код старшої цифри займає старші 4 біти. Отже, діапазон подання десяткового запакованого числа в одному байті складає від 00 до 99;

• *незапакований формат* – кожний байт містить одну десяткову цифру в чотирьох молодших бітах. Старші чотири біти мають нульове значення. Це так звана зона. Отже, діапазон подання десяткового незапакованого числа в одному байті складає від 0 до 9.

Двійково-десяткові числа оголошуються і ініціалізуються директивами `db, dt`.

Запаковані константи BCD у стилі x87 можна використовувати в тих самих контекстах, що й 80-розрядні числа з плаваючою крапкою. Вони мають суфікс `p` або префікс `0p` і можуть містити до 18 десяткових цифр. Як і з іншими числовими константами, підкреслення можна використовувати для розділення цифр. Наприклад:

```
dt 12_345_678_901_245_678p
dt -12_345_678_901_245_678p
dt +0p33
dt 33p
```

3.1. Команди перетворення типу

Операнди команд арифметичних операцій можуть мати різні розміри. Для збільшення розмірів операндів використовуються команди перетворення типу. Ці команди розширюють байти у слова, слова – у подвійні слова, подвійні слова – в чотирні слова. Команди перетворення типу особливо корисні при перетворенні цілих із знаком, так як вони автоматично заповнюють старші біти новостворюваного операнда значеннями знакового біта старого об’єкта. Ця операція приводить до цілих значень того ж знаку і тієї ж величини, що і початкова, але вже у більш довгому форматі. Подібна операція називається операцією поширення знаку.

Існує два види команд перетворення типу:

1. Команди без операндів, які працюють з фіксованими регістрами:

• cbw (Convert byte to word)– перетворення байта у регістрі al у слово, шляхом поширення значення старшого біта al на всі біти регістра ah;

Перетворюване значення з регістрі al до розміру слова поміщається в ax, при цьому вільні старші біти ax заповнюються знаковим бітом al (al -> ax).

Результат виконання для негативного і позитивного числа:

```
mov al,-1 ; al = 10000001b
cbw       ; ax = 11111111_10000001b

mov al,1 ; al = 00000001b
cbw      ; ax = 00000000_00000001b
```

• cwd (Convert word to double)– перетворення слова у регістрі ax у подвійне слово в регістрі dx:ax, шляхом поширення значення старшого біта ax на всі біти регістра dx;

Перетворюване значення з регістра ax до розміру подвійного слова поміщається у пару регістрів dx:ax, при цьому вільні біти dx заповнюються знаковим бітом ax. (ax -> dx:ax)

```
mov ax,-1  ; ax = 10000000_00000001b
cwd        ; dx = 11111111_11111111   ax = 10000000_00000001b

mov ax,1   ; ax = 00000000_00000001b
cwd        ; dx = 00000000_00000001   ax = 0000000_000000001b
```

• cwde – перетворення слова у регістрі ax у подвійне слово в регістрі eax, шляхом поширення значення старшого біта ax на всі біти регістра eax;

• cdq (Convert double to quadruple) – перетворення подвійного слова у регістрі eax у чотирне слово в регістрі edx:eax, шляхом поширення значення старшого біта eax на всі біти регістра edx;

Перетворюване значення у регістрі eax у чотирне слово поміщається у пару регістрів edx:eax, при цьому вільні біти edx заповнюються знаковим бітом eax. (eax -> edx:eax)

```
mov eax,-1  ; eax = 10000000_00000000_00000000_00000001b
cdq
            ;edx =  11111111_11111111_11111111_11111111b
            ;eax =  10000000_00000000_00000000_00000001b

mov eax, 1  ; eax = 00000000_00000000_00000000_00000001b
cdq
            ;edx =  00000000_00000000_00000000_00000000b
            ;eax =  00000000_00000000_00000000_00000001b
```

2. Команди movsx, movsxd, novzx, які відносяться до команд оброблення стрічок. Ці команди мають корисні властивості, що використовуються при перетворенні типів:

• movsx, movsxd операнд1, операнд2 – переслати операнд2 з розширенням в операнд1 з поширенням значення старшого біта операнд2 на всі старші біти операнда1. Дану команду використовують для підготовки операндів із знаком для виконання арифметичних дій.

• `movzx операнд1, операнд2` – переслати `операнд2` з розширенням в `операнд1` з заповненням нулями старших розрядів `операнд2`. Дану команду використовують для підготовки операндів без знаків для виконання арифметичних дій.

Команда поширення знакового біту movs:

```
movsx reg16,reg/mem8          ; знаковий біт поширюється до 16-го біту
movsx reg32,reg/mem8/mem16    ; знаковий біт поширюється до 32-го біту
movsx reg64,reg/mem8/mem16    ; знаковий біт поширюється до 64-го біту
movsxd reg64,reg/mem16/mem32 ; знаковий біт поширюється до 64-го біту
movzx reg16/reg32/reg64, reg/mem8/mem16 ; заповнення старших розрядів 0-ми
```

Приклад копіювання знакового числа без і з поширенням знаку:

```
mov ax,-42                              mov ax,-42
mov ebx,eax ; ebx=65494 (0xffd6)        movsx ebx,ax   ; ebx=-42
```

3.2. Арифметичні операції над цілими числами

Команди додавання цілих чисел без знаку:

`inc операнд1` – операція інкременту, тобто збільшення значення операнда на 1;

```
inc word [wVvar]
inc rax
inc dword [dVar]
inc qword [qVar]
```

`add операнд1, операнд2` – команда додавання за принципом дії `операнд1 = операнд1 + операнд2;`

```
add cx, word [wVvar]
add rax, 42
add dword [dVar], eax
add qword [qVar], 300
add rdx,12        ; збільшити на 12 вміст регістра rdx
add rax,rbx       ; до значення регістра rax додати значення регістра rbx
add qword [x],12  ; до значення 8-байтової комірки з адресою x додати 12
```

`adc операнд1, операнд2` – команда додавання з врахуванням прапора перенесення `cf` `операнд1 = операнд1 + операнд2 + значення_cf;`

```
adc rcx, qword [dVvar1]
adc rax, 42
```

Команди віднімання цілих чисел без знаку:

`dec операнд1` – операція декременту, тобто зменшення значення операнда на 1;

```
dec word [wVvar]
dec rax
dec dword [dVar]
dec qword [qVar]
```

`sub операнд1, операнд2` – команда віднімання за принципом дії `операнд1 = операнд1 – операнд2;`

```
sub cx, word [wVvar]
sub rax, 42
sub dword [dVar], eax
sub qword [qVar], 300
```

```
sub [x],rcx  ; із значення 8-байтової комірки з адресою x
             ; відняти значення регістра rcx
```

`sbb операнд1, операнд2` – команда віднімання з врахуванням прапора позики `cf`
`операнд1 = операнд1 - операнд2 - значення_cf;`

В результаті виконання команд `add, sub` виставляються прапори `ZF, SF, OF, CF`.

`ZF=1 (нуль)`, якщо в результаті останньої операції отримано нуль.

`SF=1 (знак)`, якщо отримано негативне число. Для знакових чисел це означає негативне число, а для беззнакових він не має ніякого змісту.

`OF=1 (переповнення)`, якщо відбулося перенесення/позика із старшого знакового розряду при арифметичних операціях для чисел із знаком. Це означає, що в результаті додавання двох позитивних чисел отримано негативне число, або навпаки, при додаванні двох негативних чисел отримано позитивне число. Прапор можна розглядувати як вказівник "беззнакового переповнення".

`CF=1`, якщо відбулося перенесення/позика із старшого розряду при арифметичних або інших операціях. В цьому розумінні прапор `CF` аналогічний до прапору `OF` для знакових чисел (результат не помістився в розмір операнда). Для знакових чисел прапор `CF` не має змісту.

Наявність прапора перенесення дозволяє організувати додавання і віднімання чисел з врахуванням перенесення або позики із старшого розряду (команди `adc` і `sbb`). Наприклад, є два 128-бітові числа, перше записано в пару регістрів `rdx:rax`, а друге – в `rbx:rcx`. Тоді додати ці два числа можна командами

```
add rax, rcx ; додавання молодших частин
adc rdx, rbx ; додавання старших частин з врахуванням перенесення
```

якщо потрібно їх відняти, то використовуються команди

```
sub rax, rcx ; віднімання молодших частин
sbb rdx, rbx ; віднімання старших частин з врахуванням позики
```

В командах інкременту `inc` і декременту `dec` операнд може бути регістровим або типу "пам'ять" `[mem]`. Команди встановлюють прапори `ZF, OF, SF`. Приклад використання команд:

```
dec rax            ; зменшити значення регістра rax на 1
inc qword [count]  ; необхідно вказати розмір операнда в пам'яті і збільшити
                   ; його значення на 1
```

Приклад організації циклу з використанням команди `dec`:

```
mov rax,5
Do: dec rax
...
jnz Do
```

Команди цілочислового множення і ділення.

Команди цілочислового множення `mul` і ділення `div` мають один операнд, який задає *другий множник* в командах множення і *дільник* в командах ділення, причому цей операнд може бути тільки регістровим або типу "пам'ять". Для задання першого множника і діленого використовується неявний операнд, який поміщається в регістри `rax/eax/ax/al`, а при необхідності і регістрові пари `rdx:rax/edx:eax/dx:ax`. Необхідно зауважити, що результат множення двох *n*-розрядних чисел може поміститися тільки в 2*n*-розрядному регістрі результату. В табл. 2 показано розміщення неявного операнда і результату операцій цілочисельного множення і ділення в залежності від розрядності явного операнда.

Таблиця 2 – Розміщення операндів і результату операцій `mul` і `div`

Множення				Ділення		
Операнд 1 неявне множене	Операнд 2 явний множник	результат множення	Операнд 1 неявне ділене	Операнд 2 явний дільник	Частка	Залишок
al	reg8/mem8	ax	ax	reg8/mem8	al	ah
ax	reg18/mem16	dx:ax	dx:ax	reg18/mem16	ax	dx
eax	reg32/mem32	edx:eax	edx:eax	reg32/mem32	eax	edx
rax	reg64/ mem16/32/64	rdx:rax	rdx:rax	reg64/ mem16/32/64	rax	rdx

```
mul word [wVvar]
mul al
mul dword [dVar]
mul qword [qVar]
```

Для множення беззнакових чисел використовується команда `mul`, а для множення знакових – `imul`. Команди `mul` і `imul` встановлюють прапори `CF` і `OF`, якщо старша половина результату дорівнює нуль.

```
imul ax, 17
imul al
imul ebx, dword [dVar]
imul rbx, dword [dVar], 791
imul rcx, qword [qVar]
imul qword [qVar]
```

Для ділення беззнакових чисел використовується команда `div`, а для ділення знакових – `idiv`. Значення прапорів після операцій цілочисельного ділення не визначені.

```
div word [wVvar]
div bl
div dword [dVar]
div qword [qVar]
```

Приклади:

```
; 888*77
mov ax,888    ; неявний множник
mov bx,77     ; явний множник
mul bx        ; 888*777=68376=0x10b18  (результат) => edx:eax (1:0b18)
mov dx,0x1    ; неявне ділене dx:ax
mov ax,0xb18 ; неявне ділене dx:ax  ; 0x10b18
mov bx,77     ; явний дільник
div bx        ; 68376/77=88 (результат) => ax
```

3.3. Арифметичні операції над двійково-десятковими числами

Двійково-десяткові числа (BCD-числа) використовуються у застосунках, де потрібні великі і точні числа, наприклад в фінансовій сфері. Двійкові числа не можуть забезпечити таких вимог, так як вони мають обмежений діапазон значень та помилки округлення, не можуть безпосередньо подаватися у символьному виді (ASCII-код).

Спеціальних команд для роботи з BCD-числами не має, так як розрядність таких чисел може бути як потрібно великою. Додавати і віднімати BCD-числа можна як в запакованому, так і в незапакованому форматі, а множити і ділити можна тільки в незапакованому форматі.

Додавання незапакованих BCD-чисел.

Розглянемо два випадки додавання незапакованих BCD-чисел.

```
06 = 0000_0110          06 = 0000_0110
+                       +
03 = 0000_0011          07 = 0000_0111
=                       =
09 = 0000_1001          13 = 0000_1101
Результат BCD вірний    +          0110      Результат BCD не вірний
                        =     0001_0011      Корекція – додати в молодшу тетраду 6
                                             Результат BCD вірний
```

У першому випадку сума чисел у тетраді результату не більша 9, тобто вірна. У другому випадку сума чисел в тетраді результату більша 9, виникає перенесення у старшу тетраду, тому результат в тетрадах невірний. В процесорі для вирішення даної проблеми добавили команду `aaa` (ASCII adjust for Addition), яка добавляє у молодшу тетраду число 6 і тим сами корегує значення у тетрадах результату до вірного значення.

`aaa` – корекція результату додавання BCD-чисел

Команда не має операндів. Вона працює неявно тільки з регістром `al` і аналізує значення його молодшої тетради. Якщо значення не більше 9, то прапор `cf` скидається у 0, інакше – встановлюється в 1 і виконуються наступні дії:

- до вмісту молодшої тетради додається 6;
- прапор `cf` встановлюється в 1, чим фіксується перенесення у старшу тетраду для можливості врахування цього у наступних діях, наприклад командою `adc`.

Приклад:

```
a db 7
b db 5
sum db 0,0
…
mov al,[a]
adc al,[b]
aaa
```

Віднімання незапакованих BCD-чисел.

Ситуація, яка виникає при відніманні BCD-чисел, є аналогічною як при додаванні.

Розглянемо два випадки віднімання незапакованих BCD-чисел.

```
06 = 0000_0110          06 = 0000_0110
–                       –
03 = 0000_0011          07 = 0000_0111
=                       =
03 = 0000_0011          -1 = 1111_1111
                        –          0110      Результат не вірний. Має бути 16-7=9
                        =     1111_1001      Корекція – відняти від молодшої тетради 6
```

Для корекції результату віднімання існує спеціальна команда:

`aas` (ASCII adjust for substraction) – корекція результату віднімання

Команда не має операндів, а працює з регістром `al`, аналізуючи його молодшу тетраду: якщо значення не більше 9 то прапор `cf` скидається в 0, інакше команда `aas` виконує наступні дії:

- із вмісту молодшої тетради регістра `al` віднімається 6;
- обнуляється старша тетрада регістра `al`;
- встановлюється прапор `cf` в 1, тим самим фіксуючи уявну позику з старшого розряду.

Команда `aas` використовується разом з основними командами віднімання `sub, sbb`.

Приклад:

```
a db 5
b db 7
sub db 0,0
…
mov al,[a]
sbb al,[b]
aas
```

Множення і ділення незапакованих BCD-чисел.

Процесор має команди множення і ділення тільки для однорозрядних BCD-чисел. Для множення і ділення чисел довільної розрядності необхідно самостійно розробляти свій алгоритм, наприклад множення у “стовпчик”.

Для того, щоб перемножити два однорозрядні BCD-числа, необхідно:

- помістити один із співмножників у регістр `al`;
- помістити інший співмножник в регістр або пам’ять, виділивши байт;
- перемножити співмножники командою `mul`;
- результат (в `ax`) буде у двійковому коді, тому його потрібно скорегувати.

Для корекції результату застосовується спеціальна команда

`aam` (ASCII adjust for multiplication) – корекція результату множення.

Процес ділення двох незапакованих чисел дещо відрізняється від раніше розглянутих операцій. Тут також виконується корегування, але вони виконуються до основної операції ділення одного BCD-числа на інше. Попередньо у регістр `ax` поміщають дві незапаковані BCD-цифри діленого. Потім викликається команда `aad`:

`aad` (ASCII adjust for division) – корекція для ділення

Команда не має операндів і перетворює двозначне незапаковане BCD-число в регістрі `ax` в двійкове число. Це двійкове число буде використовуватися як ділене в операції ділення. Крім перетворення, команда `aad` поміщає отримане двійкове число у регістр `al`. Ділене, звичайно буде двійковим числом з діапазону 0…99. Алгоритм роботи команди `aad`:

- помножити старшу цифру BCD-числа в `ax` (вміст `ah`) на 10;
- додати `ah` + `al`, а результат (двійкове число) помістити в `al`;
- обнулити вміст `ah`.

Після виконання команди `aad` потрібно виконати звичайну команду `div` для ділення вмісту `ax` на одну BCD-цифру, яка знаходиться у байтовому регістрі або байтовій комірці пам’яті.

Додавання і віднімання запакованих BCD-чисел.

Запаковані BCD-числа використовуються рідше, порівняно з іншими формами подання чисел, тому їх можна розглянути коротко. Приклади, які показують необхідність корекції при додаванні і відніманні запакованих BCD-чисел.

67 = 0110_0111	67 = 0110_0111
+	+
75 = 0111_0101	-75 = 1011_0101
=	=
142 = 1101_1100 =>220	-8 = 0001_1100 => 28

Як видно, при додаванні у двійковому виді результат дорівнює 1101_1100, тобто 220_{10}, що невірно. В дійсності результат мав би бути 0001_0100_0010 в BCD-поданні або 142_{10}. Для корегування результату додавання є спеціальна команда:

`daa` (Decimal adjust for Addition) – корекція результату додавання

Команда `daa` перетворює вміст регістра `al` у дві запаковані десяткові цифри за спеціальним алгоритмом. Отримана в результаті одиниця (якщо результат більший 99) запам'ятовується у прапорі `cf`, тим самим враховується перенесення у старший розряд.

Як видно з прикладу, при відніманні у двійковому виді результат дорівнює 28 і є невірним. У двійково-десятковому коді результат мав би бути 0000_1000 або 8_{10}. При відніманні BCD-чисел програміст повинен контролювати знак за станом прапора `cf`, який фіксує позику із старших розрядів. Для віднімання BCD-чисел використовуються звичайні команди віднімання `sub` або `sbb`. Корегують результат командою `das`:

`das` (Decimal adjust for substraction) – корекція результату віднімання

Команда `das` корегує результат також за спеціальним алгоритмом.

3.4. Інші команди з арифметичним принципом

Команди інвертування бітів і зміни знаку.

Команда `not операнд` *інвертує біти* операнда (операція "доповнення до 1"). Команда `neg операнд` *змінює знак* операнда (операція "доповнення до 2" аналогічна до операції "унарний мінус"). При цьому не встановлюється знаковий біт, а інвертуються всі біти операнда і до результату інверсії додається біт у самий молодший розряд. Команду `neg` застосовується до знакових чисел, наприклад:

```
neg al     ; зміна знаку значення в регістрах
neg dx
neg ecx
neg byte [bx]   ; зміна знаку значення у комірці пам'яті [bx] довжиною byte
neg word [di]   ; зміна знаку значення у комірці пам'яті [di] довжиною word
neg dword [eax] ; зміна знаку значення у комірці пам'яті [eax] довжиною dword
neg qword [qax] ; зміна знаку значення у комірці пам'яті [qax] довжиною qword
```

Фрагмент програми з результатом виконання команд `neg` і `not`:

```
mov cx,0
mov cl,   0000_0101b   ;   5
;not cx ; 1111_1010b -> 250
neg cx ;  1111_1011b -> 251 -> -5
```

Сума числа і його "доповнення до 2" дає нуль:

```
mov rax,42
```

```
neg rax
add rax,42
```

В асемблері x86 знакові числа зберігаються у формі “доповнення до 2”, яке задає віддаль числа від 0 в обох напрямках, як позитивному, так і негативному.

```
0xffffffff (-1), 0x00000001 (1),
0xfffffffe (-2), 0x00000010 (2),
0xfffffffd (-3), 0x00000011 (3),
…
0x10000010(-126), 0x01111110 (126).
0x10000001(-127), 0x01111111 (127).
0x10000000 (-128)
```

Таблиця 3 – Діапазони знакових чисел

Розмір	Найбільше негативне		Найбільше позитивне	
	десяткове	шістнадцяткове	десяткове	шістнадцяткове
8	-128	80h	127	7Fh
16	-32768	8000h	32767	7FFFh
32	−2147483648	8000 0000h	2147483647	7FFF FFFFh
64	-92233720368547 75808	8000 0000 0000 0000h	9223372036854775 807	7FFF FFFF FFFF FFFFh

Команда переставлення байтів старшої і молодшої половин регістра.

Команда `bswap` переставляє байти 32- і 64-бітових регістрів:

```
bswap reg32
bswap reg64
```

Приклад:

```
mov cx,0x1234
bswap ecx      ; 0x3412_0000
```

Команда порівняння.

Команда порівняння `cmp` (від слова “compare” – “порівняти”) здійснює такі ж обчислення, як і команда `sub`, але результат нікуди не записує. Команда застосовується тільки для встановлення прапорів, а за нею звичайно слідує команда умовного переходу.

Команда порівняння з обміном.

Команда `cmpxchg` – атомарна інструкція, яка порівнює значення в пам’яті з одним аргументом, і у випадку успіху записує другий аргумент у пам’ять. Команди призначена для синхронізації паралельних потоків виконання.

```
; блокування
wait:
mov eax,-1
mov ecx,5              ; номер процесора
cmpxchg [mem],ecx      ; порівняння з вмістом [mem]=[0x105BA9D2]
jnz wait:              ; якщо ресурс заблокований
; зняття блокування, робота із спільним ресурсом
```

Команда обміну місцями операндів і додавання.

Команда `xadd операнд1, операнд2` обмінює вміст операндів, а потім виконує їх додавання і результат записує в `операнд1`. Команда може поєднуватися з lock префіксом і виконуватися *атомарно*.

4. Класифікація логічних команд і операцій

До засобів процесора для організації роботи з даними за правилами формальної логіки відносяться наступні команди:

Логічні команди:

- логічні побітові (`and, or, xor, not, test`);
- обробки бітів:
  - сканування бітів `(bsf, bsr)`;
  - перевірки і модифікації бітів `(bt, btc, btr, bts)`;
- зсуву:
  - зсув лінійний: логічний (`shl, shr, shld, shrd`) та арифметичний (`sar, sal`);
  - зсув циклічний (`rcl, rcr, rol, ror`);

Над операндами можуть виконуватися бітові операції:

- бітові `( &` (and), `|` (or), `^` (xor), `!` (not) `)`;
- зсуву `(shr, shl)`.

4.1 Логічні побітові команди

Логічні побітові команди виконуються над бітами з використанням наступних логічних операцій:

- заперечення (логічне НЕ, `not`), яке характеризується таблицею істинності:

Значення операнда	0	1
Результат операції	1	0

```
not bx
not rdx
not dword [dNum]
not qword [qNum]
```

- Логічне додавання (логічне АБО, `or`), яке характеризується таблицею істинності:

Значення операнда1	0	1	1	1
Значення операнда2	0	1	0	1
Результат операції	0	1	1	1

```
or ax, bx
or rcx, rdx
or eax, dword [dNum]
or qword [qNum], rdx
```

- Логічне множення (логічне І, `and`), яке характеризується таблицею істинності:

Значення операнда1	0	0	1	1
Значення операнда2	0	1	0	1

Результат операції	0	0	0	1

```
and ax, bx
and rcx, rdx
and eax, dword [dNum]
and qword [qNum], rdx
```

• Логічне виключальне додавання (логічне виключальне АБО, `xor`), яке характеризується таблицею істинності:

Значення операнда1	0	0	1	1
Значення операнда2	0	1	0	1
Результат операції	0	1	1	0

```
xor ax, bx
xor rcx, rdx
xor eax, dword [dNum]
xor qword [qNum], rdx
```

Синтаксис логічних команд:

and `операнд1, операнд2` – команда логічного множення виконує порозрядну операцію І над бітами операндів `операнд1, операнд2`. Результат записується в `операнд1`.

```
and ax, bx
and rcx, rdx
and eax, dword [dNum]
and qword [qNum], rdx
```

Команда дозволяє скинути вказані біти в 0. Для цього потрібно взяти бітову маску в якій 0 стоять в тих розрядках, які потрібно скинути, а у всіх інших розрядах стоять 1 і потім застосувати команду `and`. Початковий стан регістрів невідомий.

```
and rax,0ffff_ffffh  ; скинути в 0 старший біт
and al,1010_1010b    ; скинути в 0 всі парні біти
```

or `операнд1, операнд2` – команд логічного додавання виконує порозрядну операцію АБО над бітами операндів `операнд1, операнд2`. Результат записується в `операнд1`.

```
or ax, bx
or rcx, rdx
or eax, dword [dNum]
or qword [qNum], rdx
```

Команда дозволяє встановити вказані біти в 1. Для цього потрібно взяти бітову маску в якій 1 стоять в тих розрядках, які потрібно встановити і потім застосувати команду `or`. Початковий стан регістрів невідомий.

```
or rax,10b       ; встановити 1-й біт в регістрі rax
or al,0101_0101b ; встановити всі парні біти, а всі інші залишаться без змін
```

xor `операнд1, операнд2` – команд логічного виключального додавання виконує порозрядну операцію XOR над бітами операндів `операнд1, операнд2`. Результат записується в `операнд1`.

```
xor ax, bx
xor rcx, rdx
xor eax, dword [dNum]
```

```
xor qword [qNum], rdx
```

Команда дозволяє інвертувати вказані біти. Для цього потрібно взяти бітову маску в якій 1 стоять в тих розрядках, які потрібно інвертувати, а у всіх інших розрядах стоять 0 і потім застосувати команду `xor`. Початковий стан регістрів невідомий.

```
xor rax,10b        ; інвертувати 1-й біт в регістрі rax
xor al,1000_0001b  ; інвертувати 0-й і 7-й біти в регістрі el
```

Команду `xor` часто використовують для швидкого обнулення регістра, так як вона коротша від команди `mov`:

```
xor rax, rax  ;  розмір команди 3 байти
mov rax, 0    ;  розмір команди 5 байт
```

Коли застосовувати команду `xor` замість `mov`? Команда `xor` коротша, а значить, займає менше місця в процесорному кеші, менше часу тратиться на декодування. Але команда `xor` встановлює прапори. Тому, якщо потрібно зберегти стан прапорів, використовується команда `mov`.

Іноді для обнулення регістра застосовують команду `sub`. Вона також встановлює прапори.

```
sub rax,rax  ; тепер eax == 0
```

Якщо застосувати команду `xor` двічі, то отримується початкове значення. Таке поведінка команди `xor` використовується для простого шифрування: до кожного байту даних застосовується команда `xor` з постійною маскою (ключем), а для дешифрування той же ключ застосовується до шифрованих даних.

Команди `and`, `or` і `xor` подібні до інструкцій мови програмування C `&`, `|`, `^`. Всі ці команди встановлюють прапори `zf, cf і pf` у відповідності з результатом.

`test` `операнд1, операнд2` – команда ”перевірити” виконує порозрядну операцію AND над бітами операндів `операнд1, операнд2`. Стани операндів залишаються без змін, а змінюються тільки прапори `zf, sf, pf`.

Команда `test` називається також командою логічного порівняння, так як дозволяє перевірити, чи встановлені задані біти. Команда виконує побітове `&` над операндами, як і команда `and`, але результат нікуди не записує, а виставляє тільки прапори, наприклад:

```
test al,0b0000_1000   ; чи встановлений 3-й (від нуля) біт?
je not_set
   ; потрібні біти встановлені
not_set:
      /* біти не встановлені */
```

Командою `test` можна порівнювати значення регістра з нулем:

```
test rax,rax
je is_zero
   ; rax != 0 */
is_zero:
   ; rax == 0
```

Для порівняння операнда з нулем рекомендується використовувати команду `test rax,rax` замість команди `cmp rax,0`

`not` `операнд` – команда логічного заперечення `операнд` виконує порозрядне інвертування бітів операнда. Результат записується в `операнд`.

```
not rax   ; інвертувати біти регістра rax
```

4.2. Бітові операції над операндами команд

Над операндами команд можна виконувати наступні операції:

- арифметичні операції **+** (додавання), **-** (віднімання), ***** (множення), **/** (ділення беззнакове), **//** (ділення знакове), **%** (ділення за модулем беззнакове), **%%** (ділення за модулем знакове);

- унарні оператори + (плюс), **-** (мінус), ~ (інверсія, доповнення до 1), **!** (логічне заперечення), **SEG** (отримання адреси сегменту);

- бітові операції **&** (and), **|** (or), **^** (xor);

- операції зсуву **<<** (вправо), **>>** (вліво), і операції обчислення поточних адрес **$**, **$$**.

Операнди повинні бути абсолютними, тобто такими, числові значення яких можуть бути обчислені транслятором.

Синтаксис бітових операцій над виразами

```
[+ | - | ! | seg] Вираз1 [… & | | | ^ …] [[+ | - | ! | seg] Вираз2
```

Обчислення адреси сегмент:позначка:

```
mov ax,seg symbol
mov es,ax
mov bx,symbol ;   es:bx -> seg::symbol
```

Бітова операції ^ (xor):

```
mask equ 1000_0011b
     mov al,mask^01h   ; переслати в регістр al маску
                       ; з інвертованим правим бітом al=1000_0010
```

Операція зсуву:

```
mov ax, 2<<5 ; зсунути число 2 на 5 розрядів вліво і переслати число 64 в ax
```

4.3. Обчислення поточних адрес $, $$

Асемблер обчислює поточну адресу виразу з операндом **$**:

```
section .data
  var db 'HELLO WORD',10
  len equ $-var    ; різниця адрес поточної команди і позначки var
```

Так нескінченний цикл можна записати як `jump $`.

Асемблер обчислює початок поточної секції операндом **$$**. Так вираз `$-$$` визначає зміщення поточної команди від початку поточного сегменту.

4.4. Сканування, перевірка і модифікація бітів

Для сканування бітів призначені команди `bsf (bit scaning forward)` і `bsr (bit scaning reverse)`. Команди дозволяють знайти перший встановлений в 1 біт операнда. Пошук можна вести як спочатку, так і з кінця.

```
bsf операнд1, операнд2 – сканування бітів уперед
```

Команда bsf продивляється біти операнда2 починаючи з старших розрядів в пошуку першого біта встановленого в 1. В операнд1 заноситься номер цього біту (відносно біту 0). Якщо всі біти операнда2 дорівнюють 0, то прапор zf встановлюється в 1, інакше в 0.

```
bsr операнд1, операнд2 – сканування бітів у зворотному порядку
```

Команда bsr продивляється біти операнда2 починаючи з молодших розрядів в пошуку першого біта встановленого в 1. В операнд1 заноситься номер цього біту. Якщо всі біти операнда2 дорівнюють 0, то прапор zf встановлюється в 1, інакше в 0.

Для перевірки і модифікації бітів використовуються команди bt, bts, btr, btc.

Команда bt (bit test) переносить значення біта регістра у прапор cf.

```
bt операнд, зміщення_біта
bt ax,5   ; перевірити значення біта 5
jnc m1    ; перехід, якщо біт = 0
```

Команда bts (bit test and set) переносить значення біта регістра у прапор cf і потім встановлює перевірюваний біт регістра в 1.

```
bts операнд, зміщення_біта
btc ax,10 ; перевірити і встановити 10-й біт
jc m1     ; перехід, якщо перевірюваний біт = 0
```

Команда btr (bit test and reset) переносить значення біта регістра у прапор cf і потім встановлює перевірюваний біт регістра в 0.

Команда btc (bit test and convert) переносить значення біта регістра у прапор cf і потім інвертує значення цього біту регістра.

4.5. Команди зсуву

Часто приходиться виконувати операції побітового зсуву. Операції побітового зсуву виконують командами зсуву, які показані на рис. 5.1. Всі команди зсуву переміщають біти в полі операнда вліво або вправо в залежності від коду операції. Всі команди зсуву мають однакову структуру:

```
код операнд, лічильник_зсувів
```

Кількість зсовуваних розрядів, лічильник_зсувів, може задаватися двома способами:

• статично, фіксованим значенням у безпосередньому операнді;

• динамічно, занесенням значення лічильник_зсувів у регістр cl, перед виконанням команди зсуву.

За принципом дії команди зсуву можна розділити на два типи:

• лінійного зсуву;

• команди циклічного зсуву.

4.6. Лінійний зсув

До команд цього типу відносяться команди, які виконують зсув за наступним алгоритмом:

• черговий “висовуваний” біт встановлює прапор cf;

• біт, який вводиться в операнд з іншого кінця, має значення 0;

• при зсуві чергового біту він переходить у прапор `cf`, при цьому значення попереднього зсунутого біту *втрачається*.

Команди *лінійного зсуву* поділяються на:

• команди *логічного зсуву*;

• команди *арифметичного зсуву*.

До команд логічного лінійного зсуву відносяться:

`shl операнд, лічильник_зсувів (shift logical left)`- логічний зсув вліво. Вміст операнда зсовується вліво на кількість бітів, яку задає `лічильник_зсувів`. В молодший розряд записуються 0.

```
shl ax, 8
shl rcx, 32
shl eax, cl
shl qword [qNum], cl
```

`shr операнд, кількість_зсувів (shift logical right)` – логічний зсув вправо. Вміст операнда зсовується вправо на кількість бітів, яку задає `лічильник_зсувів`. В старший розряд записуються 0.

```
shr ax, 8
shr rcx, 32
shr eax, cl
shr qword [qNum], cl
```

Перший операнд може бути регістровим або типу “пам’ять”. Другий операнд може бути безпосереднім числом від 0 до 31 або регістром `cl`, в якому враховуються тільки молодші 5 біт (інші регістри використовувати не можна).

В прапорі `cf` зберігається самий останній “зсунутий” біт. Це вповні допустимо для роботи з беззнаковими числами, але числа із знаком будуть оброблені невірно, так як знаковий біт може бути втрачений. Для беззнакових чисел зсув на `n` біт вліво еквівалентний множенню на 2^n, а зсув вправо – цілочисельному ділення на 2^n із відкиданням залишку.

Для роботи з знаковими числами використовуються команди *арифметичного* лінійного зсуву:

`sal операнд, лічильник_зсувів (shift arithmetic left)` – арифметичний зсув вліво. Вміст операнда зсовується вліво на кількість бітів, яку задає `лічильник_зсувів`. В молодший розряд записуються 0. Команда не зберігає знак, але встановлює прапор `cf` у випадку зміни знаку черговим зсовуваним бітом.

```
sal ax, 8
sal rcx, 32
sal eax, cl
sal qword [qNum], cl
```

`sar операнд, лічильник_зсувів (shift arithmetic rigth)` – арифметичний зсув вправо. Вміст операнда зсовується вправо на кількість бітів, яку задає `лічильник_зсувів`. В старший розряд записуються 0. Команда зберігає знак, відновлюючи його після зсуву кожного чергового біту.

```
sar ax, 8
sar rcx, 32
sar eax, cl
```

```
sar qword [qNum], cl
```

Команди арифметичного зсуву дозволяють виконати множення і ділення операнда на 2^n.

4.7. Циклічний зсув

До команд циклічних зсувів відносяться команди, які зберігають значення зсовуваних бітів. Є два типи команд циклічного зсуву:

• команди простого циклічного зсуву;

• команди циклічного зсуву через прапор перенесення cf;

До команд простого циклічного зсуву відносяться:

`rol операнд, лічильник_зсувів (rotate left)` – циклічний зсув вліво. Вміст операнда зсовується вліво на на кількість бітів, яку задає `лічильник_зсувів`. Зсовувані вліво біти записуються в той же операнд справа і *одночасно* заносяться у прапор `cf`.

```
rol ax, 8
rol rcx, 32
rol eax, cl
rol qword [qNum], cl
```

`ror операнд, лічильник_зсувів (rotate right)` – циклічний зсув вправо. Вміст операнда зсовується вправо на кількість бітів, яку задає `лічильник_зсувів`. Зсовувані вправо біти записуються в той же операнд зліва і *одночасно* заносяться у прапор `cf`.

```
ror ax, 8
ror rcx, 32
ror eax, cl
ror qword [qNum], cl
```

Команди циклічного зсуву через прапор перенесення `cf` відрізняються від команд простого циклічного зсуву тим, що зсовуваний біт не зразу попадає в операнд з іншого кінця, а записується спочатку у прапор перенесення `cf`. Ці команди використовують прапор `cf` як продовження операнда.

`rcl операнд, лічильник_зсувів (rotate through carry left)` – циклічний зсув вліво через прапор перенесення. Вміст операнда зсовується вліво на кількість бітів, яку задає `лічильник_зсувів`. Зсовувані вліво біти записуються спочатку у прапор `cf`, а потім в той же операнд справа.

`rcr операнд, лічильник_зсувів (rotate right)` – циклічний зсув вправо через прапор перенесення. Вміст операнда зсовується вліво на кількість бітів, яку задає `лічильник_зсувів`. Зсовувані вліво біти записуються спочатку у прапор `cf`, а потім у той же операнд зліва.

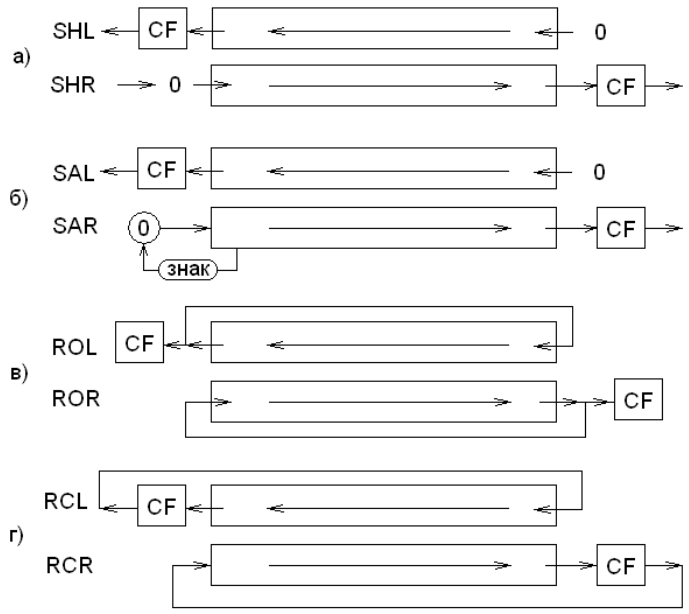


Рисунок 2 – Команди зсуву: а) логічного лінійного; б) арифметичного лінійного; в) простого циклічного; г) циклічного через прапор CF

Система команд останніх моделей процесорів Intel містить додаткові команди зсуву подвійної точності:

`shld операнд1, операнд2, лічильник_зсувів` – зсув вліво подвійної точності

Команда `shld` зсовує біти `операнда1` вліво, заповнюючи його біти справа зсовуваними вліво бітами з `операнда2`. Кількість зсовуваних бітів задається значенням `лічильник_зсувів`, яке може бути в діапазоні 0…31. Це значення може задаватися безпосередньо операндом або міститися в регістрі `cl`. Значення `операнд_2` при зсувах не змінюється.

`shrd операнд_1, операнд_2, лічильник_зсувів`

Аналогічно працює і команда `shrd`, але `операнд1 і операнд2` зсовується вправо.

5. Ланцюгові команди

Ланцюгові команди призначені для роботи з послідовностями байтів в пам’яті. Команди дозволяють виконувати дії над блоками пам’яті, які складаються з послідовностей елементів наступних розмірів: 8-, 16-, 32- і 64 біти. Вміст цих блоків не має ніякого значення, це можуть бути числа, символи. Ланцюгові команди використовують регістри `rsi/esi/si` і `rdi/edi/di`.

Загальна ідея ланцюгових команд полягає у тому, що читання з пам’яті здійснюється за адресою з регістра `rsi/esi/si`, а записування – за адресою з регістра `rdi/edi/di`, а потім значення цих регістрів одночасно збільшуються (або зменшуються) в залежності від розмірів операндів команди на 1, 2, 4, 8. Напрямок зміни адрес визначається прапором напрямку `df`. Встановити прапор напрямку `df` можна командою `std`, а скинути – командою `cld`.

Всі ланцюгові команди можна розділити на групи.

Команди пересилання. Формат команд:

`movsb, movsw, movsd, movsq` – копіювання ланцюжка байтів, слів, подвійних слів, чотирних слів, які адресуються регістрами `rsi/esi/si` і `rdi/edi/di, [edi]=[esi]`

При використанні формату **movs адреса_приймача,адреса_джерела** асемблер сам визначає за типом операндів, яку з трьох форм команд вибрати. Після виконання команди вміст регістрів rsi/esi/si і rdi/edi/di збільшується (прапор df=1) або зменшується (прапор df=0) на розмір елемента ланцюжка.

Команди порівняння. Формат команд:

smpsb, cmpsw, cmpsd, cmpsq – команда порівнює елементи, які адресуються регістрами rsi/esi/si і rdi/edi/di. Після виконання команди вміст регістрів rsi/esi/si і rdi/edi/di збільшується (прапор df=1) або зменшується (прапор df=0) на розмір елемента ланцюжка.

При використанні формату **cmps адреса_приймача,адреса_джерела** асемблер сам визначає за типом операндів, яку з трьох форм команд вибрати.

Команди сканування. Формат команд:

scasb, scasw, scasd, scasq – команда порівнює елементи, які адресуються регістром rdi/esi/si з одним з регістрів rax/eax/ax/al в залежності від команди. Після виконання команди вміст регістру rsi/esi/si збільшується (прапор df=1) або зменшується (прапор df=0) на розмір елемента ланцюжка.

При використанні формату **scas адреса_приймача,адреса_джерела** асемблер сам визначає за типом операндів, яку з трьох форм команд вибрати.

Команди читання із ланцюжка. Формат команд:

loadsb, loadsw, loadsd, loadsq – команда копіює елементи, які адресуються регістром rsi/esi/si в регістри rax/eax/ax/al в залежності від команди. Після виконання команди вміст регістру esi збільшується (прапор df=1) або зменшується (прапор df=0) на розмір елемента ланцюжка.

При використанні формату **loads адреса_приймача,адреса_джерела** асемблер сам визначає за типом операндів, яка з трьох форм команд вибрати.

Команди запису в ланцюжок. Формат команд:

stosb, stosw, stosd, stosq – команда копіює в пам'ять, яка адресуються регістром rdi/edi/di елементи з rax/eax/ax/al в залежності від команди. Після виконання команди вміст регістра rdi/edi/di збільшується (прапор df=1) або зменшується (прапор df=0) на розмір елемента ланцюжка.

При використанні формату **stos адреса_приймача,адреса_джерела** асемблер сам визначає за типом операндів, яка з трьох форм команд вибрати.

Пара інструкцій **loads/stos** може бути замінена однією інструкцією **movs**.

Логічно до ланцюжкових команд відносяться так звані *префікси повторень*:

```
rep
repe/repz
repne/repnz
```

Префікси повторень вказуються перед ланцюжковою командою. Розміщення префікса rep перед ланцюжковою командою заставляє її виконуватися у циклі. Префікс repe задає ітерації до першого співпадіння елементів, а префікс repne – до першого не співпадіння елементів. Рішення про циклічне виконання ланцюжкової команди приймається на основі стану регістра rcx/ecx/cx (repe/repne) або прапора zf (repz/repnz).

Так команда rep stow заміняє блок команд:

```
Clear:  mov es:[di],ax ; копіювання ax у es:di
inc di                 ; збільшення індексу на 2 для word у буфері
dec cx                 ; зменшення cx на 1
jnz Clear              ; цикл поки cx не 0
```

Команди читання/запису стрічок з портів введення-виведення

`insb, insw, insd` – команди читають подвійне слово/слово/байт з порту I/O, адреса якого задається в `dx` і записують їх в [`es:edi/di`]. Потім (в залежності від прапора напрямку: збільшують, якщо прапор очищений або зменшують, якщо прапор встановлений) адресний регістр `edi/di` на 4/2/1.

`outsb, outsw, outsd` – команди завантажують подвійне слово/слово/байт з [`ds:esi/si`] і записують в порт I/O, адреса якого задається в `dx`. Потім (в залежності від прапора напрямку: збільшують, якщо прапор очищений або зменшують, якщо прапор встановлений) адресний регістр `esi/si` на 4/2/1.

Контрольні запитання.

1. Як структуруються машинні команди за групами.
2. Як класифікуються за функціональними ознаками цілочисельні команди.
3. Команди одно- і двонаправленого пересилання даних.
4. Команди роботи з портами.
5. Команди роботи з адресами і вказівниками.
6. Команди роботи зі стеком.
7. Арифметичні команди двійкової арифметики.
8. Арифметичні команди десяткової арифметики.
9. Інші команди з арифметичним принципом
10. Класифікація команд для роботи з логічними даними.
11. Логічні побітові команди і бітові операції.
12. Обчислення поточних адрес.
13. Обробка, перевірка і модифікація бітів.
14. Команди логічного, арифметичного і циклічного зсуву.
15. Ланцюгові команди.

6. КОМАНДИ ПЕРЕХОДІВ. СТЕК

Мета. Вивчення групи команд передачі керування і організації стеку

Вступ. Поряд із засобами арифметичних обчислень і логічних перетворень, система команд мікропроцесора має засоби для передачі керування. Засоби передачі керування дозволяють здійснювати як умовні, так і безумовні переходи, а також викликати підпрограми. Для передачі параметрів підпрограмам і збереження локальних змінних використовується стек.

План.

1. Засоби передачі керування
2. Умовні переходи
   2.1 Прості умовні переходи
   2.2. Переходи за результатами порівнянь
   2.3. Команди організації циклів
3. Безумовні переходи
4. Стек
5. Виклики підпрограм

1. Засоби передачі керування

Команди арифметичних та логічних перетворень формують лінійні ділянки програми. Звичайно в програмах є місця де треба прийняти рішення про те, яка команда буде виконуватися наступною. Це рішення може бути:

• *безумовним* – передається керування не наступній команді, а іншій, яка знаходиться на деякій віддалені від поточної;

• *умовним* – передається керування не наступній команді, а іншій на основі аналізу деяких умов або даних.

Те, яка команда буде виконуватися наступною, визначається вмістом пари регістрів `cs:rip/eip`, де `cs` – сегментний регістр коду, `rip/eip` – регістр вказівник команд.

За принципом дії команди переходів можна розділити на три групи:

1. *Команди керування циклом*:
   • команди організації циклу `loop` з лічильником `rcx/ecx`;
   • команди організації циклу `loope/loopz, loopne/loopnz` з лічильником `rcx/ecx` і можливістю термінового виходу з циклу за додатковою умовою.
2. *Команди умовної передачі керування*:
   • команди переходу `jxx` за результатами команди порівняння `cmp`;
   • команда переходу за станом визначеного прапора регістра `rflags/eflags`;
   • команди переходу за вмістом регістра `rcx/ecx`.
3. *Команди безумовної передачі керування*:
   • команда безумовного переходу (`jmp`);
   • виклик процедури і повернення з процедури (`call, ret`);
   • виклик програмних переривань і повернення з програмних переривань.

Для позначення місця, куди необхідно передати керування, використовується позначка. *Позначка* – це символічне ім'я, яке задає адресу комірку пам'яті і яке використовується як операнд в командах передачі керування. Позначка визначається *двокрапкою* після символічного імені (`label:`).

Позначці присвоюється три атрибути:

• ім'я сегмента коду, де ця позначка описана;

• зміщення – віддаль у байтах від початку сегменту, в якому описана позначка;

• тип позначки або атрибут віддалі.

2. Умовні переходи

2.1. Прості умовні переходи

Прості команди умовного переходу забезпечують перехід за вказаною адресою у випадку, якщо один з прапорів встановлений (дорівнює одиниці) або скинутий (дорівнює нулю). Імена цих команд утворюються з букви `j` (від слова "`jump`"), першої букви назви прапора (наприклад, `z` для прапора `zf=1`) і, можливо, вставленої між ними букви `n` (від слова "`not`"), якщо перехід необхідно здійснити за умови `zf=0`.

Такі команди умовного переходу ставлять зразу після арифметичних операцій або команди `cmp`, наприклад

```
    cmp rax,rbx    ; порівняти значення регістрів
    jz are_equal   ; zf==1, якщо операнди однакові перейти на позначку are_equal
    …
    cmp rax,rbx    ; порівняти значення регістрів
    jnz not_equal   ; zf==0, якщо операнди не однакові перейти на позначку
not_equal
```

Таблиця 1. Команди умовних переходів за значеннями прапорів арифметичних операцій Нуль (zero), Перенесення (carry), Переповнення (overflow), Знак (sign), Парність (parity)

Команда	Умова переходу	
jz, je	Перехід якщо 0, перехід якщо операнди рівні	zf=1
jnz, jne	Перехід якщо не 0, перехід якщо операнди не рівні	zf=0
jc	Перехід якщо Перенесення	cf=1
jnc	Перехід якщо не Перенесення	cf=0
jo	Перехід якщо Переповнення	of=1
jno	Перехід якщо не Переповнення	of=0
js	Перехід якщо Знак	sf=1
jns	Перехід якщо не Знак (позитивне або нуль)	sf=0
jp, jpe	Перехід якщо Парність, перехід якщо парне (even)	pf=1
jnp, jpo	Перехід якщо Непарність, перехід якщо непарне (odd)	pf=0

2.2. Переходи за результатами порівняння команди cmp

За результатами команди порівняння `cmp` можуть встановлюватися два прапори, наприклад `sf=1, of=0` (число отримане негативне, переповнення не було) або `sf=0, of=1` (число позитивне, але це результат переповнення, а в дійсності результат негативний). Тобто, потрібно

аналізувати ситуації коли `sf≠of`. Для цього використовуються команди умовного переходу за результатами порівнянь `cmp a,b`.

Таблиця 2. Переходи за результатами порівнянь `cmp`

Команда	Перехід якщо	Вираз	Умова переходу	Синонім
нерівності для порівняння знакових операндів				
jl jnge	less not greater or equal	a<b	(sf xor of)=1 (sf≠of)	
jle jng	less or equal not greater	a≤b	((sf xor of) or zf) =1 (sf ≠ of) or zf = 1	
jg jnle	greater not less or equal	a>b	((sf xor of) or zf) =0 zf=0 and sf=of	
jge jnl	greater or equal not less	a≥b	(sf xor of) = 0 sf=of	
нерівності для порівняння беззнакових операндів				
jb jnae	below not above or equal	a<b	cf=1	jc
jbe jna	below or equal not above	a≤b	cf=1 or zf=1	
ja jnbe	above not below or equal	a>b	cf=0 and zf=0	
jae jnb	above or equal not below	a≥b	cf=0	jnc

Приклад використання команд:

```
.text
    mov rax,15
    cmp rax,15      ; порівняння
    jne not_equal   ; якщо операнди не рівні, перейти на позначку not_equal
                    ; команди для rax = 15
    jmp out
 not_equal:
                    ; команди для rax != 15
 out:
```

Крім команд переходів за результатами порівнянь `j`**`cc`**, існує родина команд `set`**`cc`**. Вони перевіряють стан прапорів так як `j`**`cc`**. За значенням прапорів операнд команди встановлюється в 1, якщо перевірювана умова `cc` істинна, і в 0, якщо умова фальшива. Команди `set`**`cc`** працюють тільки з операндами, які зберігаються в пам'яті і мають розмір один байт. Синтаксис команд `set`**`cc`**`:`

`set`**`cc`** `<операнд>.`

Приклади команд `set`**`cc`**:

```
   cmp   [a],5
         seta  dl      ; dl=1, [a] > 5 (cf=0 i zf=0)
         cmp   [b],10
         setae dl      ; dl=1, [b]>= 10 (cf=0)
         cmp   [c],0
         sete  dl      ; dl=1, [c]=0 (cf=1)
         cmp   [c],0
```

```
        setge dl      ; dl=1, [c]>=0 (zf=0, cf=0)
```

2.3. Команди організації циклів

Команда `loop` призначена для організації циклів з наперед відомою кількістю ітерацій.

Синтаксис команди `loop`:

```
позначка:
..
loop  позначка
```

Принцип роботи:

- зменшити значення регістра `rcx/ecx/cx` на 1;
- якщо `rcx/ecx/cx` = 0, передати керування наступній команді за позначкою `loop`;
- якщо `rcx/ecx/cx` ≠ 0, передати керування на початок циклу на *позначка*.

Як лічильник ітерацій команда `loop` використовує регістр `rcx/ecx/cx`, в який перед початком циклу записується потрібне число ітерацій. Сама команда `loop` виконує дві дії: зменшує на одиницю значення в регістрі `rcx/ecx/cx` і, якщо результат не нульовий, переходить на задану позначку початку циклу. Команда `loop` здійснює тільки короткі переходи на позначки, які розміщені від самої команди не далі як на 128 байти. Приклад підрахунку суми елементів масиву із 10 подвійних слів із проходженням вектора з початку:

```
segment .data
   array dd 1,2,3,4,5,6,7,8,9,10
segment .text
   mov ecx, 10     ; кількість ітерацій
   mov esi, array  ; адреса першого елементу
   mov eax, 0      ; початкове значення суми
lp: add eax, [esi] ; додати число до суми
   add esi, 4      ; адреса наступного елементу
    loop lp         ; зменшення лічильника ecx і якщо ecx не 0, то перехід на
lp
```

Замість команди `loop` можна використати дві команди `dec` і `jnz`:

```
segment .data
   array dd 1,2,3,4,5,6,7,8,9,10
segment .text
   mov ecx, 10     ; кількість ітерацій
   mov esi, array  ; адреса першого елементу
   mov eax, 0      ; початкове значення суми
lp: add eax, [esi] ; додати число до суми
   add esi, 4      ; адреса наступного елементу
   dec ecx         ; зменшення лічильника і якщо ecx не 0, перехід на lp
   jnz lp
```

У прикладі підрахунку суми можна також обійтися без регістра `esi`, але у цьому випадку проходження масиву здійснюється з кінця:

```
mov ecx,10
mov eax,0
lp: add eax,[array+4*ecx-4]
loop lp
```

Команда `loop` має дві модифікації.

Команда `loope` (синонім `loopz`) здійснює перехід, якщо регістр `rcx/ecx/cx != 0` і прапор `zf=1`.

Команда `loopne` (синонім `loopnz`) здійснює перехід, якщо регістр `rcx/ecx/cx !=0` і прапор `zf=0`.

Є дві додаткові команди умовного короткого переходу, які аналізують стан регістра `rcx/ecx/cx`:

`jcxz` - умовний перехід, якщо регістр `cx=0`.

`jrcxz/jecxz` - умовний перехід, якщо регістр `rcx/ecx=0`.

Команди не враховують прапори. Вони використовуються для запобігання виконання циклу у випадку, коли значення регістра `rcx/ecx/cx` є нульовим на початку циклу. Якщо на момент входу у цикл `rcx/ecx/cx=0`, то виконується тіло циклу, а потім від лічильника віднімається 1. В результаті лічильник (переповнюється 0-1=FF…) отримає значення максимально можливого цілого числа 2^{32}. Для запобігання таких ситуацій перед циклом потрібно поставити команду `jrcxz/jecxz/jcxz`:

```
; перевірка значення rcx на 0
jrcxz lpq
lp: ; тіло циклу
; ...
loop lp
lpq:
```

На асемблері можна створювати довільні цикли з передумовою, подібні до **`while(){}`** в мові програмування С. Фрагмент коду такого циклу на асемблері:

```
loop_start:                 /* початок циклу                        */
        cmp   ...           /* порівняння */
        je    loop_end      /* умова переходу для виходу з циклу    */
        /* тіло циклу */
        jmp   loop_start    /* перехід на перевірку умови циклу     */
loop_end:
```

Також на асемблері можна створювати довільні цикли з після умовою, подібні до **`do{} while()`** в мові програмування С. Фрагмент коду такого циклу на асемблері:

```
loop_start:                 /* початок циклу                        */
        /* тіло циклу */
        cmp   ...           /* порівняння */
        je     loop_end          /* команда умовного переходу для виходу з
циклу */
        jmp   loop_start    /* інакше повторити цикл спочатку       */
loop_end:
```

Реалізація циклу `do-while` на мові С і асемблера:

```
int pcount_do(unsigned x) {
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;      // x->edx, result->ecx
```

```
}

   mov ecx, 0    ;   result = 0
.L2:             ;loop:
   mov eax, edx
   and eax, 1    ;   t = x & 1
   add ecx, eax  ;   result += t
   shr edx, 1    ;   x >>= 1
   jne .L2       ;   if !0, goto loop
```

3. Безумовні переходи

Синтаксис команди безумовного переходу:

```
jmp [атрибут_віддалі] адреса_переходу (задається як позначка)
```

Приклади переходів:

```
jmp cycl    ; перехід на позначку cycl
jmp rax     ; перехід за адресою, яка міститься в регістрі rax
jmp [rax]   ; перехід за адресою, яка міститься в 64-бітовій комірці пам'яті, адреса комірки міститься в регістрі rax
```

При переходах всередині сегменту атрибут віддалі позначки може бути `short`, `near` або `far`. Переходи `short` можуть бути в межах -128÷127 (діапазон знакового 8-розрядного числа) відносно IP.

Переходи `near` можуть мати 2-байтове зміщення (-32768÷32767) в 16-розрядних сегментах в межах сегменту і 4-байтове зміщення (-2_147_483_648÷2_147_483_647) в 32-розрядних сегментах в межах вибраної області в CS селекторі.

Переходи `far` додатково використовують сегменти/селектори.

Команда переходу всередині сегменту виконує наступну дію:

```
rip/eip = rip/eip + зміщення
```

При міжсегментних переходах задають атрибут віддалі позначки `far`. У цьому випадку команда переходу виконує дію:

```
cs = адреса сегменту в який здійснюється перехід
rip/eip = rip/eip + зміщення
```

Переходи можуть бути прямими (додатне зміщення) і зворотними (від'ємне зміщення). Для зворотних переходів в одному сегменті атрибут віддалі можна не вказувати, так як асемблер автоматично визначає їх зміщення. Для прямих переходів атрибут віддалі потрібно явно задавати:

```
jmp short short_jump
mov ax,cx
short_jmp:
…
```

Ще одне важливе поняття, яке має відношення до позначок – *вказівник адрес команд* (*IP - instruction pointer*). Асемблер обробляє початкову програму – команда за командою. При цьому він використовує вказівник команд, збільшуючи його на величину кожної обробленої команди. Таким чином, кожна команда під час трансляції має адресу, яка дорівнює значенню вказівника адрес команд.

Транслятор асемблера забезпечує дві можливості для роботи з цим вказівником:

• використання позначок, атрибуту зміщення яких присвоюється значення вказівника адреси тієї команди, перед якою вони розміщені;

• використання спеціального символу `$` для позначення вказівника адреси команди. Цей символ у любому місці програми використовує числове значення вказівника адреси, наприклад:

```
.data
msg db ”Привіт світ”,0
len_mes = $-mes   ; довжина стрічки msg
```

3. Стек

Стек – це структура даних, в яку дані заносяться і видобуваються у зворотному порядку (останній зайшов, перший вийшов, Last-In, First-Out, LIFO).

Розміщення стеку в пам’яті показано на рис. 1.

Старші адреси	Стек (stack)
	...
	...
	Доступна пам’ять
	...
	...
	Купа (heap)
	Секція неініціалізованих даних (bss)
	Секція даних (data)
	Секція коду (text)
Молодші адреси	Резерв

Рисунок 1 – Структура оперативної пам’яті

У купі розміщуються динамічно виділені дані, наприклад оператор **`new`** у C++ і системний виклик `malloc()` у Сі. При розширенні області стеку і купи вони можуть перекритися. У цьому випадку робота програми завершується аварійно.

Для поміщення даних у стек використовується команда `push`, а для видобування — команда `pop`. Формати команд:

```
push <operand64>
pop  <operand64>
```

Операндами можуть бути регістри і комірки пам’яті з типом `qword`.

```
push rax                        pop rax
push qword [qVal] ; значення    pop qword [qVal]
push qVal ; адреса              pop rsi
```

Для роботи із стеком призначені два регістри `rsp` (вказівник верхівки стеку) і `rsp` (вказівник бази (основи) кадра стеку).

Команда `push` заносить вміст операнду (регістру або комірки пам’яті) в стек, при цьому вказівник стеку `rsp` зменшує своє значення на 8 байтів.

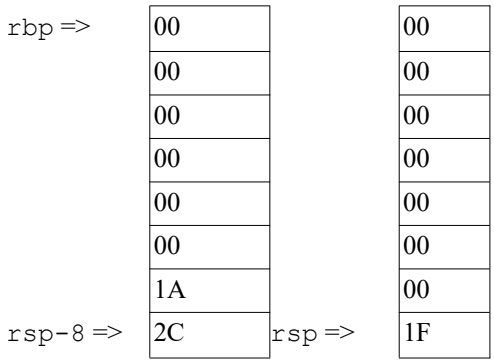
Команда `pop` видобуває значення з верхівки стеку і заносить в операнд (регістр або комірку пам'яті), при цьому вказівник стеку `rsp` збільшує своє на 8 байтів.

Команди `pop` і `push` можуть мати типи операндів (`word`, `dword`, `qword`), але не `byte`.

Не рекомендується поміщати у стек операнди менших розмірів (`word, dword`).

Приклад:

```
mov rax, 6700 ; 670010 = 00001A2C16
push rax
mov rax, 31   ; 3110 = 0000001F16
push rax
```



Якщо потрібно тільки звернутися до значення на верхівці стеку (без видобування його зі стеку) можна використати як операнд `[rsp]`:

```
mov rax,rsp     - читання адреси верхівки стеку
mov rax,[rsp]   - читання значення на верхівці стеку
mov rax,[rsp+8] - читання наступного значення нижче верхівки стеку
```

4. Виклики підпрограм

Синтаксис команди виклику процедури:

```
call [атрибут віддалі] ім'я_процедури
```

Команда `call` виконує наступні дії:

- зберігає у стеку адресу повернення (адреса команди, наступна після команди `call`);
- передає керування за адресою з символічним іменем `ім'я_процедури`.

Для повернення з підпрограми в основну програму використовуються команди:

```
ret
ret [число]
```

Команда `ret` у своїй найпростішій формі не має аргументів. Виконуючи цю команду, процесор видобуває адресу повернення з верхівки стеку і записує її в регістр `cs:rip/cs:eip`. в результаті чого керування передається в основну команду.

Команда `ret` з параметром зчитає адресу повернення із стеку і записує її в регістр `cs:rip/cs:eip`. Параметр `[число]` задає кількість байтів, які вилучаються зі стеку.

4.1. Організація стекового фрейму

Стек використовується при виклику підпрограм для розміщення аргументів виклику, значень повернення з підпрограм (функцій) і локальних змінних підпрограм (функцій).

Існують наступні методи передачі аргументів в підпрограму:

- через регістри;
- через загальну пам'ять;
- через стек.

При виклику підпрограми (в якій не використовуються локальні змінні) без параметрів фактично не використовується механізм стекових фреймів, у стеку зберігається тільки адреса повернення.

На практиці підпрограми рідко бувають такими простими. У більш складних випадках може використовуватися велике число параметрів та локальних змінних на які не вистачить регістрів. Крім того при передачі параметрів через загальну область пам'яті або регістри не можна використати рекурсію.

Тому звичайно у складних програмах (особливо при трансляції з мов програмування високого рівня) параметри передаються через стек і в стеку розміщуються локальні змінні.

Параметри для підпрограми (розміщує основна программа), адреса повернення (розміщує команда `call`) і локальні змінні (розміщує підпрограма) утворюють *стековий фрейм*. Як реперну точку, при зверненні до елементів стекового фрейму, можна використати адресу повернення. Якщо в стек занести три 4-байтові параметри, а потім викликати підпрограму, то адреса повернення буде в `[esp]`, а адреси параметрів будуть `[esp+4]`, `[esp+8]`, `[esp+12]`. Якщо в підпрограмі використовуються дві 4-розрядні локальні змінні, то їх адреси будуть `[esp-4]`, `[esp-8]`.

Так як регістр вказівник верхівки стеку `rsp/esp` може використовуватися *у підпрограмі* для *виклику інших підпрограм*, то його значення зберігають в іншому регістрі, переважно в `rbp/ebp` (регістрі вказівника бази). Тоді регістр `rbp/ebp` буде містити адресу повернення, а регістр `esp` буде використовуватися як вказівник верхівки стеку для даних підпрограми.

В *основній програмі* можуть викликатися *декілька підпрограм*, які також використовують регістр `rbp/ebp`. Для збереження значення `ebp` кожної підпрограми і, враховуючи те, що в програмі є значно більше викликів підпрограм, ніж самих підпрограм, прийняте просте правило: *кожна підпрограма сама зберігає старе значення rbp/ebp і відновлює його перед поверненням управління.*

Для зберігання `rbp/ebp` також використовується стек, причому команда `push rbp/ebp` виконується зразу після отримання управління підпрограмою. Таким чином, старе значення `rbp/ebp` розміщується в стеку *після адреси повернення* з підпрограми. Саме це старе значення `rbp/ebp` використовується як *реперна точка для адресації стекового фрейму*:

```
[ebp+16]
[ebp+12]    параметри
[ebp+8]
[ebp+4]     адреса повернення
[ebp]       збережене підпрограмою старе значення ebp
[ebp-4]
[ebp-8]     локальні змінні
```

Тому кожна підпрограма повинна виконати наступні команди перед початком роботи:

```
push ebp
mov ebp,esp
```

```
sub esp,8 ; замість 8 резервується потрібний обсяг пам'яті для локальних змінних
```

і перед завершенням роботи:

```
mov esp, ebp
pop ebp
ret
```

Процесор підтримує спеціальні команди для роботи із стеком підпрограмами. На початку підпрограми замість трьох команд використовується одна команда `enter 16,0` і в кінці підпрограми замість двох команд перед `ret` – одну команда `leave`.

ОС Linux створює стек автоматично при запуску любої програми і, більш того, під час її виконання при необхідності збільшує розмір доступної для стеку пам'яті.

Таким чином, можна записати наступну послідовність дій при роботі зі стеком:

Перед викликом підпрограми ***головна програма виконує такі дії***:

- записує значення всіх параметрів виклику підпрограми (функції) в стек у зворотному порядку (справа наліво);
- викликає команду `call`.

Команда ***call виконує наступні дії:***

- записує у стек адресу наступної команди після `call` (адресу повернення);
- модифікує регістр команд `rip/eip` так, щоб він містив адресу підпрограми.

До виконання підпрограми стек матиме наступний вигляд:

```
Аргумент N
...
Аргумент 2
Аргумент 1
Адреса повернення   <- [esp] верхівка стеку
```

Викликувана ***підпрограма на початку виконання здійснює***:

- записує в стек поточне значення вказівника бази стеку `rbp/ebp` командою `push rbp/ebp`;
- копіює вказівник верхівки стеку `rsp/esp` у вказівник бази `rbp/ebp` командою `mov rbp/ebp,rsp/esp`. Це дозволяє отримати доступ як до параметрів підпрограми, так і локальних змінних шляхом індексування вказівника бази. Вказівник бази `rbp/ebp` завжди вказуватиме на значення вказівника стеку `rsp/esp` на початку виконання підпрограми. Вказівник бази стеку є константою і дозволяє звертатися до всіх значень стекового фрейму (параметри, адреса повернення, `ebp/ebp`, локальні змінні).
- резервує місце під локальні змінні. Наприклад, під два слова `sub esp,8` (зменшує адресу бази стеку на 8 байтів).

Після цього стек матиме наступний вигляд:

```
Аргумент N                <-  [ebp+(N+1)*4]
...                       ...
Аргумент 2                <-  [ebp+3*4]
Аргумент 1                <-  [ebp+2*4]
Адреса повернення (esp)   <-  [ebp+4]
Старе (ebp)               <-  [ebp]
Локальна змінна1           <- [esp-4]
Локальна змінна2           <- [esp-8]
```

Викликувана ***підпрограма в кінці виконання здійснює***:

- записує значення, яке повертає підпрограма у регістр `rax/eax`;
- повертає стек в стан виклику підпрограми (відновлює `rsp/esp`, `rbp/ebp`);
- повертає керування в точку виклику командою `ret`, яка знімає значення з верхівки стеку (адресу повернення) і записує його в регістр вказівник команд `rip/eip`.

Ці кроки підпрограми виконують наступні команди:

```
mov esp,ebp
pop ebp
ret
```

Якщо потрібно перед викликом підпрограми зберегти регістри загального призначення то їх поміщають на зберігання в стек перед параметрами функції командою `pusha`, а потім відновлюють після завершення підпрограми командою `popa`.

Контрольні запитання.

1. Засоби передачі керування в програмах.
2. Умовні переходи. Прості умовні переходи
3. Переходи за результатами порівнянь
4. Команди організації циклів.
5. Команди безумовних переходів.
6. Стек і команди роботи із стеком.
7. Виклик підпрограм.
8. Послідовність дій викликаючої і викликуваної підпрограми.
9. Організація стекового фрейму.

7. СИСТЕМНІ ВИКЛИКИ, ПІДПРОГРАМИ

Мета. Вивчення процесів захищеного режиму, системних викликів їх аргументів, викликів підпрограм і передачі їм параметрів, багатомодульних програм і інтерфейсів Сі з асемблером, реентерабельних і рекурсивних підпрограм.

Вступ. У захищеному режимі процесор процес має доступ до 4 Гбайт логічної адреси і захищений адресний простір. У Linux процеси є нащадками іншого процесу – оболонки, в якій запущена програма. У процес можна передати параметри через командний рядок.

Системний виклик – це звернення задачі користувача за послугами до ядра операційної системи В Linux можна звернутися до ядра операційної системи через програмне переривання (команда `syscall`) з номером `60`. Кожний системний виклик має свій номер. Більш докладну інформацію про системні виклики можна отримати командою Linux `man <ім'я_виклику>`.

Підпрограма є невеликим фрагментом коду, який може бути викликаний з різних частин програми. Для виклику підпрограм використовується інструкція `jmp` або `call`. У підпрограму можна передати параметри через регістри або стек. При виклику підпрограм асемблера з мови Сі дотримуються стандартних домовленостей про правила передачі аргументів і повернення результатів. Підпрограми можуть бути реентерабельними, що дозволяє їх використовувати у багатозадачному режимі. Рекурсивні підпрограми дозволяють самовиклик.

План.

1. Процеси у захищеному режимі
2. Системні виклики x64
   2.1. Аргументи системних викликів
3. Підпрограми
   3.1. Передача параметрів у підпрограму
   3.2. Локальні змінні у стеку
4. Виклик підпрограм асемблера з Сі програми
5. Домовленості про виклики підпрограм користувача
   5.1. Особливості виклику підпрограм у мові Сі
6. Реентерабельні і рекурсивні підпрограми

1. Процеси у захищеному режимі

Linux є багатозадачною операційною системою, яка строго розділяє індивідуальні процеси. Це значить, що ні один процес не може змінити інші процеси. В Intel процесорах x86_64 процеси у пам'яті захищені так званим захищеним режимом процесора. Цей режим дозволяє контролювати дії програми: доступ програми до пам'яті і периферійних пристроїв обмежений правами доступу. Механізми захисту розділені між ядром операційної системи (якому дозволяється абсолютно все) і процесами (які можуть виконувати тільки непривілейовані команди і записувати дані тільки у свою область пам'яті).

Захищений режим також підтримує віртуальну пам'ять. Ядро операційної системи надає всі операції для роботи з віртуальною пам'яттю (звичайно крім трансляції логічних адрес у фізичні, що виконується апаратно).

Завдяки віртуальній пам’яті любий процес може адресувати 2^{32}=4 Гбайт (32-розрядні регістри) і 2^{40}=1 Тбайт (64-розрядні регістри) адресного простору. Процес розміщує у цьому адресному просторі чотири секції: коду (`.txt`), статичних даних (`.data`), динамічних даних (`.bss`, купа) і стеку (`.stack`). Порядок розміщення секцій визначається форматом виконуваного файлу. Так Linux підтримує декілька форматів, з яких найбільш поширеним є ELF-формат.

Звичайно програма завантажується з адреси `0x08048000` (приблизно 128 Мбайт). При цьому завантажується одна сторінка, а інші – за необхідністю. На диску програма зберігається без секцій `.bss` і `.stack` – ці секції появляються тільки при завантаженні програми у пам’ять. Якщо програма підключає які-небудь динамічні бібліотеки, їх модулі завантажуються у адресний простір починаючи з іншої адреси (звичайно з 1 Гбайт і вище). Між секціями `.bss` і `.stack` залишається шпара в 3 Гбайт. Ця пам'ять належить процесу, але вона не розподіляється на сторінки. Запис у цю область спричинить збій сторінки (page fault) і ядро знищить програму.

У Linux процеси є нащадками іншого процесу – оболонки, в якій запущена програма. Для передачі процесу параметрів командного рядка і змінних оточення, вони поміщаються у стек:

RSP після запуску програми	Вільна пам’ять
argc	Кількість параметрів
argv[0]	Вказівник на ім’я програми
argv[1] argv[argc-1]	Вказівники на аргументи програми
NULL	Кінець аргументів командного рядка
env[0] env[1] env[n]	Вказівники на змінні оточення
NULL	Кінець змінних оточення

Кожний елемент командного рядка зберігається у пам’яті як рядок, що закінчується нульовим байтом. Вивести значення елементів стеку на екран можна наступною програмою:

```
extern printf
segment .data
fmt: db "%s",10,0
segment .text
global main
main:
  mov rcx,[rsp+8]   ; argc
  mov rdx,[rsp+16]  ; argc
top:
  push rcx    ; збереження регістрів, які використовує printf
  push rdx
  push qword [rdx]
  push qword fmt
  call printf
  add rsp,16   ;  вилучення 2-х параметрів

  pop rdx     ; відновлення регістрів для printf
  pop rcx
  add rdx,8   ; вказівник на наступний аргумент
  dec rcx     ; лічильник аргументів
  jnz top
```

```
ret
```

2. Системні виклики ОС Linux x86_64

Команда системного виклику: **syscall**
Номер системного виклику: **`rax`**. Повернення значень: **`rax`**.
Передача аргументів у системні виклики: **rdi, rsi, rdx, r10, r8, r9**

rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c
13	rt_sigaction	sys_rt_sigaction	kernel/signal.c
14	rt_sigprocmask	sys_rt_sigprocmask	kernel/signal.c
15	rt_sigreturn	stub_rt_sigreturn	arch/x86/kernel/signal.c
16	ioctl	sys_ioctl	fs/ioctl.c
17	pread64	sys_pread64	fs/read_write.c
18	pwrite64	sys_pwrite64	fs/read_write.c
19	readv	sys_readv	fs/read_write.c
20	writev	sys_writev	fs/read_write.c
21	access	sys_access	fs/open.c
22	pipe	sys_pipe	fs/pipe.c
23	select	sys_select	fs/select.c
24	sched_yield	sys_sched_yield	kernel/sched/core.c
25	mremap	sys_mremap	mm/mmap.c
26	msync	sys_msync	mm/msync.c
27	mincore	sys_mincore	mm/mincore.c
28	madvise	sys_madvise	mm/madvise.c
29	shmget	sys_shmget	ipc/shm.c
30	shmat	sys_shmat	ipc/shm.c
31	shmctl	sys_shmctl	ipc/shm.c
32	dup	sys_dup	fs/file.c
33	dup2	sys_dup2	fs/file.c
34	pause	sys_pause	kernel/signal.c

rax	Name	Entry point	Implementation
35	nanosleep	sys_nanosleep	kernel/hrtimer.c
36	getitimer	sys_getitimer	kernel/itimer.c
37	alarm	sys_alarm	kernel/timer.c
38	setitimer	sys_setitimer	kernel/itimer.c
39	getpid	sys_getpid	kernel/sys.c
40	sendfile	sys_sendfile64	fs/read_write.c
41	socket	sys_socket	net/socket.c
42	connect	sys_connect	net/socket.c
43	accept	sys_accept	net/socket.c
44	sendto	sys_sendto	net/socket.c
45	recvfrom	sys_recvfrom	net/socket.c
46	sendmsg	sys_sendmsg	net/socket.c
47	recvmsg	sys_recvmsg	net/socket.c
48	shutdown	sys_shutdown	net/socket.c
49	bind	sys_bind	net/socket.c
50	listen	sys_listen	net/socket.c
51	getsockname	sys_getsockname	net/socket.c
52	getpeername	sys_getpeername	net/socket.c
53	socketpair	sys_socketpair	net/socket.c
54	setsockopt	sys_setsockopt	net/socket.c
55	getsockopt	sys_getsockopt	net/socket.c
56	clone	stub_clone	kernel/fork.c
57	fork	stub_fork	kernel/fork.c
58	vfork	stub_vfork	kernel/fork.c
59	execve	stub_execve	fs/exec.c
60	exit	sys_exit	kernel/exit.c
61	wait4	sys_wait4	kernel/exit.c
62	kill	sys_kill	kernel/signal.c
63	uname	sys_newuname	kernel/sys.c
64	semget	sys_semget	ipc/sem.c
65	semop	sys_semop	ipc/sem.c
66	semctl	sys_semctl	ipc/sem.c
67	shmdt	sys_shmdt	ipc/shm.c
68	msgget	sys_msgget	ipc/msg.c
69	msgsnd	sys_msgsnd	ipc/msg.c
70	msgrcv	sys_msgrcv	ipc/msg.c
71	msgctl	sys_msgctl	ipc/msg.c
72	fcntl	sys_fcntl	fs/fcntl.c
73	flock	sys_flock	fs/locks.c
74	fsync	sys_fsync	fs/sync.c
75	fdatasync	sys_fdatasync	fs/sync.c
76	truncate	sys_truncate	fs/open.c
77	ftruncate	sys_ftruncate	fs/open.c

rax	Name	Entry point	Implementation
78	getdents	sys_getdents	fs/readdir.c
79	getcwd	sys_getcwd	fs/dcache.c
80	chdir	sys_chdir	fs/open.c
81	fchdir	sys_fchdir	fs/open.c
82	rename	sys_rename	fs/namei.c
83	mkdir	sys_mkdir	fs/namei.c
84	rmdir	sys_rmdir	fs/namei.c
85	creat	sys_creat	fs/open.c
86	link	sys_link	fs/namei.c
87	unlink	sys_unlink	fs/namei.c
88	symlink	sys_symlink	fs/namei.c
89	readlink	sys_readlink	fs/stat.c
90	chmod	sys_chmod	fs/open.c
91	fchmod	sys_fchmod	fs/open.c
92	chown	sys_chown	fs/open.c
93	fchown	sys_fchown	fs/open.c
94	lchown	sys_lchown	fs/open.c
95	umask	sys_umask	kernel/sys.c
96	gettimeofday	sys_gettimeofday	kernel/time.c
97	getrlimit	sys_getrlimit	kernel/sys.c
98	getrusage	sys_getrusage	kernel/sys.c
99	sysinfo	sys_sysinfo	kernel/sys.c
100	times	sys_times	kernel/sys.c
101	ptrace	sys_ptrace	kernel/ptrace.c
102	getuid	sys_getuid	kernel/sys.c
103	syslog	sys_syslog	kernel/printk/printk.c
104	getgid	sys_getgid	kernel/sys.c
105	setuid	sys_setuid	kernel/sys.c
106	setgid	sys_setgid	kernel/sys.c
107	geteuid	sys_geteuid	kernel/sys.c
108	getegid	sys_getegid	kernel/sys.c
109	setpgid	sys_setpgid	kernel/sys.c
110	getppid	sys_getppid	kernel/sys.c
111	getpgrp	sys_getpgrp	kernel/sys.c
112	setsid	sys_setsid	kernel/sys.c
113	setreuid	sys_setreuid	kernel/sys.c
114	setregid	sys_setregid	kernel/sys.c
115	getgroups	sys_getgroups	kernel/groups.c
116	setgroups	sys_setgroups	kernel/groups.c
117	setresuid	sys_setresuid	kernel/sys.c
118	getresuid	sys_getresuid	kernel/sys.c
119	setresgid	sys_setresgid	kernel/sys.c
120	getresgid	sys_getresgid	kernel/sys.c

rax	Name	Entry point	Implementation
121	getpgid	sys_getpgid	kernel/sys.c
122	setfsuid	sys_setfsuid	kernel/sys.c
123	setfsgid	sys_setfsgid	kernel/sys.c
124	getsid	sys_getsid	kernel/sys.c
125	capget	sys_capget	kernel/capability.c
126	capset	sys_capset	kernel/capability.c
127	rt_sigpending	sys_rt_sigpending	kernel/signal.c
128	rt_sigtimedwait	sys_rt_sigtimedwait	kernel/signal.c
129	rt_sigqueueinfo	sys_rt_sigqueueinfo	kernel/signal.c
130	rt_sigsuspend	sys_rt_sigsuspend	kernel/signal.c
131	sigaltstack	sys_sigaltstack	kernel/signal.c
132	utime	sys_utime	fs/utimes.c
133	mknod	sys_mknod	fs/namei.c
134	uselib		fs/exec.c
135	personality	sys_personality	kernel/exec_domain.c
136	ustat	sys_ustat	fs/statfs.c
137	statfs	sys_statfs	fs/statfs.c
138	fstatfs	sys_fstatfs	fs/statfs.c
139	sysfs	sys_sysfs	fs/filesystems.c
140	getpriority	sys_getpriority	kernel/sys.c
141	setpriority	sys_setpriority	kernel/sys.c
142	sched_setparam	sys_sched_setparam	kernel/sched/core.c
143	sched_getparam	sys_sched_getparam	kernel/sched/core.c
144	sched_setscheduler	sys_sched_setscheduler	kernel/sched/core.c
145	sched_getscheduler	sys_sched_getscheduler	kernel/sched/core.c
146	sched_get_priority_max	sys_sched_get_priority_max	kernel/sched/core.c
147	sched_get_priority_min	sys_sched_get_priority_min	kernel/sched/core.c
148	sched_rr_get_interval	sys_sched_rr_get_interval	kernel/sched/core.c
149	mlock	sys_mlock	mm/mlock.c
150	munlock	sys_munlock	mm/mlock.c
151	mlockall	sys_mlockall	mm/mlock.c
152	munlockall	sys_munlockall	mm/mlock.c
153	vhangup	sys_vhangup	fs/open.c
154	modify_ldt	sys_modify_ldt	arch/x86/um/ldt.c
155	pivot_root	sys_pivot_root	fs/namespace.c
156	_sysctl	sys_sysctl	kernel/sysctl_binary.c
157	prctl	sys_prctl	kernel/sys.c
158	arch_prctl	sys_arch_prctl	arch/x86/um/syscalls_64.c
159	adjtimex	sys_adjtimex	kernel/time.c
160	setrlimit	sys_setrlimit	kernel/sys.c
161	chroot	sys_chroot	fs/open.c

rax	Name	Entry point	Implementation
162	sync	sys_sync	fs/sync.c
163	acct	sys_acct	kernel/acct.c
164	settimeofday	sys_settimeofday	kernel/time.c
165	mount	sys_mount	fs/namespace.c
166	umount2	sys_umount	fs/namespace.c
167	swapon	sys_swapon	mm/swapfile.c
168	swapoff	sys_swapoff	mm/swapfile.c
169	reboot	sys_reboot	kernel/reboot.c
170	sethostname	sys_sethostname	kernel/sys.c
171	setdomainname	sys_setdomainname	kernel/sys.c
172	iopl	stub_iopl	arch/x86/kernel/ioport.c
173	ioperm	sys_ioperm	arch/x86/kernel/ioport.c
174	create_module		NOT Implemented
175	init_module	sys_init_module	kernel/module.c
176	delete_module	sys_delete_module	kernel/module.c
177	get_kernel_syms		Not implemented
178	query_module		Not implemented
179	quotactl	sys_quotactl	fs/quota/quota.c
180	nfsservctl		Not implemented
181	getpmsg		Not implemented
182	putpmsg		Not implemented
183	afs_syscall		Not implemented
184	tuxcall		Not implemented
185	security		Not implemented
186	gettid	sys_gettid	kernel/sys.c
187	readahead	sys_readahead	mm/readahead.c
188	setxattr	sys_setxattr	fs/xattr.c
189	lsetxattr	sys_lsetxattr	fs/xattr.c
190	fsetxattr	sys_fsetxattr	fs/xattr.c
191	getxattr	sys_getxattr	fs/xattr.c
192	lgetxattr	sys_lgetxattr	fs/xattr.c
193	fgetxattr	sys_fgetxattr	fs/xattr.c
194	listxattr	sys_listxattr	fs/xattr.c
195	llistxattr	sys_llistxattr	fs/xattr.c
196	flistxattr	sys_flistxattr	fs/xattr.c
197	removexattr	sys_removexattr	fs/xattr.c
198	lremovexattr	sys_lremovexattr	fs/xattr.c
199	fremovexattr	sys_fremovexattr	fs/xattr.c
200	tkill	sys_tkill	kernel/signal.c
201	time	sys_time	kernel/time.c
202	futex	sys_futex	kernel/futex.c
203	sched_setaffinity	sys_sched_setaffinity	kernel/sched/core.c

rax	Name	Entry point	Implementation
204	sched_getaffinity	sys_sched_getaffinity	kernel/sched/core.c
205	set_thread_area		arch/x86/kernel/tls.c
206	io_setup	sys_io_setup	fs/aio.c
207	io_destroy	sys_io_destroy	fs/aio.c
208	io_getevents	sys_io_getevents	fs/aio.c
209	io_submit	sys_io_submit	fs/aio.c
210	io_cancel	sys_io_cancel	fs/aio.c
211	get_thread_area		arch/x86/kernel/tls.c
212	lookup_dcookie	sys_lookup_dcookie	fs/dcookies.c
213	epoll_create	sys_epoll_create	fs/eventpoll.c
214	epoll_ctl_old		Not implemented
215	epoll_wait_old		Not implemented
216	remap_file_pages	sys_remap_file_pages	mm/fremap.c
217	getdents64	sys_getdents64	fs/readdir.c
218	set_tid_address	sys_set_tid_address	kernel/fork.c
219	restart_syscall	sys_restart_syscall	kernel/signal.c
220	semtimedop	sys_semtimedop	ipc/sem.c
221	fadvise64	sys_fadvise64	mm/fadvise.c
222	timer_create	sys_timer_create	kernel/posix-timers.c
223	timer_settime	sys_timer_settime	kernel/posix-timers.c
224	timer_gettime	sys_timer_gettime	kernel/posix-timers.c
225	timer_getoverrun	sys_timer_getoverrun	kernel/posix-timers.c
226	timer_delete	sys_timer_delete	kernel/posix-timers.c
227	clock_settime	sys_clock_settime	kernel/posix-timers.c
228	clock_gettime	sys_clock_gettime	kernel/posix-timers.c
229	clock_getres	sys_clock_getres	kernel/posix-timers.c
230	clock_nanosleep	sys_clock_nanosleep	kernel/posix-timers.c
231	exit_group	sys_exit_group	kernel/exit.c
232	epoll_wait	sys_epoll_wait	fs/eventpoll.c
233	epoll_ctl	sys_epoll_ctl	fs/eventpoll.c
234	tgkill	sys_tgkill	kernel/signal.c
235	utimes	sys_utimes	fs/utimes.c
236	vserver		Not implemented
237	mbind	sys_mbind	mm/mempolicy.c
238	set_mempolicy	sys_set_mempolicy	mm/mempolicy.c
239	get_mempolicy	sys_get_mempolicy	mm/mempolicy.c
240	mq_open	sys_mq_open	ipc/mqueue.c
241	mq_unlink	sys_mq_unlink	ipc/mqueue.c
242	mq_timedsend	sys_mq_timedsend	ipc/mqueue.c
243	mq_timedreceive	sys_mq_timedreceive	ipc/mqueue.c
244	mq_notify	sys_mq_notify	ipc/mqueue.c
245	mq_getsetattr	sys_mq_getsetattr	ipc/mqueue.c
246	kexec_load	sys_kexec_load	kernel/kexec.c

rax	Name	Entry point	Implementation
247	waitid	sys_waitid	kernel/exit.c
248	add_key	sys_add_key	security/keys/keyctl.c
249	request_key	sys_request_key	security/keys/keyctl.c
250	keyctl	sys_keyctl	security/keys/keyctl.c
251	ioprio_set	sys_ioprio_set	fs/ioprio.c
252	ioprio_get	sys_ioprio_get	fs/ioprio.c
253	inotify_init	sys_inotify_init	fs/notify/inotify/inotify_user.c
254	inotify_add_watch	sys_inotify_add_watch	fs/notify/inotify/inotify_user.c
255	inotify_rm_watch	sys_inotify_rm_watch	fs/notify/inotify/inotify_user.c
256	migrate_pages	sys_migrate_pages	mm/mempolicy.c
257	openat	sys_openat	fs/open.c
258	mkdirat	sys_mkdirat	fs/namei.c
259	mknodat	sys_mknodat	fs/namei.c
260	fchownat	sys_fchownat	fs/open.c
261	futimesat	sys_futimesat	fs/utimes.c
262	newfstatat	sys_newfstatat	fs/stat.c
263	unlinkat	sys_unlinkat	fs/namei.c
264	renameat	sys_renameat	fs/namei.c
265	linkat	sys_linkat	fs/namei.c
266	symlinkat	sys_symlinkat	fs/namei.c
267	readlinkat	sys_readlinkat	fs/stat.c
268	fchmodat	sys_fchmodat	fs/open.c
269	faccessat	sys_faccessat	fs/open.c
270	pselect6	sys_pselect6	fs/select.c
271	ppoll	sys_ppoll	fs/select.c
272	unshare	sys_unshare	kernel/fork.c
273	set_robust_list	sys_set_robust_list	kernel/futex.c
274	get_robust_list	sys_get_robust_list	kernel/futex.c
275	splice	sys_splice	fs/splice.c
276	tee	sys_tee	fs/splice.c
277	sync_file_range	sys_sync_file_range	fs/sync.c
278	vmsplice	sys_vmsplice	fs/splice.c
279	move_pages	sys_move_pages	mm/migrate.c
280	utimensat	sys_utimensat	fs/utimes.c
281	epoll_pwait	sys_epoll_pwait	fs/eventpoll.c
282	signalfd	sys_signalfd	fs/signalfd.c
283	timerfd_create	sys_timerfd_create	fs/timerfd.c
284	eventfd	sys_eventfd	fs/eventfd.c
285	fallocate	sys_fallocate	fs/open.c
286	timerfd_settime	sys_timerfd_settime	fs/timerfd.c
287	timerfd_gettime	sys_timerfd_gettime	fs/timerfd.c
288	accept4	sys_accept4	net/socket.c
289	signalfd4	sys_signalfd4	fs/signalfd.c

rax	Name	Entry point	Implementation
290	eventfd2	sys_eventfd2	fs/eventfd.c
291	epoll_create1	sys_epoll_create1	fs/eventpoll.c
292	dup3	sys_dup3	fs/file.c
293	pipe2	sys_pipe2	fs/pipe.c
294	inotify_init1	sys_inotify_init1	fs/notify/inotify/inotify_user.c
295	preadv	sys_preadv	fs/read_write.c
296	pwritev	sys_pwritev	fs/read_write.c
297	rt_tgsigqueueinfo	sys_rt_tgsigqueueinfo	kernel/signal.c
298	perf_event_open	sys_perf_event_open	kernel/events/core.c
299	recvmmsg	sys_recvmmsg	net/socket.c
300	fanotify_init	sys_fanotify_init	fs/notify/fanotify/fanotify_user.c
301	fanotify_mark	sys_fanotify_mark	fs/notify/fanotify/fanotify_user.c
302	prlimit64	sys_prlimit64	kernel/sys.c
303	name_to_handle_at	sys_name_to_handle_at	fs/fhandle.c
304	open_by_handle_at	sys_open_by_handle_at	fs/fhandle.c
305	clock_adjtime	sys_clock_adjtime	kernel/posix-timers.c
306	syncfs	sys_syncfs	fs/sync.c
307	sendmmsg	sys_sendmmsg	net/socket.c
308	setns	sys_setns	kernel/nsproxy.c
309	getcpu	sys_getcpu	kernel/sys.c
310	process_vm_readv	sys_process_vm_readv	mm/process_vm_access.c
311	process_vm_writev	sys_process_vm_writev	mm/process_vm_access.c
312	kcmp	sys_kcmp	kernel/kcmp.c
313	finit_module	sys_finit_module	kernel/module.c

2.1. Аргументи системних викликів

Типові системні виклики і їх аргументи показані у табл. 1.

Таблиця 1 – Типові системні виклики і їх аргументи

Номер системного виклику	Системний сервіс	Описання
0	**SYS_read**	**Читання символів** `rdi` – файловий дескриптор `rsi` – адреса символьних стрічок `rdx` – кількість символів для читання При успішному читання повертається кількість прочитаних символів, при неуспішному – негативне значення.
1	**SYS_write**	**Запис символів** `rdi` – файловий дескриптор `rsi` – адреса символьних стрічок `rdx` – кількість записаних символів При успішному запису повертається кількість записаних символів,

		при неуспішному – негативне значення.
2	**SYS_open**	**Відкриття файлу** `rdi` – адресу стрічки з іменем файлу, яка має NULL закінчення. `rsi` – прапори режимів використання файлу (звичайний O_RDONLY). При успішному відкритті повертається дескриптор файлу, при неуспішному – негативне значення.
3	**SYS_close**	**Відкриття файлу** `rdi` – файловий дескриптор відкритого файлу для закриття. При неуспішному закритті повертається негативне значення.
8	**SYS_lseek**	**Зміщення вказівника для запису/читання файлу** `rdi` – файловий дескриптор `rsi` – зміщення `rdx` – поточне значення При неуспішному зміщенні повертається негативне значення.
57	**SYS_fork**	**Галуження поточного процесу**
59	**SYS_execv**	**Виконати програму**
60	**SYS_exit**	**Зупинити виконання процесу** `rdi` – код завершення (звичайно 0)
85	**SYS_creat**	**Відкрити/Створити файл** `rdi` – адресу стрічки з іменем файлу, яка має NULL закінчення `rsi` – прапори режимів використання файлу
96	**SYS_gettimeofday**	**Отримати дату і час дня** `rdi` – адреса структури із значенням дати `rsi` – адреса структури із значенням часу

Для файлових операцій необхідно задати режими використання (табл.2) і права доступу до файлів (табл. 3).

Таблиця 2 – Режими використання файлів

Режим використання	Значення	Описання
O_RDONLY	0	Тільки читання.
O_WRONLY	1	Тільки запис. Типово використовується при добавленні даних у файл.
O_RDWR	2	Дозвіл на одночасне читання і запис.

Таблиця 3 – Права доступу до файлів

Права доступу	Значення	Описання
S_IRWXU	00700q	Користувач (власник файлу) має права `rwe`
S_IRUSR	00400q	Користувач (власник файлу) має права `r--`
S_IWUSR	00200q	Користувач (власник файлу) має права `-w-`
S_IXUSR	00100q	Користувач (власник файлу) має права `--e`
S_IRWXG	00070q	Група має права `rwe`
S_IRGRP	00040q	Група має права `r--`

S_IWGRP	00020q	Група має права `-w-`
S_IXGRP	00010q	Група має права `--e`
S_IRWXO	00007q	Інші мають права `rwe`
S_IROTH	00004q	Інші мають права `r--`
S_IWOTH	00002q	Інші мають права `-w-`
S_IXOTH	00001q	Інші мають права `--e`

Якщо системний виклик завершується з помилкою то він повертає негативне значення. Значення кодів помилок показано в табл. 4.

Таблиця 4 – Коди помилок системних викликів

Код помилки	Символічне ім’я	Описання
-1	EPERM	Заборонена операція
-2	ENOENT	Відсутній файл або каталог
-3	ESRCH	Відсутній процес
-4	EINTR	Перерваний системний виклик
-5	EIO	Помилка I/O
-6	ENXIO	Відсутній пристрій або адреса
-7	E2BIG	Занадто довгий список аргументів
-8	ENOEXEC	Помилка EXEC формату
-9	EBADF	Помилковий номер файлу
-10	ECHILD	Відсутній дочірній процес
-11	EAGAIN	Попробувати знову
-12	ENOMEM	Вихід за межі пам’яті
-13	EACCES	Заборонений доступ
-14	EFAULT	Помилкова адреса
-15	ENOTBLK	Потрібний блоковий пристрій
-16	EBUSY	Пристрій або ресурс зайнятий
-17	EEXIST	Файл існує
-18	EXDEV	Зв’язок між пристроями
-19	ENODEV	Відсутній такий пристрій
-20	ENOTDIR	Не не каталог
-21	EISDIR	Це каталог
-22	EINVAL	Помилковий аргумент
-23	ENFILE	Переповнення файлової таблиці
-24	EMFILE	Занадто багато відкритих файлів
-25	ENOTTY	Не пристрій друку
-26	ETXTBSY	Тестовий файл зайнятий
-27	EFBIG	Файл занадто великий

-28	ENOSPC	На пристрої не залишилося місця
-29	ESPIPE	Недійсний пошук
-30	EROFS	Файлова система read-only
-31	EMLINK	Забагато посилань
-32	EPIPE	Пошкоджений канал
-33	EDOM	Математичний аргумент поза межами видимості функції
-34	ERANGE	Математичний результат неможливо подати

Повний перелік помилок знаходиться в каталозі (Ubuntu)
`/usr/include/asm-generic/errno-base.h`.

3. Підпрограми

Підпрограма є невеликим фрагментом коду, який може бути викликаний з різних частин програми. Для виклику підпрограми можна використати інструкцію `jmp`, але якщо підпрограма викликається з різних частин програми то необхідно забезпечити повернення назад у різні точки виклику. Явною формою виклику інструкції `jmp label`, де `label` – задана позначка повернення, не можна забезпечити повернення у різні точки виклику підпрограми. Але це можна зробити при *неявній* формі виклику інструкції `jmp` і прийнятті домовленостей, що адреса позначки повернення заноситься у регістр `rbx`:

```
    mov rbx, return1   ;  rbx = &return1
    jmp subprog1 -->
--> return1:       |
|     ...          |
|   subprog1: <----
|...
<-- jmp rbx           ; повернення за адресою позначки return1
```

Для неявної форми виклику можуть бути використані регістри `rax, rbx, rcx, rdx, rsi, rdi`. Але при не дотриманні прийнятих домовленостей і поміщенні у регістри не адрес повернення, помилки виводитися не будуть, але програма буде працювати не коректно.

Більш поширений метод виклику підпрограм з використанням стеку. Intel процесори апаратно підтримують роботу стеку. Інструкція `push` вставляє `qword` слово у стек віднімаючи 8 від вмісту регістра `rsp` (вказівник верхівки стеку) і записуючи його значення як `[rsp]`. Інструкція `pop` читає `qword` слово з `[rsp]` і потім додає 8 до вмісту регістра `rsp`.

Приклад роботи команд `push, pop` в припущенні, що початкове значення `rsp=1000h`.

```
push qword 1 ; 1 записано за адресою ESP = 0FF8h   (1000h - 8)
push qword 2 ; 2 записано за адресою ESP = 0FF0h   (0FF8h - 8)
push qword 3 ; 3 записано за адресою ESP = 0FE8h   (0FF0h - 8)
pop rax ; rax = 3, rsp = 0FF0h
pop rbx ; rbx = 2, rsp = 0FF8h
pop rcx ; rcx = 1, rsp = 1000h
```

Для збереження і відновлення усіх регістрів використовуються макроси:

```
%macro pushaq                          %macro popaq
```

```
        push rax                        pop rax
        push rbx                        pop rbx
        push rcx                        pop rcx
        push rdx                        pop rdx
        push rbp                        pop rbp
        push rsp                        pop rsp
        push rsi                        pop rsi
        push rdi                        pop rdi
        push rdi                        pop rdi
        push r8                         pop r8
        push r9                         pop r9
        push r10                        pop r10
        push r11                        pop r11
        push r12                        pop r12
        push r13                        pop r13
        push r14                        pop r14
        push r15                        pop r15
    %endmacro                       %endmacro
```

Для спрощення виклику підпрограм є дві інструкції `call` і `ret`. Інструкція `call` робить безумовний перехід на підпрограму і записує у верхівку стеку `rsp` вміст регістра `rip`. Регістр `rip` містить адресу наступної інструкції після `call`.

Інструкція `ret` використовується для повернення з підпрограми. Вона зчитує значення з верхівки стеку `rsp` (адресу повернення) у регістр `rip`. Виклик підпрограми у такому випадку буде наступний:

```
call subprog1   ; запис у стек адреси return1 -> rsp
return1:
     ...
subprog1:
     ...
ret             ;  читання зі стеку і повернення за адресою return1
```

При виклику підпрограм можуть передаватися або повертатися деякі дані. Мови високого рівня мають стандартні домовленості про передачу таких даних. При виклику підпрограм асемблера і передачі їм параметрів необхідно дотримуватися домовленостей виклику мов високого рівня. Одна із універсальних домовленостей полягає у тому, що підпрограма викликається інструкцією `call` і повертається інструкцією `ret`. Інші домовленості можуть залежати від компілятора.

3.1. Передача параметрів у підпрограму

Параметри підпрограми записуються у стек *перед* викликом інструкції `call`. Якщо параметри будуть змінюватися у підпрограмі, то передаються їхні *адреси*, а не *значення*. Якщо розмір параметра менший від `qword`, то він перетворюється у `qword` слово.

При передачі підпрограмі одного параметру стан стеку показано на рис.1, а. Якщо підпрограма запише в стек одну локальну змінну, то стан стеку зміниться, рис.1, б (розмір стеку збільшується при зменшенні адреси на 8).

Регістри	Стек	Регістри*	Стек	Записує в стек

rsp+8	Параметр	rsp+16	Параметр	Осн. програма
rsp	Адреса повернення	rsp+8	Адреса повернення	Осн. програма
		rsp	Локальна змінна 1	Підпрограма*

а) б)

Рисунок 1 – Стан стеку при виклику підпрограми

У випадку "а)" значення параметру буде [rsp+8], а у випадку "б)" – [rsp+16]. При використанні локальних змінних змінюється адреса параметрів відносно регістра rsp, що є незручним і може бути джерелом помилок. Для розв'язання цієї проблеми використовується регістр rbp для посилання на дані у стеку. Для цього кожний кадр стеку (англ. stack frame) визначається значенням rbp, як початком кадра стеку, і rsp, як верхівкою кадра стеку. За домовленостями викликів мови Сі прийнято, що *підпрограма* спочатку записує оригінальне значення програми rbp у стек, а потім встановлює нове значення підпрограми rbp рівним значенню rsp. Це дозволяє при поповненні стеку міняти значення rsp, без зміни rbp, рис. 2.

Регістри		Стек	Записує в стек
rsp+24	rbp+16	Параметр	Осн. програма
rsp+16	rbp+8	Адреса повернення	Осн. програма
rsp+8	rbp	Збережене rbp	Підпрограма
rsp	rbp-8	Локальна змінна 1	Підпрограма

Рисунок 2 – Стан стеку при виклику із збереженням rbp

При завершенні підпрограма має відновити початкове значення rbp. Приклад коду підпрограми, яка зберігає і відновлює оригінальне значення rbp:

```
1  subprogram:
2  push rbp      ; збереження оригінальної бази rbp у стеку
3  mov rbp,rsp   ; значення бази нового фрейму rbp = rsp
4  ; код підпрограми
5  mov rsp,rbp
6  pop rbp       ; відновлення оригінального значення rbp
7  ret
```

Рядки коду 2, 3 і 5, 6 є однаковими для всіх підпрограм. Рядки 2, 3 називаються епілогом, а 5, 6 – прологом підпрограми. Тепер аргумент є доступним у підпрограмі як [rbp+16], незалежно від того, що буде поміщено у стек підпрограмою.

Після завершення підпрограми параметри мають бути вилучені зі стеку. Згідно домовленостей виклику мови Сі параметри має вилучати *викликаюча* програма, а згідно домовленостей виклику мови Pascal – параметри вилучає *підпрограма*.

Приклад виклику підпрограми згідно домовленостей мови Сі. Параметри заноситься у стек інструкцією push, а вилучається інструкцією add. Можна було б використати інструкцію pop rsp, а але вона записує у регістр rsp непотрібне значення.

```
1  push qword 1 ; записування в стек 1 як параметра
2  call fun
```

```
3 | add rsp, 8   ; вилучення параметру зі стеку
```

3.2. Локальні змінні у стеку

Стек також використовується для зберігання локальних змінних (в термінах мови Сі вони є автоматичними). Підпрограма, яка використовує стек для зберігання локальних змінних є *реентерабельною*. Реентерабельну підпрограму можна викликати в любому місці програми, в тому числі із самої себе, тобто вона є рекурсивною. Змінні, які зберігаються у секції `.data` існують від початку до кінця існування програми (в термінах мови Сі вони є глобальними або статичними).

Локальні змінні зберігаються у стеку зразу після збереженого `rbp`. Місце під них виділяє підпрограма відніманням числа потрібних для них байтів від `rsp`.

```
1 | subprogram:
2 | push rbp      ; збереження оригінального rbp у стеку
3 | mov rbp,rsp   ; нове значення rbp = rsp
4 | sub rsp,LOCAL_BYTES  ; число байтів для локальних змінних
5 | ; код підпрограми
6 | add rsp,LOCAL_BYTES ; звільнення стеку від локальних змінних
7 | mov rsp,rbp
8 | pop rbp      ; відновлення оригінального значення rbp
9 | ret
```

У цьому випадку епілог і пролог програми можуть бути спрощені за рахунок використання двох спеціальних інструкцій. Інструкція `enter` виконує епілог, а `leave` – пролог. Для домовленостей виклику мови Сі інструкція `enter` має два параметри, перший – число байт для локальних змінних, другий – 0. Інструкція `leave` параметрів не має. Підпрограми з використанням інструкцій `enter` і `leave`.

```
1 | subprogram:
2 | enter LOCAL_BYTES,0 ; виділення байтів у стеку для локальних змінних
3 | ; код підпрограми
4 | leave               ; звільнення стеку від локальних змінних
5 | ret
```


4. Виклик підпрограм асемблера з Сі програми

На практиці при написанні програм асемблер використовується рідко. Це зумовлено складністю як програмування, так і перенесення програм на інші платформи. Тому асемблер переважно використовують для написання критичних підпрограм у програмах, написаних на мовах високого рівня. Найпростіша багатомодульна програма може використовувати основну програму на Сі як драйвер, яка викликає підпрограму написану на асемблері. Це дозволяє Сі середовищу налаштувати програму для виконання у захищеному режимі. Всі сегменти і їх сегментні регістри будуть ініціалізовані середовищем Сі, що звільняє асемблерний код від технічної роботи. Крім того, асемблерний код отримує доступ до бібліотек мови Сі.

При компонуванні багатомодульних програм необхідно узгодити посилання з одних модулів на позначки, які визначені в інших модулях. Для цього використовується директива

`external` із перерахованими через кому позначками. Директива вказує асемблеру що ці позначки є зовнішніми до даного модуля. Тобто ці позначки можуть використовуватися у даному модулі, але визначені у інших модулях. Для того, щоб позначки були доступні іншим модулям, вони мають бути оголошеними як `global`.

Приклад виклику асемблерної підпрограми із Сі програми.

```
1  /* файл 1.c. Компонування з об'єктним файлом asm 1.o
2     gcc 1.c 1.o -o 1
3  */
4  int main()
5  {
6     int ret status ;
7     ret status = asm_main();
8     return ret status ;
9  }
```

Асемблерна програма зчитує з консолі два числа і виводить їх суму на екран. Для введення/виведення використовуються підпрограми асемблера, які звертаються до бібліотек мови Сі.

```
1       ; файл 1.asm. Асемблювання в об'єктний файл:
2       ; nasm –f elf64 1.asm -o 1
3       extern printf
4       extern scanf
5       ; сегмент даних
6       segment .data
7       msg1 db "Введіть число: ",10,0
8       msg2 db "Введіть інше число: ",10,0
9       fmtint  db "Сума чисел %ld",10,0
10      fmtscan db "%ld"
11      input_len equ 5
12      segment .bss
13      input1 resq input_len+1
14      input2 resq input_len+1
15      ; сегмент коду
16      segment .text
17      global asm_main  ; глобальна позначка
18      asm_main:
19      push rbp
20      mov rbp,rsp
21
22      mov rax,1  ; 1-write
23      mov rdi,1  ; stdout
24      mov rsi,msg1
25      mov rdx,len1
26      syscall
27
28      mov rax,0        ; введення 1-го числа
29      mov rdi,fmtscan
30      mov rsi,input1
31      call scanf
32
33      mov rax,1  ; 1-write
```

```
34      mov rdi,1  ; stdout
35      mov rsi,msg2
36      mov rdx,len2
37      syscall
38
39      mov rax,0        ; введення 2-го числа
40      mov rdi,fmtscan
41      mov rsi,input2
42      call scanf
43
44      mov rax, qword [input1] ; rax =  qword з input1
45      add rax, qword [input2] ; rax += qword з input2
46
47      dump_regs 1 ; виведення значення регістра
48      dump_mem 2, outmsg1, 1 ; виведення з пам'яті
49
50      ; виведення результату
51      mov rdi,fmtint
52      mov rsi,rax
53      mov rdx,0
54      call printf
55
56      mov rax, 0         ; повернення у програму Сі
57      leave
58      ret
```

```
$ ./1
Введіть число:
6
Введіть інше число:
7
Сума чисел 13
```

У функції асемблера використовується для виведення текстових стрічок як функція Сі `printf`, так і системний виклик `syscall`. Для введення даних з консолі використовується функція Сі `scanf`.

Для написання функцій асемблера, які викликаються із Сі програм, можна використовувати наступний шаблон:

```
1       segment .data
2       ;
3       ; місце для розміщення ініціалізованих даних
4       ;
5
6       segment .bss
7       ;
8       ; місце для розміщення неініціалізованих даних
9       ;
10
11      segment .text
12      global _asm_main
13      _asm_main:
14      enter 0,0     ; налаштування входу в підпрограму
15      ;pushaq
```

```
16      ;
17      ; місце для розміщення коду
18      ;
19
20      ;popaq
21      mov rax, 0    ; повернення у Сі програму
22      leave
23      ret
```

5. Домовленості про виклики підпрограм користувача

Домовленості про виклики підпрограм користувача описують як передаються змінні у і з підпрограм. На Linux платформах стандартом такими домовленостями є System V AMD64 ABI.

Аргументи без плаваючої крапки, такі як *цілі числа та адреси*, передаються у підпрограму таким чином:

- 1-й в регістрі rdi;
- 2-й в регістрі rsi;
- 3-й в регістрі rdx;
- 4-й в регістрі rcx;
- 5-й в регістрі r8;
- 6-й в регістрі r9.

Додаткові аргументи (7, 8, 9, 10) передаються через стек у зворотному порядку, щоб їх можна було завантажити у правильному порядку. Наприклад, для 7, 8, 9, 10 аргументів:

- push 10-й аргумент;
- push 9-й аргумент;
- push 8-й аргумент;
- push 7-й аргумент

Тепер у підпрограмі, залишається лише отримати значення з регістрів. Видобуваючи значення зі стеку потрібно пам'ятати, що спочатку у стек поміщаються аргументи. Потім команда call записує адресу повернення (значення `rip` для наступної команди після call) у стек, при цьому `rsp` зменшується на 8 байт, і передає керування підпрограмі. У підпрограмі пролог поміщає (`push`) `rbp` у стек, тому `rsp` зменшується на 8 байт. Потім *вирівнюється стек на 16-байтову межу*, тож, потрібна буде ще одна команда `push` для зменшення `rsp` на 8 байт (4*8+8=40+8=48).

Таким чином, після того, як передані аргументи функції, принаймні два додаткових `qword` поміщено в стек, тобто 16 додаткових байтів (`rbp`, 8 байт вирівнювання). Отже, для отримання аргументів у підпрограмі, потрібно пропустити перші 16 байтів у стеку.

Аргументи з *плаваючою крапкою* передаються через регістри `xmm` наступним чином:

1-й аргумент передається в xmm0;
2-й аргумент передається в xmm1;
3-й аргумент передається в xmm2;
4-й аргумент передається в xmm3;
5-й аргумент передається в xmm4;
6-й аргумент передається в xmm5;

7-й аргумент передається в xmm6;
8-й аргумент передається в xmm7;

Додаткові аргументи передаються через стек, але без використання команди `push`.

Є й інші домовленості про виклики підпрограм.

Домовленості про виклики підпрограм `pascal`

Ключове слово `pascal` у шаблонах і визначеннях функцій вказує, що компілятор використовує домовленості мови Pascal:

- аргументи передаються у стек зліва направо;
- значення, яке повертається передається через змінюваний параметр `Result`. Параметр `Result` створюється неявно і передається першим аргументом функції;
- стек очищає викликувана підпрограма.

Домовленості про виклики підпрограм `stdcall`, які використовує ОС Windows для виклику функцій WinAPI Сі, також вказує на домовленості виклику мови Pascal:

- аргументи передаються у стек справа наліво;
- стек очищає викликувана підпрограма.

Домовленості про виклики підпрограм `cdecl`

Ключове слово c-decl (c-declaration) у шаблонах і визначеннях функцій вказує, що компілятор використовує домовленості мови Сі:

- аргументи передаються у стек справа наліво;
- аргументи розміром менше 4-х байт, розширюються до 4-х байт;
- стек очищає викликаюча програма.

Це основний спосіб виклику функцій із змінним числом аргументів (наприклад, `printf()`). Значення всіх внутрішніх типів (`char, int, enum` та інші) розміром 1, 2, 4 байти функція повертає через регістр `eax`. Значення з розміром менше 4 байти, розширюється до 4-х байтів при записуванні у регістр `eax`. Значення розміром 8 байт повертається через пару регістрів `edx:eax`. Вказівники на типи даних з розміром більше 8 байт (наприклад структури), повертаються через регістр `eax`. Числа з плаваючою крапкою повертаються через регістр `st0`.

Домовленості про виклики підпрограм `fastcall`:

- параметри передаються через регістри;
- якщо для передачі параметрів недостатньо регістрів, то використовується стек.

Домовленості `fastcall` не стандартизовані, тому вони використовуються функціями, які програма не експортує.

В компіляторах фірми Borland для домовленості `__fastcall (register)` параметри передаються зліва направо в регістрах `eax, edx, ecx`, а якщо параметрів більше – через стек. Початкове значення вказівника стеку `esp` повертає викликувана підпрограма.

У 32-розрядній версії компілятора фірми Microsoft, а також GCC, домовленість `__fastcall, __msfastcall` передає перші два параметри зліва направо у регістрах `ecx` і `edx`, а решта параметрів передається у стек справа наліво. Стек очищує викликувана підпрограма.

Домовленості про виклики `this call` використовують компілятори мови C++ при виклику методів класів в ООП. Аргументи функції передаються через стек справа наліво. Стек очищує викликувана підпрограма. Вказівник на об'єкт для якого викликається метод (вказівник `this`) записується в регістр `ecx`.

Домовленості про виклики для платформи AMD64 (x86-64):

- перші 4-параметри передаються через регістри `RCX` або `XMM0`, `RDX` або `XMM1`, `R8` або `XMM2`, `R9` або `XMM3`;

- ціле число (8, 16, 32, 64, _m64) передається через регістри `RCX, RDX, R8, R9`;

- число з плаваючою крапкою передається у регістрах `XMM0-XMM3`;

- інші типи, включаючи структури та `_m128` передаються через вказівник у регістрах `RCX, RDX, R8, R9`;

- результат повертається (для чисел 8, 16, 32, 64, _m64 та вказівників) у регістрі `RAX`;

- результат повертається (для усіх інших типів `float, double, _m128`) у регістрі `XMM0`.

5.1. Особливості виклику підпрограм у мові Сі

З мови Сі асемблер може викликатися як підпрограма або як вбудований (inline) асемблер. Виклик підпрограм асемблера стандартизований. Вбудований асемблер дозволяє вставляти інструкції асемблера безпосередньо у код Сі. У такому випадку асемблерний код має бути записаний у форматі відповідного компілятора Сі. Так компілятори Borland і Microsoft використовують формат MASM, а компілятор Linux GCC – формат GAS.

Більшість домовленостей виклику з мови Сі є специфіковані, але деякі аспекти з них необхідно розглянути.

Збереження регістрів. Домовленості мови Сі припускають, що значення регістрів `rbx, rsi, rdi, rbp, cs, ds, ss, es` використовуються підпрограмою. Підпрограма може їх змінити, але вона повинна відновити їх початковий стан перед поверненням з підпрограми. Значення регістрів `rbx, rsi` і `rdi` використовуються компілятором Сі для зберігання регістрових змінних (оголошених з ключовим словом `register`), тому вони повинні бути не модифікованими. Звичайно для збереження значень цих регістрів використовується стек.

Позначки функцій. Більшість компіляторів Сі ставить символ "_" на початку імен функцій або глобальних/статичних змінних. Компілятор Linux GCC не використовує додаткового символу на початку імен.

Передача параметрів. У домовленості виклику підпрограм з мови Сі прийнято, що аргументи функції заносяться у стек у зворотному порядку (справа наліво). Інструкція мови Сі `printf("x=%d\n",x)` буде скомпільована у наступний Nasm код:

```
1     segment .data
2     x      dq 0
3     format db "x = %d\n", 0
4
5     segment .text
6     ...
7     push qword [x]    ; занесення у стек значення x
8     push qword format ; занесення у стек адреси формату стрічки
9     call _printf
10    add rsp, 16        ; вилучення параметрів зі стеку
```

Регістри	Стек
rsp+24	Значення параметру x
rsp+16	Адреса формату стрічки
rsp+8	Адреса повернення

```
rsp          | Збережене ebp
```

Мова Сі підтримує функції зі змінним числом параметрів, наприклад `printf()`, `scanf()`. Домовленості виклику підпрограм з мови Сі були змінені таким чином, щоб дозволити використання таких функцій. Так як адреса формату стрічки заноситься у стек останньою після параметрів, то вона завжди матиме адресу `rsp+8`, незалежно від кількості переданих функції параметрів. Код `printf` може продивитися стрічку формату і визначити, скільки було передано параметрів і тоді прочитати їх зі стеку.

Обчислення адрес локальних змінних. При передачі локальних змінних або параметрів у функцію, виникає необхідність обчислення їх адрес. Наприклад, потрібно передати у функцію адресу локальної змінної `x`, яка розміщена у стеку за адресою `esp-8`. Але цю адресу потрібно обчислити (адреса має бути значенням, а не виразом), що можна зробити такою інструкцією:

```
mov rax,rsp-8
```

Для обчислення адрес є спеціальна інструкція `lea` (load effective address – завантаження ефективної адреси). Адресу локальної змінної `x` можна обчислити так:

```
lea rax,[rsp-8]
```

Виглядає так, ніби інструкція читає значення з `[rsp-8]`. Але це не так, інструкція `lea` ніколи не читає пам'яті. Вона тільки обчислює адресу, яка може бути зчитана іншою інструкцією і записує її в перший регістровий операнд.

Підтримка декількох домовленостей про виклики підпрограм. Деякі компілятори підтримують декілька домовленостей про виклики підпрограм, наприклад GCC. Домовленості про виклики функції можуть бути явно вказані розширенням `__atribute__`, наприклад:

```
void f(int) attribute ((cdecl ));
```

GCC також підтримує домовленості *стандартного виклику*. Для цього `cdecl` потрібно замінити на `stdcall`. Різниця між `stdcall` і `cdecl` в тому, що `stdcall` вимагає щоб підпрограма вилучала параметри зі стеку як прийнято в мові Pascal. Тому `stdcall` використовується тільки з функціями у яких постійне, а не змінне число аргументів.

GCC також підтримує додатковий атрибут `regparm`, який вказує компілятору передати до трьох аргументів функції через регістри, а не через стек.

Borland і Microsoft використовують ключові слова `__cdecl` і `__stdcall` як модифікатори функцій у їх прототипах, наприклад:

```
void __cdecl f(int);
```

Перевага домовленості `cdecl` в тому, що вона проста і гнучка, а також може застосовуватися до любого типу функцій і компіляторів мови Сі. Недолік цієї домовленості в тому, що вона повільна, порівняно з іншими і потребує більше пам'яті (так як вилучає параметри зі стеку).

Перевага `stdcall` в тому, що вона використовує менше пам'яті порівняно із `cdecl` і не потребує очищення стеку після інструкції `call`. Основний її недолік в тому, що вона не може використовувати функції зі змінним число аргументів.

6. Реентерабельні і рекурсивні підпрограми

Реентерабельна підпрограма повинна мати наступні властивості:

• Не повинна модифікувати код своїх інструкцій, наприклад:

```
mov word [cs:$+7], 5  ;скопіювати 5 у комірку word з адресою cs:$+7 байт
add rax, 2            ; попередня інструкція замінить 2 на 5!
```

Такий код буде працювати у реальному режимі, а в захищеному режимі програма завершиться з помилкою, так як код сегменту позначений тільки для читання.

• Не повинна модифікувати глобальні дані (у секціях `.data` і `.bss`). Всі змінні мають зберігатися у стеку.

Реентерабельні підпрограми мають наступні переваги над звичайними:

• реентерабельні підпрограми можна викликати рекурсивно;

• реентерабельні підпрограми можуть використовуватися багатьма процесами;

• якщо в багатозадачних операційних системах виконується декілька екземплярів програми, то тільки одна копія коду знаходиться у пам'яті. Ця ідея реалізована в спільно використовуваних бібліотеках і `dll` (dynamic link libraries);

• реентерабельні підпрограми працюють набагато краще у багатопотокових програмах.

Рекурсивні програми можуть викликати самі себе. Рекурсія може бути пряма або непряма. При прямій рекурсії підпрограма, наприклад `f`, викликає саму себе у своєму коді. При непрямій рекурсія, підпрограма `f` викликає підпрограму `g`, яка викликає підпрограму `f`.

Рекурсивна підпрограма повинна мати умову завершення рекурсії. При виконанні цієї умови рекурсія завершується. При відсутності умови завершення рекурсія зациклиться і підпрограма аварійно завершиться при переповненні стеку.

Приклад рекурсивної функції і стану її стеку:

```
1     fact:
2     push rbp
3     mov rbp,rsp
4     mov rax, [rbp+16]  ; rax = n
5     cmp rax, 0
6     jne not_zero     ; умова завершення if rax=0
7     mov rax,1
8     jmp return
9     not_zero:
10    dec rax
11    push rax
12    call fact        ; rax = fact(n-1)
13    add rsp,8
14    mov rbx, [rbp+16]
15    mul rbx
16    return:
17    mov rsp,rbp
18    pop rbp
19    ret
```

	Стек
	n(3)
N=3, фрейм	Адреса повернення
	Збережене rbp
	n(2)
N=2, фрейм	Адреса повернення

	Збережене rbp
	n(1)
N=1, фрейм	Адреса повернення
	Збережене rbp

Контрольні запитання.

1. Особливості процесу у захищеному режимі.
2. Передача параметрів командного рядка процесу.
3. Призначення системних викликів.
4. Передача аргументів системним викликам.
5. Системні виклики і коди повернення помилок.
6. Типові системні виклики.
7. Системні виклики для роботи з файлами.
8. Режими використання і права доступу до файлів.
9. Способи виклику підпрограми.
10. Призначення інструкцій `call, ret`.
11. Способи передачі параметрів у підпрограму.
12. Як змінюється стек при виклику і поверненні з підпрограми.
13. Стан стеку при виклику підпрограми зі змінним числом аргументів.
14. Збереження локальних змінних підпрограми у стеку. Інструкції `enter` і `leave`.
15. Виклик підпрограм асемблера із мови Сі.
16. Домовленості про виклики підпрограм.
17. Особливості виклику підпрограм у мові Сі.
18. Реентерабельні і рекурсивні підпрограми.

8. СПІВПРОЦЕСОР

Мета. Вивчення архітектури, програмної моделі і команд співпроцесора.

Вступ. Для чого потрібний співпроцесор, які можливості додає він до того, що робить основний процесор, окрім обробки ще одного формату даних? Перерахуємо деякі з них.

• Повна підтримка стандартів IEEE-754 і 854 на арифметику з плаваючою крапкою. Ці стандарти описують як формати даних, з якими повинен працювати співпроцесор, так і набір реалізованих функцій.

• Підтримка числових алгоритмів для обчислення значень тригонометричних функцій, логарифмів і т. п. Ця робота співпроцесора виконується абсолютно прозоро для програміста, що саме по собі дуже цінно, оскільки не вимагає від нього розробки відповідних підпрограм.

• Обробка десяткових чисел з точністю до 18 розрядів, що дозволяє співпроцесору без округлення виконувати арифметичні операції над цілими десятковими числами зі значеннями до 10^{18}.

• Обробка дійсних чисел з діапазону $3{,}37\times10^{-4932}...1{,}18\times10^{+4932}$.

План.

1. Архітектура співпроцесора
2. Регістр стану SWR
3. Регістр керування CWR
4. Регістр тегів TWR
5. Формати даних
   - 5.1. Двійкові цілі числа
   - 5.2. Запаковані цілі десяткові числа
   - 5.3. Дійсні числа
   - 5.4. Спеціальні числові значення
     - 5.4.1. Денормалізовані дійсні числа
     - 5.4.2. Нуль
     - 5.4.3. Нескінченість
     - 5.4.4. Нечисла
     - 5.4.5. Непідтримувані формати
6. Система команд співпроцесора
   - 6.1. Команди передачі даних
   - 6.2. Команди завантаження констант
   - 6.3. Команди порівняння даних
   - 6.4. Арифметичні команди
     - 6.4.1. Цілочисельні арифметичні команди
     - 6.4.2. Дійсні арифметичні команди
     - 6.4.3. Команди трансцендентних функцій
     - 6.4.4. Додаткові арифметичні команди
     - 6.4.5. Команди керування співпроцесором

1. Архітектура співпроцесора

З точки зору програміста, співпроцесор є сукупністю регістрів, кожний з яких має своє функціональне призначення (рис. 1).

Рисунок 1 – Програмна модель співпроцесора

У програмній моделі співпроцесора можна виділити три групи регістрів. Вісім регістрів R0...R7 складають основу програмної моделі співпроцесора – *стек співпроцесора*. Розмір кожного регістра – 80 бітів. Така організація характерна для пристроїв, що спеціалізуються на обробленні обчислювальних алгоритмів. Реалізація числових алгоритмів на основі стеку співпроцесора дозволяє отримати істотний виграш у швидкості обчислень.

Три службові регістри:

• *регістр стану співпроцесора* SWR (Status Word Register) відображає інформацію про поточний стан співпроцесора і містить поля, що дозволяють визначити, який регістр є поточною верхівкою стеку співпроцесора, які винятки виникли після виконання останньої команди, які особливості виконання останньої команди (деякий аналог регістра прапорів основного процесора) і т. д.;

• *регістр контролю співпроцесора* CWR (Control Word Register), який керує режимами роботи співпроцесора; за допомогою полів в цьому регістрі можна регулювати точність виконання числових обчислень, керувати округленням, маскувати винятки;

• *регістр слова тегів* TWR (Tags Word Register) використовується для контролю за станом кожного з регістрів R0...R7 (команди співпроцесора використовують цей регістр, наприклад, для того, щоб визначити можливість запису значень у вказані регістри).

Два *регістри вказівники даних* DPR (Data Point Register) і *команд* IPR (Instruction Point Register) – призначені для запам'ятовування інформації про адресу команди, що викликала виняткову ситуацію, і адресу її операнда.

Ці вказівники використовуються при обробленні виняткових ситуацій (але не для усіх команд).

Усі ці регістри є програмно доступними. Проте до одних з них доступ отримати досить легко, для цього в системі команд співпроцесора існують спеціальні команди, а до інших його отримати складніше, оскільки спеціальних команд для цього немає, тому необхідно виконувати додаткові дії.

Розглянемо загальну логіку роботи співпроцесора і детальніше схарактеризуємо вказані регістри.

Регістровий стек співпроцесора організований за принципом кільця. Це означає, що серед усіх регістрів, утворюючих стек, немає такого, який є верхівкою стеку. Усі регістри стеку з функціональної точки зору абсолютно рівноправні. Але в стеку завжди має бути верхівка і вона дійсно є, але вона плаваюча. Контроль поточної верхівки здійснюється апаратно за допомогою трирозрядного поля TOP регістра SWR (рис. 2).

У полі TOP фіксується номер регістра стеку 0...7 (R0...R7), який є поточною верхівкою.

Регістр стану SWR

b	c3	top		c2	c1	co	es	sf	pe	ue	oe	ze	de	ie
15	14	13	11	10	9	8	7	6	5	4	3	2	1	0

Рисунок 2 – Формат регістра стану співпроцесора SWR

Команди співпроцесора не оперують фізичними номерами регістрів стеку R0...R7. Замість цього вони використовують логічні номери цих регістрів ST(0)... ST(7). За допомогою логічних номерів реалізується відносна адресація регістрів стеку співпроцесора. Якщо поточною верхівкою стеку є фізичний регістр R0, то після запису чергового значення в стек співпроцесора його поточною верхівкою стане фізичний регістр R7 (рис. 3, *а*). На рис. 3, *б* показаний приклад, коли поточною верхівкою до запису в стек є фізичний регістр R3, а після запису в стек поточною верхівкою стає фізичний регістр стеку R2. Тобто у міру запису в стек вказівник його верхівки рухається у напрямку до молодших номерів фізичних регістрів (зменшується на одиницю). Що стосується логічних номерів регістрів стеку ST(0)...ST(1), то, як видно з рисунка, вони «плавають» разом зі зміною поточної верхівки стеку. Таким чином, реалізується принцип кільця.

На перший погляд, така організація стеку здається дивною. Але, як виявилося, вона має велику гнучкість. Це добре видно на прикладі передачі параметрів підпрограмі. Для підвищення гнучкості підпрограм (у розробленні і використанні) небажано прив'язувати їх за параметрами, що передаються до апаратних ресурсів (фізичних номерів регістрів співпроцесора). Набагато зручніше задавати порядок параметрів, що передаються як логічні номери, оскільки такий спосіб передавання однозначний і не вимагає від розробника знання зайвих подробиць про варіанти апаратної реалізації співпроцесора. Логічна нумерація регістрів співпроцесора, підтримувана на рівні системи команд, ідеально реалізує цю ідею. При цьому не має значення, в який фізичний регістр стеку співпроцесора були поміщені дані перед викликом підпрограми, визначальним є тільки порядок слідування параметрів в стеку. З цієї причини підпрограмі досить знати не місце, а тільки порядок розміщення параметрів, що передаються в стеку.

Перше ніж приступити до опису команд і даних, з якими працює співпроцесор, зазначимо, яким чином «уживаються» між собою ці два різні обчислювальні пристрої – процесор і співпроцесор. Кожен з них має свої несумісні одна з одною системи команд і формати

оброблюваних даних. Незважаючи на те що співпроцесор архітектурно є окремим обчислювальним пристроєм, він не може існувати окремо від основного процесора. Перші моделі процесорів Intel (i8086, i286, i386) і співпроцесорів (відповідно, i8087, i287, i387) виконувалися як окремі пристрої – співпроцесор при необхідності вставлявся в спеціальний роз'єм на системній платі. Починаючи з моделі i486 співпроцесор і основний процесор виготовляються в одному корпусі і є фізично неділимими, тобто архітектурно це два різні пристрої, а апаратно – один.

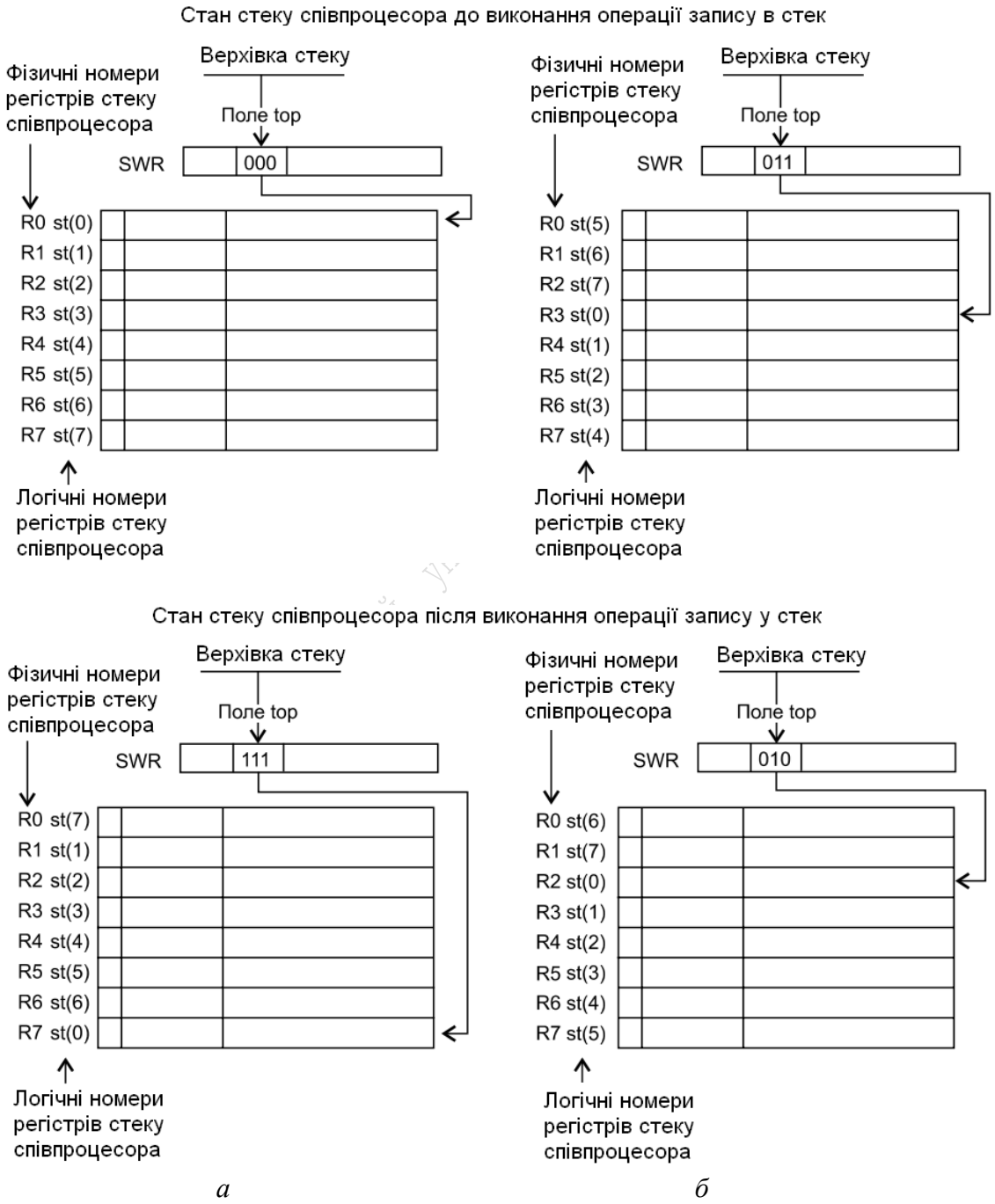


Рисунок 3 – Фізична і логічна нумерація регістрів стеку співпроцесора

Процесор і співпроцесор, як два самостійні обчислювальні пристрої, можуть працювати паралельно. Але цей паралелізм стосується тільки їх внутрішньої роботи над виконанням

чергової команди. Як реалізований цей паралелізм, в чому його суть і особливості? Обидва процесори підключені до загальної системної шини і мають доступ до однакової інформації. Ініціює процес вибірки чергової команди завжди основний процесор. Після вибірки команда потрапляє одночасно в обидва процесори. Будь-яка команда співпроцесора має код операції, перші 5 бітів якого мають значення 11011. Коли код операції починається цими бітами, то основний процесор за подальшим вмістом коду операції з'ясовує, чи вимагає ця команда звернення до пам’яті. Якщо це так, то основний процесор формує фізичну адресу операнда і звертається до пам’яті, після чого вміст комірки пам’яті виставляється на шину даних. Якщо звернення до пам’яті не вимагається, то основний процесор закінчує роботу над цією командою (не роблячи спроби її виконання!) і приступає до декодуванню наступної команди з поточного вхідного командного потоку. Що ж до співпроцесора, то вибрана команда потрапляє до нього і до основного процесора одночасно. Співпроцесор, визначивши з перших п'яти біт, що чергова команда належить його системі команд, починає її виконання. Якщо команда вимагає операнда в пам’яті, то співпроцесор звертається до шини даних за читанням вмісту елементу пам’яті, який до цього моменту має бути наданий основним процесором. З цієї схеми взаємодії виходить, що в певних випадках необхідно погоджувати роботу обох пристроїв. Наприклад, якщо у вхідному потоці зразу за командою співпроцесора йде команда основного процесора, що використовує результати роботи попередньої команди, то співпроцесор не встигне виконати свою команду за той час, поки основний процесор, пропустивши команду співпроцесора, виконує свою. Очевидно, що логіка роботи програми порушується. Можлива і інша ситуація. Якщо вхідний потік команд містить послідовність з декількох команд співпроцесора, то, очевидно, що процесор, на відміну від співпроцесора, перевірить їх дуже швидко, чого він не повинен робити, оскільки забезпечує зовнішній інтерфейс для співпроцесора. Ці і інші складніші ситуації призводять до необхідності синхронізувати між собою роботу двох процесорів. У перших моделях процесорів подібна синхронізація виконувалася програмістом «вручну» шляхом вставки перед або після кожної команди співпроцесора спеціальної команди WAIT або FWAIT. Робота цієї команди полягала в призупиненні роботи основного процесора до тих пір, поки співпроцесор не закінчить роботу над останньою командою. Починаючи з моделі процесора i486, подібна синхронізація виконується командами WAIT/FWAIT, які введені в алгоритм роботи більшості команд співпроцесора. Але деякі команди з групи команд керування співпроцесором мають два варіанти реалізації — з синхронізацією (очікуванням) і без неї.

З усього сказаного можна зробити важливий висновок: використання співпроцесора є абсолютно прозорим для програміста. У загальному випадку програмістові слід сприймати співпроцесор як набір додаткових регістрів, для роботи з якими призначені спеціальні команди. Для ефективного застосування співпроцесора програміст повинен добре розібратися в структурі регістрів і логіці їх використання. Тому перше ніж приступити до розгляду команд і даних, з якими працює співпроцесор, приведемо опис структури деяких регістрів співпроцесора.

2. Регістр стану SWR

Регістр SWR відображає поточний стан співпроцесора після виконання останньої команди. Структурно регістр SWR складається з наступних полів (див. рис. 2):

• Шість прапорів виняткових ситуацій.

• Біт SF (Stack Fault) – помилка роботи стеку співпроцесора. Біт встановлюється в одиницю, якщо виникає одна з трьох виняткових ситуацій PE, UE або IE. Зокрема, його

встановлення інформує про спробу запису в заповнений стек або, навпаки, спробі читання з порожнього стеку. Після аналізу цього біту, його потрібно знову встановити в нуль разом з бітами PE, UE або IE (якщо вони були встановлені).

• Біт ES (Error Summary) сигналізує про сумарну помилку в роботі співпроцесора. Біт встановлюється в одиницю, якщо виникає будь-яка з шести виняткових ситуацій, про які буде вказано далі.

• Чотири біти C0...C3 (Condition Code) коду умови. Призначення цих бітів аналогічно прапорам в регістрі RFLAGS основного процесора – вони відображають результат виконання останньої команди співпроцесора.

• Трирозрядне поле TOP містить вказівник регістра поточної верхівки стеку.

Майже половину регістра SWR займають біти (прапори) реєстрації виняткових ситуацій. *Виняткова ситуація* – особливий тип переривань. Переривання, підтримувані процесором Intel, за місцем їх виникнення класифікуються на зовнішні і внутрішні. Внутрішні переривання виникають в ході роботи поточної програми і діляться на синхронні (по команді `int`) і асинхронні, такі, що називаються *винятками*, або *особливими випадками*. Таким чином, винятки – це різновид переривань, за допомогою яких процесор інформує програму про деякі особливості її реального виконання. Співпроцесор також має здатність збудження подібних переривань при виникненні певних ситуацій (не обов'язково помилкових). Усі можливі винятки зведені до шести типів, кожному з яких відповідає один біт в регістрі SWR. Програмістові зовсім не обов'язково писати обробник для реакції на ситуацію, що привела до деякого винятку. Співпроцесор уміє самостійно реагувати на багато з них. Це так звана обробка винятків за замовчуванням. Для того, щоб заборонити співпроцесору обробку певного типу винятку за замовчуванням, необхідно цей виняток замаскувати. Така дія виконується шляхом встановлення в одиницю потрібного біту в регістрі керуючого співпроцесора CWR (рис. 4). Типи винятків, що фіксуються за допомогою регістра SWR :

• IE (Invalide operation Error) – недійсна операція;

• DE (Denormalized operand Error) – денормалізований операнд;

• ZE (divide by Zero Error) – помилка ділення на нуль;

• OE (Overflow Error) – помилка переповнювання (виникає у разі виходу порядку числа за максимально допустимий діапазон);

• UE(Underflow Error) – помилка антипереповнення (виникає, коли результат занадто малий);

• PE(Precision Error) – помилка точності (встановлюється, коли співпроцесору доводиться округляти результат через те, що його точне подання неможливе, наприклад 10/3).

При виникненні будь-якого з цих шести типів винятків встановлюється в одиницю відповідний біт в регістрі SWR незалежно від того, чи був замаскований цей виняток в регістрі CWR чи ні.

3. Регістр керування CWR

Регістр керування співпроцесором CWR визначає особливості обробки числових даних (рис. 4). Він складається:

• з шести масок винятків;

• поля керування точністю PC (Precision Control);

• поля керування округленням RC (Rounding Control).

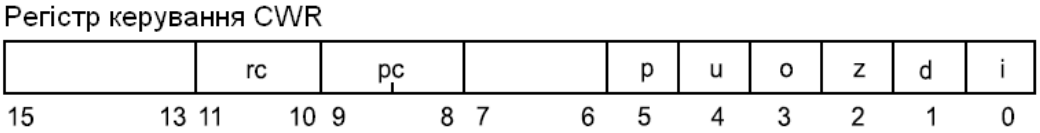


Рисунок 4 – Формат регістра керування співпроцесором CWR

Шість масок призначені для маскування виняткових ситуацій, виникнення яких фіксується за допомогою шести бітів регістра SWR. Якщо якісь біти винятків в регістрі CWR встановлені в одиницю, це означає, що відповідні винятки оброблятимуться самим співпроцесором. Якщо для якого-небудь винятку у відповідному біті масок винятків регістра CWR міститься нульове значення, то при виникненні винятку цього типу буде збуджено переривання 16(10h). Операційна система повинна містити (чи програміст повинен написати) обробник цього переривання. Він повинен з'ясувати причину переривання, після чого, якщо це необхідно, усунути її, а також виконати інші дії.

Поле керування точністю PC призначене для вибору довжини мантиси. Можливі значення в цьому полі означають:

• PC = 00 – довжина мантиси 24 біти;

• PC = 10 – довжина мантиси 53 біти;

• PC = 11 – довжина мантиси 64 біти.

За умовчанням встановлюється значення поля PC = 11.

Поле RC дозволяє керувати процесом округлення чисел в ході роботи співпроцесора. Необхідність округлення може виникнути у ситуації, коли після виконання чергової команди співпроцесора отримується результат, який не можна задати без округлення, наприклад періодичний дріб 3,333... . Встановивши одне із значень в полі RC, можна виконати округлення в необхідну сторону. Для того, щоб вияснити характер округлення, введемо позначення:

• *m* – значення в ST(0) або результат роботи деякої команди, який не може бути точно поданий і тому має бути округлений;

• *a* і *b* – найбільш близькі значення до значення *m*, які можуть бути подані в регістрі ST(0) співпроцесора, причому виконується умова $a < m < b$.

Далі приведені значення поля RC і описаний характер відповідних округлень:

• 00 – значення *m* округляється до найближчого числа *a* або b;

• 01 – значення *m* округляється у меншу сторону, тобто $m = a$;

• 10 – значення *m* округляється у більшу сторону, тобто $m = b$;

• 11 – відкидається дробова частини *m* (може використовуватися в операціях цілочисельної арифметики).

4. Регістр тегів TWR

Регістр тегів TWR є сукупністю дворозрядних полів. Кожне дворозрядне поле відповідає певному фізичному регістру стека (див. рис. 1) і характеризує його поточний стан. Зміна стану будь-якого регістра стеку відображається на вмісті відповідного цьому регістру поля регістра тега. Можливі наступні значення в полях регістра тегу :

• 00 – регістр стеку співпроцесора зайнятий допустимим ненульовим значенням;

• 01 – регістр стеку співпроцесора містить нульове значення;

• 10 – регістр стеку співпроцесора містить одне із спеціальних числових значень, за винятком нуля;

• 11 – регістр порожній, і в нього можна робити запис (треба відмітити, що це значення в одному з дворозрядних полів регістра тегів не означає, що усі біти відповідного регістра стеку обов'язково нульові).

При написанні програми розробник працює не з абсолютними, а з відносними номерами регістрів стеку. З цієї причини у нього можуть виникнути труднощі при спробі інтерпретації вмісту регістра тегів TWR з відповідними фізичними регістрами стека. Як зв'язуючу ланку необхідно використовувати інформацію з поля TOP регістра SWR.

5. Формати даних

Співпроцесор розширює номенклатуру форматів даних, з якими працює основний процесор. У цьому немає нічого незвичайного, оскільки формат даних будь-якого пристрою в істотній мірі відбиває специфіку його роботи. Співпроцесор спеціально розроблявся для обчислень з плаваючою крапкою. Але співпроцесор може працювати і з цілими числами, хоча і менш ефективно.

Формати даних, з якими працює співпроцесор :

• двійкові цілі числа в трьох форматах – 16, 32 і 64 біти;

• запаковані цілі десяткові (BCD) числа – довжина максимального числа складає 18 запакованих десяткових цифр (9 байтів);

• дійсні числа в трьох форматах – короткому (32 біти), довгому (64 біти), розширеному (80 бітів).

Окрім цих основних форматів, співпроцесор підтримує спеціальні числові значення, до яких відносяться :

• денормалізовані дійсні числа – це числа, менші мінімального нормалізованого числа (див. нижче) для кожного дійсного формату, підтримуваного співпроцесором;

• нуль;

• позитивні і негативні значення нескінченності;

• нечисла;

• невизначеності і непідтримувані формати.

Розглянемо детальніше основні формати даних, підтримувані співпроцесором. Важливо зазначити, що в самому співпроцесорі числа в цих форматах мають однакове внутрішнє подання – розширений формат дійсного числа. Це один з форматів подання дійсних чисел, який точно відповідає формату регістрів R0...R7 стеку співпроцесора (див. рис. 1).

Таким чином, навіть якщо використовуються команди співпроцесора з цілочисловими операндами, то після завантаження в співпроцесор операндів цілого типу вони автоматично перетворяться у формат розширеного дійсного числа.

5.1. Двійкові цілі числа

Співпроцесор працює з трьома типами цілих чисел (рис. 5).

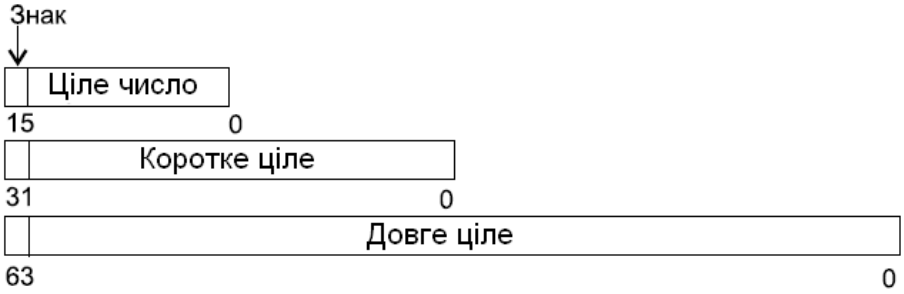


Рисунок 5 – Формати цілих чисел співпроцесора

У табл. 1 подані формат цілих чисел, їх розмірність і діапазон значень.

Таблиця 1 – Формати цілих чисел співпроцесора

Формат	**Розмір, бітів**	**Діапазон значень**
Ціле слово	16	–32 768...+32 767
Коротке ціле	32	$-2 \cdot 10^9 ... +2 \cdot 10^9$
Довге ціле	64	$-9 \cdot 10^{18} ... +9 \cdot 10^{18}$

Вибираючи формат даних, з якими працюватиме програма, необхідно пам'ятати, що співпроцесор підтримує операції з цілими числами, але робота з ними здійснюється неефективно. Причина в тому, що оброблення співпроцесором цілочислових даних буде сповільнено через додаткові перетворення цілих чисел в їх внутрішнє подання у вигляді еквівалентного дійсного числа розширеного формату.

У програмі цілі двійкові числа описуються звичайним способом – з використанням директив `dw`, `dd` і `dq`. Наприклад, ціле число 5 може бути описано таким чином:

```
i  dw 5 ; подання у пам'яті: i=05 00
i  dd 5 ; подання у пам'яті: i=05 00 00 00
i  dq 5 ; подання у пам'яті: i=05 00 00 00 00 00 00 00
```

Працювати з цілими числами може не кожна команда співпроцесора.

5.2. Запаковані цілі десяткові числа

Співпроцесор підтримує один формат запакованих цілих десяткових чисел, або BCD чисел (рис. 6). Директива DT дозволяє описати 20 цифр в запакованому десятковому числі (по дві в кожному байті). Через те, що максимальна довжина запакованого десяткового числа в співпроцесорі складає тільки 9 байт, в регістри R0...R7 можна помістити тільки 18 запакованих десяткових цифр. Старший десятий байт ігнорується. Самий старший біт цього байта використовується для зберігання знаку числа.

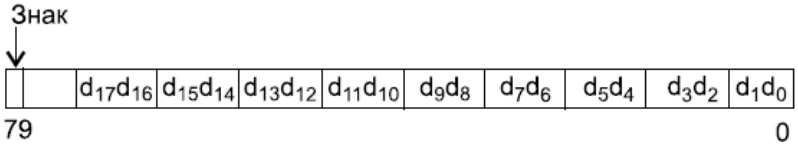


Рисунок 6 – Формат десяткового числа співпроцесора

Запаковані десяткові числа також подаються в стеку співпроцесора у розширеному форматі. Наприклад, ціле число 5 365 904 у форматі запакованого десяткового числа можна описати таким чином:

```
i dt 5365904
;подання в пам'яті: i=04 59 36 05 00 00 00 00 00 00
```

Треба зазначити, що в співпроцесорі є всього дві команди для роботи з запакованими десятковими числами — це команди збереження і завантаження.

5.3. Дійсні числа

Основний тип даних, з яким працює співпроцесор – дійсний. Дані цього типу описуються трьома форматами: звичайним (коротким), подвоєним (довгим) і розширеним (рис. 7).

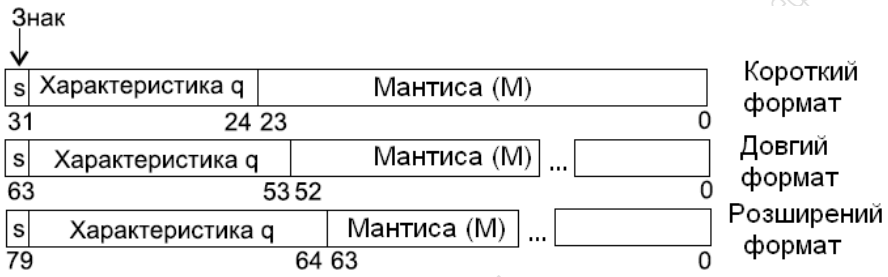


Рисунок 7 – Формати дійсних чисел співпроцесора

Для подання дійсного числа використовується наступна формула:

$$A = (\pm M) \cdot N^{\pm(p)}, \qquad (1)$$

тут:

• M – мантиса числа A (мантиса повинна задовольняти умові | M |< 1);

• N – основа системи числення, подана цілим позитивним числом;

• p – порядок числа, що показує істинне положення крапки в розрядах мантиси (з цієї причини дійсні числа мають ще назву чисел з плаваючою крапкою, оскільки її положення в розрядах мантиси залежить від значення порядку).

Для зручності оброблення в процесорі чисел з плаваючою крапкою його архітектурою накладаються деякі обмеження на компоненти формули (1).

Далі перераховано ці умови і обмеження для співпроцесорів архітектури IA 32.

• Основа системи числення $N = 2$.

• Мантиса M має бути подана в *нормалізованому* виді. Нормалізація може відрізнятися для різних типів процесорів. Для архітектури співпроцесора IA 32 нормалізованим є число :

$$A = (-1)^s \cdot N^q \cdot M, \qquad (2)$$

тут:

• s – значення знакового розряду (0 – число більше нуля, 1 – число менше нуля);

• q – характеристика числа, її призначення аналогічне призначенню порядку p у формулі (1), але як пояснюється далі, p і q – не одно і теж.

У формулі (1) знак мають і порядок і мантиса дійсного числа, а у формулі (2) – тільки характеристика. А де ж зберігається знак порядку?

У співпроцесорі Intel на апаратному рівні прийнята угода, що порядок *p* визначається у форматі дійсного числа особливим значенням *q*, що називається *характеристикою.* Величина *q* пов'язана з порядком *p* за допомогою наступної формули і є деякою константою (умовно назвемо її фіксованим зміщенням):

$$q = p + \text{фіксоване зміщення} \quad (3)$$

Для кожного з трьох можливих форматів дійсних чисел *зміщення* має різне, але фіксоване для конкретного формату значення, яке залежить від кількості розрядів, що відводяться під характеристику (табл. 2).

Таблиця 2 – Формати дійсних чисел

Формат	Короткий	Довгий	Розширений
Довжина числа(біти)	32	64	80
Розмірність мантиси *M*	24	53	64
Діапазон значень	$10^{-38}...10^{+38}$	$10^{-308}...10^{+308}$	$10^{-4932}...10^{+4932}$
Розмірність характеристики *q*	8	11	15
Значення фіксованого зміщення	+127	+1023	+16 383
Діапазон характеристики *q*	0...255	0...2047	0...32 767
Діапазон порядків *p*	–126...+127	–1022...+1023	–16 382...+16 383

У таблиці показані діапазони значень характеристик *q* і істинних порядків *p* дійсних чисел, що відповідають їм. Зазначимо, що нульовому порядку дійсного числа в короткому форматі відповідає значення характеристики рівне 127, якому в двійковому поданні відповідає значення 01_11_11_11. Негативному порядку *p*, наприклад -1, відповідатиме характеристика *q* = -1 + 127 = 126, або в двійковому виді – 01_11_11_10. Позитивному порядку *p*, наприклад +1, відповідатиме характеристика *q* = 1 + 127 = 128, або в двійковому виді – 10_00_00_00. Тобто усі позитивні порядки мають в двійковому поданні характеристики старший біт рівний одиниці, а негативні порядки – ні. Таким чином, знак порядку «схований» в старшому біті характеристики.

Оскільки нормалізоване дійсне число завжди має цілу одиничну частину (за винятком перерахованих раніше спеціальних числових значень), то при його поданні у пам'яті з'являється можливість вважати перший розряд дійсного числа одиничним за замовчуванням і враховувати його наявність тільки на апаратному рівні. Це дає можливість збільшити діапазон поданих чисел, оскільки з'являється зайвий розряд, придатний для задання мантиси числа. Але це стосується тільки для короткого і довгого форматів дійсних чисел. Розширений формат як внутрішній формат подання числа будь-якого типу в співпроцесорі містить цілу одиничну частину дійсного числа в явному виді.

Як визначити дійсне число або зарезервувати місце для його розміщення в програмі на асемблері?

Коротке дійсне 32-розрядне число визначається директивою `dd`. При цьому обов'язковим у запису числа є десяткова крапка, навіть якщо воно не має дробової частини. Для транслятора десяткова крапка є вказівкою, що число треба подати як число з плаваючою крапкою в

короткому форматі (див. рис. 7). Це ж стосується довгого і розширеного форматів подання дійсних чисел, які визначаються директивами `dq` і `dt`.

Інший спосіб визначення дійсного числа директивами `dd`, `dq` і `dt` – експоненційна форма з використанням символу «e».

Приклад визначення дійсного числа 45,56 в короткому форматі:

```
dd 45.56
dd 45.56e0
dd 0.4556e2
```

У пам' яті це число буде виглядати так 71 3d 36 42. Враховуючи, що в архітектурі Intel прийнятий порядок слідування байтів в пам'яті відповідно до принципу «молодший байт за молодшою адресою», істинне подання числа 45,56 буде наступним : 42 36 3d 71. Двійкове подання в пам'яті числа 45,56 ілюструє рис. 9. З рисунка видно, що старша одиниця мантиси при поданні у пам'яті відсутня.

Рис. 9. Двійкове подання в пам’яті дійсного числа в директиві DD

Визначимо тепер дійсне число 45,56 в довгому форматі. Це можна зробити двома способами:

```
dq 45.56
dq 45.56e0
```

У пам'яті це число виглядатиме так: `47 e1 7a 14 ae c7 46 40`.

Перевернувши його, отримаємо істинне значення: `40 46 c7 ae 14 7a e1 47`.

Нарешті, визначимо в програмі дійсне число 45,56 в розширеному форматі: `dt 45.56`.

У пам'яті це число виглядатиме так: `71 3d 0a d7 a3 70 3d b6 04 40`.

Перевернувши його, отримаємо істинне значення в пам'яті: `40 04 b6 3d 70 a3 d7 0a 3d 71`.

Двійкове подання числа 45,56 показано на рис. 10.

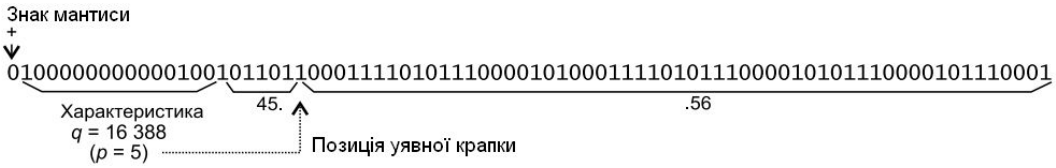


Рис. 10. Двійкове подання в пам’яті дійсного числа в директиві DT

Як видно, в мантисі явно присутня старша одиниця, чого не було в короткому і довгому форматах подання дійсного числа.

5.4. Спеціальні числові значення

Незважаючи на великий діапазон дійсних значень, які подаються в регістрах стеку співпроцесора, зрозуміло, що нескінченні значення знаходяться за рамками цього діапазону. Для того, щоб мати можливість реагувати на обчислювальні ситуації, в яких виникають такі значення, в співпроцесорі передбачені спеціальні комбінації бітів, що називаються спеціальними числовими значеннями. При необхідності програміст може сам кодувати спеціальні числові значення, оскільки дійсні числа, описані директивою dt, відповідні команди співпроцесора завантажують без їх всяких перетворень.

5.4.1. Денормалізовані дійсні числа

Денормалізовані дійсні числа – це числа, які менші мінімального нормалізованого числа для кожного дійсного формату. Пояснимо природу денормалізованих чисел з використанням числової шкали. Наприклад, для дійсного числа в розширеному форматі діапазон подаваних значень в співпроцесорі на числовій шкалі виглядатиме так, як показано на рис. 11.

Як відомо, співпроцесор зберігає числа в нормалізованому виді. У міру наближення чисел до нуля йому все важче «витягати» їх значення до нормалізованого виду, тобто до такого виду, щоб першою значущою цифрою мантиси була одиниця. Розмірність розрядної сітки, відведеної у форматах дійсних чисел співпроцесора для представлення характеристики, не безмежна.

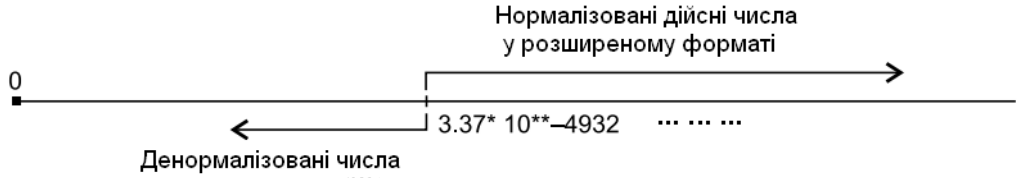


Рисунок 11 – Положення денормалізованих дійсних чисел на числовій шкалі

Тому при певних значеннях числа в розширеному форматі значення характеристики стає рівним нулю (рис. 11). Але насправді число відмінне від нуля, тобто це «не справжній» числовий нуль. Таким чином, між істинним нулем і мінімально уявним нормалізованим числом є ще нескінченна кількість дуже маленьких чисел. Це і є так звані денормалізовані числа. Вони мають нульовий порядок і ненульову мантису. Діапазон поданих в співпроцесорі денормалізованих чисел не безмежний, оскільки кількість розрядів мантиси обмежено (рис. 12).

Співпроцесор реагує на появу денормалізованих чисел генеруванням винятків. При формуванні денормалізованого значення в деякому регістрі стека у відповідному цьому регістрі тегу регістра TWR формується спеціальне значення (10).

5.4.2. Нуль

Нуль також відносять до спеціальних числових значень. Це робиться через те, що це значення особливо виділяється серед коректних дійсних значень, що формуються як результат роботи деякої команди. Більше того, нуль може формуватися як реакція співпроцесора на певну обчислювальну ситуацію.

Значення істинного нуля може мати знак (рис. 12), що, втім, не впливає на його сприйняття командами співпроцесора. Для визначення знаку нуля використовується команда FXAM. В результаті роботи цієї команди у біт C1 регістра SWR заноситься знак операнду. При завантаженні нуля в регістр стека у відповідному тегу регістра TWR формується спеціальне значення (01).

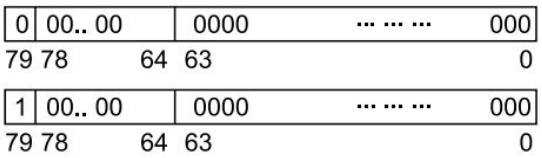


Рисунок 12 – Подання нуля в регістрі стеку співпроцесора

Значення нуля може бути сформоване в результаті виникнення ситуації антипереповнення, а також при роботі команд з нульовими операндами.

5.4.3. Нескінченність

Співпроцесор має засоби у вигляді спеціальних бітових значень для подання нескінченності. Формат регістра стеку співпроцесора, що містить значення нескінченності, показаний на рис. 13.

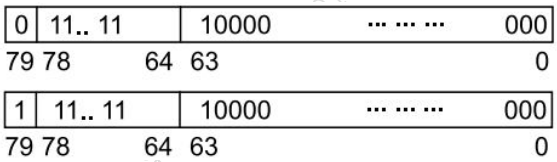


Рисунок 13 – Подання значення нескінченності в регістрі стека співпроцесора

З рис. 13 видно, що значення нескінченності може мати знак, при цьому значення мантиси і характеристики фіксовані. Саме у цьому відмінність значення нескінченності від інших спеціальних значень.

Серед причин, що призводять до формування значення нескінченності, можна виділити переповнювання і ділення на нуль. При формуванні значення нескінченності в деякому регістрі стека у відповідному тегу регістра TWR формується спеціальне значення (10).

5.4.4. Нечисла

До нечисел відносяться такі бітові послідовності в регістрі стеку співпроцесора, які не співпадають ні з одним з розглянутих раніше форматів значень. Нечисло повинне мати одиничну мантису і будь-яку характеристику, окрім 100...00, яка зарезервована для значення нескінченність. Розрізняють два типи нечисел :

• SNAN (Signaling Non a Number) – сигнальні нечисла;

• QNAN (Quiet Non A Number) – спокійні (тихі) нечисла.

Сигнальне нечисло – бітове значення з одиничним значенням полів характеристики і мантисою, перший біт якої, що слідує за першим одиничним значущим бітом, дорівнює нулю (рис. 14, а). Співпроцесор реагує на появу цього числа в регістрі стека збудженням винятку недійсної операції. Програмісти можуть формувати ці числа в регістрі стека співпроцесора

спеціально, наприклад, щоб штучно збудити в потрібній ситуації вказане виключення. Очевидно, що саме з цієї причини ці числа називаються сигнальними. Якщо зняти маску у прапора недійсної операції в регістрі CWR, то буде викликаний обробник, який виконає задані програмістом дії.

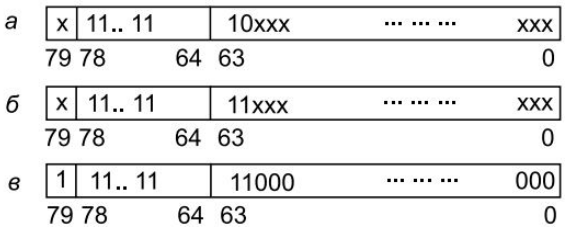


Рисунок 14 – Подання нечисел в регістрі стека співпроцесора

Спокійне нечисло – бітове значення з одиничним значенням полів характеристики і мантисою, перші два біти якої дорівнюють одиниці (рис. 14, б).

Співпроцесор самостійно не формує сигнальні числа, але як реакції на певні винятки він може формувати спокійні нечисла, наприклад нечисло дійсної невизначеності (рис. 14, в). Дійсна невизначеність формується як маскована реакція співпроцесора на виняток недійсної операції. Інші спокійні нечисла можуть формуватися після виконання команд, в яких хоч би один з операндів був спокійним нечислом.

Це може породити «ланцюгову реакцію», що веде до помилкового результату. Тому в процесі обчислень рекомендується періодично контролювати результати виконання команд на предмет появи спокійних нечисел.

При формуванні нечисла в деякому регістрі стеку у відповідному тегу регістра TWR формується спеціальне значення (10).

5.4.5. Непідтримувані формати

Необхідно мати на увазі, що окрім розглянутих існує досить багато бітових наборів, які можна подати в розширеному форматі дійсного числа. Для більшості їх значень формується виняток недійсної операції.

6. Система команд співпроцесора

Система команд співпроцесора включає близько 80 машинних команд. Розглянемо їх класифікацію (рис. 15).

Мнемонічне позначення команд співпроцесора характеризує особливості їх роботи і у зв'язку з цим може являти певний інтерес. Тому коротко розглянемо основні моменти утворення назв команд.

• Усі мнемонічні позначення розпочинаються з символу F (Float).

• Друга буква мнемонічного позначення визначає тип операнду в пам'яті з яким працює команда:

• I – ціле двійкове число;

• B – ціле десяткове число;

• відсутність букви – дійсне число.

• Остання буква P в мнемонічному позначенні команди означає, що останньою дією команди обов'язковим є видобування операнду із стека.

• Остання або передостання буква R(reversed) в мнемонічному позначенні команди означає реверсивне слідування операндів при виконанні команд віднімання і ділення, оскільки для них важливий порядок слідування операндів.

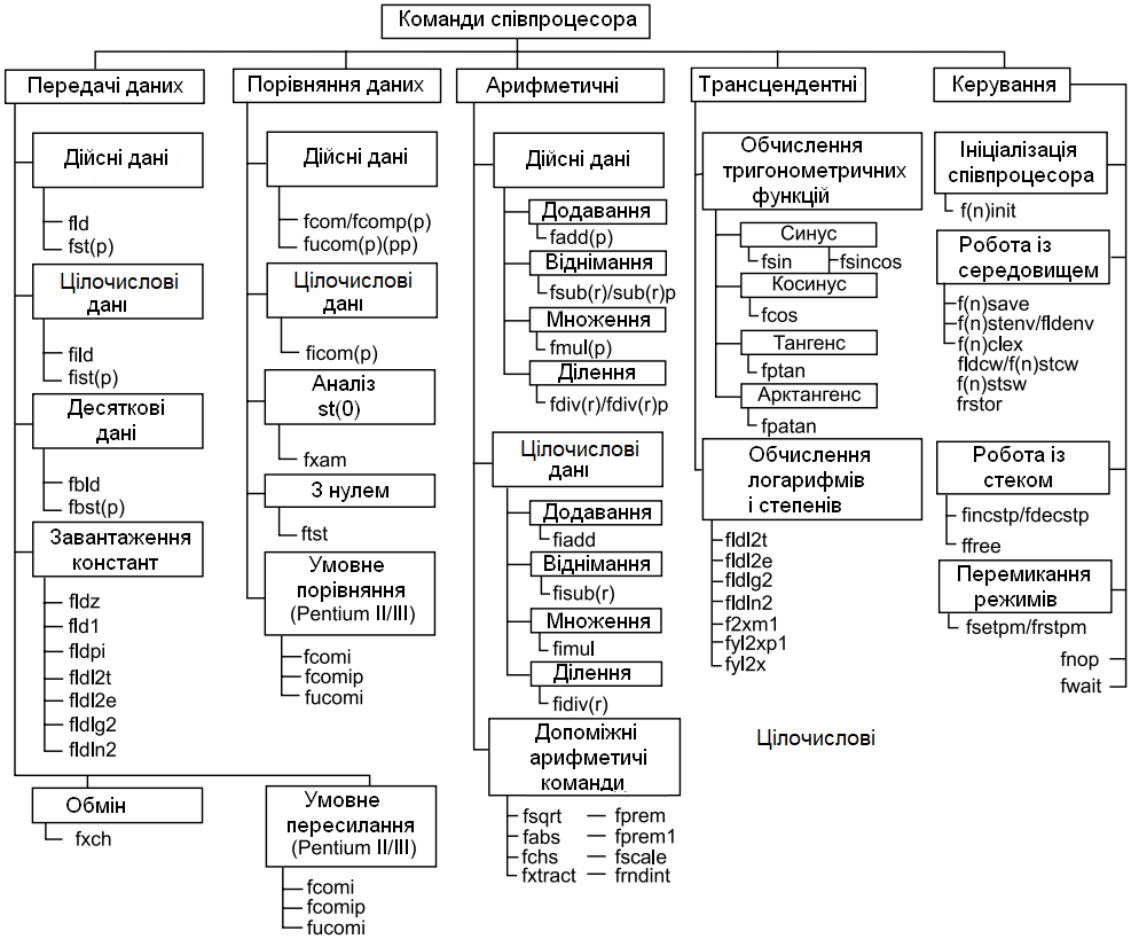


Рисунок 15. Функціональна класифікація команд співпроцесора

Корисною може виявитися і інформація про машинні формати команд співпроцесора. Якщо давати їм загальну характеристику, то важливо зазначити, що в цілому система команд співпроцесора відрізняється великою гнучкістю у виборі варіантів задання команд, що реалізовують певну операцію, і їх операндів. Мінімальна довжина команди співпроцесора – 2 байти.

Методика написання програм для співпроцесора має свої особливості. Головна причина тут – в стековій організації співпроцесора. Для того, щоб написати програму для обчислення деякого виразу, його необхідно заздалегідь перетворити в зручний для програмування співпроцесора вид.

Процес перетворення нагадує підготовку виразу для методу трансляції з використанням постфіксного запису.

Для отримання постфіксного запису будується дерево виразу, у якому вузли відповідають операціям, а листки – операндам. Початком обходу буде листок найлівішої гілки дерева. Тоді для отримання постфіксного запису виразу необхідно рухатися по дереву зліва направо, при

цьому вузли мають бути видимими тільки після обходу усіх гілок, що виходять з нього. Так дерево виразу a + b · c – d / (a + b) зображено на рис. 16, а постфіксний запис матиме вид abc × + dab + / –.

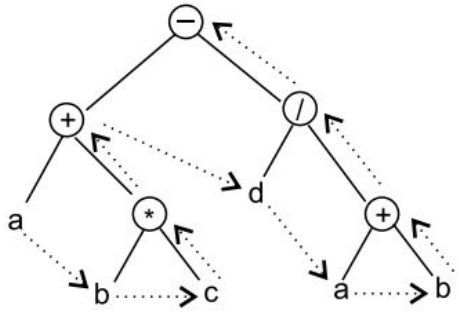


Рисунок 16 – Подання виразу у вигляді дерева

Постфіксний запис дозволяє обчислювати вирази за один прохід з урахуванням пріоритету арифметичних операцій.

Алгоритм обчислення виразів в постфіксному записі:

1. Вибрати черговий символ запису виразу у постфіксній формі.

2. Якщо черговий вибраний символ – операнд, то помістити його в стек, після чого повернутися до кроку 1.

3. Якщо черговий вибраний символ – знак операції, то виконати її з одним чи двома операндами на верхівці стеку. Результат операції необхідно помістити назад на верхівку стеку.

4. Якщо в початковому записі виразу у постфіксій формі ще залишилися символи, то повернутися до кроку 1, інакше – на верхівці стеку отримано результат обчислення виразу.

Приклад обчислення виразу a*b + c*d:

```
 a dq 1.2
 b dq 3.4
 c dq 5.6
 d dq 7.8
 res dq 0
...
finit
fld  qword [a]      ; a   -> st(0) TOP=7
fmul qword [b]      ; a*b -> St(0) TOP=7
fld  qword [c]      ; c   -> st(0) TOP=6
fmul qword [d]      ; c*d -> st(0) TOP=6
fadd                ; a*b + c*d -> st(0)
fstp qword [res]    ; st(0) -> res
```

6.1. Команди передачі даних

Група команд передачі даних призначена для організації обміну між регістрами стеку, верхівкою стеку співпроцесора і елементами оперативної пам'яті.

Команди цієї групи мають таке ж значення для програмування співпроцесора, як команда MOV – для програмування основного процесора. За допомогою команд передачі даних здійснюються усі переміщення значень операндів в співпроцесор і з нього. З цієї причини для кожного з трьох типів даних, з якими може працювати співпроцесор, існує своя підгрупа команд передачі даних. Власне, на цьому рівні усі його уміння по роботі з різними форматами даних і

закінчуються. Головною функцією усіх команд завантаження даних в співпроцесор є перетворення даних до єдиного подання як дійсного числа розширеного формату. Це ж стосується і зворотної операції – збереження в пам'яті даних із співпроцесора.

Команди передачі даних можна розділити на наступні групи:

• команди передачі даних в дійсному форматі;

• команди передачі даних в цілочисловому форматі;

• команди передачі даних в десятковому форматі.

Далі перераховані команди передачі даних в дійсному форматі.

• `FLD джерело` – завантаження дійсного числа з області пам'яті на верхівку стеку співпроцесора.

• `FST приймач` – збереження дійсного числа з верхівки стеку співпроцесора в пам'ять. Як випливає з аналізу мнемокоду команди (відсутній символ P), збереження числа в пам'яті не супроводжується виштовхуванням його із стека, тобто поточна верхівка стеку співпроцесора не міняється (поле TOP не міняється).

• `FSTP приймач` – збереження дійсного числа з верхівки стеку співпроцесора в пам'ять. На відміну від попередньої команди, у кінці мнемонічного позначення цієї команди є присутній символ P, що означає виштовхування дійсного числа із стеку після його збереження в пам'яті. Команда змінює поле TOP, збільшуючи його на одиницю. Внаслідок цього верхівкою стеку стає наступний більший за своїм фізичним номером регістр стеку співпроцесора.

Далі перераховані команди передачі даних в цілочисловому форматі.

• `FILD джерело` – завантаження цілого числа з пам'яті на верхівку стеку співпроцесора.

• `FIST приймач` – збереження цілого числа з верхівки стеку співпроцесора в пам'ять. Збереження цілого числа в пам'яті не супроводжується виштовхуванням його із стека, тобто поточна вершина стека співпроцесора не змінюється.

• `FISTP приймач` – збереження цілого числа з верхівки стеку в пам'ять. Аналогічно сказаному раніше про команду FSTP, останньою дією команди є виштовхування числа із стеку з одночасним перетворенням його в ціле значення.

Команди передачі даних в десятковому форматі.

• `FBLD джерело` – завантаження десяткового числа з пам'яті на верхівку стеку співпроцесора.

• `FBSTP приймач` – збереження десяткового числа з верхівки стеку співпроцесора в області пам'яті. Значення виштовхується із стеку після перетворення його у формат десяткового числа. Зауважимо, що для десяткових чисел немає команди збереження значення в пам'яті без виштовхування із стеку.

До групи команд передачі даних можна віднести також команду обміну верхівки регістрового стеку ST(0) з будь-яким іншим регістром стеку співпроцесора ST(i): `fxch st(i)`.

Дію команд завантаження `FLD`, `FILD` і `FBLD` можна порівняти з командою `PUSH` основного процесора. Аналогічно до неї (`PUSH` зменшує значення в регістрі SP) команди завантаження співпроцесора перед збереженням значення в регістровому стеку співпроцесора віднімають з утримуваного поля TOP регістра стану SWR одиницю. Це означає, що верхівкою стеку стає регістр з фізичним номером *на одиницю менше*. При цьому можливе переповнювання стеку. Оскільки стек співпроцесора складається з обмеженого числа регістрів, то в нього може бути записано максимум вісім значень. Через кільцеву організацію стеку дев'яте записуване значення затирає перше. Програма повинна мати можливість обробити таку ситуацію. З цієї

причини майже усі команди, що поміщають свій операнд в стек співпроцесора, після зменшення значення поля TOP перевіряють регістр – кандидат на нову верхівку стеку – на предмет його зайнятості. Для аналізу цієї і подібних ситуацій використовується регістр TWR, що містить слово тегів (див. рис. 1). Наявність регістра тегів в архітектурі співпроцесора дозволяє звільнити програміста від розробки складної процедури розпізнавання вмісту регістрів співпроцесора і дає самому співпроцесору можливість фіксувати певні ситуації, наприклад спробу читання з порожнього регістра або запис в не порожній регістр. Виникнення таких ситуацій фіксується в регістрі стану SWR (див. рис. 2), призначеному для збереження загальної інформації про співпроцесор. Використовуючи спеціальні команди співпроцесора, можна видобути з нього або, навпаки, записати в нього інформацію.

6.2. Команди завантаження констант

Основним призначенням співпроцесора є підтримка обчислень з плаваючою крапкою. У математичних обчисленнях досить часто зустрічаються наперед задані константи, і співпроцесор зберігає значення деяких з них. Інша причина використання цих констант полягає в тому, що для визначення їх в пам'яті (у розширеному форматі) потрібно 10 байт, а це для зберігання, наприклад, одиниці затратно (сама команда завантаження константи, що зберігається в співпроцесорі, займає два байти). У форматі, відмінному від розширеного, ці константи зберігати не має сенсу, оскільки втрачається час на їх перетворення в той же розширений формат. Для кожної наперед заданої константи існує спеціальна команда, яка завантажує її на верхівку регістрового стеку співпроцесора:

- `FLDZ` – завантаження нуля;
- `FLD1` – завантаження одиниці;
- `FLDPI` – завантаження числа π;
- `FLDL2T` – завантаження двійкового логарифму десяти;
- `FLDL2E` – завантаження двійкового логарифму експоненти (числа e);
- `FLDLG2` – завантаження десяткового логарифму двійки;
- `FLDLN2` – завантаження натурального логарифму двійки.

6.3. Команди порівняння даних

Команди порівняння даних порівнюють значення числа на верхівці стеку і операнду, вказаного в команді.

- `FCOM [операнд_в_пам'яті]` – команда без операндів порівнює два значення: одне знаходиться в регістрі ST(0), інше в регістрі ST(1). Якщо вказаний операнд [операнд_в_пам'яті], то порівнюється значення в регістрі ST(0) стеку співпроцесора зі значенням в пам'яті.
- `FCOMP операнд` – команда порівнює значення на верхівці стеку співпроцесора ST(0) зі значенням операнда, який знаходиться в регістрі або в пам'яті. Останньою дією команди є виштовхування значення з ST(0).
- `FCOMPP операнд` – команда аналогічна по дії команді FCOM без операндів, але останньою її дією є виштовхування із стеку значень обох регістрів, ST(0) і ST(1).
- `FICOM операнд_в_пам'яті` – команда порівнює значення на верхівці стеку співпроцесора ST(0) з цілим операндом в пам'яті. Довжина цілого операнду – 16 або 32 біти, тобто це ціле слово і коротке ціле (див. таблицю. 1).

• `FICOMP операнд` – команда порівнює значення на верхівці стеку співпроцесора ST(0) з цілим операндом в пам'яті. Після порівняння і встановлення бітів C3...C0 команда виштовхує значення з ST(0). Довжина цілого операнду – 16 або 32 біта, тобто це ціле слово і коротке ціле (див. табл. 1).

• `FTST` – команда не має операндів і порівнює значення в ST(0) з нулем(значенням 00).

Попередні команди порівняння працюють коректно, якщо операнди в них є цілими або дійсними числами. Коли один з операндів виявляється нечислом, то фіксується виняток недійсної ситуації, а коди умови C3. . .C0 відповідають винятковій ситуації непорівнянних або неупорядкованих операндів. Сама ж дія порівняння не виконується. Процесор надає три команди, що дозволяють все ж зробити порівняння таких операндів, але як дійсних чисел без врахування їх порядків.

• `FUCOM st(i)` – команда порівнює значення (без врахування їх порядків) в регістрах стеку співпроцесора ST(0) і ST(i).

• `FUCOMP st(i)` – команда порівнює значення (без врахування їх порядків) в регістрах стеку співпроцесора ST(0) і ST(i). Останньою дією команди є виштовхування значення з верхівки стеку.

• `FUCOMPP st(i)` – команда порівнює значення (без врахування їх порядків) в регістрах стеку співпроцесора ST(0) і ST(i). Останні дві дії команди однакові – виштовхування значення з верхівки стеку.

В результаті роботи команд порівняння в регістрі стану встановлюються наступні значення бітів коду умови C3, C2, C0 :

• якщо ST(0) > операнда, то 000;

• якщо ST(0) < операнда, то 001;

• якщо ST(0) = операнда, то 100;

• якщо операнди невпорядковані, то 111.

Для того, щоб отримати можливість реагувати на ці коди командами умовного переходу основного процесора (вони реагують на прапори в RFLAGS), треба якось записати сформовані біти умови C3, C2, C0 в регістр RFLAGS. У системі команд співпроцесора існує команда `FSTSW`, яка дозволяє запам'ятати слово стану співпроцесора в регістрі AX або в комірці пам'яті. Далі значення потрібних бітів видобуваються і аналізуються командами основного процесора. Наприклад, запис старшого байта слова стану, в якому знаходяться біти C0. . .C3, в молодший байт регістра RFLAGS здійснюється командою `SAHF`. Ця команда записує вміст AH в молодший байт регістра RFLAGS. Після цього біт C0 записується на місце прапора CF, C2 – на місце PF, C3 – на місце ZF. Біт C1 випадає із загального правила, оскільки в регістрі прапорів на місці того, що відповідає цьому біту знаходиться одиниця. Аналіз цього біту треба проводити за допомогою логічних команд основного процесора. Знаючи усе це, залишається, виходячи з особливостей алгоритму, застосовувати ті команди умовного переходу, які аналізують стан вказаних прапорів.

До групи команд порівняння даних логічно віднести і команду `FXAM`, яка аналізує операнд на верхівці стеку співпроцесора ST(0) і формує значення бітів C0, C1, C2, C3 в регістрі стану співпроцесора SWR. За станом цих бітів можна судити про:

• знак мантиси – знаковий біт операнду в ST(0) заноситься у біт C0 регістра SWR;

• коректність запису дійсного числа в ST(0) – ідентифікуються порожній регістр, коректне дійсне число, нечисло і невідомий формат;

• тип спеціального числового значення: нескінченність, нуль, денормалізований операнд.

6.4. Арифметичні команди

Команди співпроцесора, що входять до групи арифметичних команд, реалізують чотири основні арифметичні операції – додавання, віднімання, множення і ділення. Є також декілька додаткових команд, призначених для підвищення ефективності використання основних арифметичних команд.

З точки зору типів операндів арифметичні команди співпроцесора можна розділити на команди, працюючі з дійсними і цілими числами.

6.4.1. Цілочислові арифметичні команди

Цілочислові арифметичні команди призначені для роботи в тих місцях обчислювальних алгоритмів, де як початкові дані використовуються цілі числа в пам'яті у форматі слово і коротке слово, що мають розмір 16 і 32 біта.

Цілочислові арифметичні команди забезпечують велику гнучкість задання операндів.

• `FIADD джерело` – команда додає значення ST(0) і цілочислове джерело, яким виступає 16- або 32-розрядний операнд в пам'яті. Результат додавання запам'ятовується в регістрі стеку співпроцесора ST(0).

• `FISUB джерело` – команда віднімає значення цілочислового джерела з ST(0). Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисловий операнд в пам'яті.

• `FIMUL джерело` – команда множить значення цілочислового джерела на вміст ST(0). Результат множення запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисловий операнд в пам'яті.

• `FIDIV джерело` – команда ділить вміст ST(0) на значення цілочислового джерела. Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(0).

Джерелом виступає 16- або 32-розрядний цілочисловий операнд в пам'яті.

Для команд, що реалізують арифметичні дії ділення і віднімання, важливий порядок розміщення операндів. З цієї причини система команд співпроцесора містить відповідні реверсивні команди, що підвищують зручність програмування обчислювальних алгоритмів. Щоб відрізнити ці команди від звичайних команд ділення і віднімання, їх мнемокоди закінчуються символом R.

• `FISUBR джерело` – команда віднімає значення ST(0) від цілочислового джерела. Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисловий операнд в пам'яті.

• `FIDIVR джерело` – команда ділить значення цілочислового джерела на вміст ST(0). Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисловий операнд в пам'яті.

6.4.2. Дійсні арифметичні команди

Схема розміщення операндів дійсних команд традиційна для команд співпроцесора. Один з операндів розміщується на верхівці стеку співпроцесора – регістрі ST(0), куди після виконання команди записується і результат, а другий операнд може бути розміщений або в пам'яті, або в іншому регістрі стеку співпроцесора. Допустимими типами операндів в пам'яті є усі перераховані раніше дійсні формати за винятком розширеного.

На відміну від цілочислових арифметичних команд, дійсні арифметичні команди допускають більшу різноманітність у поєднанні місця розміщення операндів і самих команд для виконання конкретної арифметичної дії. Так, наприклад, можна виділити три можливі варіанти команди додавання. На додаток до цих трьох варіантів існує ще одна команда додавання, що виконує додаткову дію, – вилучення значення із стеку.

• `FADD` – команда додає значення в ST(0) і ST(1). Результат додавання запам’ятовується в регістрі стеку співпроцесора ST(0).

• `FADD джерело` – команда додає значення ST(0) і джерела, що задає адресу елемента пам'яті. Результат додавання запам'ятовується в регістрі стеку співпроцесора ST(0).

• `FADD st(i), st` – команда додає значення в регістрі стеку співпроцесора ST(i) із значенням на верхівці стеку ST(0). Результат додавання запам'ятовується в регістрі ST(i).

• `FADDP st(i), st` – команда додає дійсні операнди аналогічно команді `FADD st(i), st`, проте останньою дією команди є виштовхування значення з верхівки стеку співпроцесора ST(0). Результат додавання залишається в регістрі ST(i-1).

Для виконання операції віднімання також є великий набір команд.

• `FSUB` – команда віднімає значення в ST(1) від значення в ST(0). Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0).

• `FSUB джерело` – команда віднімає значення джерела від значення в ST(0). Джерело задає адресу елемента пам'яті, що містить допустиме дійсне число. Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0).

• `FSUB st(i), st` – команда віднімає значення на верхівці стеку ST(0) від значення в регістрі стеку співпроцесора ST(i). Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(i).

• `FSUBP st(i), st` – команда віднімає дійсні операнди аналогічно команді `FSUB st(i), st`. Останньою дією команди є виштовхування значення з верхівки стеку співпроцесора ST(0). Результат віднімання залишається в регістрі ST(i-1).

Для зручності група команд віднімання дійсних чисел доповнена командами реверсивного віднімання.

• `FSUBR st(i), st` – команда віднімає значення на верхівці стеку ST(0) від значення в регістрі стеку співпроцесора ST(i). Результат віднімання запам'ятовується у верхівці стеку співпроцесора – регістрі ST(0).

• `FSUBRP st(i), st` – команда віднімає подібно до команди `FSUBR st(i), st`. Останньою дією команди є виштовхування значення з верхівки стеку співпроцесора ST(0). Результат віднімання залишається в регістрі ST(i-1).

В командах множення дійсних операндів, операнди розміщуються винятково в стеку співпроцесора.

• `FMUL` – команда не має операндів. Множить значення в ST(0) на вміст в ST(1). Результат множення запам'ятовується в регістрі стеку співпроцесора ST(0).

• FMUL st(0),st(i) – команда множить значення в ST(0) на вміст регістра стеку ST(i). Результат множення запам'ятовується в регістрі стеку співпроцесора ST(0).

• FMUL st(i), st(0) – команда множить значення в ST(i) на вміст регістра стеку ST(0). Результат множення запам'ятовується в регістрі стеку співпроцесора ST(i).

• FMULP st(i), st(0) – команда множить подібно до команди FMUL st(i), st. Останньою дією команди є виштовхування значення з верхівки стеку співпроцесора ST(0). Результат множення залишається в регістрі ST(i-1).

Команди ділення дійсних даних. Подібно до команд множення, операнди цих команд розміщуються в стеку співпроцесора:

• FDIV – команда (без операндів) ділить вміст регістра ST(1) на значення регістра співпроцесора ST(0). Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(0).

• FDIV st(0), st(i) – команда ділить вміст регістра ST(0) на вміст регістра співпроцесора ST(i). Результат ділення запам'ятовується в регістрі стеку співпроцесора st(0).

• FDIV st(i), st(0) – ділить вміст регістра ST(i) на вміст регістра співпроцесора ST(0). Результат ділення запам'ятовується в регістрі стеку співпроцесора st(i).

• FDIVP st(i), st – команда ділить аналогічно команді FDIV st(i), st(0). Останньою дією команди є виштовхування значення з верхівки стеку співпроцесора ST(0). Результат ділення залишається в регістрі ST(i-1). Для реалізації ділення в співпроцесорі також передбачені дві реверсивні команди, особливою ознакою яких є наявність символу в як останнього або передостаннього символу мнемокоду:

• FDIVR st(0), st(i) – команда ділить вміст регістра ST(i) на вміст верхівки регістра співпроцесора ST(0). Результат ділення запам'ятовується в регістрі стек співпроцесора ST(0).

FDIVR st(i), st(0) – команда ділить вміст регістра ST(0) на вміст верхівки регістра співпроцесора ST(i). Результат ділення запам'ятовується в регістрі стек співпроцесора ST(i).

• FDIVRP st(i), st(0) – команда ділить вміст регістра ST(i) на вміст верхівки регістра співпроцесора ST(0). Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(i), після чого виштовхується вміст ST(0) із стека. Результат ділення залишається в регістрі ST(i-1).

6.4.3. Команди трансцендентних функцій

Співпроцесор має ряд команд, призначених для обчислення значень тригонометричних функцій, таких як синус, косинус, тангенс, арктангенс, а також значень логарифмічних і показникових функцій.

Команди трансцендентних функцій (операнди в радіанах).

• FCOS – команда обчислює косинус кута, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат повертається в регістр ST(0).

• FSIN – команда обчислює синус кута, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат повертається в регістр ST(0).

• FSINCOS – команда обчислює синус і косинус кута, що знаходяться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат повертається в регістрах ST(0) і ST(1). При цьому синус поміщається в ST(0), а косинус – в ST(1).

• FPTAN – команда обчислює частковий тангенс кута, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат повертається в регістрах ST(0) і ST(1).

• FPRATAN – команда обчислює частковий арктангенс кута, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат повертається в регістрах ST(0) і ST(1).

6.4.4. Додаткові арифметичні команди

До додаткових арифметичних команд відносяться:

• FSQRT – обчислення квадратного кореня із значення, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат обчислення поміщається в регістр ST(0). Слід зазначити, що ця команда має певні переваги. По перше, результат видобування досить точний, а по друге, швидкість виконання трохи більше швидкості виконання команди ділення дійсних чисел FDIV;

• FABS – обчислення модуля значення, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат обчислення поміщається в регістр ST(0);

• FCHS – зміна знаку значення, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат обчислення поміщається назад в регістр ST(0). Відмінність команди FCHS від команди FABS в тому, що команда FCHS тільки інвертує знаковий розряд значення в регістрі ST(0), не міняючи значення інших бітів. Команда обчислення модуля FABS за наявності негативного значення в регістрі ST(0), разом з інвертуванням знакового розряду, виконує зміну інших бітів значення так, щоб в ST(0) вийшло відповідне позитивне число.

• FXTRACT – команда виділення порядку і мантиси значення, що знаходиться на верхівці стеку співпроцесора – регістрі ST(0). Команда не має операндів. Результат виділення поміщається в два регістри стеку – мантиса в ST(0), а порядок в ST(1). При цьому мантиса подається дійсним числом з тим же знаком, що і у початкового числа, і порядком рівним нулю. Порядок, поміщений в ST(1), подається як істинний порядок, тобто без константи зміщення, у вигляді дійсного числа зі знаком і відповідає величині *p* формули (1).

6.4.5. Команди керування співпроцесором

Остання група команд призначена для загального керування роботою співпроцесора. Команди цієї групи мають особливість – перед початком свого виконання вони не перевіряють наявність незамаскованих винятків. Проте така перевірка може знадобитися, зокрема, для того, щоб при паралельній роботі основного процесора і співпроцесора запобігти руйнуванню інформації, необхідної для коректної обробки винятків, що виникають в співпроцесорі. Тому деякі команди керування мають аналоги, що виконують ті ж дії плюс одну додаткову функцію – перевірку наявності винятку в співпроцесорі. Ці команди мають однакові мнемокоди (і машині коди теж), що розрізняються тільки другим символом – символом *n*:

• мнемокод, що не містить другого символу *n*, означає команду, яка перед початком свого виконання перевіряє наявність незамаскованих винятків;

• мнемокод, що містить другий символ *n*, означає команду, яка перед початком свого виконання не перевіряє наявності незамаскованих винятків, тобто виконується негайно, що дозволяє заощадити декілька машинних тактів.

Як уже згадувалося, ці команди мають однаковий машинний код. Відмінність лише в тому, що перед командами, що не містять символу *n*, транслятор асемблера вставляє команду `WAIT`. Команда `WAIT` є повноцінною командою основного процесора, і її при необхідності можна вказувати явно. Команда `WAIT` має аналог серед команд співпроцесора – `FWAIT`. Обом цим командам відповідає код операції 9bh.

Команди `WAIT/FWAIT` – це команди очікування. Вони призначені для синхронізації роботи процесора і співпроцесора. Оскільки основний процесор і співпроцесор працюють паралельно, то може створитися ситуація, коли за командою співпроцесора, що змінює дані в пам'яті, слідує команда основного процесора, якій ці дані потрібно. Щоб синхронізувати роботу цих команд, необхідно включити між ними команду `WAIT/FWAIT`. Зустрівши цю команду в потоці команд, основний процесор призупинить свою роботу до тих пір, поки не поступить апаратний сигнал про завершення чергової команди в співпроцесорі. Тут є ще один ефект, відмічений раніше. Він торкається коректної обробки винятків і пов'язаної з ними інформації.

З усіх команд управління першою логічно розглянути команду, що переводить співпроцесор в деякий початковий стан, – це команда ініціалізації співпроцесора `FINIT/FNINIT`. Вона ініціалізує керуючі регістри співпроцесора певними значеннями.

• Регістр керування CWR ініціалізується числом 037h, що означає встановлення наступних режимів:

• поле округлення RC = 00 – округлення до найближчого цілого;

• біти 0...5 встановлені в одиницю, що означає маскування усіх винятків;

• поле керування точністю PC = 11 — максимальна точність (64 біта).

• Регістр стану SWR ініціалізується нульовим значенням, що означає відсутність винятків і вказує на те, що фізичний регістр стеку співпроцесора R0 є верхівкою стеку і відповідає логічному регістру ST(0).

• Регістр тегів TWR ініціалізується одиничним значенням – це означає, що усі регістри стеку співпроцесора порожні.

• Регістри вказівників даних DPR і команд IPR ініціалізуються нульовими значеннями.

Цю команду використовують перед першою командою співпроцесора в програмі і в інших випадках, коли необхідно привести співпроцесор в початковий стан.

Наступні дві команди працюють з регістром стану SWR.

• `FSTSW/FNSTSW ax` – команда збереження вмісту регістра стану SWR в регістрі AX. Цю команду доцільно використати для підготовки до умовних переходів за описаною при розгляді команд порівняння схемі.

• `FSTSW/FNSTSW` приймач – команда збереження вмісту регістра стану SWR в комірці пам'яті. Від розглянутої раніше команда відрізняється типом операнду – тепер це комірка пам'яті розміром два байти (відповідно до розміру регістра SWR).

Наступні дві команди, які працюють з інформацією в регістрі керування CWR, підтримують запис і читання вмісту цього регістра.

• `FSTCW/FNSTCW приймач` – команда збереження вмісту регістра керування CWR в комірці пам'яті розміром два байти. Цю команду доцільно використати для аналізу полів маскування винятків, керування точністю і округлення. Слід зазначити, що операндом не є регістр AX, на відміну від команди `FSTSW/FNSTSW`.

• `FLDCW джерело` – команда завантаження значення комірки пам'яті розміром 16 бітів в регістр керування CWR. Ця команда виконує дію, протилежну до дії команди `FSTCW/FNSTCW`.

Команду FLDCW доцільно використати для задання або зміни режиму роботи співпроцесора. Слід зазначити, що якщо в регістрі стану SWR встановлений будь-який біт винятку, то спроба завантаження нового вмісту в регістр керування CWR приведе до збудження винятку. З цієї причини необхідно перед завантаженням регістра CWR скинути усі прапори винятків в регістрі SWR.

Наступна команда – команда без операндів FCLEX/FNCLEX – дозволяє скинути прапори винятків в регістрі стану SWR співпроцесора, які, зокрема, потрібні для коректного виконання команди FLDCW.

Команди, які працюють з вказівником стеку в регістрі SWR.

• FINCSTP – команда збільшення вказівника стеку на одиницю (поле TOP) в регістрі SWR. Команда не має операндів. Дія команди FINCSTP подібна до дії команди FST, але вона видобуває значення операнду із стеку «в нікуди». Таким чином, цю команду можна використати для виштовхування операнду, що став непотрібним, з верхівки стеку. Команда працює тільки з полем TOP і не змінює поле, що відповідає цьому регістру, в регістрі тегів TWR, тобто регістр залишається зайнятим і його вміст із стеку не видобувається.

• FDECSTP – команда зменшення покажчика стеку (поле TOP) в регістрі SWR. Команда не має операндів. Дія команди FDECSTP подібно до дії команди FLD, але вона не поміщає значення операнду в стек. Таким чином, цю команду можна використати для проштовхування всередину стеку операндів, раніше включених в нього. Команда працює тільки з полем TOP і не змінює поле, що відповідає цьому регістру, в регістрі тегів TWR, тобто регістр залишається порожнім.

Наступна команда, FFREE st(i), позначає будь-який регістр стеку співпроцесора як порожній. Це команда звільнення регістра стеку ST(i). Команда записує в поле регістра тегів, що відповідає регістру ST(i), значення 11b, що відповідає порожньому регістру. При цьому вказівник стеку (поле TOP) в регістрі SWR і вміст самого регістра не змінюються. Необхідно зазначити, що ST(i) – це відносний, а не фізичний номер регістра стеку співпроцесора. Необхідність в цій команді може виникнути при спробі запису в регістр ST(i), який помічений як непорожній. В цьому випадку буде збуджено виняток. Для запобігання цього застосовується команда FFREE.

У співпроцесорі є також команда FNOP, яка не виконує ніяких дій і впливає тільки на регістр вказівника команди IPR.

У групі команд управління можна виділити підгрупу команд, працюючих з так званим середовищем співпроцесора. Середовище співпроцесора – це сукупність регістрів співпроцесора і їх значень. Середовище співпроцесора може бути частковим або повним. Про яке саме середовище йде мова, визначається однією з розглянутих далі команд:

• FSAVE/FNSAVE приймач – команда збереження повного стану середовища співпроцесора в пам'ять, адреса якої вказана операндом приймач. Розмір області пам'яті залежить від розміру операнду сегмента коду (use16 або use32) :

• use16 – область пам'яті повинна складати 94 байти: 80 байтів для восьми регістрів із стеку співпроцесора і 14 байтів для інших регістрів співпроцесора з додатковою інформацією;

• use32 – область пам'яті повинна складати 108 байтів: 80 байтів для восьми регістрів із стека співпроцесора і 28 байтів для інших регістрів співпроцесора з додатковою інформацією;

• FRSTOR джерело – команда відновлення повного стану середовища співпроцесора з області пам'яті, адреса якої вказана операндом джерело. Співпроцесор працюватиме в новому середовищі відразу після закінчення роботи команди FRSTOR.

Співпроцесор може працювати не лише в реальному, але і в захищеному режимі. Для цього необхідно виконувати перемикання співпроцесора між цими режимами. Операція перемикання реалізується спеціальними командами.

• `FSETPM` – команда перемикання співпроцесора з реального в захищений режим. Команда не має операндів. Дія команди впливає тільки на виконання команд збереження і відновлення середовища. Для реального і захищеного режимів склад і формат інформації середовища співпроцесора дещо розрізняється.

• `FRSTPM` – команда перемикання співпроцесора із захищеного в реальний режим. Команда не має операндів. Дія команди впливає тільки на виконання команд збереження і відновлення середовища.

Контрольні запитання.

1. Програмна модель співпроцесора.
2. Фізична і логічна нумерація регістрів стеку співпроцесора
3. Регістр стану SWR, регістр керування CWR, регістр тегів TWR.
4. Формати даних співпроцесора.
5. Спеціальні числові значення співпроцесора.
6. Функціональна класифікація команд співпроцесора.
7. Команди передачі даних співпроцесора.
8. Команди порівняння даних співпроцесора.
9. Арифметичні команди співпроцесора.
10. Команди трансцендентних функцій співпроцесора.
11. Команди керування співпроцесором.

9. MMX РОЗШИРЕННЯ

Мета. Вивчення команд MMX-розширення.

Вступ. Команди MMX – це SIMD команди обробки цілочислових даних для використання у мультимедійних застосунках (MultiMedia extensions). Вони призначені для одночасного оброблення декількох елементів даних за одну інструкцію. Розширення MMX з'явилося в процесорах модифікації Pentium MMX.

План.

1. Особливості і передумови технології MMX.
2. Типи даних і команди MMX
   2.1. Призначення команд MMX
3. Особливості реалізації MMX
4. Приклади використання MMX команд

1. Особливості і передумови технології MMX

Абревіатура MMX походить від виразу MultiMedia eXtension – розширення для мультимедіа, яке реалізоване фірмою Intel в 1997 році у своїй новій серії процесорів MMX з тактовою частотою 166 і більше МГц.

Процесор Pentium MMX відрізнявся від "звичайного" Pentium за шістьма основними пунктами:

• додані 57 нових команд оброблення даних;

• збільшений в два рази об'єм внутрішнього кешу (16 Кб для команд і стільки ж – для даних);

• збільшений обсяг буфера адрес переходу (Branch Target Buffer – BTB), використовуваного в системі передбачення переходів (Branch Prediction);

• оптимізована робота конвеєра (Pipeline);

• збільшена кількість буферів запису (Write Buffers);

• введено так зване подвійне електроживлення процесора.

Набір з 57 нових команд і є основною відмінністю; інші пункти – не більше, ніж "супутні зміни". Хоча збільшений об'єм кешу і внутрішніх буферів, оптимізований конвеєр дещо прискорюють роботу будь-яких застосувань, проте основне збільшення продуктивності до 60% – можливо тільки при використанні програм, що правильно застосовують технологію MMX в обробці даних.

Фактично уся історія розвитку комп'ютерів є безперервним змаганням між швидкодією центрального процесора та інших систем – пам'яті і зовнішніх пристроїв. Особливо це видно в системах мультимедіа, де йде обробка звуку і зображення, цифрове подання яких займає великі обсяги пам'яті. Для ефективної обробки звуку і відео при відносно низькій пропускній спроможності системної магістралі (шини) все більша кількість функцій переноситься в апаратуру – модеми, відео і звукові адаптери. Це викликає їх значне подорожчання порівняно із загальною вартістю комп'ютера.

Нездатність комп'ютерів з процесором Pentium ефективно обробляти в реальному часі звук і відео без спеціальних карт залежить вже не стільки від загальної швидкодії процесора або

шини, а від характеру його набору команд обробки даних, відомого під назвою CISC (Common Instruction Set Computer – комп'ютер із загальним набором команд). Цей набір, що складається з відносно складних арифметико-логічних команд, орієнтований на типові завдання обробки даних. Ця ефективна для більшості застосунків архітектура виявляється абсолютно неефективною при швидкісній і специфічній обробці великих масивів даних.

Технологія MMX є компромісним рішенням, що об'єднує підходи технологій RISC (Reduced Instruction Set Computer – комп'ютер із спрощеним набором команд), а також в технологію комп'ютерів з паралельною архітектурою SIMD (Single Instruction, Multiple Data – одна команда, багато даних). Так в класичний процесор Pentium (CISC) додано ряд простих команд (RISC) і команд паралельної обробки даних (SIMD).

Команди MMX, які є першими SIMD цілочисловими командами для процесора з плаваючою крапкою (FPU) x86. Команди MMX використовують 64-бітові MMX регістри, які насправді є псевдонімами для мантис регістрів x87 (але на них не впливала позиція верхівки стеку FPU). Це було зроблено для збереження сумісності з існуючими операційними системами (які зберігали стек FPU під час перемикання контексту). Використання команди MMX і команд з плаваючою крапкою є нетривіальною задачею, оскільки вони не можуть одночасно працювати. На даний час команди MMX рідко використовуються і повністю замінені різними розширеннями SSE. Однак існує ще досить багато працюючого коду з MMX командами.

2. Типи даних і команди MMX

Команди технології MMX працюють з новими типами даних: 64-розрядними цілочисловими даними, а також з даними, запакованими в групи загальною довжиною 64 біти. Такі дані можуть розміщуватися в пам'яті або у восьми 64-розрядних MMX-регістрах MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7. Апаратно це можуть бути різні пристрої, але з точки зору програміста — це одні й ті ж регістри. Таким чином, не можна одночасно користуватися командами математичного співпроцесора і MMX.

Команди MMX-розширення виконуються в тому ж режимі процесора, що і команди з плаваючою крапкою.

При роботі з MMX-командами використовуються регістри стеку математичного співпроцесора R0 – R7. При цьому використовуються лише 64 розряди (мантиса), а стекова організація, яка потрібна для операцій співпроцесора, не використовується. При сумісному використанні математичного співпроцесора і MMX-розширення останньою виконуваною командою MMX-розширення повинна бути команда *emms*. Ця команда забезпечує коректний перехід процесора від виконання фрагмента програмного коду з MMX-командами до оброблення звичайних команд з плаваючою крапкою співпроцесора.

Команди технології MMX працюють з такими типами даних, рис. 1:

• запаковані байти (packed byte) – один 64-розрядний регістр містить запакованих 8 байтів;

• запаковані слова (packed word) – один 64-розрядний регістр містить чотири 16-розрядні слова;

• запаковані подвійні слова (packed doubleword) – один 64-розрядний регістр містить два 32-розрядні слова;

• 64-розрядні слова (quadword).

Кожний елемент із запакованого типу даних є цілим числом з фіксованою крапкою. Користувач контролює розміщення фіксованої крапки для кожного елементу і її розміщення при обчисленнях.

Оброблення даних MMX-розширення може виконуватися одним з двох способів:

• з використанням **циклічної арифметики** (wraparound arithmetic);

• з використанням **арифметики з насиченням** (saturation arithmetic).

Якщо команда використовує циклічну арифметику (wraparound) то результат операції який виходить за двійкову розрядну сітку (overflow або underflow) обрізається (біти carry і overflow ігноруються) і тільки молодші значущі біти записуються у результуючий операнд.

Якщо команда використовує арифметику з насиченням і результат операції більший максимального значення (overload) або менший мінімального значення (underflow) межі значень типу даних, то у результуючий операнд записується максимальна або мінімальна межа (відбувається “насичення”). Наприклад, при додаванні беззнакових 16-розрядних чисел може генеруватися одиничний 17-й біт результату і нульові біти від 0 до 16. Тому в арифметиці з насиченням у цьому випадку 17-й біт відкидається, а у всі інші біти записується значення FFFFh. Це важливо при обробленні візуальних даних, так як чорні кольори раптово можуть стати білими. В MMX технології арифметика з насиченням не є режимом, тобто вона не активується виставлення деякого прапора. Тому деякі команди можуть виконувати як циклічну арифметику, так і арифметику з насиченням.

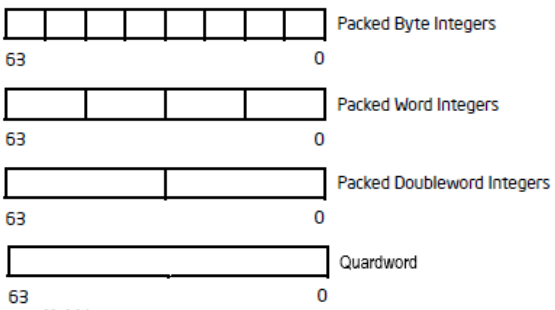


Рисунок 1 – Типи даних MMX

Систему команд MMX утворюють 57 команд, які дозволяють паралельно обробляти декілька елементів даних.

Першою буквою MMX-команд є буква “p”, наприклад paddb.

Більшість команд мають суфікс, який визначає тип даних і використовувану арифметику:

• us (unsigned saturation) – арифметика з насиченням, дані без знаку. При беззнаковому насиченні, якщо результат операції перевищує максимальне значення, у початковий операнд записується максимальне значення (відбувається “насичення”). Якщо результат операції виявився меншим за нижню межу допустимого діапазону, то у початковий операнд записується мінімально можливе значення;

• s або ss (signed saturation) – арифметика з насиченням, дані із знаком. При знаковому насиченні, використовується арифметика з насиченням, але для даних із знаком;

• якщо в суфіксі немає ні символу s, ні символів ss, us то застосовується циклічна арифметика (wraparound). У цьому випадку, якщо результат операції виходить за двійкову розрядну сітку використовуваного типу даних, то “зайві” старші біти результату відкидаються;

• b, w, d, q – ці букви вказують тип даних. Якщо в суфіксі є дві з цих букв, то перша з них відповідає вхідному операнду, а друга – вихідному.

Дані із знаком і без знаку мають різний діапазон допустимих значень. Отже, якщо використовується арифметика з насиченням, то при виході результату операції за межі допустимого діапазону у вихідний операнд будуть записані різні значення залежно від типу даних. Допустимі діапазони значень для даних різних типів показано у табл. 1.

Таблиця 1 – Діапазони значень типів даних

Тип даних	Мінімальне значення	Максимальне значення
Байт із знаком	-128 (80h, 1000 0000)	127 (7Fh, 0111 1111)
Байт без знаку	0 (00h, 0000 0000)	255 (FFh, 1111 1111)
Слово із знаком	-32768 (8000h)	32767 (7FFFh)
Слово без знаку	0000h	65535 (FFFFh))
Подвійне слово із знаком	-2147483648 (80000000h)	2147483647(7FFFFFFF)
Подвійне слово без знаку	00000000h	FFFFFFFFh (4294967295)
Чотирне слово із знаком	1000 0000 0000 0000h	7FFF FFFF FFFF FFFFh
Чотирне слово без знаку	0000 0000 0000 0000	FFFF FFFF FFFF FFFF

При додаванні слова зі знаком 7F38h (32568) та слова 1707h (5895) результат дорівнює 963F (38463). Математична сума виявилася більшою за межу 7FFFh (32767) і тому відбулося "насичення". Якщо ті ж самі два значення додати як слова без знаку, то вийде 963Fh (38463). В даному випадку "насичення" не відбувається, оскільки результат менший за максимально можливий FFFFh.

MMX-команди умовно можна розділити на такі групи (табл. 2):

• *команди пересилання даних*;
• *арифметичні команди*;
• *команди зсуву*;
• *логічні команди*;
• *команди порівняння*;
• *команди пакування і розпакування*.

Команди пересилання даних

В MMX добавлено *команди пересилання* 64-розрядних запакованих даних в і з пам’яті.

MOV**D** пересилає 32-розрядні дані між пам’яттю і MMX регістрами (молодші 32-розряди).

MOV**Q** пересилає 64-розрядні дані між MMX регістрами і пам’яттю, або між регістрами.

Запаковані арифметичні команди

Запаковані арифметичні команди – додавання, віднімання, множення запакованих цілих чисел.

PADD[**B**,**W**,**D**] – додавання запакованих [байт, слово, подвійне слово] з циклічною арифметикою.

PADD**S**[**B,W**] – додавання запакованих знакових цілих [байт, слово] із знаковим насиченням.

PADD**US**[**B,W**] – додавання запакованих беззнакових цілих [байт, слово] із беззнаковим насиченням.

PSUB[**B**,**W**,**D**] – віднімання запакованих цілих [байт, слово, подвійне слово] із циклічною арифметикою.

PSUB**S**[**B**,**W**] – віднімання запакованих знакових цілих [байт, слово] із знаковим насиченням.

PSUB**US**[**B**,**W**] – віднімання запакованих беззнакових цілих [байт, слово] із беззнаковим насиченням.

PMUL**HW**, **P**MUL**LW** – множення запакованих цілих слів і збереження старшої, молодшої половини результату.

PMADD**WD** – множення запакованих слів і додавання результатів множення.

Кожна одинична команда незалежна від інших, а разом вони виконуються паралельно. Є версії команд для циклічної арифметики, беззнакового насичення і знакового насичення. У версії команди регістр-регістр реалізовано пересилання даних між MMX регістрами і цілочисловими регістрами.

Команди MMX не впливають на прапори умов. Регістри MMX, на відміну від регістрів FPU, адресуються фізично, а не відносно поля TOS регістра стану FPU. Будь-яка інструкція MMX обнуляє поле TOS. Інструкції MMX доступні з будь-якого режиму процесора.

Є команда множення з додаванням PMADDWD, рис. 2. Команда перемножує чотири пари 16-розрядних операндів і дає в результаті запаковані чотири 32-розрядні добутки. Потім вона додає два суміжних добутки в один 32-розрядний результат, а інші два суміжні добутки в інший 32-розрядний результат і остаточно генерує запаковане подвійне слово. Таким чином, виконується чотири 16-розрядні множення і два 32-розрядні додавання в одній команді.

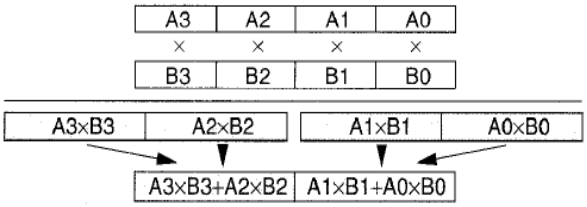


Рисунок 2 – Команда множення з додаванням

Є дві команди множення 16-бітової точності `PMULHW, PMULLW`. Перша виконує чотири 16-бітові множення і дозволяє користувачу вибрати старшу або молодшу частину 32-бітового результату. Таким, чином операнди і результат є 16-бітовими.

Команди логічного зсуву

В MMX реалізовано дві групи команд:

- *логічного зсуву* (вліво, вправо);

PSLL[**W**,**D**,**Q**] – логічний зсув вліво запакованого [слово, подвійне слово, чотирне слово] на величину вказану у MMX регістрі або безпосереднє значення.

PSRL[**W**,**D**,**Q**] – логічний зсув вправо запакованого [слово, подвійне слово, чотирне слово] на величину вказану у MMX регістрі або безпосереднє значення.

- *арифметичного зсуву* (вправо).

PSRA[**W**,**D**] – арифметичний зсув вправо [запаковане слово, подвійне слово] на величину вказану у MMX регістрі або безпосереднє значення.

Запаковані елементи зсовуються паралельно і незалежно. Операція зсуву підтримується для слів і подвійних слів. Є команди логічного зсуву (вліво, вправо) усього 64-розрядного ММХ регістру. Ці команди працюють з операндами, які завантажуються з 64-розрядної пам'яті.

Логічні команди

В ММХ добавлено набір 64-розрядних *логічних команд* AND, ANDNOT, OR, XOR.

PAND – побітове логічне І.

PANDN – побітове логічне І-НЕ.

POR – побітове логічне АБО.

PXOR – побітове логічне виключальне АБО.

Приклад використання масок з логічними командами для реалізації вибору за умовами показано на рис. 3.

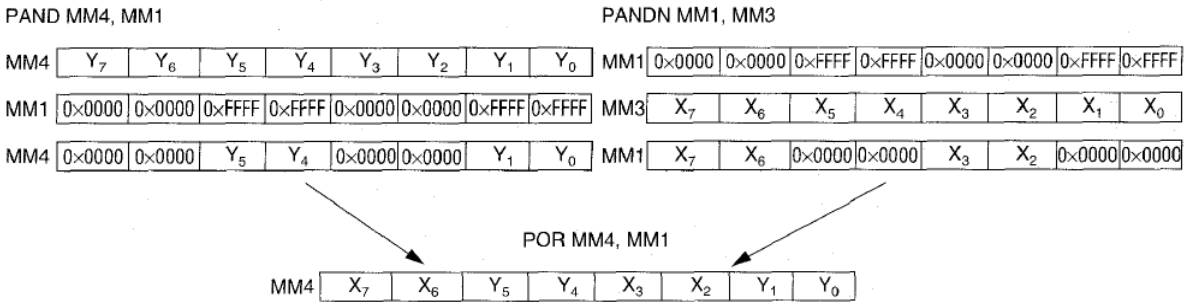


Рисунок 3 – Використання логічних команд PAND, PANDN, POR

Команди порівняння

Команди *порівняння* порівнюють запаковані байти, слова, подвійні слова.

PCMPEQ[**B**,**W**,**D**] порівняння на рівність запакованих [байт, слово, подвійне слово].

PCMPGT[**B**,**W**,**D**] – порівняння на більше запакованих [байт, слово, подвійне слово].

Команди порівняння виконуються паралельно і незалежно порівнюють всі відповідні елементи двох запакованих типів даних. Вони генерують маску з 1 і 0, залежно від умов порівняння true або false, рис. 4.

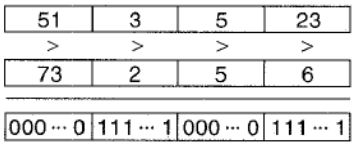


Рисунок 4 – Команда порівняння ">" запакованих слів

Приклад результату команди порівняння PCMPEQB, рис. 5.

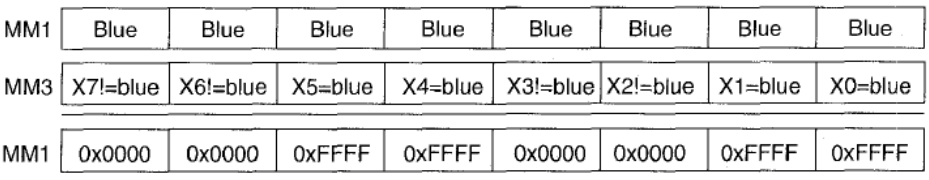


Рисунок 5 – Команда порівняння PCMPEQB

Команди пакування і розпакування

MMX команди *перетворення* пакують і розпаковують байти, слова і подвійні слова.

PACK**SS**[**WB**,**DW**] – пакування слова у байти, подвійного слова у слова із знаковим насиченням.

PACK**USWB** – пакування слова у байти із беззнаковим насиченням.

PUNPCK**H**[**BW**,**WD**,**DQ**] – розпакування з перемішуванням (старшого порядку) байтів у слово, слів у подвійне слово, подвійних слів у чотирне слово з MMX регістру.

PUNPCK**L**[**BW**,**WD**,**DQ**] – розпакування з перемішуванням (молодшого порядку) байтів у слово, слів у подвійне слово, подвійних слів у чотирне слово з MMX регістру..

Команди unpack розпаковують дані меншої точності у дані більшої точності. Крім цього ці команди дозволяють перемішувати дані, рис. 6. Так команди `PUNPCKLBW` бере молодші чотири байти з кожного операнду і перемішує їх у результуючий регістр. Ця команда має наступне використання:

• операції інтерполяції, коли новий піксел потрібно вставити між існуючими;

•транспонування матриць (переставлення стовпців у рядки);

• перетворення між піксельними форматами RGB або RGBA і кольоровими площинами.

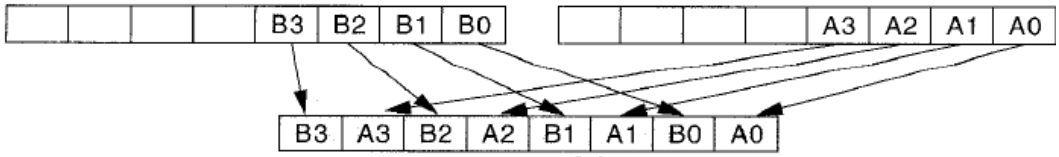


Рисунок 6 – Розпакування байтових даних

Команди пересилання даних

MOV[**D,Q**] – пересилання подійного, чотирного слова в MMX регістр з іншого MMX регістра.

Команда керування станом

EMMS – завершення режиму роботу в MMX.

У табл. 2 показано об’єднані за групами MMX команди.

Таблиця 2 – MMX команди

Циклічна арифметика	Знакове насичення	Беззнакове насичення	Операція
Арифметичні команди			
PADDB, PADDW, PADDD PSUBB, PSUBW, PSUBD PMULL, PMULH PMADD	PADDSB, PADDSW PSUBSB, PSUBSW	PADDUSB, PADDUSW PSUBUSB, PSUBUSW	Додавання Віднімання Множення Множення і додавання
Команди порівняння			
PCMPEQB, PCMPEQW, PCMPEQD PCMPGTPB, PCMPGTPW, PCMPGTPD			Порівняння на рівність Порівняння на більше
Команди пакування			
	PACKSSWB,	PACKUSWB	Пакування

	PACKSSDW		
Команди розпакування			
PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ			Розпакувати старшу половину Розпакувати молодшу половину
Логічні команди			
Запаковані		Повне чотирне слово	
		PAND PANDN POR PXOR	І І-НЕ АБО Виключальне АБО
Команди зсуву			
PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD		PSLLQ PSRLQ	Логічний зсув вліво Логічний зсув вліво Арифметичний зсув вправо
Команди пересилання даних			
Подвійне слово		Чотирне слово	
MOVD MOVD MOVD		MOVQ MOVQ MOVQ	Register to Register Load from Memory Store to Memory
EMMS			Завершення режиму роботи в ММХ

2.1. Призначення команд ММХ

Таблиця 3 – Призначення команд ММХ

Команда	Описання
EMMS	Підготовка співпроцесора до виконання команд
MASKMOVQ	Запис байту в пам'ять з регістра ММХ по масці (вибіркова)
MOVD mmxi, mmxj/mem mem, mmxj	Копіювання 32-розрядного числа (молодші розряди mmx)
MOVNTQ	Запис 64 біт в пам'ять з регістра ММХ (без використання кеш-пам'яті)
MOVQ mmxi, mmxj/mem mem, mmxj	Копіювання 64-розрядного числа
PACKSSWB mmxi, mmxj/mem	Пакування із знаковим насиченням слів у байти
PACKSSDW mmxi, mmxj/mem	Пакування із знаковим насиченням подвійних слів в слова
PACKUSWB mmxi, mmxj/mem	Пакування із беззнаковим насиченням слів у байти
PADDB/W/D/Q mmxi, mmxj,memi	Додавання (паралельне) запакованих байтів/слів/подвійних слів/чотирних слів
PADDSB/W mmxi, mmxj/memi	Додавання запакованих байтів/слів із знаковим насиченням
PADDUSB/W mmxi, mmxj/memi	Додавання запакованих байтів/слів із беззнаковим насиченням
PAND mmx_i, mmx_j/mem64	Запаковане логічне І операндів
PANDN mmx_i, mmx_j/mem64	Запаковане логічне І-НЕ операндів

PAVGB/W mmxi, mmxj/mem64	Середнє значення байтів/слів
PCMPEQB/W/D	Порівняння на рівність запакованих байтів/слів/подвійних слів
PCMPGTB/W/D	Порівняння за умовою "більше ніж" запакованих байтів/слів/подвійних слів
PEXTRW r32/r64, mmx, imm8	Видобування 16-бітового слова з регістра MMX по масці imm
PINSRW mmx, r32/mem16, imm8	Вставка 16-бітового слова в регістр MMX
PMADDWD mmx_i, mmx_j/mem	Запаковане знакове множення слів операндів з подальшим додаванням проміжних результатів в подвійному слові (матрично-векторне множення)
PMAXSW mmx_i, mmx_j/mem64	Повернення максимальних запакованих знакових слів
PMAXUB mmx_i, mmx_j/mem64	Повернення максимальних запакованих беззнакових байтів
PMINSW mmx_i, mmx_j/mem64	Повернення мінімальних запакованих знакових слів
PMINUB mmx_i, mmx_j/mem64	Повернення мінімальних запакованих беззнакових байтів
PMOVMSKB pmovmskb r32/r64, mmx/mem64	Переміщення байтової знакової маски в цілочисловий регістр
PMULHUW mmx_i, mmx_j/mem	Запаковане беззнакове множення слів з поверненням старшого слова результату
PMULHW mmx_i, mmx_j/mem64	Запаковане знакове множення слів з поверненням старшого слова результату
PMULLW mmx_i, mmx_j/mem64	Запаковане знакове множення слів з поверненням молодшого слова результату
PMULUDQ xmm,xmm/mem128	Множення 64-бітових операндів. Результат не повинен перевищувати 128-бітовий розмір
PMUUDQ xmm, xmm/mem128	Множення 64-бітових операндів. Результат не повинен перевищувати 128-бітовий розмір
POR mmx, mmx/mem64	Запаковане логічне АБО операндів
PSADBW mmx_i, mmx_j/mem64	Сумарна різниця значень пар беззнакових запакованих байт
PSIGNB/W/D psignb/w/d mmx_i, mmx_j /mem64	Заперечення запакованих цілих чисел в mmxi, якщо знак в mmxj менше нуля
PSHUFW/D/Q xmm xmm/mem128, imm	Пересилання з перегрупуванням чотирьох 16//32/64-бітових слів з молодшої половині dest в молодшу половину src. Перегрупування задається вмістом imm
PSLLW/D/Q mmx_i, imm8/mmx_j/mem64	Логічний зсув вліво запакованих слів/подвійних слів/чотирних слів
PSLLDQ xmm, imm8	Логічний зсув вмісту вліво на imm8 бітів
PSRAW/D mmx_i, imm8/mmx_j/mem64	Арифметичний зсув вправо запакованих подвійних слів/чотирних слів
PSRLW/D/Q mmx_i, imm8/mmx_j/mem64	Логічний зсув вправо запакованих слів/подвійних слів/чотирних слів
PSRLDQ xmm, imm8	Логічний зсув вмісту вправо на imm8 бітів
PSUBB/W/D/Q mmxi,mmxj mmxi,memi	Віднімання (паралельне) запакованих байтів/слів/ подвійних слів/чотирних слів за правилами циклічної арифметики

PSUBSB/W mmxi,mmxj mmxi,memi	Віднімання запакованих байтів/слів із знаковим насиченням
PSUBUSB/W mmxi,mmxj mmxi,memi	Віднімання запакованих беззнакових байт/слів із насиченням
PUNPCKHBW/D/Q mmx_i, mmx_j/mem64	Розпаковування старших запакованих байт/слів/подвійних слів в слово/подвійне слово/чотирне слово
PUNPCKLBW/D/Q mmx_i, mmx_j/mem64	Розпаковування молодших запакованих байт/слів/подвійних слів в слово/подвійне слово/чотирне слово
PXOR mmx_i, mmx_j/mem64	Запаковане логічне виключальне АБО операндів

3. Особливості реалізації MMX

Для обробки даних і зберігання проміжних результатів в Pentium MMX використовуються вісім 64-розрядних регістрів MM0,...,MM7, які фізично поєднані із стеком регістрів математичного співпроцесора. При виконанні будь-якої з MMX-команд відбувається встановлення "режиму MMX" з позначкою цього в слові стану співпроцесора (FPU Tag Word). Після цього стек регістрів співпроцесора розглядається як набір MMX-регістрів. Завершує роботу в режимі MMX команда EMMS (End MultiMedia State). З одного боку, така реалізація дозволила забезпечити нормальну роботу застосунків з використанням MMX команд у багатозадачних системах, що не підтримують цю технологію, оскільки усі подібні системи створюють власну копію утримуваного стека співпроцесора і слова його стану для кожного процесу. З іншого боку, перехід між режимами займає значний час, і поєднання, наприклад, в одному циклі команд співпроцесора з командами MMX може не лише не прискорити, а навіть істотно уповільнити виконання програми. Тому для досягнення найкращих результатів рекомендується групувати ці команди окремо.

Оскільки MMX – досить вузькоспеціалізоване розширення системи команд процесора, не можна чекати кардинального прискорення роботи тільки від самого факту переходу на процесор MMX. На застосунках загального призначення без команд MMX, реальна продуктивність зростає лише на одиниці відсотків, хоча тести можуть показувати її зростання на 20-30% – це відбувається через циклічність більшості тестів, коли велика частина циклу потрапляє у збільшений внутрішній кеш.

4. Приклади використання MMX команд

Приклади.

1. Виведення графічної картинки на екран з накладенням на існуючі байти на екрані.

Звичайна програма:

```
rdi - адреса екрану (буфера екрану)
rbx - адреса картинки

mov al, [rdi] ; завантаження байту з екрану
mov ah, [rbx] ; завантаження байту картинки
add al,ah ; їх додавання
jnc loop ; якщо отримане число менше 255, то перехід на loop
```

```
mov al,255 ; якщо числ0 >255 то встановлюємо його рівним 255
loop: mov [rdi],al ; помістити вміст регістру al на екран
inc rdi ; наступний байт екрану
int rbx ; наступний байт картинки
....
.... ; (так 3 рази, тобто окремо для кожної R,G,B-компоненти)
```

MMX варіант:

```
movd mm0,[rdi] ; завантаження чотирьох байт з екрану
movd mm1,[rbx] ; завантаження чотирьох байт картинки
paddusb mm0,mm1 ; додавання зразу 4-х байтів з їх перевіркою на переповнення
на 255
movd [rdi],mm0 ; переслати ці чотири байти назад на екран
add rdi,4 ; наступні байти екрану
add rbx,4 ; наступні байти картинки
```

Зразу видно приріст в швидкості завдяки паралельному обробленню зразу чотирьох байтів, а також через відсутність команд умовних переходів.

2. Виведення графічної картинки на екран з врахуванням прозорості.

```
rdi - адреса екрану (буфера екрану)
rbx — адреса картинки

mov al,[rbx] ; завантаження байту картинки
cmp al,0 ; перевірка на прозорість
; (тобто при al=0 на екран байт не записується)
jz loop
mov [rdi],al ; якщо al != 0 — записуємо на екран
loop:
inc rdi ; наступний байт екрану
inc rbx ; наступний байт картинки
....
.... ; (так 3 рази, тобто окремо для кожної R,G,B-компоненти)
```

MMX варіант:

```
pxor mm2,mm2 ; обнулення регістру mm2
movd mm0,[rbx] ; завантаження 4-х байтів картинки
movd mm1,[rsi] ; завантаження 4-х байтів екрану (фону)
pcmpeqb mm2,mm0 ; створення маски тих байт які потрібно вивести
pand mm2,mm1 ; обнулення непотрібних байтів
por mm1,mm0 ; додавання їх з байтами картинки
movd [rsi],mm1 ; переслати байти назад на екран
add rbx,4 ; наступні байти картинки
add rsi,4 ; наступні байти екрану
```

У цьому прикладі, як і в попередньому, видно приріст в швидкості завдяки паралельному обробленню зразу чотирьох байтів, а також через відсутність команд умовних переходів.

3. Розмиття зображення при рухові (motion blur). Об’єкти залишають за собою повільно зникаючий слід.

```
mask7f1 = 0x7f7f7f7f7f7f7f7f
movq mm4,mask7f1 ; завантаження маски в регістр mm4

movq mm0,[rsi] ; завантаження 8-байтів поточного кадру
; (зображення на екрані в даний момент)
movq mm1,[rdi] ; завантаження 8-байтів попереднього кадру
psrlq mm0,1 ; швидка процедура ділення на два кожного з 8 байтів
```

```
psrlq mm1,1 ; (як в mm0, так і в mm1)
pand mm0,mm4 ; логічне I
pand mm1,mm4 ; логічне I
paddb mm0,mm1 ; додавання mm0 з mm1(середнє арифметичне між mm0 і mm1)
movq [rsi],mm0 ; переслати на екран (відеобуфер)
movq [rdi],mm0 ; переслати в буфер попереднього кадру
add rsi,8
add rdi,8
```

Чотирикратне прискорення за рахунок одночасного оброблення восьми байтів.

4. Канал прозорості (alpha blending) – одне зображення плавно появляється або розчиняється над іншим.

Формула ab: a = b + (a-b)*alpha

де: a – основне зображення, b – зображення яке накладається, alpha – кількість градацій 0...255.

Ініціалізація регістрів:

біти 31...24 23...16 15...8 7...0

```
eax =  alpha   alpha   alpha  alpha

movd mm4,eax ; переміщення даних з eax в mm4
punpcklwd mm4,mm4 ; створення чотирьох слів alpha-каналу
psrlw mm4,8 ; переміщення старшої частини слова в молодшу
pxor mm7,mm7 ; обнулення mm7
```

Основна процедура:

```
movd mm0,[rsi] ; завантажити в mm1 4-байти накладуваного зображення
movd mm1,[rdx] ; завантажити в mm0 4-байти основного зображення
punpcklbw mm0,mm7
punpcklbw mm1,mm7
psubw mm0,mm1 ; відняти із накладуваного, основне зображення
psllw mm1,8 ; перенести молодшу частину слова регістра mm1 в старшу
pmullw mm0,mm4 ; множення накладуваного зображення на alpha-канал
paddw mm1,mm0 ; додавання його з основним
psrlw mm1,8 ; перенесення результату із старшої частини слова в молодшу
packuswb mm1,mm1 ; і переведення його в 32 біти
movd [rdi],mm1 ; пересилання отриманого зображення на екран (у відеобуфер)
add rsi,4
add rdx,4
add rdi,4
```

У цьому прикладі використовується швидке 16-розрядне множення, яке і дає прискорення виконання.

5. Асемблерна функція для змішування двох ARGB пікселів.

```
; DWORD LerpARGB(DWORD a, DWORD b, DWORD f)
global LerpARGB
LerpARGB:
    ; завантаження пікселів і розширення в 4 words
    movd        mm1, [rsp+4]    ; mm1 = 0 0 0 0 aA aR aG aB
    movd        mm2, [rsp+8]    ; mm2 = 0 0 0 0 bA bR bG bB
    pxor        mm5, mm5        ; mm5 = 0 0 0 0 0 0 0 0
    punpcklbw   mm1, mm5        ; mm1 = 0 aA 0 aR 0 aG 0 aB
    punpcklbw   mm2, mm5        ; mm2 = 0 bA 0 bR 0 bG 0 bB

    ; завантаження коефіцієнтів і збільшення діапазону до [0-256]
```

```
movd        mm3, [rsp+12]   ; mm3 = 0 0 0 0 faA faR faG faB
punpcklbw   mm3, mm5        ; mm3 = 0 faA 0 faR 0 faG 0 faB
movq        mm6, mm3        ; mm6 = faA faR faG faB [0 - 255]
psrlw       mm6, 7          ; mm6 = faA faR faG faB [0 - 1]
paddw       mm3, mm6        ; mm3 = faA faR faG faB [0 - 256]

; fb = 256 - fa
pcmpeqw     mm4, mm4        ; mm4 = 0xFFFF 0xFFFF 0xFFFF 0xFFFF
psrlw       mm4, 15         ; mm4 =   1   1   1   1
psllw       mm4, 8          ; mm4 = 256 256 256 256
psubw       mm4, mm3        ; mm4 = fbA fbR fbG fbB

; res = (a*fa + b*fb)/256
pmullw      mm1, mm3        ; mm1 = aA aR aG aB
pmullw      mm2, mm4        ; mm2 = bA bR bG bB
paddw       mm1, mm2        ; mm1 = rA rR rG rB
psrlw       mm1, 8          ; mm1 = 0 rA 0 rR 0 rG 0 rB

; пакування в eax
packuswb    mm1, mm1        ; mm1 = 0 0 0 0 rA rR rG rB
movd        eax, mm1        ; eax = rA rR rG rB
ret
```

5. Прикладне застосування

5.1. Накладення фону

Для полегшення перетворення між новими типами запакованих даних існують інструкції пакування та розпакування. Це особливо важливо, коли алгоритм потребує вищої точності в своїх проміжних обчисленнях, як-от фільтрація зображень. Фільтр на зображенні звичайно передбачає набір операцій множення між коефіцієнтами фільтра та набором суміжних пікселів зображення, накопичуючи всі значення разом. Ці множення та накопичення потребують більшої точності, ніж 8-бітовий тип даних пікселів. Рішення полягає в тому, щоб розпакувати 8-бітові пікселі зображення в 16-бітові слова, виконати обчислення в 16-бітових словах, не турбуючись про переповнення, а потім запакувати назад у 8-бітові слова пікселів перед збереженням відфільтрованих пікселів у пам’ять.

У телебаченні використовують накладення зображення ведучого на якийсь фон, наприклад на зображення карти погоди. Розглянемо приклад, накладення зображення жінки на зображення весняного цвітіння з використанням паралельної обробки чотирьох 16-бітових пікселів. Команди також дозволяють обробляти вісім 8-бітових пікселів паралельно для значного прискорення продуктивності.

Рисунок 7 - Накладення зображення на фон

Спочатку візьмемо чотири пікселі із зображення жінки на зеленому тлі.

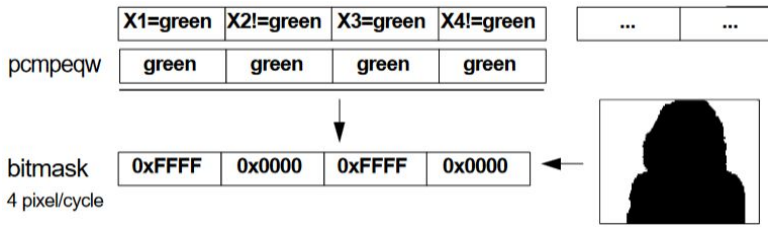


Рисунок 8 - Вилучення фону за допомогою команд порівняння

Верхній ряд наведених нижче даних задають пікселі, які чергуються між зеленим і не зеленим. Інструкція порівняння створює маску для цих даних. Ця маска – це послідовність слів, які є всі одиниці або всі нулі, що подають логічні значення істини та хиби. Тепер відомо, що що таке небажаний фон і те, що потрібно зберегти, показано нижче за допомогою тіні.

Ця маска тепер використовується на тих самих чотирьох пікселях із зображення з жінкою та еквівалентом чотири пікселі від весняного цвітіння. Інструкції «AND NOT» та «AND» використовують маску, щоб визначити, які пікселі слід зберегти від весняного цвіту та жінки. Вони також встановлюють небажані пікселі в нулі. Інструкція «АБО» створює остаточну картинку. Чотири пікселі були зіставлені за допомогою лише чотирьох інструкцій MMX без будь-яких розгалужень. Під час роботи над цим прикладом інструкція PANDN інвертує всі біти в масці раніше застосування операції AND.

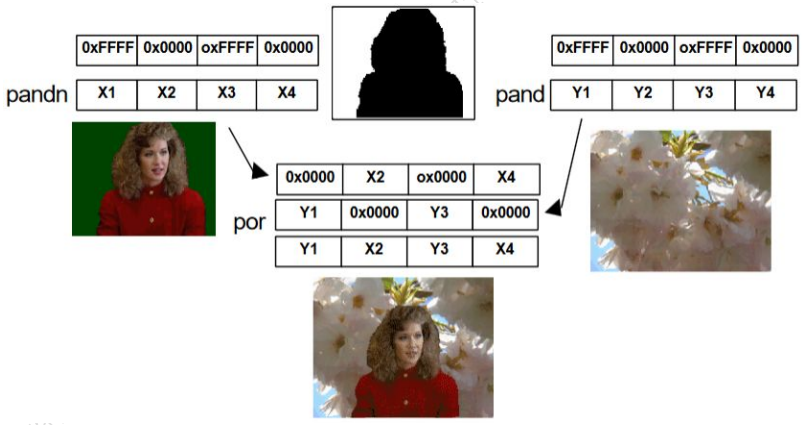


Рисунок 9 - Накладення бажаного зображення на фон цвітіння

Без технології MMX кожен піксель обробляється окремо і потребує умовного розгалуження. Використовуючи інструкції MMX, можна обробляти паралельно без умовних розгалужень вісім 8-бітових пікселів.

5.2. Векторний скалярний добуток

Векторний скалярний добуток є одним із найпростіших алгоритмів, які використовуються в обробці сигналів природних даних наприклад зображення, аудіо, відео та звук. У наступному прикладі показано, як команда PMADD допомагає пришвидшити алгоритми, використовуючи векторні скалярні добутки.

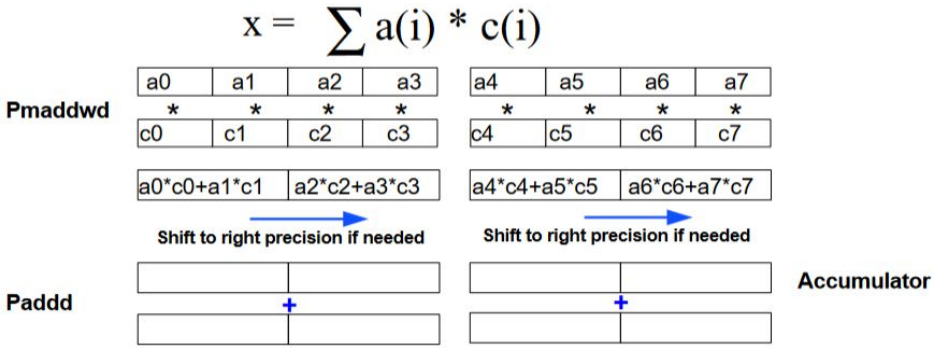


Рисунок 10 – Реалізація векторного скалярного добутку

Примітка. Вхідні дані і коефіцієнти 16-бітові. Якщо не так то розпакувати їх у 16-біт.

Команда PMADD буде обробляти чотири множення і два додавання одночасно. У поєднанні з командою PADD, виконується вісім операцій множення-накопичення. Цей 8-елементний вектор добре підходять для двох команд PMADD і двох команд PADD.

Якщо припустити, що точність, яку підтримує команда PMADD, достатня, цей скалярний добуток 8-елементних векторів можна завершити за допомогою восьми команд MMX: дві PMADD, ще дві PADD, два зсуви (якщо потрібно виправити точність після операції множення), і два переміщення в пам'яті для завантаження одного з векторів (інший вектор завантажує команда PMADD, яка може мати один із своїх операндів в пам'яті)

Більшість команд MMX можна виконати за один такт, тому підвищення продуктивності буде більш суттєвим, ніж просте співвідношення кількості інструкцій.

5.3. Матричне множення

Обчислення, які маніпулюють 3D-об'єктами, базуються на матрицях 4 на 4, які багато разів помножуються на 4-елементні вектори. Вектор містить X, Y, Z і перспективну коригувальну інформацію для кожного пікселя. Матриця 4 на 4 використовується для обертання, масштабування, переміщення та оновлення інформації про перспективу для кожного пікселя. Ця матриця 4 на 4 застосовується до багатьох векторів.

Програми, які вже використовують 16-розрядні цілі чи дані з фіксованою крапкою, можуть використовувати інструкцію PMADD. Загалом буде одна інструкція PMADD на рядок матриці з чотирьох елементів.

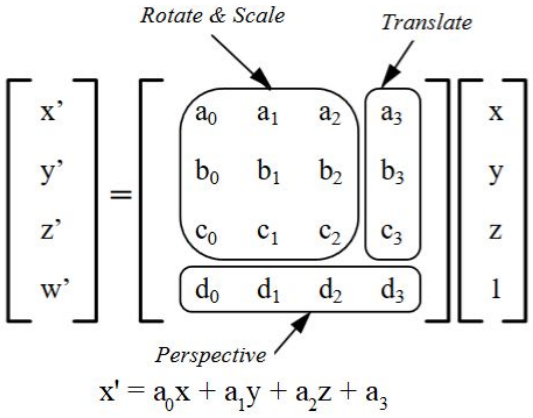


Рисунок 11 – Структура даних матричного множення

5.3. 24-бітові кольори

Набір інструкцій MMX надає графічним програмам можливість переходу від 8-бітової або 16-бітової таблиці пошуку кольорів до 24-бітової, або «справжнього» кольору, функції, яка значно підвищить реалістичність ігрової графіки. У багатьох випадках це можна зробити за той самий час, який зараз потрібен для 8-бітової графіки. Для 24-бітових і 32-бітових кольорів кожен червоний, зелений і синій колір задається 8-бітовими значеннями. Є вісім додаткових бітів у 32-бітовому кольорі для значення альфа, рис. 12.

Складання зображень і альфа-змішування – це операції, які можна виконувати з 24-бітовим кольором зображення.

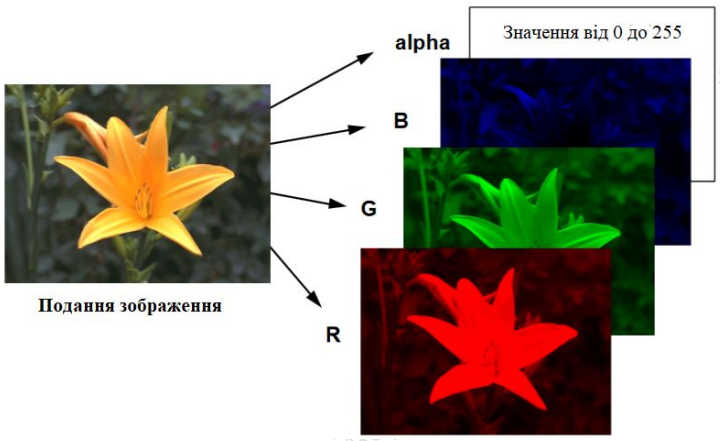


Рисунок 12 – Подання 32-бітового зображення

5.4. Розчинення зображення за допомогою альфа-змішування

Набір команд MMX використовується для швидкого компонування зображень. Наприклад, зображення квітки розчиняється в зображенні лебедя, рис. 12. Математика для розчинення є простою функцією. Альфа визначає інтенсивність квітки. При повній інтенсивності 8-бітове значення альфа-каналу квітки дорівнює FFH, або 255. Якщо вставити 255 у рівняння розчинення, кожен піксель квітки дорівнюватиме 100 відсоткам, а кожен піксель лебедя – 0 відсоткам. Показане нижче рівняння обчислює кожен піксель:

```
Result_pixel = Flower_pixel * (alpha/255) + Swan_pixel * (1 - (alpha/255))
```

Рисунок 12 – Розчинення зображення

Коли значення альфа дорівнює 230, отримане зображення складається з 90 відсотків квітки та 10 відсотків лебедя. При уважному розгляді частина зображення лебедя з’являється на малюнку праворуч від знака рівності.

У цьому прикладі припускається, що 24-розрядні кольорові дані організовані таким чином, що одночасно обробляються чотири пікселі з однієї колірної площини, тобто зображення розділене на окремі кольорові площини: червону, зелену і синю. Перші чотири значення червоного кольору з квітки та лебедя будуть оброблені першими. Після закінчення червоної площини обробка переходить до зеленої та синьої площин.

Команда розпакування бере перші чотири байти "червоних" даних, які подані у 8-бітовими значеннями, і розпаковує кожен піксель у 16-бітові елементи, поміщаючи їх у 64-бітовий регістр MMX. Значення альфа, яке обчислюється один раз на кадр, є іншим операндом. PMUL множить два вектори паралельно. Подібним чином Unpack і PMUL створюють проміжний результат для лебедя. Тепер два проміжні результати додаються разом за допомогою PADD, а кінцевий результат надсилається в пам’ять за допомогою PACK, який перетворює проміжні 16-бітові значення назад у 8-бітові значення пікселів, які можна зберегти.

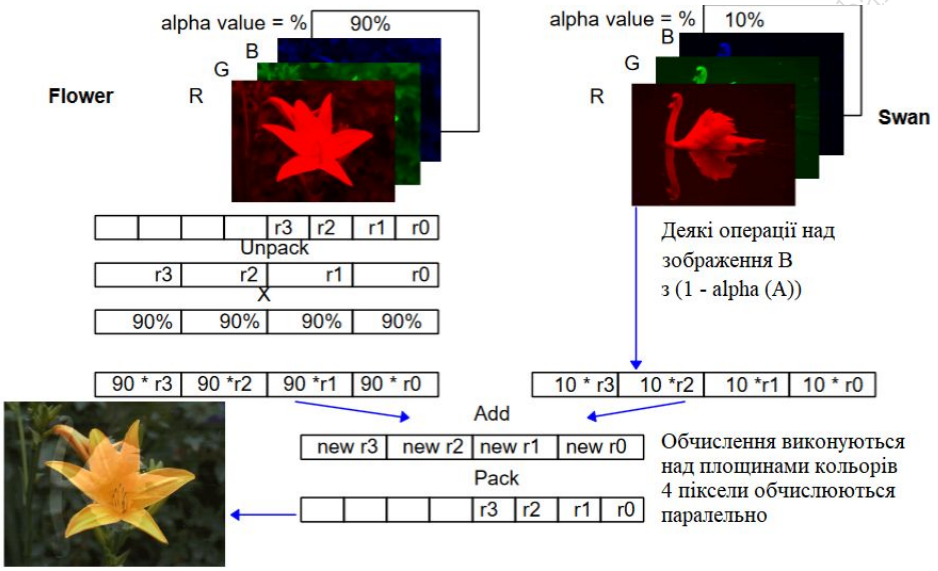


Рисунок 13 – Обчислення з використанням команд Unpack, Add, Pack

Якщо зображення використовують роздільну здатність 640x480, а техніка розчинення використовує всі 255 кроків альфа-значення, тоді використовується 117 мільйонів PUNPCK і PMUL та 58 мільйонів PADD і PACK.

Контрольні запитання.

1. Передумови виникнення технології MMX.
2. Типи даних використовуваних у технології MMX.
3. Циклічна арифметика і арифметика з насиченням у MMX
4. Групи команд MMX.
5. Команди пересилання даних MMX.
6. Арифметичні команди MMX.
7. Команди зсуву MMX.
8. Логічні команди MMX.
9. Команди порівняння MMX.
10. Команди пакування і розпакування MMX
11. Прикладні задачі, в яких використовуються команди MMX.

10. Розширення команд SSE

Мета. Вивчення команд SSE розширення.

Вступ. Розширення команд SSE – це команди потокової обробки та виконання арифметичних операцій над цілими і дійсними числами. SSE (Streaming SIMD Extensions, потокове SIMD-розширення процесора) набір інструкцій, розроблених Intel і вперше представлених у процесорах серії Pentium III.

План.

1. Розширення команд SSE.
2. Розширення команд SSE2.
3. Розширення команд SSE3.
4. Розширення команд SSSE3.
5. Розширення команд SSE4.

1. Розширення команд SSE

Розширення команд SSE (англ. Streaming SIMD Extensions, потокове SIMD-розширення) дозволило вирішити основні проблеми MMX:

- неможливість одночасного використання команд з плаваючою крапкою співпроцесора FPU x87 і цілочислових команд MMX на спільних регістрах FPU;
- MMX підтримує тільки цілочислові команди.

Для підтримки розширення команд SSE в архітектурі процесора додано вісім **128-бітових** регістрів (xmm0, ... ,xmm7), окремих від регістрів FPU і набір окремих команд доя роботи з числами з плаваючою крапкою.

Розширення команд SSE використовує наступні дані:

• чотири 32-бітові числа з плаваючою крапкою одинарної точності;

Перевага у швидкості обчислень досягається в тому випадку, коли необхідно виконати одну і ту саму послідовність дій над різними даними.

Реалізація блоків SIMD виконується розпаралелюванням обчислювального процесу між даними. Тобто коли через один блок даних проходить по черзі багато потоків даних.

Команди SSE діляться на чотири групи:

- SIMD команди з плаваючою крапкою одинарної точності працюють з запакованими і скалярними числами які розміщуються в XMM регістрах або пам'яті.
- MXSCR команди керування станом;
- 64-бітові SIMD команди працюють з цілочисловими даними, які розміщуються на MMX регістрах;
- команду керування кешом, попереднього завантаження, впорядкування функціональності.
- команди перетворення.

SIMD команди з плаваючою крапкою одинарної точності поділяються на наступні групи команд:

- команди пересилання даних (SSE);

- команди запакованої арифметики (SSE);
- команди порівняння (SSE);
- логічні команди (SSE);
- команди перетасування і розпакування (SSE);
- команди перетворення (SSE).

Команди пересилання даних (SSE) (префікси: R - зворотні значення, U - невпорядковні (unordered); суфікси: A - вирівняні (alugned), U - невирівняні (unaligned), P - запаковані (packed), S - одинарної точності (single-precision), S - скаляр, l - молодше чотирне слово, h - старше чотирне слово, I - встановлює прапори в регістрі EFLAGS):

MOVAPS – перемістити чотири вирівняні упаковані значення з плаваючою крапкою одинарної точності між регістрами XMM або пам'яттю.

MOVHLPS – перемістити два запакованих значення з плаваючою крапкою одинарної точності зі старшого чотирного слова регістра XMM до молодшого чотирного слова іншого регістра XMM.

MOVHPS – переміщення двох запакованих значень з плаваючою крапкою одинарної точності до або з старшого чотирного слова регістра або пам'яті XMM.

MOVLHPS – перемістити два запакованих значення з плаваючою крапкою одинарної точності з молодшого чотирного слова регістра XMM до старшого чотирного слова іншого регістра XMM.

MOVLPS – перемістити два запакованих значення з плаваючою крапкою одинарної точності до або з молодшого чотирного слова регістра або пам'яті XMM.

MOVMSKPS – витягти маску знака з чотирьох упакованих значень з плаваючою крапкою одинарної точності.

MOVSS – перемістити скалярне значення з плаваючою крапкою одинарної точності між регістрами XMM або пам'яттю.

MOVUPS – перемістити чотири невирівняні запаковані значення з плаваючою крапкою одинарної точності між регістрами XMM або пам'яттю.

Команди запакованої арифметики (SSE):

ADDPS – додає запаковані значення з плаваючою крапкою одинарної точності.

ADDSS – додає скалярні значення з плаваючою крапкою одинарної точності.

DIVPS – ділить запаковані значення одинарної точності з плаваючою крапкою.

DIVSS – ділить скалярні значення з плаваючою крапкою одинарної точності.

MAXPS – повертає максимальні запаковані значення з плаваючою крапкою одинарної точності.

MAXSS – повертає максимальні скалярні значення з плаваючою крапкою одинарної точності

MINPS – повертає мінімальні запаковані значення з плаваючою крапкою одинарної точності.

MINSS – повертає мінімальні скалярні значення з плаваючою крапкою одинарної точності.

MULPS – множить запаковані значення з плаваючою крапкою одинарної точності.

MULSS – множить скалярні значення з плаваючою крапкою одинарної точності.

RCPPS – обчислює зворотні величини запакованих значень з плаваючою крапкою одинарної точності.

RCPSS – обчислює зворотні значення скалярних значень з плаваючою крапкою одинарної точності.

RSQRTPS – обчислює зворотні величини квадратних коренів із запакованих значень з плаваючою крапкою одинарної точності.

RSQRTSS – обчислює зворотні значення квадратного кореня зі скалярних значень з плаваючою крапкою одинарної точності

SQRTPS – обчислює квадратні корені з запакованих значень з плаваючою крапкою одинарної точності.

SQRTSS – обчислює квадратний корінь зі скалярних значень з плаваючою крапкою одинарної точності.

SUBPS – віднімає запаковані значення з плаваючою крапкою одинарної точності.

SUBSS – віднімає скалярні значення з плаваючою крапкою одинарної точності.

Команди порівняння (SSE):

CMPPS – порівнює запаковані значення з плаваючою крапкою одинарної точності.

CMPSS – порівнює скалярні значення з плаваючою крапкою одинарної точності.

COMISS – виконує впорядковане порівняння скалярних значень з плаваючою крапкою одинарної точності та встановлює прапори в регістрі EFLAGS.

UCOMISS – виконує невпорядковане порівняння скалярних значень з плаваючою крапкою одинарної точності та встановлює прапори в регістрі EFLAGS.

Логічні команди (SSE):

ANDNPS – виконує порозрядну логічну операцію І-НЕ запакованих значень з плаваючою крапкою одинарної точності.

ANDPS – виконує порозрядне логічне І запакованих значень з плаваючою крапкою одинарної точності.

ORPS – виконує порозрядне логічне АБО запакованих значень з плаваючою крапкою одинарної точності.

XORPS – виконує побітове логічне XOR запакованих значень з плаваючою крапкою одинарної точності.

Команди перетасування і розпакування (SSE):

SHUFPS – перетасування значень в запакованих операндах з плаваючою крапкою одинарної точності.

UNPCKHPS – розпаковує та перемежовує два значення високого порядку з двох операндів з плаваючою крапкою одинарної точності.

UNPCKLPS – розпаковує та перемежовує два молодших значення з двох операндів з плаваючою крапкою одинарної точності.

Команди перетворення (SSE) (суфікси: I2 - подвійні цілі, 2PI - подвійні запаковані цілі, 2SI - подвійне скалярне ціле):

CVTPI2PS – перетворювати запаковані цілі числа з подвійними словами в запаковані значення з плаваючою крапкою одинарної точності.

CVTPS2PI – перетворювати запаковані значення з плаваючою крапкою одинарної точності в запаковані цілі числа з подвійним словом.

CVTSI2SS – перетворювати ціле число з подвійним словом у скалярне значення з плаваючою крапкою одинарної точності

CVTSS2SI – перетворити скалярне значення з плаваючою крапкою одинарної точності у ціле число з подвійним словом.

CVTTPS2PI – конвертувати зі скороченими запакованими значеннями одинарної точності з плаваючою крапкою в запаковані подвійні цілі числа.

CVTTSS2SI – перетворити за допомогою скорочення скалярне значення з плаваючою крапкою одинарної точності на скалярне подвійне ціле число.

MXCSR команди керування станом (SSE):

LDMXCSR – завантажити регістр %mxcsr.

STMXCSR – зберегти стан регістру %mxcsr.

64-бітові SIMD цілочислові команди (SSE) для роботи з байтами, словами і подвійними словами на XMM регістрах (префікс P - запаковане (packed):

PAVGB – обчислює середнє для запакованих цілих чисел без знаку.

PAVGW – обчислює середнє для запакованих цілих чисел без знаку.

PEXTRW – видобуває слово.

PINSRW – вставляє слово.

PMAXSW – максимальна кількість запакованих цілих слів зі знаком.

PMAXUB – максимальна кількість запакованих цілих чисел без знаку.

PMINSW – мінімум запакованих цілих слів зі знаком.

PMINUB – мінімум запакованих цілих чисел без знаку.

PMOVMSKB – переміщає байтову маску.

PMULHUW – множить запаковані цілі числа без знаку та зберігає старший результат.

Різні команди (SSE) (керування кешом, попередня вибірка даних, впорядкування команд):

MASKMOVQ – нетимчасове зберігання вибраних байтів із регістра MMX у пам’ять.

MOVNTPS – нетимчасове зберігання чотирьох запакованих значень з плаваючою крапкою одинарної точності з регістра XMM у пам'ять.

MOVNTQ – не тимчасове зберігання чотирного слова з регістра MMX у пам’ять.

PREFETCHNTA – попередньо вибирає дані в нетимчасову структуру кешу та в розташування поблизу процесора.

PREFETCHT0 – попередньо вибирає дані на всіх рівнях ієрархії кешу

PREFETCHT1 – попередньо вибирає дані в кеш рівня 1 і вище.

PREFETCHT2 – попередньо вибирає дані в кеш рівня 2 і вище

SFENCE – серіалізує операції зберігання.

Наступний приклад демонструє перемноження чотирьох пар чисел з плаваючою крапкою однією командою mulps: (Програма написана мовою ANSI C++ з використанням асемблерної вставки __asm і інструкцій асемблера для роботи з SSE)

```
float a[4] = { 300.0, 4.0, 4.0, 12.0 };
float b[4] = {   1.5, 2.5, 3.5,  4.5 };
_asm {
movups xmm0, a   ; // помістити 4 змінні з плаваючою крапкою із a в регістр xmm0
movups xmm1, b   ; // помістити 4 змінні з плаваючою крапкою із b в регістр xmm1

mulps xmm1, xmm0 ; // перемножити вміст регістрів: xmm1=xmm1*xmm0
                   ; // xmm10 = xmm10*xmm00
                   ; // xmm11 = xmm11*xmm01
                   ; // xmm12 = xmm12*xmm02
                   ; // xmm13 = xmm13*xmm03

movups a, xmm1   ; // вивантажити результати із регістра xmm1 за адресою a
};
```

Приклад використання вбудованих функцій SSE в програмі на мові Сі

```
// скалярний добуток векторів
#include <stdio.h>
```

```
#include <stdlib.h>
#include <stdint.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include <xmmintrin.h>
#define N 10000000

int64_t timestamp_now (void)
{
    struct timeval tv;
    gettimeofday (&tv, NULL);
    return (int64_t) tv.tv_sec * CLOCKS_PER_SEC + tv.tv_usec;
}

double timestamp_to_seconds (int64_t timestamp)
{
    return timestamp / (double) CLOCKS_PER_SEC;
}

float inner1(float *x,float *y,int n)
{
     float s;
     int i;
     s=0;
     for(i=0;i<n;i++) s+=x[i]*y[i];
     return s;
}
```

// функція з використанням внутрішніх особливостей SSE

```
float inner2(float *x,float *y,int n)
{
     float sum;
     int i;
     __m128 *xx,*yy;
     __m128 p,s;
    xx=(__m128 *)x;
    yy=(__m128 *)y;
    s=_mm_set_ps1(0);

   for (i=0;i<n/4;i++)
   {
          // попередня вибірка даних в кеш (на декілька ітерацій вперед)
          _mm_prefetch((char *)&xx[i+4],_MM_HINT_NTA);
          _mm_prefetch((char *)&yy[i+4],_MM_HINT_NTA);
          p=_mm_mul_ps(xx[i], yy[i]); // векторне множення чотирьох чисел
          s=_mm_add_ps(s,p);          // векторне додавання чотирьох чисел
   }

 p=_mm_movehl_ps(p,s); // переміщення двох старших значень s в молодші p
 s=_mm_add_ps(s,p);    // векторне додавання
 p=_mm_shuffle_ps(s,s,1); //переміщення другого значення з s в молодшу
позицію в p
 s=_mm_add_ss(s,p);    // скалярне додавання
 _mm_store_ss(&sum,s); // запис молодшого значення в пам'ять
 return sum;

int main()
{
     float *x,*y,s;
     long t;
     int i;
```

```
        int64_t start, stop;
        double diff;

        // виділення пам'яті з вирівнюванням
        x=(float *)_mm_malloc(N*sizeof(float),16);
        y=(float *)_mm_malloc(N*sizeof(float),16);
        for (i=0;i<N;i++) { x[i]=10*i/N; y[i]=10*(N-i-1)/N; }

        // використання x87
        start = timestamp_now();
        s=inner1(x,y,N);
        stop = timestamp_now();
        diff = timestamp_to_seconds(stop - start);
        printf("Time x87: %f sec\n",diff);

        // Використання SSE
        start = timestamp_now();
        s=inner2(x,y,N);
        stop = timestamp_now();
        diff = timestamp_to_seconds(stop - start);
        printf("Time SSE: %f sec\n",diff);

        _mm_free(x); _mm_free(y);
        return 0;
}

$ gcc 1. -o 1
$ ./1
Time x87: 0.076518 sec
Time SSE: 0.040590 sec
```

2. Розширення команд SSE2

Розширення команд SSE2 (2001, Pentium 4) використовують XMM регістри для операцій з наступними типами даних:

• два 64-бітові числа з плаваючою крапкою подвійної точності;

• два 64-бітові цілі;

• чотири 32-бітові цілі;

• вісім 16-бітових коротких цілих;

• шістнадцять 8-бітових байтів або символів.

Перевага в швидкості обчислень досягається в тому випадку, коли необхідно виконати одну й ту ж послідовність дій над різними даними.

Особливості команд SSE2:

- використовують вісім **128-бітових регістрів** (xmm0,...,xmm7), які добавлені в архітектуру x86, і кожен з яких трактується як два послідовні значення з плаваючою крапкою подвійної точності.

- має набір команд, які виконують операції зі скалярними і запакованими типами даних.

- має команди для потокової обробки цілочислових даних у 128-бітових xmm регістрах, що дозволяє відмовитися від цілочислових команд MMX.

- включає дві частини – розширення команд SSE і розширення команд MMX.

- працює з дійсними числами.

- має ряд команд управління кешем, призначених для мінімізації забруднення кешу при обробці невизначених потоків інформації.

- має складні доповнення до команд перетворення чисел.

Розширення команд SSE2 поділені на чотири підгрупи:

• команди для запакованих і скалярних значень з плаваючою крапкою подвійної точності;

• команди перетворення запакованих значень з плаваючою крапкою одинарної точності;

• 128-бітові SIMD цілочислові команди;

• команди контролю кешу і впорядкування функціональності.

SSE2 Команди для запакованих і скалярних значень з плаваючою крапкою подвійної точності

SSE2 Команди пересилання даних з плаваючою крапкою подвійної точності між XMM регістрами і пам'яттю.

MOV**APD** – перемістити два вирівняних запакованих значення з плаваючою крапкою подвійної точності між регістрами XMM і пам’яттю

MOV**HPD** – переміщення старше запакованого значення подвійної точності з плаваючою крапкою до або з старшого чотирного слова регістра та пам’яті XMM

MOV**LPD** – перемістити молодше запаковане значення з плаваючою крапкою одинарної точності до або з молодшого чотирного слова регістра та пам’яті XMM

MOVMSK**PD** – витягти маску знака з двох запакованих значень з плаваючою крапкою подвійної точності

MOV**SD** – перемістити скалярне значення з плаваючою крапкою подвійної точності між регістрами XMM і пам’яттю

MOV**UPD** – перемістити два невирівняних запакованих значення з плаваючою крапкою подвійної точності між регістрами XMM і пам’яттю

SSE2 Команди запакованої арифметики

ADD**PD** – додати запаковані значення подвійної точності з плаваючою крапкою

ADD**SD** – додати скалярні значення подвійної точності з плаваючою крапкою

DIV**PD** – ділити запаковані значення з плаваючою крапкою подвійної точності

DIV**SD** – ділити скалярні значення подвійної точності з плаваючою крапкою

MAX**PD** – повертає максимальні запаковані значення з плаваючою крапкою подвійної точності

MAX**SD** – повертає максимальне скалярне значення з плаваючою крапкою подвійної точності

MIN**PD** – повертає мінімальні запаковані значення з плаваючою крапкою подвійної точності

MIN**SD** – повертає мінімальне скалярне значення з плаваючою крапкою подвійної точності

MUL**PD** – багаторазово запаковані значення з плаваючою крапкою подвійної точності

MUL**SD** – множити скалярні значення подвійної точності з плаваючою крапкою

SQRT**PD** – обчислювати запаковані квадратні корені з запакованих значень з плаваючою крапкою подвійної точності

SQRT**SD** – обчислити скалярний квадратний корінь зі скалярного значення з плаваючою крапкою подвійної точності

SUB**PD** – відняти запаковані значення з плаваючою крапкою подвійної точності

SUB**SD** – відняти скалярні значення подвійної точності з плаваючою крапкою.

SSE2 Логічні команди

ANDN**PD** – побітове логічне І-НЕ запакованих значень з плаваючою крапкою подвійної точності

AND**PD** – порозрядне логічне І запакованих значень з плаваючою крапкою подвійної точності

OR**PD** – порозрядне логічне АБО запакованих значень з плаваючою крапкою подвійної точності

XOR**PD** – побітове логічне XOR запакованих значень з плаваючою крапкою подвійної точності

SSE2 Команди порівняння:

CMP**PD** – порівняти запаковані значення з плаваючою крапкою подвійної точності

CMP**SD** – порівнювати скалярні значення з плаваючою крапкою подвійної точності

COM**ISD** – виконувати впорядковане порівняння скалярних значень з плаваючою крапкою подвійної точності та встановлювати прапорці в регістрі EFLAGS

UCOM**ISD** – виконувати невпорядковане порівняння скалярних значень з плаваючою крапкою подвійної точності та встановлювати прапорці в регістрі EFLAGS

SSE2 Команди перетасування та розпакування

SHUF**PD** – перетасувати значення в запакованих операндах з плаваючою крапкою подвійної точності

UNPCKH**PD** – розпаковувати та чергувати старші значення з двох запакованих операндів із плаваючою крапкою подвійної точності

UNPCKL**PD** – розпаковувати та чергувати молодші значення з двох запакованих операндів з плаваючою крапкою подвійної точності

SSE2 Команди перетворення

CVT**DQ2PD** – перетворювати запаковані цілі числа з подвійної точності у запаковані значення з плаваючою крапкою подвійної точності

CVT**PD2DQ** – перетворювати запаковані значення подвійної точності з плаваючою крапкою в запаковані цілі числа подвійної точності.

CVT**PD2PI** – перетворювати запаковані значення подвійної точності з плаваючою крапкою в запаковані цілі числа з подвійним словом

CVT**PD2PS** – перетворювати запаковані значення з плаваючою крапкою подвійної точності в запаковані значення з плаваючою крапкою одинарної точності

CVT**PI2PD** – перетворювати запаковані цілі числа з подвійним словом у запаковані значення з плаваючою крапкою подвійної точності

CVT**PS2PD** – перетворювати запаковані значення з плаваючою крапкою одинарної точності в запаковані значення з плаваючою крапкою подвійної точності

CVT**SD2SI** – перетворювати скалярні значення з плаваючою крапкою подвійної точності на ціле число з подвійним словом

CVT**SD2SS** – перетворювати скалярні значення з плаваючою крапкою подвійної точності на скалярні значення з плаваючою крапкою одинарної точності

CVT**SI2SD** – перетворювати ціле число з подвійним словом у скалярне значення з плаваючою крапкою подвійної точності

CVT**SS2SD** – перетворювати скалярні значення з плаваючою крапкою одинарної точності на скалярні значення з плаваючою крапкою з подвійною точністю

CVTT**PD2DQ** – конвертувати зі скороченими упакованими значеннями подвійної точності з плаваючою крапкою в упаковані подвійні цілі числа

CVTT**PD2PI** – конвертувати з скороченнням запаковані значення подвійної точності з плаваючою крапкою в запаковані подвійні цілі числа

CVTT**SD2SI** – перетворити з скороченнням скалярні значення подвійної точності з плаваючою крапкою за допомогою скорочення на скалярні цілі числа з подвійним словом

SSE2 Команди запакованих чисел одинарної точності з плаваючою крапкою

Виконують дії над операндами одинарної точності з плаваючою крапкою і цілими.

CVT**DQ2PS** – перетворювати запаковані цілі числа з подвійними словами в запаковані значення з плаваючою крапкою одинарної точності

CVT**PS2DQ** – перетворювати запаковані значення з плаваючою крапкою одинарної точності в запаковані цілі числа з подвійним словом

CVTT**PS2DQ** – конвертувати зі скороченням запаковані значення одинарної точності з плаваючою крапкою в запаковані подвійні цілі числа

SSE2 128–Bit SIMD цілочислові команди

MOV**DQ2Q** – переміщення ціле чотирне слово з регістрів XMM до MMX

MOV**DQA** – перемістити вирівняне подвійне чотирне слово

MOV**DQU** – перемістити невирівняне подвійне чотирне слово

MOV**Q2DQ** – перемістити ціле чотирне слово з регістрів MMX в XMM

PADD**Q** – додати запаковані цілі числа з чотирьох слів

PMUL**UDQ** – множення запаковані беззнакові подвійні цілі числа

PSHUF**D** – перетасувати запаковані подвійні слова

PSHUF**HW** – перетасувати старші слова

PSHUF**LW** – перетасувати молодші слова

PSLL**DQ** – зсув подвійного чотирного слова ліворуч логічний

PSRL**DQ** – зсув подвійного чотирного слова праворуч логічний

PSUB**Q** – відняти запаковані цілі числа з чотирьох слів

PUNPCK**HQDQ** – розпакувати старше чотирне слово

PUNPCK**LQDQ** – розпакувати молодше чотирне слово

SSE2 Різні команди

CLFLUSH – очищає та робить недійсним операнд пам'яті та пов'язаний з ним рядок кешу з усіх рівнів ієрархії кешу процесора

LFENCE – серіалізує операції завантаження

MASKMOVDQU – нетимчасове зберігання вибраних байтів із регістра XMM у пам’ять

MFENCE – серіалізує операції завантаження та зберігання

MOVNTDQ – нетимчасове зберігання подвійного чотирного слова з регістра XMM у пам’ять

MOVNTI – нетимчасове зберігання подвійного слова з регістра загального призначення в пам'ять

MOVNTPD – нетимчасове зберігання двох упакованих значень із плаваючою крапкою подвійної точності з регістра XMM у пам’ять

PAUSE – покращує продуктивність циклів обертання-очікування

Приклад використання розширення команд SSE2 в афінних перетвореннях (при яких паралельні прямі переходять у паралельні прямі, пересічні – в пересічні, мимобіжні – в мимобіжні):

```
;Завантаження трансформаційної матриці у xmm0-xmm2 регістри:
; register     Low  | High
;-------------------------
; xmm0         m00  | m10
; xmm1         m01  | m11
; xmm2         m20  | m21
movsd  xmm0,_m00
movhpd xmm0,_m10
movsd  xmm1, _m01
movhpd xmm1, _m11
movsd  xmm2, _m20
movhpd xmm2, _m21

;завантаження меж обрізання у xmm6, xmm7 регістри
; register     Low  | High
;-------------------------
; xmm6         xMin | yMin
; xmm7         xMax | yMax

movsd   xmm6,   _xMin
movhpd  xmm6,   _yMin
movsd   xmm7,   _xMax
movhpd  xmm7,   _yMax
;ознака розриву завантажується у mm7 регістр
mov eax, 80000000h
push eax
push eax
movq mm7, [esp]
add esp,8h
xor eax, eax
push eax     ;виділення місця в стеку для лічильника вихідних точок.
xor ebx, ebx ;очищення регістру для зберігання ознаки розриву
;---------------------------------------------------------------------
;Функція виконує афінне перетворення точки в ptr[esi]
;збільшує [esi], щоб вказувати на наступну точку
;обчислена точка зберігається в xmm3 регістрі, Low - X, High-Y
;---------------------------------------------------------------------
TransformPoint PROC PRIVATE:
   movupd  xmm3, [esi] ;xmm3 = x,y
   movapd xmm4,xmm3    ;xmm4 = x,y
   ;Calculation
   mulpd xmm3, xmm0    ;xmm3 = M00*X | M10*Y
   mulpd xmm4, xmm1    ;xmm4 = M01*X | M11*Y
   haddpd xmm3,xmm4    ;xmm3 = M00*X + M10*Y | M01X+M11Y
   addpd xmm3, xmm2    ;xmm3 = M00*X + M10*Y+ M20 | M01*X + M11*Y + M20
   add  esi, 10h       ;збільшення esi для наступного виклику функції
   ret 0
TransformPoint ENDP
```

3. Розширення команд SSE3

Розширення команд SSE3 (2004, Pentium Prescott) має додаткових 13 команд порівняно з розширенням команд SSE2. Найбільш помітна зміна – можливість **горизонтальної роботи з регістрами**. Тобто добавлені команди додавання та віднімання кількох значень, що зберігаються в одному регістрі. Ці команди спростили низку DSP- та 3D-операцій. Добавлена також нова команда для перетворення значень з плаваючою крапкою на цілі без необхідності вносити зміни в глобальному режимі округлення.

Нові команди:

Арифметичні:

- ADDSUB**PD** – (*add-subtract-packed-double*)
  - Input: { A0, A1 }, { B0, B1 }
  - Output: { A0 − B0, A1 + B1 }
- ADDSUB**PS** – (*add-subtract-packed-single*)
  - Input: { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Output: { A0 − B0, A1 + B1, A2 − B2, A3 + B3 }

AOS (Array Of Structures — Масив структур)

- **H**ADD**PD** – (*horizontal-add-packed-double*)
  - Input: { A0, A1 }, { B0, B1 }
  - Output: { A0 + A1, B0 + B1 }
- **H**ADD**PS** (*horizontal-add-packed-single*)
  - Input: { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Output: { A0 + A1, A2 + A3, B0 + B1, B2 + B3 }
- **H**SUB**PD** — (*horizontal-subtract-packed-double*)
  - Input: { A0, A1 }, { B0, B1 }
  - Output: { A0 − A1, B0 − B1 }
- **H**SUB**PS** – (*horizontal-subtract-packed-single*)
  - Input: { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Output: { A0 − A1, A2 − A3, B0 − B1, B2 − B3 }
- LDD**QU** – завантаження невирівняного цілочисельного значення 128-біт.
- MOV**DDUP** – тиражує дійсні числа подвійної точності. Використовується при роботі з комплексними числами.
- MOV**SHDUP** – тиражує старшу частину дійсного числа подвійної точності
- MOV**SLDUP** – тиражує молодшу частину дійсного числа подвійної точності.

4. Розширення команд SSSE3

Новими в SSSE3, у порівнянні з SSE3 є **16 команд, які працюють із запакованими цілими.** Кожна з них може працювати і з 64-бітовими (MMX), і з 128-бітовими (XMM) регістрами, тому Intel у своїх матеріалах посилається на 32 нові команди:

- Дванадцять інструкцій, які виконують горизонтальні операції додавання або віднімання.
- Шість інструкцій, які оцінюють абсолютні значення.

- Дві інструкції, які виконують операції множення та додавання та прискорюють оцінку скалярного добутку.

- Дві інструкції, які прискорюють операції множення запакованого цілого числа та виробляють цілі значення з масштабуванням.

- Дві інструкції, які виконують побайтове перемішування на місці відповідно до другого операнда керування перемішуванням.

- Шість інструкцій, які змінюють знак запакованих цілих чисел в операнді призначення, якщо знаки відповідного елемента в операнді джерела менші за нуль.

- Дві інструкції, які вирівнюють дані зі складу двох операндів.

Нові команди

Робота зі знаком.

- **P**ABS[**B,W,D**] – *(packed absolute value {bytes/words/dwords})*
    - Вхід – { A0, A1… }
    - Вихід – { A0 * sign(A0), A1 * sign (A1)… }

Кожне поле результату є абсолютна величина відповідного поля з src1. Фактично це ті ж операції PSIGNB, PSIGNH, PSIGNW, у яких обидва аргументи один і той же регістр.

- **P**SIGN[**B**,**W**,**D**] – *(packed sign {bytes/words/dwords})*
    - Вхід – { A0, A1… }, { B0, B1… }
    - Вихід – { A0 * sign(B0), A1 * sign (B1)… }

Кожне поле результату є добуток поля з src1 на {-1,0,1} в залежності від знаку відповідного поля в src2 (множення на 0, коли поле в src2 дорівнює нулю).

Зсуви

- **P**ALIGN**R** – *(packed align right)*
    - Вхід – { A0, A1 }, { B0, B1 }, imm8
    - Вихід – { B1_B0_A1_A0 >> (imm8 * 8) }

Два регістра операнда розглядаються як одне беззнакове значення подвоєної розмірності, з якого виймається 64-/128-бітове значення починаючи з байта, вказаного у безпосередньому аргументі-константі команди.

Перетасування байтів

- **P**SHUF**B** – *(packed shuffle bytes)*
    - Вхід – { A_0, A_1, A_2,.. A_7/A_{15} }, { B_0, B_1, B_2,.. B_7/B_{15} }
    - Вихід – { [A_{B0} A_{B1} A_{B2} …] }

Переставлення байтів, кожен байт результату є деякий байт з першого аргументу, який визначається за відповідним байтом другого аргументу (якщо байт від'ємний, то в байт результату записується нуль, у іншому випадку використовуються молодші 3-й або 4-й біти як номер байта у першому аргументі).

Множення

- **P**MULHRS**W** – *(packed multiply high with round and scale)*
    - Вхід – { A0, A1… }, { B0, B1… }
    - Вихід – { A0 * B0, A1 * B1… }

Аргументи A і B розглядаються як вектори 16-бітових знакових чисел з фіксованою крапкою, представлених у діапазоні [-1,+1) (тобто 0x4000 — це 0.5, а 0xa000 — це −0.75 і т. д.), які множать одне на друге з коректним округленням.

- **P**MADD**UBSW** – *(multiply and add packed signed and unsigned bytes)*
  - Вхід – { A0, A1, A2, A3,.. }, { B0, B1, B2, B3,.. }
  - Вихід – { (A0*B0+A1*B1), (A2*B2+A3*B3), … }

Виконується побайтове перемноження векторів A і B, проміжні 16-бітові результати попарно складається між собою з насиченням і видаються як результат.

Горизонтальні додавання/віднімання цілих

- **PH**SUB[**W,D**] – *(packed horizontal subtract (16-ти- або 32-х бітові поля))*
  - Вхід – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Вихід – { A0-A1 A2-A3 … B0-B1 B2-B3 …}

Горизонтальне віднімання цілих 16-/32-бітових полів.

- **PH**SUB**SW** – *(packed horizontal subtract and saturate words (16-бітові поля))*
  - Вхід – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Вихід – { A0-A1 A2-A3 B0-B1 B2-B3 }

Горизонтальне віднімання цілих 16-бітових полів з насиченням.

- **PH**ADD[**W,D**] – *(packed horizontal add (16-ти- або 32-х бітові поля))*
  - Вхід – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Вихід – { A0+A1 A2+A3 … B0+B1 B2+B3 …}

Горизонтальне додавання цілих 16-/32-бітових полів.

- **PH**ADD**SW** – *(packed horizontal add and saturate words (16-бітові поля))*
  - Вхід – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
  - Вихід – { A0+A1 A2+A3 … B0+B1 B2+B3 …}

Горизонтальне додавання цілих 16-бітових полів з насиченням.

5. Розширення команд SSE4

Розширення команд SSE4 (2007, Penryn) – набір команд мікроархітектури Intel Core, вперше реалізований у процесорах серії Penryn.

SSE4 складається з 54 команд, 47 з них відносять до SSE4.1 (вони є у процесорах Penryn). Повний набір команд (SSE4.1 і SSE4.2, тобто 47 + 7 команд, що залишилися) доступний у процесорах Intel з мікроархітектурою Nehalem (2008 рік) і пізніших редакціях.

Розширення команд SSE4 працює тільки з **128-бітовими регістрами** xmm0, ... ,xmm15 (64-бітові mmx регістри не використовуються).

Зміни. Додано команди, які прискорюють компенсацію руху у відеокодеках, команди швидкого читання з USWC* (Uncacheable Speculative Write Combining) пам'яті, множину команд для спрощення векторизації програм компіляторами. Крім того, в SSE4.2 додано команди обробки рядків 8/16-бітових символів, обчислення CRC32, POPCNT. Вперше у SSE4 регістр xmm0 став використовуватися як неявний аргумент для деяких команд.

Компілятор мови Сі від Intel, починаючи з версії 10, генерує команди SSE4 при заданні опції -QxS. Компілятор Sun Studio від Sun Microsystems починаючи з версії 12 update 1 генерує команди SSE4 за допомогою опцій -xarch=sse4_1 (SSE4.1) та -xarch=sse4_2 (SSE4.2). Компілятор GCC підтримує SSE4.1 та SSE4.2 починаючи з версії 4.3, опції -msse4.1 та -msse4.2, або -msse4, що включає обидва варіанти.

*Примітка. USWC – це налаштування BIOS комп'ютера для зв'язку пам'ятей ЦП і графічної карти. Таке налаштування забезпечує швидший зв'язок, ніж параметр «не кешувати», якщо графічна карта підтримує об'єднання запису. Тоді дані тимчасово зберігаються в буферах об'єднання запису (WCB) і пересилаються в пакетному режимі, а не окремими бітами.

Нові команди.

Прискорення відео.

MPSAD**BW** xmm1, xmm2/m128, imm8 – *(multiple packed sums of absolute difference)*

- Input – { A_0, A_1,… A_{14} }, { B_0, B_1,… B_{15} }, Shiftmode
- Output – { SAD_0, SAD_1, SAD_2,… SAD_7 }

Обчислення восьми сум абсолютних значень різниць (SAD) зміщених 4-байтових беззнакових груп. Розміщення операндів для 16-бітових SAD визначається трьома бітами безпосереднього аргументу imm8.

```
s1 = imm8[2]*4
s2 = imm8[1:0]*4
SAD0 = |A(s1+0)-B(s2+0)| + |A(s1+1)-B(s2+1)| + |A(s1+2)-B(s2+2)| + |A(s1+3)-B(s2+3)|
SAD1 = |A(s1+1)-B(s2+0)| + |A(s1+2)-B(s2+1)| + |A(s1+3)-B(s2+2)| + |A(s1+4)-B(s2+3)|
SAD2 = |A(s1+2)-B(s2+0)| + |A(s1+3)-B(s2+1)| + |A(s1+4)-B(s2+2)| + |A(s1+5)-B(s2+3)|
...
SAD7 = |A(s1+7)-B(s2+0)| + |A(s1+8)-B(s2+1)| + |A(s1+9)-B(s2+2)| + |A(s1+10)-B(s2+3)|
```

PHMINPOS**UW** xmm1, xmm2/m128 – *(packed horizontal word minimum)*

- Input – { A_0, A_1,… A_7 }
- Output – { MinVal, MinPos, 0, 0… }

Пошук серед 16-бітових беззнакових полів A_0…A_7 такого, яке має мінімальне значення (і позицію з меншим номером, якщо таких полів декілька). Повертається 16-бітове значення і його позиція.

- PMOV[sx,zx][b,w,d] xmm1, xmm2/m[64,32,16] – *(packed move with sign/zero extend)*

Група з 12-ти команд для розширення формату запакованих полів. Запаковані 8, 16, або 32-бітові поля з молодшої частини аргументу розширюються (із знаком або без) в 16, 32 або 64-бітові поля результату.

Вхідний формат			**Результуючий формат**
8 біт	**16 біт**	**32 біти**	
PMOV**SXBW**			16 біт
PMOV**ZXBW**	**P**MOV**ZXWW**		
PMOV**SXBD**	**P**MOV**SXWD**		32 біти
PMOV**ZXBD**	**P**MOV**ZXWD**	**P**MOV**SXDD**	
PMOV**SXBQ**	**P**MOV**SXWQ**	**P**MOV**SXDQ**	64 біти
PMOV**ZXBQ**	**P**MOV**ZXWQ**	**P**MOVZX**DQ**	

Векторні примітиви

P[MIN,MAX][SB,UW,SD,UD] xmm1, xmm2/m128 – *(minimum/maximum of packed signed/unsigned byte/word/dword integers)*

Кожне поле результату є мінімальним/максимальним значенням відповідних полів двох аргументів. Байтові поля розглядаються тільки як числа зі знаком, 16-бітові – тільки як числа без знаку. Для 32-бітових запакованих полів передбачено варіант як зі знаком, так і без.

- **P**MUL**DQ** xmm1, xmm2/m128 – *(multiply packed signed dword integers)*
  - Input – { A_0, A_1, A_2, A_3 }, { B_0, B_1, B_2, B_3 }
  - Output – { A_0*B_0, A_2*B_2 }

Перемноження 32-бітових полів зі знаком із видачею повних 64 біт результату (дві операції множення над 0 та 2 полями аргументів).

- **P**MUL**LD** xmm1, xmm2/m128 – *(multiply packed signed dword integers and store low result)*
  - Input – { A_0, A_1, A_2, A_3 }, { B_0, B_1, B_2, B_3 }
  - Output – { low32(A_0*B_0), low32(A_1*B_1), low32(A_2*B_2), low32(A_3*B_3) }

Перемноження 32-бітових полів зі знаком та видачу молодших 32 біт результатів (чотири операції множення над усіма полями аргументів).

- PACK**USDW** xmm1, xmm2/m128 – *(pack with unsigned saturation)*

Пакування 32-бітових полів зі знаком у 16-бітові поля без знаку з насиченням.

- **P**CMPEQ**Q** xmm1, xmm2/m128 – *(Compare Packed Qword Data for Equal)*

Перевірка 64-бітових полів на рівність та видача 64-бітових масок.

Вставки/Видобування.

- INSERT**PS** xmm1, xmm2/m32, imm8 – *(insert packed single precision floating-point value)*

Вставка 32-бітового поля з xmm2 (можливо вибрати будь-який з 4 полів цього регістра) або з 32-бітової комірки пам'яті в довільне поле результату. Крім того, для кожного з полів результату можна задати скидання його +0.0.

- EXTRACT**PS** r/m32, xmm, imm8 – *(extract packed single precision floating-point value)*

Видобування 32-бітового поля з xmm регістру, номер поля вказується в молодших 2 бітах imm8. Якщо як результат вказано 64-бітовий регістр, його старші 32 біти скидаються (розширення без знака).

- **P**INS**R[**b,d,q] xmm, r/m*, imm8 – *(insert byte/dword/qword)*

Вставка 8, 32, або 64-бітового значення у вказане поле xmm регістру (інші поля не змінюються).

- **P**EXTR[b,w,d,q] r/m*, xmm, imm8 — *(extract byte/word/dword/qword)*

Видобування 8, 16, 32, 64-бітового поля з вказаного в imm8 поля xmm регістра. Якщо як результат вказано регістр, його старша частина скидається (розширення без знака).

Скалярне множення векторів

- DP**PS** xmm1, xmm2/m128, imm8 – *(dot product of packed single precision floating-point values)*

- DP**PD** xmm1, xmm2/m128, imm8 – *(dot product of packed double precision floating-point values)*

Скалярне множення векторів (dot product) 32/64-бітових полів. За допомогою бітової маски в imm8 вказується, які добутки полів повинні підсумовуватись і що слід прописати у кожне поле результату: суму зазначених добутків або +0.0.

Змішування

- BLENDV[ps,pd] xmm1, xmm2/m128, <xmm0> – *(variable blend packed single/double precision floating-point values)*

Вибір кожного 32/64-бітового поля результату здійснюється в залежності від знаку такого ж поля в неявному аргументі xmm0: або з першого або другого аргументу.

- BLEND[ps,pd] xmm1, xmm2/m128, imm8 – *(blend packed single/double precision floating-point values)*

Бітова маска (4 або 2 біти) в imm8 вказує, з якого аргументу слід взяти кожне 32/64-бітове поле результату.

- **P**BLENDV**B** xmm1, xmm2/m128, <xmm0> – *(variable blend packed bytes)*

Вибір кожного байтового поля результату здійснюється в залежності від знаку байта такого ж поля в неявному аргументі xmm0: або з першого або другого аргументу.

- **P**BLEND**W** xmm1, xmm2/m128, imm8 – *(blend packed words)*

Бітова маска (8 біт) в imm8 вказує, з якого аргументу слід взяти кожне 16-бітове поле результату.

Перевірки бітів

- **P**TEST xmm1, xmm2/m128 – *(logical compare)*

Встановити прапор ZF, якщо в xmm2/m128 всі біти позначені маскою з xmm1 дорівнюють нулю. Якщо всі не позначні біти дорівнюють нулю, то встановити прапор CF. Інші прапори (AF, OF, PF, SF) завжди скидаються. Інструкція не модифікує xmm1.

Округлення

- ROUND [ps, pd] xmm1, xmm2/m128, imm8 – *(round packed single/double precision floating-point values)*

Округлення всіх 32/64-бітових полів. Режим округлення (4 варіанти) вибирається або MXCSR.RC, або задається безпосередньо в imm8. Також можна подавити генерацію відключення втрати точності.

- ROUND [ss, sd] xmm1, xmm2/m128, imm8 – *(round scalar single/double precision floating-point values)*

Округлення лише молодшого 32/64-бітового поля (інші біти залишаються незмінними).

Читання WC пам’яті

- MOVNTDQA xmm1, m128 – *(load double quadword non-temporal aligned hint)*

Операція читання, що дозволяє прискорити (до 7.5 разів) роботу з write-combining областями пам'яті.

Нові інструкції SSE4.2

Оброблення стрічок символів

Ці інструкції виконують арифметичні порівняння між усіма можливими парами полів (64 або 256 порівнянь) з обох рядків, заданих вмістом xmm1 та xmm2/m128 (memory128). Потім булеві результати порівнянь обробляються для отримання потрібних результатів. Безпосередній аргумент imm8 керує розміром (байтові чи unicode рядки, до 16/8 елементів кожний), знаковістю полів (елементів рядків), типом порівняння та інтерпретацією результатів.

Ними можна здійснювати у рядку (області пам'яті) пошук символів із заданого набору чи заданих діапазонах. Можна порівнювати рядки (області пам'яті) або шукати підрядки.

Всі вони впливають на прапори процесора: SF встановлюється якщо xmm1 не повний рядок, ZF – якщо xmm2/m128 не повний рядок, CF – якщо результат не нульовий, OF – якщо молодший біт результату не нульовий. Прапори AF та PF скидаються.

- **P**CMPESTRI <ecx>, xmm1, xmm2/m128, <eax>, <edx>, imm8 – *()*

Явне завдання розміру рядків <eax>, <edx> (береться абсолютна величина регістрів з насичення до 8/16, залежно від розміру елементів рядків. Результат у регістрі ecx.

- **P**CMPESTRM <xmm0>, xmm1, xmm2/m128, <eax>, <edx>, imm8 – *()*

Явне задання розміру рядків <eax>, <edx> (береться абсолютна величина регістрів з насичення до 8/16, залежно від розміру елементів рядків. Результат у регістрі xmm0.

- **P**CMPISTRI <ecx>, xmm1, xmm2/m128, imm8 – *()*

Неявне задання розміру рядків (здійснюється пошук нульових елементів до кожного рядка). Результат у регістрі ecx.

- **P**CMPISTRM <xmm0>, xmm1, xmm2/m128, imm8 – *()*

Неявне задання розміру рядків (здійснюється пошук нульових елементів до кожного рядка). Результат у регістрі xmm0.

Підрахунок CRC32

- CRC32 r32, r/m* – *(підрахунок crc32)*

Накопичення значення CRC-32C (інші позначення CRC-32/ISCSI CRC-32/CASTAGNOLI) для 8, 16, 32 або 64-бітового аргументу (використовується поліном 0x1EDC6F41).

Підрахунок популяції одиничних бітів

- POPCNT r, r/m* – *(return the count of number of bits set to 1)*

Підрахунок числа одиничних бітів. Три варіанти інструкції: для 16, 32 та 64-бітових регістрів.

Векторні примітиви

- **P**CMPGT**Q** xmm1, xmm2/m128 – *(compare packed qword data for greater than)*

Перевірка 64-бітових полів на «більше ніж» та видача 64-бітових масок.

Контрольні запитання.

1. Призначення і особливості команд розширення SSE.
2. Призначення і особливості команд розширення SSE2.
3. Призначення і особливості команд розширення SSE3.
4. Призначення і особливості команд розширення SSSE3.
5. Призначення і особливості команд розширення SSE4.

11. Команди розширення x86-AVX, AVX2, AVX-512

Мета. Вивчення команд розширення x86-AVX, AVX2, AVX-512.

Вступ. Команди розширення x86-AVX, AVX2, AVX-512 – це команди потокової обробки та виконання арифметичних операцій над запакованими цілими і дійсними числами.

План.

1. Команди розширення AVX.
2. Історія розвитку AVX
3. Середовище виконання x86-AVX
   3.1 Набір регістрів x86-AVX
   3.2. Типи даних x86-AVX
   3.3. Синтаксис команд x86-AVX
   3.4. Особливості розширень x86-AVX
4. Огляд команд x86-AVX
   4.1. Розширені команди x86-SSE
5. Нові команди
   5.1. Широкомовна трансляція
   5.2. Команди змішування
   5.3. Команди переставлення
   5.4. Видобування і вставка
   5.5. Масковане переміщення
   5.6. Змінний бітовий зсув
   5.7. Команди збирання
6. Команди з розширеними функціями
   6.1. Команди половинної точності з плаваючою крапкою
   6.2. Команди злитого множення-додавання (FMA)
   6.2.1. Підгрупа VFMADD
   6.2.2. Підгрупа VFMSUB
   6.2.3. Підгрупа VFMADDSUB
   6.2.4. Підгрупа VFMSUBADD
   6.2.5. Підгрупа VFNMADD
   6.2.6. Підгрупа VFNMSUB
   6.2.7. Регістри загального призначення
7. Команди AVX-512
   7.1. Запаковані типи даних
   7.2. Регістр mask і масковані операції
   7.3. Генерація масок

1. Команди розширення AVX

Найновіше розширення SIMD для x86 називається Advanced Vector Extensions (x86-AVX). x86-AVX додає нові регістри, типи даних та команди для платформи x86. x86-AVX надає сучасні команди мови асемблера з трьома операндами, що допомагає оптимізувати програми на

мові асемблера та покращити їх продуктивність. Разом із запровадженням x86-AVX з'явилося кілька окремих розширень функцій, включаючи перетворення чисел з плаваючою крапкою половинної точності, операції злитого множення-додавання (FMA), і нові команди для регістрів загального призначення.

Для розгляду команд x86-AVX взято середовище виконання x86-32, розуміння якого дозволяє також використовувати і середовище x86-64.

Розширення AVX підходить для інтенсивних обчислень з плаваючою крапкою в мультимедіа програмах та наукових завданнях. Там, де можлива більш висока ступінь паралелізму, збільшується продуктивність з дійсними числами.

В AVX використовуються 256-розрядні регістри ymm0-ymm15 і система команд з трьома операндами.

Набір команд AVX запроваджує наступні нововведення:

- Нова схема кодування інструкцій VEX.
- Ширина векторних регістрів SIMD збільшується з 128 (XMM) до 256 бітів (регістри YMM0 - YMM15). Існуючі 128-бітові SSE інструкції будуть використовувати молодшу половину нових YMM регістрів, не змінюючи старшу частину. Для роботи з YMM регістрами додані нові 256-бітові AVX інструкції (новіший стандарт AVX-512 розширює векторні регістри SIMD до 512 біт). Процесорна архітектура Intel Larrabee мала векторні регістри ZMM шириною в 512 біт, і використовувала для роботи з ними SIMD команди з префіксами MVEX і VEX, але при цьому вони не підтримували AVX.
- Неруйнівні операції. Набір AVX команд використовує 3-операндний синтаксис. Наприклад, замість $a = a + b$ можна використовувати $c = a + b$, при цьому регістр a залишається незміненим. У випадках, коли значення a використовується далі в обчисленнях, це підвищує продуктивність, оскільки позбавляє від необхідності зберігати перед обчисленням і відновлювати після обчислення регістр, що містив a, з іншого регістру або пам'яті.
- Для більшості нових інструкцій відсутні вимоги до вирівнювання операндів в пам'яті. Однак, рекомендується стежити за вирівнюванням на розмір операнда, щоб уникнути значного зниження продуктивності.
- Набір інструкцій AVX містить в собі аналоги 128-бітових SSE інструкцій для дійсних чисел. При цьому, на відміну від оригіналів, збереження 128-бітового результату буде обнуляти старшу половину YMM регістру. 128-бітові AVX інструкції зберігають інші переваги AVX, такі як нова схема кодування, 3-операндний синтаксис і невирівняний доступ до пам'яті. Рекомендується відмовитися від старих SSE інструкцій на користь нових 128-бітових AVX інструкцій, навіть якщо достатньо двох операндів.

Нова схема кодування

Нова схема кодування інструкцій VEX[en] використовує префікси VEX. Існують два таких префікси, довжиною 2 і 3 байти. Для 2-байтового VEX префікса перший байт дорівнює 0xC5, для три байтового 0xC4. У 64-бітовому режимі перший байт VEX префікса унікальний. У 32-бітовому режимі виникає конфлікт з інструкціями LES і LDS, який дозволяється старшим бітом другого байта, він має значення тільки в 64-бітовому режимі, через непідтримувані форми інструкцій LES і LDS. Довжина наявних AVX інструкцій, разом з VEX префіксом, не перевищує 11 байт.

2. Історія розвитку x86-AVX

Перше розширення x86-AVX під назвою AVX було подано в 2011 році з мікроархітектурою Sandy Bridge. AVX розширює пакетні можливості x86-SSE з одинарною та подвійною точністю з плаваючою крапкою з 128 біт до 256 біт. Він також підтримує новий синтаксис інструкції з трьома операндами з використанням неруйнівних операндів джерела, що значно спрощує програмування мовою асемблера. Програмісти можуть використовувати цей новий синтаксис команд із запакованим 128-бітовим цілим числом, запакованим 128-бітовим числом з плаваючою крапкою та запаковані 256-розрядні операнди з плаваючою крапкою. Новий синтаксис інструкцій також можна використовувати для виконання скалярної арифметики з плаваючою крапкою одинарної та подвійної точності. У 2012 році Intel подала оновлену версію мікроархітектури Sandy Bridge під назвою Ivy Bridge, яка додала інструкції, що виконують перетворення з плаваючою крапкою половинної точності.

У 2013 році Intel запустила нову мікроархітектуру під назвою Haswell. Процесори, засновані на цій мікроархітектурі, включають AVX2, який розширює можливості запакованого цілого числа AVX з 128 біт до 256 біт. Він також включає розширені команди перетворення даних, змішування та переставлення, а також подає новий режим адресації векторних індексів, який полегшує завантаження в пам’ять (або збирання) елементів даних із несуміжних місць. Усі процесори на основі Haswell включають кілька технологій, пов’язаних з AVX2, включаючи FMA, розширену маніпуляцію бітами та команди циклічного зсуву без прапорів та команди зсуву. У липні 2013 року Intel анонсувала AVX-512, який розширював можливості SIMD AVX і AVX2 з 256 біт до 512 біт у майбутніх процесорах. У табл. 1 показано поточні та плановані технології x86-AVX. У цій таблиці використовуються акроніми SPFP і DPFP для позначення одинарної точності з плаваючою крапкою та подвійної точності з плаваючою крапкою, відповідно.

Таблиця 1 – Технології AVX

Реалізація	Підтримуваний тип	Особливості та вдосконалення
AVX	Packed 128-bit integer Packed 128-bit SPFP Packed 128-bit DPFP Packed 256-bit SPFP Packed 256-bit DPFP Scalar SPFP, DPFP	Операції SIMD з використанням підтримуваних типів даних і синтаксис інструкції з трьох операндів Умовно запаковані дані завантажуються та зберігаються Запаковане перетворення та переставлення з плаваючою крапкою Нові предикати порівняння з плаваючою крапкою Перетворення чисел з плаваючою крапкою половинної точності
AVX2	Packed 256-bit integer	Операції SIMD з використанням запакованих 256-розрядних цілих чисел Команди зі збору даних Покращені команди перетворення і переставлення Команди FMA Покращені інструкції щодо роботи з бітами Команди циклічного зсуву без перемикання прапора та звичайного зсуву
AVX-512	Packed 512-bit integer Packed 512-bit SPFP Packed 512-bit DPFP	Операції SIMD з використанням запакованих 512-розрядних операндів Умовні запаковані команди з елементами даних Перевизначення округлення на рівні команд Команди розкидування даних

Мікроархітектура Sandy Bridge використовувалася в другому поколінні Intel Core (серії i3, i5 та i7). Це процесори Intel Core третього та четвертого поколінь на основі мікроархітектур Ivy Bridge і Haswell відповідно. Сервер і родина процесорів Xeon E3, E3 v2 і E3 v3, орієнтовані на робочі станції, які також базуються на мікроархітектури Sandy Bridge, Ivy Bridge і Haswell відповідно.

Інформаційний веб-сайт Intel, містить додаткову інформацію щодо родин процесорів і відповідних їм мікроархітектур.

3. Середовище виконання x86-AVX

У Linux для вияснення розширень, які підтримує даний Intel процесор потрібно виконати команди:

```
$ grep "model name" /proc/cpuinfo | uniq
model name : Intel(R) Xeon(R) Gold 6230N CPU @ 2.30GHz
$ egrep "^flags" /proc/cpuinfo | uniq | egrep -o "avx512\S*"
avx512f
avx512dq
avx512cd
avx512bw
avx512vl
avx512_vnni
```

3.1. Набір регістрів x86-AVX

Середовище виконання x86-AVX включає набір регістрів і підтримувані типи даних.

X86-AVX додає до платформи x86 вісім нових 256-бітових регістрів під назвою YMM0-YMM7. Ці регістри з прямою адресацією можна використовувати для маніпулювання різноманітними типами даних, включаючи запаковані цілі числа, запаковані значення з плаваючою крапкою та скалярні значення з плаваючою крапкою. Молодші 128 бітів кожного регістра YMM пов’язані з відповідним регістром XMM, як показано на рис. 1. Інструкції X86-AVX можуть використовувати регістри XMM або YMM як операнди. Якщо інструкція x86-AVX використовує регістр XMM як операнд призначення, процесор обнуляє старші 128 бітів відповідного регістру YMM під час виконання. У процесорах, які підтримують x86-AVX, старші 128 бітів регістра YMM ніколи не змінюються під час виконання інструкції x86-SSE. Обробка за замовчуванням старших 128 бітів регістра YMM під час виконання інструкції обговорюється далі в цьому розділі.

YMM0	XMM0
YMM1	XMM1
YMM2	XMM2
YMM3	XMM3
YMM4	XMM4
YMM5	XMM5
YMM6	XMM6
YMM7	XMM7

255 128 127 0

Рисунок 1 – Набір x86-AVX регістрів в режимі x86-32

3.2. Типи даних x86-AVX

AVX підтримує операції SIMD, використовуючи 256- та 128-бітові запаковані операнди з плаваючою крапкою одинарної або подвійної точності. 256-бітовий регістр YMM або область пам’яті може містити вісім значень одинарної точності або чотири значення подвійної точності, як показано на рис. 2. При використанні з 128-бітовим регістром XMM або місцем в пам’яті команда AVX може обробляти чотири значення одинарної точності або два значення подвійної точності. Як SSE і SSE2, AVX маніпулює молодшим подвійним словом або чотирним словом регістра XMM під час виконання скалярної арифметики одинарної або подвійної точності з плаваючою крапкою відповідно.

AVX також дозволяє використовувати регістри XMM для виконання операцій SIMD з використанням різноманітних запакованих цілих операндів, включаючи байти, слова, подвійні слова та чотирні слова. AVX2 розширює можливості оброблення запакованих цілих чисел AVX на YMM регістрах та 256-бітових комірках пам'яті. На рис. 2 показано ці типи даних.

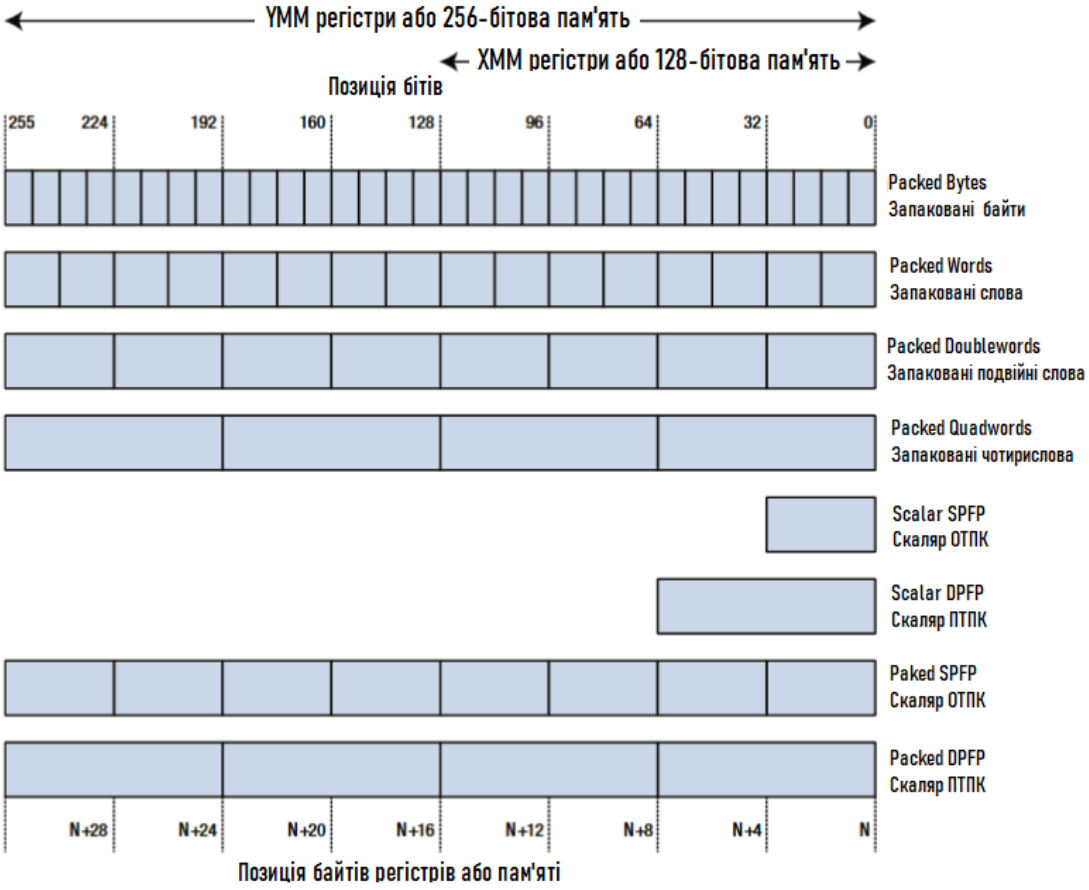


Рисунок 2 – Типи даних x86-AVX

3.3. Синтаксис команд x86-AVX

Можливо, найбільш примітним аспектом x86-AVX є використання нового синтаксису команд. Більшість команд x86-AVX використовують формат з трьох операндів, який складається

з двох операндів джерела та одного операнда призначення. Загальний синтаксис, який використовується для цих команд:

```
InstrMnemonic DesOp, SrcOp1, SrcOp2
```

де `InstrMnemonic` означає мнемоніку команд x86-AVX, а `DesOp, SrcOp1` і `SrcOp2` позначають операнди призначення та джерел відповідно. Решта x86-AVX команд використовують один або три операнди джерела. Майже всі команди x86-AVX операнди джерела є неруйнівними (тобто операнд не змінюється під час виконання команди), за винятком випадків, коли регістр операнда призначення збігається з одним із регістрів операндів джерела.

Табл. 2 містить кілька прикладів, які ілюструють загальний синтаксис команд x86-AVX. Всі мнемоніки команд починаються з літери `v`. Багато команд x86-AVX є простими розширеннями відповідної інструкції x86-SSE. Це розширення стає очевидним, якщо вилучити префікс `v` з мнемоніки команди, показаної в табл. 2.

Таблиця 2 – Синтаксис команд x86-AVX

Команда	Дія
`vaddpd ymm0,ymm1,ymm2` Додання запакованих чисел подвійної точності з плаваючою крапкою	ymm0[63:0] = ymm1[63:0] + ymm2[63:0] ymm0[127:64] = ymm1[127:64] + ymm2[127:64] ymm0[191:128] = ymm1[191:128] + ymm2[191:128] ymm0[255:192] = ymm1[255:192] + ymm2[255:192]
`vmulps xmm0,xmm1,xmm2` Множення запакованих чисел одинарної точності з плаваючою крапкою	xmm0[31:0] = xmm1[31:0] * xmm2[31:0] xmm0[63:31] = xmm1[63:31] * xmm2[63:31] xmm0[95:64] = xmm1[95:64] * xmm2[95:64] xmm0[127:96] = xmm1[127:96] * xmm2[127:96] ymm0[255:128] = 0
`vunpcklps xmm0,xmm1,xmm2` Розпакування запакованих значень одинарної точності з плаваючою крапкою	xmm0[31:0] = xmm1[31:0] xmm0[63:31] = xmm2[31:0] mm0[95:64] = xmm1[63:32] xmm0[127:96] = xmm2[63:32] ymm0[255:128] = 0
`vpxor ymm0,ymm1,ymm2` Логічне виключальне АБО	ymm0[255:0] = ymm1[255:0] ^ ymm2[255:0]
`vmovdqa ymm0,ymm1` Переслати вирівняні подвійні чотирні слова	ymm0[255:0] = ymm1[255:0]
`vblendpd ymm0,ymm1,ymm2,06h` Запакована суміш подвійної точності значень з плаваючою крапкою	ymm0[63:0] = ymm1[63:0] ymm0[127:64] = ymm2[127:64] ymm0[191:128] = ymm2[191:128] ymm0[255:192] = ymm1[255:192]

Здатність x86-AVX підтримувати синтаксис інструкцій із трьома операндами з'явилася завдяки новому префіксу кодування інструкцій. Префікс векторного розширення (VEX) дозволяє кодувати інструкції x86-AVX за допомогою більш ефективного формату, ніж префікси, що використовуються для інструкцій x86-SSE. Він також забезпечує шлях міграції для майбутніх удосконалень інструкцій x86-AVX. Більшість нових команд з використанням регістра загального призначення також використовують префікс VEX.

3.4. Особливості розширень x86-AVX

Разом із впровадженням AVX і AVX2 з'явилося кілька нових особливостей розширень для платформи x86. Особливості розширень включають перетворення чисел з плаваючою крапкою половинної точності, FMA обчислення та вдосконалені команди регістрів загального призначення.

Процесори на основі мікроархітектур Ivy Bridge і Haswell містять команди, які перетворюють числа з плаваючою крапкою з половинною точністю. У порівнянні зі стандартним значенням з плаваючою крапкою одинарної точності, значення з плаваючою крапкою з половинною точності – це число яка має експоненту (5 біт), значуще (11 біт) і знаковий біт. Кожне значення половинної точності з плаваючою крапкою має ширину 16 біт, початкова цифра значущого мається на увазі. Сумісні процесори містять інструкції, які можуть перетворювати запаковані значення з плаваючою крапкою половинної точності в запаковані значення з плаваючою крапкою одинарної точності та навпаки. Однак неможливо виконати звичайні арифметичні обчислення, такі як додавання, віднімання, множення та ділення, використовуючи значення з плаваючою крапкою половинної точності. Значення половинної точності з плаваючою крапкою насамперед призначені для зменшення потреб у просторі пам'яті або на пристрої зберігання даних. Недоліками використання значень із плаваючою крапкою половинної точності є низька точність і обмежений діапазон.

Процесори на основі Haswell також містять команди, які виконують операції FMA. Команда FMA (Fused Multiply Add) поєднує в собі команди множення та додавання (або віднімання). Точніше, обчислення "злитого" множення-додавання (або злитого множення-віднімання) виконує множення з плаваючою крапкою, а потім додавання (або віднімання) з плаваючою крапкою з використанням однієї операції округлення. Як приклад розглянемо вираз a = (b * c) + d. Використовуючи стандартну арифметику з плаваючою крапкою, процесор спочатку виконує множення що включає операцію округлення. Далі йде додавання з плаваючою крапкою та інша операція округлення. Якщо вираз обчислюється за допомогою арифметики FMA, процесор не округляє проміжний добуток b * c. Округлення проводиться лише один раз при кінцевому додаванні (b * c) + d. Команди FMA можна використовувати для покращення продуктивності і точності обчислень множення та їх накопичення, таких як скалярний добуток і множення матриць. Багато алгоритмів обробки сигналів також широко використовують FMA операції. Набір команд FMA підтримує операції з використанням як скалярних, так і запакованих операцій для значень з плаваючою крапкою одинарної та подвійної точності.

Останні особливості розширень додають нові команди для регістрів загального призначення. Ці команди (доступні для процесорів Haswell) підтримують покращені маніпуляції бітами, команди без прапорного циклічного зсуву, звичайного зсуву, а також без прапорне беззнакове цілочислове множення. Команди без прапорного циклічного зсуву, звичайного зсуву, а також без прапорного беззнакового цілочислового множення не впливають на будь-який із прапорів стану в регістрі EFLAGS. Це може покращити продуктивність цілочислових обчислень і алгоритмів. Більшість нових команд для регістрів загального призначення також використовують новий синтаксис мови асемблера з трьома операндами.

Розглянуті особливості розширень, вважаються окремими функціями процесора. З точки зору програмування це означає, що розробник програмного забезпечення не може вважати, що відповідні набори команд доступні залежно від того, чи підтримує процесор AVX або AVX2. Наприклад, процесор, орієнтований на мобільні пристрої може включати підтримку AVX2, але

не FMA. Наявність певної особливості розширення завжди слід явно перевіряти за допомогою коианди `cpuid`.

4. Огляд команд x86-AVX

Набір команд x86-AVX можна умовно розділити на три групи. Перша група включає інструкції x86-SSE, які були розроблені для використання нового синтаксису з трьох операндів та використання 128- або 256-бітових операндів. Наступна група складається з нових команд, які були додані в AVX або AVX2. Остання група включає в себе інструкції з розширеними функціями x86-AVX, включаючи перетворення з плаваючою крапкою половинної точності, FMA та нові інструкції для регістрів загального призначення.

Перше ніж перейти до огляду, є деякі спільні особливості синтаксису та виконання щодо набору команд x86-AVX, які вимагають кількох коментарів. Як згадувалося раніше, усі команди x86-AVX використовують синтаксис, який складається з мнемоніки команди, операнда призначення та до трьох операндів джерел. Якщо команда виконує операцію передачі даних, операнд призначення може вказати розташування в пам'яті, інакше це має бути регістр XMM або YMM. Тільки один з операндів джерела може вказувати розташування в пам'яті, решта операндів джерела має бути регістром XMM, регістром YMM або безпосереднім операндом.

X86-AVX послаблює вимоги до вирівнювання операндів команд у пам’яті. За винятком команд передачі даних, які явно посилаються на 16- або 32-байтовий вирівняний операнд у пам’яті, правильне вирівнювання операндів команди x86-AVX у пам’яті не вимагається. Незважаючи на це послаблення, настійно рекомендується правильно вирівняти всі 16- та 32-байтові операнди в пам’яті для найбільшої продуктивності. Команди x86-SSE, які виконуються на процесорах, які підтримують x86-AVX, повинні використовувати належним чином вирівняні операнди пам’яті.

4.1. Розширені команди x86-SSE

Більшість команд x86-SSE, які маніпулюють 128-бітовими операндами, мають відповідну команду x86-AVX. Це включає запаковані значення з плаваючою крапкою одинарної точності, подвійної точності та цілі числа. Наприклад, команда x86-SSE множить `mulps xmm0,xmm1`, множить запаковані значення з плаваючою крапкою одинарної точності в регістрах XMM0 і XMM1 і зберігає результат запакованого продукту в регістр XMM0. Паралельною командою x86-AVX є `vmulps xmm0,xmm0,xmm1.` Іншим прикладом є команда x86-SSE додавання запакованих цілих `paddb xmm0,xmm1` і відповідна команда x86-AVX `vpaddb xmm0,xmm0,xmm1`. Зверніть увагу, що в обох цих прикладах вміст регістра XMM0 що використовується, змінюється. Приклади неруйнівних команд x86-AVX включають `vmulps xmm0,xmm1,xmm2` і `vpaddb xmm0,xmm1,xmm2`, виконання яких не змінює значення в XMM1 і XMM2.

Майже всі 128-бітові команди x86-SSE мають форму x86-AVX, яку можна використовувати з 256-бітовими операндами. Наприклад, команда `vsubpd ymm7,ymm0,ymm1` виконує пакетне віднімання з плаваючою крапкою, використовуючи чотири пари значень подвійної точності. Команда `vdivps ymm7,ymm0,ymm1` виконує запаковане ділення одинарної точності з плаваючою крапкою, використовуючи вісім пар значень, а `vpsubb ymm7,ymm0,ymm1` віднімає 32 пари значень цілих байтів.

Усередині процесора кожен 256-бітовий регістр AVX розділений на верхню та нижню 128-бітову половини. Більшість команд x86-AVX виконують свої операції за допомогою однієї половини елемента операнда джерела та призначення. Це незалежне виконання половин, як правило, непомітне при використанні команд x86-AVX, які виконують арифметичні обчислення.

Однак при використанні команд, які змінюють порядок елементів запакованих даних (таких як `vshufps` і `vpunpcklwd`), ефект виконання окремих половин більш очевидний, як показано на рис. 3. У цих прикладах операції перетасування з плаваючою крапкою та розпакування слів виконуються незалежно як у старшій (біти 255:128), так і в молодшій половині (біти 127:0) подвійних чотирних слів.

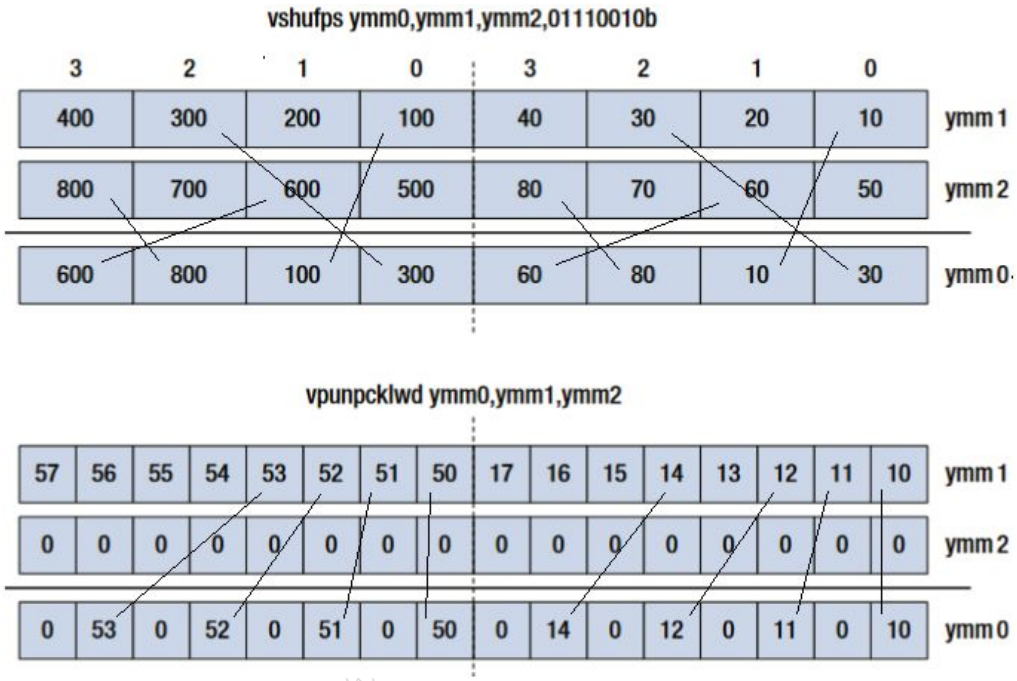


Рисунок 3 – Приклад виконання x86-64 команд з використання старшої та молодшої половин регістрів YMM

Набір команд x86-AVX також підтримує 3-операндні форми скалярних команд з плаваючою крапкою x86-SSE. Команда `vaddss xmm0,xmm1,xmm2,` наприклад, додає скалярні значення з плаваючою крапкою одинарної точності в XMM1 і XMM2 і зберігає суму в XMM0. Команда `vmulsd mm0,xmm1,xmm2` множить скалярне значення подвійної точності в XMM1 і XMM2 і отриманий добуток потім зберігається в регістрі XMM0. Повний перелік усіх команд x86-SSE і x86-AVX міститься в довідкових посібниках Intel і AMD.

Високий ступінь симетрії команд між x86-SSE і x86-AVX та псевдонімами наборів регістрів XMM і YMM вводить кілька проблем програмування, про які розробники програмного забезпечення повинні пам’ятати. Перша проблема стосується обробки процесором старших 128 бітів регістра YMM, коли відповідний регістр XMM використовується як операнд призначення. Під час виконання на процесорі, який підтримує технологію x86-AVX, інструкція x86-SSE, яка використовує регістр XMM як операнд призначення, ніколи не матиме доступу до старших 128 бітів відповідного регістру YMM. Проте еквівалент команди x86-AVX обнулить старші 128 бітів відповідного регістра YMM. Розглянемо, наприклад, такі екземпляри команд `(v)cvtps2pd`

(перетворення запакованих значень одинарної точності в запаковані значення з плаваючою крапкою подвійної точності):

```
cvtps2pd xmm0,xmm1
vcvtps2pd xmm0,xmm1
vcvtps2pd ymm0,ymm1
```

Інструкція x86-SSE `cvtps2pd` перетворює два запакованих значення з плаваючою крапкою одинарної точності в молодшому чотирному слові XMM1 на значення з плаваючою крапкою подвійної точності та зберігає результат у регістрі XMM0. Старші 128 бітів регістра YMM0 не змінюються. Перша інструкція `vcvtps2pd` виконує таку саму операцію перетворення запакованої одинарної точності в запаковану подвійної точності, а також обнуляє старші 128 бітів YMM0. Друга інструкція `vcvtps2pd` перетворює чотири запаковані значення з плаваючою крапкою одинарної точності в молодших 128 бітах YMM1 на запаковані значення з плаваючою крапкою подвійної точності та зберігає результат у YMM0.

Усі скалярні команди з плаваючою крапкою x86-AVX встановлюють старші 128 бітів регістра YMM в нуль. Ці команди також копіюють невикористані біти першого операнда джерела в операнд призначення, як показано в табл. 3, використовуючи x86-AVX `vaddss` і `vaddsd` (додати скалярні значення одинарної/подвійної точності з плаваючою крапкою). Табл. 3 також ілюструє роботу команд `vsqrtss` і `vsqrtsd` (обчислення скалярного квадратного кореня з одинарної/подвійної точності з плаваючою крапкою). Ці команди використовують два операнди джерела, навіть якщо вони виконують унарну операцію, використовуючи лише другий операнд джерело.

Таблиця 3 – X86-AVX скалярні команди з плаваючою крапкою

Команда	Дія
`vaddss xmm0,xmm1,xmm2` Додати скалярні значення одинарної точності з плаваючою крапкою	xmm0[31:0] = xmm1[31:0] + xmm2[31:0] xmm0[127:32] = xmm1[127:32] ymm0[255:128] = 0
`vaddsd xmm0,xmm1,xmm2` Додати скалярні значення подвійної точності з плаваючою крапкою	xmm0[63:0] = xmm1[63:0] + xmm2[63:0] xmm0[127:64] = xmm1[127:64] ymm0[255:128] = 0
`vsqrtss xmm0,xmm1,xmm2` Корінь квадратний із значення одинарної точності з плаваючою крапкою	xmm0[31:0] = sqrt(xmm2[31:0]) xmm0[127:32] = xmm1[127:32] ymm0[255:128] = 0
`vsqrtsd xmm0,xmm1,xmm2` Корінь квадратний із значення подвійної точності з плаваючою крапкою	xmm0[63:0] = sqrt(xmm2[63:0]) xmm0[127:64] = xmm1[127:64] ymm0[255:128] = 0

Остання проблема, про яку слід знати програмістам, стосується змішування інструкцій x86-AVX і x86-SSE. Програмам дозволено змішувати інструкції x86-AVX і x86-SSE, *але будь-яке змішування має бути зведене до мінімуму, щоб уникнути штрафних санкцій за зміну стану внутрішнього процесора*, які можуть вплинути на продуктивність. Ці штрафні санкції можуть

виникнути, якщо від процесора вимагається зберегти старші 128 біт кожного регістра YMM під час переходу від виконання команд x86-AVX до виконання команд x86-SSE. Штрафи за перехід станів можна повністю уникнути, використовуючи команду `vzeroupper` (нуль верхніх бітів регістрів YMM), яка обнуляє старші 128 бітів усіх регістрів YMM. Цю команду слід використовувати перед будь-яким переходом від 256-бітового коду x86-AVX (тобто будь-якої інструкції x86-AVX, яка використовує регістр YMM) до коду x86-SSE.

Одним із поширених випадків використання команди `vzeroupper` є публічна функція, яка використовує 256-розрядні команди x86-AVX. Ці типи функцій повинні включати команду `vzeroupper` перед виконанням будь-якої команди `ret`, оскільки це запобігає штрафи процесору за перехід станів у будь-якому коді мови високого рівня, який використовує команди x86-SSE. Команду `vzeroupper` слід також використовувати перед викликом будь-яких бібліотечних функцій, які можуть містити код x86-SSE. Функції також можуть використовувати команду `vzeroall` (обнулити всі регістри YMM), щоб уникнути штрафних санкцій за перехід стану x86-AVX /x86-SSE.

5. Нові команди

Нові команди x86-AVX поділені на наступні підгрупи:

- Широкомовна трансляція (Broadcast).
- Змішати (Blend).
- Переставити (Permute).
- Видобути та вставити (Extact and Insert).
- Переслати з маскою (Masked move).
- Змінний бітовий зсув (Variale bit sift).
- Збирати (Gather).

У зведених таблицях команд перераховано необхідну версію x86-AVX. Якщо вказано AVX і AVX2, це означає, що в AVX2 додано додаткові форми команд.

5.1. Широкомовна трансляція

Команди широкомовної трансляції передають або копіюють одне значення даних до кількох елементів запакованого операнда призначення. Широкомовні команди доступні для всіх типів запакованих даних, включаючи одинарну точність з плаваючою крапкою, подвійну точність з плаваючою крапкою та цілі числа. Табл. 4 підсумовує команди широкомовної трансляції.

Таблиця 4 – X86-AVX широкомовні команди

Команда	Описання	Версія
vbroadcastss	Копіює значення одинарної точності з плаваючою крапкою до всіх елементів операнда призначення	AVX AVX2
vbroadcastsd	Копіює значення подвійної точності з плаваючою крапкою до всіх елементів операнда призначення	AVX AVX2

vbroadcastf128	Копіює запаковане 128-бітове з плаваючою крапкою значення з пам'яті у молодшу і старшу половини подвійного чотирного слова операнда призначення	AVX
vbroadcasti128	Копіює запаковане 128-бітове цілочислове значення з пам'яті у молодшу і старшу половини подвійного чотирного слова операнда призначення	AVX2
vpbroadcastb **vpbroadcastw** **vpbroadcastd** **vpbroadcastq**	Копіює 8-, 16-, 32- або 64-бітове цілочислове значення до всіх елементів операнда призначення	AVX2

5.2. Команди змішування

Команди змішування умовно об’єднують елементи двох типів запакованих даних, табл. 5.

Таблиця 5 – Команди зміщування

Команда	Описання	Версія
vpblendd	Умовне копіювання значень подвійних слів з перших двох операндів джерела до операнда призначення з використанням маски керування, яка визначається безпосереднім значенням	AVX2

5.3. Команди переставлення

Команди переставлення змінюють порядок або копіюють елементи запакованого типу даних. Підтримуються кілька типів запакованих даних, включаючи подвійні слова, одинарну та подвійну точність з плаваючою крапкою, табл. 6.

Таблиця 6 – Команди переставлення

Команда	Описання	Версія
vpermd	Переставляє елементи подвійного слова другого операнда джерела, використовуючи індекси, визначені першим операндом джерелом. Ця команда може бути використана для зміни порядку або копіювання значень подвійних слів у другому операнді джерелі.	AVX2
vpermpd	Переставляє елементи DPFP першого операнда джерела за допомогою індексів, визначених безпосереднім операндом. Ця команда може бути використана для зміни порядку або копіювання значень DPFP у операнді джерелі.	AVX2
vpermps	Переставляє елементи SPFP другого операнда джерела, використовуючи індекси, визначені першим операндом джерелом. Цю команду	AVX2

	можна використати для зміни порядку або копіювання значень SPFP у другому операнді джерелі.	
vpermq	Переставляє елементи чотирних слів першого операнда джерела, використовуючи індекси, визначені безпосереднім операндом. Ця команда може бути використана для зміни порядку або копіювання значень чотирних слів у операнді джерелі.	AVX2
vperm2i128	Переставляє запаковані 128-розрядні цілі значення перших двох операндів джерел, використовуючи індекси, визначені безпосередньою маскою. Цю команда можна використати для зміни порядку, тиражування або чергування значень у перших двох операндах джерел.	AVX2
vpermilpd	Переставляє значення DPFP у першому операнді джерела, використовуючи керуюче значення, визначене другим операндом джерела. Кожна 128-бітова половина змінюється незалежно.	AVX
vpermilps	Переставляє значення SPFP у першому операнді джерелі за допомогою керуючого значення, заданого другим операндом джерела. Кожен 128-бітова половина змінюється незалежно.	AVX
vperm2f128	Переставляє запаковані 128-розрядні значення з плаваючою крапкою перших двох операндів джерел, використовуючи індекси, визначені безпосередньою маскою. Цю команду можна використовувати для зміни порядку, тиражування або чергування значень у перших двох операндах джерелах.	AVX

5.4. Видобування і вставка

Група видобування та вставки містить команди, які копіюють 128-розрядні запаковані цілі значення між регістром YMM і регістром XMM або місцем у пам’яті. Табл. 7 підсумовує команди видобування і вставок.

Таблиця 7 – Команди видобування і вставок

Команда	Описання	Версія
vextracti128	Видобуває запаковані молодші або старші 128-біти цілочислового значення з операнда джерела і копіює його в операнд призначення. Значення, яке потрібно видобути, визначається як безпосередній операнд.	AVX2
vinserti128	Вставляє 128-бітове запаковане ціле число з другого операнд джерела в операнд призначення. Розташування в операнді призначення (молодші або старші 128-біт)	AVX2

	визначається безпосереднім операндом. Залишок елемента операнда призначення заповнюється за допомогою відповідного елемента першого операнда джерела.	

5.5. Масковане переміщення

Команди маскованих переміщень виконують за умовою переміщення елементів у значення запакованих даних. Контрольна маска визначає, чи конкретний елемент копіюється з операнда джерела в операнд призначення. Якщо елемент не скопійовано, нуль зберігається у відповідному елементі операнда призначення. Масковані команди показано в табл. 8.

Таблиця 8 – Команди маскованого переміщення

Команда	Описання	Версія
vmaskmovps	Умовно копіює елементи одинарної точності з плаваючою крапкою другого операнда джерела до відповідних елементів операнда призначення відповідно до керуючої маски, яка вказана в першому операнді джерела.	AVX
vmaskmovpd	Умовно копіює елементи подвійної точності з плаваючою крапкою другого операнда джерела до відповідних елементів операнда призначення відповідно до керуючої маски, яка вказана в першому операнді джерела.	AVX
vpmaskmovd	Умовно копіює елементи подвійних слів другого операнда джерела до відповідних елементів операнда призначення відповідно до керуючої маски, яка вказана в першому операнді джерела.	AVX2
vpmaskmovq	Умовно копіює елементи чотирислів другого операнда джерела до відповідних елементів операнда призначення відповідно до керуючої маски, яка вказана в першому операнді джерела.	AVX2

5.6. Змінний бітовий зсув

Команди зі змінним бітовим зсувом виконують арифметичні або логічні зсуви для елементів запакованого значення даних подвійного або чотирного слова з використанням різних бітів лічильників. Ці інструкції підсумовано в табл. 9.

Таблиця 9 – Команди змінного бітового зсуву

Команда	Описання	Версія
vpsllvd **vpsllvq**	Зсовує кожний подвійний/чотири словний елемент даних першого операнда джерела ліворуч із заповненням зсувів нулями. Лічильник бітового зсуву визначається відповідним елементом даних другого операнда джерела.	AVX2

vpsravd	Зміщує кожен елемент даних подвійного слова першого операнда джерела вправо, одночасно зберігаючи біт знаку елемента. Лічильник бітового зсуву визначається відповідним елементом даних другого операнда джерела.	AVX2
vpsrlvd **vpsrlvq**	Зміщує кожний елемент даних подвійного /чотирного слова першого операнда джерела праворуч із заповненням зсувів нулями. Лічильник бітового зсуву визначається відповідним елементом даних другого операнда джерела	AVX2

5.7. Команди збирання

Команди збирання за умови копіюють елементи даних із масиву на основі пам’яті в регістр XMM або YMM. У цих командах використовується спеціальний режим адресації пам’яті, який називається VSIB (vector scale-index-base). VSIB пам'ять адресація використовує такі компоненти для визначення операнда:

• база – регістр загального призначення, який вказує на початок масиву в пам'яті;

• масштаб – коефіцієнт масштабування розміру елемента масиву (1, 2, 4 або 8).

• індекс – векторний регістр (XMM або YMM), який містить знак індексу масиву подвійних слів або чотирних слів зі знаком.

• зміщення – додаткове фіксоване зміщення від початку масиву.

Залежно від команди векторний регістр повинен містити два, чотири або вісім цілочислових індексів зі знаком. Індекси використовуються для вибору елементів з масиву. На рис. 4 показано виконання команди `vgatherdps xmm0,[esi+xmm1*4],xmm2`.

Рисунок 4 – Команда `vgatherps`

У цьому наприклад, регістр ESI вказує на початок масиву, що містить одинарну точність значення з плаваючою крапкою. Регістр XMM1 містить чотири знакові індекси масиву подвійних слів і регістр XMM2 містить маску умови керування копіюванням.

Операнд-приймач і другий операнд-джерело (маска керування копіюванням) команди збирання мають бути регістром XMM або YMM. Перший операнд-джерело визначає компоненти VSIB (тобто базовий регістр масиву, масштабний коефіцієнт, індекси масиву та необов'язкове зміщення). Зауважимо, що команди збирання не перевіряють недійсний індекс масиву, тому використання недійсного індексу масиву дасть некоректний результат. Табл. 10 підсумовує команди збирання. У цій таблиці кожна команда використовує префікс `vgatherd` або `vgatherq` для визначення індексів масиву подвійних слів або чотирних слів відповідно.

Таблиця 10 – x86-AVX команди збирання

Команда	Описання	Версія
vgatherdpd **vgatherqpd**	Копіює за умовою два або чотири значення подвійної точності з плаваючою крапкою з масиву в пам'яті за допомогою VSIB адресації.	AVX2
vgatherdps **vgatherqps**	Копіює за умовою два або чотири значення одинарної точності з плаваючою крапкою з масиву в пам'яті за допомогою VSIB адресації.	AVX2
vgatherdd **vgatherqd**	Копіює за умовою чотири або вісім значень подвійних слів з масиву в пам'яті за допомогою VSIB адресації.	AVX2
vgatherdq **vgatherqq**	Копіює за умовою два або чотири значень чотирислів з масиву в пам'яті за допомогою VSIB адресації.	AVX2

6. Команди з розширеними функціями

Команди з розширеними функціями x86-AVX, включають перетворення з плаваючою крапкою половинної точності, FMA, розширення для регістрів загального призначення. Щоб використовувати будь-яку з команд у цих групах, вони повинні підтримуватися процесором, як зазначено відповідним прапором команди `cpuid`. Групи команд половинної точності та FMA доступні на процесорі, який підтримує AVX або AVX2 відповідно, разом із операційною системою, яка зберігає стан регістра YMM під час перемикання контексту потоку та процесу.

6.1. Команди половинної точності з плаваючою крапкою

Команди половинної точності з плаваючою крапкою містять команди, які виконують пакетні перетворення половинної точності з плаваючою крапкою в одинарну точність з плаваючою крапкою і навпаки. Підтримка процесора для цих команд вказується за допомогою ознаки прапора `cpuid F16C` . Таблиця 11 містить короткий опис команд перетворення чисел з плаваючою крапкою половинної точності.

Таблиця 11 – x86-AVX команди половинної точності з плаваючою крапкою

Команда	Описання	Версія
vcvtph2ps	Перетворює чотири або вісім значень із плаваючою крапкою половинної точності операнда джерела до значень з плаваючою крапкою одинарної точності і зберігає результати в операнді призначення. Кількість виконаних перетворень залежить від розміру операнда призначення, який має бути XMM або YMM регістром.	AVX
vcvtps2ph	Перетворює чотири або вісім значень із плаваючою крапкою одинарної точності у операнді джерела до значень з плаваючою крапкою половинної точності і зберігає результати в операнді призначення. Кількість виконаних перетворень залежить від розміру першого операнда джерела, який має бути регістром XMM або YMM. Ця інструкція також вимагає безпосереднього операнда, який задає режим округлення.	AVX

6.2. Команди злитого множення-додавання (FMA)

Команди FMA (злите множення-додавання) виконують операції злитого множення-додавання або злитого множення-віднімання, використовуючи запаковані операнди з плаваючою крапкою або скалярні операнди з плаваючою крапкою. Команди FMA виконують свої обчислення за допомогою одного з наступних загальних виразів:

```
a = (b * c) + d
a = (b * c) - d
a = -(b * c) + d
a = -(b * c) - d
```

У кожному з цих виразів процесор застосовує лише одну операцію округлення для обчислення кінцевого результату, що може підвищити швидкість і точність обчислення.

Усі мнемоніки команд FMA використовують тризначну схему впорядкування операндів, яка визначає операнди джерел які використовуються для множення та додавання (або віднімання). Перша цифра вказує операнд джерело, як множене, друга цифра вказує операнд джерело як множник; а третя цифра вказує операнд джерело, який додається до добутку (або віднімається від нього). Наприклад, команда `vfmadd132sd xmm0,xmm1,xmm2` виконує злите множення-додавання скалярних значень з плаваючою крапкою подвійної точності). У команді регістри XMM0, XMM1 і XMM2 є операндами джерел 1, 2 і 3 відповідно. Команда `vfmadd132sd` обчислює `(xmm0[63:0] * xmm2[63:0]) + xmm1[63:0]`, округлює суму добутку відповідно до режиму округлення, визначеного MXCSR.RC, і зберігає кінцевий результат у `xmm0 [63:0]`.

Набір команд FMA підтримує операції з використанням запакованих і скалярних значень з плаваючою крапкою одинарної та подвійної точності. Запаковані операції FMA можна виконувати за допомогою регістрів XMM або YMM. Регістри XMM (YMM). підтримують запаковані обчислення FMA з використанням двох (чотирьох) значень з плаваючою крапкою з

подвійною точністю або чотирьох (восьми) значень одинарної точності. Скалярні обчислення FMA необхідно виконувати з використанням встановленого регістра XMM. Для всіх команд FMA перший і другий операнди джерела повинні бути регістром. Третій операнд джерела може бути регістром або місцем пам'яті. Якщо інструкція FMA використовує регістр XMM як операнд призначення, старші 128 бітів відповідного регістра YMM встановлюються на нуль. Інструкції FMA виконують єдину операцію округлення за допомогою режиму, визначеного в MXCSR.RC.

Для полегшення розуміння набір команд FMA розділено на шість підгруп У таблицях опису підгруп команда використовує наступні дволітерні суфікси мнемоніки: `pd` (запаковане число з плаваючою крапкою подвійної точності), `ps` (запаковане число з плаваючою крапкою одинарної точності), `sd` (скалярне число з плаваючою крапкою подвійної точності) і `ss` (скалярне число одинарної точності з плаваючою крапкою). Символи `src1`, `src2` і `src3` позначають три операнди джерела, а `des` позначає операнд призначення, який є таким самим, як і `src1`.

6.2.1. Підгрупа VFMADD

Підгрупа VFMADD містить команди, які виконують операції злитого множення-додавання, використовуючи або запаковані типи даних з плаваючою крапкою, або скалярні типи даних з плаваючою крапкою. Ці команди показані в табл. 12.

Таблиця 12 – Підгрупа команд FMA VFMADD

Мнемоніка	Операція
vfmadd132(pd|ps|sd|ss)	des = src1 * src3 + src2
vfmadd213(pd|ps|sd|ss)	des = src2 * src1 + src3
vfmadd231(pd|ps|sd|ss)	des = src2 * src3 + src1

6.2.2. Підгрупа VFMSUB

Підгрупа VFMSUB містить команди, які виконують операції злитого множення-віднімання з використанням запакованих чи скалярних типів даних з плаваючою крапкою. У таблиці 13 показано ці команди та їхні операції.

Таблиця 13 – Підгрупа команд FMA VFMSUB

Мнемоніка	Операція
vfmsub132(pd|ps|sd|ss)	des = src1 * src3 - src2
vfmsub213(pd|ps|sd|ss)	des = src2 * src1 - src3
vfmsub231(pd|ps|sd|ss)	des = src2 * src3 - src1

6.2.3. Підгрупа VFMADDSUB

Підгрупа VFMADDSUB включає команди, які виконують операції злитого множення над запакованими типами даних, використовуючи додавання для непарних елементів і віднімання для парних елементів. Таблиця 14 описує ці команди.

Таблиця 14 – Підгрупа команд FMA VFMADDSUB

Мнемоніка	Операція
vfmaddsub132(pd|ps)	des = src1 * src3 + src2 (непарний) des = src1 * src3 - src2 (парний)
vfmaddsub213(pd|ps)	des = src2 * src1 + src3 (непарний) des = src2 * src1 - src3 (парний)
vfmaddsub231(pd|ps)	des = src2 * src3 + src1 (непарний) des = src2 * src3 - src1 (парний)

6.2.4. Підгрупа VFMSUBADD

Підгрупа VFMSUBADD містить команди, які виконують операції злитого множення над типами запакованих даних, використовуючи віднімання для непарних елементів і додавання для парних елементів. Таблиця 15 підсумовує ці команди.

Таблиця 15 – Підгрупа команд FMA VFMSUBADD

Мнемоніка	Операція
vfmsubadd132(pd|ps)	des = src1 * src3 - src2 (непарний) des = src1 * src3 + src2 (парний)
vfmsubadd213(pd|ps)	des = src2 * src1 - src3 (непарний) des = src2 * src1 + src3 (парний)
vfmsubadd231(pd|ps)	des = src2 * src3 - src1 (непарний) des = src2 * src3 + src1 (парний)

6.2.5. Підгрупа VFNMADD

Підгрупа VFNMADD містить команди, які виконують злиті негативні операції множення-додавання. У таблиці 16 показано ці команди.

Таблиця 16 – Підгрупа команд FMA VFNMSUBADD

Мнемоніка	Операція
vfnmadd132(pd|ps|sd|ss)	des = -(src1 * src3) + src2
vfnmadd213(pd|ps|sd|ss))	des = -(src2 * src1) + src3
vfnmadd231(pd|ps|sd|ss))	des = -(src2 * src3) + src1

6.2.6. Підгрупа VFNMSUB

Підгрупа VFNMSUB містить команди, які виконують злиті негативні операції множення-віднімання. Таблиця 7 описує роботу цих команд.

Таблиця 17 – Підгрупа команд FMA VFNMSUB

Мнемоніка	Операція
vfnmsub132(pd|ps|sd|ss)	des = -(src1 * src3) - src2

vfnmsub213(pd|ps|sd|ss))	des = -(src2 * src1) - src3
vfnmsub231(pd|ps|sd|ss))	des = -(src2 * src3) - src1

6.2.7. Регістри загального призначення

Група регістрів загального призначення включає нові команди, які підтримують розширені маніпуляції з бітами, операції циклічного зсуву без прапорів і звичайного зсуву, беззнакове множення цілих чисел без прапорів. Ці команди подано в таблиці 18. Підтримка процесора для цих команд вказується за допомогою позначок функції CPUID.

Таблиця 18 – Команди регістрів загального призначення

Мнемоніка	Операція	Прапор ознак
`andn`	Виконує порозрядне логічне І інвертованого першого операнда джерела з другим операндом-джерелом і зберігає результат до операнда призначення. Перший операнд джерела і операнд призначення повинен бути регістром загального призначення. Другим операндом-джерелом може бути розташування пам'яті або а регістр загального призначення.	`BMI1`
`bextr`	Видобуває бітове поле з першого операнда джерела за допомогою індексу і довжина, що вказується другим операндом джерелом. Результат записується в операнд призначення. Другий операнд джерело та операнд призначення повинні бути регістрами загального призначення. Перший операнд джерело може бути місцем пам'яті або регістром загального призначення.	`BMI1`
`blsi`	Видобуває молодший "1" біт із операнда джерела та встановлює відповідний біт в операнді призначення. Всі інші біти операнда призначення встановлені на нуль. Операнд джерело має бути місцем пам'яті або регістром загального призначення. Операнд призначення повинен бути регістром загального призначення.	`BMI1`
`blsmsk`	Визначає бітову позицію наймолодшого встановленого біта в операнді джерелі, встановлює цей біт і всі молодші біти в 1 у операнді призначення. Біти без маски в операнді призначення встановлюються в нуль. Операнд джерело може бути місцем в пам'яті або а регістром загального призначення. Операнд призначення має бути регістром загального призначення.	`BMI1`
`blsr`	Копіює операнд джерела в операнд призначення, скидає біт операнда призначення, що відповідає молодшому встановленому в 1 біту в операнді джерела. Операнд джерело може бути місцем пам'яті або регістром загального призначення. The операнд призначення повинен бути регістром загального призначення.	`BMI1`
`bzhi`	Копіює перший операнд джерела в операнд призначення; очищає старші біти операнда призначення за допомогою значення індексу, визначеного другим операндом джерела. Операнд призначення та другий операнд джерела мають бути регістрами загального призначення. Перший операнд джерела може бути розташованим у пам'яті або регістром загального призначення.	`BMI1`
`lzcnt`	Підраховує кількість початкових нульових бітів у операнді джерелі і зберігає це значення в операнді призначення. Якщо значення операнда джерела дорівнює нулю, то операнд	`LZCNT`

	призначення встановлюється в розмір операнда. Ця команда використовується як альтернатива команді bsr, яка залишає операнд призначення не визначеним, якщо значення операнда джерела дорівнює нуль.	
`mulx`	Виконує беззнакове множення між регістрами EDX і операндом джерела. Старша і молодша частини добутку зберігаються до першого та другого операндів призначення відповідно. Біти стану в EFLAGS не оновлюються. Операнд джерело може бути місцем пам'яті або регістром загального призначення. Операнди першого і другого призначення повинні бути регістрами загального призначення.	`BMI2`
`pdep`	Переносить і розсіює молодші біти операнда першого джерела до операнда призначення за допомогою бітової маски, яка визначається другим операндом джерела. Біти операнда призначення, не включені в бітову маску, встановлюються в нуль. Операнд призначення та перший операнд джерела мають бути регістрами загального призначення. Другий операнд джерела може бути місцем пам'яті або регістром загального призначення.	`BMI2`
`pext`	Переносить біти з першого операнда джерела в молодший бітові позиції операнда призначення, використовуючи бітову маску, яка визначається другим операндом джерела. Операнд призначення і перший операнд джерела повинні бути регістрами загального призначення. Другим операндом джерела може бути місце в пам'яті або регістр загального призначення.	`BMI2`
`rdrand`	Завантажує апаратно згенероване випадкове число в указаний операнд призначення, який має бути регістром загального призначення.	`RDRAND`
`roxr`	Прокручує біти операнда джерела за допомогою значення лічильника, що визначається безпосереднім операндом; результат зберігається в операнді призначення. Команда не міняє біти стану в EFLAGS. Операнд джерела може бути місцем пам'яті або регістром загального призначення. Операнд призначення повинен бути регістром загального призначення.	`BMI2`

`sarx`
`shlx`
`shrx`
	Зсунути (правий арифметичний, лівий логічний або правий логічний) перший операнд джерело, використовуючи вказане значення лічильника другим операндом джерела. Результат зберігається в операнд призначення. Біти стану в EFLAGS не оновлюються. Перший операнд джерела може бути місцем в пам'яті або регістром загального призначення. Другий операнд джерела і операнд призначення повинні бути регістрами загального призначення.	`BMI2`

`tzcnt`	Підраховує кількість кінцевих нульових бітів у операнді джерела і зберігає це значення в операнді призначення. Якщо значення операнда джерела дорівнює нулю, то операнд призначення встановлюється в розмір операнда. Ця інструкція використовується як альтернатива інструкції bsf, яка залишає операнд призначення не визначений, якщо значення операнда джерела дорівнює нулю.	`BMI1`

7. Команди AVX-512

Як випливає з назви, AVX-512 розширює регістри AVX2 з 256 біт до 512 біт. Щоб детально зрозуміти концепцію AVX-521, розглянемо приклад додавання 64-розрядного вектора

Набір команд	Сі формат команд	Запаковані 64-бітові цілі	Команди асемблера
SSE2	__m128i _mm_add_epi64 (__m128i a, __m128i b)	2 (128/64)	paddq
AVX2	__m256i _mm256_add_epi64 (__m256i a, __m256i b)	4 (256/64)	vpaddq
AVX-512	__m512i _mm512_add_epi64 (__m512i a, __m512i b)	8 (512/64)	vpaddq

Команда SSE2 `_mm_add_epi64 (paddq)` додає два 128-розрядних регістри *a* і *b* і повертає результат в єдиному 128-бітовому цілочисловому регістрі. Кожен регістр містить *два* запакованих 64-розрядних цілих числа, таким чином виконуються два окремі 64-розрядні додавання в одній команді.

Команда AVX2 `_mm256_add_epi64 (vpaddq)` додає два 256-розрядних регістри *a* і *b* і повертає результат в єдиному 256-бітовому цілочисловому регістрі. Кожен регістр містить *чотири* запакованих 64-розрядних цілих числа, таким чином виконуються чотири окремі 64-розрядні додавання в одній команді.

Команда AVX-512 `_mm512_add_epi64 (vpaddq)` додає два 512-розрядних регістри *a* і *b* і повертає результат в єдиному 512-бітовому цілочисловому регістрі. Кожен регістр містить *вісім* запакованих 64-розрядних цілих числа, таким чином виконуються вісім окремі 64-розрядні додавання в одній команді.

На рис. 5 показано графічне подання інструкцій додавання векторів і показано як розмір регістра, так і обсяг роботи, виконаної при подвоєнні з кожним поколінням набору інструкцій, заданих запакованими цілими числами від a0 до a7.

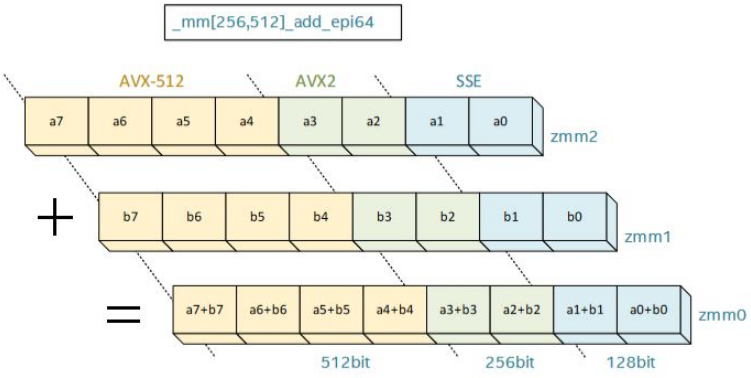


Рисунок 5 – Графічне подання команди додавання векторів

7.1. Запаковані типи даних

У наборі комаед AVX-512 кожен внутрішній суфікс використовується для вказівки того, як обробляються операнди, приймаючи однакові правила іменування інструкцій, як і його попередники. Суфікс `pi` вказує на запаковані цілі операнди, суфікс `pu` вказує на запаковані беззнакові цілі числа, суфікс `pd` вказує на запаковані операнди з плаваючою крапкою подвійної точності, а суфікс `ps` вказує на запаковані операнди з плаваючою крапкою одинарної точності.

Тип даних Сі для внутрішніх операндів оголошується як __m512i, що вказує на запаковані цілі числа, або __m512d, що вказує на запаковані числа з плаваючою крапкою подвійної точності, або __m512, що вказує на запаковані числа з плаваючою крапкою одинарної точності. Звідси випливає, що внутрішні елементи AVX-512 із суфіксом `ps` матимуть операнди з типом даних __m512. Щоб детально зрозуміти цю концепцію, розглянемо приклад додавання векторів, наведений у наступній таблиці.

Внутрішній суфікс	Втутрішня форма Сі команд	Тип запакованих даних
_epi64	__m512i _mm512_add_epi64 (__m512i a, __m512i b)	8 x 64-bit ціле
_epi32	__m512i _mm512_add_epi32 (__m512i a, __m512i b)	16 x 32-bit ціле
_epi16	__m512i _mm512_add_epi16 (__m512i a, __m512i b)	32 x 16-bit ціле
_epi8	__m512i _mm512_add_epi8 (__m512i a, __m512i b)	64 x 8-bit ціле
_pd	__m512d _mm512_add_pd (__m512d a, __m512d b)	8 x 64-bit подвійна точність плаваюча крапка
_ps	__m512 _mm512_add_ps (__m512 a, __m512 b)	16 x 32-bit одинарна точність плаваюча крапка

Усі внутрішні елементи є варіантами 512-бітового векторного додавання, при цьому кожен внутрішній суфікс вказує на те, як операнди оброблюються під час додавання. Наприклад, суфікс epi64 вказує, що внутрішній `_mm512_add_epi64` оброблятиме операнди *a* і *b* як запаковані 64-розрядні цілі числа, з вісьмома 64-розрядними цілими числами, запакованими в 512-розрядний регістр. Так само суфікс epi32 вказує на те, що внутрішній `_mm512_add_epi32` розглядатиме операнди *a* та b як запаковані 32-розрядні цілі числа, з шістнадцятьма 32-розрядними цілими числами, запакованими в один 512-бітовий регістр.

7.2. Регістр mask і масковані операції

AVX-512 надає масковані операції, щоб надати розробникам набагато точніший контроль над поведінкою SIMD інструкції, рис.6. Замасковані інструкції та маскові регістри дозволяють розробникам вказувати, які запаковані значеннями слід використовувати або ігнорувати для даної команди, де кожна позиція біта в регістрі маски відповідає запакованому значенню.

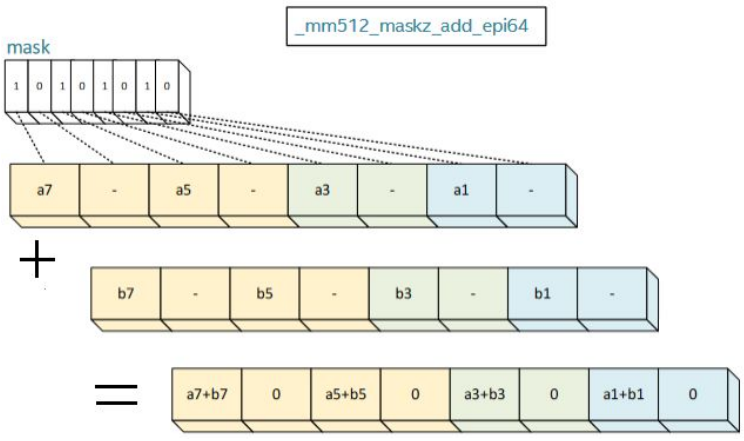


Рисунок 6 – Додавання замаскованих векторів, регістр `mask`

Результат замаскованої 64-розрядної запакованої цілочислової арифметичної операції контролюється 8-розрядним регістром маски. Кожна позиція біта у 8-бітовій масці відповідає одному з восьми запакованих 64-бітових значень в одному 512-бітовому регістрі (див. рис. 5). Так само 32-, 16-, і 8-бітова цілочислова арифметика контролюється 16-, 32- і 64-бітовими значеннями маски відповідно.

Коли біти встановлені в регістрі маски, арифметична операція над відповідними запакованими значеннями виконується при завершенні команди. Однак, коли біти очищаються, замість результату арифметичної операції повертається інше значення. У цьому випадку повернене значення може бути нульовим або прочитаним із регістра джерела, залежно від того, який варіант команди з *maskz* чи *mask* префіксом використовується. Щоб детально зрозуміти цю концепцію, розглянемо приклад додавання маскованих 64-бітових векторів.

Набір команд	Сі внутрішня форма команд
SSE 4.2	__m128i _mm_add_epi64 (__m128i a, __m128i b)
AVX2	__m256i _mm256_add_epi64 (__m256i a, __m256i b)
AVX-512	__m512i _mm512_add_epi64 (__m512i a, __m512i b)
AVX-512	__m512i _mm512_mask_add_epi64 (__m512i src, __mmask8 k, __m512i a, __m512i b)
AVX-512	__m512i _mm512_maskz_add_epi64 (__mmask8 k, __m512i a, __m512i b)

У таблиці показано команди AVX-512 з двома новими варіантами, які не мають еквіваленти в попередніх поколіннях набору інструкцій:

• `_mm512_mask_add_epi64`: регістр маски `k` керує генерацією запакованого значення, що повертається. Для кожного запакованого 64-розрядного цілого числа, він вказує, чи значення має бути прочитано з регістра `src` чи має бути результатом запакованого 64-бітового додавання *a* і *b;*

• `_mm512_maskz_add_epi64:` аналогічно, тут також регістр маски `k` вказує, що повернуте значення має бути нульовим або результатом запакованого додавання *a* і *b*.

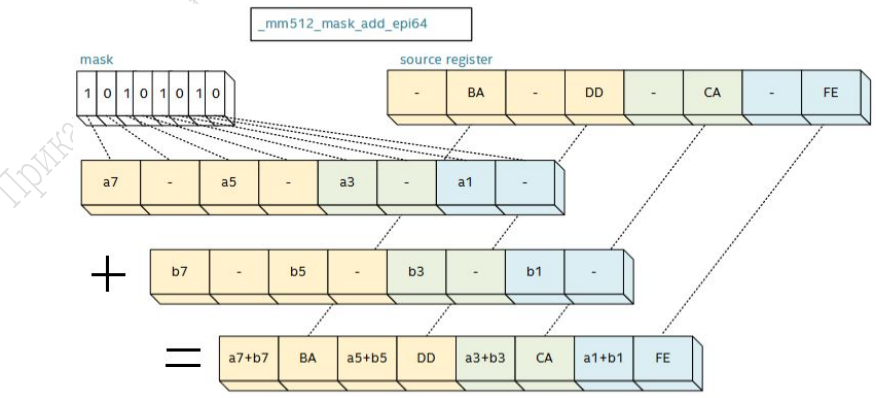


Рисунок 7 – Додавання замаскованих векторів, регістр `mask` і регістр джерело

7.3. Генерація масок

AVX-512 надає ряд нових команд для створення масок із запакованих значень. Внутрішня частина цих інструкцій має спільний суфікс _mask і повертає значення від 8 до 64 бітів залежно від варіанту інструкції.

Набір команд	Сі внутрішня форма команд
AVX-512	__mmask8 _mm512_movepi64_mask (__m512i a)
AVX-512	__mmask8 _mm512_cmp_epu64_mask (__m512i a, __m512i b, _MM_CMPINT_ENUM imm8)

Особливості команд генерації масок:

• `_mm512_movepi64_mask`: встановлює кожний біт повернутої 8-бітової маски на основі старшого біта відповідного 8-запакованого 64-розрядне цілого числа в 512-розрядному регістрі *a*.

• `_mm512_cmp_epu64_mask`: порівнює запаковані 64-розрядні цілі числа без знаку в *a* і *b* на основі операнда порівняння, вказаного в в `imm8`, і зберігає результати у 8-розрядному регістрі маски `mask`.

На рис. 4 4 показано графічне подання команди `_mm512_cmp_epu64_mask`. Вона генерує маску на основі порівняння двох запакованих 64-розрядних значень *a* і *b*. Типом порівняння керує безпосередній оператор `imm8` із діапазоном можливих значень, як показано таблиці: на рис. 8.

Набір команд AVX-512 має також ряд команд для роботи з масками, усі з яких мають префікс `_k`. Приклади внутрішніх маніпуляцій з 64-розрядною маскою: `_knot_mask64` (не), `_kand_mask64` (і), `_kor_mask64` (або), `_kxor_mask64` (xor) `_kshiftri_mask64` (зсув вправо) `and_kshiftli_mask64` (зсув вліво).

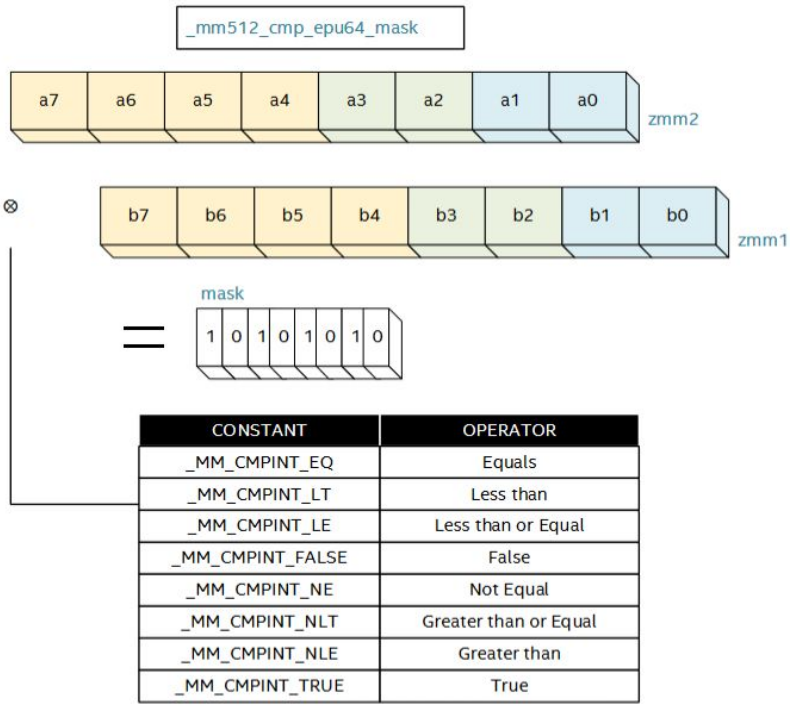


Рисунок 8 – Генерація маски на основі порівняння двох регістрів

Контрольні запитання.

1. Розширення AVX.
2. Історія розвитку x86-AVX.
3. Середовище виконання x86-AVX.
4. Набір регістрів x86-AVX.
5. Типи даних x86-AVX.
6. Синтаксис команд x86-AVX.
7. Огляд команд x86-AVX.
8. Команди широкомовної трансляції.
9. Команди змішування.
10. Команди переставлення.
11. Команди видобування і вставлення.
12. Команди маскованого переміщення.
13. Команди із змінним бітовим зсувом.
14. Команди збирання.
15. Команди з розширеними функціями.
16. Команди половинної точності з плаваючою крапкою.
17. Команди злитого множення-додавання (FMA).
18. Нові команд для регістрів загального призначення.
19. Команди AVX-512.
20. Запаковані типи даних AVX-512.
21. Регістр mask і масковані операції.
22. Генерація масок.

СПИСОК ЛІТЕРАТУРИ

Основний

1. Зілінський Ю. В., Перекрест А. Л., Юдіна А. Л. Системне програмування. Програмування на асемблері: навч. Посібник. Кременчук: Кременчуцький національний університет імені Михайла Остроградського, 2023. 258 с.
2. Системне програмування. Програмування на асемблері: комп'ютерний практикум [Електронний ресурс]: навч. посіб. для студ. освітньої програми «Комп'ютерні системи та мережі» спеціальності 123 «Комп'ютерна інженерія» / КПІ ім. Ігоря Сікорського; уклад. Порєв В.М. – Електронні текстові дані (1 файл: 3,2 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 146 с. – Назва з екрана.
3. Методичні вказівки для виконання лабораторних робіт з дисципліни «Системне програмування» для студентів денної та заочної форми навчання розроблені у відповідності з навчальним планом спеціальності 123 «Комп’ютерна інженерія» / Уклад. Паламар А.М., Паламар М.І. – Тернопіль: ТНТУ, 2020. – 70 с.
4. Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмування" для студентів спеціальності 125 "Кібербезпека" всіх форм навчання – Частина 1 / Укл.: В.В. Шкарупило. – Київ: НУБіП, 2022. – 42 с.
5. Основи комп’ютерної техніки та програмування мікропроцесорів: навч. посіб. / Д.О. Гололобов. – К. : Редакційно-видавничий центр Державного університету телекомунікацій, 2019. – 58с. : іл
6. Рисований О. М. Системне програмування: підручник для студентів напрямку «Комп’ютерна інженерія» вищих навчальних закладів у 2-х томах. Том 1. Видання четверте: виправлено та доповнено – Харків: «Слово», 2015. – 576 с.
7. Рисований О. М. Системне програмування: підручник для студентів напрямку «Комп’ютерна інженерія» вищих навчальних закладів у 2-х томах. Том 2. Видання четверте: виправлено та доповнено – Харків: «Слово», 2015. – 378 с.

Додатковий

8. Randall Hyde. The Art Of 64-bit Assembly. No Starch Press, 2021. – 1032 p.
9. Sherwyn Allibang. Assembly Language: Simple, Short, and Straightforward Way of Learning Assembly Programming. Independently published, 2020. – 160 p.
10. Irvine Kip. Assembly Language for x86 Processors 8th ed. Pearson, 2019. – 880 p.
11. Jo Van Hoey. Beginning x64 Assembly Programming, From Novice to AVX Professional. Apress, 2019. – 432 p.
12. Daniel Kusswurm. Modern X86 Assembly Language Programming: Covers x86 64-bit, AVX, AVX2 and AVX-512 1st ed. Apress, 2018. – 625 p.
13. Ray Seyfarth. Introduction to 64 Bit Windows Assembly Language Programming 8th ed. CreateSpace Independent Publishing Platform, 2017. – 288 p.
14. Alexey Lyashko. Follow Mastering Assembly Programming: From instruction set to kernel module with Intel processor 1st ed. Packt Publishing, 2017. – 290 p.
15. Jeff Duntemann. Follow Assembly Language Step-by-Step 3rd ed. Wiley, 2011. – 656 p.

Інформаційні ресурси

16. GDB: The GNU Project Debugger [електронний ресурс]: – Режим доступу: https://www.gnu.org/software/gdb/ – назва з екрану
17. DDD (data display debugger) [електронний ресурс]: – Режим доступу: https://www.gnu.org/software/ddd/ – назва з екрану
18. KGDB A Graphical Debugger Interface [електронний ресурс]: – Режим доступу: https://www.kdbg.org/ – назва з екрану
19. SASM – data display debugger [електронний ресурс]: – Режим доступу: https://www.gnu.org/software/ddd/manual/ – назва з екрану.
20. NASM [Електронний ресурс]. – Режим доступу: https://www.nasm.us. – Заголовок з екрану